IBM System Blue Gene Solution: Blue Gene/P Application Development

Understand the Blue Gene/P programming environment

Learn how to run and debug MPI programs

Learn about Bridge and Real-time APIs

Carlos Sosa
Brant Knudson

Redbooks

ibm.com/redbooks

International Technical Support Organization

IBM System Blue Gene Solution: Blue Gene/P Application Development

September 2009

SG24-7287-03

Note: Before using this information and the product it supports, read the information in “Notices” on page ix.

Fourth Edition (September 2009)

This edition applies to Version 1, Release 4, Modification 0 of IBM System Blue Gene/P Solution (product number 5733-BGP).

Contents

Notices

This information was developed for products and services offered in the U.S.A.

IBM might not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service might be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right might be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM might have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement might not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM might make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM might use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You might copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

AIX®
Blue Gene/L™
Blue Gene/P™
Blue Gene®
DB2 Universal Database™
DB2®
eServer™
General Parallel File System™
GPFS™
IBM®
LoadLeveler®
POWER™
POWER4™
POWER5™
POWER6™
PowerPC®
Redbooks®
Redbooks (logo) ®
System p®
Tivoli®

The following terms are trademarks of other companies:

Snapshot, and the NetApp logo are trademarks or registered trademarks of NetApp, Inc. in the U.S. and other countries.

SUSE, the Novell logo, and the N logo are registered trademarks of Novell, Inc. in the United States and other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Intel logo, Intel Inside logo, and Intel Centrino logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names might be trademarks or service marks of others.

Preface

This IBM® Redbooks® publication is one in a series of IBM books written specifically for the IBM System Blue Gene/P Solution. The Blue Gene/P system is the second generation of a massively parallel supercomputer from IBM in the IBM System Blue Gene Solution series. In this book, we provide an overview of the application development environment for the Blue Gene/P system. We intend to help programmers understand the requirements to develop applications on this high-performance massively parallel supercomputer.

In this book, we explain instances where the Blue Gene/P system is unique in its programming environment. We also attempt to look at the differences between the IBM System Blue Gene/L Solution and the Blue Gene/P Solution. In this book, we do not delve into great depth about the technologies that are commonly used in the supercomputing industry, such as Message Passing Interface (MPI) and Open Multi-Processing (OpenMP), nor do we try to teach parallel programming. References are provided in those instances for you to find more information if necessary.

Prior to reading this book, you must have a strong background in high-performance computing (HPC) programming. The high-level programming languages that we use throughout this book are C/C++ and Fortran95. Previous experience using the Blue Gene/L system can help you better understand some concepts in this book that we do not extensively discuss. However, several IBM Redbooks publications about the Blue Gene/L system are available for you to obtain general information about the Blue Gene/L system. We recommend that you refer to “IBM Redbooks” on page 371 for a list of those publications.

The team who wrote this book

This book was produced in collaboration with the IBM Blue Gene developers at IBM Rochester, Minnesota, and IBM Blue Gene® developers at the IBM T. J. Watson Center in Yorktown Heights, N.Y. The information presented in this book is direct documentation of many of the Blue Gene/P hardware and software features. This information was published by the International Technical Support Organization, Rochester, Minnesota.

Carlos Sosa is a Senior Technical Staff Member in the Blue Gene Development Group of IBM, where he has been the team lead of the Chemistry and Life Sciences high-performance effort since 2006. For the past 18 years, he focused on scientific applications with emphasis in Life Sciences, parallel programming, benchmarking, and performance tuning. He received a Ph.D. degree in physical chemistry from Wayne State University and completed his post-doctoral work at the Pacific Northwest National Laboratory. His areas of interest are future IBM POWER™ architectures, Blue Gene, Cell Broadband, and cellular molecular biology.

Brant Knudson is a Staff Software Engineer in the Advanced Systems SW Development group of IBM in Rochester, Minnesota, where he has been a programmer on the Control System team since 2003. Prior to working on Blue Gene, he worked on IBM Tivoli® Directory Server.

We thank the following people and their teams for their contributions to this book:

- Tom Liebsch for being the lead source for hardware information
- Harold Rodakowski for software information
- Thomas M. Gooding for kernel information
- Michael Blocksome for parallel paradigms
- Michael T. Nelson and Lynn Boger for their help with the compiler
- Thomas A. Budnik for his assistance with APIs
- Paul Allen for his extensive contributions

We also thank the following people for their contributions to this project:

Gary Lakner
Gary Mullen-Schultz
ITSO, Rochester, MN

Dino Quintero
ITSO, Poughkeepsie, NY

Paul Allen
John Attinella
Mike Blocksome
Lynn Boger
Thomas A. Budnik
Ahmad Faraj
Thomas M. Gooding
Nicholas Goracke
Todd Inglet
Tom Liebsch
Mark Megerian
Sam Miller
Mike Mundy
Tom Musta
Mike Nelson
Jeff Parker
Ruth J. Poole
Joseph Ratterman
Richard Shok
Brian Smith
IBM Rochester

Philip Heidelberg
Sameer Kumar
Martin Ohmacht
James C. Sexton
Robert E. Walkup
Robert Wisniewski
IBM Watson Center

Mark Mendell
IBM Toronto

Ananthanaraya Sugavanam
Enci Zhong
IBM Poughkeepsie

Kirk Jordan
IBM Waltham

Jerrold Heyman
IBM Raleigh

Subba R. Bodda
IBM India

Become a published author

Join us for a two- to six-week residency program! Help write a book dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbooks@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Summary of changes

This section describes the technical changes made in this edition of the book and in previous editions. This edition might also include minor corrections and editorial changes that are not identified.

September 2009, Fourth Edition

Summary of Changes
for SG24-7287-03
for *IBM Blue Gene/P Application Development*
as created or updated on September 2009

This revision reflects the addition, deletion, or modification of new and changed information described in the following sections.

New information

- The compute and I/O node daemon creates a /jobs directory, described in 3.3.1, “Control and I/O daemon” on page 33.
- The Compute Node Kernel now supports multiple application threads per core, described in Chapter 4, “Execution process modes” on page 37.
- Support for GOMP, described in 8.5, “Support for pthreads and OpenMP” on page 100.
- Package configuration can be forwarded to the Blue Gene/P compute nodes, described in 8.10, “Configuring Blue Gene/P builds” on page 105.
- mpirun displays APPLICATION RAS events with -verbose 2 or higher, described in Chapter 11, “mpirun” on page 177.
- The Real-time APIs now support RAS events, described in Chapter 14, “Real-time Notification APIs” on page 251.
- Environment variable for specifying the cores that generate binary core files, described in Appendix D, “Environment variables” on page 339.
- Two new environment variables have been added that affect latency on broadcast and allreduce, DCMF_SAFEBCAST and DCMF_SAFEALLREDUCE, see Appendix D, “Environment variables” on page 339 for more information.

Modified information

- The prefix of several constants were changed from DCMF_ to MPIO_ in 7.3.2, “Configuring MPI algorithms at run time” on page 77.
- The Python language version is now 2.6.
- The IBM XL compilers support the -qsigtrap compiler option, described in Chapter 8, “Developing applications with IBM XL compilers” on page 97.
- Users can debug HTC applications using submit, described in 12.3, “Running a job using submit” on page 202.

December 2008, Third Edition

Summary of Changes
for SG24-7287-02
for *IBM Blue Gene/P Application Development*
as created or updated on December 2008

This revision reflects the addition, deletion, or modification of new and changed information described in the following sections.

New information

- The IBM Blue Gene/P™ hardware now supports compute nodes with 4 GB of DDR memory.
- STAR-MPI API in “Buffer alignment sensitivity” on page 73.
- Per-site Message Passing Interface (MPI) configuration API in 7.3.2, “Configuring MPI algorithms at run time” on page 77.
- Blue Gene/P MPI environment variables in “MPI functions” on page 80.
- Scalable Debug API in 9.2.10, “Scalable Debug API” on page 161.
- **mpikill** command-line utility in “mpikill” on page 180.
- **mpirun** -start_tool and -tool_args arguments in “Invoking mpirun” on page 183.
- Tool-launching interface in “Tool-launching interface” on page 188.
- In “Examples” on page 191, failing **mpirun** prints out reliability, availability, and serviceability (RAS) events.
- job_started() function in “mpirun APIs” on page 199.
- Immediate High-Throughput Computing (HTC) partition user list modification in “Altering the HTC partition user list” on page 206.
- Partition options modifications in section “Partition-related APIs” on page 215.
- RM_PartitionBootOptions specification in “Field specifications for the rm_get_data() and rm_set_data() APIs” on page 229.
- Server-side filtering, notification of HTC events, and new job callbacks in Chapter 14, “Real-time Notification APIs” on page 251.

Modified information

- In “mpiexec” on page 179, removing some limitations that are no longer present
- Use of Compute Node Kernel’s thread stack protection mechanism in more situations as described in “Memory protection” on page 47
- Changes to Dynamic Partition Allocator APIs, described in Chapter 15, “Dynamic Partition Allocator APIs” on page 295, in non-backwards compatible ways
- HTC partitions support for group permissions, as described in Appendix G, “htcpartition” on page 359

September 2008, Second Edition

Summary of Changes
for SG24-7287-01
for *IBM Blue Gene/P Application Development*
as created or updated on September 2008

This revision reflects the addition, deletion, or modification of new and changed information described in the following sections.

New information

- High Throughput Computing in Chapter 12, “High-Throughput Computing (HTC) paradigm” on page 201
- Documentation on htcpartition in Appendix G, “htcpartition” on page 359

Modified information

- The book was reorganized to include Part 4, “Job scheduler interfaces” on page 207. This section contains the Blue Gene/P APIs. Updates to the API chapters for HTC are included.
- Appendix F, “Mapping” on page 355 is updated to reflect the predefined mapping for mpirun.

Part 1

Blue Gene/P: System and environment overview

IBM Blue Gene/P is the next generation of massively parallel systems that IBM produced. It follows in the tradition established by the IBM Blue Gene/L™ Solution in challenging our thinking to take advantage of this innovative architecture. This next generation of supercomputers follows the winning formula provided as part of the Blue Gene/L Solution, that is, orders of magnitude in size and substantially more efficient in power consumption.

In this part, we present an overview of the two main topics of this book: hardware and software environment. This part includes the following chapters:

- Chapter 1, “Hardware overview” on page 3
- Chapter 2, “Software overview” on page 15

1

Hardware overview

In this chapter, we provide a brief overview of hardware. This chapter is intended for programmers who are interested in learning about the Blue Gene/P system. In this chapter, we provide an overview for programmers who are already familiar with the Blue Gene/L system and who want to understand the differences between the Blue Gene/L and Blue Gene/P systems.

It is important to understand where the Blue Gene/P system fits within the multiple systems that are currently available in the market. To gain a historical perspective and a perspective from an applications point-of-view, we recommend that you read the first chapter of the book *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686. Although this book is written for the Blue Gene/L system, these concepts apply to the Blue Gene/P system.

In this chapter, we describe the Blue Gene/P architecture. We also provide an overview of the machine with a brief description of some of the components. Specifically, we address the following topics:

- System architecture overview
- Differences between Blue Gene/L and Blue Gene/P hardware
- Microprocessor
- Compute nodes
- I/O nodes
- Networks
- Blue Gene/P programs
- Blue Gene/P specifications
- Host system
- Host system software

1.1 System architecture overview

The IBM System Blue Gene Solution is a revolutionary and important milestone for IBM in the high-performance computing arena. The Blue Gene/L system was the fastest supercomputer from the fall of 2004 until the fall of 2007 as noted by the TOP500 organization.[1] Now IBM introduces the Blue Gene/P system as the next generation of massively parallel supercomputers, based on the same successful architecture that is in the Blue Gene/L system.

The Blue Gene/P system includes the following key features and attributes, among others:

- Dense number of cores per rack: 4096 cores per rack
- IBM PowerPC®, Book E compliant, 32-bit microprocessor, 850 MHz
- Double-precision, dual pipe floating-point acceleration on each core
- 24-inch/42U server rack air cooled
- Low power per FLOP ratio on IBM Blue Gene/P compute application-specific integrated circuit (ASIC), 1.8 watts per GFlop/sec. per SOC
- Includes memory controllers, caches, network controllers, and high-speed input/output (I/O)
- Linux® kernel running on I/O Nodes
- Message Passing Interface (MPI)[2] support between nodes via MPI library support
- Open Multi-Processing (OpenMP)[3] application programming interface (API)
- Scalable control system based on external service node and front end node
- Standard IBM XL family of compilers support with XL C/C++, XLF, and GNU Compiler Collection[5]
- Software support for IBM LoadLeveler®,[6] IBM General Parallel File System™ (GPFS™),[7] and Engineering and Scientific Subroutine Library (ESSL)[8]

Figure 1-1 on page 5 illustrates the Blue Gene/P system architecture. It provides an overview of the multiple system components, from the microprocessor to the full system.

The system contains the following components:

Chip The Blue Gene/P base component is a quad-core chip (also referred throughout this book as a *node*). The frequency of a single core is 850 MHz.

Compute card One chip is soldered to a small processor card, one per card, together with memory (DRAM), to create a compute card (one node). The amount of DRAM per card is 2 GB or 4 GB.

Node card The compute cards are plugged into a node card. There are two rows of sixteen compute cards on the card (planar). From zero to two I/O nodes per compute node card can be added to the node card.

Rack A rack holds a total of 32 node cards.

System A full petaFLOP system consists of 72 racks.

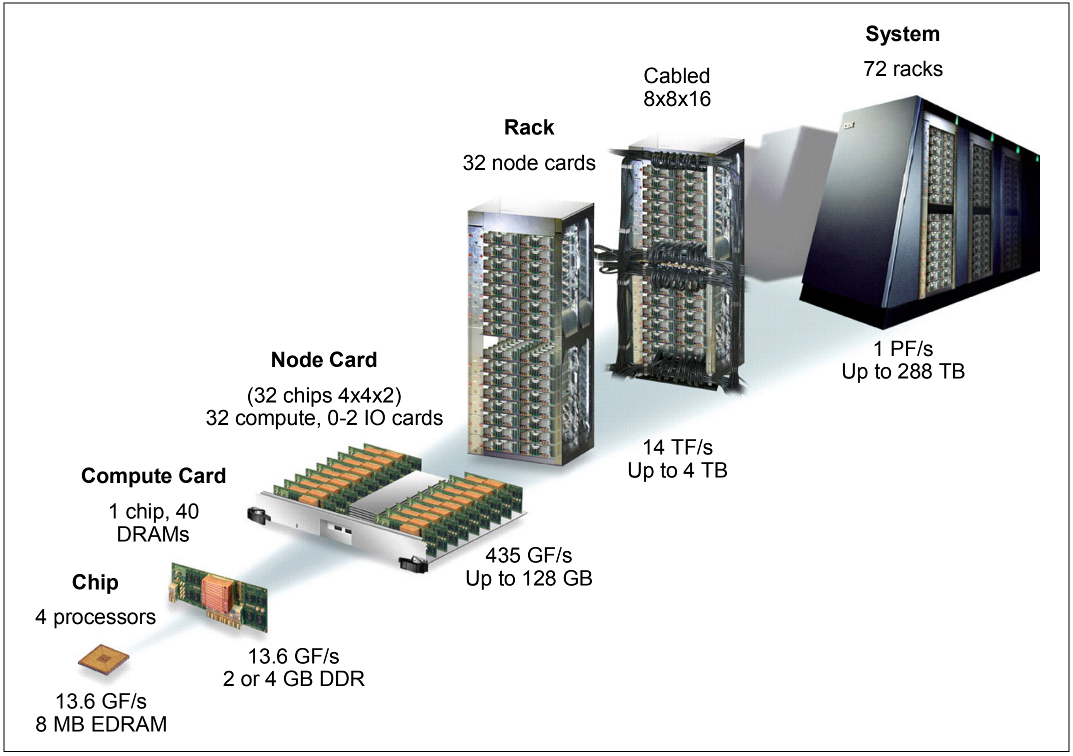


Figure 1-1 Blue Gene/P system overview from the microprocessor to the full system

1.1.1 System buildup

The number of cores in a system can be computed using the following equation:

```
Number of cores = (number of racks) x (number of node cards per rack) x (number of
compute cards per node card) x (number of cores per compute card)
```

This equation corresponds to cores and memory. However, I/O is carried out through the I/O node that is connected externally via a 10 gigabit Ethernet network. This network corresponds to the functional network. I/O Nodes are not considered in the previous equation.

Finally, the compute and I/O Nodes are connected externally (to the outside world) through the following peripherals:

- One service node
- One or more front end nodes
- Global file system

1.1.2 Compute and I/O nodes

Nodes are made of one quad-core CPU with 2 GB or 4 GB of memory. These nodes do not have a local file system. Therefore, they must route I/O operations to an external device. To reach this external device (outside the environment), a compute node sends data to an I/O node, which in turn, carries out the I/O requests.

The hardware for both types of nodes is virtually identical. The nodes differ only in the way they are used, for example, extra RAM might be on the I/O Nodes, and the physical connectors thus are different. A compute node runs a light, UNIX®-like proprietary kernel, referred to as the *Compute Node Kernel (CNK)*. The CNK ships all network-bound requests to the I/O node.

The I/O node is connected to the external device through an Ethernet port to the 10 gigabit functional network and can perform file I/O operations.In the next section, we provide an overview of the Blue Gene environment, including all the components that fully populate the system.

1.1.3 Blue Gene/P environment

The Blue Gene/P environment consists of all the components that form part of the full system. Figure 1-2 illustrates the multiple components that form the Blue Gene/P environment. The Blue Gene/P system consists of the following key components:

Service node This node provides control of the Blue Gene/P system.

Front end node This node provides access to the users to submit, compile, and build applications.

Compute node This node runs applications. Users cannot log on to this node.

I/O node This node provides access to external devices, and all I/O requests are routed through this node.

Functional network This network is used by all components of the Blue Gene/P system except the compute node.

Control network This network is the service network for specific system control functions between the service node and the I/O node.

In the remainder of this chapter, we describe these key components.

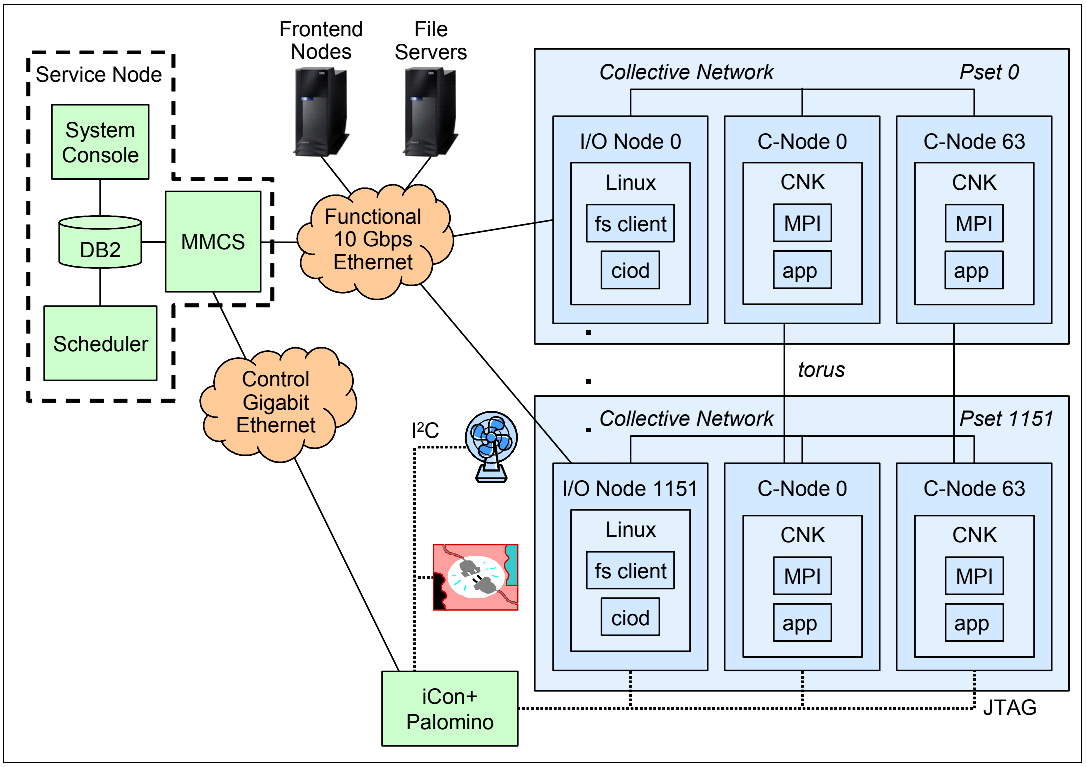


Figure 1-2 Blue Gene/P environment

1.2 Differences between Blue Gene/L and Blue Gene/P hardware

The Blue Gene/P solution is a highly scalable multi-node supercomputer. Table 1-1 on page 8 shows key differences between the Blue Gene/L and Blue Gene/P systems. Each node consists of a single ASIC and forty 512 MB SDRAM-DDR2 memory chips. The nodes are interconnected through six networks, one of which connects the nearest neighbors into a three-dimensional (3D) torus or mesh. A system with 72 racks has a (x, y, z) 72 x 32 x 32 3D torus. The ASIC that is powering the nodes is in IBM CU-08 (CMOS9SF) system-on-a-chip technology and incorporates all of the compute and communication functionality needed by the core Blue Gene/P system. It contains 8 MiB of high-bandwidth embedded DRAM that can be accessed by the four cores in approximately 20 cycles for most L1 cache misses.

MiB: 1 MiB = 2^{20} bytes = 1,048,576 bytes = 1,024 kibibytes (a contraction of kilo binary byte)

The scalable unit of Blue Gene/P packaging consists of 512 compute nodes on a doubled-sided board, called a *midplane*, with dimensions of approximately 20 inches x 25 inches x 34 inches.

Note: A midplane is the smallest unit that supports the full 3D torus.

Each node operates at Voltage Drain Drain (VDD) = 1.1v, 1.2v, or 1.3v, Temp_junction <70C, and a frequency of 850 MHz. Using an IBM PowerPC 450 processor and a single-instruction, multiple-data (SIMD), double-precision floating-point multiply add unit (double floating-point multiply add (FMA)), it can deliver four floating-point operations per cycle, or a theoretical maximum of 7.12 teraFLOPS at peak performance for a single midplane. Two midplanes are contained within a single cabinet.

A midplane set of processing nodes, from a minimum of 16 to a maximum of 128, can be attached to a dedicated quad-processor I/O node for handling I/O communications to and from the compute nodes. The I/O node is assembled using the same ASIC as a compute node. Each compute node has a separate lightweight kernel, the CNK, which is designed for high-performance scientific and engineering code. With help from the I/O node kernel, the CNK provides Linux-like functionality to user applications. The I/O nodes run an embedded Linux operating system that is extended to contain additional system software functionality to handle communication with the external world and other services.

The I/O Nodes of the Blue Gene/P system are connected to an external 10 gigabit Ethernet switch, as previously mentioned, which provides I/O connectivity to file servers of a cluster-wide file system as illustrated in Figure 1-2 on page 6. The 10 gigabit Ethernet switch connects the Blue Gene/P system to the front end node and other computing resources. The front end node supports interactive logons, compiling, and overall system management.

Table 1-1 compares selected features between the Blue Gene/L and Blue Gene/P systems.

Table 1-1 Feature comparison between the Blue Gene/L and Blue Gene/P systems

Feature	Blue Gene/L	Blue Gene/P
Node		
Cores per node	2	4
Core clock speed	700 MHz	850 MHz
Cache coherency	Software managed	SMP
Private L1 cache	32 KB per core	32 KB per core
Private L2 cache	14 stream prefetching	14 stream prefetching
Shared L3 cache	4 MB	8 MB
Physical memory per node	512 MB-1 GB	2 GB or 4 GB
Main memory bandwidth	5.6 GBps	13.6 GBps
Peak performance	5.6 GFLOPS per node	13.6 GLOPS per node
Network topologies		
Torus		
Bandwidth	2.1 GBps	5.1 GBps
Hardware latency (nearest neighbor)	200 ns (32-byte packet) and 1.6 µs (256-byte packet)	100 ns (32-byte packet) and 800 ns (256-byte packet)
Tree		
Bandwidth	700 MBps	1.7 GBps
Hardware latency (round trip worst case)	5.0 µs	3.0 µs
Full system		
Peak performance	410 TFLOPS (72 racks)	1 PFLOPS (72 racks)
Power	1.7 MW (72 racks)	2.1 MW (72 racks)

Appendix A, “Blue Gene/P hardware-naming conventions” on page 325 provides an overview of how the Blue Gene/P hardware locations are assigned. Names are used consistently throughout both the hardware and software chapters. Understanding the naming convention is particularly useful when running applications on the Blue Gene/P system.

1.3 Microprocessor

The microprocessor is a PowerPC 450, Book E compliant, 32-bit microprocessor with a clock speed of 850 MHz. The PowerPC 450 microprocessor, with double-precision floating-point multiply add unit (double FMA), can deliver four floating-point operations per cycle with 3.4 GLOPS per core.

1.4 Compute nodes

The compute node contains four PowerPC 450 processors with 2 GB or 4 GB of shared RAM and run a lightweight kernel to execute user-mode applications only. Typically all four cores are used for computation either in dual mode, virtual node mode, or symmetrical multiprocessing. (Chapter 4, “Execution process modes” on page 371 covers these different modes.) Data is moved to and from the I/O nodes over the global collective network. Figure 1-3 illustrates the components of a compute node.

Compute nodes consist of the following components:

- Four 850 MHz PowerPC 450 cores
- Two GB or 4 GB RAM per node
- Six connections to the torus network at 3.4 Gbps per link
- Three connections to the global collective network at 6.8 Gbps per link
- Four connections to the global interrupt network
- One connection to the control network (JTAG)

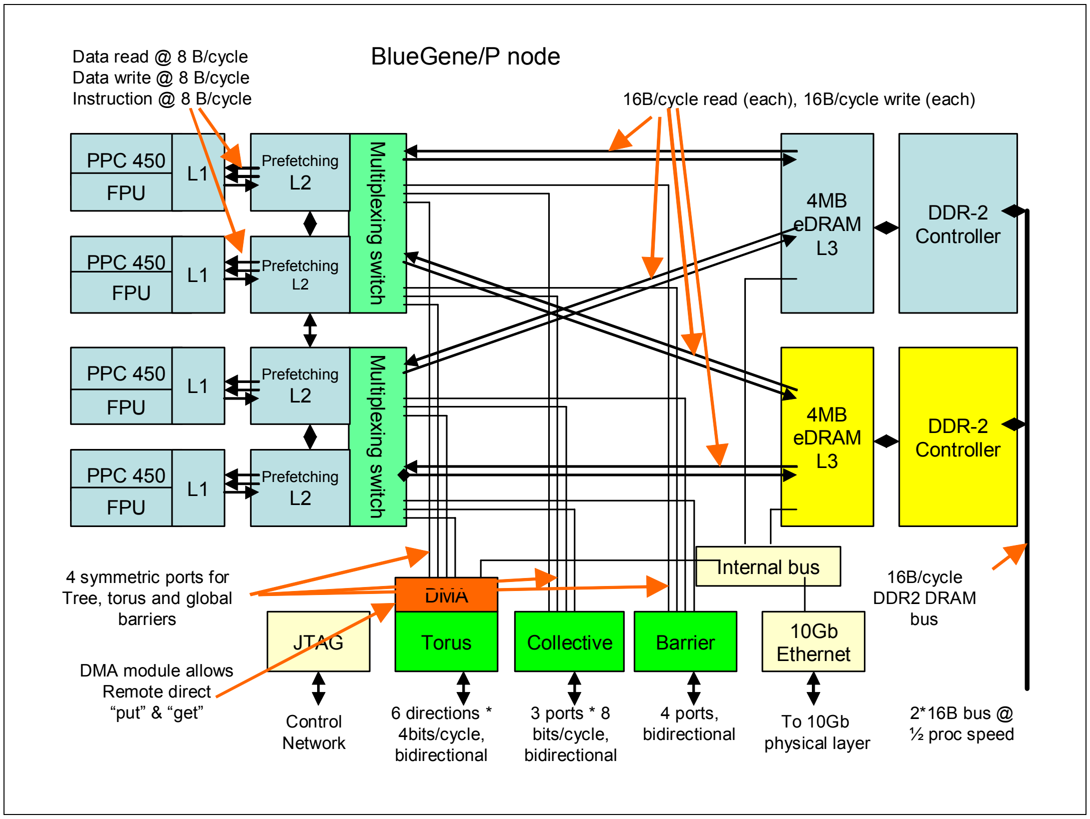


Figure 1-3 Blue Gene/P ASIC

1.5 I/O nodes

I/O nodes run an embedded Linux kernel with minimal packages required to support a Network File System (NFS) client and Ethernet network connections. They act as a gateway for the compute nodes in their respective rack to the external world (see Figure 1-4). The I/O Nodes present a subset of standard Linux operating interfaces to the user. The 10 gigabit Ethernet interface of the I/O nodes is connected to the core Ethernet switch.

The node cards have the following components among others:

- Four 850 MHz PowerPC 450 cores
- Two GB or 4 GB DDR2 SDRAM
- One 10 gigabit Ethernet adapter connected to the 10 gigabit Ethernet network
- Three connections to the global collective network at 6.8 Gbps per link
- Four connections to the global interrupt network
- One connection to the control network (JTAG)

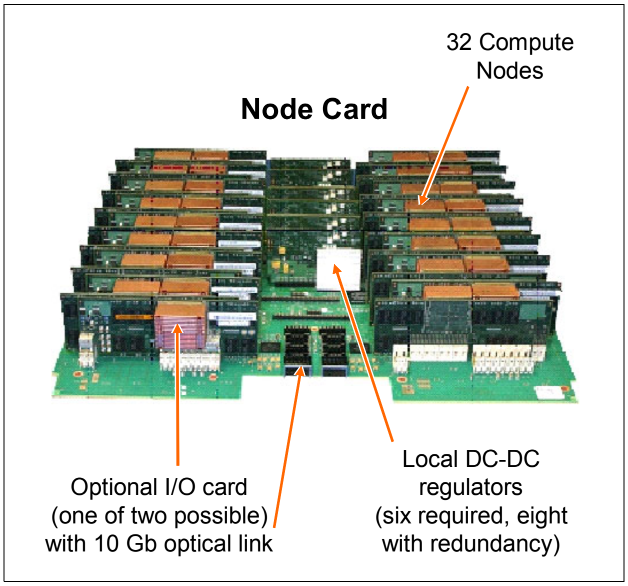

32 Compute Nodes
Node Card
Optional I/O card (one of two possible) with 10 Gb optical link
Local DC-DC regulators (six required, eight with redundancy)

Figure 1-4 Blue Gene/P I/O node card

1.6 Networks

Five networks are used for various tasks on the Blue Gene/P system:

- Three-dimensional torus: point-to-point

  The torus network is used for general-purpose, point-to-point message passing and multicast operations to a selected "class" of nodes. The topology is a three-dimensional torus constructed with point-to-point, serial links between routers embedded within the Blue Gene/P ASICs. Therefore, each ASIC has six nearest-neighbor connections, some of which can traverse relatively long cables. The target hardware bandwidth for each torus link is 425 MBps in each direction of the link for a total of 5.1 GBps bidirectional bandwidth per node. The three-dimensional torus network supports the following features:

  - Interconnection of all compute nodes (73,728 for a 72-rack system)
  - Virtual cut-through hardware routing
  - 3.4 Gbps on all 12 node links (5.1 GBps per node)
  - Communications backbone for computations
  - 1.7/3.8 TBps bisection bandwidth, 67 TBps total bandwidth

- Global collective: global operations

  The global collective network is a high-bandwidth, one-to-all network used for collective communication operations, such as broadcast and reductions, and to move process and application data from the I/O Nodes to the compute nodes. Each Compute and I/O node has three links to the global collective network at 850 MBps per direction for a total of 5.1 GBps bidirectional bandwidth per node. Latency on the global collective network is less than 2 µs from the bottom to top of the collective, with an additional 2 µs latency to broadcast to all. The global collective network supports the following features:

  – One-to-all broadcast functionality
  – Reduction operations functionality
  – 6.8 Gbps of bandwidth per link; latency of network traversal 2 µs
  – 62 TBps total binary network bandwidth
  – Interconnects all compute and I/O Nodes (1088)

- Global interrupt: low latency barriers and interrupts

  The global interrupt network is a separate set of wires based on asynchronous logic, which forms another network that enables fast signaling of global interrupts and barriers (global AND or OR). Round-trip latency to perform a global barrier over this network for a 72 K node partition is approximately 1.3 µs.

- 10 gigabit Ethernet: file I/O and host interface

  The 10 gigabit Ethernet (optical) network consists of all I/O Nodes and discrete nodes that are connected to a standard 10 gigabit Ethernet switch. The compute nodes are not directly connected to this network. All traffic is passed from the compute node over the global collective network to the I/O node and then onto the 10 gigabit Ethernet network.

- Control: boot, monitoring, and diagnostics

  The control network consists of a JTAG interface to a 1 gigabit Ethernet interface with direct access to shared SRAM in every Compute and I/O node. The control network is used for system boot, debug, and monitoring. It enables the service node to provide run-time non-invasive reliability, availability, and serviceability (RAS) support as well as non-invasive access to performance counters.

1.7 Blue Gene/P programs

The Blue Gene/P software for the Blue Gene/P core rack includes the following programs:

- Compute Node Kernel (CNK)

  MPI support for hardware implementation and abstract device interface, control system, and system diagnostics.

- Compute Node services (CNS)

  Provides an environment for execution of user processes. The services that are provided are process creation and management, memory management, process debugging, and RAS management.

- I/O node kernel and services

  Provides file system access and sockets communication to applications executing in the compute node.

- GNU Compiler Collection Toolchain Patches (Blue Gene/P changes to support GNU Compiler Collection).

The system software that is provided with each Blue Gene/P core rack or racks includes the following programs:

- IBM DB2® Universal Database™ Enterprise Server Edition: System administration and management
- Compilers: XL C/C++ Advanced Edition for Linux with OpenMP support and XLF (Fortran) Advanced Edition for Linux

1.8 Blue Gene/P specifications

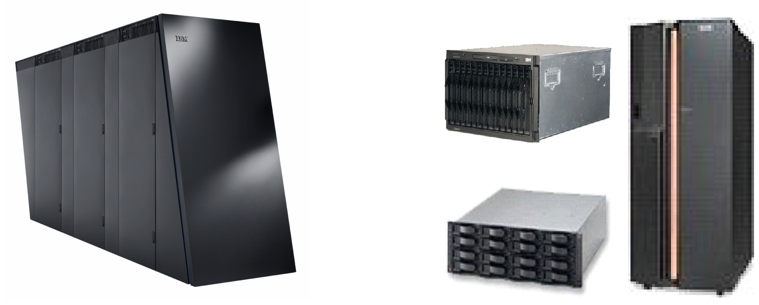
Table 1-2 lists the features of the Blue Gene/P compute nodes and I/O nodes.

Table 1-2 Blue Gene/P node properties

Node properties	
Node processors (compute and I/O)	Quad 450 PowerPC
Processor frequency	850 MHz
Coherency	Symmetrical multiprocessing
L1 Cache (private)	32 KB per core
L2 Cache (private)	14 stream prefetching
L3 Cache size (shared)	8 MB
Main store memory/node	2 GB or 4 GB
Main store memory bandwidth	16 GBps
Peak performance	13.6 GFLOPS (per node)
Torus network	
Bandwidth	6 GBps
Hardware latency (nearest neighbor)	64 ns (32-byte packet), 512 ns (256-byte packet)
Hardware latency (worst case)	3 μs (64 hops)
Global collective network	
Bandwidth	2 GBps
Hardware latency (round-trip worst case)	2.5 μs
System properties (for 73,728 compute nodes)	
Peak performance	1 PFLOPS
Average/peak total power	1.8 MW/2.5 MW (25 kW/34 kW per rack)

1.9 Host system

In addition to the Blue Gene/P core racks, the host system shown in Figure 1-5 is required for a complete Blue Gene/P system. There is generally one host rack for the core Ethernet switch, service node, and front end node. It might also house the Hardware Management Console (HMC) control node, monitor, keyboard, KVM switch, terminal server, and Ethernet modules.



Blue Gene core rack
1024 Compute Nodes/rack
Up to 64 I/O Nodes/rack

Blue Gene program software
Compute Node Kernel

Host system
Service and Front End Nodes, storage system,
Ethernet switch, cabling, SLES10

System software
DB2, XLF/C compilers

Optional HPC software
LoadLeveler, GPFS, ESSL

Figure 1-5 Blue Gene/P rack and host system

1.9.1 Service node

The service node performs many functions for the operation of the Blue Gene/P system, including system boot, machine partitioning, system performance measurements, and system health monitoring. The service node uses IBM DB2 as the data repository for system and state information.

1.9.2 Front end nodes

The front end node provides interfaces for users to log on, compile their applications, and submit their jobs to run from these nodes. They have direct connections to both the Blue Gene/P internal VLAN and the public Ethernet networks.

1.9.3 Storage nodes

The storage nodes provide mass storage for the Blue Gene/P system. We recommend that the storage nodes run IBM GPFS locally to provide a single unified file system name space to the Blue Gene/P system. However the I/O nodes access the GPFS file system over standard NFS mounts.

The storage rack generally contains the terminal server, storage nodes with RAM, gigabit Ethernet adapters connected to the core Ethernet switch, and adapters connected to a hard disk drive (HDD).

1.10 Host system software

The operating system requires installation of SUSE® Linux Enterprise Server 10 (SLES10, 64 bit) on the service node and front end node.

The following software applications for high-performance computing are optionally available for the Blue Gene/P system:

- Cluster System Management V1.5
- File system: GPFS for Linux Server with NFS Client
- Job Scheduler: LoadLeveler for Blue Gene/P
- Engineering and Scientific Subroutine Library
- Application development tools for Blue Gene/P, which include debugging environments, application performance monitoring and tuning tools, and compilers.

2

Software overview

In this chapter, we provide an overview of the software that runs on the Blue Gene/P system. As shown in Chapter 1, “Hardware overview” on page 3, the Blue Gene/P environment consists of compute and I/O nodes. It also has an external set of systems where users can perform system administration and management, partition and job management, application development, and debugging. In this heterogeneous environment, software must be able to interact.

Specifically, we cover the following topics:

- IBM Blue Gene/P software at a glance
- Compute Node Kernel
- Message Passing Interface on Blue Gene/P
- Memory considerations
- Other considerations
- Compilers overview
- I/O node software
- Management software

2.1 Blue Gene/P software at a glance

Blue Gene/P software includes the following key attributes among others:

- Full Linux kernel running on I/O nodes
- Proprietary kernel dedicated for the Compute Nodes
- Message Passing Interface (MPI)[9] support between nodes through MPI library support
- Open Multi-Processing (OpenMP)[10] application programming interface (API)
- Scalable control system based on an external Service Node and Front End Node
- Standard IBM XL family of compilers[11] support with XLC/C++, XLF, and GNU Compiler Collection[12]
- Software support that includes IBM LoadLeveler,[13] IBM GPFS,[14] and Engineering and Scientific Subroutine Library (ESSL)[15]

From a software point of view, the Blue Gene/P system is comprised of the following components:

- Compute nodes
- I/O nodes
- Front end nodes where users compile and submit jobs
- Control management network
- Service node, which provides capabilities to manage jobs running in the racks
- Hardware in the racks

The Front End Node consists of the interactive resources on which users log on to access the Blue Gene/P system. Users edit and compile applications, create job control files, launch jobs on the Blue Gene/P system, post-process output, and perform other interactive activities.

An Ethernet switch is the main communication path for applications that run on the Compute Node to the external devices. This switch provides high-speed connectivity to the file system, which is the main disk storage for the Blue Gene/P system. This switch also gives other resources access to the files on the file system.

A control and management network provides system administrators with a separate command and control path to the Blue Gene/P system. This private network is not available to unprivileged users.

The software for the Blue Gene/P system consists of the following integrated software subsystems:

- System administration and management
- Partition and job management
- Application development and debugging tools
- Compute Node Kernel (CNK) and services
- I/O node kernel and services

The five software subsystems are required in three hardware subsystems:

- Host complex (including Front End Node and Service Node)
- I/O node
- Compute node

Figure 2-1 illustrates these components.

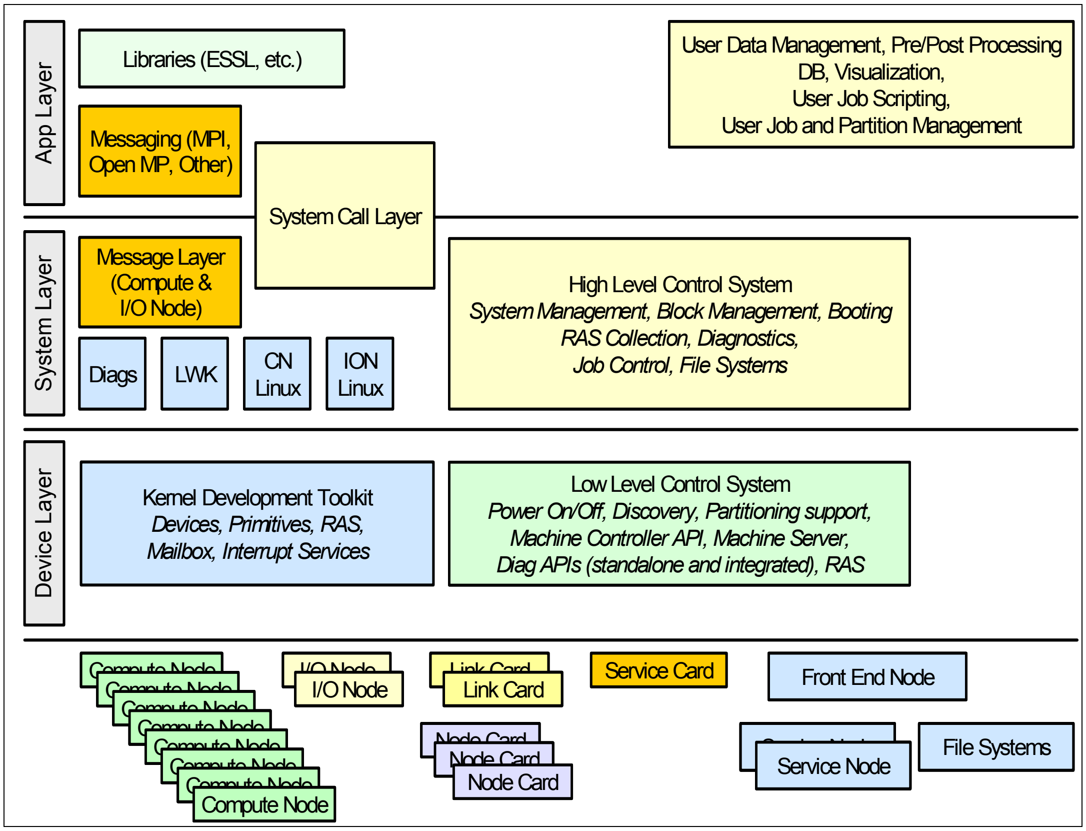

App Layer
System Layer
Device Layer
Libraries (ESSL, etc.)
Messaging (MPI, Open MP, Other)
System Call Layer
User Data Management, Pre/Post Processing
DB, Visualization,
User Job Scripting,
User Job and Partition Management
Message Layer (Compute & I/O Node)
Diags
LWK
CN Linux
ION Linux
High Level Control System
System Management, Block Management, Booting
RAS Collection, Diagnostics,
Job Control, File Systems
Kernel Development Toolkit
Devices, Primitives, RAS,
Mailbox, Interrupt Services
Low Level Control System
Power On/Off, Discovery, Partitioning support,
Machine Controller API, Machine Server,
Diag APIs (standalone and integrated), RAS
Compute Node
I/O Node
Link Card
Service Card
Front End Node
Node Card
Service Node
File Systems

Figure 2-1 Software stack overview

The software environment illustrated in Figure 2-1 relies on a series of header files and libraries. A selected set is listed in Appendix C, "Header files and libraries" on page 335.

2.2 Compute Node Kernel

The Compute Node Kernel (CNK) provides an environment for executing user processes. The CNK includes the following services:

- Process creation and management
- Memory management
- Process debugging
- Reliability, availability, and serviceability (RAS) management
- File I/O
- Network

The Compute Nodes on Blue Gene/P are implemented as quad cores on a single chip with 2 GB or 4 GB of dedicated physical memory in which applications run.

A process is executed on Blue Gene/P nodes in the following three main modes:

- Symmetrical Multiprocessing (SMP) node mode
- Virtual node (VN) mode
- Dual mode (DUAL)

Application programmers see the Compute Node Kernel software as a Linux-like operating system. This type of operating system is accomplished on the Blue Gene/P software stack by providing a standard set of run-time libraries for C, C++, and Fortran95. To the extent that is possible, the supported functions maintain open standard POSIX-compliant interfaces. We discuss the Compute Node Kernel further in Part 2, "Kernel overview" on page 27. Applications can access system calls that provide hardware or system features, as illustrated by the examples in Appendix B, "Files on architectural features" on page 331.

2.2.1 High-performance computing and High-Throughput Computing modes

When discussing Blue Gene/P, we refer to the parallel paradigms that rely on the network for communication, mainly through the Message-Passing Interface (MPI) as *high-performance computing (HPC)*. This topic is discussed in Chapter 7, "Parallel paradigms" on page 65. Blue Gene/P also offers a paradigm where applications do not require communication between tasks and each node is running a different instance of the application. We referred to this paradigm as *High-Throughput Computing (HTC)*. This topic is discussed in Chapter 12, "High-Throughput Computing (HTC) paradigm" on page 201.

2.2.2 Threading support on Blue Gene/P

The threading implementation on the Blue Gene/P system supports OpenMP. The XL OpenMP implementation provides a futex-compatible syscall interface, so that the Native POSIX Thread Library (NPTL) pthreads implementation in glibc runs without modification. These syscalls allow a limited number of threads and limited support for mmap(). The Compute Node Kernel provides a thread for I/O handling in MPI.

Important: The Compute Node Kernel supports a limited number of threads bound to CPU cores. The thread limit depends on the mode of the job and the application thread depth as described in detail in Chapter 4, "Execution process modes" on page 37.

2.3 Message Passing Interface on Blue Gene/P

The implementation of MPI on the Blue Gene/P system is the MPICH2 standard that was developed by Argonne National Labs. For more information about MPICH2, see the Message Passing Interface (MPI) standard Web site at:

`http://www-unix.mcs.anl.gov/mpi/`

A function of the MPI-2 standard that is not supported by Blue Gene/P is dynamic process management (creating new MPI processes).[16] However, the various thread modes are supported.

2.4 Memory considerations

On the Blue Gene/P system, the entire physical memory of a Compute Node is either 2 GB or 4 GB. Of that space, some is allocated for the CNK itself. In addition, shared memory space is also allocated to the user process at the time at which the process is created.

Important: In C, C++, and Fortran, the malloc routine returns a NULL pointer when users request more memory than the physical memory available. We recommend you always check malloc() return values for validity.

The Compute Node Kernel keeps track of collisions of stack and heap as the heap is expanded with a brk() syscall. The Blue Gene/P system includes stack guard pages.

The Compute Node Kernel and its private data are protected from read/write by the user process or threads. The code space of the process is protected from writing by the process or threads. Code and read-only data are shared between the processes in Virtual Node Mode unlike in the Blue Gene/L system.

In general, give careful consideration to memory when writing applications for the Blue Gene/P system. At the time this book was written, each node has 2 GB or 4 GB of physical memory.

As previously mentioned, memory addressing is an important topic in regard to the Blue Gene/P system. An application that stores data in memory falls into one of the following classifications:

data	Initialized static and common variables
bss	Uninitialized static and common variables
heap	Controlled allocatable arrays
stack	Controlled automatic arrays and variables

You can use the Linux **`size`** command to gain an idea of the memory size of the program. However, the **`size`** command does not provide any information about the run-time memory usage of the application nor on the classification of the types of data. Figure 2-2 illustrates memory addressing in HPC based on the different node modes that are available on the Blue Gene/P system.

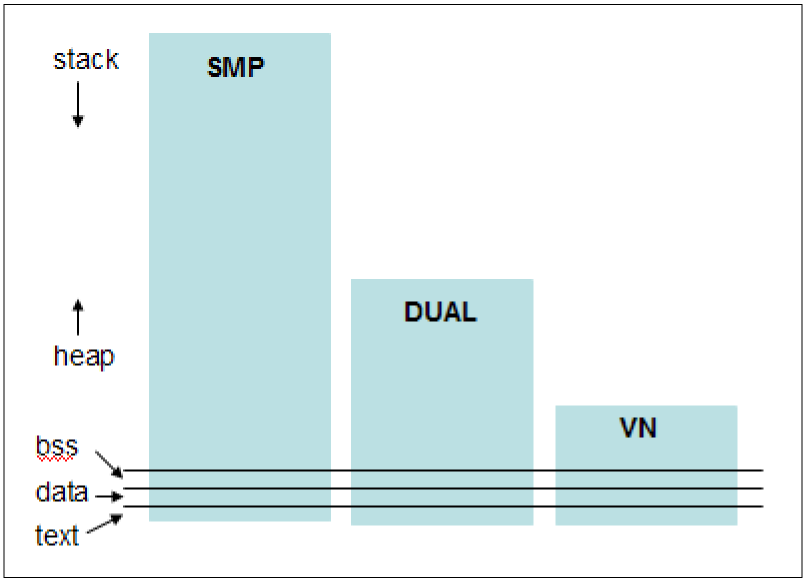


Figure 2-2 Memory addressing on the Blue Gene/P system as a function of the different node modes

2.4.1 Memory leaks

Given that no virtual paging exists on the Blue Gene/P system, any memory leaks in your application can quickly consume available memory. When writing applications for the Blue Gene/P system, you must be especially diligent that you release all memory that you allocate. This situation is true on any machine. Most of the time, having an application running on multiple architectures helps identify this type of problem.

2.4.2 Memory management

The Blue Gene/P computer implements a 32-bit memory model. It does not support a 64-bit memory model, but provides *large file support* and *64-bit integers*.

Two types of Blue Gene/P compute nodes are available: One provides 2 GB of memory per compute node, and the other provides 4 GB of memory per compute node. In the case of the 2 GB compute nodes, if the memory requirement per MPI task is greater than 512 MB in virtual node mode, greater than 1 GB in Dual mode, or greater than 2 GB in SMP mode, the application cannot run on the Blue Gene/P system. In the case of the 4 GB compute nodes, if the memory requirement per MPI task is greater than 1 GB in Virtual Node Mode, greater than 2 GB in Dual mode, or greater than 4 GB in SMP mode, the application cannot run on the Blue Gene/P system. The application works only if you take steps to reduce the memory footprint.

In some cases, you can reduce the memory requirement by distributing data that was replicated in the original code. In this case, additional communication might be needed. It might also be possible to reduce the memory footprint by being more careful about memory management in the application, such as by not defining arrays for the index that corresponds to the number of nodes.

2.4.3 Uninitialized pointers

Blue Gene/P applications run in the same address space as the Compute Node Kernel and the communications buffers. You can create a pointer that does not reference your own application’s data, but rather references the area used for communications. The Compute Node Kernel itself is well protected from rogue pointers.

2.5 Other considerations

It is important to understand that the operating system present on the Compute Node, the Compute Node Kernel, is not a full-fledged version of Linux. Therefore, you must use care in some areas, as explained in the following sections, when writing applications for the Blue Gene/P system. For a full list of supported system calls, see Chapter 6, “System calls” on page 51.

2.5.1 Input/output

I/O is an area where you must pay special attention in your application. The CNK does not perform I/O. This is carried out by the I/O node.

File I/O

A limited set of file I/O is supported. Do *not* attempt to use asynchronous file I/O because it results in run-time errors.

Standard input

Standard input (stdin) is supported on the Blue Gene/P system.

Sockets calls

Sockets are supported on the Blue Gene/P system. For additional information, see Chapter 6, “System calls” on page 51.

2.5.2 Linking

Dynamic linking is not supported on the Blue Gene/L system. However, it is supported on the Blue Gene/P system. You can now statically link all code into your application or use dynamic linking.

2.6 Compilers overview

Read-only sections are supported in the Blue Gene/P system. However, *this is not true of read-only sections within dynamically located modules*.

2.6.1 Programming environment overview

The diagram in Figure 2-3 provides a quick view into the software stack that supports the execution of Blue Gene/P applications.

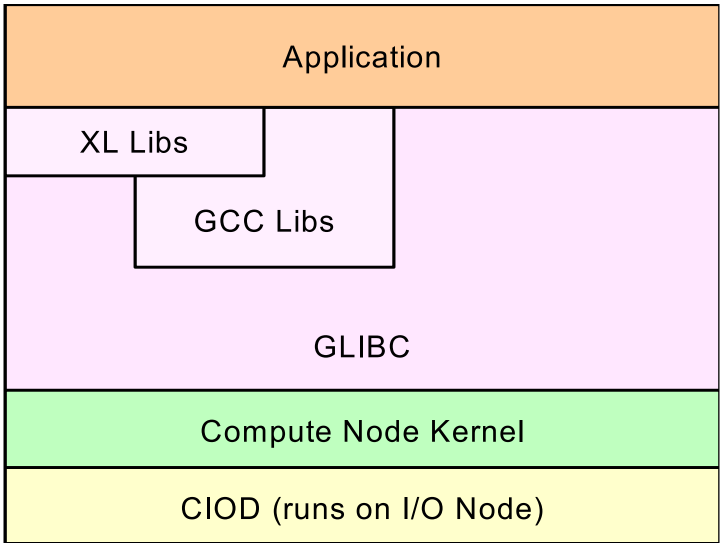


Figure 2-3 Software stack supporting the execution of Blue Gene/P applications

2.6.2 GNU Compiler Collection

The standard GNU Compiler Collection V4.1.2 for C, C++, and Fortran is supported on the Blue Gene/P system. The current versions are:

- gcc 4.1.2
- binutils 2.17
- glibc 2.4

You can find the GNU Compiler Collection in the /bgsys/drivers/ppcfloor/gnu-linux/bin directory. For more information, see Chapter 8, “Developing applications with IBM XL compilers” on page 97.

2.6.3 IBM XL compilers

The following IBM XL compilers are supported for developing Blue Gene/P applications:

- XL C/C++ Advanced Edition V9.0 for Blue Gene/P
- XL Fortran Advanced Edition V11.1 for Blue Gene/P

See Chapter 8, “Developing applications with IBM XL compilers” on page 97, for more compiler-related information.

2.7 I/O node software

The Blue Gene/P system is a massively parallel system with a large number of nodes. Compute Nodes are reserved for computations, and I/O is carried out using the I/O nodes. These nodes serve as links between the Compute Nodes and external devices. For instance, applications running on Compute Nodes can access file servers and communicate with processes in other machines.

The I/O node software and the Service Node software communicate to exchange various data relating to machine configuration and workload. Communications use a key-based authentication mechanism with keys using at least 256 bits.

The I/O node kernel is a standard Linux kernel and provides file system access and sockets communication to applications that execute on the Compute Nodes.

2.7.1 I/O nodes kernel boot considerations

The I/O node kernel is designed to be booted as infrequently as possible due to the numerous possible failures of mounting remote file systems. The bootstrap process involves loading a ramdisk image and booting the Linux kernel. The ramdisk image is extracted to provide the initial file system, which contains minimal commands to mount the file system on the Service Node using the Network File System (NFS). The boot continues by running startup scripts from the NFS and running customer-supplied startup scripts to perform site-specific actions, such as logging configuration and mounting high-performance file systems.

The Blue Gene/P system has considerably more content over the Blue Gene/L system in the ramdisk image to reduce the load on the Service Node exported by the NFS as the I/O node boot. Toolchain shared libraries and all of the basic Linux text and shell utilities are local to the ramdisk. Packages, such as GPFS, and customer-provided scripts are NFS mounted for administrative convenience.

2.7.2 I/O node file system services

The I/O node kernel supports an NFS client or GPFS client, which provides a file system service to application processes that execute on its associated Compute Node. The NFSv3 and GPFS file systems supported as part of the Blue Gene/L system continue with the Blue Gene/P system. As with the Blue Gene/L system, customers can still add their own parallel file systems by modifying Linux on the I/O node as needed.

2.7.3 Socket services for the Compute Node Kernel

The I/O node includes a complete Internet Protocol (IP) stack, with TCP and UDP services. A subset of these services is available to user processes running on the Compute Node that is associated with an I/O node. Application processes communicate with processes running on other systems using client-side sockets through standard socket permissions and network connectivity. In addition, server-side sockets are available.

Note that the I/O node implements the sockets so that all the Compute Nodes within a processor set (pset) behave as though the compute tasks are executing on the I/O node. In particular, this means that the socket port number is a single address space within the pset and they share the IP address of the I/O node.

2.7.4 I/O node daemons

The I/O node includes the following daemons:

- Control and I/O services daemons
- File system client daemons
- Syslog
- sshd
- ntpd on at least one I/O node

2.7.5 Control system

The control system retains the high-level components from the IBM Blue Gene/L system with a considerable change in low-level components to accommodate the updated control hardware in the Blue Gene/P system as well as to increase performance for the monitoring system. The MMCS server and mcserver are now the processes that make up the control system on the Blue Gene/P system:

- The Midplane Management Control System (MMCS, console and server) is similar to the Blue Gene/L system in the way it handles commands, interacts with IBM DB2, boots blocks, and runs jobs.
- mcServer is the process through which MMCS makes contact with the hardware (replacing idoproxy of the Blue Gene/L system). mcServer handles all direct interaction with the hardware. Low-level boot operations are now part of this process and not part of MMCS.
- The standard **`mpirun`** command to launch jobs can be used from any Front End Node or the Service Node. This command is often invoked automatically from a higher level scheduler.

The components that reside on the Service Node contain the following functions:

- Bridge APIs

  A scheduler that dynamically creates and controls blocks typically uses Bridge APIs. A range of scheduler options includes ignoring these APIs and using **`mpirun`** on statically created blocks to full dynamic creation of blocks with pass-through midplanes. The Blue Gene/P system includes a new set of APIs that notifies the caller of any changes in real time. Callers can register for various entities (*entities* are jobs, blocks, node cards, midplanes, and switches) and only see the changes. Callers can also set filters so that notifications occur for only specific jobs or blocks.

- ciodb

  ciodb is now integrated as part of the MMCS server for the Blue Gene/P system, which is different from the Blue Gene/L system. ciodb is responsible for launching jobs on already booted blocks. Communication to ciodb occurs through the database and can be initiated by either **`mpirun`** or the Bridge APIs.

- MMCS

  The MMCS daemon is responsible for configuring and booting blocks. It can be controlled either by a special console interface (similar to the Blue Gene/L system) or by the Bridge APIs. The MMCS daemon also is responsible for relaying RAS information into the RAS database.

- mcServer

  The mcServer daemon has low-level control of the system, which includes a parallel efficient environmental monitoring capability as well as a parallel efficient reset and code load capability for configuring and booting blocks on the system. The diagnostics for the Blue Gene/P system directly leverage this daemon for greatly improved diagnostic performance over that of the Blue Gene/L system.

- bgpmaster

  The bgpmaster daemon monitors the other daemons and restarts any failed components automatically.

- Service actions

  Service actions are a suite of administrative shell commands that are used to service hardware. They are divided into device-specific actions with a “begin” and “end” action. Typically the “begin” action powers down hardware so it can be removed from the system, and the “end” action powers up the replacement hardware. The databases are updated with these operations, and they coordinate automatically with the scheduling system as well as the diagnostic system.

- Submit server daemon

  The submit server deamon is the central resource manager for High-Throughput Computing (HTC) partitions. When MMCS boots a partition in HTC mode, each partition registers itself with the submit server deamon before going to initialized state. This registration process includes several pieces of information, such as partition mode (SMP, DUAL, VN), partition size, user who booted the partition, and the list of users who can run on the partition. This information is maintained in a container and is used to match resource requests from submit commands based on their job requirements.

- mpirund

  The mpirund is a daemon process running on the service node whose purpose is to handle connections from frontend mpirun processes, and fork backend mpirun processes.

- Real-time server

  The Real-time Notification APIs are designed to eliminate the need for a resource management system to constantly have to read in all of the machine state to detect changes. The APIs allow the caller to be notified in real-time of state changes to jobs, blocks, and hardware, such as base partitions, switches, and node cards. After a resource management application has obtained an initial snapshot of the machine state using the Bridge APIs, the Bridge APIs can then determine to be notified only of changes, and the Real-time Notification APIs provides that mechanism.

2.8 Management software

The Blue Gene/P management software is based on a a set of databases that run on the Service Node. The database software is DB2.

2.8.1 Midplane Management Control System

Both Blue Gene/P hardware and software are controlled and managed by the MMCS. The Service Node, Front End Node, and the file servers are not under the control of the MMCS. The MMCS currently consists of several functions that interact with a DB2 database running on the Service Node.

Part 2

Kernel overview

The kernel provides the glue that makes all components in Blue Gene/P work together. In this part, we provide an overview of the kernel functionality for applications developers. This part is for those who require information about system-related calls and interaction with the kernel.

This part contains the following chapters:

3

Kernel functionality

In this chapter, we provide an overview of the functionality that is implemented as part of the Compute Node Kernel and I/O node kernel. We discuss the following topics:

- System software overview
- Compute Node Kernel
- I/O node kernel

3.1 System software overview

In general, the function of the kernel is to enable applications to run on a particular hardware system. This enablement consists of providing such services as applications execution, file I/O, memory allocation, and many others. In the case of the Blue Gene/P system, the system software provides two kernels:

- Compute Node Kernel (CNK)
- I/O node kernel

3.2 Compute Node Kernel

The kernel that runs on the Compute Node is called the *Compute Node Kernel (CNK)* and is IBM proprietary. It has a subset of the Linux system calls. The CNK is a flexible, lightweight kernel for Blue Gene/P Compute Nodes that can support both diagnostic modes and user applications.

The CNK is intended to be a Linux-like operating system, from the application point-of-view, supporting a large subset of Linux compatible system calls. This subset is taken from the subset used successfully on the Blue Gene/L system, which demonstrates good compatibility and portability with Linux.

Now, as part of the Blue Gene/P system, the CNK supports threads and dynamic linking for further compatibility with Linux. The CNK is tuned for the capabilities and performance of the Blue Gene/P System. In Figure 3-1, you see the interaction between the application space and the kernel space.

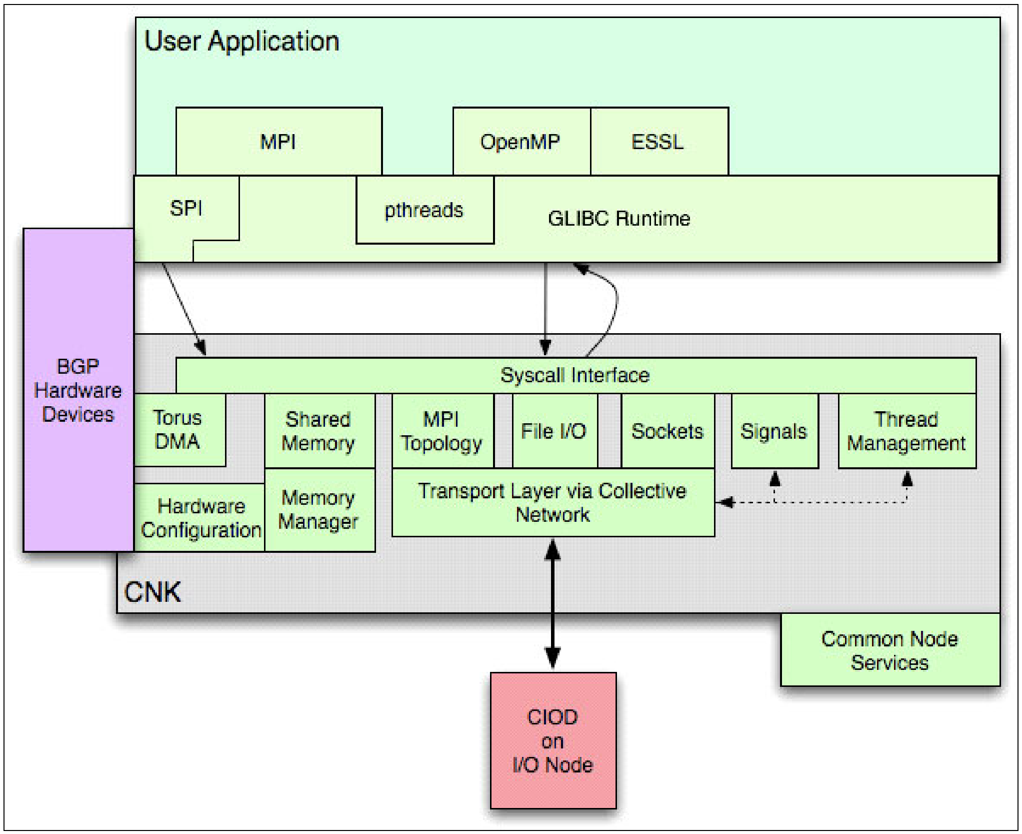


Figure 3-1 Compute Node Kernel overview

When running a user application, the CNK connects to the I/O node through the collective network. This connection communicates to a process that is running on the I/O node called the *control and I/O daemon (CIOD)*. All function-shipped system calls are forwarded to the CIOD process and executed on the I/O node.

At the user-application level, the Compute Node Kernel supports the following APIs among others:

- Message Passing Interface (MPI)[17] support between nodes using MPI library support
- Open Multi-Processing (OpenMP)[18] API
- Standard IBM XL family of compilers support with XLC/C++, XLF, and GNU Compiler Collection[19]
- Highly optimized mathematical libraries, such as IBM Engineering and Scientific Subroutine Library (ESSL)[20]
- GNU Compiler Collection (GCC) C Library, or glibc, which is the C standard library and interface of GCC for a provider library plugging into an other library (system programming interfaces (SPIs))

CNK provides the following services:

- Torus direct memory access (DMA),[21] which provides memory access for reading, writing, or doing both independently of the processing unit
- Shared-memory access on a local node
- Hardware configuration
- Memory management
- MPI topology
- File I/O
- Sockets connection
- Signals
- Thread management
- Transport layer via collective network

3.2.1 Boot sequence of a Compute Node

The Blue Gene/P hardware is a stateless system. When power is initially applied, the hardware must be externally initialized. Given the architectural and reliability improvements in the Blue Gene/P design, reset of the Compute Nodes should be an infrequent event.

The following procedure explains how to boot a Compute Node as part of the main partition. Independent reset of a single Compute Node and independent reset of a single I/O node are different procedures.

The CNK must be loaded into memory after every reset of the Compute Node. To accomplish this task, several steps must occur to prepare a CNK for running an application:

1. The control system loads a small bootloader into SRAM.
2. The control system loads the *personality* into SRAM. The personality is a data structure that contains node-specific information, such as the X, Y, Z coordinates of the node.

> **Note:** See Appendix B, “Files on architectural features” on page 331, for an example of how to use a personality.

3. The control system releases the Compute Node from reset.
4. The bootloader starts executing and initializes the hardware.
5. The bootloader communicates with the control system over the mailbox to load Common Node Services and CNK images.
6. The bootloader then transfers control to the Common Node Services.
7. Common Node Services performs its set up and then transfers control to the CNK.
8. The Compute Node Kernel performs its set up and communicates to the CIOD.

At this point, the Compute Node Kernel submits the job to the CIOD.

3.2.2 Common Node Services

Common Node Services provide low-level services that are both specific to the Blue Gene/P system and common to the Linux and the CNK. As such, these services provide a consistent implementation across node types while insulating the kernels from the details of the control system.

The common node services provide the same low-level hardware initialization and setup interfaces to both Linux and the CNK.

The common node services provide the following services:

- Access to the SRAM mailbox for performing I/O operations over the service network to the console
- Initialization of hardware for various networks
- Access to the personality
- Low-level services for RAS, including both event reporting and recovery handling
- Access to the IBM Blue Gene interrupt controller

3.3 I/O node kernel

The kernel of the I/O node, shown in Figure 3-2 on page 33, is referred to as the *Mini-Control Program* (*MCP*). It is a port of the Linux kernel, which means it is GPL/LGPL licensed. It is similar to the Blue Gene/L I/O node. The characteristics of the I/O node kernel on the Blue Gene/P system are:

- Embedded Linux kernel:
  - Linux Version 2.6.16
  - Four-way symmetrical multiprocessing (SMP)
  - Paging disabled (no swapping available)
- Ethernet:
  - 10 gigabit Ethernet driver
  - Large Maximum Transmission Unit (MTU) support, which allows Ethernet frames to be increased from the default value of 1500 bytes to 9000 bytes
  - TCP checksum offload engine support
  - Availability of /proc files for configuring and gathering status

- File systems supported:
  - Network File System (NFS)
  - Parallel Virtual File System (PVFS)
  - IBM General Parallel File System (GPFS)
  - Lustre File System
- CIOD:
  - Lightweight proxy between Compute Nodes and the outside world
  - Debugger access into the Compute Nodes
  - SMP support

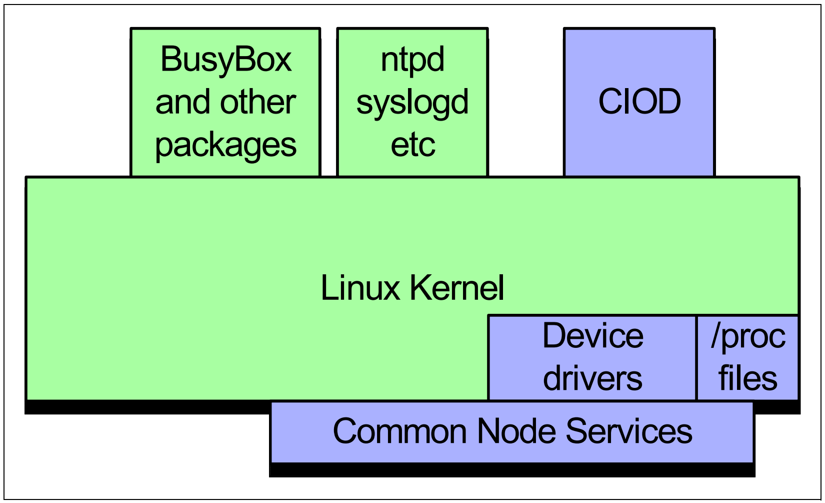


Figure 3-2 I/O node kernel overview

3.3.1 Control and I/O daemon

The Control and I/O daemon, CIOD, is a user-mode daemon that runs on the I/O node and provides a bridge between the compute nodes and the outside world. CIOD takes messages from multiple input sources, processes each message, and returns a result. CIOD accepts messages from three sources:

- MMCS sends messages using the CioStream and DataStream protocols on two socket connections from the Service Node. MMCS sends messages to load and start a new job, send standard input, and send a signal to a running job. CIOD sends messages to report status and send standard output and standard error.
- Compute nodes send messages using the Cio protocol on the collective network. The messages are for control, function shipping I/O, and debugging.
- An external tool, such as a debugger, sends messages using the CioDebug protocol on pipes from another I/O node process.

Figure 3-3 on page 34 shows a high-level overview of CIOD.

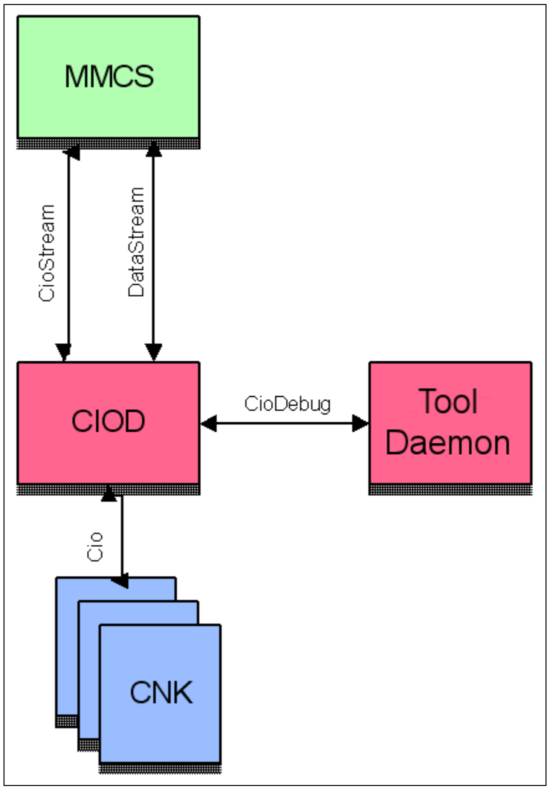


Figure 3-3 High-level overview of CIOD

CIOD threading architecture

CIOD reads from the collective network and places the message into the shared memory dedicated to the sending node I/O proxy. Figure 3-4 on page 35 shows the threading architecture of the CIOD.

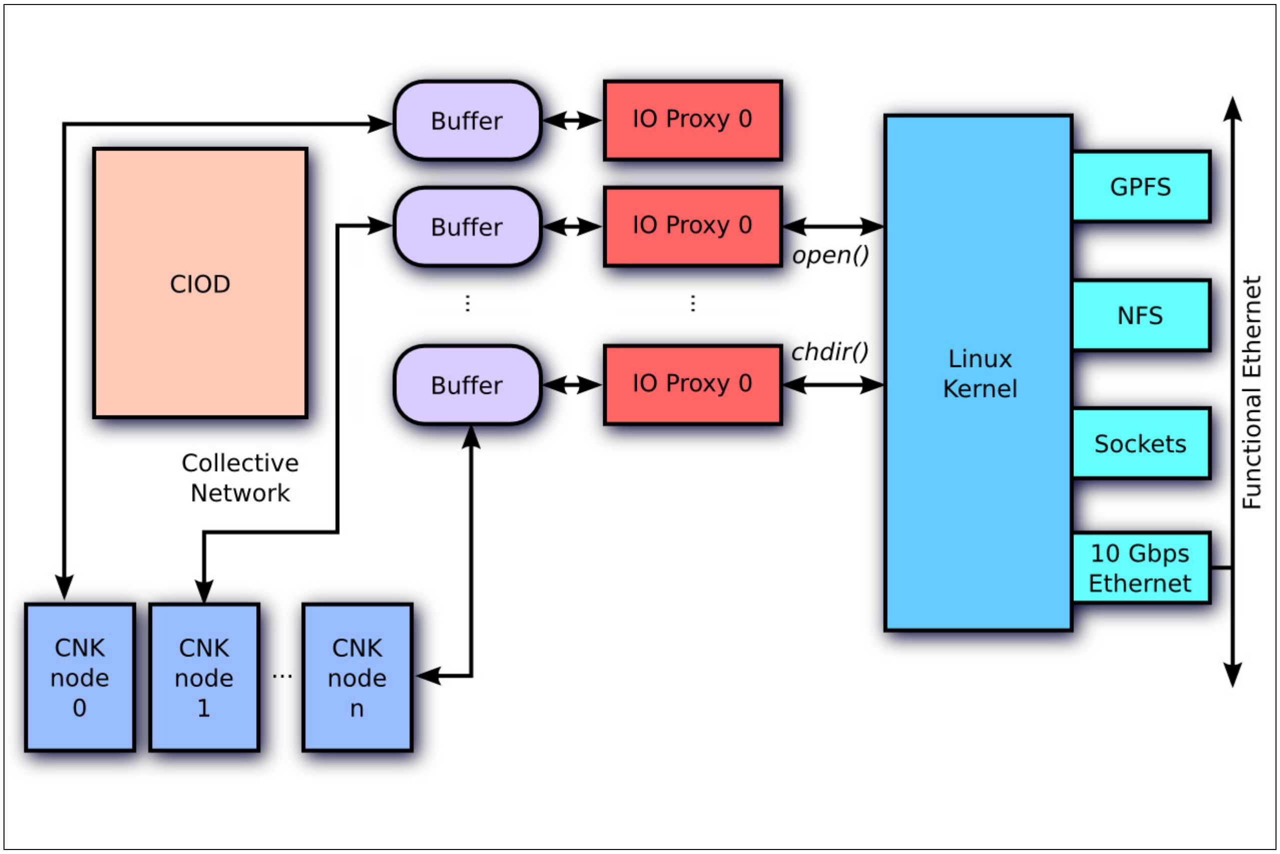


Figure 3-4 CIOD threading architecture

Jobs directory

Before CIOD starts a job it creates a directory called /jobs/<jobId>, where <jobId> is the ID of the job as assigned by MMCS. This directory can be accessed by jobs running on the compute nodes that are connected to the I/O node. The directory for each job contains the following entries:

- `exe`

  A symlink to the executable.
- `cmdline`

  A file with the list of arguments given to the program.
- `environ`

  A file with the list of environment variables given to the program.
- `wdir`

  A symlink to the initial working directory for the program.
- `noderankmap`

  A file that contains the mapping of node location to MPI rank. This file is created only when a tool daemon is started.

The /jobs directory is owned by root and has read and execute permission for everybody (r-xr-xr-x), whereas the individual job directory is owned by the user that started the job and has read and execute permission for everybody.

When the job completes, CIOD removes the /jobs/<jobId> directory.

4

Execution process modes

The Compute Nodes on IBM Blue Gene/P are implemented as four cores on a single chip with 2 GB or 4 GB of dedicated physical memory in which applications run. A process executes on Blue Gene/P in one of the following three modes:

- Symmetric multiprocessing (SMP) mode
- Virtual node mode (VN)
- Dual mode (DUAL)

In this chapter, we explore these modes in detail.

4.1 Symmetrical Multiprocessing mode

In Symmetric Multiprocessing (SMP) mode each physical Compute Node executes a single process per node with a default maximum of four threads. The Blue Gene/P system software treats the four cores in a Compute Node symmetrically. This mode provides the maximum memory per process.

Figure 4-1 shows the interaction of SMP mode between the application space and the kernel space. The task or process can have up to four threads. Pthreads and OpenMP are supported in this mode.

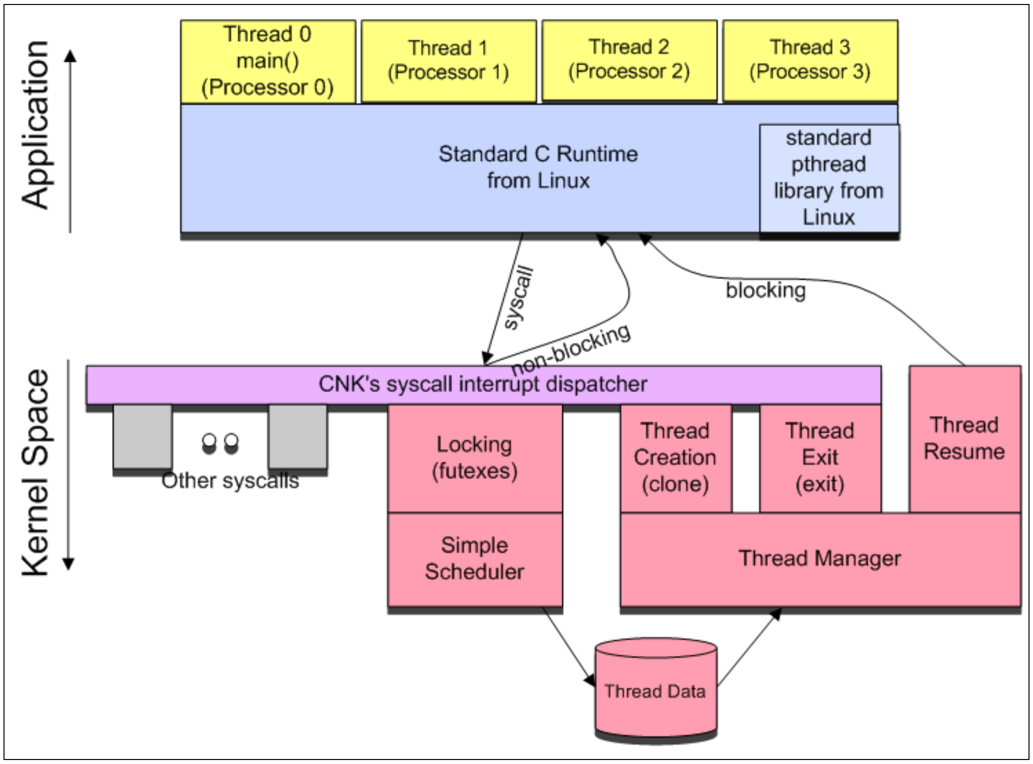


Figure 4-1 SMP Node mode

4.2 Virtual Node mode

In Virtual Node mode (VN mode) the kernel can run four separate processes on each Compute Node. Node resources (primarily the memory and the torus network) are shared by all processes on a node. In Figure 4-2 on page 39, VN mode is illustrated with four tasks per node and one thread per process. Shared memory is available between processes.

Note: Shared memory is available only in HPC mode and not in HTC mode.

In VN mode, an MPI application can use all of the cores in a node by quadrupling the number of MPI tasks. The distinct MPI tasks that run on each of the four cores of a Compute Node communicate with each other transparently through direct memory access (DMA) on the node. DMA puts data that is destined for a physically different node on the torus, while it locally copies data when it is destined for the same physical node.

In VN mode, the four cores of a Compute Node act as different processes. Each has its own rank in the message layer. The message layer supports VN mode by providing a correct torus to rank mapping and first in, first out (FIFO) pinning. The hardware FIFOs are shared equally between the processes. Torus coordinates are expressed by quadruplets instead of triplets. In VN mode, communication between the four threads in a Compute Node occurs through DMA local copies.

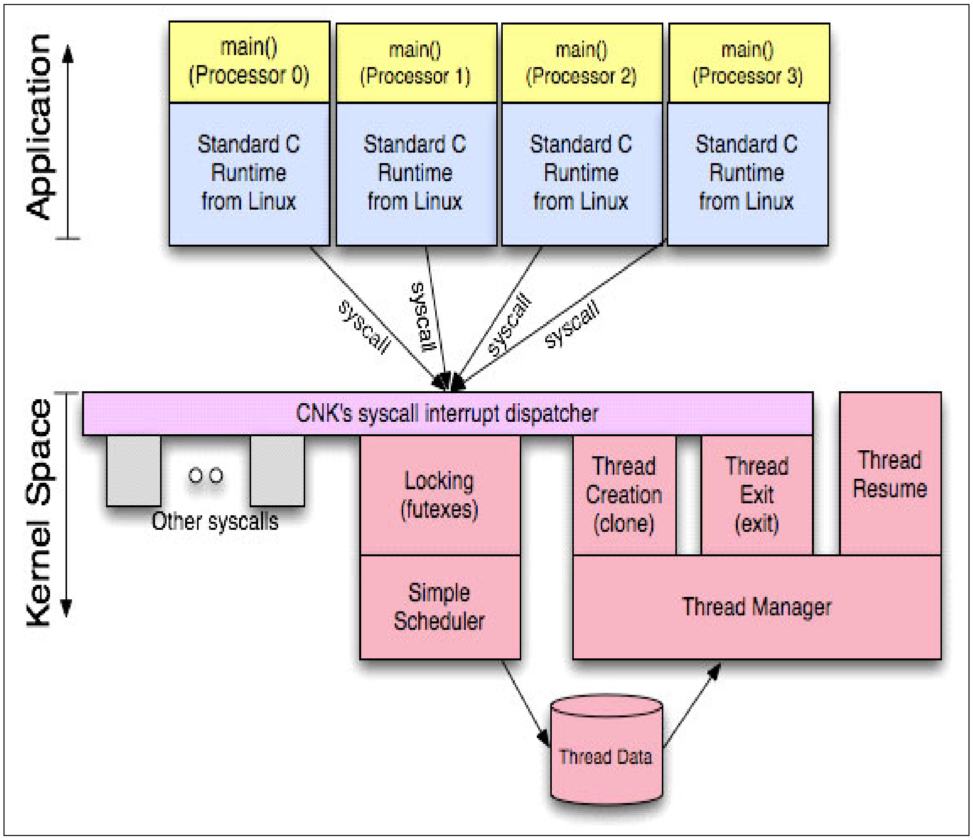


Figure 4-2 Virtual node mode

Each core executes one compute process. Processes that are allocated in the same Compute Node share memory, which can be reserved at job launch. An application that wants to run with four tasks per node can dedicate a large portion for shared memory, if the tasks need to share global data. This data can be read/write, and data coherency is handled in hardware.

The Blue Gene/P MPI implementation supports VN mode operations by sharing the systems communications resources of a physical Compute Node between the four compute processes that execute on that physical node. The low-level communications library of the Blue Gene/P system, that is the message layer, virtualizes these communications resources into logical units that each process can use independently.

4.3 Dual mode

In Dual mode (DUAL mode), each physical Compute Node executes two processes per node with a default maximum of two threads per process. Each task in Dual mode gets half the memory and cores so that it can run two threads per task.

Figure 4-3 shows two processes per node. Each process can have up to two threads. OpenMP and pthreads are supported. Shared memory is available between processes. Threads are pinned to a processor.

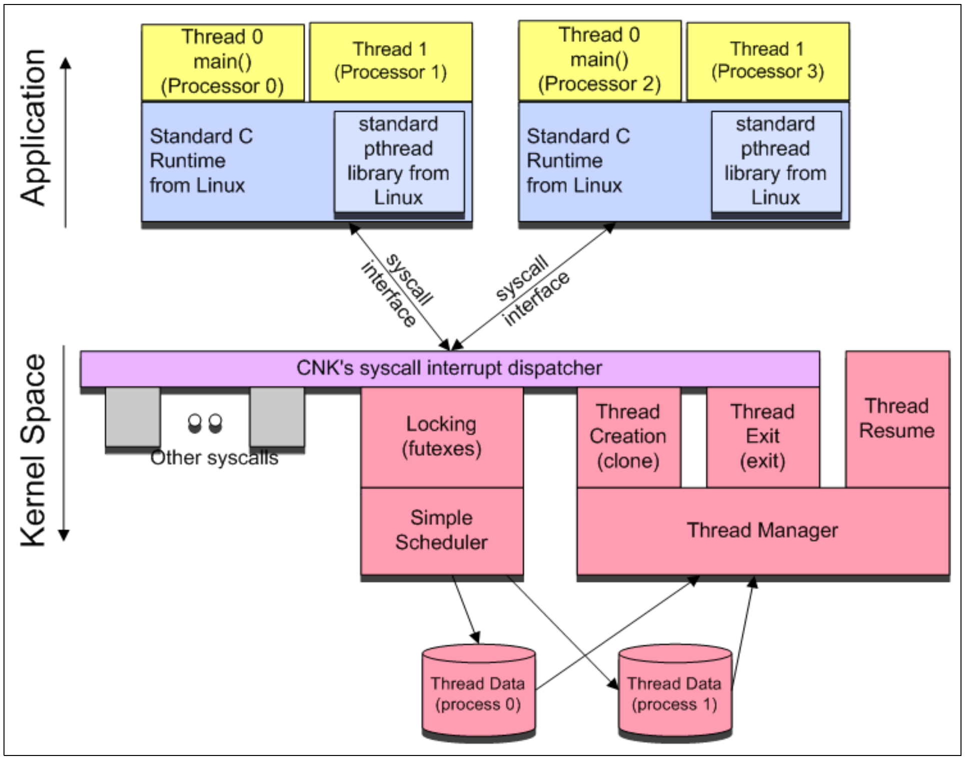


Figure 4-3 Dual mode

4.4 Shared memory support

Shared memory is supported in Dual mode and Virtual Node mode. Shared memory in SMP mode is not necessary because each processor already has access to all of the node's memory.

Shared memory is allocated using standard Linux methods (shm_open() and mmap()). However, because the CNK does not have virtual pages, the physical memory that backs the shared memory must come out of a memory region that is dedicated for shared memory. This memory region has its size fixed at job launch.

BG_SHAREDMEMPOOLSIZE: The BG_SHAREDMEMPOOLSIZE environment variable specifies in MB the amount of memory to be allocated, which you can do using the **`mpirun -env`** flag, for example, `BG_SHAREDMEMPOOLSIZE=8` allocates 8 MB of shared memory storage.

You can change the amount of memory to be set aside for this memory region at job launch. Figure 4-4 on page 41 illustrates shared-memory *allocation*.

```
fd = shm_open( SHM_FILE, O_RDWR, 0600 );
ftruncate( fds[0], MAX_SHARED_SIZE );
shmptr1 = mmap( NULL, MAX_SHARED_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd,
0);
```

Figure 4-4 Shared-memory allocation

Figure 4-5 illustrates shared-memory *deallocation*.

```
munmap(shmptrl, MAX_SHARED_SIZE);
close(fd)
shm_unlink(SHM_FILE);
```

Figure 4-5 shared-memory deallocation

The shm_open() and shm_unlink() routines access a pseudo-device, /dev/shm/*filename*, which the kernel interprets. Because multiple processes can access or close the shared-memory file, allocation and deallocation are tracked by a simple reference count; therefore, the processes do not need to coordinate deallocation of the shared memory region.

4.5 Deciding which mode to use

The choice of the job mode largely depends on the type of application and the parallel paradigm that was implemented for a particular application. The obvious case involves applications where a hybrid paradigm between MPI and OpenMP or pthreads was implemented. In this case, it is beneficial to use SMP mode. If you are writing single-threaded applications, consider Virtual Node mode.

I/O-intensive tasks that require a relatively large amount of data interchange between Compute Nodes benefit more by using Virtual Node mode. Those applications that are primarily CPU bound do not have large working memory requirements and can take advantage of more processors and run more efficiently in Virtual Node mode.

Finally, High-Throughput Computing (HTC) mode offers the possibility of running multiple instances of an application that does not require communication between nodes. This mode allows applications to run on single nodes and permits other users to share single nodes in the same partition. See Chapter 12, “High-Throughput Computing (HTC) paradigm” on page 201 for more information.

4.6 Specifying a mode

The default mode for **mpirun** is SMP. To specify Virtual Node mode or Dual mode, use the following commands:

```
mpirun ... -mode VN ...
mpirun ... -mode DUAL ...
```

See Chapter 11, “mpirun” on page 177 for more information about the **mpirun** command.

In the case of the **submit** command, you use the following commands:

```
submit ... -mode VN ...
submit ... -mode DUAL ...
```

See Chapter 12, “High-Throughput Computing (HTC) paradigm” on page 201 for more information about the **submit** command.

4.7 Multiple application threads per core

In the previous sections of this chapter we described the CNK as allowing applications a single application thread per core. CNK can be configured to allow a job to have multiple application threads per core. Support for multiple application threads per core is meant primarily for an application that switches between programming models in phases, such as an application where part of the program is written to use OpenMP and another part uses pthreads. The application needs to use some method to put one set of threads to sleep and wake the other set of threads when switching between programming models.

There are limitations that the developer must keep in mind regarding CNK support for multiple application threads per core that do not apply when running on a full Linux kernel. The number of application threads per core can be configured to be between one and three. If the application attempts to create more threads than are allowed the creation attempt will fail. When a thread is created, the CNK pins the thread to the core with the fewest threads on it. A core does not automatically switch between threads on a timed basis. Switches occur either through a sched_yield() system call, signal delivery, or futex wakeup.

The number of application threads per core is specified by the BG_APPTHREADDEPTH environment variable. If this environment variable is not set, the number of application threads per core is one. The allowed values for the environment variable are 1, 2, and 3. Even if this environment variable is not set, the kernel puts the variable with the value in the job’s environment so that an application can get the value. Changing the environment variable after the job is running does not change the number of allowed application threads per core.

You can use this feature in any node mode (SMP, DUAL, or VN) and any partition mode (HPC or HTC).

To pass an environment variable to a Blue Gene/P job, the user of **mpirun** or **submit** can use the -env option on the command line, for example, the following command sets the number of application threads per core to be 3.

```
mpirun ... -env BG_APPTHREADDEPTH=3 ...
```

The ability to run multiple application threads per core is new in Blue Gene/P release V1R4M0.

5

Memory

In this chapter, we provide an overview of the memory subsystem and explain how it relates to the Compute Node Kernel. This chapter includes the following topics:

- Memory overview
- Memory management
- Memory protection
- Persistent memory

5.1 Memory overview

Similar to the Blue Gene/L system, Blue Gene/P nodes support virtual memory. Memory is laid out as a single, flat, fixed-size virtual address space shared between the operating system kernel and the application program.

The Blue Gene/P system is a distributed-memory supercomputer, which includes an on-chip cache hierarchy, and memory is off-chip. It contains optimized on-chip symmetrical multiprocessing (SMP) support for locking and communication between the four ASIC processors.

The aggregate memory of the total machine is distributed in the style of a multi-computer, with no hardware sharing between nodes. The total physical memory amount supported is either 2 GB or 4 GB per Compute Node, depending on the type of compute node that was installed.

The first level (L1) cache is contained within the IBM PowerPC 450 core (see Figure 1-3 on page 9). The PowerPC 450 L1 cache is 64-way set associative.

The second level (L2R and L2W) caches, one dedicated per core, are 2 KB in size. They are fully associative and coherent. They act as prefetch and write-back buffers for L1 data. The L2 cache line is 128 bytes in size. Each L2 cache has one connection toward the L1 instruction cache running at full processor frequency. Each L2 cache also has two connections toward the L1 data cache, one for the writes and one for the loads, each running at full processor frequency. Read and write are 16 bytes wide.

The third level (L3) cache is 8-way set associative, 8 MB in size, with 128-byte lines. Both banks can be accessed by all processor cores. The L3 cache has three write queues and three read queues: one for each processor core and one for the 10 gigabit network. Ethernet and direct memory access (DMA) share the L3 ports. Only one unit can use the port at a time. The Compute Nodes use DMA, and the I/O Nodes use Ethernet. The last one is used on the I/O Node and for torus network DMA on the compute networks. All the write queues go across a four-line write buffer to access the eDRAM bank. Each of the two L3 banks implements thirty 128-byte-wide write combining buffers, for a total of sixty 128-byte-wide write combining buffers per chip.

Table 5-1 provides an overview of some of the features of different memory components.

Table 5-1 Memory system overview

Cache	Total per node	Size	Replacement policy	Associativity
L1 instruction	4	32 KB	Round-Robin	► 64-way set-associative ► 16 sets ► 32-byte line size
L1 data	4	32 KB	Round-Robin	► 64-way set-associative ► 16 sets ► 32-byte line size
L2 prefetch	4	14 x 256 bytes	Round-Robin	► Fully associative (15-way) ► 128-byte line size
L3	2	2 x 4 MB	Least recently used	► 8-way associative ► 2 bank interleaved ► 128-byte line size
Double data RAM (DDR)	2	► Minimum 2 x 512 MB ► Maximum 4 GB	N/A	► 128-byte line size

5.2 Memory management

You must give careful consideration to managing memory on the Blue Gene/P system. This is particularly true in order to achieve optimal performance. The memory subsystem of Blue Gene/P nodes has specific characteristics and limitations that the programmer should know about.

5.2.1 L1 cache

On the Blue Gene/P system, the PowerPC 450 internal L1 cache does not have automatic prefetching. Explicit cache touch instructions are supported. Although the L1 instruction cache was designed with support for prefetches, it was disabled for efficiency reasons.

Figure 1-3 on page 9 shows the L1 caches in the PowerPC 450 architecture. The size of the L1 cache line is 32 bytes. The L1 cache has two buses toward the L2 cache: one for the stores and one for the loads. The buses are 128 bits in width and run at full processor frequency. The theoretical limit is 16 bytes per cycle. However, 4.6 bytes is achieved on L1 load misses, and 5.6 bytes is achieved on all stores (write through). This value of 5.6 bytes is achieved for the stores but not for the loads. The L1 cache has only a three-line fetch buffer. Therefore, there are only three outstanding L1 cache line requests. The fourth one waits for the first one to complete before it can be sent.

The L1 hit latency is four cycles for floating point and three cycles for integer. The L2 hit latency is at about 12 cycles for floating point and 11 cycles for integer. The 4.6-byte throughput limitation is a result of the limited number of line fill buffers, L2 hit latency, the policy when a line fill buffer commits its data to L1, and the penalty of delayed load confirmation when running fully recoverable.

Because only three outstanding L1 cache line load requests can occur at the same time, at most three cache lines can be obtained every 18 cycles. The maximum memory bandwidth is three times 32 bytes divided by 18 cycles, which yields 5.3 bytes per cycle, which written as an equation looks like this:

```
(3 x 32 bytes) / 18 cycles = 5.3 bytes per cycle
```

Important:

- Avoid instructions when prefetching data in the L1 cache on the Blue Gene/P system. Using the processor, you can concurrently fill in three L1 cache lines. Therefore, it is mandatory to reduce the number of prefetching streams to three or less.

  To optimize the floating-point units (FPUs) and feed the floating-point registers, you can use the XL compiler directives or assembler instructions (dcbt) to prefetch data in the L1 data cache. The applications that are specially tuned for IBM POWER4™ or POWER5™ processors that take advantage of four or eight prefetching engines will choke the memory subsystem of the Blue Gene/P processor.

- To take advantage of the single-instruction, multiple-data (SIMD) instructions, it is essential to keep the data in the L1 cache as much as possible. Without an intensive reuse of data from the L1 cache and the registers, because of the number of registers, the memory subsystem is unable to feed the double FPU and provide two multiply-addition operations per cycle.

In the worst case, SIMD instructions can hurt the global performance of the application. For that reason, we advise that you disable the SIMD instructions in the porting phase by compiling with `-qarch=450`. Then recompile the code with `-qarch=450d` and analyze the performance impact of the SIMD instructions. Perform the analysis with a data set and a number of processors that is realistic in terms of memory usage.

Optimization tips:

- The optimization of the applications must be based on the 32 KB of the L1 cache.
- The benefits of the SIMD instructions can be canceled out if data does not fit in the L1 cache.

5.2.2 L2 cache

The L2 cache is the hardware layer that provides the link between the embedded cores and the Blue Gene/P devices, such as the 8 MB L3-eDRAM and the 32 KB SRAM. The 2 KB L2 cache line is 128 bytes in size. Each L2 cache is connected to one processor core.

The L2 design and architecture were created to provide optimal support for the PowerPC 450 cores for scientific applications. Thus, a logic for automatic sequential stream detection and prefetching to the L2 added on the PowerPC 440 is still available on PowerPC 450. The logic is optimized to perform best on sequential streams with increasing addresses. The L2 boosts overall performance for almost any application and does not require any special software provisions. It autonomously detects streams, issues the prefetch requests, and keeps the prefetched data coherent.

You can achieve latency and bandwidth results close to the theoretical limits (4.6 bytes per cycle) dictated by the PowerPC 450 core by doing careful programming. The L2 accelerates memory accesses for one to seven sequential streams.

5.2.3 L3 cache

The L3 cache is 8 MB in size. The line size is 128 bytes. Both banks are directly accessed by all processor cores and the 10 Gb network, only on the I/O Node, and are used in Compute Nodes for torus DMA. There are three write queues and three read queues. The read queues directly access both banks.

Each L3 cache implements two sets of write buffer entries. Into each of the two sets, one 32-byte data line can deposit a set per cycle from any queue. In addition, one entry can be allocated for every cycle in each set. The write rate for random data is much higher in the Blue Gene/P system than in the Blue Gene/L system. The L3 cache can theoretically complete an aggregate of four write hits per chip every two cycles. However, banking conflicts reduce this number in most cases.

Optimization tip: Random access can divide the write sustained bandwidth of the L3 cache by a factor of three on Compute Nodes and more on I/O Nodes.

5.2.4 Double data RAM

The theoretical memory bandwidth on a Blue Gene/P node to transfer a 128-byte line from the external DDR to the L3 cache is 16 cycles. Nevertheless, this bandwidth can only be sustained with sequential access. Random access can reduce bandwidth significantly.

Table 5-2 illustrates latency and bandwidth estimates for the Blue Gene/P system.

Table 5-2 Latency and bandwidth estimates

	Latency[a]	**Sustained bandwidth (bytes per cycle)**[b, c]
		Sequential access
L1	3	8
L2	11	4.6
L3	50	4.6
External DDR (single processor)	104	40
External DDR (dual processor)		
External DDR (triple processor)		
External DDR (quad processor)		3.7

a. This corresponds to integer load latency. Floating-point latency is one cycle higher.
b. This is the maximum sustainable bandwidth for linear sequential access.
c. Random access bandwidth is dependent on the access width and overlap access, respectively.

5.3 Memory protection

The IBM PowerPC 450 processor has limited flexibility with regard to supported translation look-aside buffer (TLB) sizes and alignments. There is also a small number of TLB slots per processor. These limitations create situations where the dual goal of both static TLBs and memory protection is difficult to achieve with access to the entire memory space. This depends on the node's memory configuration, process model, and size of the applications sections.

On the Blue Gene/P system, the Compute Node Kernel (CNK) reads only sections from the application. This prevents an application from accidentally corrupting its text (that is, its code) section due to an errant memory write. Additionally, the CNK prevents an application from corrupting the CNK text segments or any kernel data structures.

The CNK can protect the active thread's stack using the data-address-compare debug registers in the PowerPC 450 processor. You can use this mechanism for stack protection without incurring TLB miss penalties. For the main thread, this protection is just above the maximum mmap() address. For a spawned thread, this protection is at the lower bound of the thread's stack. This protection is not available when the debugger is used to set a hardware watchpoint.

The CNK is strict in terms of TLB setup, for example, the CNK does not create a 256 MB TLB that covers only 128 MB of real memory. By precisely creating the TLB map, any user-level page faults (also known as *segfaults*) are immediately caught.

In the default mode of operation of the Blue Gene/P system, which is SMP node mode, each physical Compute Node executes a single task per node with a maximum of four threads. The Blue Gene/P system software treats those four core threads in a Compute Node symmetrically. Figure 5-1 illustrates how memory is accessed in SMP node mode. The user space is divided into user space “read, execute” and user space “read/write, execute”. The latter corresponds to global variables, stack, and heap. In this mode, the four threads have access to the global variables, stack, and heap.

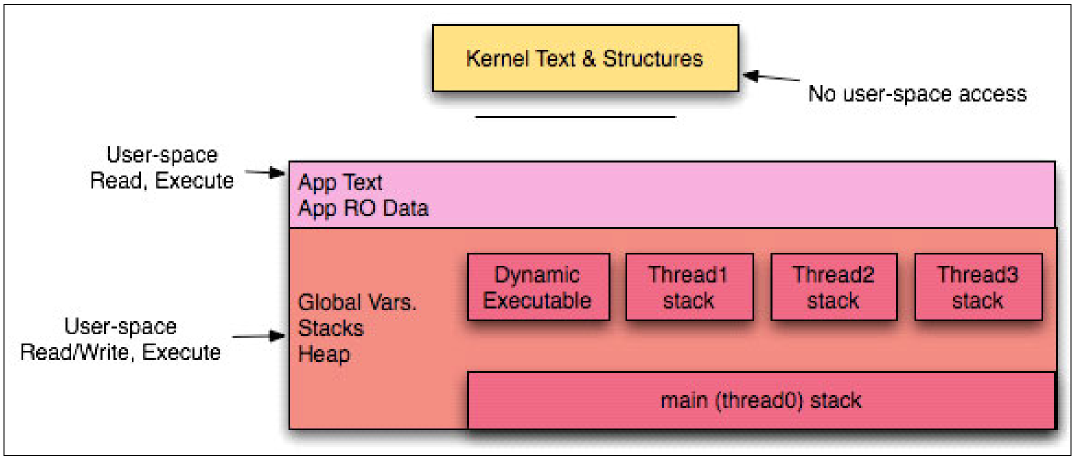


Figure 5-1 Memory access protection in SMP Node mode

Figure 5-2 shows how memory is accessed in Virtual Node Mode. In this mode, the four core threads of a Compute Node act as different processes. The CNK reads only sections of an application from local memory. No user access occurs between processes in the same node. User space is divided into user-space “read, execute” and user-space “read/write, execute”. The latter corresponds to global variables, stack, and heap. These two sections are designed to avoid data corruption.

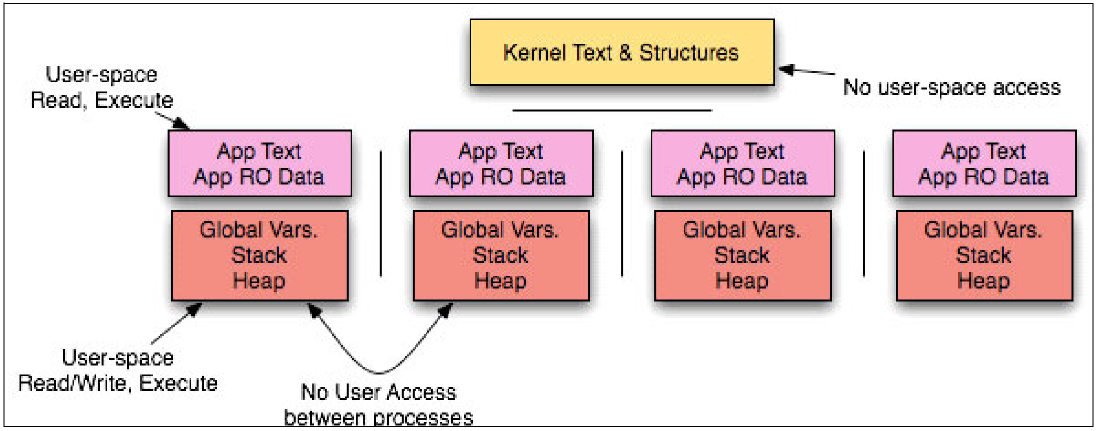


Figure 5-2 Memory access protections in Virtual Node Mode

Each task in Dual mode gets half the memory and cores so it can run two threads per task. Figure 5-3 shows that no user access occurs between the two processes. Although a layer of shared-memory per node and the user-space “read, execute” is common to the two tasks, the two user-spaces “read/write, execute” are local to each process.

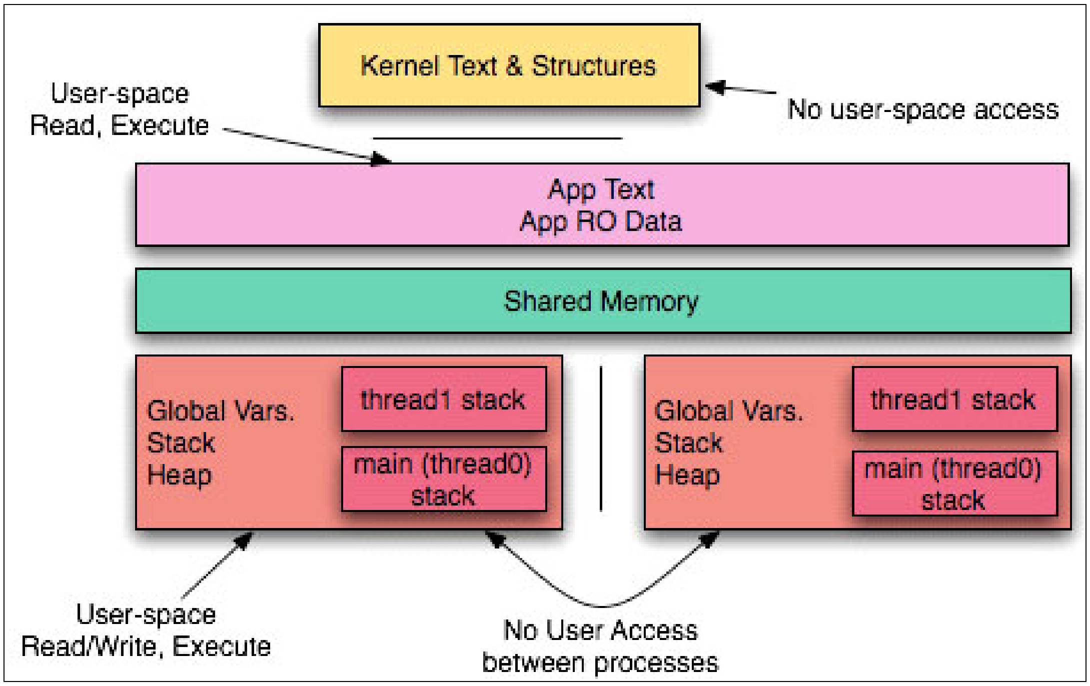


Figure 5-3 Memory access protections in Dual mode

5.4 Persistent memory

Persistent memory is process memory that retains its contents from job to job. To allocate persistent memory, the environment variable BG_PERSISTMEMSIZE=X must be specified, where X is the number of megabytes to be allocated for use as persistent memory. In order for the persistent memory to be maintained across jobs, all job submissions must specify the same value for BG_PERSISTMEMSIZE. The contents of persistent memory can be re-initialized during job startup either by changing the value of BG_PERSISTMEMSIZE or by specifying the environment variable BG_PERSISTMEMRESET=1. A new kernel function was added to support persistent memory, the persist_open() function.

6

System calls

System calls provide an interface between an application and the kernel. In this chapter, we provide information about the service points through which applications running on the Compute Node request services from the Compute Node Kernel (CNK). This set of entry points into the CNK is referred to as *system calls* (*syscall*). System calls on the Blue Gene/P system have substantially changed from system calls on the Blue Gene/L system. In this chapter, we describe system calls that are defined on the Blue Gene/P system.

We cover the following topics in this chapter:

- Compute Node Kernel
- Supported and unsupported system calls
- System programming interfaces
- Socket support
- Signal support

In general, the two types of system calls are:

- Local system calls
- Function-shipped system calls

Local system calls are handled by the CNK only and provide Blue Gene/P-specific functionality. The following examples are of standard, local system calls:

- brk()
- mmap()
- clone()

Alternatively, function-shipped system calls are forwarded by the CNK over the collective network to the control and I/O daemon (CIOD). The CIOD then executes those system calls on the I/O Node and replies to the CNK with the resultant data. Examples of function-shipped system calls are functions that manipulate files and socket calls.

6.1 Introduction to the Compute Node Kernel

The role of the kernel on the Compute Node is to create an environment for the execution of a user process that is “Linux-like.” It is not a full Linux kernel implementation, but rather implements a subset of POSIX functionality.

The Compute Node Kernel (CNK) is a single-process operating system. It is designed to provide the services needed by applications that are expected to run on the Blue Gene/P system, but not services for all applications. The CNK is not intended to run system administration functions from the Compute Node.

To achieve the best reliability, a small and simple kernel is a design goal. This enables a simpler checkpoint function. See Chapter 10, “Checkpoint and restart support for applications” on page 169.

Compute node application user: The Compute Node application never runs as the root user. In fact, it runs as the same user (uid) and group (gid) under which the job was submitted.

6.2 System calls

The CNK system calls are divided into the following categories:

- File I/O
- Directory operations
- Time
- Process information
- Signals
- Miscellaneous
- Sockets
- Compute Node Kernel (CNK)

6.2.1 Return codes

As is true for return codes on a standard Linux system, a return code of zero from a system call indicates success. A value of negative one (-1) indicates a failure. In this case, errno contains further information about exactly what caused the problem.

6.2.2 Supported system calls

Table 6-1 lists all the function prototypes for system calls by category that are supported on the Blue Gene/P system.

Table 6-1 Supported system calls

Function prototype	Category	Header required	Description and type
`int access(const char *path, int mode);`	File I/O	<unistd.h>	Determines the accessibility of a file; function-shipped to CIOD; mode: R_OK, X_OK, F_OK; returns 0 if OK or -1 on error.
`int chmod(const char *path, mode_t mode);`	File I/O	<sys/types.h> <sys>/<stat.h>	Changes the access permissions on an already open file; function-shipped to CIOD; mode: S_ISUID, S_ISGID, S_ISVTX, S_IRWXU, S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXG, S_IRGRP, S_IWGRP, S_IXGRP, S_IRWXO, S_IROTH, S_IWOTH, and S_IXOTH; returns 0 if OK or -1 on error.
`int chown(const char *path, uid_t owner, gid_t group);`	File I/O	<sys/types.h> <sys>/<stat.h>	Changes the owner and group of a file; function-shipped to CIOD.
`int close(int fd);`	File I/O	<unistd.h>	Closes a file descriptor; function-shipped to CIOD; returns 0 if OK or -1 on error.
`int dup(int fd);`	File I/O	<unistd.h>	Duplicates an open descriptor; function-shipped to CIOD; returns new file descriptor if OK or -1 on error.
`int dup2(int fd, int fd2);`	File I/O	<unistd.h>	Duplicates an open descriptor; function-shipped to CIOD; returns new file descriptor if OK or -1 on error.
`int fchmod(int fd, mode_t mode);`	File I/O	<sys/types.h> <sys>/<stat.h>	Changes the mode of a file; function-shipped to CIOD; returns 0 if OK or -1 on error.
`int fchown(int fd, uid_t owner, gid_t group);`	File I/O	<sys/types.h> <unistd.h>	Changes the owner and group of a file; function-shipped to CIOD; returns 0 if OK or -1 on error.
`int fcntl(int fd, int cmd, int arg);`	File I/O	<sys/types.h> <unistd.h> <fcntl.h>	Manipulates a file descriptor; function-shipped to CIOD; supported commands are F_GETFL, F_DUPFD, F_GETLK, F_SETLK, F_SETLKW, F_GETLK64, F_SETLK64, and F_SETLKW64.
`int fstat(int fd, struct stat *buf);`	File I/O	<sys/types.h> <sys>/<stat.h>	Gets the file status; function-shipped to CIOD; returns 0 if OK or -1 on error.
`int stat64(const char *path, struct stat64 *buf);`	File I/O	<sys/types.h> <sys>/<stat.h>	Gets the file status.
`int statfs(const char *path, struct statfs *buf);`	File I/O	<sys/vfs.h>	Gets file system statistics.
`long fstatfs64 (unsigned int fd, size_t sz, struct statfs64 *buf);`	File I/O	<sys/vfs.h>	Gets file system statistics.

Function prototype	Category	Header required	Description and type
`int fsync(int fd);`	File I/O	<unistd.h>	Synchronizes changes to a file; returns 0 if OK or -1 on error.
`int ftruncate(int fd, off_t length);`	File I/O	<sys/types.h> <unistd.h>	Truncates a file to a specified length; returns 0 if OK or -1 on error.
`int ftruncate64(int fildes, off64_t length);`	File I/O	<unistd.h>	Truncates a file to a specified length for files larger than 2 GB; returns 0 if OK or -1 on error.
`int lchown(const char *path, uid_t owner, gid_t group);`	File I/O	<sys/types.h> <unistd.h>	Changes the owner and group of a symbolic link; function-shipped to CIOD; returns 0 if OK or -1 on error.
`int link(const char *existingpath, const char *newpath);`	File I/O	<unistd.h>	Links to a file; function-shipped to CIOD; returns 0 if OK or -1 on error.
`off_t lseek(int fd, off_t offset, int whence);`	File I/O	<sys/types.h> <unistd.h>	Moves the read/write file offset; function-shipped to CIOD; returns 0 if OK or -1 on error.
`int _llseek(unsigned int fd, unsigned long offset_high, unsigned long offset_low, loff_t *result, unsigned int whence);`	File I/O	<unistd.h> <sys/types.h> <linux/unistd.h> <errno.h>	Moves the read/write file offset.
`int lstat(const char *path, struct stat *buf);`	File I/O	<sys/types.h> <sys>/<stat.h>	Gets the symbolic link status; function-shipped to CIOD; returns 0 if OK or -1 on error.
`int lstat64(const char *path, struct stat64 *buf);`	File I/O	<sys/types.h> <sys/stat.h>	Gets the symbolic link status; determines the size of a file larger than 2 GB.
`int open(const char *path, int oflag, mode_t mode);`	File I/O	<sys/types.h> <sys>/<stat.h> <fcntl.h>	Opens a file; function-shipped to CIOD; oflag: O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT, O_EXCL, O_TRUNC, O_NOCTTY, O_NONBLOCK, O_SYNC, mode: S_IRWXU, S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXG, S_IRGRP, S_IWGRP, S_IXGRP, S_IRWXO, S_IROTH, S_IWOTH, and S_IXOTH; returns file descriptor if OK or -1 on error.
`ssize_t pread(int fd, void *buf, size_t nbytes, off64_t offset);`	File I/O	<unistd.h>	Reads from a file at offset; function-shipped to CIOD; returns number of bytes read if OK, 0 if end of file, or -1 on error. Introduced in V1R3M0.
`ssize_t pwrite(int fd, const void *buf, size_t nbytes, off64_t offset);`	File I/O	<unistd.h>	Writes to a file at offset; function-shipped to CIOD; returns number of bytes written if OK or -1 on error. Introduced in V1R3M0.
`ssize_t read(int fd, void *buf, size_t nbytes);`	File I/O	<unistd.h>	Reads from a file; function-shipped to CIOD; returns number of bytes read if OK, 0 if end of file, or -1 on error.
`int readlink(const char *path, char *buf, int bufsize);`	File I/O	<unistd.h>	Reads the contents of a symbolic link; function-shipped to CIOD; returns number of bytes read if OK or -1 on error.

Function prototype	Category	Header required	Description and type
ssize_t readv(int fd, const struct iovec iov[], int iovcnt)	File I/O	<sys/types.h> <sys/uio.h>	Reads a vector, function-shipped to CIOD; returns number of bytes read if OK or -1 on error.
int rename(const char *oldname, const char *newname);	File I/O	<stdio.h>	Renames a file; function-shipped to CIOD; returns 0 if OK or -1 on error.
int stat(const char *path, struct stat *buf);	File I/O	<sys/types.h> <sys/stat.h>	Gets the file status; function-shipped to CIOD; returns 0 if OK or -1 on error.
int stat64(const char *path, struct stat64 *buf);	File I/O	<sys/types.h> <sys/stat.h>	Gets the file status.
int statfs (char *path, struct statfs *buf);	File I/O	<sys/statfs.h>	Gets file system statistics.
long statfs64 (const char *path, size_t sz, struct statfs64 *buf);	File I/O	<sys/statfs.h>	Gets file system statistics.
int symlink(const char *actualpath, const char *sympath);	File I/O	<unistd.h>	Makes a symbolic link to a file; function-shipped to CIOD; returns 0 if OK or -1 on error.
int truncate(const char *path, off_t length);	File I/O	<sys/types.h> <unistd.h>	Truncates a file to a specified length; function-shipped to CIOD; returns 0 if OK or -1 on error.
truncate64(const char *path, off_t length);	File I/O	<unistd.h> <sys/types.h>	Truncates a file to a specified length.
mode_t umask(mode_t cmask);	File I/O	<sys/types.h> <sys/stat.h>	Sets and gets the file mode creation mask; function-shipped to CIOD; returns the previous file mode creation mask.
int unlink(const char *path);	File I/O	<unistd.h>	Removes a directory entry; function-shipped to CIOD; returns 0 if OK or -1 on error.
int utime(const char *path, const struct utimbuf *times);	File I/O	<sys/types.h> <utime.h>	Sets file access and modification times; function-shipped to CIOD; returns 0 if OK or -1 on error.
ssize_t write(int fd, const void *buff, size_t nbytes);	File I/O	<unistd.h>	Writes to a file; function-shipped to CIOD; returns the number of bytes written if OK or -1 on error.
ssize_t writev(int fd, const struct iovec iov[], int iovcntl);	File I/O	<sys/types.h> <sys/uio.h>	Writes a vector; function-shipped to CIOD; returns the number of bytes written if OK or -1 on error.
int chdir(const char *path);	Directory	<unistd.h>	Changes the working directory; function-shipped to CIOD; returns 0 if OK or -1 on error.
char *getcwd(char *buf, size_t size);	Directory	<unistd.h>	Gets the path name of the current working directory; function-shipped to CIOD; returns buf if OK or NULL on error.
int getdents(int fildes, char **buf, unsigned nbyte);	Directory	<sys/types.h>	Gets the directory entries in a file system; function-shipped to CIOD; returns 0 if OK or -1 on error.

Function prototype	Category	Header required	Description and type
`int getdents64(unsigned int fd, struct dirent *dirp, unsigned int count);`	Directory	<sys/dirent.h>	Gets the directory entries in a file system.
`int mkdir(const char *path, mode_t mode);`	Directory	<sys/types.h> <sys/stat.h>	Makes a directory; function-shipped to CIOD; mode S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, and S_IXOTH; returns 0 if OK or -1 on error.
`int rmdir(const char *path);`	Directory	<unistd.h>	Removes a directory; returns 0 if OK or -1 on error.
`int getitimer(int which, struct itimerval *value);`	Time	<sys/time.h>	Gets the value of the interval timer; local system call; returns 0 if OK or -1 on error.
`int gettimeofday(struct timeval *restrict tp, void *restrict tzp);`	Time	<sys/time.h>	Gets the date and time; local system call; returns 0 if OK or NULL on error.
`int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue);`	Time	<sys/time.h>	Sets the value of an interval timer; only the following operations are supported: ▸ ITIMER_PROF ▸ ITIMER_REAL **Note**: An application can only set one active timer at a time.
`time_t time(time_t *calptr);`	Time	<time.h>	Gets the time; local system call; returns the value of time if OK or -1 on error.
`gid_t getgid(void);`	Process information	<unistd.h>	Gets the real group ID.
`pid_t getpid(void);`	Process information	<unistd.h>	Gets the process ID. The value is the MPI rank of the node, meaning that 0 is a valid value.
`int getrlimit(int resource, struct rlimit *rlp)`	Process information	<sys/resource.h>	Gets information about resource limits.
`int getrusage(int who, struct rusage *r_usage);`	Process information	<sys/resource.h>	Gets information about resource utilization. All time reported is attributed to the user application. so the reported system time is always zero.
`uid_t getuid(void);`	Process information	<unistd.h>	Gets the real user ID.
`int setrlimit(int resource, const struct rlimit *rlp);`	Process information	<sys/resource.h>	Sets resource limits. Only RLIMIT_CORE can be set.
`clock_t times(struct tms *buf);`	Process information	<sys/times.h>	Gets the process times. All time reported is attributed to the user application, so the reported system time is always zero.
`int brk(void *end_data_segment);`	Miscellaneous	<unistd.h>	Changes the data segment size.
`void exit(int status)`	Miscellaneous	<stdlib.h>	Terminates a process.
`int sched_yield(void);`	Miscellaneous	<sched.h>	Force the running thread to relinquish the processor.

Function prototype	Category	Header required	Description and type
`int uname(struct utsname *buf);`	Miscellaneous	<sys/utsname.h>	Gets the name of the current system and other information, for example, version and release.

6.2.3 Other system calls

Although many system calls are unsupported, you must be aware of the following unsupported calls:

- The Blue Gene/P system does not support the use of the system() function. Therefore, for example, you cannot use something, such as the `system('chmod -w file')` call. Although, system() is not a system call, it uses `fork()` and exec() via **`glibc`**. Both fork() and exec() are currently not implemented.
- The Blue Gene/P system does not provide the same support for gethostname() and getlogin() as Linux provides.
- Calls to usleep() are not supported.

See 6.6, “Unsupported system calls” on page 60, for a complete list of unsupported system calls.

6.3 System programming interfaces

Low-level access to IBM Blue Gene/P-specific interfaces, such as direct memory access (DMA), is provided by the system programming interfaces (SPIs). These interfaces provide a consistent interface for Linux and Compute Node Kernel-based applications to access the hardware.

The following Blue Gene/P-specific interfaces *are* included in the SPI:

- Collective network
- Torus network
- Direct memory access
- Global interrupts
- Performance counters
- Lockbox

The following items *are not* included in the SPI:

- L2
- Snoop
- L3
- DDR hardware initialization
- serdes
- Environmental monitor

This hardware is set up by either the bootloader or Common Node Services. The L1 interfaces, such as TLB miss handlers, are typically extremely operating system specific, and therefore an SPI is not defined. TOMAL and XEMAC are present in the Linux 10 Gb Ethernet device driver (and therefore open source), but there are no plans for an explicit SPI.

6.4 Socket support

The CNK provides socket support via the standard Linux socketcall() system call. The socketcall() is a kernel entry point for the socket system calls. It determines which socket function to call and points to a block that contains the actual parameters, which are passed through to the appropriate call. The CNK function-ships the socketcall() parameters to the CIOD, which then performs the requested operation. The CIOD is a user-level process that controls and services applications in the Compute Node and interacts with the Midplane Management Control System (MMCS).

This socket support allows the creation of both outbound and inbound socket connections using standard Linux APIs, for example, an outbound socket can be created by calling socket(), followed by connect(). An inbound socket can be created by calling socket() followed by bind(), listen(), and accept().

Communication through the socket is provided via the **`glibc`** send(), recv(), and select() function calls. These function calls invoke the socketcall() system call with different parameters. Table 6-2 summarizes the list of Linux 2.4 socket system calls.

Table 6-2 Supported socket calls

Function prototype	Category	Header required	Description and type
`int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`	Sockets	<sys/types.h> <sys/socket.h>	Extracts the connection request on the queue of pending connections; creates a new connected socket; returns a file descriptor if OK or -1 on error.
`int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);`	Sockets	<sys/types.h> <sys/socket.h>	Assigns a local address; returns 0 if OK or -1 on error.
`int connect(int socket, const struct sockaddr *address, socklen_t address_len);`	Sockets	<sys/types.h> <sys/socket.h>	Connects a socket; returns 0 if OK or -1 on error.
`int getpeername(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len);`	Sockets	<sys/socket.h>	Gets the name of the peer socket; returns 0 if OK or -1 on error.
`int getsockname(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len);`	Sockets	<sys/socket.h>	Retrieves the locally bound socket name; stores the address in sockaddr; and stores its length in the address_len argument; returns 0 if OK or -1 on error.
`int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);`	Sockets	<sys/types.h> <sys/socket.h>	Manipulates options that are associated with a socket; returns 0 if OK or -1 on error.
`int listen(int sockfd, int backlog);`	Sockets	<sys/socket.h>	Accepts connections; returns 0 if OK or -1 on error.
`ssize_t recv(int s, void *buf, size_t len, int flags);`	Sockets	<sys/types.h> <sys/socket.h>	Receives a message only from a connected socket; returns 0 if OK or -1 on error.
`ssize_t recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);`	Sockets	<sys/types.h> <sys/socket.h>	Receives a message from a socket regardless of whether it is connected; returns 0 if OK or -1 on error.

Function prototype	Category	Header required	Description and type
`ssize_t recvmsg(int s, struct msghdr *msg, int flags);`	Sockets	<sys/types.h> <sys/socket.h>	Receives a message from a socket regardless of whether it is connected; returns 0 if OK or -1 on error.
`ssize_t send(int socket, const void *buffer, size_t length, int flags);`	Sockets	<sys/types.h> <sys/sockets.h>	Sends a message only to a connected socket; returns 0 if OK or -1 on error.
`ssize_t sendto(int socket, const void *message, size_t length, int flags, const struct sockaddr *dest_addr, socklen_t dest_len);`	Sockets	<sys/types.h> <sys/socket.h>	Sends a message on a socket; returns 0 if OK or -1 on error.
`ssize_t sendmsg(int s, const struct msghdr *msg, int flags);`	Sockets	<sys/types.h> <sys/socket.h>	Sends a message on a socket; returns 0 if OK or -1 on error.
`int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);`	Sockets	<sys/types.h> <sys/socket.h>	Manipulates options that are associated with a socket; returns 0 if OK or -1 on error.
`int shutdown(int s, int how);`	Sockets	<sys/socket.h>	Causes all or part of a connection on the socket to shut down; returns 0 if OK or -1 on error.
`int socket(int domain, int type, int protocol);`	Sockets	<sys/types.h> <sys/socket.h>	Opens a socket; returns a file descriptor if OK or -1 on error.
`int socketpair(int d, int type, int protocol, int sv[2]);`	Sockets	<sys/types.h> <sys/socket.h>	Creates an unnamed pair of connected sockets; returns 0 if OK or -1 on error.

6.5 Signal support

The Compute Node Kernel provides ANSI-C signal support via the standard Linux system calls signal() and kill(). Additionally, signals can be delivered externally by using **`mpirun`** or for HTC using **`submit`**. Table 6-3 summarizes the supported signals.

Table 6-3 Supported signals

Function prototype	Category	Header required	Description and type
`int kill(pid_t pid, int sig);`	Signals	<sys/types.h> <signal.h>	Sends a signal. A signal can be sent only to the same process.
`int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`	Signals	<signal.h>	Manages signals. The only flags supported are SA_RESETHAND and SA_NODEFER.

`typedef void (*sighandler_t)(int)`
`sighandler_t signal(int signum, sighandler_t handler);`
	Signals	<signal.h>	Manages signals.

`typedef void (*sighandler_t)(int);`
`sighandler_t signal(int signum, sighandler_t handler);`
	Signals	<signal.h>	Returns from a signal handler.

6.6 Unsupported system calls

The role of the kernel on the Compute Node is to create an environment for the execution of a user process that is “Linux like.” It is not a full Linux kernel implementation, but rather implements a subset of the POSIX functionality. The following list indicates the system calls that are not supported:

- acct
- adjtimex
- afs_syscall
- bdflush
- break
- capget
- capset
- chroot
- clock_getres
- clock_gettime
- clock_nanosleep
- clock_settime
- create_module
- delete_module
- epoll_create
- epoll_ctl
- epoll_wait
- execve
- fadvise64
- fadvise64_64
- fchdir
- fdatasync
- fgetxattr
- flistxattr
- flock
- fork
- fremovexattr
- fsetxattr
- ftime
- get_kernel_syms
- ioperm
- iopl
- ipc
- kexec_load
- lgetxattr
- listxattr
- llistxattr
- lock
- lookup_dcookie
- lremovexattr
- lsetxattr
- mincore
- mknod
- modify_lft
- mount
- mpxmq_getsetattr
- mq_notify
- mq_open
- mq_timedreceive
- mq_timedsend
- mq_unlink
- multiplexer
- nfsservctl
- nice
- oldfstat
- oldlstat
- oldolduname
- olduname
- oldstat
- pciconfig_iobase
- removexattr
- rtas
- rts_device_map
- rts_dma
- sched_get_priority_max
- sched_get_priority_min
- sched_getaffinity
- sched_getparam
- sched_getscheduler
- sched_rr_get_interval
- sched_setaffinity
- sched_setparam
- sched_setscheduler
- select
- sendfile
- sendfile64
- setdomainname
- setgroups
- sethostname
- setpriority
- settimeofday
- setxattr
- stime
- stty
- swapcontext
- swapoff
- swapon
- sync
- sys_debug_setcontext
- sysfs

- getgroups
- getpgrp
- getpmsg
- getppid
- getpriority
- gettid
- getxattr
- gtty
- idle
- init_module
- io_cancel
- io_destroy
- io_getevents
- io_setup
- io_submit
- pciconfig_read
- pciconfig_write
- personality
- pipe
- pivot_root
- prof
- profil
- ptrace
- putpmsg
- query_module
- quotactl
- readahead
- readdir
- reboot
- remap_file_pages
- syslog
- timer_create
- timer_delete
- timer_getoverrun
- timer_gettime
- timer_settime
- tuxcall
- umount
- umount2
- uselib
- ustat
- utimes
- vfork
- vhangup
- vm86

You can find additional information about these system calls on the syscalls(2) - Linux man page on the Web at:

http://linux.die.net/man/2/syscalls

Part 3

Applications environment

In this part, we provide an overview of some of the software that forms part of the applications environment. Throughout this book, we consider the applications environment as the collection of programs that are required to develop applications.

This part includes the following chapters:

- Chapter 7, “Parallel paradigms” on page 65
- Chapter 8, “Developing applications with IBM XL compilers” on page 97
- Chapter 9, “Running and debugging applications” on page 139
- Chapter 10, “Checkpoint and restart support for applications” on page 169
- Chapter 11, “mpirun” on page 177
- Chapter 12, “High-Throughput Computing (HTC) paradigm” on page 201

7

Parallel paradigms

In this chapter, we discuss the parallel paradigms that are offered on the Blue Gene/P system. One such paradigm is the Message Passing Interface (MPI),[22] for a distributed-memory architecture, and OpenMP,[23] for shared-memory architectures. We refer to this paradigm as *High-Performance Computing* (*HPC*). Blue Gene/P also offers a paradigm where applications do not require communication between tasks and each node is running a different instance of the application. We refer to this paradigm as *High-Throughput Computing* (*HTC*). This topic is discussed in Chapter 14.

In this chapter, we address the following topics:

- Programming model
- IBM Blue Gene/P MPI implementation
- Blue Gene/P MPI extensions
- MPI functions
- Compiling MPI programs on Blue Gene/P
- MPI communications performance
- OpenMP

7.1 Programming model

The Blue Gene/P system has a distributed memory system and uses explicit message passing to communicate between tasks that are running on different nodes. It also has shared memory on each node; OpenMP and thread parallelism are supported as well.

MPI is the supported message-passing standard. It is the industry standard for message passing. For further information about MPI, refer to the Message Passing Interface Forum site on the Web at the following address:

http://www.mpi-forum.org/

The Blue Gene/P MPI implementation uses the Deep Computing Messaging Framework (DCMF) as a low-level messaging interface. The Blue Gene/P DCMF implementation directly accesses the Blue Gene/P hardware through the DMA SPI interface. The MPI, DCMF, and SPI interfaces are public, supported interfaces on Blue Gene/P, and all can be used by an application to perform communication operations. For further information about DCMF and the DMA SPI, refer to the open source documentation at the following locations:

http://dcmf.anl-external.org/wiki
http://wiki.bg.anl-external.org/index.php/Base
http://wiki.bg.anl-external.org/index.php/Runtime

Other programming paradigms have been ported to Blue Gene/P using one or more of the supported software interfaces as illustrated in Figure 7-1. The respective open source communities provide support for using these alternative paradigms, as described in the rest of this section.

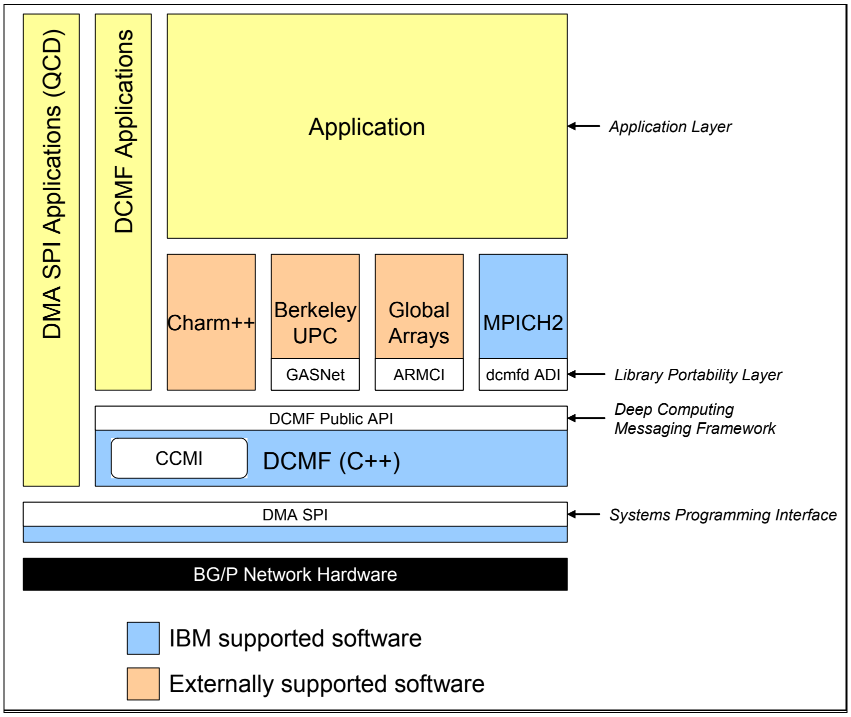


Figure 7-1 Software stack

Aggregate Remote Memory Copy Interface

Aggregate Remote Memory Copy Interface (ARMCI) is an open source project developed and maintained by the Pacific Northwest National Laboratory. The purpose of the ARMCI library is to provide general-purpose, efficient, and widely portable remote memory access (RMA) operations (one-sided communication) optimized for contiguous and noncontiguous (strided, scatter/gather, I/O vector) data transfers. To obtain further information about ARMCI, refer to the following Web site:

http://www.emsl.pnl.gov/docs/parsoft/armci/

You also can obtain information about ARMCI by sending an e-mail to hpctools@emsl.pnl.gov.

Global Arrays

The Global Arrays (GA) toolkit is an open source project developed and maintained by the Pacific Northwest National Laboratory. The toolkit provides an efficient and portable "shared-memory" programming interface for distributed-memory computers. Each process in a MIMD parallel program can asynchronously access logical blocks of physically distributed dense multidimensional arrays without need for explicit cooperation by other processes. Unlike other shared-memory environments, the GA model exposes to the programmer the non-uniform memory access (NUMA) characteristics of the high-performance computers and acknowledges that access to a remote portion of the shared data is slower than access to the local portion. The locality information for the shared data is available, and direct access to the local portions of shared data is provided. For information about the GA toolkit, refer to the following Web site:

http://www.emsl.pnl.gov/docs/global/

You also can obtain information about the GA toolkit by sending an e-mail to hpctools@pnl.gov.

Charm++

Charm++ is an open source project developed and maintained by the Parallel Programming Laboratory at the University of Illinois at Urbana-Champaign. Charm++ is an explicitly parallel language based on C++ with a runtime library for supporting parallel computation called the Charm kernel. It provides a clear separation between sequential and parallel objects. The execution model of Charm++ is message driven, thus helping one write programs that are latency tolerant. Charm++ supports dynamic load balancing while creating new work as well as periodically, based on object migration. Several dynamic load balancing strategies are provided. Charm++ supports both irregular as well as regular, data-parallel applications. It is based on the Converse interoperable runtime system for parallel programming. You can access information from the following Parallel Programming Laboratory Web site:

http://charm.cs.uiuc.edu/

You also can access information about the Parallel Programming Laboratory by sending an e-mail to ppl@cs.uiuc.edu.

Berkeley Unified Parallel C

Berkeley UPC is an open source project developed and maintained by Lawrence Berkeley National Laboratory and the University of California, Berkeley. UPC is an extension of the C programming language designed for High-Performance Computing on large-scale parallel machines.The language provides a uniform programming model for both shared and distributed memory hardware. The programmer is presented with a single shared, partitioned address space, where variables can be directly read and written by any processor, but each variable is physically associated with a single processor. UPC uses a Single Program Multiple

Data (SPMD) model of computation in which the amount of parallelism is fixed at program startup time, typically with a single thread of execution per processor. You can access more information at the following Web site:

http://upc.lbl.gov/

You also can access more information by sending an e-mail to upc-users@lbl.gov.

Global-Address Space Networking

Global-Address Space Networking (GASNet) is an open source project developed and maintained by Lawrence Berkeley National Laboratory and the University of California, Berkeley. GASNet is a language-independent, low-level networking layer that provides network-independent, high-performance communication primitives tailored for implementing parallel global address space SPMD languages, such as UPC, Titanium, and Co-Array Fortran. The interface is primarily intended as a compilation target and for use by runtime library writers (as opposed to users), and the primary goals are high-performance, interface portability, and expressiveness. You can access more information at the following Web site:

http://gasnet.cs.berkeley.edu/

7.2 Blue Gene/P MPI implementation

The current MPI implementation on the Blue Gene/P system supports the MPI-2.1 standard. The only exception is the process creation and management functions. The MPI-2.1 standard is available from the following Web site:

http://www.mpi-forum.org/mpi2_1/index.htm

When starting applications on the Blue Gene/P system, you must consider that the microkernel running on the Compute Nodes does not provide any mechanism for a command interpreter or shell. Only the executables can be started. Shell scripts are not supported. Therefore, if your application consists of a number of shell scripts that control its workflow, the workflow must be adapted. If you start your application with the **mpirun** command, you cannot start the main shell script with this command. Instead, you must run the scripts on the front end node and only call **mpirun** at the innermost shell script level where the main application binary is called.

The MPI implementation on the Blue Gene/P system is derived from the MPICH2 implementation of the Mathematics and Computer Science Division (MCS) at Argonne National Laboratory. For additional information, refer to the following MPICH2 Web site:

http://www.mcs.anl.gov/research/projects/mpich2/

To support the Blue Gene/P hardware, the following additions and modifications have been made to the MPICH2 software architecture:

- A Blue Gene/P driver has been added that implements the MPICH2 abstract device interface (ADI).
- Optimized versions of the Cartesian functions exist (MPI_Dims_create(), MPI_Cart_create(), MPI_Cart_map()).
- MPIX functions create hardware-specific MPI extensions.

From the application programmer's view, the most important aspect of these changes is that the collective operations can use different networks under different circumstances. In 7.2.1, "High-performance network for efficient parallel execution" on page 69, we briefly summarize the different networks on the Blue Gene/P system and network routing.

In sections 7.2.2, “Forcing MPI to allocate too much memory” on page 71 through 7.2.7, “Buffer alignment sensitivity” on page 73, we discuss several sample MPI codes to explain some of the implementation-dependent behaviors of the MPI library. Section 7.3.3, “Self Tuned Adaptive Routines for MPI” on page 79 discusses an automatic optimization technique available on the Blue Gene/P MPI implementation.

7.2.1 High-performance network for efficient parallel execution

The Blue Gene/P system provides three different communication networks for hardware acceleration for certain collective operations.

Global interrupt network

The global interrupt network connects all compute nodes and provides a low latency barrier operation.

Collective network

The collective network connects all the Compute Nodes in the shape of a tree. Any node can be the tree root. The MPI implementation uses the collective network, which is more efficient than the torus network for collective communication on global communicators, such as MPI_COMM_WORLD.

Point-to-point network

All MPI point-to-point and subcommunicator communication operations are carried out through the torus network. The route from a sender to a receiver on a torus network has the following two possible paths:

- Deterministic routing

  Packets from a sender to a receiver go along the same path. One advantage of this path is that the packet order is always maintained without additional logic. However, this technique also creates network hot spots if several point-to-point communications occur at the same time and their deterministic routes cross on some node.

- Adaptive routing

  Different packets from the same sender to the same receiver can travel along different paths. The exact route is determined at run time depending on the current load. This technique generates a more balanced network load but introduces a latency penalty.

Selecting deterministic or adaptive routing depends on the protocol used for communication. The Blue Gene/P MPI implementation supports three different protocols:

- MPI short protocol

  The MPI short protocol is used for short messages (less than 224 bytes), which consist of a single packet. These messages are always deterministically routed. The latency for short messages is around 3.3 µs.

- MPI eager protocol

  The MPI eager protocol is used for medium-sized messages. It sends a message to the receiver without negotiating with the receiving side that the other end is ready to receive the message. This protocol also uses deterministic routes for its packets.

- MPI rendezvous protocol

  Large messages are sent using the MPI rendezvous protocol. In this case, an initial connection between the two partners is established. Only after that connection is established, does the receiver use direct memory access (DMA) to obtain the data from the sender. This protocol uses adaptive routing and is optimized for maximum bandwidth.

By default, MPI send operations use the rendezvous protocol, instead of the eager protocol, for messages larger than 1200 bytes. Naturally, the initial rendezvous handshake increases the latency.

There are two types of rendezvous protocols: *default* and *optimized*. The optimized rendezvous protocol generally has less latency than the default rendezvous protocol, but does not wait for a receive to be posted first. Therefore, unexpected messages can be received, consuming storage until the receives are issued. The default rendezvous protocol waits for a receive to be posted first. Therefore, no unexpected messages will be received. The optimized rendezvous protocol also avoids filling injection FIFOs which can cause delays while larger FIFOs are allocated. In general, the optimized rendezvous protocol should be used with smaller rendezvous messages, while the default rendezvous protocol should be used for larger rendezvous messages. By default, the default rendezvous protocol is used, and the optimized rendezvous protocol is disabled, since the default protocol is guaranteed to not run out of memory with unexpected messages. Enabling the optimized protocol for smaller rendezvous messages improves performance in some applications. Enabling the optimized rendezvous protocol is done by setting environment variables, as described below.

The Blue Gene/P MPI library supports a DCMF_EAGER environment variable (which can be set using **`mpirun`**) to set the message size (in bytes) above which the rendezvous protocol should be used. Consider the following guidelines:

- Decrease the rendezvous threshold if any of the following situations are true:
  - Many short messages are overloading the network.
  - Eager messages are creating artificial hot spots.
  - The program is not latency-sensitive.
- Increase the rendezvous threshold if any of the following situations are true:
  - Most communication is a nearest neighbor or at least close in Manhattan distance, where this distance is the shortest number of hops between a pair of nodes.
  - You mainly use relatively long messages.
  - You need better latency on medium-sized messages.

The DCMF_OPTRZV environment variable specifies the number of bytes on the low end of the rendezvous range where the optimized rendezvous protocol will be used. That is, the optimized rendezvous protocol will be used if `eager_limit <= message_size < (eager_limit + DCMF_OPTRZV)`, for example, if the eager limit (DCMF_EAGER) is 1200 (the default), and DCMF_OPTRZV is 1000, the eager protocol will be used for message sizes less than 1200 bytes, the optimized rendezvous protocol will be used for message sizes 1200 - 2199 bytes, and the default rendezvous protocol will be used for message sizes 2200 bytes or larger. The default DCMF_OPTRZV value is 0, meaning that the optimized rendezvous protocol is not used.

Several other environment variables can be used to customize MPI communications. Refer to Appendix D, “Environment variables” on page 339 for descriptions of these environment variables.

An efficient MPI application on Blue Gene/P observes the following guidelines:

- Overlap communication and computation using `MPI_Irecv` and `MPI_Isend`, which allow DMA to work in the background.

  DMA and the collective and GI networks: The collective and GI networks do not use DMA. In this case, operations cannot be completed in the background.

- Avoid load imbalance.

  This is important for all parallel systems. However, when scaling to the high numbers of tasks that are possible on the Blue Gene/P system, it is important to pay close attention to load balancing.

- Avoid buffered and synchronous sends; post receives in advance.

  The MPI standard defines several specialized communication modes in addition to the standard send function, MPI_Send(). Avoid the buffered send function, MPI_Bsend(), because it causes the MPI library to perform additional memory copies. Using the synchronous send function, MPI_Ssend(), is discouraged because it is a non-local operation that incurs an increased latency compared to the standard send without saving memory allocation.

- Avoid vector data and non-contiguous data types.

  While the MPI-derived data types can elegantly describe the layout of complex data structures, using these data types is generally detrimental to performance. Many MPI implementations, including the Blue Gene/P MPI implementation, pack (that is, memory-copy) such data objects before sending them. This packing of data objects is contrary to the original purpose of MPI-derived data types, namely to avoid such memory copies. Memory copies are particularly expensive on Blue Gene/P due to the relatively slow processor clock compared to the fast network hardware capabilities. Avoiding noncontiguous MPI data types, and memory copies in general, improves application performance.

7.2.2 Forcing MPI to allocate too much memory

Forcing MPI to allocate too much memory is relatively easy to do with basic code, for example, the snippets of legal MPI code shown in Example 7-1 and Example 7-2 run the risk of forcing the MPI support to allocate too much memory, resulting in failure, because it forces excessive buffering of messages.

Example 7-1 CPU1 MPI code that can cause excessive memory allocation

```
MPI_ISend(cpu2, tag1);
MPI_ISend(cpu2, tag2);
...
MPI_ISend(cpu2, tagn);
```

Example 7-2 CPU2 MPI code that can cause excessive memory allocation

```
MPI_Recv(cpu1, tagn);
MPI_Recv(cpu1, tagn-1);
...
MPI_Recv(cpu1, tag1);
```

Keep in mind the following points:

- The Blue Gene/P MPI rendezvous protocol does not allocate a temporary buffer to receive unexpected messages. This proper buffer allocation prevents most problems by drastically reducing the memory footprint of unexpected messages.
- The message queue is searched linearly to meet MPI matching requirements. If several messages are on the queue, the search can take longer.

You can accomplish the same goal and avoid memory allocation issues by recoding as shown in Example 7-3 and Example 7-4.

Example 7-3 CPU1 MPI code that can avoid excessive memory allocation

```
MPI_ISend(cpu2, tag1);
MPI_ISend(cpu2, tag2);
...
MPI_ISend(cpu2, tagn);
```

Example 7-4 CPU2 MPI code that can avoid excessive memory allocation

```
MPI_Recv(cpu1, tag1);
MPI_Recv(cpu1, tag2);
...
MPI_Recv(cpu1, tagn);
```

7.2.3 Not waiting for MPI_Test

According to the MPI standard, an application must either wait or continue testing until MPI_Test returns *true*. Not doing so causes small memory leaks, which can accumulate over time and cause a memory overrun. Example 7-5 shows the code and the problem.

Example 7-5 Potential memory overrun caused by not waiting for MPI_Test

```
req = MPI_Isend( ... );
MPI_Test (req);
... do something else; forget about req ...
```

Remember to use MPI_Wait or loop until MPI_Test returns *true*.

7.2.4 Flooding of messages

The code shown in Example 7-6, while legal, floods the network with messages. It can cause CPU 0 to run out of memory. Even though it can work, it is not scalable.

Example 7-6 Flood of messages resulting in a possible memory overrun

```
CPU 1 to n-1 code:
MPI_Send(cpu0);

CPU 0 code:
for (i=1; i<n; i++)
    MPI_Recv(cpu[i]);
```

7.2.5 Deadlock the system

The code shown in Example 7-7 is illegal according to the MPI standard. Each side does a blocking send to its communication partner before posting a receive for the message coming from the other partner.

Example 7-7 Deadlock code

```
TASK1 code:
MPI_Send(task2, tag1);
MPI_Recv(task2, tag2);
TASK2 code:
```

```
MPI_Send(task1, tag2);
MPI_Recv(task1, tag1);
```

In general, this code has a high probability of deadlocking the system. Obviously, you should not program this way. Make sure that your code conforms to the MPI specification. You can achieve this by either changing the order of sends and receives or by using non-blocking communication calls.

While you should not rely on the run-time system to correctly handle nonconforming MPI code, it is easier to debug such situations when you receive a run-time error message than to try and detect a deadlock and trace it back to its root cause.

7.2.6 Violating MPI buffer ownership rules

A number of problems can arise when the send/receive buffers that participate in asynchronous message-passing calls are accessed before it is legal to do so. All of the following examples are illegal, and therefore, you must avoid them.

The most obvious case is when you write to a send buffer before the MPI_Wait() for that request has completed as shown in Example 7-8.

Example 7-8 Write to a send buffer before MPI_Wait() has completed

```
req = MPI_Isend(buffer,&req);
buffer[0] = something;
MPI_Wait(req);
```

The code in Example 7-8 results in a race condition on any message-passing machine. Depending on the run-time factors that are outside the application's control, sometimes the old buffer[0] is sent and sometimes the new value is sent.

In the last example in this thread, a receive buffer is read before MPI_Wait() because the asynchronous receive request completed (see Example 7-9).

Example 7-9 Receive buffer before MPI_Wait() has completed

```
req = MPI_Irecv(buffer);
z = buffer[0];
MPI_Wait (req);
```

The code shown in Example 7-9 is also illegal. The contents of the receive buffer are not guaranteed until after MPI_Wait() is called.

7.2.7 Buffer alignment sensitivity

It is important to note that the MPI implementation on the Blue Gene/P system is sensitive to the alignment of the buffers that are being sent or received. Aligning buffers on 32-byte boundaries can improve performance. If the buffers are at least 16-bytes aligned, the messaging software can use internal math routines that are optimized for the double hummer architecture. Additionally, the L1 cache and DMA are optimized on 32-byte boundaries.

For buffers that are dynamically allocated (via malloc()), the following techniques can be used:

- Instead of using malloc(), use the following statement and specify 32 for the alignment parameter:

  ```
  int posix_memalign(void **memptr, size_t alignment, size_t size)
  ```

  This statement returns a 32-byte aligned pointer to the allocated memory. You can use free() to free the memory.

- Use malloc(), but request 32 bytes of more storage than required. Then round the returned address up to a 32-byte boundary as shown in Example 7-10.

Example 7-10 Request 32 bytes more storage than required

```
buffer_ptr_original = malloc(size + 32);
buffer_ptr          = (char*)( ( (unsigned)buffer_ptr_original + 32 ) &
0xFFFFFFE0 );
.
.
.
/* Use buffer_ptr on MPI operations */
.
.
.
free(buffer_ptr_original);
```

For buffers that are declared in static (global) storage, use **`__attribute__((aligned(32)))`** on the declaration as shown in Example 7-11.

Example 7-11 Buffers that are declared in static (global) storage

```
struct DataInfo
{
unsigned int iarray[256];
unsigned int count;
} data_info __attribute__ ((aligned ( 32)));
or
unsigned int data __attribute__ ((aligned ( 32)));
or
char data_array[512] __attribute__((aligned(32)));
```

For buffers that are declared in automatic (stack) storage, only up to a 16-byte alignment is possible. Therefore, use dynamically allocated aligned static (global) storage instead.

7.3 Blue Gene/P MPI extensions

This section describes extensions to the MPI library available on Blue Gene/P. It includes the following sections:

- Section 7.3.1, “Blue Gene/P communicators” on page 75 describes functions to create communicators that reflect the unique aspects of the Blue Gene/P hardware.
- Section 7.3.2, “Configuring MPI algorithms at run time” on page 77 describes functions to dynamically configure the algorithms used by the MPI collectives while the application is running.

- 7.3.3, “Self Tuned Adaptive Routines for MPI” on page 79 describes a way to automatically tune the collective routines used by an application.

7.3.1 Blue Gene/P communicators

Three new APIs make it easier to map nodes to specific hardware or processor set (pset) configurations. Application developers can use these functions, as explained in the following list, by including the mpix.h file:

- `int MPIX_Cart_comm_create (MPI_Comm *cart_comm);`

  This function creates a four-dimensional (4D) Cartesian communicator that mimics the exact hardware on which it is run. The X, Y, and Z dimensions match those of the partition hardware, while the T dimension has cardinality 1 in symmetrical multiprocessing (SMP) mode, cardinality 2 in Dual mode, and cardinality 4 in Virtual Node Mode. The communicator wrap-around links match the true mesh or torus nature of the partition. In addition, the coordinates of a node in the communicator match exactly its coordinates in the partition. The communicator created by this function always contains the dimensions and coordinates in TZYX order.

  It is important to understand that this is a collective operation and it must be run on all nodes. The function might be unable to complete successfully for several different reasons, mostly likely when it is run on fewer nodes than the entire partition. It is important to ensure that the return code is MPI_SUCCESS before continuing to use the returned communicator.

- `int MPIX_Pset_same_comm_create (MPI_Comm *pset_comm);`

  This function is a collective operation that creates a set of communicators (each node seeing only one), where all nodes in a given communicator are part of the same pset (all share the same I/O node), see Figure 7-2 on page 76.

  The most common use for this function is to coordinate access to the outside world to maximize the number of I/O Nodes, for example, node 0 in each of the communicators can be arbitrarily used as the “master node” for the communicator, collecting information from the other nodes for writing to disk.

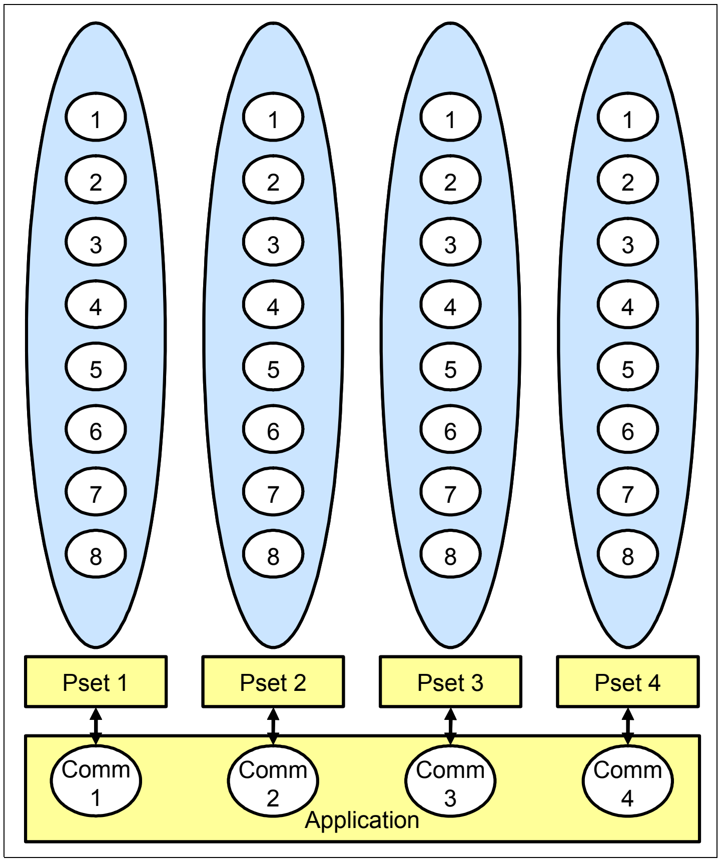


Figure 7-2 MPIX_Pset_same_comm_create() creating communicators

- `int MPIX_Pset_diff_comm_create (MPI_Comm *pset_comm);`

  This function is a collective operation that creates a set of communicators (each node seeing only one), where no two nodes in a given communicator are part of the same pset (all have different I/O Nodes), see Figure 7-3 on page 77. The most common use for this function is to coordinate access to the outside world to maximize the number of I/O Nodes, for example, an application that has an extremely high bandwidth per node requirement can run both MPIX_Pset_same_comm_create() and MPIX_Pset_diff_comm_create().

  Nodes without rank 0 in MPIX_Pset_same_comm_create() can sleep, leaving those with rank 0 independent and parallel access to the functional Ethernet. Those nodes all belong to the same communicator from MPIX_Pset_diff_comm_create(), allowing them to use that communicator instead of MPI_COMM_WORLD for group communication or coordination.

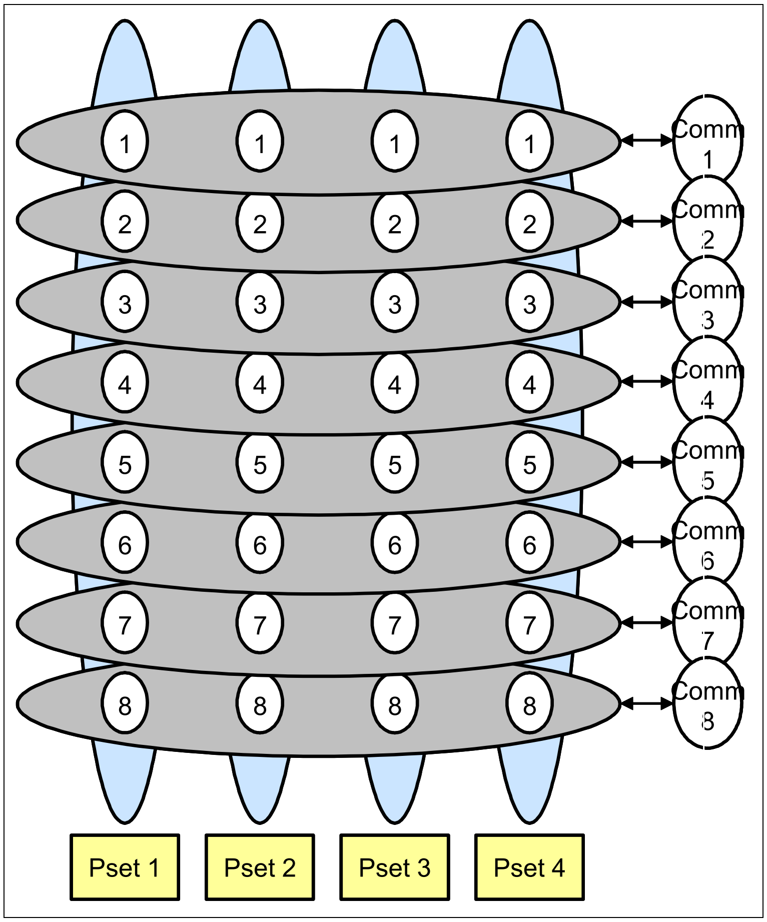


Figure 7-3 PMI_Pset_diff_comm_create() creating communicators

7.3.2 Configuring MPI algorithms at run time

The Blue Gene/P MPI implementation allows developers to enable or disable the algorithms used by the collective operations at run time. Used properly, this feature can provide better application performance.

A single collective operation might appear at different places in the code and have different characteristics, for example, the message size, number of processes involved, and operation can differ at one call site versus another call site for the same collective operation. The MPI library cannot differentiate among the invocations of a given collective in terms of treatment, that is, what algorithm to use to realize that instance of the collective operation. The library might use the same algorithm to realize different collective operation call sites, which in some cases cannot be efficient. It is desirable to give application developers some control over the algorithm selection for a collective operation. One method of controlling the collective operation is by manipulating the environment variables described in Appendix D, “Environment variables” on page 339. The algorithm specified by the environment variables can be applied to all instances of the pertinent communication operation, which might not provide enough granularity to get maximum performance.

Support to enable algorithm selection per collective communication call site was added in Blue Gene/P release V1R3M0.

The algorithms that are available to a communicator for collective operations are defined by the values of properties of the communicator. Refer to the directory /bgsys/drivers/ppcfloor/

comm/include/mpido_properties.h for the definitions of the available properties. These properties represent the collective communication algorithms. The value of a property can be true or false. The initial values for the properties of a communicator depend on its size, shape, and parent communicator. If a communicator is duplicated, the values of the properties of the old communicator are copied to the new communicator.

Application developers get or set the value of a property of a communicator using the following functions, which are declared in the mpix.h header file:

- `int MPIX_Get_properties(MPI_Comm comm, int *prop_array);`
- `int MPIX_Get_property(MPI_Comm comm, int prop, int *result);`
- `int MPIX_Set_property(MPI_Comm comm, int prop, int value);`

> **Important:** These functions give developers full control of the algorithm selection process, which could lead to program crashes if the selection is done inappropriately.

For these functions, *comm* is the communicator, and *prop* is the property from mpido_properties.h. MPIX_Get_properties() retrieves the values for all of the properties of the communicator. The size of *prop_array* must be at least MPIDO_MAX_NUM_BITS integers (`ints`). If the call returns MPI_SUCCESS, each of the elements of *prop_array* are set to the value of the property in the communicator at the offset given by the property number, for example, after calling MPIX_Get_properties(), prop_array[MPIDO_USE_MPICH_ALLTOALL] is set to 1 if the value of the MPIDO_USE_MPICH_ALLTOALL property is true for the communicator or 0 if it is false.

MPIX_Get_property() is used to get the current value of a property for a communicator. If MPIX_Get_property() returns success, *result* is set to 1 if the property is true or 0 if the property is false.

The MPIX_Set_property() function sets the value of the property to true if *value* is 1 or false if *value* is 0. This function is the only one that affects a property's value for a communicator once it is created.

The functions return the following values, indicating the result of the operation:

MPI_SUCCESS	The property was successfully retrieved or set.
MPI_ERR_ARG	The property is not valid.
MPI_ERR_COMM	The communicator is not valid.

Example 7-12 illustrates through pseudocode the use of MPIX_Get_property() and MPIX_Set_property() to specify different algorithms for a MPI_Alltoall() call site. Currently, MPI_Alltoall() has two properties, the property MPIDO_USE_TORUS_ALLTOALL and the property MPIDO_USE_MPICH_ALLTOALL. In the example, we first get the MPIDO_USE_TORUS_ALLTOALL value to see if it is false. If it is, we set it to true so that the algorithm can be used during the call to MPI_Alltoall(). Then we make sure that the MPIDO_USE_MPICH_ALLTOALL property is false, which forces MPI_Alltoall() to use the torus algorithm. We then reset the torus property so that MPI does not use the torus protocol for subsequent invocations of MPI_Alltoall(). Note that if there were three algorithms X, Y, and Z for the alltoall operation and the user wants to force the use of Z, the user is required to set the properties for algorithms X and Y to false.

Example 7-12 Disabling the MPICH protocol on MPI_Alltoall

```
int main(int argc, char **argv)
{
   …
   MPIX_Get_property(comm, MPIDO_USE_TORUS_ALLTOALL, &result);
   if (result == 0)
      /* this causes the following MPI_Alltoall to use torus protocol */
      MPIX_Set_property(comm, MPIDO_USE_TORUS_ALLTOALL, 1);

   MPIX_Get_property(comm, MPIDO_USE_MPICH_ALLTOALL, &result);
   if (result == 1)
      /* turn off the mpich protocol */
      MPIX_Set_property(comm, MPIDO_USE_MPICH_ALLTOALL, 0);

   MPI_Alltoall(…);
   /* this resets the MP_Alltoall algorithm selection to its previous state */
   MPIX_Set_property(comm, MPIDO_USE_TORUS_ALLTOALL, 0);
   …
   …
}
```

7.3.3 Self Tuned Adaptive Routines for MPI

The Blue Gene/P MPI library as of the V1R3M0 release includes a collectives component called the Self Tuned Adaptive Routines for MPI (STAR-MPI)[24]. STAR-MPI can be used to improve application performance by dynamically choosing the best algorithm for the collective operations the application invokes at each invocation point. Best results will be seen when the application has code segments that call collective operations several times (more than 100 calls in a segment). Use of STAR-MPI requires no changes to the application. STAR-MPI can be used when running an application by setting environment variables as described below.

Note: Publishing results obtained from executing STAR-MPI requires the acknowledgment and citation of the software and its owners. The full citation is given in [24].

Table 7-1 describes the environment variables that affect STAR-MPI. Refer to section 11, “mpirun” on page 177 for information about setting environment variables for MPI programs.

Table 7-1 STAR-MPI environment variables

Environment variable	Usage
DCMF_STAR	Enable STAR-MPI for tuning MPI collective operations. STAR-MPI is disabled by default. Set to 1 to enable STAR-MPI.
DCMF_STAR_THRESHOLD	The message size above which STAR-MPI is used. If the message size in the request is smaller than this value then STAR-MPI will not be used. If set, the value must be an integer greater than or equal to 0. If not set, the default value of 2048 is used.
DCMF_STAR_VERBOSE	Enable verbose output for STAR-MPI. If verbose output is enabled, then output files with names of the form “<executable>-star-rank<#>.log”, where <executable> is the name of the executable and <#> is the rank of the task, will be written to the current directory. Verbose output is disabled by default. Set to 1 to enable verbose output. The application must be compiled with debug symbols (the -g option) in order for the verbose output to contain readable function names.

Environment variable	Usage
DCMF_STAR_NUM_INVOCS	The number of invocations that STAR-MPI uses to examine performance of each communication algorithm. If set, the value must be an integer greater than 0. If not set, the default is 10.
DCMF_STAR_TRACEBACK_LEVEL	The function call trace back level to the application. STAR-MPI calculates the invocation point by looking at the call stack. By default, STAR-MPI looks back 3 levels to find where the application calls into the MPI routine. If the application uses wrapper functions (for example, PMPI wrappers) around MPI calls, then the user must set this environment variable to a larger value equal to the number of levels of function calls that are added by the wrappers. The value must be an integer greater than or equal to 3.
DCMF_STAR_CHECK_CALLSITE	Disable a sanity check that ensures that all ranks in the application are involved in the same collective call site. If the application is written in such a way that all ranks are not involved in the same collective call site, then this sanity checking must not be disabled. If the application is known to always have all ranks involved in the same collective call, then this environment variable can be set to 0 to disable the sanity check and eliminate the overhead required to perform the check.

The STAR-MPI verbose output can be used to pre-tune the collectives for an application if an application is called multiple times with similar inputs. The verbose output will show the algorithm that STAR-MPI determined to be optimal for each MPI collective operation invocation. The next time the application is run, the caller can indicate to the MPI library the algorithm to use by setting the DCMF environment variables described in Appendix D, “Environment variables” on page 339 or using the techniques described in 7.3.2, “Configuring MPI algorithms at run time” on page 77. In this way, the application will avoid STAR-MPI’s less-optimal tuning phase while getting the benefit of using the best algorithms on subsequent runs.

7.4 MPI functions

MPI functions have been extensively documented in the literature. In this section, we provide several useful references that provide a comprehensive description of the MPI functions.

Appendix A in *Parallel Programming in C with MPI and OpenMP,* by Michael J. Quinn,[25] describes all the MPI functions as defined in the MPI-1 standard. This reference also provides additional information and recommendations when to use each function.

In addition, you can find information about the MPI standard on the Message Passing Interface (MPI) standard Web site at:

http://www.mcs.anl.gov/research/projects/mpi/

A comprehensive list of the MPI functions is available on the MPI Routines page at:

http://www.mcs.anl.gov/research/projects/mpi/www/www3/

The MPI Routines page includes MPI calls for C and Fortran. For more information, refer to the following books about MPI and MPI-2:

- *MPI: The Complete Reference, 2nd Edition, Volume 1*, by Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra[26]
- *MPI: The Complete Reference, Volume 2: The MPI-2 Extensions*, by William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir[27]

For general information about MPICH2, refer to the MPICH2 Web page at:

http://www.mcs.anl.gov/research/projects/mpich2/

Because teaching MPI is beyond the scope of this book, refer to the following Web page for tutorials and extensive information about MPI:

http://www.mcs.anl.gov/research/projects/mpi/learning.html

7.5 Compiling MPI programs on Blue Gene/P

The Blue Gene/P software provides several scripts to compile and link MPI programs. These scripts make building MPI programs easier by setting the include paths for the compiler and linking in the libraries that implement MPICH2, the common BGP message layer interface (DCMF), and the common BGP message layer interface for general collectives (CCMI) that are required by Blue Gene/P MPI programs.

There are two versions of the libraries and the scripts, a *default* version and a *fast* version. The *default* version of the libraries was built with the GNU Compiler Collection. The GNU Compiler Collection compilers are used in the compiler scripts, such as mpicc. This version also has most error checking enabled. The *fast* version of the libraries has most error checking disabled. The libraries were built using the IBM XL C, C++, and Fortran compilers at a high optimization level. The XL compilers are used in the compiler scripts, such as mpicc. Applications compiled with these scripts tend to see lower message latency because of the improved messaging performance. However, the lack of error checking in the *fast* libraries means that incorrect code in applications or misbehaving hardware can cause incorrect results that would be raised as errors if built with the *default* version.

The *default* scripts are in /bgsys/drivers/ppcfloor/comm/default/bin and the *fast* scripts are in /bgsys/drivers/ppcfloor/comm/fast/bin. Scripts identical to the *default* scripts are in /bgsys/drivers/ppcfloor/comm/bin for backwards compatibility with previous releases of the Blue Gene/P software.

The following scripts are provided to compile and link MPI programs:

mpicc	C compiler
mpicxx	C++ compiler
mpif77	Fortran 77compiler
mpif90	Fortran 90 compiler
mpixlc	IBM XL C compiler
mpixlc_r	Thread-safe version of mpixlc
mpixlcxx	IBM XL C++ compiler
mpixlcxx_r	Thread-safe version of mpixlcxx
mpixlf2003	IBM XL Fortran 2003 compiler

mpixlf2003_r	Thread-safe version of mpixlf2003
mpixlf77	IBM XL Fortran 77 compiler
mpixlf77_r	Thread-safe version of mpixlf77
mpixlf90	IBM XL Fortran 90 compiler
mpixlf90_r	Thread-safe version of mpixlf90
mpixlf95	IBM XL Fortran 95 compiler
mpixlf95_r	Thread-safe version of mpixlf95
mpich2version	Prints MPICH2 version information

> **Note:** When you invoke the previous scripts, if you do not set the optimization level using -O, the default is set to no optimization (-O0).

The following environment variables can be set to override the compilers used by the scripts:

MPICH_CC	C compiler
MPICH_CXX	C++ compiler
MPICH_FC	Fortran 77 compiler

The IBM XL Fortran 90 compiler is incompatible with the Fortran 90 MPI bindings in the MPICH library built with GCC. Therefore, the *default* version of the mpixlf90 script cannot be used with the Fortran 90 MPI bindings.

Example 7-13 shows how to use the mpixlf77 script in a makefile.

Example 7-13 Use of MPI script mpixlf77

```
XL         = /bgsys/drivers/ppcfloor/comm/default/bin/mpixlf77

EXE         =  fhello
OBJ         =  hello.o
SRC         =  hello.f
FLAGS       =  -O3 -qarch=450 -qtune=450

$(EXE): $(OBJ)
        ${XL} $(FLAGS) -o $@ $^

$(OBJ): $(SRC)
        ${XL} $(FLAGS) -c $<

clean:
        $(RM) $(OBJ) $(EXE)
```

To build MPI programs for Blue Gene/P, the compilers can be invoked directly rather than using the above scripts. When invoking the compilers directly you must explicitly include the required MPI libraries. Example 7-14 shows a makefile that does not use the scripts.

Example 7-14 Makefile with explicit reference to libraries and include files

```
BGP_FLOOR   = /bgsys/drivers/ppcfloor
BGP_IDIRS   = -I$(BGP_FLOOR)/arch/include -I$(BGP_FLOOR)/comm/include
BGP_LIBS    = -L$(BGP_FLOOR)/comm/lib -lmpich.cnk \
 -L$(BGP_FLOOR)/comm/lib -ldcmf.cnk -ldcmfcoll.cnk \
 -lpthread -lrt \
 -L$(BGP_FLOOR)/runtime/SPI -lSPI.cna

XL          = /opt/ibmcmp/xlf/bg/11.1/bin/bgxlf

EXE          =  fhello
OBJ          =  hello.o
SRC          =  hello.f
FLAGS        =  -O3 -qarch=450 -qtune=450 $(BGP_IDIRS)

$(EXE): $(OBJ)
        ${XL} $(FLAGS) -o $@ $^ $(BGP_LIBS)

$(OBJ): $(SRC)
        ${XL} $(FLAGS) -c $<

clean:
        $(RM) $(OBJ) $(EXE)
```

7.6 MPI communications performance

Communications performance is an important aspect when running parallel applications, particularly, when running on a distributed memory system, such as the Blue Gene/P system. On both the Blue Gene/L and Blue Gene/P systems, instead of implementing a single type of network capable of transporting all required protocols, these two systems have separate networks for different types of communications.

Usually the following measurements provide information about the network and can be used to look at the parallel performance of applications:

Bandwidth The number of MB of data that can be sent from one node to another node in one second

Latency The amount of time it takes for the first byte that is sent from one node to reach its target node

The values for bandwidth and latency provide information about communication.

Here we illustrate two cases. The first case corresponds to a benchmark that involves a single transfer. The second case corresponds to a collective as defined in the “Intel® MPI Benchmarks” (see the URL that follows). “Intel MPI Benchmarks” was formerly known as “Pallas MPI Benchmarks” (PMB-MPI1 for MPI1 standard functions only). Intel MPI Benchmarks - MPI1 provides a set of elementary MPI benchmark kernels.

For more details, see the product documentation included in the package that you can download from the following Intel Web page:

http://www.intel.com/cd/software/products/asmo-na/eng/219848.htm

7.6.1 MPI point-to-point

In the Intel MPI Benchmarks, a single transfer corresponds to the PingPong and PingPing benchmarks. We illustrate a comparison between the Blue Gene/L and Blue Gene/P systems for the case of PingPong. This benchmark illustrates a single message that is transferred between two MPI tasks, in our case, on two different nodes.

To run this benchmark, we used the Intel MPI Benchmark Suite Version 2.3, MPI-1 part. On the Blue Gene/L system, the benchmark was run in coprocessor mode. (See *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686.) On the Blue Gene/P system, we used the SMP Node mode. **mpirun** was invoked as shown in Example 7-15 and Example 7-16 for the Blue Gene/L and Blue Gene/P systems respectively.

Example 7-15 mpirun on the Blue Gene/L system

```
mpirun -nofree -timeout 120 -verbose 1 -mode CO -env "BGL_APP_L1_WRITE_THROUGH=0
BGL_APP_L1_SWOA=0" -partition R000 -cwd /bglscratch/pallas -exe
/bglscratch/pallas/IMB-MPI1.4MB.perf.rts -args "-msglen 4194304.txt -npmin 512
PingPong" | tee IMB-MPI1.4MB.perf.PingPong.4194304.512.out) >>
run.IMB-MPI1.4MB.perf.PingPong.4194304.512.out 2>&1
```

Example 7-16 mpirun on the Blue Gene/P system

```
mpirun -nofree -timeout 300 -verbose 1 -np 512 -mode SMP  -partition R01-M1 -cwd
/bgusr/BGTH_BGP/test512nDD2BGP/pallas/pall512DD2SMP/bgpdd2sys1-R01-M1 -exe
/bgusr/BGTH_BGP/test512nDD2BGP/pallas/pall512DD2SMP/bgpdd2sys1-R01-M1/IMB-MPI1.4MB
.perf.rts -args "-msglen 4194304.txt -npmin 512 PingPong" | tee
IMB-MPI1.4MB.perf.PingPong.4194304.512.out) >>
run.IMB-MPI1.4MB.perf.PingPong.4194304.512.out 2>&1
```

Figure 7-4 shows the bandwidth on the torus network as a function of the message size, for one simultaneous pair of nearest neighbor communications. The protocol switch from short to eager is visible in both cases, where the eager to rendezvous switch is most pronounced on the Blue Gene/L system (see the asterisks (*)). This figure also shows the improved performance on the Blue Gene/P system (see the diamonds).

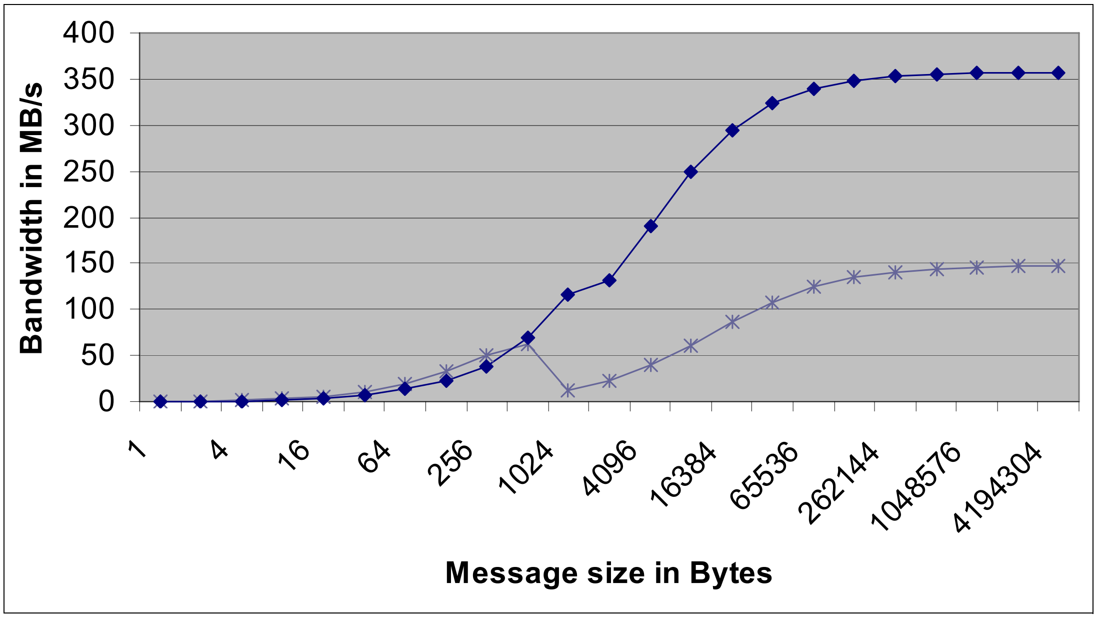


Figure 7-4 Bandwidth versus message size

7.6.2 MPI collective

In the Intel MPI Benchmarks, the collective benchmarks correspond to the Bcast, Allgather, Allgatherv, Alltoall, Alltoallv, Reduce, Reduce_scatter, Allreduce, and Barrier benchmarks. We illustrate a comparison between the Blue Gene/L and Blue Gene/P systems for the case of Allreduce, which is a popular collective that is used in certain scientific applications. These benchmarks measure the message-passing power of a system as well as the quality of the implementation.

To run this benchmark, we used the Intel MPI Benchmark Suite Version 2.3, MPI-1 part. On the Blue Gene/P system, the benchmark was run in coprocessor mode. On the Blue Gene/P system, we used SMP Node mode. **`mpirun`** was invoked as shown in Example 7-17 and Example 7-18 for the Blue Gene/L and Blue Gene/P systems, respectively.

Example 7-17   mpirun on the Blue Gene/L system

```
mpirun -nofree -timeout 120 -verbose 1 -mode CO -env "BGL_APP_L1_WRITE_THROUGH=0
BGL_APP_L1_SWOA=0" -partition R000 -cwd
/bglscratch/BGTH/testsmall512nodeBGL/pallas -exe
/bglscratch/BGTH/testsmall512nodeBGL/pallas/IMB-MPI1.4MB.perf.rts -args "-msglen
4194304.txt -npmin 512 Allreduce" | tee
IMB-MPI1.4MB.perf.Allreduce.4194304.512.out) >>
run.IMB-MPI1.4MB.perf.Allreduce.4194304.512.out 2>&1
```

Example 7-18   mpirun on the Blue Gene/P system

```
mpirun -nofree -timeout 300 -verbose 1 -np 512 -mode SMP  -partition R01-M1 -cwd
/bgusr/BGTH_BGP/test512nDD2BGP/pallas/pall512DD2SMP/bgpdd2sys1-R01-M1 -exe
/bgusr/BGTH_BGP/test512nDD2BGP/pallas/pall512DD2SMP/bgpdd2sys1-R01-M1/IMB-MPI1.4MB
.perf.rts -args "-msglen 4194304.txt -npmin 512 Allreduce" | tee
IMB-MPI1.4MB.perf.Allreduce.4194304.512.out) >>
run.IMB-MPI1.4MB.perf.Allreduce.4194304.512.out 2>&1
```

Collective operations are more efficient on the Blue Gene/P system. You should try to use collective operations instead of point-to-point communication wherever possible. The overhead for point-to-point communications is much larger than for collectives. Unless all of your point-to-point communication is purely to the nearest neighbor, it is difficult to avoid network congestion on the torus network.

Alternatively, collective operations can use the barrier (global interrupt) network or the torus network. If they run over the torus network, they can still be optimized by using specially designed communication patterns that achieve optimum performance. Doing this manually with point-to-point operations is possible in theory, but in general, the implementation in the Blue Gene/P MPI library offers superior performance.

With point-to-point communication, the goal of reducing the point-to-point Manhattan distances necessitates a good mapping of MPI tasks to the physical hardware. For collectives, mapping is equally important because most collective implementations prefer certain communicator shapes to achieve optimum performance. In general, collectives using “rectangular” subcommunicators (with the ranks organized in lines, planes, or cubes) will out perform “irregular” subcommunicators” (any communicator that is not rectangular). Refer to Appendix F, “Mapping” on page 355, which illustrates the technique of mapping.

Similar to point-to-point communications, collective communications also work best if you do not use complicated derived data types, and if your buffers are aligned to 16-byte boundaries. While the MPI standard explicitly allows for MPI collective communications to take place at the same time as point-to-point communications (on the same communicator), generally we do not recommend this for performance reasons.

Table 7-2 summarizes the MPI collectives that have been optimized on the Blue Gene/P system. All data values are for a 512-node partition running in SMP mode. Many collectives make use of both the torus and collective networks as indicated in the table.

Table 7-2 MPI collectives optimized on the Blue Gene/P system

MPI routine	Communicator	Data type	Network	Latency	Bandwidth
MPI_Barrier	MPI_COMM_WORLD	N/A	Global Interrupts	1.25 µs	N/A
	Rectangular	N/A	Torus	10.96 µs	N/A
	All other subcommunicators	N/A	Torus	22.61 µs	N/A
MPI_Bcast	MPI_COMM_WORLD	Byte	Collective for latency, torus for BW	3.61µs	2047 MBps[a]
	Rectangular	Byte	Torus	11.58 µs	2047 MBps[a]
	All other subcommunicators	Byte	Torus	15.36 µs	357 MBps
MPI_Allreduce	MPI_COMM_WORLD	Integer	Collective for latency, torus for BW	3.76 µs	780 MBps[a]
		Double	Collective for latency, torus for BW	5.51 µs	363 MBps[a]
	Rectangular	Integer	Torus	17.66 µs	261 MBps[a]
		Double	Torus	17.54 µs	363 MBps[a]
	All other subcommunicators	Integer	Torus	38.06 µs	46 MBps
		Double	Torus	37.96 µs	49 MBps
MPI_Alltoallv	All communicators	Byte	Torus	355 µs[b]	~97% of peak torus bisection bandwidth
MPI_Allgatherv	MPI_COMM_WORLD	Byte	Collective for latency, torus for BW	18.79 µs[c]	3.6x MPICH[d]
	Rectangular	Byte	Torus	276.32 µs[c]	3.6x MPICH[d]
MPI_Gather	MPI_COMM_WORLD	Byte	Collective for BW	10.77 µs	2.0x MPICH[d]
MPI_Scatter	MPI_COMM_WORLD	Byte	Collective for BW	9.91 µs	4.2x MPICH[a]
MPI_Scatterv	All	Byte	Torus for BW	167 µs	2.6x MPICH[d]
	MPI_COMM_WORLD	Byte	Collective	20 µs	3.7x MPICH[e]
MPI_Reduce	MPI_COMM_WORLD	Integer	Collective	3.82 µs	780 MBps
		Double	Torus	4.03 µs	304 MBps
	Rectangular	Integer	Torus	17.27 µs	284 MBps

MPI routine	Communicator	Data type	Network	Latency	Bandwidth
		Double	Torus	17.32 µs	304 MBps
	All other subcommunicators	Integer	Torus	8.43 µs	106 MBps
		Double	Torus	8.43 µs	113 MBps

a. Maximum bandwidth performance requires 16-byte aligned buffers.

b. A 1-byte alltoall is moving 512 bytes of data per node on a 512 node partition.

c. A 1-byte allgatherv is moving 512 bytes of data per node on a 512 node partition.

d. Calculating these bandwidths can be done in multiple ways so we simply compare the time for the optimized routine to the time using the MPICH default point-to-point based algorithm.

e. This option is off by default and can be turned on with the DCMF_SCATTERV=B option. It requires that all nodes have a valid sendcounts array, not just the root. Turning it on without valid sendcounts can lead to hangs or unexpected results.

Figure 7-5 and Figure 7-6 on page 88 show a comparison between the IBM Blue Gene/L and Blue Gene/P systems for the MPI_Allreduce() type of communication on integer data types with the sum operation.

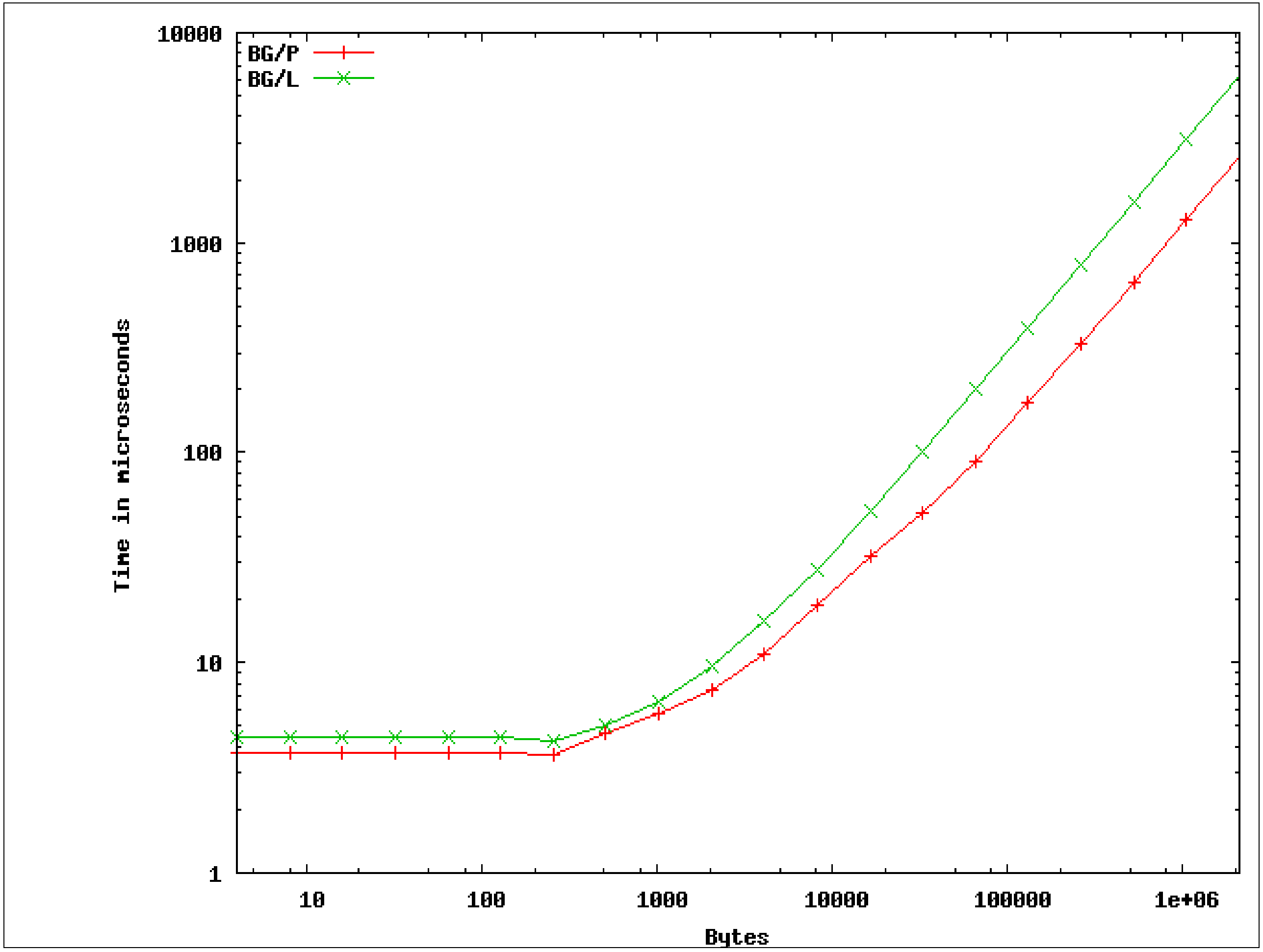


Figure 7-5 MPI_Allreduce() integer sum wall time performance on 512 nodes.

Figure 7-6 on page 88 shows MPI_Allreduce() integer sum bandwidth performance on 512 nodes.

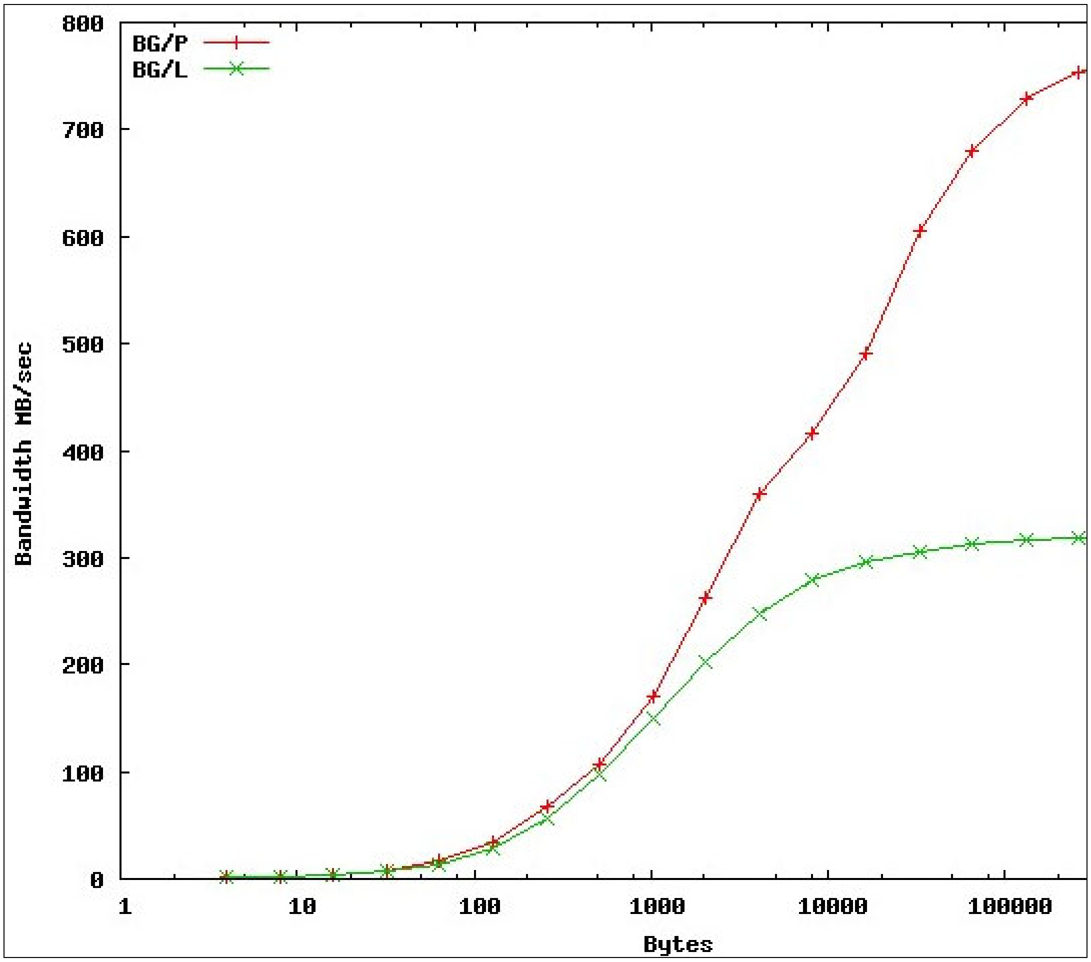


Figure 7-6 MPI_Allreduce() integer sum bandwidth performance on 512 nodes

7.7 OpenMP

The OpenMP API is supported on the Blue Gene/P system for shared-memory parallel programming in C/C++ and Fortran. This API has been jointly defined by a group of hardware and software vendors and has evolved as a standard for shared-memory parallel programming.

OpenMP consists of a collection of compiler directives and a library of functions that can be invoked within an OpenMP program. This combination provides a simple interface for developing parallel programs on shared-memory architectures. In the case of the Blue Gene/P system, it allows the user to exploit the SMP mode on each Compute Node. Multi-threading is now enabled on the Blue Gene/P system. Using OpenMP, the user can have access to data parallelism as well as functional parallelism.

For additional information, refer to the official OpenMP Web site at:

`http://www.openmp.org/`

7.7.1 OpenMP implementation for Blue Gene/P

The Blue Gene/P system supports shared-memory parallelism on single nodes. The XL compilers support the following constructs:

- Full support for OpenMP 2.5 standard
- Support for the use of the same infrastructure as the OpenMP on IBM AIX® and Linux
- Interoperability with MPI:
  - MPI at outer level, across the Compute Nodes
  - OpenMP at the inner level, within a Compute Node
- Autoparallelization based on the same parallel execution framework

  Enables autoparallelization as one of the loop optimizations
- Thread-safe version for each compiler:
  - `bgxlf_r`
  - `bgxlc_r`
  - `bgcc_r`
- Use of the thread-safe compiler version with any threaded, OpenMP, or SMP application:
  - `-qsmp` must be used on OpenMP or SMP applications.
  - `-qsmp` by itself automatically parallelizes loops.
  - `-qsmp=omp` parallelizes based on OpenMP directives in the code.
  - Shared-memory model is on the Blue Gene/P system.

7.7.2 Selected OpenMP compiler directives

The latest set of OpenMP compiler directives is documented in the OpenMP ARB release Version 2.5 specification. Version 2.5 combines Fortran and C/C++ specifications into a single specification. It also fixes inconsistencies. We summarize some of the directives as follows:

parallel Directs the compiler for that section of the code to be executed in parallel by multiple threads

for Directs the compiler to execute a `for` loop with independent iterations; iterations can be executed by different threads in parallel

parallel for The syntax for parallel loops

sections Directs the compiler of blocks of non-iterative code that can be executed in parallel

parallel sections Syntax for parallel sections

critical Restricts the following section of the code to be executed by a single thread at a time

single Directs the compiler to execute a section of the code by a single thread

Parallel operations are often expressed in C/C++ and Fortran95 programs as `for` loops as shown in Example 7-19.

Example 7-19 for loops in Fortran and C

```
for (i = start; i < num; i += end)
{ array[i] = 1; m[i] = c;}

or

   integer i, n, sum
   sum = 0
   do 5 i = 1, n
   sum = sum + i
5  continue
```

The compiler can automatically locate and, where possible, parallelize all countable loops in your program code in the following situations:

- There is no branching into or out of the loop.
- An increment expression is not within a critical section.
- A countable loop is automatically parallelized only if all of the following conditions are met:
  - The order in which loop iterations start or end does not affect the results of the program.
  - The loop does not contain I/O operations.
  - Floating-point reductions inside the loop are not affected by round-off error, unless the `-qnostrict` option is in effect.
  - The `-qnostrict_induction` compiler option is in effect.
  - The `-qsmp=auto` compiler option is in effect.
  - The compiler is invoked with a thread-safe compiler mode.

In the case of C/C++ programs, OpenMP is invoked via pragmas as shown in Example 7-20.

Pragma: The word *pragma* is short for *pragmatic information.*[28] Pragma is a way to communicate information to the compiler:

```
#pragma omp <rest of pragma>
```

Example 7-20 pragma usage

```
#pragma omp parallel for
    for (i = start; i < num;i += end)
{ array[i] = 1; m[i] = c;}
```

The `for` loop must not contain statements, such as the following examples, that allow the loop to be exited prematurely:

- `break`
- `return`
- `exit`
- `go to labels outside the loop`

In a `for` loop, the master thread creates additional threads. The loop is executed by all threads, where every thread has its own address space that contains all of the variables the thread can access. Such variables might be:

- Static variables
- Dynamically allocated data structures in the heap
- Variables on the run-time stack

In addition, variables must be defined according to the type. Shared variables have the same address in the execution context of every thread. It is important to understand that all threads have access to shared variables. Alternatively, private variables have a different address in the execution memory of every thread. A thread can access its own private variables, but it *cannot* access the private variable of another thread.

Pragma parallel: In the case of the parallel for pragma, variables are shared by default, with exception of the loop index.

Example 7-21 shows a simple Fortran95 example that illustrates the difference between private and shared variables.

Example 7-21 Fortran example using the parallel do directive

```
program testmem
        integer n
        parameter (n=2)
        parameter (m=1)
        integer a(n), b(m)
!$OMP parallel do
        do i = 1, n
          a(i) = i
        enddo
        write(6,*)'Done: testmem'
        end
```

In Example 7-21, no variables are explicitly defined as either private or shared. In this case, by default, the compiler assigns the variable that is used for the do-loop index as private. The rest of the variables are shared. Figure 7-7 on page 92 illustrates both private and shared variables as shown in *Parallel Programming in C with MPI and OpenMP*.[29] In this figure, the blue and yellow arrows indicate which variables are accessible by all the threads.

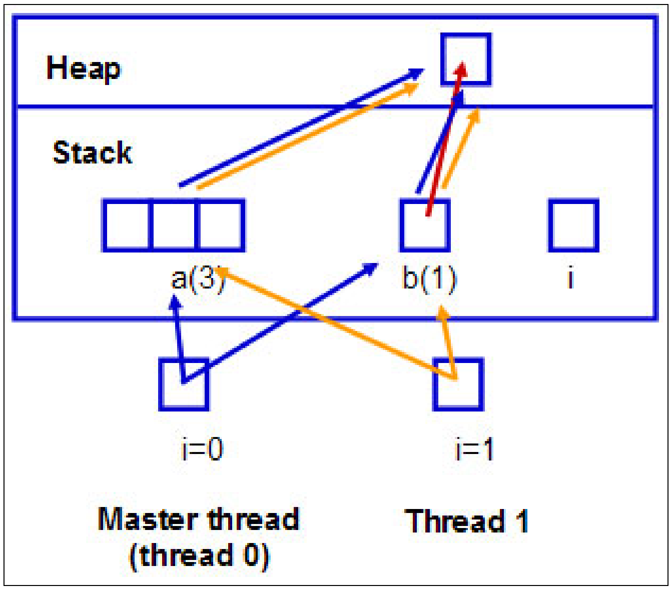


Figure 7-7 Memory layout for private and shared variables

7.7.3 Selected OpenMP compiler functions

The following functions are selected for the OpenMP compiler:

omp_get_num_procs	Returns the number of processors
omp_get_num_threads	Returns the number of threads in a particular parallel region
omp_get_thread_num	Returns the thread identification number
omp_set_num_threads	Allocates numbers of threads for a particular parallel region

7.7.4 Performance

To illustrate the effect of selected OpenMP compiler directives and the implications in terms of performance of a particular do loop, we chose the π programs presented in *Parallel Programming in C with MPI and OpenMP*[30] and apply them to the Blue Gene/P system. These simple examples illustrate how to use these directives and some of the implications in selecting a particular directive over another directive. Example 7-22 shows a simple program to compute π.

Example 7-22 Sequential version of the pi.c program

```
int main(argc, argv)
int argc;
char *argv[];
{
   long n, i;
   double area, pi, x;
       n    = 1000000000;
       area = 0.0;
       for (i = 0; i < n; i++) {
          x = (i+0.5)/n;
            area += 4.0 / (1.0 + x*x);
       }
       pi = area / n;
       printf ("Estimate of pi: %7.5f\n", pi);
}
```

The first way to parallelize this code is to include an OpenMP directive to parallelize the `for` loop as shown in Example 7-23.

Example 7-23 Simple use of parallel for loop

```
#include <omp.h>

long long timebase(void);

int main(argc, argv)
int argc;
char *argv[];
{
   int num_threads;
   long n, i;
   double area, pi, x;
   long long time0, time1;
   double cycles, sec_per_cycle, factor;
       n    = 1000000000;
       area = 0.0;
       time0 = timebase();
#pragma omp parallel for private(x)
       for (i = 0; i < n; i++) {
          x = (i+0.5)/n;
            area += 4.0 / (1.0 + x*x);
       }
       pi = area / n;
       printf ("Estimate of pi: %7.5f\n", pi);
       time1 = timebase();
       cycles = time1 - time0;
       factor = 1.0/850000000.0;
       sec_per_cycle = cycles * factor;
       printf("Total time %lf \n",sec_per_cycle, "Seconds \n");
}
```

Unfortunately this simple approach creates a race condition when computing the area. While different threads compute and update the value of the area, other threads might be computing and updating area as well, therefore producing the wrong results. This particular race condition can be solved in two ways. One way is to use a *critical pragma* to ensure mutual exclusion among the threads, and the other way is to use the *reduction clause*.

Example 7-24 illustrates use of the critical pragma.

Example 7-24 Usage of critical pragma

```
#include <omp.h>

long long timebase(void);

int main(argc, argv)
int argc;
```

```
char *argv[];
{
   int num_threads;
   long n, i;
   double area, pi, x;
   long long time0, time1;
   double cycles, sec_per_cycle, factor;
       n    = 1000000000;
       area = 0.0;
       time0 = timebase();
#pragma omp parallel for private(x)
       for (i = 0; i < n; i++) {
          x = (i+0.5)/n;
#pragma omp critical
            area += 4.0 / (1.0 + x*x);
       }
       pi = area / n;
       printf ("Estimate of pi: %7.5f\n", pi);
       time1 = timebase();
       cycles = time1 - time0;
       factor = 1.0/850000000.0;
       sec_per_cycle = cycles * factor;
       printf("Total time %lf \n",sec_per_cycle, "Seconds \n");
}
```

Example 7-25 corresponds to the reduction clause.

Example 7-25 Usage of the reduction clause

```
#include <omp.h>

long long timebase(void);

int main(argc, argv)
int argc;
char *argv[];
{
   int num_threads;
   long n, i;
   double area, pi, x;
   long long time0, time1;
   double cycles, sec_per_cycle, factor;
       n    = 1000000000;
       area = 0.0;
       time0 = timebase();
#pragma omp parallel for private(x) reduction(+: area)
       for (i = 0; i < n; i++) {
          x = (i+0.5)/n;
          area += 4.0 / (1.0 + x*x);
       }
       pi = area / n;
       printf ("Estimate of pi: %7.5f\n", pi);
       time1 = timebase();
       cycles = time1 - time0;
       factor = 1.0/850000000.0;
```

```
        sec_per_cycle = cycles * factor;
        printf("Total time %lf \n",sec_per_cycle, "Seconds \n");
}
```

To compile these two programs on the Blue Gene/P system, the makefile for pi_critical.c shown in Example 7-26 can be used. A similar makefile can be used for the program illustrated in Example 7-25 on page 94.

Example 7-26 Makefile for the pi_critical.c program

```
BGP_FLOOR    = /bgsys/drivers/ppcfloor
BGP_IDIRS    = -I$(BGP_FLOOR)/arch/include -I$(BGP_FLOOR)/comm/include
BGP_LIBS     = -L$(BGP_FLOOR)/comm/lib -L$(BGP_FLOOR)/runtime/SPI -lmpich.cnk
-ldcmfcoll.cnk -ldcmf.cnk -lrt -lSPI.cna -lpthread

XL           = /opt/ibmcmp/vac/bg/9.0/bin/bgxlc_r

EXE          =  pi_critical_bgp
OBJ          =  pi_critical.o
SRC          =  pi_critical.c
FLAGS        =  -O3 -qsmp=omp:noauto -qthreaded -qarch=450  -qtune=450
-I$(BGP_FLOOR)/comm/include
FLD          =  -O3 -qarch=450 -qtune=450

$(EXE): $(OBJ)
        ${XL} $(FLAGS) -o $(EXE) $(OBJ) timebase.o $(BGP_LIBS)
$(OBJ): $(SRC)
        ${XL} $(FLAGS) $(BGP_IDIRS) -c $(SRC)

clean:
        rm pi_critical.o pi_critical_bgp
```

Table 7-3 illustrates the performance improvement by using the reduction clause.

Table 7-3 Parallel performance using critical pragma versus reduction clause

	Execution time in (seconds)	
Threads	**Using critical pragma**	**Using reduction clause**
1		
IBM POWER4 1.0 GHz	586.37	20.12
IBM POWER5 1.9 GHz	145.03	5.22
IBM POWER6™ 4.7 GHz	180.80	4.78
Blue Gene/P	560.08	12.80
2		
POWER4 1.0 GHz	458.84	10.08
POWER5 1.9 GHz	374.10	2.70
POWER6 4.7 GHz	324.71	2.41
Blue Gene/P	602.62	6.42
4		

	Execution time in (seconds)	
Threads	**Using critical pragma**	**Using reduction clause**
POWER4 1.0 GHz	552.54	5.09
POWER5 1.9 GHz	428.42	1.40
POWER6 4.7 GHz	374.51	1.28
Blue Gene/P	582.95	3.24

For more in-depth information with additional examples, we recommend you read *Parallel Programming in C with MPI and OpenMP*.[31] In this section, we selected to illustrate only the π program.

8

Developing applications with IBM XL compilers

With the IBM XL family of optimizing compilers, you can develop C, C++, and Fortran applications for the IBM Blue Gene/P system. This family comprises the following products, which we refer to in this chapter as *Blue Gene XL compilers*:

- XL C/C++ Advanced Edition V9.0 for Blue Gene
- XL Fortran Advanced Edition V11.1 for Blue Gene

The information that we present in this chapter is specific to the IBM Blue Gene/P supercomputer. It does not include general XL compiler information. For complete documentation about these compilers, refer to the libraries at the following Web addresses:

- XL C/C++

  `http://www.ibm.com/software/awdtools/xlcpp/library/`

- XL Fortran

  `http://www-306.ibm.com/software/awdtools/fortran/xlfortran/library/`

In this chapter, we discuss specific considerations for developing, compiling, and optimizing C/C++ and Fortran applications for the IBM Blue Gene/P PowerPC 450 processor and a single-instruction multiple-data (SIMD), double-precision floating-point multiply add unit (double floating-point multiply add (FMA). The following topics are discussed:

- Compiler overview
- Compiling and linking applications on Blue Gene/P
- Default compiler options
- Unsupported options
- Support for pthreads and OpenMP
- Creation of libraries on Blue Gene/P
- XL runtime libraries
- Mathematical Acceleration Subsystem libraries
- IBM Engineering Scientific Subroutine Library
- Configuring Blue Gene/P builds
- Python
- Tuning your code for Blue Gene/P

- Tips for optimizing applications
- Identifying performance bottlenecks

Several documents cover part of the material presented in this chapter. In addition to the XL family of compilers manuals that we reference throughout this chapter, we recommend that you read the following documents:

- *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686
- *IBM System Blue Gene Solution: Application Development*, SG24-7179

We also recommend that you read the article by Mark Mendell, “Exploiting the Dual Floating Point Units in Blue Gene/L,” which provides detailed information about the SIMD functionality in the XL family of compilers. You can find this article on the Web at:

http://www-1.ibm.com/support/docview.wss?uid=swg27007511

8.1 Compiler overview

The Blue Gene/P system uses the same XL family of compilers as the IBM Blue Gene/L system. The Blue Gene/P system supports cross-compilation, and the compilers run on the Front End Node. The compilers for the Blue Gene/P system have specific optimizations for its architecture. In particular, the XL family of compilers generate code appropriate for the double floating-point unit (FPU) of the Blue Gene/P system.

The Blue Gene/P system has compilers for the C, C++, and Fortran programming languages. The compilers on the Blue Gene/P system take advantage of the double FPU available on the Blue Gene/P system. They also incorporate code optimizations specific to the Blue Gene/P instruction scheduling and memory hierarchy characteristics.

In addition to the XL family of compilers, the Blue Gene/P system supports a version of the GNU compilers for C, C++, and Fortran. These compilers do not generate highly optimized code for the Blue Gene/P system. In particular, they do not automatically generate code for the double FPUs, and they do not support OpenMP.

Tools that are commonly associated with the GNU compilers (known as *binutils*) are supported in the Blue Gene/P system. The same set of compilers and tools is used for both Linux and the Blue Gene/P proprietary operating system. The Blue Gene/P system supports the execution of Python-based user applications.

The GNU compiler toolchain also provides the dynamic linker, which is used both by Linux and the Blue Gene/P proprietary operating system to support dynamic objects. The toolchain is tuned to support both environments. The GNU “aux vector” technique is employed to pass kernel-specific information to the C library when tuning must be specific to one of the kernels.

8.2 Compiling and linking applications on Blue Gene/P

In this section, we provide information about compiling and linking applications that run on the Blue Gene/P system. For complete information about compiler and linker options, see the following documents available on the Web:

- XL C/C++ Compiler Reference

  http://www-306.ibm.com/software/awdtools/xlcpp/library/

- XL Fortran User Guide

  http://www-306.ibm.com/software/awdtools/fortran/xlfortran/library/

You can also find these documents in the following directories:

- /opt/ibmcmp/vacpp/bg/9.0/doc (C and C++)
- /opt/ibmcmp/xlf/bg/11.1/doc (Fortran)

The compilers are in the following directories:

- /opt/ibmcmp/vac/bg/9.0/bin
- /opt/ibmcmp/vacpp/bg/9.0/bin
- /opt/ibmcmp/xlf/bg/11.1/bin

The Blue Gene/P release includes the following differences for compiling and linking applications:

- Blue Gene/P compiler wrapper names changed:
  - `blrts_` is replaced by `bg`.
  - xlf 11.1, vacpp 9.0, and vac 9.0 on the Blue Gene/L system support both `blrts_` and `bg`.
- `-qarch=450d/450` is for the Blue Gene/P system, and `440d/440` is for the Blue Gene/L system.

8.3 Default compiler options

Compilations most commonly occur on the Front End Node. The resulting program can run on the Blue Gene/P system without manually copying the executable to the Service Node. See Chapter 9, “Running and debugging applications” on page 139, and Chapter 11, “mpirun” on page 177, to learn how to run programs on the Blue Gene/P system.

The script or makefile that you use to invoke the compilers should have certain compiler options. Specifically the architecture-specific options, which optimize processing for the Blue Gene/P 450d processor architecture, should be set to the following defaults:

- `-qarch=450`

  This option generates code for a single FPU only, but it can give correct results if invalid code is generated by `-qarch=450d`. You can follow up with `-qarch=450d` when optimizing performance via more aggressive compilation.

- `-qtune=450`

  Optimizes object code for the 450 family of processors. Single FPU only.

- `-qcache=level=1:type=i:size=32:line=32:assoc=64:cost=8`

  Specifies the L1 instruction cache configuration for the Blue Gene/P architecture to allow greater optimization with options `-O4` and `-O5`.

- `-qcache=level=1:type=d:size=32:line=32:assoc=64:cost=8`

  Specifies the L1 data cache configuration for the Blue Gene/P architecture to allow greater optimization with options `-O4` and `-O5`.

- `-qcache=level=2:type=c:size=4096:line=128:assoc=8:cost=40`

  Specifies the L2 (combined data and instruction) cache configuration for the Blue Gene/P architecture to allow greater optimization with options `-O4` and `-O5`.

- `-qnoautoconfig`

  Allows code to be cross-compiled on other machines at optimization levels `-O4` or `-O5`, by preserving the Blue Gene/P architecture-specific options.

Scripts are already available that do much of this for you. They reside in the same bin directory as the compiler binary (/opt/ibmcmp/xlf/bg/11.1/bin or /opt/ibmcmp/vacpp/bg/9.0/bin or /opt/ibmcmp/vac/bg/9.0/bin). Table 8-1 lists the names.

Table 8-1 Scripts available in the bin directory for compiling and linking

Language	Script name or names
C	bgc89, bgc99, bgcc, bgxlc bgc89_r, bgc99_r bgcc_r, bgxlc_r
C++	bgxlc++, bgxlc++_r, bgxlC, bgxlC_r
Fortran	bgf2003, bgf95, bgxlf2003, bgxlf90_r, bgxlf_r, bgf77, bgfort77, bgxlf2003_r, bgxlf95, bgf90, bgxlf, bgxlf90, bgxlf95_r

Important: The double FPU does not generate exceptions. Therefore, the `-qflttrap` option is invalid with the 450d processor. Instead you should reset the 450d processor to `-qarch=450`.

8.4 Unsupported options

The following compiler options, although available for other IBM systems, are not supported by the Blue Gene/P hardware; therefore, do not use them:

- `-q64`: The Blue Gene/P system uses a 32-bit architecture; you cannot compile in 64-bit mode.
- `-qaltivec`: The 450 processor does not support VMX instructions or vector data types.

Note: -qsigtrap is supported on Blue Gene/P release V1R4M0 and later.

8.5 Support for pthreads and OpenMP

The Blue Gene/P system supports shared-memory parallelism on single nodes. The XL compilers support the following constructs:

- Full support for the OpenMP 2.5 standard[32]
- Use of the same infrastructure as the OpenMP that is supported on IBM AIX and Linux
- Interoperability with MPI
  - MPI at outer level, across the Compute Nodes
  - OpenMP at the inner level, within a Compute Node
- Autoparallelization based on the same parallel execution framework

  Enablement of autoparallelization as one of the loop optimizations
- All the thread-safe scripts of the compiler end in _r, as shown in the following examples:
  - `bgxlf_r`
  - `bgxlc_r`
  - `bgxlC_r`
  - `bgcc_r`

  The thread-safe compiler version should be used with any threaded, OpenMP, or SMP application.

Thread-safe libraries: Thread-safe libraries ensure that data access and updates are synchronized between threads.

- Usage of `-qsmp` OpenMP and pthreaded applications:
  - `-qsmp` by itself automatically parallelizes loops.
  - `-qsmp=omp` automatically parallelizes based on OpenMP directives in the code.
  - `-qsmp=omp:noauto` should be used when parallelizing codes manually. It prevents the compiler from trying to automatically parallelize loops.

Note: `-qsmp` must be used only with *thread-safe compiler mode invocations* such as `xlc_r`. These invocations ensure that the pthreads, xlsmp, and thread-safe versions of all default run-time libraries are linked to the resulting executable. See the language reference for more details about the `-qsmp` suboptions at:

http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/index.jsp

OpenMP can be used with the GNU compiler but support for OpenMP requires a newer compiler than is shipped with Blue Gene/P. Instructions to build the 4.3.2 GNU compiler with GOMP are provided in /bgsys/drivers/ppcfloor/toolchain/README.toolchain.gomp, which is shipped with the Blue Gene/P software.

8.6 Creation of libraries on Blue Gene/P

On Blue Gene/P three types of libraries can be created:

- Static libraries
- Shared libraries
- Dynamically loaded libraries

Static libraries are loaded into the program when the program is built. Static libraries are embedded as part of the Blue Gene/P executable that resides on the Front End Node. Example 8-1 illustrates how to create a static library on Blue Gene/P using the XL family of compilers.

Example 8-1 Static library creation using the XL compiler

```
# Compile with the XL compiler
/opt/ibmcmp/vac/bg/9.0/bin/bgxlc -c pi.c
/opt/ibmcmp/vac/bg/9.0/bin/bgxlc -c main.c
#
# Create library
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-ar rcs libpi.a pi.o
#
# Create executable
/opt/ibmcmp/vac/bg/9.0/bin/bgxlc -o pi main.o -L. -lpi
```

Example 8-2 shows the same procedure using the GNU collection of compilers.

Example 8-2 Static library creation using the GNU compiler

```
# Compile with the GNU compiler
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-gcc -c pi.c
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-gcc -c main.c
#
# Create library
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-ar rcs libpi.a pi.o
#
# Create executable
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-gcc -o pi main.o -L. -lpi
```

On the other hand, shared libraries are loaded at execution time and shared among different executables.

Note: -qnostaticlink, used with the C and C++ compilers, indicates to build a dynamic binary, but by default the static libgcc.a is linked in. To indicate that the shared version of libgcc should be linked in, also specify -qnostaticlink=libgcc, for example, /opt/ibmcmp/vacpp/bg/9.0/bin/bgxlc -o hello hello.c -qnostaticlink -qnostaticlink=libgcc

Example 8-3 Shared library creation using the XL compiler

```
# Use XL to create shared library
/opt/ibmcmp/vac/bg/9.0/bin/bgxlc -qpic -c libpi.c
/opt/ibmcmp/vac/bg/9.0/bin/bgxlc -qpic -c main.c
#
# Create shared library
/opt/ibmcmp/vac/bg/9.0/bin/bgxlc -qmkshrobj -qnostaticlink -Wl,-soname, \
   libpi.so.0 -o libpi.so.0.0 libpi.o
#
# Set up the soname
ln -sf libpi.so.0.0 libpi.so.0
#
# Create a linker name
ln -sf libpi.so.0 libpi.so
#
# Create executable
/opt/ibmcmp/vac/bg/9.0/bin/bgxlc -o pi main.o -L. -lpi \
  -qnostaticlink \
  -qnostaticlink=libgcc
```

Example 8-4 illustrates the same procedure with the GNU collection of compilers.

Example 8-4 Shared library creation using the GNU compiler

```
# Compile with the GNU compiler
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-gcc -fPIC -c libpi.c
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-gcc -fPIC -c main.c
#
# Create shared library
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-gcc -shared \
   -Wl,-soname,libpi.so.0 -o libpi.so.0.0 libpi.o -lc
#
# Set up the soname
ln -sf libpi.so.0.0 libpi.so.0
#
# Create a linker name
ln -sf libpi.so.0 libpi.so
#
# Create executable
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-gcc -o pi main.o -L. -lpi -dynamic
```

The command line option to the Blue Gene/P XL Fortran compiler to create a dynamic executable is -Wl,-dy. The order of the -Wl,-dy is significant. It must come before any -L and -l options, for example, to build a dynamic executable from the Fortran source file hello.f, run `/opt/ibmcmp/xlf/bg/11.1/bin/bgxlf -Wl,-dy -qpic -o hello hello.f`. Example 8-5 illustrates creating a shared library using the XL Fortran compiler.

Example 8-5 Fortran shared library creation

```
# Create the .o file with -qpic for position independent code.
/opt/ibmcmp/xlf/bg/11.1/bin/bgxlf90 -c foo.c -qpic

# Create the shared library using ld from the BGP toolchain.
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-ld -o libfoo.so foo.o -shared

# Or, depending on which functions are called, you may need to link in more libraries.
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-ld -o libfoo.so foo.o \
   -shared -L/opt/ibmcmp/xlf/bg/11.1/bglib -L/opt/ibmcmp/xlsmp/bg/1.7/bglib -lxlf90 -lxlsmp

# Create the executable. In this case -L. means search the current directory for libfoo.so.
/opt/ibmcmp/xlf/bg/11.1/bin/bgxlf90 -o main main.f90 -qpic -Wl,-dy -L. -lfoo
```

8.7 XL runtime libraries

The libraries listed in Table 8-2 are linked into your application automatically by the XL linker when you create your application.

MASS libraries: The exception to this statement is for the libmassv.a file (the Mathematical Acceleration Subsystem (MASS) libraries). This file must be explicitly specified on the linker command. See 8.8, "Mathematical Acceleration Subsystem libraries" on page 104 for information about the MASS libraries.

Table 8-2 XL static and dynamic libraries

File name	Description
libibmc++.a, libibmc++.so	IBM C++ library
libxlf90.a, libxlf90.so	IBM XLF run-time library
libxlfmath.a, libxlfmath.so	IBM XLF stubs for math routines in system library libm, for example, _sin() for sin(), _cos() for cos(), and so on
libxlfpmt4.a, libxlfpmt4.so	IBM XLF to be used with `-qautobdl=dbl4` (promote floating-point objects that are single precision)
libxlfpad.a, libxlfpad.so	IBM XLF run-time routines to be used with `-qautobdl=dblpad` (promote floating-point objects and pad other types if they can share storage with promoted objects)
libxlfpmt8.a, libxlfpmt8.so	IBM XLF run-time routines to be used with `-qautobdl=dbl8` (promote floating-point objects that are double precision)
libxlsmp.a, libxlsmp.so	IBM XL SMP runtime library functions
libxl.a	IBM low-level run-time library
libxlopt.a	IBM XL optimized intrinsic library ▶ Vector intrinsic functions ▶ BLASS routines
libmass.a	IBM XL MASS library: scalar intrinsic functions
libmassv.a	IBM XL MASSV library: vector intrinsic functions
ibxlomp_ser.a	IBM XL Open MP compatibility library

8.8 Mathematical Acceleration Subsystem libraries

The MASS consists of libraries of tuned mathematical intrinsic functions that are available in versions for the AIX and Linux machines, including the Blue Gene/P system. The MASS libraries provide improved performance over the standard mathematical library routines, are thread-safe, and support compilations in C, C++, and Fortran applications. For more information about MASS, refer to the Mathematical Acceleration Subsystem Web page at:

http://www-306.ibm.com/software/awdtools/mass/index.html

8.9 Engineering and Scientific Subroutine Library libraries

The Engineering and Scientific Subroutine Library (ESSL) for Linux on IBM POWER supports the Blue Gene/P system. ESSL provides over 150 math subroutines that have been specifically tuned for performance on the Blue Gene/P system. For more information about ESSL, refer to the Engineering Scientific Subroutine Library and Parallel ESSL Web page at:

http://www.ibm.com/systems/p/software/essl.html

> **Important:** When using IBM XL Fortran V11.1 for IBM System Blue Gene, customers must use ESSL V4.4. If an attempt is made to install a wrong mix of ESSL and XLF, the rpm installation fails with a dependency error message.

8.10 Configuring Blue Gene/P builds

When building an application in a cross-compile environment, such as Blue Gene/P, build tools, such as configure and make, sometimes compile and execute small code snippets to identify characteristics of the target platform as part of the build process. If these code snippets are compiled with a cross-compiler and then executed on the build machine instead of the target machine, the program might fail to execute or produce results that do not reflect the target machine. When that happens, the configure fails or does not configure as expected. To avoid this problem, the Blue Gene/P system provides a way to transparently run Blue Gene/P executables on a Blue Gene/P partition when executed on a Blue Gene/P Front End Node.

This feature was introduced in Blue Gene/P release V1R4M0.

Use of this feature requires some configuration by your Blue Gene/P administrator. The documentation for configuring this feature is in the *IBM System Blue Gene Solution: Blue Gene/P System Administration* which can be found at the following URL:
http://www.redbooks.ibm.com/abstracts/sg247417.html?Open

After the Blue Gene/P system is configured to support this feature, its operation is nearly transparent to developers. Developers must ensure that the small programs that are built during the configuration of the package are built using the Blue Gene/P compilers. Each configuration script is unique so general instructions for how to force the compiler cannot be provided, but the following example works for many packages:

```
$ ./configure CC=/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-gcc
```

The developer can verify that the system is set up and working properly by compiling a program using the Blue Gene/P cross-compiler and then executing the program on the Front End Node. Example 8-6 shows a sample program.

Example 8-6 Program to verify the Blue Gene/P loader configuration

```
#include <stdio.h>
#include <sys/utsname.h>

int main(int argc, char** argv)
{
  struct utsname uts;

  uname(&uts);
    printf("sizeof uts: %d\n", sizeof(uts));
    printf("sysname: %s\n", uts.sysname);
    printf("nodename: %s\n", uts.nodename);
    printf("release: %s\n", uts.release);
    printf("version: %s\n", uts.version);
    printf("machine: %s\n", uts.machine);
  if (strcmp(uts.sysname, "Linux") == 0) {
    printf("We are on Linux!\n");
  }
  else {
    printf("We are NOT on Linux!\n");
  }
   return 0;
}
```

To compile and run this program, run the following commands:

```
$ /bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-gcc -o test-uname
test-uname.c
$ ./test-uname
```

Code in the program loader on the Front End Node checks if the program was compiled for Blue Gene/P, and then rather than execute the program it invokes **submit** with the same program and arguments to run the program on the Blue Gene/P partition. In addition to remote program execution, all input and output from the program is handled transparently, including stdin, stdout, stderr, arguments, return codes, and signals.

You can set the HTC_SUBMIT_OPTS environment variable toset extra command-line options for the **submit** command. In particular, setting HTC_SUBMIT_OPTS to "--trace 3" will print out more tracing information that can be used to verify that the program is running on the Blue Gene/P hardware instead of natively. We describe the arguments to the **submit** program in 9.1.3, "submit" on page 141.

8.11 Python

Python is a dynamic, object-oriented programming language that can be used on Blue Gene/P in addition to C, C++ and Fortran. Version 2.6[1] of the Python interpreter compiled for Blue Gene/P is installed in /bgsys/drivers/ppcfloor/gnu-linux/bin. Example 8-7 illustrates how to invoke a simple Python program.

Example 8-7 How to invoke Python on Blue Gene/P

```
$ mpirun -partition MYPARTITION /bgsys/drivers/ppcfloor/gnu-linux/bin/python
test_array.py
```

Additional information about Python can be found at the following web sites:

- Python official site at

  http://www.python.org/

- Python Tutorial at

  http://docs.python.org/tut/tut.html

[1] In Blue Gene/P release V1R3M0, the Python version is 2.5.

- Python Library Reference at

  http://docs.python.org/lib/lib.html

- pyMPI at

  http://pympi.sourceforge.net/

8.12 Tuning your code for Blue Gene/P

In the sections that follow, we describe strategies that you can use to best exploit the SIMD capabilities of the Blue Gene/P 450 processor and the XL compilers' advanced instruction scheduling.

8.12.1 Using the compiler optimization options

The `-O3` compiler option provides a high level of optimization and automatically sets other options that are especially useful on the Blue Gene/P system. The `-qhot=simd` option enables SIMD vectorization of loops. It is enabled by default if you use `-O4`, `-O5`, or `-qhot`.

For more information about optimization options, see the following references:

- "Optimizing your applications" in the *XL C/C++ Programming Guide*, under Product Documentation on the following Web page

  http://www-306.ibm.com/software/awdtools/xlcpp/library/

- "Optimizing XL Fortran programs" in the *XL Fortran User Guide*, under Product Documentation on the following Web page

  http://www-306.ibm.com/software/awdtools/fortran/xlfortran/library/

8.12.2 Parallel Operations on the PowerPC 450

Similar to the Blue Gene/L system, floating-point instructions can operate simultaneously on the primary and secondary registers. Figure 8-1 on page 108 illustrates these registers.

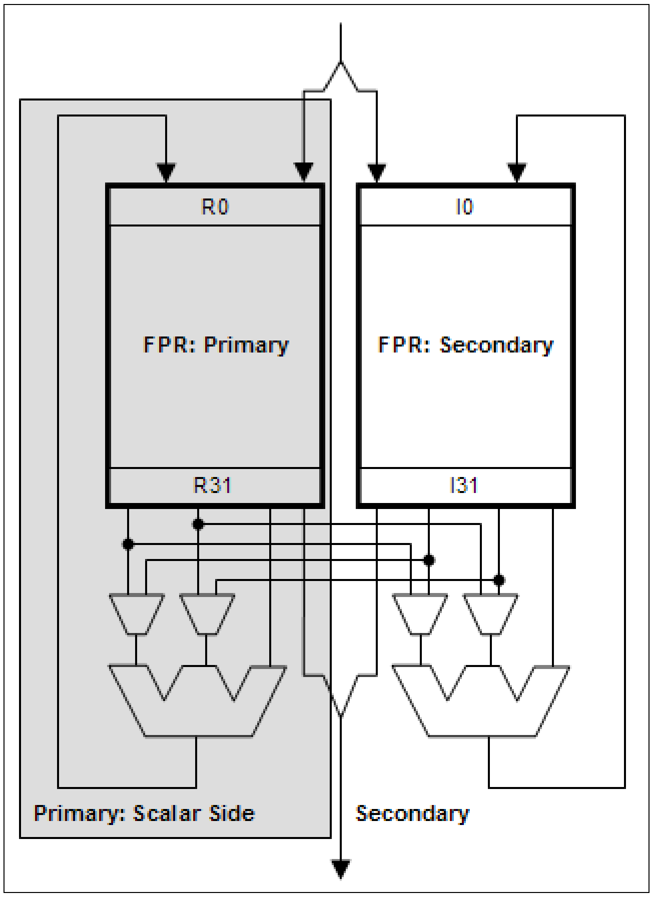


Figure 8-1 Blue Gene/P dual floating-point unit

The registers allow the PowerPC 450 processor to operate certain identical operations in parallel. Load/store instructions can also be issued with a single instruction. For more detailed information, see the white paper "Exploiting the Dual Floating Point Units in Blue Gene/L" on the Web at:

`http://www-01.ibm.com/support/docview.wss?uid=swg27007511`

The IBM XL compilers leverage this functionality under the following conditions:

- Parallel instructions are issued for load/store instructions if the alignment and size are aligned with *natural alignment*. This is 16 bytes for a pair of doubles, but only 8 bytes for a pair of floats.
- The compiler can issue parallel instructions when the application vectors have stride-one memory accesses. However, the compiler via IPA issues parallel instructions with non-stride-one data in certain loops, if it can be shown to improve performance.
- `-qhot=simd` is the default with `-qarch=450d`.
- `-O4` provides analysis at compile time with limited scope analysis and issuing parallel instructions (SIMD).
- `-O5` provides analysis for the entire program at link time to propagate alignment information. You must compile and link with `-O5` to obtain the full benefit.

8.12.3 Using single-instruction multiple-data instructions in applications

On the Blue Gene/P system, normal PowerPC assembler instructions use the primary floating-point pipe. To enable instructions in parallel, special assembly instructions must be generated using the following compiler options:

-qarch=450d This flag in the compiler enables parallel instructions to use the primary and secondary registers (SIMD instructions). See Figure 8-1 on page 108.

-qtune=450 This flag optimizes code for the IBM 450 microprocessors, as previously mentioned.

-O2 and up This option in the compiler enables parallel instructions.

The XL compiler optimizer consists of two major parts:

- Toronto Portable Optimizer (TPO) for high-level inter-procedural optimization
- Toronto Optimizing Back End with Yorktown (TOBEY) for low-level back-end optimization

SIMD instructions occur in both optimizers. SIMD instruction generation in TOBEY is activated by default for `-O2` and up. SIMD generation in TPO is added when using `-qhot`, `-O4`, or `-O5`. Specifically, the `-qhot` option adds SIMD generation, but options `-O4` and `-O5` automatically call `-qhot`. For more details, see the C, C++, and Fortran manuals on the Web at the following addresses:

- XL C/C++

  http://www.ibm.com/software/awdtools/xlcpp/library/
- XL Fortran

  http://www-306.ibm.com/software/awdtools/fortran/xlfortran/library/

For some applications, the compiler generates more efficient code without the TPO SIMD level. If you have statically allocated array, and a loop in the same routine, call TOBEY with `-qhot` or `-O4`. Nevertheless, on top of SIMD generation from TOBEY, with `-qhot`, optimizations are enabled that can alter the semantic of the code and on rare occasions can generate less efficient code. Also, with `-qhot=nosimd`, you can suppress some of these optimizations.

To use the SIMD capabilities of the XL compilers:

1. Start to compile:

   ```
   -O3 -qarch=450d -qtune=450
   ```

   We recommend that you use `-qarch=450d -qtune=450`, in this order. The compiler only generates SIMD instructions from `-O2` and up.

2. Increase the optimization level, and call the high-level inter-procedural optimizer:
   - `-O5` (link time, whole-program analysis, and SIMD instruction)
   - `-O4` (compile time, limited scope analysis, and SIMD instructions)
   - `-O3 -qhot=simd` (compile time, less optimization, and SIMD instructions)

3. Tune your program:

   a. Check the SIMD instruction generation in the object code listing (`-qsource -qlist`).

   b. Use compiler feedback (`-qreport -qhot`) to guide you.

c. Help the compiler with extra information (directives and pragmas):

   - Enter the alignment information with directives and pragmas.
     In C, enter:

     ```
     __alignx
     ```

     In Fortran, enter:

     ```
     ALIGNX
     ```

   - Tell the compiler that data accessed through pointers is disjoint.
     In C, enter:

     ```
     #pragma disjoint
     ```

   - Use constant loop bound, `#define`, when possible.
   - Use data flow instead of control flow.
   - Use select instead of if/then/else. Use macros instead of calls.
   - Tell the compiler not to generate SIMD instructions if it is not profitable (trip count low).
     In C, enter:

     ```
     #pragma nosimd
     ```

     In Fortran, enter the following line just before the loop:

     ```
     !IBM* NOSIMD
     ```

– Many applications can require modifying algorithms. The previous bullet, which explains how *not* to generate SIMD instructions, gives constructs that might help to modify the code. Here are hints to use when modifying your code:

   - Loops must be stride one accesses.
   - For function calls in loop:

     – Try to inline the calls.
     – Loop with if statement.
     – Use pointer and aliasing.
     – Use integer operations.

   - Assumed shape arrays in Fortran 90 can hurt enabling SIMD instructions.

– Generate compiler diagnostics to help you modify and understand how the compiler is optimizing sections of your applications:

   The `-qreport` compiler option generates a diagnostic report about SIMD instruction generation. To analyze the generated code and the use of quadword loads and stores, you must look at the pseudo assembler code within the object file listing. The diagnostic report provides two types of information about SIMD generation:

   - Information on success

     ```
     (simdizable) [feature][version]
     ```

     `[feature]` further characterizes the simdizable loop:

     misalign (compile time store)
     Refers to a simdizable loop with misaligned accesses.

     shift (4 compile time)
     Refers to a simdizable loop with 4 stream shift inserted. Shift refers to the number of misaligned data references that were found. It has a performance impact because these loops must be loaded across, and then an extra select instruction must be inserted.

priv Indicates that the compiler generated a private variable. priv means a private variable was found. In general, it should have no performance impact, but in practice it sometimes does.

reduct Indicates that a simdizable loop has a reduction construct. reduct means that a reduction was found. It is simdized using partial sums, which must be added at the end of the loop.

`[version]` further characterizes if and why versioned loops were created:

relative align Indicates the version for relative alignment. The compiler generated a test and two versions.

trip count Versioned for a short run-time trip count.

- Information on failure
  - In case of misalignment: misalign(...):
    - Non-natural: non-naturally aligned accesses
    - Run time: run-time alignment
  - About the structure of the loop:
    - Irregular loop structure (while-loop).
    - Contains control flow: if/then/else.
    - Contains function call: Function call bans SIMD instructions.
    - Trip count too small.
  - About dependences: dependence due to aliasing
  - About array references:
    - Access not stride one
    - Memory accesses with unsupported alignment
    - Contains run-time shift
  - About pointer references: Non-normalized pointer accesses

8.13 Tips for optimizing applications

The following sections are an excerpt from *IBM System Blue Gene Solution: Application Development*, SG24-7179, tailored to the Blue Gene/P system. They provide useful tips on how to optimize certain constructs in your code.

Identifying performance bottlenecks

The first step in applications tuning requires identifying where the bottlenecks are located in the entire application. If multiple locations are identified as potential bottlenecks, prioritization is required. Figure 8-2 on page 112 illustrates the initial set of steps required to identify where the bottle bottlenecks might be located.

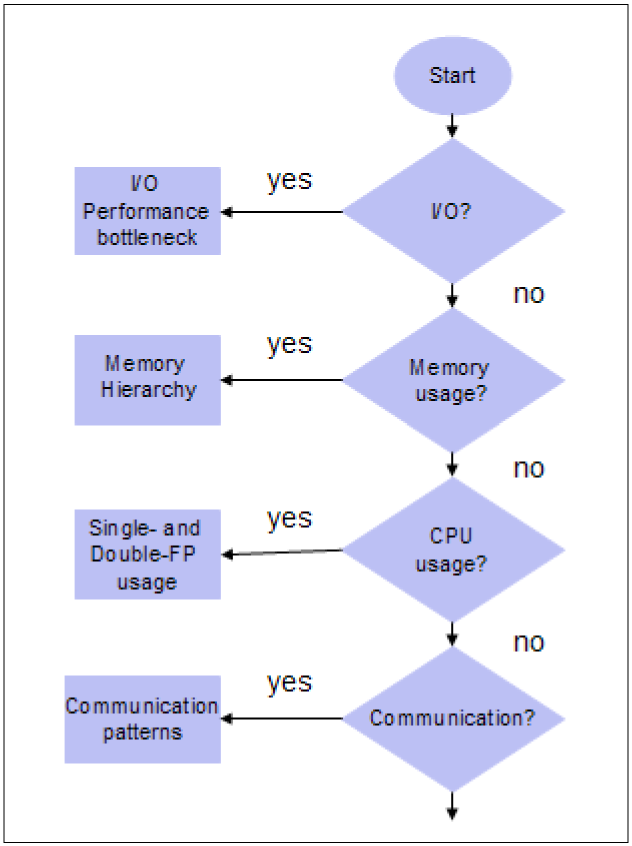


Figure 8-2 Steps to identify performance bottlenecks

Proper I/O utilization tends to be a problem, and in many applications I/O optimization is required. Identify if that is the case for your application. The IBM toolkit for Blue Gene/P provides the Modular I/O (MIO) library that can be used for an applications-level I/O performance improvement (See *IBM System Blue Gene Solution: Performance Analysis Tools*, REDP-4256). Memory utilization on Blue Gene/P involves optimal utilization of the memory hierarchy. This needs to be coordinated with the double floating-point unit to leverage the execution of instructions in parallel. As part of the IBM toolkit Xprofiler helps analyze applications by collecting data using the -pg compiler option to identify functions that are most CPU intensive. gmon profiler also provides similar information. Appendix H, “Use of GNU profiling tool on Blue Gene/P” on page 361 provides additional information about gmon. The IBM toolkit provides the MPI profiler and a tracing library for MPI programs.

Structuring data in adjacent pairs

The Blue Gene/P 450d processor’s dual FPU includes special instructions for parallel computations. The compiler tries to pair adjacent single-precision or double-precision floating-point values to operate on them in parallel. Therefore, you can accelerate computations by defining data objects that occupy adjacent memory blocks and are naturally aligned. These include arrays or structures of floating-point values and complex data types.

Whether you use an array, a structure, or a complex scalar, the compiler searches for sequential pairs of data for which it can generate parallel instructions, for example, using the C code in Example 8-8 on page 113, each pair of elements in a structure can be operated on in parallel.

Example 8-8 Adjacent paired data

```
struct quad {
   double a, b, c, d;
};

struct quad x, y, z;

void foo()
{
   z.a = x.a + y.a;
   z.b = x.b + y.b;/* can load parallel (x.a,x.b), and (y.a, y.b), do parallel add, and 
store parallel (z.a, z.b) */

   z.c = x.c + y.c;
   z.d = x.d + y.d;/* can load parallel (x.c,x.d), and (y.c, y.d), do parallel add, and 
store parallel (z.c, z.d) */
}
```

The advantage of using complex types in arithmetic operations is that the compiler automatically uses parallel add, subtract, and multiply instructions when complex types appear as operands to addition, subtraction, and multiplication operators. Furthermore, the data that you provide does not need to represent complex numbers. In fact, both elements are represented internally as two real values. See “Complex type manipulation functions” on page 121, for a description of the set of built-in functions that are available for the Blue Gene/P system. These functions are especially designed to efficiently manipulate complex-type data and include a function to convert non-complex data to complex types.

Using vectorizable basic blocks

The compiler schedules instructions most efficiently within *extended basic blocks*. Extended basic blocks are code sequences that can contain conditional branches but have no entry points other than the first instruction. Specifically, minimize the use of branching instructions for:

- Handling special cases, such as the generation of not-a-number (NaN) values.
- C/C++ error handling that sets a value for `errno`.

  To explicitly inform the compiler that none of your code will set `errno`, you can compile with the `-qignerrno` compiler option (automatically set with `-O3`).
- C++ exception handlers.

  To explicitly inform the compiler that none of your code will throw any exceptions, and therefore, that no exception-handling code must be generated, you can compile with the `-qnoeh` compiler option.

In addition, the optimal basic blocks remove dependencies between computations, so that the compiler views each statement as entirely independent. You can construct a basic block as a series of independent statements or as a loop that repeatedly computes the same basic block with different arguments.

If you specify the `-qhot=simd` compilation option, along with a minimum optimization level of `-O2`, the compiler can then vectorize these loops by applying various transformations, such as unrolling and software pipelining. See “Removing possibilities for aliasing (C/C++)” on page 114, for additional strategies for removing data dependencies.

Using inline functions

An inline function is expanded in any context in which it is called. This expansion avoids the normal performance overhead associated with the branching for a function call, and it allows functions to be included in basic blocks. The XL C/C++ and Fortran compilers provide several options for inlining.

The following options instruct the compiler to automatically inline all functions it deems appropriate:

- XL C/C++:
  - `-O` through `-O5`
  - `-qipa`
- XL Fortran:
  - `-O4` or `-O5`
  - `-qipa`

With the following options, you can select or name functions to be inlined:

- XL C/C++ :
  - `-qinline`
  - `-Q`
- XL Fortran:

  `-Q`

In C/C++, you can also use the standard `inline` function specifier or the `__attribute__(always_inline)` extension in your code to mark a function for inlining.

> **Using inlining:** Do not overuse inlining because of the limits on how much inlining can be done. Mark the most important functions.

For more information about the various compiler options for controlling function inlining, see the following publications:

- "XL C/C++ Compiler Reference"

  http://www-306.ibm.com/software/awdtools/xlcpp/library/
- "XL Fortran User Guide"

  http://www-306.ibm.com/software/awdtools/fortran/xlfortran/library/

  Also available from this Web address, the "XL C/C++ Language Reference" provides information about the different variations of the `inline` keyword supported by XL C and C++, as well as the inlining function attribute extensions.

Removing possibilities for aliasing (C/C++)

When you use pointers to access array data in C/C++, the compiler cannot assume that the memory accessed by pointers is not altered by other pointers that refer to the same address, for example, if two pointer input parameters share memory, the instruction to store the second parameter can overwrite the memory read from the first load instruction, which means that, after a store for a pointer variable, any load from a pointer must be reloaded. Consider the code in Example 8-9.

Example 8-9 Sample code

```
int i = *p;
*q = 0;
```

```
j = *p;
```

If *q aliases *p, then the value must be reloaded from memory. If *q does not alias *p, the old value that is already loaded into i can be used.

To avoid the overhead of reloading values from memory every time they are referenced in the code, and to allow the compiler to simply manipulate values that are already resident in registers, you can use several strategies. One approach is to assign input array element values to local variables and perform computations only on the local variables, as shown in Example 8-10.

Example 8-10 Array parameters assigned to local variables

```
#include <math.h>
void reciprocal_roots (const double* x, double* f)
{
       double x0 = x[0] ;
       double x1 = x[1] ;
       double r0 = 1.0/sqrt(x0) ;
       double r1 = 1.0/sqrt(x1) ;
       f[0] = r0 ;
       f[1] = r1 ;
}
```

If you are certain that two references do not share the same memory address, another approach is to use the #pragma disjoint directive. This directive asserts that two identifiers do not share the same storage, within the scope of their use. Specifically, you can use pragma to inform the compiler that two pointer variables do not point to the same memory address. The directive in Example 8-11 indicates to the compiler that the pointers-to-arrays of double x and f do not share memory.

Example 8-11 The #pragma disjoint directive

```
__inline void ten_reciprocal_roots (double* x, double* f)
{
#pragma disjoint (*x, *f)
int i;
     for (i=0; i < 10; i++)
    f[i]= 1.0 / sqrt (x[i]);
}
```

Important: The correct functioning of this directive requires that the two pointers be disjoint. If they are not, the compiled program cannot run correctly.

Structure computations in batches

Floating-point operations are pipelined in the 450 processor, so that one floating-point calculation is performed per cycle, with a latency of approximately five cycles. Therefore, to keep the 450 processor’s floating-point units busy, organize floating-point computations to perform step-wise operations in batches - for example, arrays of five elements and loops of five iterations. For the 450d, which has two FPUs, use batches of ten, for example, with the 450d, at high optimization, the function in Example 8-12 on page 116 should perform ten parallel reciprocal roots in about five cycles more than a single reciprocal root. This is because the compiler performs two reciprocal roots in parallel and then uses the “empty” cycles to run four more parallel reciprocal roots.

Example 8-12 Function to calculate reciprocal roots for arrays of 10 elements

```
__inline void ten_reciprocal_roots (double* x, double* f)
{
#pragma disjoint (*x, *f)

    int i;
    for (i=0; i < 10; i++)
   f[i]= 1.0 / sqrt (x[i]);
}
```

The definition in Example 8-13 shows “wrapping” the inlined, optimized `ten_reciprocal_roots` function, in Example 8-12, inside a function that allows you to pass in arrays of any number of elements. This function then passes the values in batches of ten to the `ten_reciprocal_roots` function and calculates the remaining operations individually.

Example 8-13 Function to pass values in batches of ten

```
static void unaligned_reciprocal_roots (double* x, double* f, int n)
{
#pragma disjoint (*x, *f)
    while (n >= 10) {
   ten_reciprocal_roots (x, f);
   x += 10;
   f += 10;
    }
    /* remainder */
    while (n > 0) {
   *f = 1.0 / sqrt (*x);
   f++, x++;
    }
}
```

Checking for data alignment

Floating-point data alignment requirements are based on the size of the data: Four-byte data must be word aligned, 8-byte data must be double-word aligned, and 16-byte data must be quad-word aligned. If data that is not properly aligned is accessed or modified, the hardware generates an alignment exception. The user can determine how alignment exceptions are to be handled by the setting of the environment variable BG_MAXALIGNEXP. If this variable is not set, the kernel can handle up to 1000 alignment exceptions, and after this amount of time, a SIGBUS signal is raised, the program ends, and generates a core file. The core file provides information about the instruction address and stack trace where the alignment exception occurred. Setting BG_MAXALIGNEXP=-1 indicates that all alignment exceptions are to be handled. Setting BG_MAXALIGNEXP=0 indicates that no alignment exceptions are to be handled. Because alignment exceptions can cause a severe performance penalty, this technique can be used to find code that is taking alignment exceptions unexpectedly.

The compiler does not generate these parallel load and store instructions unless it is sure that it is safe to do so. For non-pointer local and global variables, the compiler knows when this is safe. To allow the compiler to generate these parallel loads and stores for accesses through pointers, include code that tests for correct alignment and that gives the compiler hints.

To test for alignment, first create one version of a function that asserts the alignment of an input variable at that point in the program flow. You can use the C/C++ `__alignx` built-in function or the Fortran ALIGNX function to inform the compiler that the incoming data is correctly aligned according to a specific byte boundary, so it can efficiently generate loads and stores.

The function takes two arguments. The first argument is an integer constant that expresses the number of alignment bytes (must be a positive power of two). The second argument is the variable name, typically a pointer to a memory address.

Example 8-14 shows the C/C++ prototype for the function.

Example 8-14   C/C++ prototype

```
extern
#ifdef __cplusplus
"builtin"
#endif
void __alignx (int n, const void *addr)
```

Example 8-14, *n* is the number of bytes, for example, `__alignx(16, y)` specifies that the address *y* is 16-byte aligned.

In Fortran95, the built-in subroutine is ALIGNX(K,M), where K is of type INTEGER(4), and M is a variable of any type. When M is an integer pointer, the argument refers to the address of the pointee.

Example 8-15 asserts that the variables *x* and *f* are aligned along 16-byte boundaries.

Example 8-15   Using the __alignx built-in function

```
#include <math.h>
#include <builtins.h>
__inline void aligned_ten_reciprocal_roots (double* x, double* f)
{
#pragma disjoint (*x, *f)
int i;
    __alignx (16, x);
    __alignx (16, f);
    for (i=0; i < 10; i++)
   f[i]= 1.0 / sqrt (x[i]);
}
```

The __alignx function: The `__alignx` function does not perform any alignment. It merely informs the compiler that the variables are aligned as specified. If the variables are not aligned correctly, the program does not run properly.

After you create a function to handle input variables that are correctly aligned, you then can create a function that tests for alignment and then calls the appropriate function to perform the calculations. The function in Example 8-16 checks to see whether the incoming values are correctly aligned. Then it calls the “aligned” (Example 8-15) or “unaligned” (Example 8-12 on page 116) version of the function according to the result.

Example 8-16   Function to test for alignment

```
void reciprocal_roots (double *x, double *f, int n)
{
    /* are both x & f 16 byte aligned? */
    if ( ((((int) x) | ((int) f)) & 0xf) == 0) /* This could also be done as:
                                   if (((int) x % 16 == 0) && ((int) f % 16) == 0) */
   aligned_ten_reciprocal_roots (x, f, n);
   else
   ten_reciprocal_roots (x, f, n);
}
```

The alignment test in Example 8-16 on page 117 provides an optimized method of testing for 16-byte alignment by performing a bit-wise OR on the two incoming addresses and testing whether the lowest four bits are 0 (that is, 16-byte aligned).

Using XL built-in floating-point functions for Blue Gene/P

The XL C/C++ and Fortran95 compilers include a large set of built-in functions that are optimized for the PowerPC architecture. For a full description of them, refer to the following documents:

- Appendix B: “Built-In Functions” in *XL C/C++ Compiler Reference*

  http://www-306.ibm.com/software/awdtools/xlcpp/library/

- “Intrinsic Procedures” in *XL Fortran Language Reference*

  http://www-306.ibm.com/software/awdtools/fortran/xlfortran/library/

On the Blue Gene/P system, the XL compilers provide a set of built-in functions that are specifically optimized for the PowerPC 450d dual FPU. These built-in functions provide an almost one-to-one correspondence with the dual floating-point instruction set.

All of the C/C++ and Fortran built-in functions operate on complex data types, which have an underlying representation of a two-element array, in which the real part represents the *primary* element and the imaginary part represents the *second* element. The input data that you provide does not need to represent complex numbers. In fact, both elements are represented internally as two real values. None of the built-in functions performs complex arithmetic. A set of built-in functions designed to efficiently manipulate complex-type variables is also available.

The Blue Gene/P built-in functions perform several types of operations, as explained in the following paragraphs.

Parallel operations perform SIMD computations on the primary and secondary elements of one or more input operands. They store the results in the corresponding elements of the output. As an example, Figure 8-3 illustrates how a parallel-multiply operation is performed.

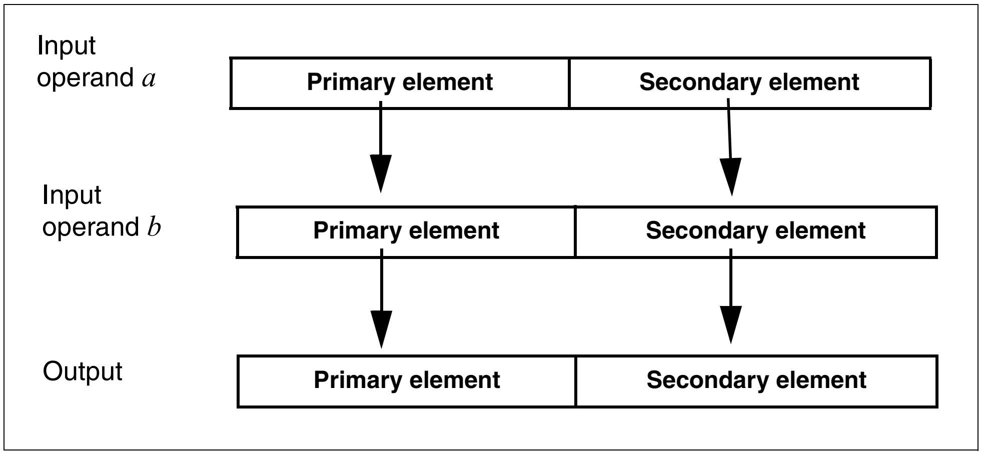


Figure 8-3 Parallel operations

Cross operations perform SIMD computations on the opposite primary and secondary elements of one or more input operands. They store the results in the corresponding elements in the output. As an example, Figure 8-4 on page 119 illustrates how a cross-multiply operation is performed.

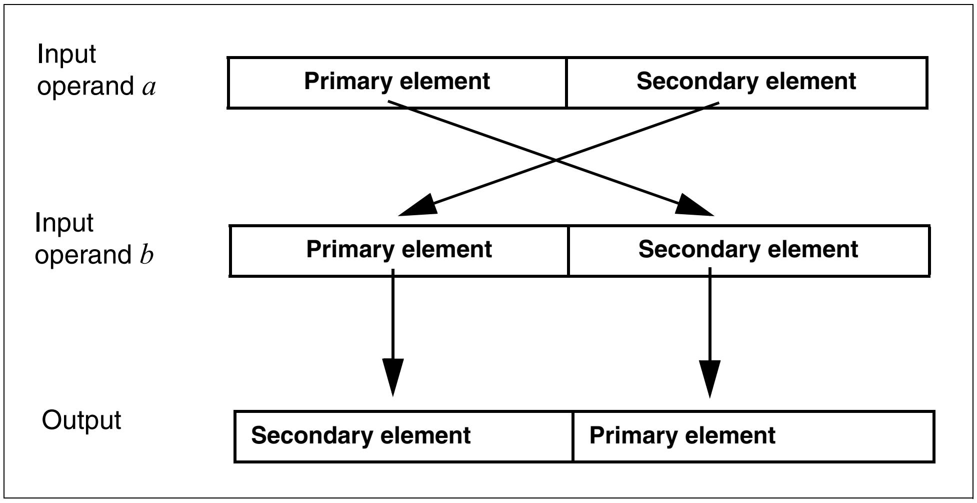


Figure 8-4 Cross-multiply operations

Copy-primary operations perform SIMD computation between the corresponding primary and secondary elements of two input operands, where the primary element of the first operand is replicated to the secondary element. As an example, Figure 8-5 illustrates how a cross-primary-multiply operation is performed.

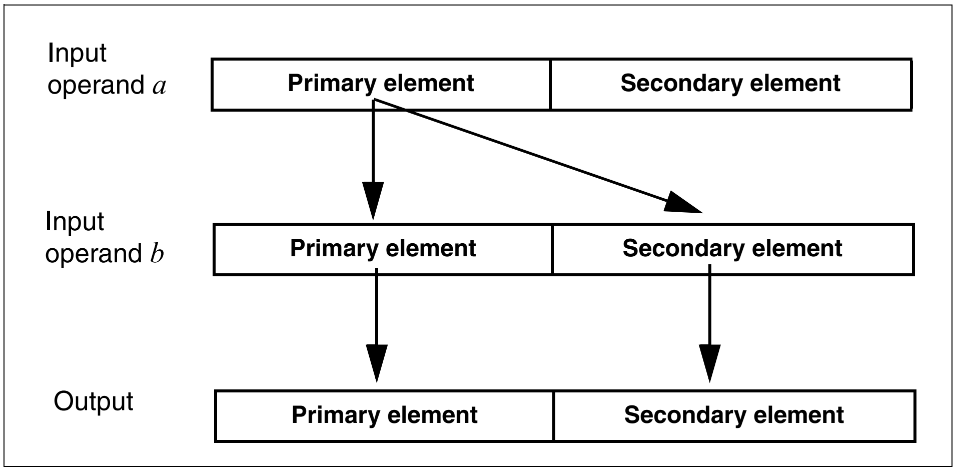


Figure 8-5 Copy-primary multiply operations

Copy-secondary operations perform SIMD computation between the corresponding primary and secondary elements of two input operands, where the secondary element of the first operand is replicated to the primary element. As an example, Figure 8-6 illustrates how a cross-secondary multiply operation is performed.

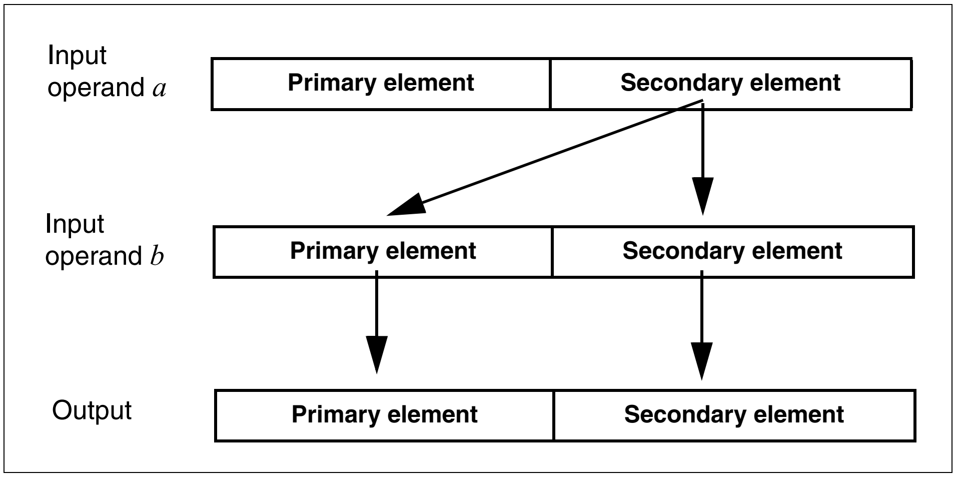


Figure 8-6 Copy-secondary multiply operations

In *cross-copy operations*, the compiler crosses either the primary or secondary element of the first operand, so that copy-primary and copy-secondary operations can be used interchangeably to achieve the same result. The operation is performed on the total value of the first operand. As an example, Figure 8-7 illustrates the result of a cross-copy multiply operation.

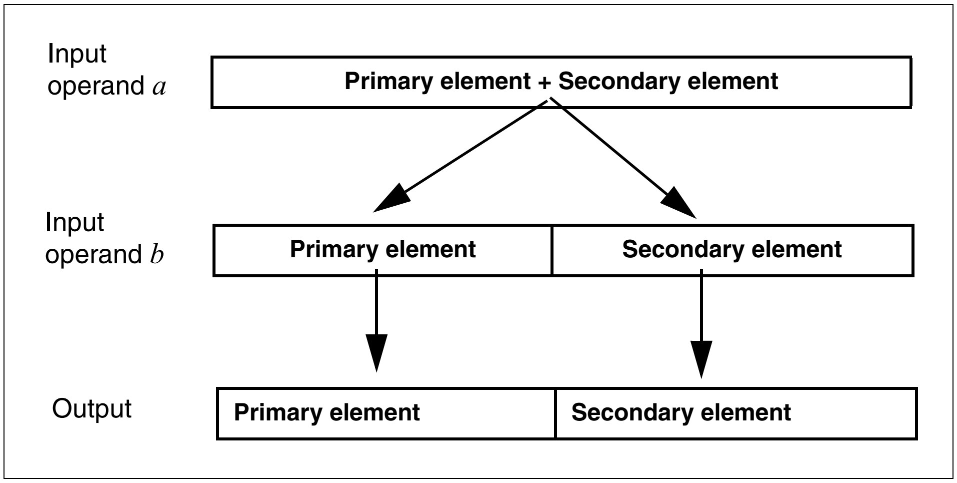


Figure 8-7 Cross-copy multiply operations

In the following paragraphs, we describe the available built-in functions by category. For each function, the C/C++ prototype is provided. In C, you do not need to include a header file to obtain the prototypes. The compiler includes them automatically. In C++, you must include the header file builtins.h.

Fortran does not use prototypes for built-in functions. Therefore, the interfaces for the Fortran95 functions are provided in textual form. The function names *omit* the double underscore (__) in Fortran95.

All of the built-in functions, with the exception of the complex type manipulation functions, require compilation under `-qarch=450d`. This is the default setting on the Blue Gene/P system.

To help clarify the English description of each function, the following notation is used:

```
element(variable)
```

Here `element` represents one of *primary* or *secondary*, and `variable` represents input variable *a*, *b*, or *c*, and the output variable *result*, for example, consider the following formula:

```
primary(result) = primary(a) + primary(b)
```

This formula indicates that the primary element of input variable *a* is added to the primary element of input variable *b* and stored in the primary element of the result.

To optimize your calls to the Blue Gene/P built-in functions, follow the guidelines that we provide in 8.12, “Tuning your code for Blue Gene/P” on page 107. Using the `alignx` built-in function (described in “Checking for data alignment” on page 116) and specifying the `disjoint` pragma (described in “Removing possibilities for aliasing (C/C++)” on page 114) are recommended for code that calls any of the built-in functions.

Complex type manipulation functions

Complex type manipulation functions, listed in Table 8-3, are useful for efficiently manipulating complex data types. Using these functions, you can automatically convert real floating-point data to complex types and extract the real (primary) and imaginary (secondary) parts of complex values.

Table 8-3 Complex type manipulation functions

Function	**Convert dual reals to complex (single-precision): __cmplxf**
Purpose	Converts two single-precision real values to a single complex value. The real *a* is converted to the primary element of the return value, and the real *b* is converted to the secondary element of the return value.
Formula	primary(result) =a secondary(result) = b
C/C++ prototype	float _Complex __cmplxf (float a, float b);
Fortran descriptions	CMPLXF(A,B) where A is of type REAL(4) where B is of type REAL(4) result is of type COMPLEX(4)
Function	**Convert dual reals to complex (double-precision): __cmplx**
Purpose	Converts two double-precision real values to a single complex value. The real *a* is converted to the primary element of the return value, and the real *b* is converted to the secondary element of the return value.
Formula	primary(result) =a secondary(result) = b
C/C++ prototype	double _Complex __cmplx (double a, double b); long double _Complex __cmplxl (long double a, long double b);[a]
Fortran descriptions	CMPLX(A,B) where A is of type REAL(8) where B is of type REAL(8) result is of type COMPLEX(8)

Function	**Extract real part of complex (single-precision): __crealf**
Purpose	Extracts the primary part of a single-precision complex value *a*, and returns the result as a single real value.
Formula	result =primary(a)
C/C++ prototype	float __crealf (float _Complex a);
Fortran descriptions	CREALF(A) where A is of type COMPLEX(4) result is of type REAL(4)
Function	**Extract real part of complex (double-precision): __creal, __creall**
Purpose	Extracts the primary part of a double-precision complex value *a*, and returns the result as a single real value.
Formula	result =primary(a)
C/C++ prototype	double __creal (double _Complex a); long double __creall (long double _Complex a);[a]
Fortran descriptions	CREAL(A) where A is of type COMPLEX(8) result is of type REAL(8) CREALL(A) where A is of type COMPLEX(16) result is of type REAL(16)
Function	**Extract imaginary part of complex (single-precision): __cimagf**
Purpose	Extracts the secondary part of a single-precision complex value *a*, and returns the result as a single real value.
Formula	result =secondary(a)
C/C++ prototype	float __cimagf (float _Complex a);
Fortran descriptions	CIMAGF(A) where A is of type COMPLEX(4) result is of type REAL(4)
Function	**Extract imaginary part of complex (double-precision): __cimag, __cimagl**
Purpose	Extracts the imaginary part of a double-precision complex value *a*, and returns the result as a single real value.
Formula	result =secondary(a)
C/C++ prototype	double __cimag (double _Complex a); long double __cimagl (long double _Complex a);[a]
Fortran descriptions	CIMAG(A) where A is of type COMPLEX(8) result is of type REAL(8) CIMAGL(A) where A is of type COMPLEX(16) result is of type REAL(16)

a. 128-bit C/C++ long double types are not supported on Blue Gene/L. Long doubles are treated as regular double-precision longs.

Load and store functions

Table 8-4 lists and explains the various parallel load and store functions that are available.

Table 8-4 Load and store functions

Function	Parallel load (single-precision): __lfps
Purpose	Loads parallel single-precision values from the address of *a*, and converts the results to double-precision. The first word in *address(a)* is loaded into the primary element of the return value. The next word, at location *address(a)*+4, is loaded into the secondary element of the return value.
Formula	primary(result) = a[0] secondary(result) = a[1]
C/C++ prototype	double _Complex __lfps (float * a);
Fortran description	LOADFP(A) where A is of type REAL(4) result is of type COMPLEX(8)
Function	**Cross load (single-precision): __lfxs**
Purpose	Loads single-precision values that have been converted to double-precision, from the address of *a*. The first word in *address(a)* is loaded into the secondary element of the return value. The next word, at location *address(a)*+4, is loaded into the primary element of the return value.
Formula	primary(result) = a[1] secondary(result) = a[0]
C/C++ prototype	double _Complex __lfxs (float * a);
Fortran description	LOADFX(A) where A is of type REAL(4) result is of type COMPLEX(8)
Function	**Parallel load: __lfpd**
Purpose	Loads parallel values from the address of *a*. The first word in *address(a)* is loaded into the primary element of the return value. The next word, at location *address(a)*+8, is loaded into the secondary element of the return value.
Formula	primary(result) = a[0] secondary(result) = a[1]
C/C++ prototype	double _Complex __lfpd(double* a);
Fortran description	LOADFP(A) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross load: __lfxd**
Purpose	Loads values from the address of *a*. The first word in *address(a)* is loaded into the secondary element of the return value. The next word, at location *address(a)*+8, is loaded into the primary element of the return value.
Formula	primary(result) = a[1] secondary(result) = a[0]

C/C++ prototype	double _Complex __lfxd (double * a);
Fortran description	LOADFX(A) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Parallel store (single-precision): __stfps**
Purpose	Stores in parallel double-precision values that have been converted to single-precision, into *address(b)*. The primary element of *a* is converted to single-precision and stored as the first word in *address(b).* The secondary element of *a* is converted to single-precision and stored as the next word at location *address(b)*+4.

Formula	b[0] = primary(a)
	b[1]= secondary(a)

C/C++ prototype	void __stfps (float * b, double _Complex a);
Fortran description	STOREFP(B, A) where B is of type REAL(4) A is of type COMPLEX(8) result is none
Function	**Cross store (single-precision): __stfxs**
Purpose	Stores double-precision values that have been converted to single-precision, into *address(b)*. The secondary element of *a* is converted to single-precision and stored as the first word in *address(b).* The primary element of *a* is converted to single-precision and stored as the next word at location *address(b)*+4.

Formula	b[0] = secondary(a)
	b[1] = primary(a)

C/C++ prototype	void __stfxs (float * b, double _Complex a);
Fortran description	STOREFX(B, A) where B is of type REAL(4) A is of type COMPLEX(8) result is none
Function	**Parallel store: __stfpd**
Purpose	Stores in parallel values into *address(b)*. The primary element of *a* is stored as the first double word in *address(b).* The secondary element of *a* is stored as the next double word at location *address(b)*+8.

Formula	b[0] = primary(a)
	b[1] = secondary(a)

C/C++ prototype	void __stfpd (double * b, double _Complex a);
Fortran description	STOREFP(B, A) where B is of type REAL(8) A is of type COMPLEX(8) result is none

Function	**Cross store: __stfxd**
Purpose	Stores values into *address(b)*. The secondary element of *a* is stored as the first double word in *address(b).* The primary element of *a* is stored as the next double word at location *address(b)*+8.

Formula	b[0] = secondary(a)
	b[1] = primary(a)

C/C++ prototype	void __stfxd (double * b, double _Complex a);
Fortran description	STOREFP(B, A) where B is of type REAL(8) A is of type COMPLEX(8) result is none
Function	**Parallel store as integer: __stfpiw**
Purpose	Stores in parallel floating-point double-precision values into *b* as integer words. The lower-order 32 bits of the primary element of *a* are stored as the first integer word in *address(b).* The lower-order 32 bits of the secondary element of *a* are stored as the next integer word at location *address(b)*+4. This function is typically preceded by a call to the `__fpctiw` or `__fpctiwz` built-in functions, described in "Unary functions" on page 126, which perform parallel conversion of dual floating-point values to integers.

Formula	b[0] = primary(a)
	b[1] = secondary(a)

C/C++ prototype	void __stfpiw (int * b, double _Complex a);
Fortran description	STOREFP(B, A) where B is of type INTEGER(4) A is of type COMPLEX(8) result is none

Move functions

Table 8-5 lists and explains the parallel move functions that are available.

Table 8-5 Move functions

Function	**Cross move: __fxmr**
Purpose	Swaps the values of the primary and secondary elements of operand *a*.

Formula	primary(result) = secondary(a)
	secondary(result) = primary(a)

C/C++ prototype	double _Complex __fxmr (double _Complex a);
Fortran description	FXMR(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)

Arithmetic functions

In the following sections, we describe all the arithmetic built-in functions, categorized by their number of operands.

Unary functions

Unary functions, listed in Table 8-6, operate on a single input operand.

Table 8-6 Unary functions

Function	**Parallel convert to integer: __fpctiw**
Purpose	Converts in parallel the primary and secondary elements of operand *a* to 32-bit integers using the current rounding mode. After a call to this function, use the `__stfpiw` function to store the converted integers in parallel, as explained in "Load and store functions" on page 123.

Formula	primary(result) = primary(a)
	secondary(result) = secondary(a)

C/C++ prototype	double _Complex __fpctiw (double _Complex a);
Fortran purpose	FPCTIW(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel convert to integer and round to zero: __fpctiwz**
Purpose	Converts in parallel the primary and secondary elements of operand *a* to 32-bit integers and rounds the results to zero. After a call to this function, use the `__stfpiw` function to store the converted integers in parallel, as explained in "Load and store functions" on page 123.

Formula	primary(result) = primary(a)
	secondary(result) = secondary(a)

C/C++ prototype	double _Complex __fpctiwz(double _Complex a);
Fortran description	FPCTIWZ(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel round double-precision to single-precision: __fprsp**
Purpose	Rounds in parallel the primary and secondary elements of double-precision operand *a* to single precision.

Formula	primary(result) = primary(a)
	secondary(result) = secondary(a)

C/C++ prototype	double _Complex __fprsp (double _Complex a);
Fortran description	FPRSP(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel reciprocal estimate: __fpre**
Purpose	Calculates in parallel double-precision estimates of the reciprocal of the primary and secondary elements of operand *a*.

Formula	primary(result) = primary(a)
	secondary(result) = secondary(a)

C/C++ prototype	double _Complex __fpre(double _Complex a);
Fortran description	FPRE(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel reciprocal square root: __fprsqrte**
Purpose	Calculates in parallel double-precision estimates of the reciprocals of the square roots of the primary and secondary elements of operand *a*.

Formula	primary(result) = primary(a)
	secondary(result) = secondary(a)

C/C++ prototype	double _Complex __fprsqrte (double _Complex a);
Fortran description	FPRSQRTE(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel negate: __fpneg**
Purpose	Calculates in parallel the negative values of the primary and secondary elements of operand *a*.

Formula	primary(result) = primary(a)
	secondary(result) = secondary(a)

C/C++ prototype	double _Complex __fpneg (double _Complex a);
Fortran description	FPNEG(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel absolute: __fpabs**
Purpose	Calculates in parallel the absolute values of the primary and secondary elements of operand *a*.

Formula	primary(result) = primary(a)
	secondary(result) = secondary(a)

C/C++ prototype	double _Complex __fpabs (double _Complex a);
Fortran description	FPABS(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel negate absolute: __fpnabs**
Purpose	Calculates in parallel the negative absolute values of the primary and secondary elements of operand *a*.

Formula	primary(result) = primary(a)
	secondary(result) = secondary(a)

C/C++ prototype	double _Complex __fpnabs (double _Complex a);

Fortran description	FPNABS(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)

Binary functions

Binary functions, listed in Table 8-7, operate on two input operands.

Table 8-7 Binary functions

Function	**Parallel add: __fpadd**
Purpose	Adds in parallel the primary and secondary elements of operands *a* and *b*.

Formula	primary(result) = primary(a) + primary(b)
	secondary(result) = secondary(a) + secondary(b)

C/C++ prototype	double _Complex __fpadd (double _Complex a, double _Complex b);
Fortran description	FPADD(A,B) where A is of type COMPLEX(8) where B is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel subtract: __fpsub**
Purpose	Subtracts in parallel the primary and secondary elements of operand *b* from the corresponding primary and secondary elements of operand *a*.

Formula	primary(result) = primary(a) - primary(b)
	secondary(result) = secondary(a) - secondary(b)

C/C++ prototype	double _Complex __fpsub (double _Complex a, double _Complex b);
Fortran description	FPSUB(A,B) where A is of type COMPLEX(8) where B is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel multiply: __fpmul**
Purpose	Multiplies in parallel the values of primary and secondary elements of operands *a* and *b*.

Formula	primary(result) = primary(a) × primary(b)
	secondary(result) = secondary(a) × secondary(b)

C/C++ prototype	double _Complex __fpmul (double _Complex a, double _Complex b);
Fortran description	FPMUL(A,B) where A is of type COMPLEX(8) where B is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Cross multiply: __fxmul**
Purpose	The product of the secondary element of *a* and the primary element of *b* is stored as the primary element of the return value. The product of the primary element of *a* and the secondary element of *b* is stored as the secondary element of the return value.

Formula	primary(result) = secondary(a) x primary(b)
	secondary(result) = primary(a) × secondary(b)

C/C++ prototype	double _Complex __fxmul (double _Complex a, double _Complex b);
Fortran description	FXMUL(A,B) where A is of type COMPLEX(8) where B is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Cross copy multiply: _fxpmul, __fxsmul**
Purpose	Both of these functions can be used to achieve the same result. The product of *a* and the primary element of *b* is stored as the primary element of the return value. The product of *a* and the secondary element of *b* is stored as the secondary element of the return value.

Formula	primary(result) = a x primary(b)
	secondary(result) = a x secondary(b)

C/C++ prototype	double _Complex __fxpmul (double _Complex b, double a); double _Complex __fxsmul (double _Complex b, double a);
Fortran description	FXPMUL(B,A) or FXSMUL(B,A) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)

Multiply-add functions

Multiply-add functions take three input operands, multiply the first two, and add or subtract the third. Table 8-8 lists these functions.

Table 8-8 Multiply-add functions

Function	Parallel multiply-add: __fpmadd
Purpose	The sum of the product of the primary elements of *a* and *b*, added to the primary element of *c*, is stored as the primary element of the return value. The sum of the product of the secondary elements of *a* and *b,* added to the secondary element of *c,* is stored as the secondary element of the return value.

Formula	primary(result) = primary(a) × primary(b) + primary(c)
	secondary(result) = secondary(a) × secondary(b) + secondary(c)

C/C++ prototype	double _Complex __fpmadd (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FPMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel negative multiply-add: __fpnmadd**
Purpose	The sum of the product of the primary elements of *a* and *b,* added to the primary element of *c,* is negated and stored as the primary element of the return value. The sum of the product of the secondary elements of *a* and *b,* added to the secondary element of *c,* is negated and stored as the secondary element of the return value.

Formula	primary(result) = -(primary(a) × primary(b) + primary(c))
	secondary(result) = -(secondary(a) × secondary(b) + secondary(c))

C/C++ prototype	double _Complex __fpnmadd (double _Complex c, double _Complex b, double _Complex a);

Fortran description	FPNMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel multiply-subtract: __fpmsub**
Purpose	The difference of the primary element of *c,* subtracted from the product of the primary elements of *a* and *b,* is stored as the primary element of the return value. The difference of the secondary element of *c,* subtracted from the product of the secondary elements of *a* and *b,* is stored as the secondary element of the return value.

Formula	primary(result) = primary(a) × primary(b) - primary(c)
	secondary(result) = secondary(a) × secondary(b) - secondary(c)

C/C++ prototype	double _Complex __fpmsub (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FPMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel negative multiply-subtract: __fpnmsub**
Purpose	The difference of the primary element of *c,* subtracted from the product of the primary elements of *a* and *b,* is negated and stored as the primary element of the return value. The difference of the secondary element of *c,* subtracted from the product of the secondary elements of *a* and *b,* is negated and stored as the secondary element of the return value.

Formula	primary(result) = -(primary(a) × primary(b) - primary(c))
	secondary(result) = -(secondary(a) × secondary(b) - secondary(c))

C/C++ prototype	double _Complex __fpnmsub (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FPNMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Cross multiply-add: __fxmadd**
Purpose	The sum of the product of the primary element of *a* and the secondary element of *b,* added to the primary element of *c,* is stored as the primary element of the return value. The sum of the product of the secondary element of *a* and the primary *b,* added to the secondary element of *c,* is stored as the secondary element of the return value.

Formula	primary(result) = primary(a) × secondary(b) + primary(c)
	secondary(result) = secondary(a) × primary(b) + secondary(c)

C/C++ prototype	double _Complex __fxmadd (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FXMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)

Function	**Cross negative multiply-add: __fxnmadd**
Purpose	The sum of the product of the primary element of *a* and the secondary element of *b*, added to the primary element of *c,* is negated and stored as the primary element of the return value. The sum of the product of the secondary element of *a* and the primary element of *b,* added to the secondary element of *c,* is negated and stored as the secondary element of the return value.

Formula	primary(result) = -(primary(a) × secondary(b) + primary(c))
	secondary(result) = -(secondary(a) × primary(b) + secondary(c))

C/C++ prototype	double _Complex __fxnmadd (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FXNMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Cross multiply-subtract: __fxmsub**
Purpose	The difference of the primary element of *c,* subtracted from the product of the primary element of *a* and the secondary element of *b*, is stored as the primary element of the return secondary element of *a*, and the primary element of *b* is stored as the secondary element of the return value.

Formula	primary(result) = primary(a) × secondary(b) - primary(c)
	secondary(result) = secondary(a) × primary(b) - secondary(c)

C/C++ prototype	double _Complex __fxmsub (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FXMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Cross negative multiply-subtract: __fxnmsub**
Purpose	The difference of the primary element of *c,* subtracted from the product of the primary element of *a* and the secondary element of *b*, is negated and stored as the primary element of the return value. The difference of the secondary element of *c,* subtracted from the product of the secondary element of *a* and the primary element of *b*, is negated and stored as the secondary element of the return value.

Formula	primary(result) = -(primary(a) × secondary(b) - primary(c))
	secondary(result) = -(secondary(a) × primary(b) - secondary(c))

C/C++ prototype	double _Complex __fxnmsub (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FXNMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Cross copy multiply-add: __fxcpmadd, __fxcsmadd**

Purpose	Both of these functions can be used to achieve the same result. The sum of the product of *a* and the primary element of *b,* added to the primary element of *c*, is stored as the primary element of the return value. The sum of the product of *a* and the secondary element of *b,* added to the secondary element of *c*, is stored as the secondary element of the return value.

Formula	primary(result) = a x primary(b) + primary(c)
	secondary(result) = a x secondary(b) + secondary(c)

C/C++ prototype	double _Complex __fxcpmadd (double _Complex c, double _Complex b, double a); double _Complex __fxcsmadd (double _Complex c, double _Complex b, double a);
Fortran description	FXCPMADD(C,B,A) or FXCSMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross copy negative multiply-add: __fxcpnmadd, __fxcsnmadd**
Purpose	Both of these functions can be used to achieve the same result. The difference of the primary element of *c*, subtracted from the product of *a* and the primary element of *b,* is negated and stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the secondary element of *b,* is negated and stored as the secondary element of the return value.

Formula	primary(result) = -(a x primary(b) + primary(c))
	secondary(result) = -(a x secondary(b) + secondary(c))

C/C++ prototype	double _Complex __fxcpnmadd (double _Complex c, double _Complex b, double a); double _Complex __fxcsnmadd (double _Complex c, double _Complex b, double a);
Fortran description	FXCPNMADD(C,B,A) or FXCSNMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross copy multiply-subtract: __fxcpmsub, __fxcsmsub**
Purpose	Both of these functions can be used to achieve the same result. The difference of the primary element of *c*, subtracted from the product of *a* and the primary element of *b,* is stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the secondary element of *b,* is stored as the secondary element of the return value.

Formula	primary(result) = a x primary(b) - primary(c)
	secondary(result) = a x secondary(b) - secondary(c)

C/C++ prototype	double _Complex __fxcpmsub (double _Complex c, double _Complex b, double a); double _Complex __fxcsmsub (double _Complex c, double _Complex b, double a);
Fortran description	FXCPMSUB(C,B,A) or FXCSMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross copy negative multiply-subtract: __fxcpnmsub, __fxcsnmsub**

Purpose	Both of these functions can be used to achieve the same result. The difference of the primary element of *c*, subtracted from the product of *a* and the primary element of *b,* is negated and stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the secondary element of *b,* is negated and stored as the secondary element of the return value.

Formula	primary(result) = -(a x primary(b) - primary(c))
	secondary(result) = -(a x secondary(b) - secondary(c))

C/C++ prototype	double _Complex __fxcpnmsub (double _Complex c, double _Complex b, double a); double _Complex __fxcsnmsub (double _Complex c, double _Complex b, double a);
Fortran description	FXCPNMSUB(C,B,A) or FXCSNMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross copy sub-primary multiply-add: __fxcpnpma, __fxcsnpma**
Purpose	Both of these functions can be used to achieve the same result. The difference of the primary element of *c*, subtracted from the product of *a* and the primary element of *b,* is negated and stored as the primary element of the return value. The sum of the product of *a* and the secondary element of *b,* added to the secondary element of *c*, is stored as the secondary element of the return value.

Formula	primary(result) = -(a x primary(b) - primary(c))
	secondary(result) = a x secondary(b) + secondary(c)

C/C++ prototype	double _Complex __fxcpnpma (double _Complex c, double _Complex b, double a); double _Complex __fxcsnpma (double _Complex c, double _Complex b, double a);
Fortran description	FXCPNPMA(C,B,A) or FXCSNPMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross copy sub-secondary multiply-add: __fxcpnsma, __fxcsnsma**
Purpose	Both of these functions can be used to achieve the same result. The sum of the product of *a* and the primary element of *b,* added to the primary element of *c*, is stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the secondary element of *b,* is negated and stored as the secondary element of the return value.

Formula	primary(result) = a x primary(b) + primary(c))
	secondary(result) = -(a x secondary(b) - secondary(c))

C/C++ prototype	double _Complex __fxcpnsma (double _Complex c, double _Complex b, double a); double _Complex __fxcsnsma (double _Complex c, double _Complex b, double a);
Fortran description	FXCPNSMA(C,B,A) or FXCSNSMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross mixed multiply-add: __fxcxma**

Purpose	The sum of the product of *a* and the secondary element of *b,* added to the primary element of *c*, is stored as the primary element of the return value. The sum of the product of *a* and the primary element of *b,* added to the secondary element of *c*, is stored as the secondary element of the return value.

Formula	primary(result) = a x secondary(b) + primary(c)
	secondary(result) = a x primary(b) +secondary(c)

C/C++ prototype	double _Complex __fxcxma (double _Complex c, double _Complex b, double a);
Fortran description	FXCXMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross mixed negative multiply-subtract: __fxcxnms**
Purpose	The difference of the primary element of *c*, subtracted from the product of *a* and the secondary element of *b,* is negated and stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the primary element of *b,* is negated and stored as the primary secondary of the return value.

Formula	primary(result) = -(a × secondary(b) - primary(c))
	secondary(result) = -(a × primary(b) - secondary(c))

C/C++ prototype	double _Complex __fxcxnms (double _Complex c, double _Complex b, double a);
Fortran description	FXCXNMS(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross mixed sub-primary multiply-add: __fxcxnpma**
Purpose	The difference of the primary element of *c*, subtracted from the product of *a* and the secondary element of *b,* is stored as the primary element of the return value. The sum of the product of *a* and the primary element of *b,* added to the secondary element of *c*, is stored as the secondary element of the return value.

Formula	primary(result) = -(a × secondary(b) - primary(c))
	secondary(result) = a × primary(b) + secondary(c)

C/C++ prototype	double _Complex __fxcxnpma (double _Complex c, double _Complex b, double a);
Fortran description	FXCXNPMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross mixed sub-secondary multiply-add: __fxcxnsma**
Purpose	The sum of the product of *a* and the secondary element of *b,* added to the primary element of *c*, is stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the primary element of *b,* is stored as the secondary element of the return value.

Formula	primary(result) = a x secondary(b) + primary(c))
	secondary(result) = -(a x primary(b) - secondary(c))

C/C++ prototype	double _Complex __fxcxnsma (double _Complex c, double _Complex b, double a);
Fortran description	FXCXNSMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)

Select functions

Table 8-9 lists and explains the parallel select functions that are available.

Table 8-9 Select functions

Function	**Parallel select: __fpsel**
Purpose	The value of the primary element of *a* is compared to zero. If its value is equal to or greater than zero, the primary element of *c* is stored in the primary element of the return value. Otherwise, the primary element of *b* is stored in the primary element of the return value. The value of the secondary element of *a* is compared to zero. If its value is equal to or greater than zero, the secondary element of *c* is stored in the secondary element of the return value. Otherwise, the secondary element of *b* is stored in the secondary element of the return value.

Formula	primary(result) = if primary(a) ≥ 0 then primary(c); else primary(b)
	secondary(result) = if secondary(a) ≥ 0 then primary(c); else secondary(b)

C/C++ prototype	double _Complex __fpsel (double _Complex a, double _Complex b, double _Complex c);
Fortran description	FPSEL(A,B,C) where A is of type COMPLEX(8) where B is of type COMPLEX(8) where C is of type COMPLEX(8) result is of type COMPLEX(8)

Examples of built-in functions usage

Using the following definitions, you can create a custom parallel add function that uses the parallel load and add built-in functions to add two double floating-point values in parallel and return the result as a complex number. See Example 8-17 for C/C++ and Example 8-18 on page 136 for Fortran.

Example 8-17 Using built-in functions in C/C++

```
double _Complex padd(double *x, double *y)
{
double _Complex a,b,c;
/* note possibility of alignment trap if (((unsigned int) x) % 32) >= 17) */

a = __lfpd(x); //load x[0] to the primary part of a, x[1] to the secondary part of a
b = __lfpd(y); //load y[0] to primary part of b, y[1] to the secondary part of b
c = __fpadd(a,b); // the primary part of c = x[0] + y[0]
        /* the secondary part of c = x[1] + y[1] */
return c;

/* alternately: */
```

```
   return __fpadd(__lfpd(x), __lfpd(y));    /* same code generated with optimization
enabled */
}
```

Example 8-18 Using built-in functions in Fortran

```
FUNCTION PADD (X, Y)
        COMPLEX(8) PADD
        REAL(8) X, Y
        COMPLEX(8) A, B, C

        A = LOADFP(X)
        B = LOADFP(Y)
        PADD = FPADD(A,B)

        RETURN
        END
```

Example 8-19 provides a sample of double-precision square matrix-matrix multiplication. This version uses 6x4 outer loop unrolling.

Example 8-19 Double-precision square matrix multiply example

```
subroutine dsqmm(a, b, c, n)
!
!# (C) Copyright IBM Corp. 2006 All Rights Reserved.
!# Rochester, MN
!
  implicit none
  integer i, j, k, n
  integer ii, jj, kk
  integer istop, jstop, kstop
  integer, parameter :: nb = 36   ! blocking factor
  complex(8) zero
  complex(8) a00, a01
  complex(8) a20, a21
  complex(8) b0, b1, b2, b3, b4, b5
  complex(8) c00, c01, c02, c03, c04, c05
  complex(8) c20, c21, c22, c23, c24, c25
  real(8) a(n,n), b(n,n), c(n,n)

  zero = (0.0d0, 0.0d0)

  !-----------------------------------------------------
  ! Double-precision square matrix-matrix multiplication.
  !-----------------------------------------------------
  ! This version uses 6x4 outer loop unrolling.
  ! The cleanup loops have been left out, so the results
  ! are correct for dimensions that are multiples of the
  ! two unrolling factors: 6 and 4.
  !-----------------------------------------------------

  do jj = 1, n, nb

    if ((jj + nb - 1) .lt. n) then
      jstop = (jj + nb - 1)
    else
      jstop = n
    endif
```

```
do ii = 1, n, nb

  if ((ii + nb - 1) .lt. n) then
    istop = (ii + nb - 1)
  else
    istop = n
  endif

  !---------------------------------
  ! initialize a block of c to zero
  !---------------------------------
  do j = jj, jstop - 5, 6
    do i = ii, istop - 1, 2
      call storefp(c(i,j)  , zero)
      call storefp(c(i,j+1), zero)
      call storefp(c(i,j+2), zero)
      call storefp(c(i,j+3), zero)
      call storefp(c(i,j+4), zero)
      call storefp(c(i,j+5), zero)
    end do
  end do

  !--------------------------------------------------------
  ! multiply block by block with 6x4 outer loop un-rolling
  !--------------------------------------------------------
  do kk = 1, n, nb
    if ((kk + nb - 1) .lt. n) then
      kstop = (kk + nb - 1)
    else
      kstop = n
    endif

    do j = jj, jstop - 5, 6
    do i = ii, istop - 3, 4

      c00 = loadfp(c(i,j  ))
      c01 = loadfp(c(i,j+1))
      c02 = loadfp(c(i,j+2))
      c03 = loadfp(c(i,j+3))
      c04 = loadfp(c(i,j+4))
      c05 = loadfp(c(i,j+5))

      c20 = loadfp(c(i+2,j  ))
      c21 = loadfp(c(i+2,j+1))
      c22 = loadfp(c(i+2,j+2))
      c23 = loadfp(c(i+2,j+3))
      c24 = loadfp(c(i+2,j+4))
      c25 = loadfp(c(i+2,j+5))

      a00 = loadfp(a(i,kk ))
      a20 = loadfp(a(i+2,kk ))
      a01 = loadfp(a(i,kk+1))
      a21 = loadfp(a(i+2,kk+1))

      do k = kk, kstop - 1, 2
        b0 = loadfp(b(k,j  ))
        b1 = loadfp(b(k,j+1))
        b2 = loadfp(b(k,j+2))
        b3 = loadfp(b(k,j+3))
```

```
            b4 = loadfp(b(k,j+4))
            b5 = loadfp(b(k,j+5))
            c00 = fxcpmadd(c00, a00, real(b0))
            c01 = fxcpmadd(c01, a00, real(b1))
            c02 = fxcpmadd(c02, a00, real(b2))
            c03 = fxcpmadd(c03, a00, real(b3))
            c04 = fxcpmadd(c04, a00, real(b4))
            c05 = fxcpmadd(c05, a00, real(b5))
            c20 = fxcpmadd(c20, a20, real(b0))
            c21 = fxcpmadd(c21, a20, real(b1))
            c22 = fxcpmadd(c22, a20, real(b2))
            c23 = fxcpmadd(c23, a20, real(b3))
            c24 = fxcpmadd(c24, a20, real(b4))
            c25 = fxcpmadd(c25, a20, real(b5))
            a00 = loadfp(a(i,k+2  ))
            a20 = loadfp(a(i+2,k+2  ))
            c00 = fxcpmadd(c00, a01, imag(b0))
            c01 = fxcpmadd(c01, a01, imag(b1))
            c02 = fxcpmadd(c02, a01, imag(b2))
            c03 = fxcpmadd(c03, a01, imag(b3))
            c04 = fxcpmadd(c04, a01, imag(b4))
            c05 = fxcpmadd(c05, a01, imag(b5))
            c20 = fxcpmadd(c20, a21, imag(b0))
            c21 = fxcpmadd(c21, a21, imag(b1))
            c22 = fxcpmadd(c22, a21, imag(b2))
            c23 = fxcpmadd(c23, a21, imag(b3))
            c24 = fxcpmadd(c24, a21, imag(b4))
            c25 = fxcpmadd(c25, a21, imag(b5))
            a01 = loadfp(a(i,k+3))
            a21 = loadfp(a(i+2,k+3))
          end do

          call storefp(c(i  ,j  ), c00)
          call storefp(c(i  ,j+1), c01)
          call storefp(c(i  ,j+2), c02)
          call storefp(c(i  ,j+3), c03)
          call storefp(c(i  ,j+4), c04)
          call storefp(c(i  ,j+5), c05)

          call storefp(c(i+2,j  ), c20)
          call storefp(c(i+2,j+1), c21)
          call storefp(c(i+2,j+2), c22)
          call storefp(c(i+2,j+3), c23)
          call storefp(c(i+2,j+4), c24)
          call storefp(c(i+2,j+5), c25)

        end do
        end do

      end do !kk

    end do !ii

  end do !jj

end
```

9

Running and debugging applications

In this chapter, we explain how to run and debug applications on the IBM Blue Gene/P system. These types of tools are essential for application developers. Although we do not cover all of the existing tools, we provide an overview of some of the currently available tools.

We cover the following topics:

- Running applications
- Debugging applications

9.1 Running applications

Blue Gene/P applications can be run in several ways. We briefly discuss each method and provide references for more detailed documentation.

9.1.1 MMCS console

It is possible to run applications directly from the MMCS console. The main drawback to using this approach is that it requires users to have direct access to the Service Node, which is undesirable from a security perspective.

When using the MMCS console, it is necessary to first manually select and allocate a block. A *block* in this case refers to a partition or set of nodes to run the job. (See Appendix A, “Blue Gene/P hardware-naming conventions” on page 325, for more information.) At this point, it is possible to run Blue Gene/P applications. The set of commands in Example 9-1 from the MMCS console window show how to accomplish this. The names can be site specific, but the example illustrates the procedure.

To start the console session, use the sequence of commands shown in Example 9-1 on the Service Node.

Example 9-1 Starting the console session

```
cd /bgsys/drivers/ppcfloor/bin
source ~bgpsysdb/sqllib/db2profile
mmcs_db_console --bgpadmingroup p/bluegene/bgpall
connecting to mmcs_server
connected to mmcs_server
connected to DB2
mmcs$list_blocks
OK
N00_64_1         B manojd  (1)  connected
N02_32_1         I walkup  (0)  connected
N04_32_1         B manojd  (1)  connected
N05_32_1         B manojd  (1)  connected
N06_32_1         I sameer77(1)  connected
N07_32_1         I gdozsa  (1)  connected
N08_64_1         I vezolle (1)  connected
N12_32_1         I vezolle (0)  connected
mmcs$ allocate N14_32_1
OK
mmcs$ list_blocks
OK
N00_64_1         B manojd  (1)  connected
N02_32_1         I walkup  (0)  connected
N04_32_1         B manojd  (1)  connected
N05_32_1         B manojd  (1)  connected
N06_32_1         I sameer77(1)  connected
N07_32_1         I gdozsa  (1)  connected
N08_64_1         I vezolle (1)  connected
N12_32_1         I vezolle (0)  connected
N14_32_1         I cpsosa  (1)  connected
mmcs$ submitjob N14_32_1 /bgusr/cpsosa/hello/c/omp_hello_bgp /bgusr/cpsosa/hello/c
OK
jobId=14008
mmcs$ free N14_32_1
```

```
OK
mmcs$ quit
OK
mmcs_db_console is terminating, please wait...
mmcs_db_console: closing database connection
mmcs_db_console: closed database connection
mmcs_db_console: closing console port
mmcs_db_console: closed console port
```

For more information about using the MMCS console, see *IBM System Blue Gene Solution: Blue Gene/P System Administration*, SG24-7417.

9.1.2 mpirun

In the absence of a scheduling application, we recommend that you use **mpirun** to run Blue Gene/P applications on statically allocated partitions. Users can access this application from the Front End Node, which provides better security protection than using the MMCS console. For more complete information about using **mpirun**, see Chapter 11, “mpirun” on page 177.

With **mpirun**, you can select and allocate a block and run a Message Passing Interface (MPI) application, all in one step as shown in Example 9-2.

Example 9-2 Using mpirun

```
cpsosa@descartes:/bgusr/cpsosa/red/pi/c> csh
descartes pi/c> set MPIRUN="/bgsys/drivers/ppcfloor/bin/mpirun"
descartes pi/c> set MPIOPT="-np 1"
descartes pi/c> set MODE="-mode SMP"
descartes pi/c> set PARTITION="-partition N14_32_1"
descartes pi/c> set WDIR="-cwd /bgusr/cpsosa/red/pi/c"
descartes pi/c> set EXE="-exe /bgusr/cpsosa/red/pi/c/pi_critical_bgp"
descartes pi/c> $MPIRUN $PARTITION $MPIOPT $MODE $WDIR $EXE -env "OMP_NUM_THREADS=1"
Estimate of pi: 3.14159
Total time 560.055988
```

All output in this example is sent to the display. To specify that you want this information sent to a file, you must add the following line, for example, to the end of the **mpirun** command:

```
>/bgusr/cpsosa/red/pi/c/pi_critical.stdout 2>/bgusr/cpsosa/red/pi/c/pi_critical.stderr
```

This line sends standard output to the pi_critical.stdout file and standard error to the pi_critical.stderr file. Both files are in the /bgusr/cpsosa/red/pi/c directory.

9.1.3 submit

In HTC mode you must use the **submit** command, which is analogous to **mpirun** because its purpose is to act as a shadow of the job. It transparently forwards stdin, and receives stdout and stderr. More detailed usage information is available in Chapter 12, “High-Throughput Computing (HTC) paradigm” on page 201.

9.1.4 IBM LoadLeveler

At present, LoadLeveler support for the Blue Gene/P system is provided via a programming request for price quotation (PRPQ). The IBM Tivoli Workload Scheduler LoadLeveler product is intended to manage both serial and parallel jobs over a cluster of servers. This distributed environment consists of a pool of machines or servers, often referred to as a *LoadLeveler cluster*. Machines in the pool can be of several types: desktop workstations available for batch jobs (usually when not in use by their owner), dedicated servers, and parallel machines.

LoadLeveler allocates machine resources in the cluster to run jobs. The scheduling of jobs depends on the availability of resources within the cluster and various rules, which can be defined by the LoadLeveler administrator. A user submits a job using a job command file. The LoadLeveler scheduler attempts to find resources within the cluster to satisfy the requirements of the job. LoadLeveler maximizes the efficiency of the cluster by maximizing the utilization of resources, while at the same time minimizing the job turnaround time experienced by users.

LoadLeveler provides a rich set of functions for job scheduling and cluster resource management. Some of the tasks that LoadLeveler can perform include:

- Choosing the next job to run.
- Examining the job requirements.
- Collecting available resources in the cluster.
- Choosing the “best” machines for the job.
- Dispatching the job to the selected machine.
- Controlling running jobs.
- Creating reservations and scheduling jobs to run in the reservations.
- Job preemption to enable high-priority jobs to run immediately.
- Fair share scheduling to automatically balance resources among users or groups of users.
- Co-scheduling to enable several jobs to be scheduled to run at the same time.
- Multi-cluster support to allow several LoadLeveler clusters to work together to run user jobs.

The LoadLeveler documentation contains information for setting up and using LoadLeveler with Blue Gene/P. The documentation is available online at:

http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp

9.1.5 Other scheduler products

You can use custom scheduling applications to run applications on the Blue Gene/P system. You write custom “glue” code between the scheduler and the Blue Gene/P system by using the Bridge APIs, which are described in Chapter 13, “Control system (Bridge) APIs” on page 209, and Chapter 14, “Real-time Notification APIs” on page 251.

9.2 Debugging applications

In this section, we discuss the debuggers that are supported by the Blue Gene/P system.

9.2.1 General debugging architecture

Four pieces of code are involved when debugging applications on the Blue Gene/P system:

- The Compute Node Kernel, which provides the low-level primitives that are necessary to debug an application
- The control and I/O daemon (CIOD) running on the I/O Nodes, which provides control and communications to Compute Nodes
- A "debug server" running on the I/O Nodes, which is vendor-supplied code that interfaces with the CIOD
- A debug client running on a Front End Node, which is where the user does their work interactively

A debugger must interface to the Compute Node through an API implemented in CIOD to debug an application running on a Compute Node. This debug code is started on the I/O Nodes by the control system and can interface with other software, such as a GUI or command-line utility on a Front End Node. The code running on the I/O Nodes using the API in CIOD is referred to as a *debug server*. It is provided by the debugger vendor for use with the Blue Gene/P system. Many possible debug servers are possible.

A *debug client* is a piece of code that runs on a Front End Node that the user interacts with directly. It makes remote requests to the debug server running on the I/O Nodes, which in turn passes the request through CIOD and eventually to the Compute Node. The debug client and debug server usually communicate using TCP/IP.

9.2.2 GNU Project debugger

The GNU Project debugger (GDB) is the primary debugger of the GNU project. You can learn more about GDB on the Web at the following address:

`http://www.gnu.org/software/gdb/gdb.html`

A great amount of documentation is available about the GDB. Because we do not discuss how to use it in this book, refer to the following Web site for details:

`http://www.gnu.org/software/gdb/documentation/`

Support has been added to the Blue Gene/P system for which the GDB can work with applications that run on Compute Nodes. IBM provides a simple debug server called *gdbserver*. Each running instance of GDB is associated with one, and only one, Compute Node. If you must debug an MPI application that runs on multiple Compute Nodes, and you must, for example, view variables that are associated with more than one instance of the application, you run multiple instances of GDB.

Most people use GDB to debug local processes that run on the same machine on which they are running GDB. With GDB, you also have the ability to debug remotely via a GDB server on the remote machine. GDB on the Blue Gene/L system is used in this mode. We refer to GDB as the *GDB client*, although most users recognize it as GDB used in a slightly different manner.

Limitations

Gdbserver implements the minimum number of primitives required by the GDB remote protocol specification. As such, advanced features that might be available in other implementations are not available in this implementation. However, sufficient features are implemented to make it a useful tool. This implementation has some of the following limitations:

- Each instance of a GDB client can connect to and debug one Compute Node. To debug multiple Compute Nodes at the same time, you must run multiple GDB clients at the same time. Although you might need multiple GDB clients for multiple Compute Nodes, one gdbserver on each I/O Node is all that is required. The Blue Gene/P control system manages that part.
- IBM does not ship a GDB client with the Blue Gene/P system. The user can use an existing GDB client to connect to the IBM-supplied gdbserver. Most functions do work, but standard GDB clients are not aware of the full "double hummer" floating-point register set that Blue Gene/L provides. The GDB clients that come with SUSE Linux Enterprise Server (SLES) 10 for IBM PowerPC are known to work.
- To debug an application, the debug server must be started and running before you attempt to debug. Using an option on the **mpirun** or **submit** command, you can get the debug server running before your application does. If you do not use this option and you must debug your application, you do not have a mechanism to start the debug server and thus have no way to debug your application.
- Gdbserver is not aware of user-specified MPI topologies. You still can debug your application, but the connection information given to you by **mpirun** for each MPI rank can be incorrect.

Prerequisite software

The GDB should have been installed during the installation procedure. You can verify the installation by seeing whether the /bgsys/drivers/ppcfloor/gnu-linux/bin/gdb file exists on your Front End Node.

The rest of the software support required for GDB should be installed as part of the control programs.

Preparing your program

The MPI, OpenMP, MPI-OpenMP, or CNK program that you want to debug must be compiled in a manner that allows for debugging information (symbol tables, ties to source, and so on) to be included in the executable. In addition, do *not* use compiler optimization because it makes it difficult, if not impossible, to tie object code back to source, for example, when compiling a program written in Fortran that you want to debug, compile the application using an invocation similar to one shown in Example 9-3.

Example 9-3 Makefile used for building the program with debugging flags

```
BGP_FLOOR   = /bgsys/drivers/ppcfloor
BGP_IDIRS   = -I$(BGP_FLOOR)/arch/include -I$(BGP_FLOOR)/comm/include
BGP_LIBS    = -L$(BGP_FLOOR)/comm/lib -lmpich.cnk -L$(BGP_FLOOR)/comm/lib -ldcmfcoll.cnk
-ldcmf.cnk -lpthread -lrt -L$(BGP_FLOOR)/runtime/SPI -lSPI.cna

XL         = /opt/ibmcmp/xlf/bg/11.1/bin/bgxlf90

EXE         =  example_9_4_bgp
OBJ         =  example_9_4.o
SRC         =  example_9_4.f
```

```
FLAGS       =  -g -O0 -qarch=450  -qtune=450 -I$(BGP_FLOOR)/comm/include

$(EXE): $(OBJ)
        ${XL} $(FLAGS) -o $(EXE) $(OBJ) $(BGP_LIBS)
$(OBJ): $(SRC)
        ${XL} $(FLAGS) $(BGP_IDIRS) -c $(SRC)

clean:
        rm *.o example_9_4_bgp

cpsosa@descartes:/bgusr/cpsosa/red/debug> make

/opt/ibmcmp/xlf/bg/11.1/bin/bgxlf90 -g -O0 -qarch=450  -qtune=450
-I/bgsys/drivers/ppcfloor/comm/include -I/bgsys/drivers/ppcfloor/arch/include
-I/bgsys/drivers/ppcfloor/comm/include -c example_9_4.f
** nooffset   === End of Compilation 1 ===
1501-510  Compilation successful for file example_9_4.f.
/opt/ibmcmp/xlf/bg/11.1/bin/bgxlf90 -g -O0 -qarch=450  -qtune=450
-I/bgsys/drivers/ppcfloor/comm/include -o example_9_4_bgp example_9_4.o
-L/bgsys/drivers/ppcfloor/comm/lib -lmpich.cnk -L/bgsys/drivers/ppcfloor/comm/lib -ldcmfcoll.cnk
-ldcmf.cnk -lpthread -lrt -L/bgsys/drivers/ppcfloor/runtime/SPI -lSPI.cna
```

The -g switch tells the compiler to include debug information. The -O0 (the letter capital “O” followed by a zero) switch tells it to disable optimization.

For more information about the IBM XL compilers for the Blue Gene/P system, see Chapter 8, “Developing applications with IBM XL compilers” on page 97.

Important: Make sure that the text file that contains the source for your program is located in the same directory as the program itself and has the same file name (different extension).

Debugging

Follow the steps in this section to start debugging your application. In this example, the MPI program’s name is example_9_4_bgp as illustrated in Example 9-4 on page 146 (source code not shown), and the source code file is example_9_4.f. The partition (block) used is called N14_32_1.

An extra parameter (-start_gdbserver...) is passed in on the **mpirun** or **submit** command. In this example the application uses MPI so mpirun is used, but the process for **submit** is the same. The extra option changes the way **mpirun** loads and executes your code. Here is a brief summary of the changes:

1. The code is loaded onto the Compute Nodes (in our example, the executable is example_9_4_bgp), but it does not start running immediately.
2. The control system starts the specified debug server (gdbserver) on all of the I/O Nodes in the partition that is running your job, which in our example is N14_32_1.
3. The **mpirun** command pauses, so that you get a chance to connect GDB clients to the Compute Nodes that you are going to debug.
4. When you are finished connecting GDB clients to Compute Nodes, you press **Enter** to signal the **mpirun** command, and then the application starts running on the Compute Nodes.

During the pause in step 3, you have an opportunity to connect the GDB clients to the Compute Nodes before the application runs, which is desirable if you must start the application under debugger control. This step is optional. If you do not connect before the application starts running on the Compute Nodes, you can still connect later because the debugger server was started on the I/O Nodes.

To start debugging your application:

1. Open two separate console shells.
2. Go to the first shell window:
   a. Change to the directory (**`cd`**) that contains your program executable. In our example, the directory is /bgusr/cpsosa/red/debug.
   b. Start your application using **`mpirun`** with a command similar to the one shown in Example 9-4. You should see messages in the console, similar to those shown in Example 9-4.

Example 9-4 Messages in the console

```
set MPIRUN="/bgsys/drivers/ppcfloor/bin/mpirun"
set MPIOPT="-np 1"
set MODE="-mode SMP"
set PARTITION="-partition N14_32_1"
set WDIR="-cwd /bgusr/cpsosa/red/debug"
set EXE="-exe /bgusr/cpsosa/red/debug/example_9_4_bgp"
#
$MPIRUN $PARTITION $MPIOPT $MODE $WDIR $EXE -env "OMP_NUM_THREADS=4" -start_gdbserver
/sbin.rd/gdbserver -verbose 1
#
echo "That's all folks!!"

descartes red/debug> set EXE="-exe /bgusr/cpsosa/red/debug/example_9_4_bgp"
descartes red/debug> $MPIRUN $PARTITION $MPIOPT $MODE $WDIR $EXE -env "OMP_NUM_THREADS=4"
-start_gdbserver /bgsys/drivers/ppcfloor/ramdisk/sbin/gdbserver -verbose 1
<Sep 15 10:14:58.642369> FE_MPI (Info) : Invoking mpirun backend
<Sep 15 10:14:05.741121> BRIDGE (Info) : rm_set_serial() - The machine serial number (alias) is
BGP
<Sep 15 10:15:00.461655> FE_MPI (Info) : Preparing partition
<Sep 15 10:14:05.821585> BE_MPI (Info) : Examining specified partition
<Sep 15 10:14:10.085997> BE_MPI (Info) : Checking partition N14_32_1 initial state ...
<Sep 15 10:14:10.086041> BE_MPI (Info) : Partition N14_32_1 initial state = READY ('I')
<Sep 15 10:14:10.086059> BE_MPI (Info) : Checking partition owner...
<Sep 15 10:14:10.086087> BE_MPI (Info) : partition N14_32_1 owner is 'cpsosa'
<Sep 15 10:14:10.088375> BE_MPI (Info) : Partition owner matches the current user
<Sep 15 10:14:10.088470> BE_MPI (Info) : Done preparing partition
<Sep 15 10:15:04.804078> FE_MPI (Info) : Adding job
<Sep 15 10:14:10.127380> BE_MPI (Info) : Adding job to database...
<Sep 15 10:15:06.104035> FE_MPI (Info) : Job added with the following id: 14035
<Sep 15 10:15:06.104096> FE_MPI (Info) : Loading Blue Gene job
<Sep 15 10:14:11.426987> BE_MPI (Info) : Loading job 14035 ...
<Sep 15 10:14:11.450495> BE_MPI (Info) : Job load command successful
<Sep 15 10:14:11.450525> BE_MPI (Info) : Waiting for job 14035 to get to Loaded/Running state
...
<Sep 15 10:14:16.458474> BE_MPI (Info) : Job 14035 switched to state LOADED
<Sep 15 10:14:21.467401> BE_MPI (Info) : Job loaded successfully
<Sep 15 10:15:16.179023> FE_MPI (Info) : Starting debugger setup for job 14035
```

```
<Sep 15 10:15:16.179090> FE_MPI (Info) : Setting debug info in the block record
<Sep 15 10:14:21.502593> BE_MPI (Info) : Setting debugger executable and arguments in block
description
<Sep 15 10:14:21.523480> BE_MPI (Info) : Debug info set successfully
<Sep 15 10:15:16.246415> FE_MPI (Info) : Query job 14035 to find MPI ranks for compute nodes
<Sep 15 10:15:16.246445> FE_MPI (Info) : Getting process table information for the debugger
<Sep 15 10:14:22.661841> BE_MPI (Info) : Query job completed - proctable is filled in
<Sep 15 10:15:17.386617> FE_MPI (Info) : Starting debugger servers on I/O nodes for job 14035
<Sep 15 10:15:17.386663> FE_MPI (Info) : Attaching debugger to a new job.
<Sep 15 10:14:22.721982> BE_MPI (Info) : Debugger servers are now spawning
<Sep 15 10:15:17.446486> FE_MPI (Info) : Notifying debugger that servers have been spawned.

Make your connections to the compute nodes now - press [Enter] when you
are ready to run the app.  To see the IP connection information for a
specific compute node, enter its MPI rank and press [Enter].  To see
all of the compute nodes, type 'dump_proctable'.

>
<Sep 15 10:17:20.754179> FE_MPI (Info) : Debug setup is complete
<Sep 15 10:17:20.754291> FE_MPI (Info) : Waiting for Blue Gene job to get to Loaded state
<Sep 15 10:16:26.118529> BE_MPI (Info) : Waiting for job 14035 to get to Loaded/Running state
...
<Sep 15 10:16:31.128079> BE_MPI (Info) : Job loaded successfully
<Sep 15 10:17:25.806882> FE_MPI (Info) : Beginning job 14035
<Sep 15 10:16:31.129878> BE_MPI (Info) : Beginning job 14035 ...
<Sep 15 10:16:31.152525> BE_MPI (Info) : Job begin command successful
<Sep 15 10:17:25.871476> FE_MPI (Info) : Waiting for job to terminate
<Sep 15 10:16:31.231304> BE_MPI (Info) : IO - Threads initialized
<Sep 15 10:27:31.301600> BE_MPI (Info) : I/O output runner thread terminated
<Sep 15 10:27:31.301639> BE_MPI (Info) : I/O input runner thread terminated
<Sep 15 10:27:31.355816> BE_MPI (Info) : Job 14035 switched to state TERMINATED ('T')
<Sep 15 10:27:31.355848> BE_MPI (Info) : Job successfully terminated - TERMINATED ('T')
<Sep 15 10:28:26.113983> FE_MPI (Info) : Job terminated normally
<Sep 15 10:28:26.114057> FE_MPI (Info) : exit status = (0)
<Sep 15 10:27:31.435578> BE_MPI (Info) : Starting cleanup sequence
<Sep 15 10:27:31.435615> BE_MPI (Info) : cleanupDatabase() - job already terminated / hasn't
been added
<Sep 15 10:27:31.469474> BE_MPI (Info) : cleanupDatabase() - Partition was supplied with READY
('I') initial state
<Sep 15 10:27:31.469504> BE_MPI (Info) : cleanupDatabase() - No need to destroy the partition
<Sep 15 10:28:26.483855> FE_MPI (Info) : ==   FE completed   ==
<Sep 15 10:28:26.483921> FE_MPI (Info) : == Exit status:   0 ==
```

c. Find the IP address and port of the Compute Node that you want to debug. You can do this using either of the following ways:

   - Enter the rank of the program instance that you want to debug and press **Enter**.
   - Dump the address or port of each node by typing `dump_proctable` and press **Enter**.

   See Example 9-5.

   Example 9-5 Finding the IP address and port of the Compute Node for debugging

```
> 2
MPI Rank 2: Connect to 172.30.255.85:7302
> 4
MPI Rank 4: Connect to 172.30.255.85:7304
```

```
>
or
> dump_proctable
MPI Rank 0: Connect to 172.24.101.128:7310
>
```

3. From the second shell, complete the following steps:
   a. Change to the directory (**cd**) that contains your program executable.
   b. Type the following command, using the name of your own executable instead of `example_9_4_bgp`:

      ```
      /bgsys/drivers/ppcfloor/gnu-linux/bin/gdb example_9_4_bgp
      ```

   c. Enter the following command, using the address of the Compute Node that you want to debug and determined in step 2:

      ```
      target remote ipaddr:port
      ```

   You are now debugging the specified application on the configured Compute Node.
4. Set one or more breakpoints (using the GDB **break** command). Press Enter from the first shell to continue that application.

   If successful, your breakpoint should eventually be reached in the second shell, and you can use standard GDB commands to continue.

Debugging dynamically linked applications

When debugging dynamically linked applications, the GDB client provides variables that can assist in getting the correct debugging symbols loaded when your application is running on the compute node. The two variables in the GDB client are solib-search-path and solib-absolute-prefix. Setting solib-search-path to the directories containing the shared libraries that your application uses causes GDB to load the symbols from the libraries found in the path or if not found in the path, take them from the default location. The solib-absolute-prefix variable is used to specify a prefix that can be put in front of each of the entries specified in the path variable.

The .gdbinit file in your home directory is read automatically when GDB starts, making it a useful place to set up these variables. Example 9-6 shows a sample .gdbinit file that causes the dynamic symbols to be loaded from the libraries in the Blue Gene/P install directories.

Example 9-6 Sample .gdbinit file

```
# The following 2 lines are a single line in the .gdbinit file
set solib-search-path
/bgsys/drivers/ppcfloor/gnu-linux/lib:/bgsys/drivers/ppcfloor/gnu-linux/powerpc-bgp-linux/lib

set solib-absolute-prefix /none
```

The GDB **info shared** command can be used to display where GDB found dynamic libraries when debugging a dynamically linked application. Example 9-7 shows sample output using the GDB **info shared** command with the sample .gdbinit file from Example 9-6 on page 148.

Example 9-7 Sample GDB info shared output

```
Program received signal SIGINT, Interrupt.
0x010024f0 in _start ()
(gdb) info shared
From        To          Syms Read   Shared Object Library
0x006022f0  0x0061a370  Yes         /bgsys/drivers/DRV200_2008-080512P-GNU10/ppc/gnu-linux/powerpc-bgp-linux/lib/ld.so.1
0x81830780  0x81930730  Yes         /bgsys/drivers/DRV200_2008-080512P-GNU10/ppc/gnu-linux/lib/libpython2.5.so.1.0
0x81a36850  0x81a437d0  Yes         /bgsys/drivers/DRV200_2008-080512P-GNU10/ppc/gnu-linux/powerpc-bgp-linux/lib/libpthread.so.0
0x81b36da0  0x81b38300  Yes         /bgsys/drivers/DRV200_2008-080512P-GNU10/ppc/gnu-linux/powerpc-bgp-linux/lib/libdl.so.2
0x81c38c00  0x81c39ac0  Yes         /bgsys/drivers/DRV200_2008-080512P-GNU10/ppc/gnu-linux/powerpc-bgp-linux/lib/libutil.so.1
0x81d47e10  0x81d95fd0  Yes         /bgsys/drivers/DRV200_2008-080512P-GNU10/ppc/gnu-linux/powerpc-bgp-linux/lib/libm.so.6
0x81e5d6d0  0x81f5e8d0  Yes         /bgsys/drivers/DRV200_2008-080512P-GNU10/ppc/gnu-linux/powerpc-bgp-linux/lib/libc.so.6
```

9.2.3 Core Processor debugger

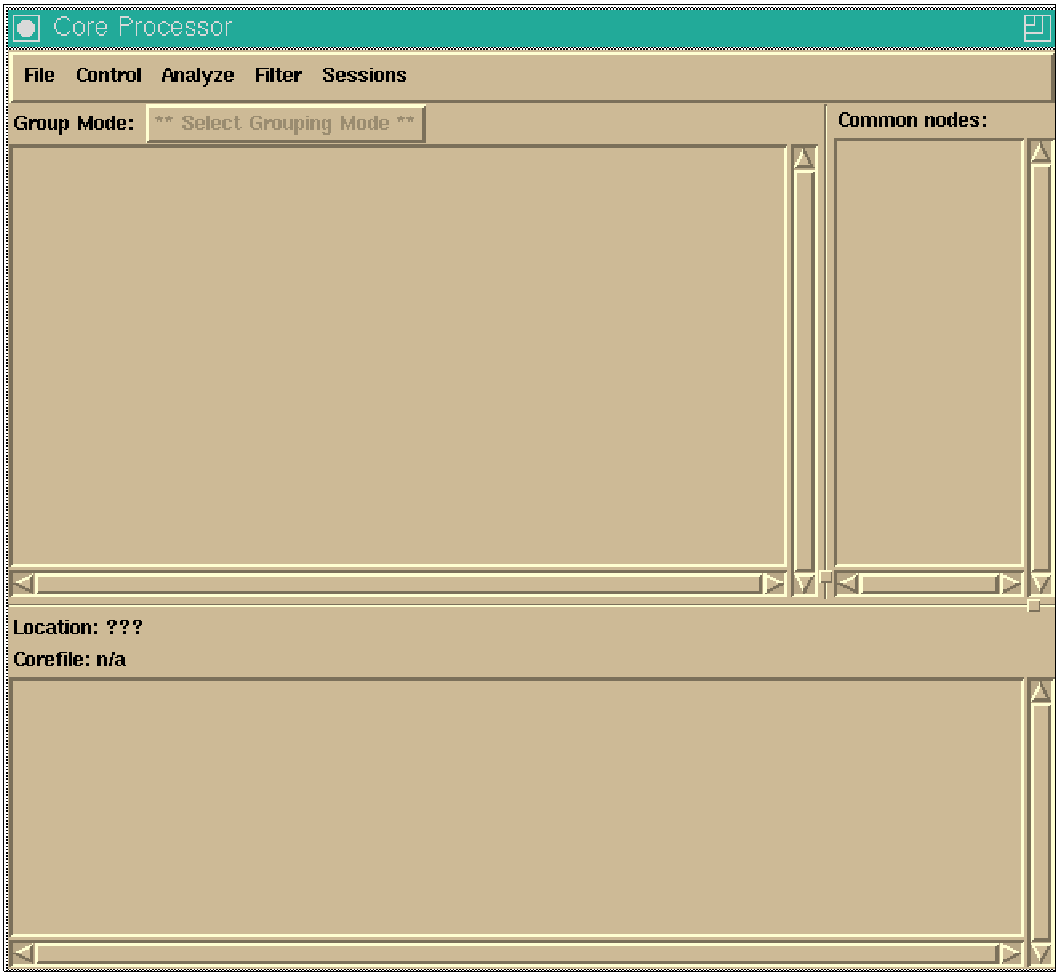
Core Processor is a basic tool that can help you debug your application. This tool is discussed in detail in *IBM System Blue Gene Solution: Blue Gene/P System Administration*, SG24-7417. In the following sections, we briefly describe how to use it to debug applications.

9.2.4 Starting the Core Processor tool

To start the Core Processor tool:

1. Export DISPLAY and make sure it works.
2. Type `coreprocessor.pl` to specify the Core Processor tool. You might need to specify the full path.
3. From the GUI window that opens, click **OK**. The Perl script is invoked automatically.

Figure 9-1 shows how the Core Processor tool GUI looks after the Perl script is invoked. The Core Processor windows do not provide any initial information. You must explicitly select a task that is provided via the GUI.



Location: ???

Corefile: n/a

Figure 9-1 Core Processor initial window

9.2.5 Attaching running applications

To do a live debug on Compute Nodes:

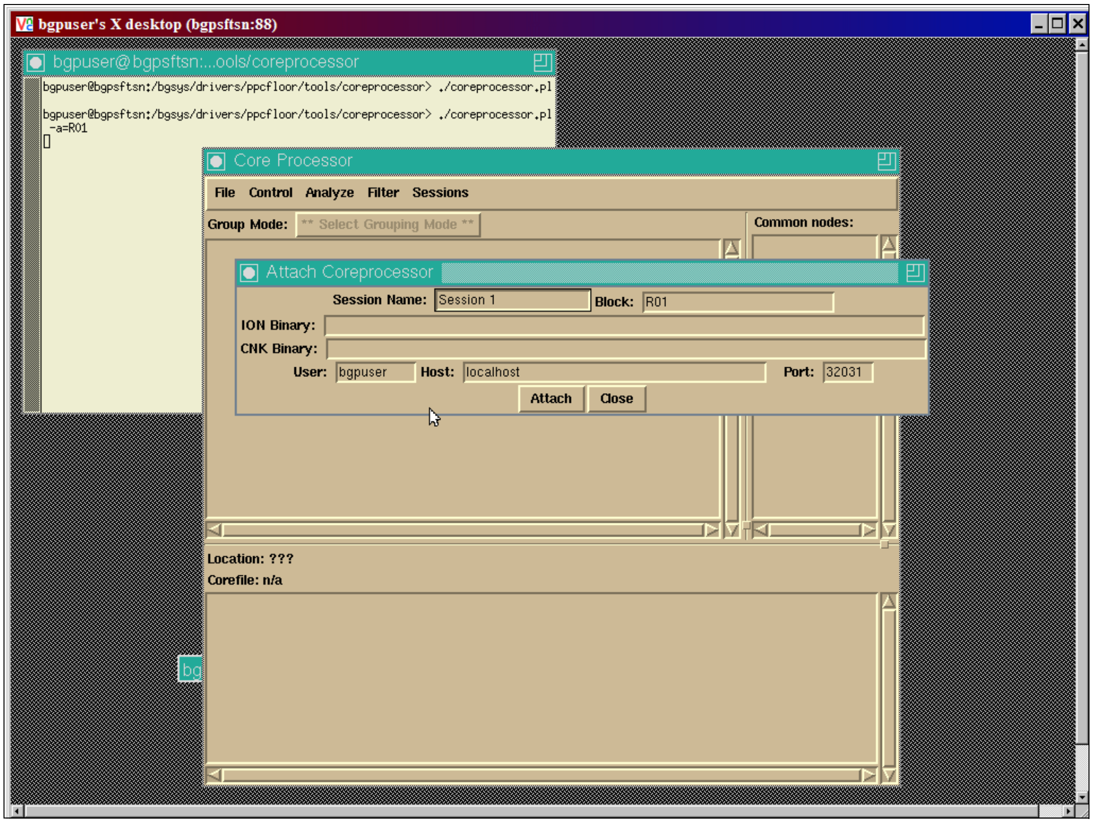
1. Start the Core Processor GUI as explained in the previous section.
2. Select **File** → **Attach To Block**.

3. In the Attach Coreprocessor window (see Figure 9-2), supply the following information:
   - Session Name: You can run more than one session at a time, so use this option to distinguish between multiple sessions.
   - Block name.
   - CNK binary (with path): To see both your application and the Compute Node Kernel in the stack, specify your application binary and the Compute Node Kernel image separated by a colon (:) as shown in the following example:

     ```
     /bgsys/drivers/ppcfloor/cnk/bgp_kernel.cn:/bguser/bgpuser/hello_mpi_loop.rts
     ```

   - User name or owner of the Midplane Management Control System (MMCS) block.
   - Port: TCP port on which the MMCS server is listening for console connections, which is probably 32031.
   - Host name or TCP/IP address for the MMCS server: Typically it is `localhost` or the Service Node’s TCP/IP address.

   Click the **Attach** button.



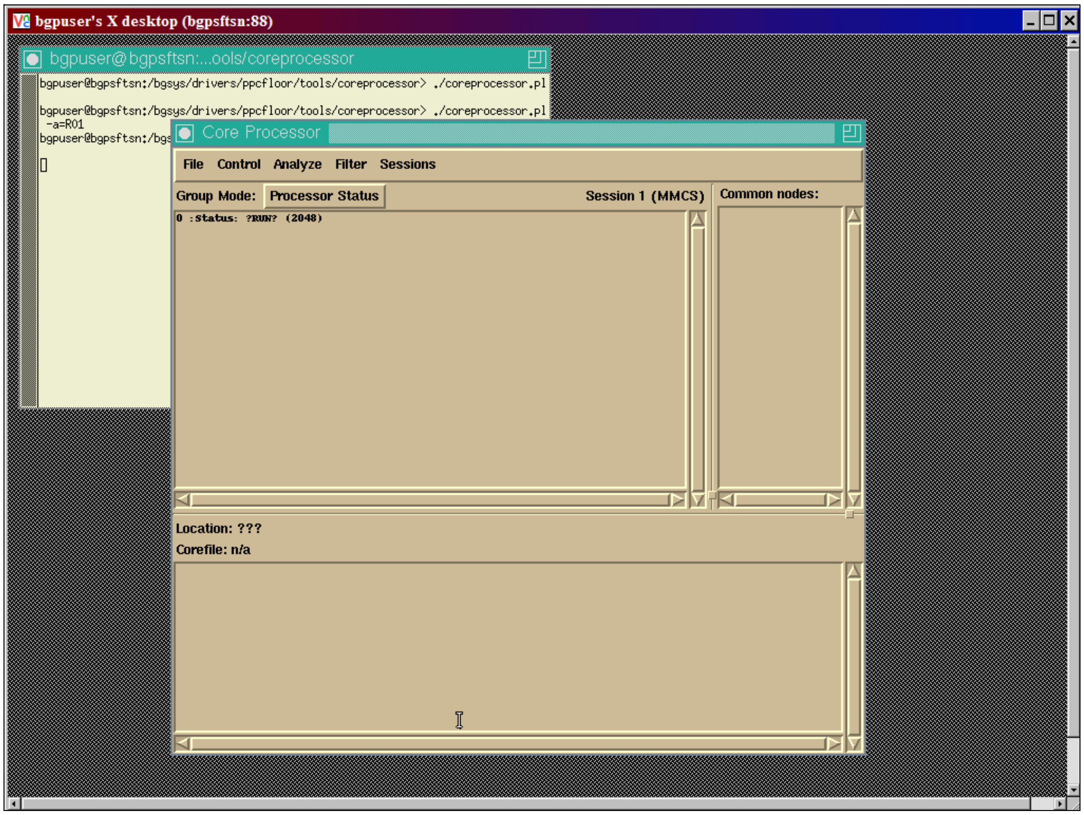
```
bgpuser@bgpsftsn:/bgsys/drivers/ppcfloor/tools/coreprocessor> ./coreprocessor.pl

bgpuser@bgpsftsn:/bgsys/drivers/ppcfloor/tools/coreprocessor> ./coreprocessor.pl
 -a=R01
```

Session Name: Session 1
Block: R01
ION Binary:
CNK Binary:
User: bgpuser
Host: localhost
Port: 32031

Figure 9-2 Core Processor attach window

4. At this point, you have not yet affected the state of the processors. Choose **Select Grouping Mode** → **Processor Status**.

   Notice the text in the upper-left pane (Figure 9-3). The Core Processor tool posts the status `?RUN?` because it does not yet know the state of the processors. `(2048)` is the number of nodes in the block that are in that state. The number in parentheses always indicates the number of nodes that share the attribute displayed on the line, which is the processor state in this case.



```
bgpuser@bgpsftsn:/bgsys/drivers/ppcfloor/tools/coreprocessor> ./coreprocessor.pl

bgpuser@bgpsftsn:/bgsys/drivers/ppcfloor/tools/coreprocessor> ./coreprocessor.pl
 -a=R01
bgpuser@bgpsftsn:/bgs
```

```
0 :status: ?RUN? (2048)
```

Location: ???
Corefile: n/a

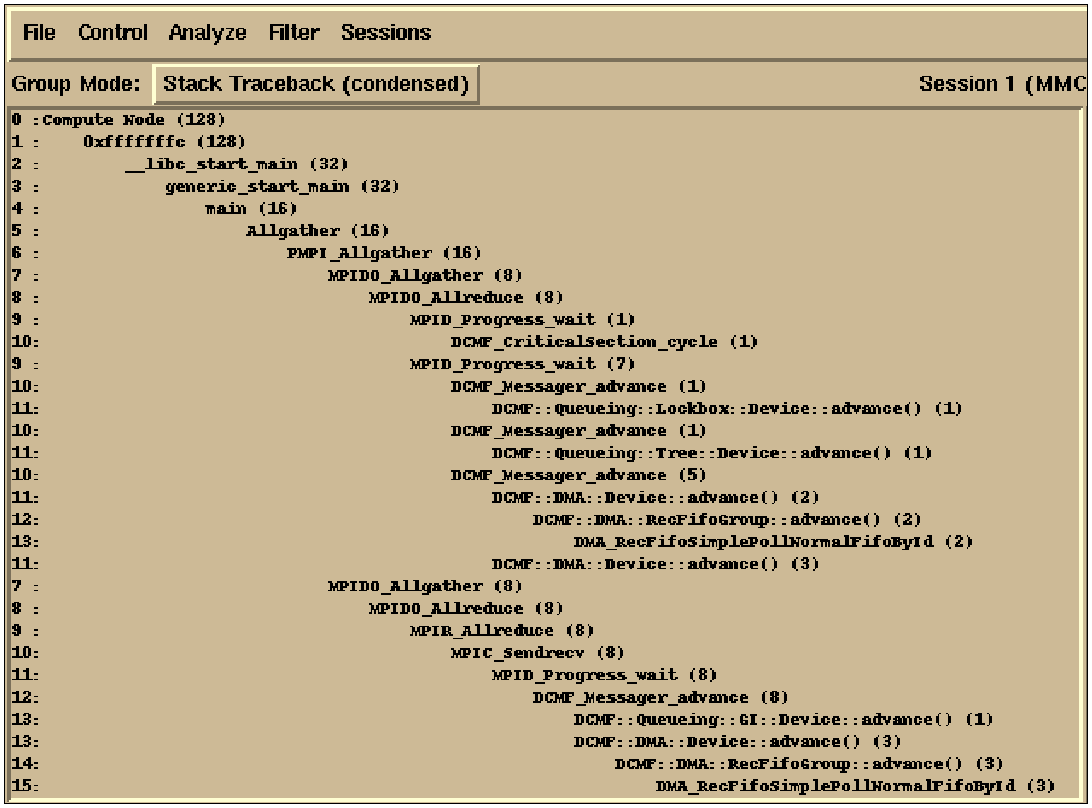
Figure 9-3 Processor status

5. Back at the main window (refer to Figure 9-1 on page 150), click the **Select Grouping Mode** button.

6. Choose one of the **Stack Traceback** options. The Core Processor tool halts all the Compute Node processor cores and displays the requested information. Choose each of the options on that menu in turn so that you can see the variety of data formats available.

Stack Traceback (condensed)

In the condensed version of Stack Traceback, data from all nodes is captured. The unique instruction addresses per stack frame are grouped and displayed. However, the last stack frame is grouped based on the function name, not the IAR. This is normally the most useful mode for debug (Figure 9-4).

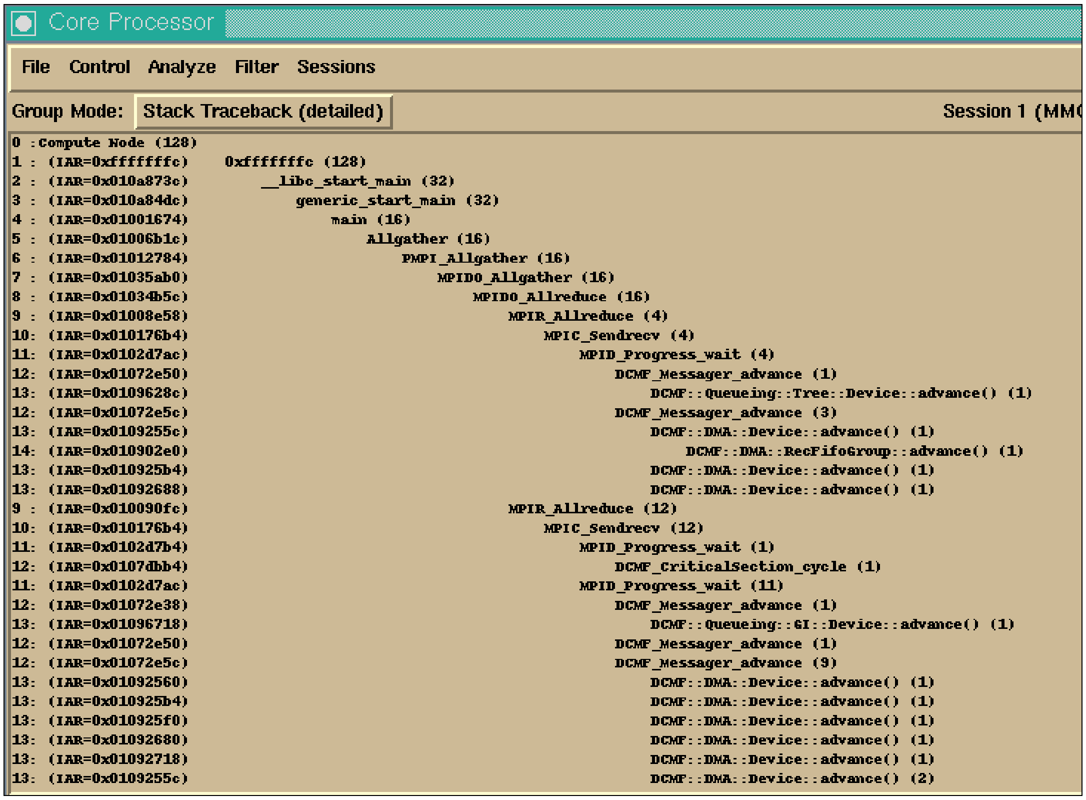


```
0 :Compute Node (128)
1 :    0xfffffffc (128)
2 :        __libc_start_main (32)
3 :            generic_start_main (32)
4 :                main (16)
5 :                    Allgather (16)
6 :                        PMPI_Allgather (16)
7 :                            MPIDO_Allgather (8)
8 :                                MPIDO_Allreduce (8)
9 :                                    MPID_Progress_wait (1)
10:                                        DCMF_CriticalSection_cycle (1)
9 :                                    MPID_Progress_wait (7)
10:                                        DCMF_Messager_advance (1)
11:                                            DCMF::Queueing::Lockbox::Device::advance() (1)
10:                                        DCMF_Messager_advance (1)
11:                                            DCMF::Queueing::Tree::Device::advance() (1)
10:                                        DCMF_Messager_advance (5)
11:                                            DCMF::DMA::Device::advance() (2)
12:                                                DCMF::DMA::RecFifoGroup::advance() (2)
13:                                                    DMA_RecFifoSimplePollNormalFifoById (2)
11:                                            DCMF::DMA::Device::advance() (3)
7 :                            MPIDO_Allgather (8)
8 :                                MPIDO_Allreduce (8)
9 :                                    MPIR_Allreduce (8)
10:                                        MPIC_Sendrecv (8)
11:                                            MPID_Progress_wait (8)
12:                                                DCMF_Messager_advance (8)
13:                                                    DCMF::Queueing::GI::Device::advance() (1)
13:                                                    DCMF::DMA::Device::advance() (3)
14:                                                        DCMF::DMA::RecFifoGroup::advance() (3)
15:                                                            DMA_RecFifoSimplePollNormalFifoById (3)
```

Figure 9-4 Stack Traceback (condensed)

Stack Traceback (detailed)

In Stack Traceback (detailed), data from all nodes is captured (Figure 9-5). The unique instruction addresses per stack frame are grouped and displayed. The IAR at each stack frame is also displayed.

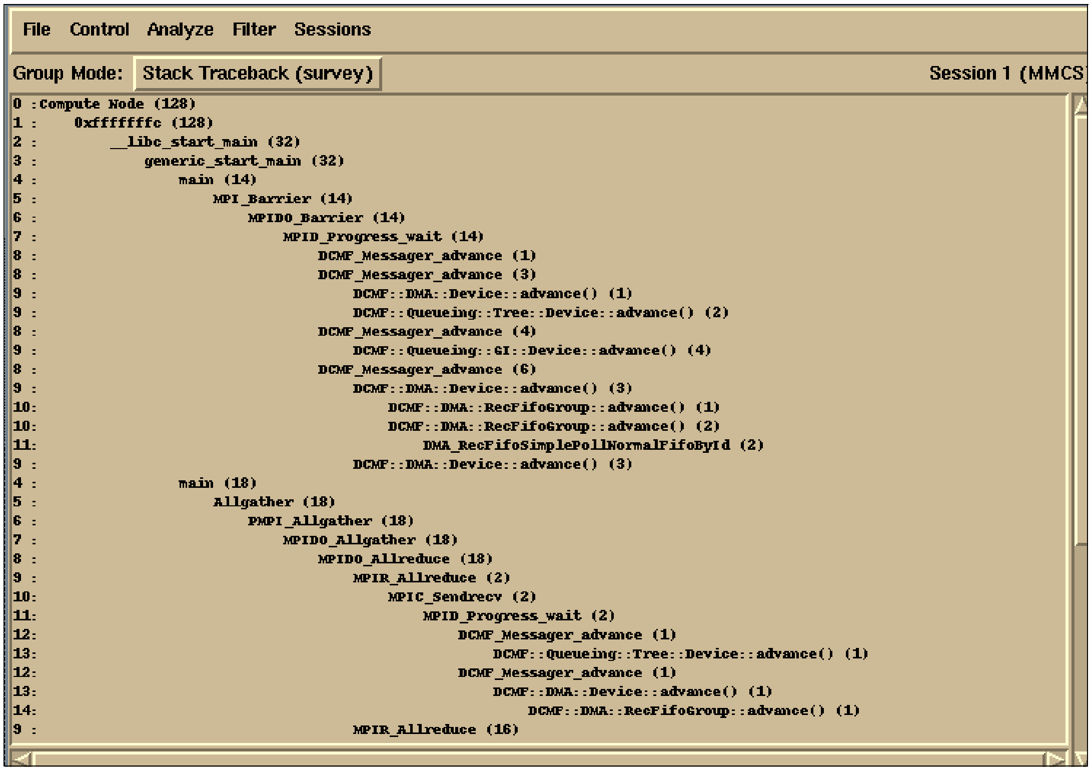


```
0 :Compute Node (128)
1 : (IAR=0xfffffffc)    0xfffffffc (128)
2 : (IAR=0x010a873c)        __libc_start_main (32)
3 : (IAR=0x010a84dc)            generic_start_main (32)
4 : (IAR=0x01001674)                main (16)
5 : (IAR=0x01006b1c)                    Allgather (16)
6 : (IAR=0x01012784)                        PMPI_Allgather (16)
7 : (IAR=0x01035ab0)                            MPIDO_Allgather (16)
8 : (IAR=0x01034b5c)                                MPIDO_Allreduce (16)
9 : (IAR=0x01008e58)                                    MPIR_Allreduce (4)
10: (IAR=0x010176b4)                                        MPIC_Sendrecv (4)
11: (IAR=0x0102d7ac)                                            MPID_Progress_wait (4)
12: (IAR=0x01072e50)                                                DCMF_Messager_advance (1)
13: (IAR=0x0109628c)                                                    DCMF::Queueing::Tree::Device::advance() (1)
12: (IAR=0x01072e5c)                                                DCMF_Messager_advance (3)
13: (IAR=0x0109255c)                                                    DCMF::DMA::Device::advance() (1)
14: (IAR=0x010902e0)                                                        DCMF::DMA::RecFifoGroup::advance() (1)
13: (IAR=0x010925b4)                                                    DCMF::DMA::Device::advance() (1)
13: (IAR=0x01092688)                                                    DCMF::DMA::Device::advance() (1)
9 : (IAR=0x010090fc)                                    MPIR_Allreduce (12)
10: (IAR=0x010176b4)                                        MPIC_Sendrecv (12)
11: (IAR=0x0102d7b4)                                            MPID_Progress_wait (1)
12: (IAR=0x0107dbb4)                                                DCMF_CriticalSection_cycle (1)
11: (IAR=0x0102d7ac)                                            MPID_Progress_wait (11)
12: (IAR=0x01072e38)                                                DCMF_Messager_advance (1)
13: (IAR=0x01096718)                                                    DCMF::Queueing::GI::Device::advance() (1)
12: (IAR=0x01072e50)                                                DCMF_Messager_advance (1)
12: (IAR=0x01072e5c)                                                DCMF_Messager_advance (9)
13: (IAR=0x01092560)                                                    DCMF::DMA::Device::advance() (1)
13: (IAR=0x010925b4)                                                    DCMF::DMA::Device::advance() (1)
13: (IAR=0x010925f0)                                                    DCMF::DMA::Device::advance() (1)
13: (IAR=0x01092680)                                                    DCMF::DMA::Device::advance() (1)
13: (IAR=0x01092718)                                                    DCMF::DMA::Device::advance() (1)
13: (IAR=0x0109255c)                                                    DCMF::DMA::Device::advance() (2)
```

Figure 9-5 Stack Traceback (detailed)

Stack Traceback (survey)

Stack Traceback (survey) is a quick but potentially inaccurate mode. IARs are initially captured, and stack data is collected for each node from a group of nodes that contain the same IAR. The stack data fetched for that one node is then applied to all nodes with the same IAR. Figure 9-6 shows an example of the survey mode.



Group Mode: Stack Traceback (survey)

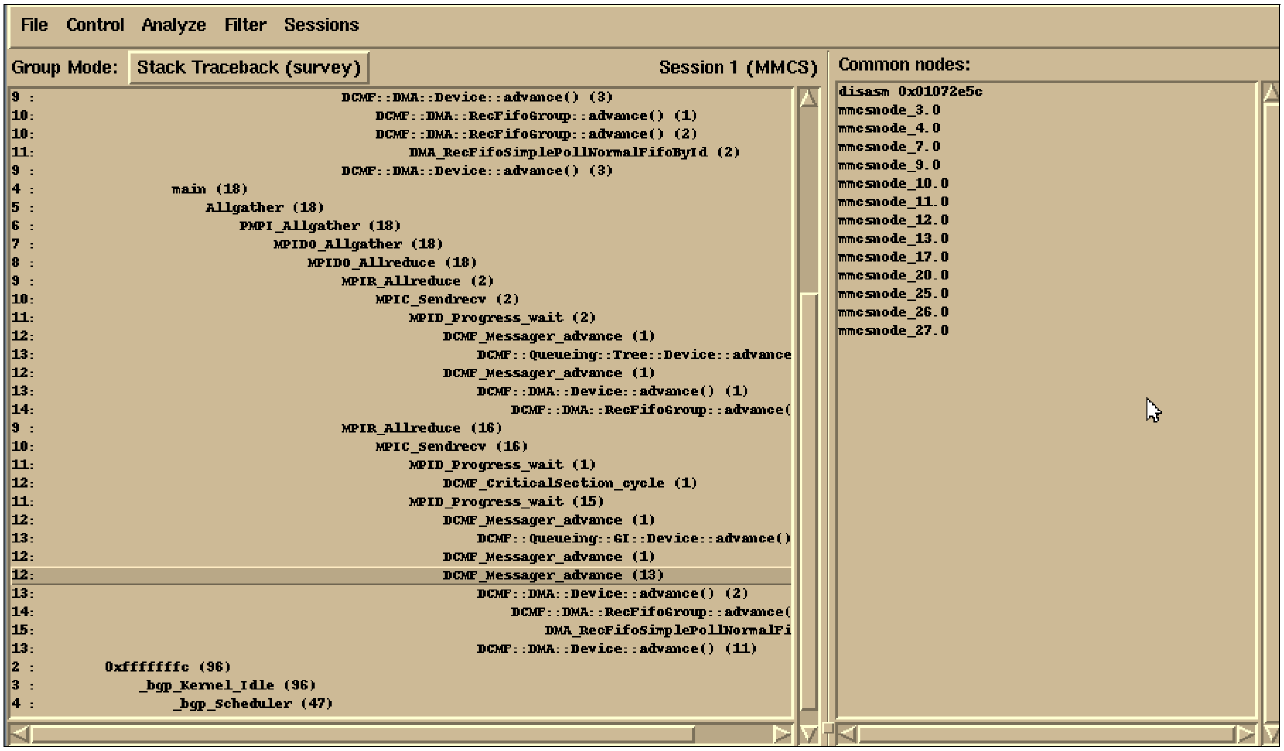
Session 1 (MMCS

```
0 :Compute Node (128)
1 :    0xfffffffc (128)
2 :        __libc_start_main (32)
3 :            generic_start_main (32)
4 :                main (14)
5 :                    MPI_Barrier (14)
6 :                        MPIDO_Barrier (14)
7 :                            MPID_Progress_wait (14)
8 :                                DCMF_Messager_advance (1)
8 :                                DCMF_Messager_advance (3)
9 :                                    DCMF::DMA::Device::advance() (1)
9 :                                    DCMF::Queueing::Tree::Device::advance() (2)
8 :                                DCMF_Messager_advance (4)
9 :                                    DCMF::Queueing::GI::Device::advance() (4)
8 :                                DCMF_Messager_advance (6)
9 :                                    DCMF::DMA::Device::advance() (3)
10:                                        DCMF::DMA::RecFifoGroup::advance() (1)
10:                                        DCMF::DMA::RecFifoGroup::advance() (2)
11:                                            DMA_RecFifoSimplePollNormalFifoById (2)
9 :                                    DCMF::DMA::Device::advance() (3)
4 :                main (18)
5 :                    Allgather (18)
6 :                        PMPI_Allgather (18)
7 :                            MPIDO_Allgather (18)
8 :                                MPIDO_Allreduce (18)
9 :                                    MPIR_Allreduce (2)
10:                                        MPIC_Sendrecv (2)
11:                                            MPID_Progress_wait (2)
12:                                                DCMF_Messager_advance (1)
13:                                                    DCMF::Queueing::Tree::Device::advance() (1)
12:                                                DCMF_Messager_advance (1)
13:                                                    DCMF::DMA::Device::advance() (1)
14:                                                        DCMF::DMA::RecFifoGroup::advance() (1)
9 :                                    MPIR_Allreduce (16)
```

Figure 9-6 Stack Traceback (survey)

Refer to the following points to help you use the tool more effectively:

- The number at the far left, before the colon, indicates the depth within the stack.
- The number in parentheses at the end of each line indicates the number of nodes that share the same stack frame.
- If you click any line in the stack dump, the pane on the right (labeled Common nodes) shows the list of nodes that share that stack frame. See Figure 9-7 on page 156.
- When you click one of the stack frames and then select **Control** → **Run**, the action is performed for all nodes that share that stack frame. A new Processor Status summary is displayed. If you again chose a Stack Traceback option, the running processors are halted and the stacks are refetched.
- You can hold down the Shift key and click several stack frames if you want to control all procedures that are at a range of stack frames.
- From the **Filter** menu option, you can select **Group Selection** → **Create Filter** to add a filter with the name that you specify in the Filter pull-down. When the box for your filter is highlighted, only the data for those processors is displayed in the upper-left window. You can create several filters if you want.
- Set Group Mode to Ungrouped or Ungrouped with Traceback to control one processor at a time.



```
9 :                             DCMF::DMA::Device::advance() (3)
10:                                 DCMF::DMA::RecFifoGroup::advance() (1)
10:                                 DCMF::DMA::RecFifoGroup::advance() (2)
11:                                     DMA_RecFifoSimplePollNormalFifoById (2)
9 :                             DCMF::DMA::Device::advance() (3)
4 :         main (18)
5 :             Allgather (18)
6 :                 PMPI_Allgather (18)
7 :                     MPIDO_Allgather (18)
8 :                         MPIDO_Allreduce (18)
9 :                             MPIR_Allreduce (2)
10:                                 MPIC_Sendrecv (2)
11:                                     MPID_Progress_wait (2)
12:                                         DCMF_Messager_advance (1)
13:                                             DCMF::Queueing::Tree::Device::advance
12:                                         DCMF_Messager_advance (1)
13:                                             DCMF::DMA::Device::advance() (1)
14:                                                 DCMF::DMA::RecFifoGroup::advance(
9 :                             MPIR_Allreduce (16)
10:                                 MPIC_Sendrecv (16)
11:                                     MPID_Progress_wait (1)
12:                                         DCMF_CriticalSection_cycle (1)
11:                                     MPID_Progress_wait (15)
12:                                         DCMF_Messager_advance (1)
13:                                             DCMF::Queueing::GI::Device::advance()
12:                                         DCMF_Messager_advance (1)
12:                                         DCMF_Messager_advance (13)
13:                                             DCMF::DMA::Device::advance() (2)
14:                                                 DCMF::DMA::RecFifoGroup::advance(
15:                                                     DMA_RecFifoSimplePollNormalFi
13:                                             DCMF::DMA::Device::advance() (11)
2 :     0xfffffffc (96)
3 :         _bgp_Kernel_Idle (96)
4 :             _bgp_Scheduler (47)
```

```
disasm 0x01072e5c
mmcsnode_3.0
mmcsnode_4.0
mmcsnode_7.0
mmcsnode_9.0
mmcsnode_10.0
mmcsnode_11.0
mmcsnode_12.0
mmcsnode_13.0
mmcsnode_17.0
mmcsnode_20.0
mmcsnode_25.0
mmcsnode_26.0
mmcsnode_27.0
```

Figure 9-7 Stack Traceback Common nodes

9.2.6 Saving your information

To save the current contents of Traceback information about the upper-left pane, select **File** → **Save Traceback** to a file of your choice.

To gain more complete data, select **File** → **Take Snapshot™**. Notice that you then have two sessions to choose from on the Sessions menu. The original session is (MMCS), and the second one is (SNAP). The snapshot is exactly what the name implies, a picture of the debug session at a particular point. Notice that you cannot start or stop the processors from the snapshot session. You can choose **File** → **Save Snapshot** to save the snapshot to a file. If you are sending data to IBM for debug, Save Snapshot is a better choice than Save Traceback because the snapshot includes objdump data.

If you choose **File** → **Quit** and processors are halted, you are given an option to restart them before quitting.

9.2.7 Debugging live I/O Node problems

It is possible to debug I/O Nodes as well as Compute Nodes, but you normally want to avoid doing so. Collecting data causes the processor to be stopped, and stopping the I/O Node processors can cause problems with your file system. In addition, the Compute Nodes are not able to communicate with the I/O Nodes. If you want to debug an I/O Node, you must specify the I/O Node binary when you select **File** → **Attach** to block the window, and choose **Filter** → **Debug I/O Nodes**.

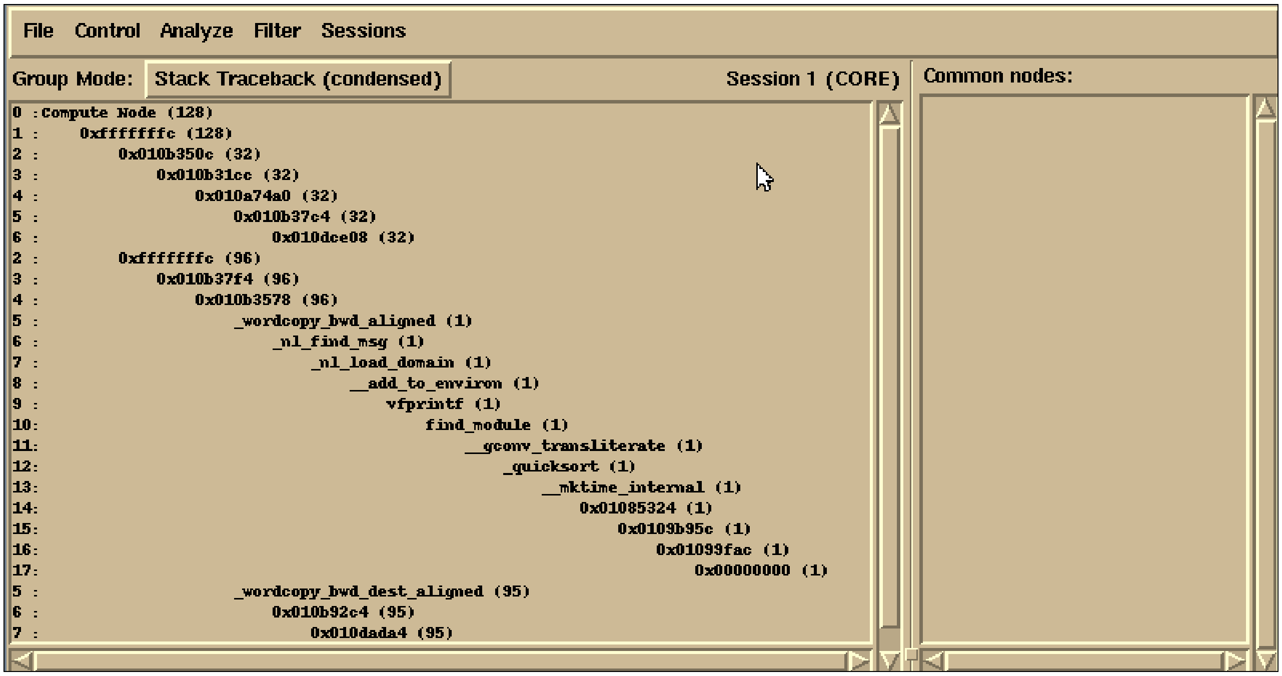
9.2.8 Debugging core files

To work with core files, select **File** → **Load Core**. In the window, specify the following information:

- The location of the Compute Node Kernel binary or binaries
- The core files location
- The lowest and highest-numbered core files that you want to work with (The default is all available core files.)

Click the **Load Cores** button when you have specified the information.

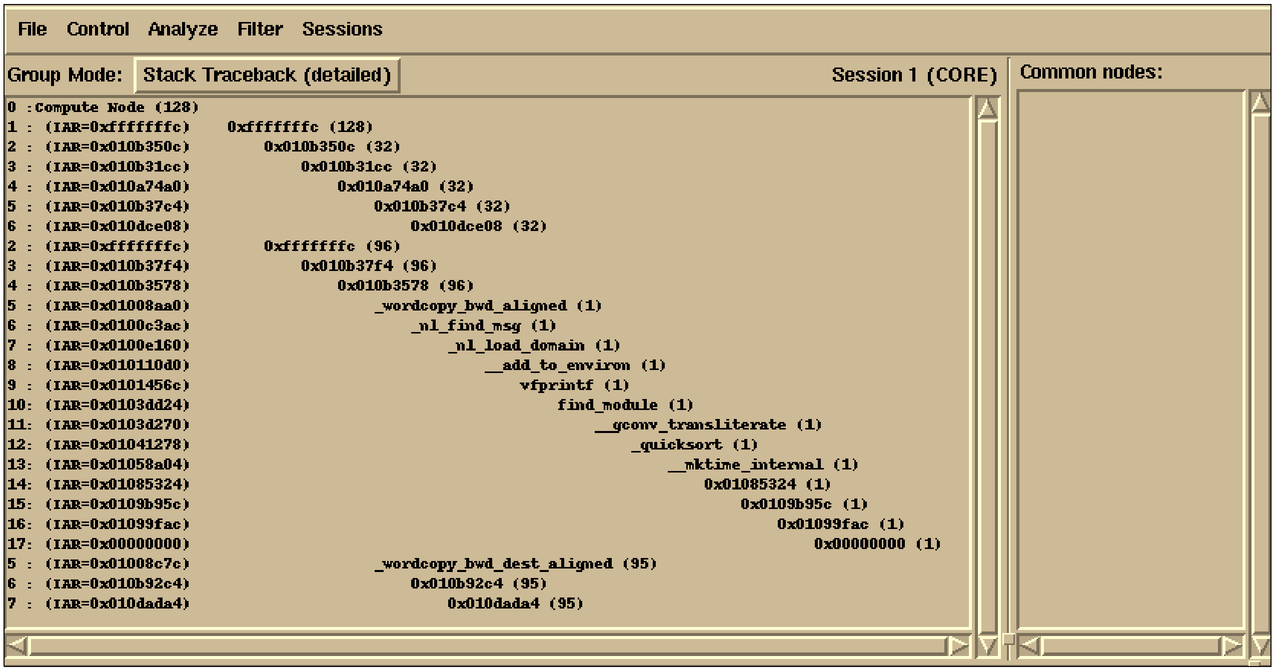
The same Grouping Modes are available for core file debug as for live debug. Figure 9-8 shows an output example of the Condensed Stack Traceback options from a core file. Condensed mode is the easiest format to work with.



```
0 :Compute Node (128)
1 :    0xfffffffc (128)
2 :        0x010b350c (32)
3 :            0x010b31cc (32)
4 :                0x010a74a0 (32)
5 :                    0x010b37c4 (32)
6 :                        0x010dce08 (32)
2 :        0xfffffffc (96)
3 :            0x010b37f4 (96)
4 :                0x010b3578 (96)
5 :                    _wordcopy_bwd_aligned (1)
6 :                        _nl_find_msg (1)
7 :                            _nl_load_domain (1)
8 :                                __add_to_environ (1)
9 :                                    vfprintf (1)
10:                                        find_module (1)
11:                                            __gconv_transliterate (1)
12:                                                _quicksort (1)
13:                                                    __mktime_internal (1)
14:                                                        0x01085324 (1)
15:                                                            0x0109b95c (1)
16:                                                                0x01099fac (1)
17:                                                                    0x00000000 (1)
5 :                    _wordcopy_bwd_dest_aligned (95)
6 :                        0x010b92c4 (95)
7 :                            0x010dada4 (95)
```

Figure 9-8 Core file condensed stack trace

Figure 9-9 shows the detailed version of the same trace.

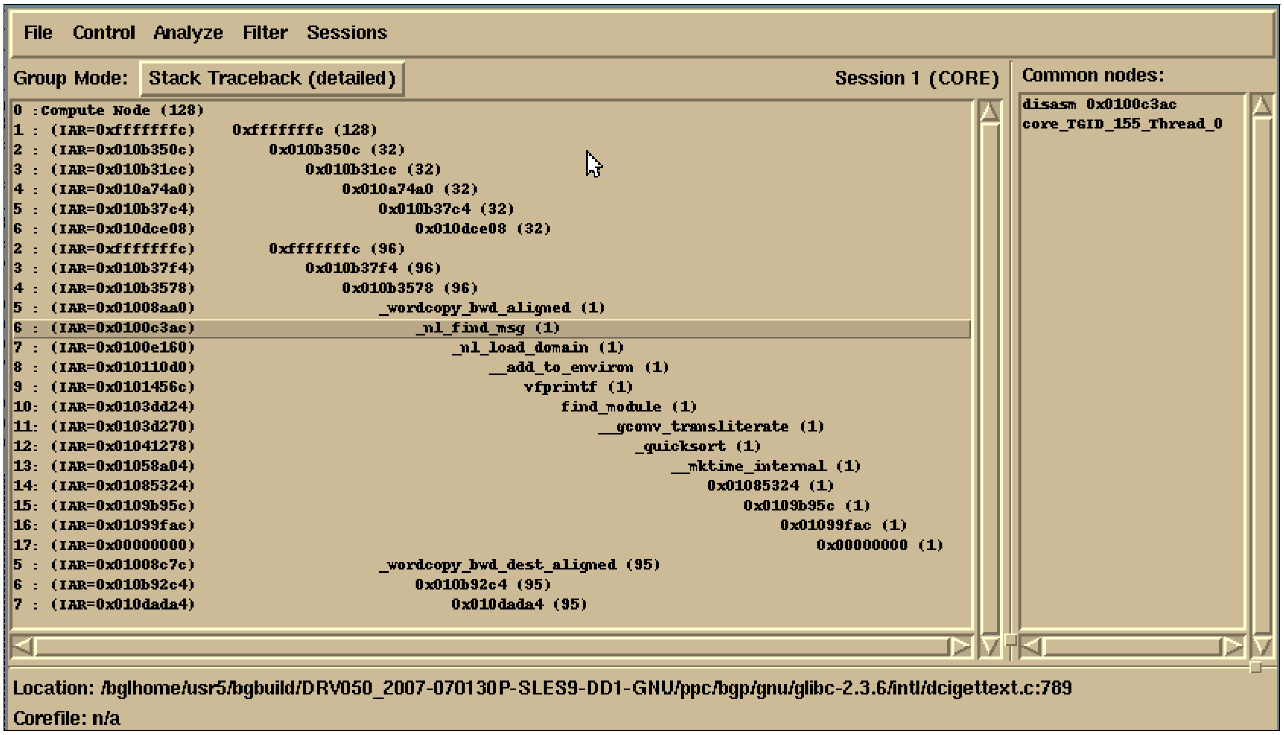


```
0 :Compute Node (128)
1 : (IAR=0xfffffffc)     0xfffffffc (128)
2 : (IAR=0x010b350c)         0x010b350c (32)
3 : (IAR=0x010b31cc)             0x010b31cc (32)
4 : (IAR=0x010a74a0)                 0x010a74a0 (32)
5 : (IAR=0x010b37c4)                     0x010b37c4 (32)
6 : (IAR=0x010dce08)                         0x010dce08 (32)
2 : (IAR=0xfffffffc)         0xfffffffc (96)
3 : (IAR=0x010b37f4)             0x010b37f4 (96)
4 : (IAR=0x010b3578)                 0x010b3578 (96)
5 : (IAR=0x01008aa0)                     _wordcopy_bwd_aligned (1)
6 : (IAR=0x0100c3ac)                         _nl_find_msg (1)
7 : (IAR=0x0100e160)                             _nl_load_domain (1)
8 : (IAR=0x010110d0)                                 __add_to_environ (1)
9 : (IAR=0x0101456c)                                     vfprintf (1)
10: (IAR=0x0103dd24)                                         find_module (1)
11: (IAR=0x0103d270)                                             __gconv_transliterate (1)
12: (IAR=0x01041278)                                                 _quicksort (1)
13: (IAR=0x01058a04)                                                     __mktime_internal (1)
14: (IAR=0x01085324)                                                         0x01085324 (1)
15: (IAR=0x0109b95c)                                                             0x0109b95c (1)
16: (IAR=0x01099fac)                                                                 0x01099fac (1)
17: (IAR=0x00000000)                                                                     0x00000000 (1)
5 : (IAR=0x01008c7c)                     _wordcopy_bwd_dest_aligned (95)
6 : (IAR=0x010b92c4)                         0x010b92c4 (95)
7 : (IAR=0x010dada4)                             0x010dada4 (95)
```

Figure 9-9 Core file detailed stack trace

The Survey option is less useful for core files because speed is not such a concern.

When you select a stack frame in the Traceback output (Figure 9-10), two additional pieces of information are displayed. The core files that share that stack frame are displayed in the Common nodes pane. The Location field under the Traceback pane displays the location of that function and the line number represented by the stack frame. If you select one of the core files in the Common nodes pane, the contents of that core file are displayed in the bottom pane.



```
0 :Compute Node (128)
1 : (IAR=0xfffffffc)    0xfffffffc (128)
2 : (IAR=0x010b350c)        0x010b350c (32)
3 : (IAR=0x010b31cc)            0x010b31cc (32)
4 : (IAR=0x010a74a0)                0x010a74a0 (32)
5 : (IAR=0x010b37c4)                    0x010b37c4 (32)
6 : (IAR=0x010dce08)                        0x010dce08 (32)
2 : (IAR=0xfffffffc)        0xfffffffc (96)
3 : (IAR=0x010b37f4)            0x010b37f4 (96)
4 : (IAR=0x010b3578)                0x010b3578 (96)
5 : (IAR=0x01008aa0)                    _wordcopy_bwd_aligned (1)
6 : (IAR=0x0100c3ac)                        _nl_find_msg (1)
7 : (IAR=0x0100e160)                            _nl_load_domain (1)
8 : (IAR=0x010110d0)                                __add_to_environ (1)
9 : (IAR=0x0101456c)                                    vfprintf (1)
10: (IAR=0x0103dd24)                                        find_module (1)
11: (IAR=0x0103d270)                                            __gconv_transliterate (1)
12: (IAR=0x01041278)                                                _quicksort (1)
13: (IAR=0x01058a04)                                                    __mktime_internal (1)
14: (IAR=0x01085324)                                                        0x01085324 (1)
15: (IAR=0x0109b95c)                                                            0x0109b95c (1)
16: (IAR=0x01099fac)                                                                0x01099fac (1)
17: (IAR=0x00000000)                                                                    0x00000000 (1)
5 : (IAR=0x01008c7c)                    _wordcopy_bwd_dest_aligned (95)
6 : (IAR=0x010b92c4)                        0x010b92c4 (95)
7 : (IAR=0x010dada4)                            0x010dada4 (95)
```

```
disasm 0x0100c3ac
core_TGID_155_Thread_0
```

Location: /bglhome/usr5/bgbuild/DRV050_2007-070130P-SLES9-DD1-GNU/ppc/bgp/gnu/glibc-2.3.6/intl/dcigettext.c:789

Corefile: n/a

Figure 9-10 Core file Common nodes

9.2.9 The addr2line utility

The **addr2line** utility is a standard Linux program. You can find additional information about this utility in any Linux manual as well as at the following Web site:

http://www.linuxcommand.org/man_pages/addr2line1.html

The **addr2line** utility translates an address into file names and line numbers. Using an address and an executable, this utility uses the debugging information in the executable to provide information about the file name and line number. To take advantage of this utility, compile your program with the -g option. On the Blue Gene/P system, the core file is a plain text file that you can view with the vi editor.

You can the use the Linux **addr2line** command on the Front End Node and enter the address found in the core file and the -g executable. Then the utility points you to the source line where the problem occurred.

Example 9-8 on page 160 shows a core file and how to use the **addr2line** utility to identify potential problems in the code. In this particular case, the program was *not* compiled with the -g flag option because this was a production run. However, notice in Example 9-8 on page 160 that **addr2line** points to malloc(). This can be a hint that perhaps the amount of memory is insufficient to run this particular calculation, or some other problems might be related to the usage of malloc() in the code.

Example 9-8 Using addr2line to identify potential problems in your code

```
vi core.0 and select the addresses between +++STACK and ---STACK and use them as input for
addr2line
+++STACK
0x01342cb8
0x0134653c
0x0106e5f8
0x010841ec
0x0103946c
0x010af40c
0x010b5e44
0x01004fa0
0x010027cc
0x0100c028
0x0100133c
0x013227ec
0x01322a4c
0xfffffffc
---STACK

Run addr2line with your executable
$addr2line -e a.out
0x01342cb8
0x0134653c
0x0106e5f8
0x010841ec
0x0103946c
0x010af40c
0x010b5e44
0x01004fa0
0x010027cc
0x0100c028
0x0100133c
0x013227ec
0x01322a4c
0xfffffffc/bglhome/usr6/bgbuild/DRV360_2007-070906P-SLES10-DD2-GNU10/ppc/bgp/gnu/glibc-2.4/mallo
c/malloc.c:3377
/bglhome/usr6/bgbuild/DRV360_2007-070906P-SLES10-DD2-GNU10/ppc/bgp/gnu/glibc-2.4/malloc/malloc.c
:3525
modify.cpp:0
??:0
??:0
??:0
??:0
main.cpp:0
main.cpp:0
main.cpp:0
??:0
../csu/libc-start.c:231
../sysdeps/unix/sysv/linux/powerpc/libc-start.c:127
```

9.2.10 Scalable Debug API

In this section we describe the Scalable Debug API. This API provides the capability of examining a job running on the compute nodes. APIs are available to attach to a running job and extract the current stack information for that job. The API also enables the user to select specific nodes of interest and to export information that a debugger or other tool could use to further analyze a job.

The Scalable Debug API enables the user to interface with the compute nodes from the Front End Node. When the API is initialized, a process is started on the Front End Node. The process on the Front End Node uses the mpirun framework to launch a process on the service node. This back-end process uses the Bridge APIs to launch a debug tool. The tool runs on the I/O Nodes and communicates with CIOD using the CioDebuggerProtocol. The tool pauses the compute nodes that are associated with the I/O Node and gathers the stack information of each task to be sent back to the Front End Node. The raw data then is available to the application using the Scalable Debug API to determine which nodes might be of interest for further analysis.

An application using the Scalable Debug APIs completes these steps:

1. Call the initialization function to set up the API
2. Attach to a job being run by the current user
3. Get the extracted stack data and proctable information for the job
4. Add or remove any nodes that require further analysis
5. Export the node list to the specified file
6. Detach from the job

The rest of this section covers the following topics:

- API requirements
- API specification
- Example code

API requirements

The following requirements are for writing programs using the Scalable Debug API:

- Currently, SUSE Linux Enterprise Server (SLES) 10 for PowerPC is the only supported platform.
- C and C++ are supported with the GNU gcc V4.1.2 level compilers. For more information and downloads, refer to the following Web address:

  http://gcc.gnu.org/
- Required include files are installed in /bgsys/drivers/ppcfloor/tools/ScalableDebug/include and /bgsys/drivers/ppcfloor/include. See Appendix C, “Header files and libraries” on page 335 for additional information about include files. The include file for the Scalable Debug API is ScalableDebug.h.
- The Scalable Debug API supports applications that use dynamic linking using shared objects. The required library file is /bgsys/drivers/ppcfloor/tools/ScalableDebug/lib/libscalabledebug.so for 32-bit, or /bgsys/drivers/ppcfloor/tools/ScalableDebug/lib64/libscalabledebug.so for 64-bit.

The include files and shared objects are installed with the standard system installation procedure. They are contained in the bgpbase.rpm file.

API specification

This section describes the functions, return codes, and data structures that make up the Scalable Debug API.

Functions

This section describes the functions in the Scalable Debug API.

The functions return a status code that indicates success or failure, along with the reason for the failure. An exit value of SDBG_NO_ERROR or 0 (zero) indicates that the function was successful, while a non-zero return code indicates failure.

These functions are not thread safe. They must be called only from a single thread.

The following functions are in the Scalable Debug API:

- `int SDBG_Init();`

  This initialization function must be called before any other functions in the Scalable Debug API.

  This function returns the following values:

  - SDBG_NO_ERROR
  - SDBG_FORK_FE_ERROR
  - SDBG_EXEC_FE_ERROR
  - SDBG_CONNECT_FE_ERROR

- `int SDBG_AttachJob(unsigned int jobId);`

  Attach to a running job. `jobId` is the database job identifier from MMCS. This function fails if the user calling the function is not the user that started the job.

  This function returns the following values:

  - SDBG_NO_ERROR
  - SDBG_INIT_NOT_CALLED
  - SDBG_CONNECT_FE_ERROR
  - SDBG_TIMEOUT_ERROR
  - SDBG_JOB_NOT_RUNNING
  - SDBG_JOB_HTC
  - SDBG_JOB_USER_DIFF
  - SDBG_TOOL_SETUP_FAIL
  - SDBG_TOOL_LAUNCH_FAIL
  - SDBG_PROC_TABLE

- `int SDBG_DetachJob();`

  Detach from a running job.

  This function returns the following values:

  - SDBG_NO_ERROR
  - SDBG_CONNECT_FE_ERROR

- `int SDBG_GetStackData(uint32_t *numMsgs, SDBG_StackMsg_t **stackMsgPtr);`

  Get the stack data for the job. `numMsgs` is set to the number of stack data messages put in `stackMsgPtr`. When the application is finished using the stack data, it must free the stack data using SDBG_FreeStackData() to prevent a memory leak.

  This function returns the following values:

  - SDBG_NO_ERROR
  - SDBG_NO_MEMORY

- `int SDBG_FreeStackData(uint32_t numMsgs, SDBG_StackMsg_t *stackMsgPtr);`

  Free the stack data allocated by the library. numMsgs and stackMsgPtr are the values set by SDBG_GetStackData().

  – This function returns the following value:

  SDBG_NO_ERROR

- `int SDBG_GetProcTable(uint32_t *numTableEntries, MPIR_PROCDESC **procTablePtr);`

  Get the proc table data for job. `numTableEntries` is set to the number of proc table entries put in `procTablePtr`. When the application is finished using the proc table, it must free the proc table using SDBG_FreeProcTable() to prevent a memory leak.

  This function returns the following values:

  – SDBG_NO_ERROR
  – SDBG_NO_MEMORY

- `int SDBG_FreeProcTable(uint32_t numTableEntries, MPIR_PROCDESC *procTablePtr);`

  Free the proc table data allocated by the library.

  This function returns the following value:

  SDBG_NO_ERROR

- `int SDBG_AddNode(unsigned int rank);`

  Add a node to subset attach list.

  This function returns the following values:

  – SDBG_NO_ERROR
  – SDBG_RANK_NOT_FOUND

- `int SDBG_RemoveNode(unsigned int rank);`

  Remove a node from subset attach list.

  This function returns the following values:

  – SDBG_NO_ERROR
  – SDBG_RANK_NOT_FOUND

- `int SDBG_ExportNodeList(const char *exportFile);`

  Export node list to the specified file, or to standard output. If `exportFile` is NULL, the node list is printed to standard output.

  This function returns the following values:

  – SDBG_NO_ERROR
  – SDBG_FILE_ERROR

- `int SDBG_ClearNodeList();`

  Clear node list of all nodes.

  This function returns the following value:

  SDBG_NO_ERROR

Return codes

This section summarizes the following return codes used by the Scalable Debug API:

- SDBG_NO_ERROR: No error.
- SDBG_TIMEOUT_ERROR: Timeout communicating with front-end process.
- SDBG_JOBID_NOT_FOUND: Job ID passed not found.

- SDBG_JOB_NOT_RUNNING: Job not running
- SDBG_JOB_HTC: HTC Jobs not supported.
- SDBG_JOB_USER_DIFF: User ID running job different than API user.
- SDBG_TOOL_SETUP_FAIL: Tool setup in database failed.
- SDBG_TOOL_LAUNCH_FAIL: Tool launch on IO Node failed.
- SDBG_PROC_TABLE: Unable to get proc table information.
- SDBG_FILE_ERROR: Error opening Export file passed in.
- SDBG_RANK_NOT_FOUND: Rank not found in proc table.
- SDBG_FORK_FE_ERROR: Unable to fork front-end process.
- SDBG_EXEC_FE_ERROR: Unable to exec front-end process.
- SDBG_CONNECT_FE_ERROR: Unable to connect to front-end process.
- SDBG_INIT_NOT_CALLED: SDBG_Init not called successfully.
- SDBG_NO_MEMORY: No memory available to return data.

Structures

This section describes the structures defined by the Scalable Debug API.

The `SDBG_StackMsg_t` structure contains the data returned when getting stack data. The following fields are in the structure:

- `uint32_t node;`

  Node or pid.

- `uint32_t rank;`

  Rank as defined in proctable.

- `uint32_t threadId;`

  Thread ID for this node.

- `uint32_t linkReg;`

  Current link register.

- `uint32_t iar;`

  Current instruction register.

- `uint32_t currentFrame;`

  Current stack frame (R1).

- `uint32_t numStackFrames;`

  Number of stack frames in stackFramesPtr.

- `SDBG_StackFrame_t *stackFramesPtr;`

  Pointer to array of stack frames. This structure is NULL if there is no stack data.

The `SDBG_StackFrame_t` structure contains the saved frame address and saved link register when looking at stack data. The following fields are in the structure:

- `uint32_t frameAddr;`

  Stack frame address.

- `uint32_t savedLR;`

  Saved link register for this stack frame address.

Environment variable

One environment variable affects the operation of the Scalable Debug API. If the MPIRUN_CONFIG_FILE environment variable is set, its value is used as the mpirun configuration file name. The mpirun configuration file contains the shared secret needed for the API to authenticate with the mpirun daemon on the service node. If not specified, the mpirun configuration file is located by looking for these files in order: /etc/mpirun.cfg or <release-dir>/bin/mpirun.cfg (where <release-dir> is the Blue Gene/P system software directory, for example, /bgsys/drivers/V1R2M0_200_2008-080513P/ppc).

Example code

Example 9-9 illustrates use of the Scalable Debug API.

Example 9-9 Sample code using the Scalable Debug API

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <ScalableDebug.h>
#include <unistd.h>
#include <attach_bg.h>

void printStackData( unsigned int numMsgs, SDBG_StackMsg_t *stackMsgPtr);
void printProcTable( unsigned int numEntries, MPIR_PROCDESC *procTablePtr);

int main(int argc, char **argv)
{
  setbuf(stdout, NULL); setbuf(stderr, NULL);

  if ( argc != 2 )
  {
    printf("Invoke passing jobid to attach to\n");
    return(1);
  }

  int jobId = atoi( argv[1] );

  printf("MAIN: Calling SDBG_Init\n");

  int rc = SDBG_Init(NULL);

  printf("MAIN: Back from SDBG_Init, rc=%d\n", rc);

  printf("MAIN: Calling SDBG_AttachJob with jobid=%i\n", jobId);

  rc = SDBG_AttachJob(jobId);
```

```
printf("MAIN: Back from SDBG_AttachJob, rc=%d\n", rc);

printf("MAIN: Calling SDBG_GetProcTable\n");

MPIR_PROCDESC *procTablePtr;
unsigned int numEntries;
rc = SDBG_GetProcTable(&numEntries, &procTablePtr);

printf("MAIN: Back from SDBG_GetProcTable, numEntries=%u, rc=%d\n", numEntries, rc);

if ( rc == 0 )
{
  printProcTable( numEntries, procTablePtr );
}

printf("MAIN: Calling SDBG_GetStackData\n");
unsigned int numStackMsgs;
SDBG_StackMsg_t *stackMsgPtr;  // Pointer for stack message buffer data
rc = SDBG_GetStackData(&numStackMsgs, &stackMsgPtr);

printf("MAIN: Back from SDBG_GetStackData, numStackMsgs=%u, rc=%d\n", numStackMsgs, rc);

if ( rc == 0 )
{
  printStackData( numStackMsgs, stackMsgPtr );
}

printf("MAIN: Calling SDBG_AddNode with rank=2\n");

rc = SDBG_AddNode( 2 );

printf("MAIN: Back from SDBG_AddNode, rc=%d\n", rc);

printf("MAIN: Calling SDBG_ExportNodeList after adding node 2\n");

rc = SDBG_ExportNodeList( NULL );

printf("MAIN: Back from SDBG_ExportNodeList, rc=%d\n", rc);

printf("MAIN: Calling SDBG_RemoveNode with rank=2\n");

rc = SDBG_RemoveNode( 2 );

printf("MAIN: Back from SDBG_RemoveNode, rc=%d\n", rc);

printf("MAIN: Calling SDBG_ExportNodeList after remove of node 2\n");

rc = SDBG_ExportNodeList( NULL );

printf("MAIN: Back from SDBG_ExportNodeList, rc=%d\n", rc);

printf("MAIN: Calling SDBG_DetachJob\n");

rc = SDBG_DetachJob();
```

```
  printf("MAIN: Back from SDBG_DetachJob, rc=%d\n", rc);

  printf("MAIN: Freeing storage allocated by libary\n");

  rc = SDBG_FreeStackData(numStackMsgs, stackMsgPtr);

  printf("MAIN: Back from SDBG_FreeStackData, rc=%d\n", rc);

  rc = SDBG_FreeProcTable(numEntries, procTablePtr);

  printf("MAIN: Back from SDBG_FreeProcTable, rc=%d\n", rc);

  return 0;

}

void printProcTable( unsigned int numEntries, MPIR_PROCDESC *procTablePtr )
{
  unsigned int i = 0;

  for ( i = 0; i < numEntries; i++ )
  {
    printf("rank=%u, HostName=%s, Exe=%s, pid=%d\n",
           i, procTablePtr[ i ].host_name, procTablePtr[ i ].executable_name,
           procTablePtr[ i ].pid);

  }

}


void printStackData( unsigned int numMsgs, SDBG_StackMsg_t *stackMsgPtr)
{
  SDBG_StackMsg_t *msgPtr = stackMsgPtr;
  unsigned int i = 0;

  for ( i = 0; i < numMsgs; i++ )
  {
    printf("Stack Data node=%u, rank=%u, threadid=%u, numFrames=%u ",
           msgPtr->node, msgPtr->rank, msgPtr->threadId, msgPtr->numStackFrames);

    if ( msgPtr->numStackFrames )
    {
      printf(", first saved frame addr=0x%08x, first saved LR=0x%08x",
             msgPtr->stackFramesPtr->frameAddr, msgPtr->stackFramesPtr->savedLR);
    }
    printf("\n");
    msgPtr++;
  }

}
```

10

Checkpoint and restart support for applications

In this chapter, we provide details about the checkpoint and restart support provided by the IBM Blue Gene/P system. The contents of this chapter reflect the information presented in *IBM System Blue Gene Solution: Application Development*, SG24-7179 but have been updated for the Blue Gene/P system.

Scientific and engineering applications tend to consume most of the compute cycles on high-performance computers. This is certainly the case on the Blue Gene/P system. Many of the simulations run for extended periods of time and checkpoint and restart capabilities are critical for fault recovery.

Checkpoint and restart capabilities are critical for fault recovery. If an application is running for a long period of time, you do not want it to fail after consuming many hours of compute cycles, losing all the calculations made up until the failure. By using checkpoint and restart, you can restart the application at the last checkpoint position, losing a much smaller slice of processing time. In addition, checkpoint and restart are helpful in cases where the given access to a Blue Gene/P system is in relatively small increments of time and you know that your application run will take longer than your allotted amount of processing time. With checkpoint and restart capabilities, you can execute your application in fragmented periods of time rather than an extended interval of time.

We discuss the following topics in this chapter:

- Checkpoint and restart
- Technical overview
- Checkpoint API
- Directory and file-naming conventions
- Restart

10.1 Checkpoint and restart

Checkpoint and restart are among the primary techniques for fault recovery. A special user-level checkpoint library has been developed for Blue Gene/P applications. Using this library, application programs can take a checkpoint of their program state at the appropriate stages. Then the program can be restarted later from the last successful checkpoint.

10.2 Technical overview

The *checkpoint library* is a user-level library that provides support for user-initiated checkpoints in parallel applications. The current implementation requires application developers to insert calls manually to checkpoint library functions at proper places in the application code. However, the restart is transparent to the application and requires only the user or system to set specific environment variables while launching the application.

The application is expected to make a call to the BGCheckpointInit() function at the beginning of the program, to initialize the checkpoint-related data structures and to carry out an automated restart when required. The application can then make calls to the BGCheckpoint() function to store a snapshot of the program state in stable storage (files on a disk). The current model assumes that when an application must take a checkpoint, all of the following points are true:

- All processes of the application make a call to the BGCheckpoint() function.
- When a process makes a call to BGCheckpoint(), no outstanding messages are in the network or buffers. That is, the **recv** that corresponds to all the send calls has occurred.
- After a process has made a call to BGCheckpoint(), other processes do not send messages to the process until their checkpoint is complete. Typically, applications are expected to place calls to BGCheckpoint() immediately after a barrier operation, such as MPI_Barrier(), or after a collective operation, such as MPI_Allreduce(), when no outstanding messages are in the Message Passing Interface (MPI) buffers and the network.

BGCheckpoint() can be called multiple times. Successive checkpoints are identified and distinguished by a checkpoint sequence number. A program state that corresponds to different checkpoints is stored in separate files. It is possible to safely delete the old checkpoint files after a newer checkpoint is complete.

The data that corresponds to the checkpoints is stored in a user-specified directory. A separate checkpoint file is made for each process. This checkpoint file contains header information and a dump of the process's memory, including its data and stack segments, but excluding its text segment and read-only data. It also contains information that pertains to the input/output (I/O) state of the application, including open files and the current file positions.

For restart, the same job is launched again with the environment variables BG_CHKPTRESTARTSEQNO and BG_CHKPTDIRPATH set to the appropriate values. The BGCheckpointInit() function checks for these environment variables and, if specified, restarts the application from the desired checkpoint.

10.2.1 Input/output considerations

All the external I/O calls made from a program are shipped to the corresponding I/O Node using a function-shipping procedure implemented in the Compute Node Kernel.

The checkpoint library intercepts calls to the following main file I/O functions:

- open()
- close()
- read()
- write()
- lseek()

The function name open() is a weak alias that maps to the **`_libc_open`** function. The checkpoint library intercepts this call and provides its own implementation of open() that internally uses the **`_libc_open`** function.

The library maintains a file state table that stores the file name, current file position, and the mode of all the files that are currently open. The table also maintains a translation that translates the file descriptors used by the Compute Node Kernel to another set of file descriptors to be used by the application. While taking a checkpoint, the file state table is also stored in the checkpoint file. Upon a restart, these tables are read. Also the corresponding files are opened in the required mode, and the file pointers are positioned at the desired locations as given in the checkpoint file.

The current design assumes that the programs either always read the file or write the files sequentially. A read followed by an overlapping write, or a write followed by an overlapping read, is not supported.

10.2.2 Signal considerations

Applications can register handlers for signals using the signal() function call. The checkpoint library intercepts calls to signal() and installs its own signal handler instead. It also updates a signal-state table that stores the address of the signal handler function (**`sighandler`**) registered for each signal (**`signum`**). When a signal is raised, the checkpoint signal handler calls the appropriate application handler given in the signal-state table.

While taking checkpoints, the signal-state table is also stored in the checkpoint file in its signal-state section. At the time of restart, the signal-state table is read, and the checkpoint signal handler is installed for all the signals listed in the signal-state table. The checkpoint handler calls the required application handlers when needed.

Signals during checkpoint

The application can potentially receive signals while the checkpoint is in progress. If the application signal handlers are called while a checkpoint is in progress, it can change the state of the memory being checkpointed. This can make the checkpoint inconsistent. Therefore, the signals arriving while a checkpoint is under progress must be handled carefully.

For certain signals, such as SIGKILL and SIGSTOP, the action is fixed, and the application terminates without much choice. The signals without any registered handler are simply ignored. For signals with installed handlers, the two choices are as follows:

- Deliver the signal immediately.
- Postpone the signal delivery until the checkpoint is complete.

All signals are classified into one of these two categories as shown in Table 10-1. If the signal must be delivered immediately, the memory state of the application might change, making the current checkpoint file inconsistent. Therefore, the current checkpoint must be aborted. The checkpoint routine periodically checks if a signal has been delivered since the current checkpoint began. In case a signal has been delivered, it aborts the current checkpoint and returns to the application.

For signals that are to be postponed, the checkpoint handler simply saves the signal information in a pending signal list. When the checkpoint is complete, the library calls application handlers for all the signals in the pending signal list. If more than one signal of the same type is raised while the checkpoint is in progress, the checkpoint library ensures that the handler registered by the application is called at least once. However, it does not guarantee in-order-delivery of signals.

Table 10-1 Action taken on signal

Signal name	Signal type	Action to be taken
SIGINT	Critical	Deliver
SIGXCPU	Critical	Deliver
SIGILL	Critical	Deliver
SIGABRT/SIGIOT	Critical	Deliver
SIGBUS	Critical	Deliver
SIGFPE	Critical	Deliver
SIGSTP	Critical	Deliver
SIGSEGV	Critical	Deliver
SIGPIPE	Critical	Deliver
SIGSTP	Critical	Deliver
SIGSTKFLT	Critical	Deliver
SIGTERM	Critical	Deliver
SIGHUP	Non-critical	Postpone
SIGALRM	Non-critical	Postpone
SIGUSR1	Non-critical	Postpone
SIGUSR2	Non-critical	Postpone
SIGTSTP	Non-critical	Postpone
SIGVTALRM	Non-critical	Postpone
SIGPROF	Non-critical	Postpone
SIGPOLL/SIGIO	Non-critical	Postpone
SIGSYS/SIGUNUSED	Non-critical	Postpone
SIGTRAP	Non-critical	Postpone

Signals during restart

The pending signal list is not stored in the checkpoint file. Therefore, if an application is restarted from a checkpoint, the handlers for pending signals received during the checkpoint are not called. If some signals are raised while the restart is in progress, they are ignored. The checkpoint signal handlers are installed only after the memory state, I/O state, and signal-state table have been restored. This ensures that, when the application signal handlers are called, they see a consistent memory and I/O state.

10.3 Checkpoint API

The checkpoint interface consists of the following items:

- A set of library functions used by the application developer to *checkpoint enable* the application
- A set of conventions used to name and store the checkpoint files
- A set of environment variables used to communicate with the application

This section describes each of these components in detail.

> **Note:** Blue Gene/P supplied checkpoint and restart APIs are not supported for HTC applications.

To ensure minimal overhead, the basic interface has been kept fairly simple. Ideally, a programmer must call only two functions, one at the time of initialization and the other at the places where the application must be checkpointed. Restart is done transparently using the environment variable `BG_CHKPTRESTARTSEQNO` specified at the time of job launch. Alternatively, an explicit restart API is also provided to the programmer to manually restart the application from a specified checkpoint. The remainder of this section describes in detail the functions that make up the checkpoint API.

void BGCheckpointInit(char * ckptDirPath)

BGCheckpointInit() is a *mandatory function* that must be invoked at the *beginning* of the program. You use this function to initialize the data structures of the checkpoint library. In addition, you use this function for transparent restart of the application program.

The `ckptDirPath` parameter specifies the location of checkpoint files. If `ckptDirPath` is NULL, then the default checkpoint file location is assumed as explained in 10.4, “Directory and file-naming conventions” on page 175.

int BGCheckpoint()

BGCheckpoint() takes a *snapshot of the program state* at the instant at which it is called. All the processes of the application must make a call to BGCheckpoint() to take a consistent global checkpoint.

When a process makes a call to BGCheckpoint(), no outstanding messages should be in the network or buffers. That is, the **`recv`** that corresponds to all the send calls should have occurred. In addition, after a process has made a call to BGCheckpoint(), other processes must not send messages to the process until their call to BGCheckpoint() is complete. Typically, applications are expected to place calls to BGCheckpoint() immediately after a barrier operation, such as MPI_Barrier(), or after a collective operation, such as MPI_Allreduce(), when no outstanding message is in the MPI buffers and the network.

The state that corresponds to each application process is stored in a separate file. The location of checkpoint files is specified by `ckptDirPath` in the call to BGCheckpointInit(). If `ckptDirPath` is NULL, then the checkpoint file location is decided by the storage rules mentioned in 10.4, "Directory and file-naming conventions" on page 175.

void BGCheckpointRestart(int restartSqNo)

`BGCheckpointRestart()` restarts the application from the checkpoint given by the argument `restartSqNo`. The directory where the checkpoint files are searched is specified by `ckptDirPath` in the call to **`BGCheckpointInit()`**. If `ckptDirPath` is NULL, then the checkpoint file location is decided by the storage rules provided in 10.4, "Directory and file-naming conventions" on page 175.

An application developer does not need to explicitly invoke this function. BGCheckpointInit() automatically invokes this function whenever an application is restarted. The environment variable `BG_CHKPTRESTARTSEQNO` is set to an appropriate value. If the **`restartSqNo`**, the environment variable `BG_CHKPTRESTARTSEQNO`, is zero, then the system picks up the most recent consistent checkpoint files. However, the function is available for use if the developer chooses to call it explicitly. The developer must know the implications of using this function.

int BGCheckpointExcludeRegion(void *addr, size_t len)

BGCheckpointExcludeRegion() marks the specified region (`addr to addr + len - 1`) to be excluded from the program state, while a checkpoint is being taken. The state that corresponds to this region is not saved in the checkpoint file. Therefore, after restart the corresponding memory region in the application is not overwritten. You can use this facility to protect critical data that should not be restored at the time of restart such as personality and checkpoint data structures. An application programmer can also use this call to exclude a scratch data structure that does not have to be saved at checkpoint time.

int BGAtCheckpoint((void *) function(void *arg), void *arg)

BGAtCheckpoint() registers the functions to be called just before taking the checkpoint. You can use this function to take some action at the time of checkpoint, for example, you can call this function to close all the communication states open at the time of checkpoint. The functions registered are called in the reverse order of their registration. The argument **`arg`** is passed to the function that is being called.

int BGAtRestart((void *) function (void *arg), void *arg)

BGAtRestart() registers the functions to be called during restart after the program state has been restored, but before jumping to the appropriate position in the application code. The functions that are registered are called in the reverse order of their registration. You can use this function to resume or re-initialize functions or data structures at the time of restart, for example, in the symmetrical multiprocessing node mode (SMP Node mode), the SMP must be re-initialized at the time of restart. The argument **`arg`** is passed to the function that is being called.

int BGAtContinue((void *) function (void *arg), void *arg)

BGAtContinue() registers the functions to be called when continuing after a checkpoint. You can use this function to re-initialize or resume some functions or data structures that were closed or stopped at the time of checkpoint. The functions that are registered are called in the reverse order of their registration. The argument **`arg`** is passed to the function that is being called.

10.4 Directory and file-naming conventions

By default, all the checkpoint files are stored, or retrieved during restart, in the directory specified by `ckptDirPath` in the initial call to BGCheckpointInit(). If `ckptDirPath` is not specified (or is NULL), the directory is picked from the environment variable `BG_CHKPTDIRPATH`. This environment variable can be set by the job control system at the time of job launch to specify the default location of the checkpoint files. If this variable is not set, the Blue Gene/P system looks for a $(HOME)/checkpoint directory. Finally, if this directory is also not available, $(HOME) is used to store all checkpoint files.

The checkpoint files are automatically created and named with the following convention:

```
<ckptDirPath>/ckpt.<xxx-yyy-zzz>.<seqNo>
```

Note the following explanation:

<ckptDirPath>	Name of the executable, for example, sweep3d or mg.W.2
<xxx-yyy-zzz>	Three-dimensional torus coordinates of the process
<seqNo>	The checkpoint sequence number

The checkpoint sequence number starts at one and is incremented after every successful checkpoint.

10.5 Restart

A transparent restart mechanism is provided through the use of the BGCheckpointInit() function and the `BG_CHKPTRESTARTSEQNO` environment variable. Upon startup, an application is expected to make a call to BGCheckpointInit(). The BGCheckpointInit() function initializes the checkpoint library data structures.

Moreover the BGCheckpointInit() function checks for the environment variable `BG_CHKPTRESTARTSEQNO`. If the variable is not set, a job launch is assumed, and the function returns normally. In case the environment variable is set to zero, the individual processes restart from their individual latest consistent global checkpoint. If the variable is set to a positive integer, the application is started from the specified checkpoint sequence number.

10.5.1 Determining the latest consistent global checkpoint

Existence of a checkpoint file does not guarantee consistency of the checkpoint. An application might have crashed before completely writing the program state to the file. We have avoided this by adding a checkpoint write complete flag in the header of the checkpoint file. As soon as the checkpoint file is opened for writing, this flag is set to zero and written to the checkpoint file. When complete checkpoint data is written to the file, the flag is set to one indicating the consistency of the checkpoint data. The job launch subsystem can use this flag to verify the consistency of checkpoint files and delete inconsistent checkpoint files.

During a checkpoint, some of the processes can crash, while others might complete. This can create consistent checkpoint files for some processes and inconsistent or non-existent checkpoint files for other processes. The latest consistent global checkpoint is determined by the latest checkpoint for which all the processes have consistent checkpoint files.

It is the responsibility of the job launch subsystem to make sure that BG_CHKPTRESTARTSEQNO corresponds to a consistent global checkpoint. In case BG_CHKPTRESTARTSEQNO is set to zero, the job launch subsystem must make sure that files with the highest checkpoint sequence number correspond to a consistent global checkpoint. The behavior of the checkpoint library is undefined if BG_CHKPTRESTARTSEQNO does not correspond to a global consistent checkpoint.

10.5.2 Checkpoint and restart functionality

It is often desirable to enable or disable the checkpoint functionality at the time of job launch. Application developers are not required to provide two versions of their programs: one with checkpoint enabled and another with checkpoint disabled. We have used environment variables to transparently enable and disable the checkpoint and restart functionality.

The checkpoint library calls check for the environment variable BG_CHKPTENABLED. The checkpoint functionality is invoked only if this environment variable is set to a value of 1. Table 10-2 summarizes the checkpoint-related function calls.

Table 10-2 Checkpoint and restart APIs

Function name	Usage
`BGCheckpointInit(char *ckptDirPath);`	Sets the checkpoint directory to `ckptDirPath`. Initializes the checkpoint library data structures. Carries out restart if environment variable BG_CHKPTRESTARTSEQNO is set.
`BGCheckpoint();`	Takes a checkpoint. Stores the program state in the checkpoint directory.
`BGCheckpointRestart(int rstartSqNo);`	Carries out an explicit restart from the specified sequence number.
`BGCheckpointExcludeRegion (void *addr, size_t len);`	Excludes the specified region from the checkpoint state.

Table 10-3 summarizes the environment variables.

Table 10-3 Checkpoint and restart environment variables

Environment variables	Usage
BG_CHKPTENABLED	Is set (to 1) if checkpoints are desired; otherwise, it is not specified.
BG_CHKPTDIRPATH	Default path to keep checkpoint files.
BG_CHKPTRESTARTSEQNO	Set to a desired checkpoint sequence number from where a user wants the application to restart. If set to zero, each process restarts from its individual latest consistent checkpoint. This option must not be specified, if no restart is desired.

The following environment variable settings are the most common:

- `BG_CHKPTENABLED=1`
- `BG_CHKPTDIRPATH= checkpoint directory`
- `BG_CHKPTRESTARTSEQNO=0`

A combination of BG_CHKPTENABLED and BG_CHKPTRESTARTSEQNO (as in Table 10-3) automatically signifies that after restart, further checkpoints are taken. A developer can restart an application but disable further checkpoints by simply unsetting (removing altogether) the BG_CHKPTENABLED variable.

11

mpirun

`mpirun` is a software utility for launching, monitoring, and controlling programs (applications) that run on the BlueGene/ P system. `mpirun` on the Blue Gene/P system serves the same function as on the Blue Gene/L system.

The name `mpirun` comes from Message Passing Interface (MPI) because its primary use is to launch parallel jobs. `mpirun` can be used as a standalone program by providing parameters either directly through a command line or from environmental variable arguments, or indirectly through the framework of a scheduler that submits the job on the user's behalf. In the former case, `mpirun` can be invoked as a shell command. It allows you to interact with the running job through the job's standard input, standard output, and standard error. The `mpirun` software utility acts as a shadow of the actual IBM Blue Gene/P job by monitoring its status and providing access to standard input, output, and errors. After the job terminates, `mpirun` terminates as well. If the user wants to prematurely end the job before it terminates, `mpirun` provides a mechanism to do so explicitly or through a timeout period.

The `mpirun` software utility provides the capability to debug the job. In this chapter, we describe the *standalone interactive* use of `mpirun`. We also provide a brief overview of `mpirun` on the Blue Gene/P system. In addition, we define a list of APIs that allow interaction with the `mpirun` program. These APIs are used by applications, such as external resource managers, that want to programmatically invoke jobs using `mpirun`.

We address the following topics in this chapter and provide examples:

- mpirun implementation on IBM Blue Gene/P
- mpirun setup
- Invoking mpirun
- Environment variables
- Tool-launching interface
- Return codes
- mpirun APIs

11.1 mpirun implementation on Blue Gene/P

The **mpirun** software utility accepts a rich set of parameters, following the philosophy of the Blue Gene/L system, that describe its behavior prior to submitting the application for execution on the Compute Nodes and during execution of the application. These parameters can be divided into three groups. The first group identifies resources that are required to run the application. The second group identifies the application (binary) to execute and the environment settings for that particular run or executable. The third group identifies the level of verbosity that **mpirun** prints to STDOUT or STDERR.

Although **mpirun** kept all of the functionality that is available on the Blue Gene/L system, its implementation on the Blue Gene/P system differs in the following ways:

- The rsh/ssh mechanism was eliminated for starting the back end process due to security concerns of allowing users access to the Service Node. In the Blue Gene/P system, this is replaced with a daemon process that runs on the Service Node whose purpose is to handle connections from front-end **mpirun** processes and fork back-end **mpirun** processes, as illustrated in Figure 11-1.

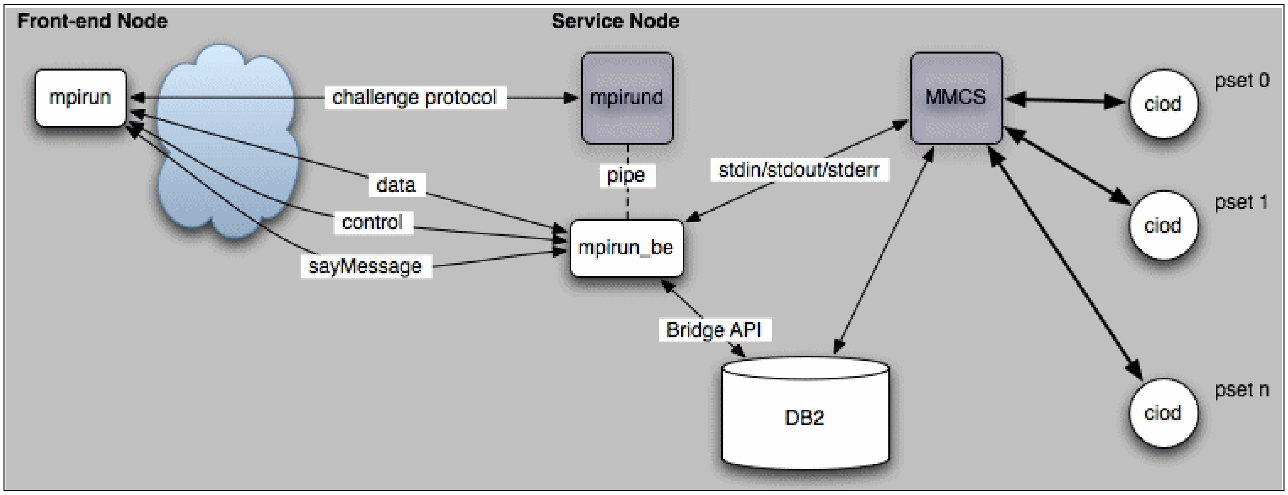


Figure 11-1 mpirun interacting with the rest of the control system on the Blue Gene/P system

- After `mpirun_be` is forked, the sequence of events for booting partitions, starting jobs, and collecting stdout/stderr is similar to the use of **mpirun** on the Blue Gene/L system.
- The **freepartition** program was integrated as an option in **mpirun** for the Blue Gene/P system. Example 11-1 shows how the free option is now used as part of **mpirun** on the Blue Gene/P system.

Example 11-1 mpirun example with -free option

```
$ mpirun -partition N01_32_1 -free wait -verbose 1
<Jul 06 15:10:48.401421> FE_MPI (Info) : Invoking free partition
<Jul 06 15:10:48.414677> FE_MPI (Info) : freePartition() - connected to mpirun server at spinoza
<Jul 06 15:10:48.414768> FE_MPI (Info) : freePartition() - sent free partition request
<Jul 06 15:11:19.202335> FE_MPI (Info) : freePartition() - partition N01_32_1 was freed successfully
<Jul 06 15:11:19.202746> FE_MPI (Info) : ==   FE completed   ==
<Jul 06 15:11:19.202790> FE_MPI (Info) : == Exit status:   0 ==
```

- Also new in **mpirun** for the Blue Gene/P system is the support for multiple program, multiple data (MPMD)[33] style jobs where a different executable, arguments, environment, and current working directory can be supplied for a single job on a processor set (pset) basis, for example, with this capability, a user can run four different executables on a partition with four psets.

This capability is handled by a new tool called **mpiexec**, which is not to be confused with the **mpiexec** style of submitting a Single Program Multiple Data (SPMD) parallel MPI job.

11.1.1 mpiexec

mpiexec is the method for launching and interacting with parallel Multiple Program Multiple Data (MPMD) jobs on Blue Gene/P. It is very similar to **mpirun** with the only exception being that the arguments supported by **mpiexec** are slightly different.

Unsupported parameters

The parameters listed in Table 11-1 are supported by **mipirun** but not supported by **mpiexec** because they do not apply to MPMD.

Table 11-1 Unsupported parameters

Parameter	Environment variables
-exe	MPIRUN_CWD MPIRUN_WDIR
-env	MPIRUN_ENV
-exp_env	MPIRUN_EXP_ENV
-env_all	MPIRUN_EXP_ENV_ALL
-mapfile	MPIRUN_ARGS
-args	MPIRUN_ARGS

New parameters

The only parameter that **mpiexec** supports that is not supported by **mpirun** is the -configfile argument. See “mipexec example” on page 180 for sample usage.

```
-configfile MPIRUN_MPMD_CONFIGFILE
```

The MPMD configuration file must end with a newline character.

Limitations

Due to some underlying designs in the Blue Gene/P software stack, when using MPMD, the following limitations are applicable:

- A pset is the smallest granularity for each executable, though one executable can span multiple psets.
- You must use every compute node of each pset; specifically different -np values are not supported.
- The job mode (SMP, DUAL, or VNM) must be uniform across all psets.

mipexec example

Example 11-2 illustrates running /bin/hostname on a single 32-node pset, helloworld.sh on another 32-node pset and goodbyeworld.sh on two 32-node psets. The partition bar consists of 128 nodes, with 4 I/O nodes.

Example 11-2 mpiexec example

```
$ mpiexec -partition bar : -n 32 -wdir /bgusr/hello /bin/hostname : -n 32 -wdir
/bgusr/goodbye /bglhome/helloworld.sh : -n 64 -wdir /bgusr/samjmill/temp
/bglhome/goodbyeworld.sh
```

11.1.2 mpikill

The **mpikill** command sends a signal to an MPI job running on the compute nodes. A signal can cause a job to terminate or an application might catch and handle signals to affect its behavior.

The format of the **mpikill** command is:

```
mpikill [options] <pid> | --job <jobId>
```

The job to receive the signal can be specified by either the PID of the **mpirun** process or the job database ID. The PID can be used only if the **mpirun** process is on the same system that **mpikill** is run on. By default, the signal sent to the job is KILL. Only the user that the job is running as can signal a job using **mpikill**. Table 11-2 lists the options that can be used with the **mpikill** command.

Table 11-2 mpikill command options

Option	Description
-s <*signal*> or -*SIGNAL*	The signal to send to the job. The signal can be a signal name, such as TERM, or a signal number, such as 15. The default signal is KILL.
-h or --help	Displays help text.
--hostname <*hostname*>	Specifies the Service Node to use. The default is the value of the MMCS_SERVER_IP environment variable, if that environment variable is set, or 127.0.0.1.
--port <*port*>	Specifies the listening port of **mpirund**. The default is 9874.
--trace <0-7>	Tracing level. The default is 0.
--config <*filename*>	mpirun configuration file, which contains the shared secret needed for **mpikill** to authenticate with the mpirun daemon on the service node. If not specified, the mpirun configuration file is located by looking for these files in order: /etc/mpirun.cfg or <release-dir>/bin/mpirun.cfg (where <release-dir> is the Blue Gene/P system software directory, for example, /bgsys/drivers/V1R2M0_200_2008-080513P/ppc).

The **mpikill** command was introduced in Blue Gene/P release V1R3M0.

Example 11-3 illustrates signaling a job running on the same front end node by providing the PID of the mpirun process.

Example 11-3 Use mpikill to signal a job using the PID of mpirun

Start a mpirun job in shell 1:

```
1$ mpirun -partition MYPARTITION sleeper.bg
```

In shell 2, use ps to get the PID of the mpirun process and use **mpikill** to signal the job with SIGINT. In this case, the PID was 21630:

```
2$ mpikill -INT 21630
```

The job receives the signal and exits, causing the following output from **mpirun** in shell 1:

```
<Jul 06 15:12:10.792041> BE_MPI (ERROR): The error message in the job record is as
follows:
<Jul 06 15:12:10.792136> BE_MPI (ERROR):   "killed with signal 2"
```

Example 11-4 illustrates signaling a job running on a different front end node by providing the job database ID.

Example 11-4 Use mpikill to signal a job using the job ID

Start a mpirun job on FEN 1, using the verbose output to display the job ID. In this case the job ID is 21203:

```
FEN1$ mpirun -partition MYPARTITION -verbose 1 sleeper.bg
... -- verbose output
<Jul 06 15:18:10.547452> FE_MPI (Info) : Job added with the following id: 21203
... -- verbose output
```

On FEN 2, use mpikill to signal the job with SIGINT:

```
FEN2$ mpikill -INT --job 21203
```

The job receives the signal and exits, causing the following output from mpirun in shell 1:

```
... -- verbose output
<Jul 06 15:19:06.745821> BE_MPI (ERROR): The error message in the job record is as
follows:
<Jul 06 15:19:06.745856> BE_MPI (ERROR):   "killed with signal 2"
<Jul 06 15:19:07.106672> FE_MPI (Info) : ==   FE completed   ==
<Jul 06 15:19:07.106731> FE_MPI (Info) : == Exit status: 130 ==
```

11.2 mpirun setup

`mpirun` does not require set up from a user point-of-view. However, on the Service Node, `mpirun` requires slightly more set up for a system administrator. We classified the setup of `mpirun` as the following types:

- User setup
- System administrator setup

11.2.1 User setup

Some set up changed for Blue Gene/P as compared to **mpirun** for the Blue Gene/L system. The following changes are among those for user setup:

- It is not required to set up .rhosts or ssh-agent.
- It is not required to set up .bashrc, .tcshrc, or .profile to include BRIDGE_CONFIG_FILE or DB_PROPERTY environment variables.
- The **freepartition** program is now an option in **mpirun**.
- The -backend option is no longer available.

Due to the removal of the ssh/rsh mechanism to start a back end **mpirun** process, users no longer are required to create an .rhosts file in their home directory for **mpirun** to work properly.

11.2.2 System administrator set up

System administrators can change the following configuration files for the **mpirun** daemon (**mpirund**):

db.properties Contains information about the IBM DB2 database

bridge.config Contains locations of the default I/O Node and Compute Node images when allocating partitions

mpirun.cfg Contains the shared secret that is used for challenge authentication between **mpirun** and **mpirund**

Database properties and Bridge configuration files

The location of the database properties and bridge configuration files can be changed by passing the appropriate arguments to bgpmaster when starting **mpirund**. The **mpirun** daemon then passes these locations to each mpirun_be it forks. Example 11-5 shows a sample Bridge configuration file.

Example 11-5 Sample Bridge configuration file

```
BGP_MACHINE_SN     BGP
BGP_MLOADER_IMAGE  /bgsys/drivers/ppcfloor/boot/uloader
BGP_CNLOAD_IMAGE
/bgsys/drivers/ppcfloor/boot/cns,/bgsys/drivers/ppcfloor/boot/cnk
BGP_IOLOAD_IMAGE
/bgsys/drivers/ppcfloor/boot/cns,/bgsys/drivers/ppcfloor/boot/linux,/bgsys/drivers
/ppcfloor/boot/ramdisk
BGP_LINUX_MLOADER_IMAGE  /bgsys/drivers/ppcfloor/boot/uloader
BGP_LINUX_CNLOAD_IMAGE
/bgsys/drivers/ppcfloor/boot/cns,/bgsys/drivers/ppcfloor/boot/linux,/bgsys/drivers
/ppcfloor/boot/ramdisk
BGP_LINUX_IOLOAD_IMAGE
/bgsys/drivers/ppcfloor/boot/cns,/bgsys/drivers/ppcfloor/boot/linux,/bgsys/drivers
/ppcfloor/boot/ramdisk
BGP_BOOT_OPTIONS
BGP_DEFAULT_CWD    $PWD
BGP_ENFORCE_DRAIN
```

BGP_DEFAULT_CWD is used for **mpirun** jobs when a user does not give the -cwd argument or one of its environment variables. You can change this value to something more site specific, such as /bgp/users, /gpfs/, and so on. The special keyword $PWD is expanded to the user's current working directory from where the user executed **mpirun**.

Challenge protocol

The challenge protocol, which is used to authenticate the mpirun front end when connecting to the mpirun daemon on the Service Node, is a challenge/response protocol. It uses a shared secret to create a hash of a random number, thereby verifying that the mpirun front end has access to the secret.

To protect the secret, the challenge protocol is stored in a configuration file that is accessible only to the bgpadmin user on the Service Node and to a special mpirun user on the front end nodes. The front end mpirun binary has its setuid flag enabled so that it can change its uid to match the **mpirun** user and read the configuration file to access the secret.

11.3 Invoking mpirun

The first method of using **mpirun** is to specify the parameters explicitly, as shown in the following example:

```
mpirun [options]
```

Here is a practical example of using **mpirun**:

```
mpirun -partition R00-M0 -mode VN -cwd /bgusr/tmp -exe a.out --args "--timeout 50"
```

Alternatively, you can use the mpiexec style where the executable and arguments are implicit, as shown in the following example (see 11.1.1, "mpiexec" on page 179):

```
mpirun [options] binary [arg1 arg2 ... argn]
```

Here is a practical example of using the mpiexec style of executable and arguments:

```
mpirun -partition R00-M0 -mode VN -cwd /bgusr/tmp a.out --timeout 50
```

Standard input and output

Output generated to standard output or standard error by the MPI processes on the Blue Gene/P compute nodes, such as through printf, is transparently redirected by the **mpirun** process on the front end node. When different ranks print output, the output of all ranks is aggregated. **mpirun** can also be instructed to prepend each line of output with the source MPI rank using the -label option, as described in Table 11-5 on page 186.

Any input to the program from standard input goes only to the MPI process at rank 0. Programs should only request input from rank 0 because requesting input from other ranks fails.

Specifying parameters

You can specify most parameters for the **mpirun** program in the following different ways:

- Command-line arguments
- Environment variables
- Scheduler interface plug-in

In general, users normally use the command-line arguments and the environment variables. Certain schedulers use the scheduler interface plug-in to restrict or enable **mpirun** features according to their environment, for example, the scheduler might have a policy where interactive job submission with **mpirun** can be allowed only during certain hours of the day.

Command-line arguments

The **mpirun** arguments consist of the following categories:

- Job control
- Block control
- Output
- Other

Job control arguments

Table 11-3 lists the job control arguments to miprun.

Table 11-3 Job control arguments

Arguments	Description
`-args "`*`program args`*`"`	Passes "program args" to the BlueGene job on the Compute Nodes.
`-env "`*`ENVVAR=value`*`"`	Sets an environment variable in the environment of the job on the Compute Nodes.
`-exp_env <`*`ENVVAR`*`>`	Exports an environment variable in the current environment of **mpirun** to the job on the Compute Nodes.
`-env_all`	Exports all environment variables in the current environment of **mpirun** to the job on the Compute Nodes.
`-np <`*`n`*`>`	Creates exactly *n* MPI ranks for the job. Aliases are -nodes and -n.
`-mode <SMP or DUAL or VN>`	Specifies the mode in which the job will run. Choices are SMP (1 rank, 4 threads), DUAL (2 ranks, 2 threads each), or Virtual Node Mode (4 ranks, 1 thread each).
`-exe <`*`executable`*`>`	Specifies the full path to the executable to run on the Compute Nodes. The path is specified as seen by the I/O and Compute Nodes.
`-cwd <`*`path`*`>`	Specifies the full path to use as the current working directory on the Compute Nodes. The path is specified as seen by the I/O and Compute Nodes.
`-mapfile <`*`mapfile`*`>`	Specifies an alternative MPI topology. The mapfile path must be fully qualified as seen by the I/O and Compute Nodes.[a]
`-timeout <`*`n`*`>`	Timeout after *n* seconds. **mpirun** monitors the job and terminates it if the job runs longer than the time specified. The default is never to timeout.

[a] For additional information about mapping, see Appendix F, "Mapping" on page 355.

Block control options

mpirun can also allocate partitions and create new partitions if necessary. Use the following general rules for block control:

- If **mpirun** is told to use a pre-existing partition and it is already booted, **mpirun** uses it as is without trying to boot it again.
- If **mpirun** creates a partition or is told to use a pre-existing partition that is not already allocated, **mpirun** allocates the partition.
- If **mpirun** allocates a partition, it deallocates the partition when it is done.

Table 11-4 summarizes the options that modify this behavior.

Table 11-4 Block control options

Arguments	Description
`-partition <block>`	Specifies a predefined block to use.
`-nofree`	If **mpirun** booted the block, it does not deallocate the block when the job is done. This is useful for when you want to run a string of jobs back-to-back on a block but do not want **mpirun** to boot and deallocate the block each time (which happens if you had not booted the block first using the console.) When your string of jobs is finally done, use the **freepartition** command to deallocate the block.
`-free <wait|nowait>`	Frees the partition specified with -partition. No job is run. The wait parameter does not return control until the partition has changed state to free. The nowait parameter returns control immediately after submitting the free partition request.
`-noallocate`	This option is more interesting for job schedulers. It tells **mpirun** not to use a block that is not already booted.
`-shape <XxYxZ>`	Specifies a hardware configuration to use. The dimensions are in the Compute Nodes. If hardware matching is found, a new partition is created and booted. Implies that -partition is not specified.
`-psets_per_bp <n>`	Specifies the I/O Node to Compute Node ratio. The default is to use the best possible ratio of I/O Nodes to Compute Nodes. Specifying a higher number of I/O Nodes than what is available results in an error.
`-connect <MESH|TORUS>`	Specifies a mesh or a torus when **mpirun** creates new partitions.
`-reboot`	Reboots all the Compute Nodes of an already booted partition that is specified with -partition before running the job. If the partition is in any other state, this is an error.
`-boot_options <options>`	Specifies boot options to use when booting a freshly created partition.

Output options

The output options in Table 11-5 on page 186 control information that is sent to STDIN, STDOUT, and STDERR.

Table 11-5 Output options

Arguments	Description
`-verbose [0-4]`	Sets the verbosity level. The default is 0, which means that **mpirun** does not output any status or diagnostic messages unless a severe error occurs. If you are curious about what is happening, try levels 1 or 2. All **mpirun** generated status and error messages appear on STDERR.
`-label`	Use this option to have **mpirun** label the source of each line of output. The source is the MPI rank, and STDERR or STDOUT from which the output originated.
`-enable_tty_reporting`	By default, **mpirun** tells the control system and the C run time on the Compute Nodes that STDIN, STDOUT, and STDERR are tied to TTY type devices. While semantically correct for the Blue Gene system, this prevents blocked I/O to these file descriptors, which can slow down operations. If you use this option, **mpirun** senses whether these file descriptors are tied to TTYs and reports the results accurately to the control system.
`-strace <all|none|`*n*`>`	Use this argument to enable a syscall trace on all Compute Nodes, no Compute Nodes, or a specific Compute Node (identified by MPI rank). The extra output from the syscall trace appears on STDERR. The default is none.

Other options

Table 11-6 lists other options. These options provide general information about selected software and hardware features.

Table 11-6 Other options

Arguments	Description
`-h`	Displays help text.
`-version`	Displays **mpirun** version information.
`-host` <*host_name*>	Specifies the Service Node to use.
`-port` <*port*>	Specifies the listening port of **mpirund**.
`-start_gdbserver` <*path_to_gdbserver*>	Loads the job in such a way as to enable GDB debugging, either right from the first instruction or later on while the job is running. Either this option or -start_tool can be used, but not both.
`-start_tool` <*path*>	Specifies the tool to launch on the I/O nodes with the job. Either this option or -start_gdbserver can be used, but not both.
`-tool_args` "*program args*"	Specifies the arguments to the tool started on the I/O nodes with the job.
`-config` <*path*>	mpirun configuration file, which contains the shared secret needed for **mpirun** to authenticate with the mpirun daemon on the service node. If not specified, the mpirun configuration file is located by looking for these files in order: /etc/mpirun.cfg or <release-dir>/bin/mpirun.cfg (where <release-dir> is the Blue Gene/P system software directory, for example, /bgsys/drivers/V1R2M0_200_2008-080513P/ppc).

Arguments	Description
-nw	Reports **mpirun**-generated return code instead of an application-generated return code. Useful only for debugging **mpirun**.
-only_test_protocol	Simulates a job without using any hardware or talking to the control system. It is useful for making sure that **mpirun** can start **mpirun_be** correctly.

11.4 Environment variables

An alternative way to control **mpirun** execution is to use environment variables. Most command-line options for **mpirun** can be specified using an environment variable. The variables are useful for options that are used in production runs. If you do need to alter the option, you can modify it on the command line to override the environment variable. Table 11-7 summarizes all the environmental variables. The variables must be defined before execution of **mpirun** starts.

Table 11-7 Environmental variables

Arguments	Environment variables
-partition	MPIRUN_PARTITION
-nodes	MPIRUN_NODES MPIRUN_N MPIRUN_NP
-mode	MPIRUN_MODE
-exe	MPIRUN_EXE
-cwd	MPIRUN_CWD MPIRUN_WDIR
-host	MMCS_SERVER_IP MPIRUN_SERVER_HOSTNAME
-port	MPIRUN_SERVER_PORT
-env	MPIRUN_ENV
-exp_env	MPIRUN_EXP_ENV
-env_all	MPIRUN_EXP_ENV_ALL
-mapfile	MPIRUN_MAPFILE
-args	MPIRUN_ARGS
-timeout	MPIRUN_TIMEOUT
-start_gdbserver	MPIRUN_START_GDBSERVER
-label	MPIRUN_LABEL
-nw	MPIRUN_NW
-nofree	MPIRUN_NOFREE
-noallocate	MPIRUN_NOALLOCATE
-reboot	MPIRUN_REBOOT
-boot_options	MPIRUN_BOOT_OPTIONS MPIRUN_KERNEL_OPTIONS

Arguments	Environment variables
-verbose	MPIRUN_VERBOSE
-only_test_protocol	MPIRUN_ONLY_TEST_PROTOCOL
-shape	MPIRUN_SHAPE
-psets_per_bp	MPIRUN_PSETS_PER_BP
-connect	MPIRUN_CONNECTION
-enable_tty_reporting	MPIRUN_ENABLE_TTY_REPORTING
-config	MPIRUN_CONFIG_FILE

11.5 Tool-launching interface

A tool-launching interface is available through **mpirun** that enables users to start an additional program that runs on the I/O nodes in addition to the job running on the compute nodes. This second program, for example, might provide an alternative debugging interface.

The tool is started when the user specifies the -start_tool option when invoking **mpirun**. Command-line parameters can be specified for the tool using the -tool_args option. The tool runs with the identity of the user running the job, and the initial current working directory is the directory specified for the job.

The following example shows the use of the -start_tool and -tool_args **mpirun** options:

```
mpirun -partition <xx> -exe <path> -start_tool <path> -tool_args <args>
```

Environment variables available to the tool include those from the CIOD environment and the environment variables specified for the job. The tool can communicate with CIOD using the CioDebugger protocol.

When the job running on the compute nodes ends, the tool running on the I/O nodes is sent a SIGTERM signal. If the tool fails to end before the timeout interval, the tool is forcibly terminated. The default timeout interval is 10 seconds.

The tool-launching interface was added in Blue Gene/P release V1R3M0.

11.6 Return codes

If **mpirun** fails for any reason, such as a bug, boot failure, or job failure, it returns a return code to your shell if you supply the -nw argument. If you omit the -nw argument, it returns the job's return code if it is present in the job table. Table 11-8 lists the possible error codes.

Table 11-8 Return codes

Return code	Description
0	OK; successful
10	Communication error
11	Version handshake failed
12	Front-end initialization failed

Return code	Description
13	Failed to execute back-end **mpirun** on Service Node
14	Back-end initialization failed
15	Failed to locate db.properties file
16	Failed to get the machine serial number (bridge configuration file not found?)
17	Execution interrupted by message from the front end
18	Failed to prepare the partition
19	Failed to initialize allocator
20	Partition name already exists
21	No free space left to allocate partition for this job
22	Failed to allocate partition
23	Failed to allocate a partition; job has illegal requirements
24	Specified partition does not exist
25	Failed to get a partition state
26	Specified partition is in an incompatible state
27	Specified partition is not ready
28	Failed to get a partition owner
29	Failed to set a partition owner
30	Failed while checking to see if the partition is busy
31	Partition is occupied by another job
32	Failed while checking to see if the user is in the partition’s user list
33	A user does not have permission to run the job on the specified partition
34	Failed while examining the specified partition
35	Failed while setting kernel options; the rm_modify_partition() API failed
36	Kernel options were specified but the partition is not in a FREE state
37	Failed to boot the partition
38	Failed to reboot the partition
39	Failed to get the number of psets in the partition
40	Failed to create MPMD configuration file on the Service Node
41	Found a zero-length line while writing to the MPMD configuration file
42	Failed to write a line to the MPMD configuration file
43	Failed to validate the MPMD configuration file
44	Failed to add the new job to the database
45	Failed to get an ID for the new job
46	Failed to start the job

Return code	Description
13	Failed to execute back-end **mpirun** on Service Node
14	Back-end initialization failed
15	Failed to locate db.properties file
16	Failed to get the machine serial number (bridge configuration file not found?)
17	Execution interrupted by message from the front end
18	Failed to prepare the partition
19	Failed to initialize allocator
20	Partition name already exists
21	No free space left to allocate partition for this job
22	Failed to allocate partition
23	Failed to allocate a partition; job has illegal requirements
24	Specified partition does not exist
25	Failed to get a partition state
26	Specified partition is in an incompatible state
27	Specified partition is not ready
28	Failed to get a partition owner
29	Failed to set a partition owner
30	Failed while checking to see if the partition is busy
31	Partition is occupied by another job
32	Failed while checking to see if the user is in the partition’s user list
33	A user does not have permission to run the job on the specified partition
34	Failed while examining the specified partition
35	Failed while setting kernel options; the rm_modify_partition() API failed
36	Kernel options were specified but the partition is not in a FREE state
37	Failed to boot the partition
38	Failed to reboot the partition
39	Failed to get the number of psets in the partition
40	Failed to create MPMD configuration file on the Service Node
41	Found a zero-length line while writing to the MPMD configuration file
42	Failed to write a line to the MPMD configuration file
43	Failed to validate the MPMD configuration file
44	Failed to add the new job to the database
45	Failed to get an ID for the new job
46	Failed to start the job

Return code	Description
47	An error occurred while **mpirun** was waiting for the job to terminate
48	Job timed out
49	The job was moved to the history table before it terminated
50	Job execution failed; job switched to an error state
51	Job execution interrupted; job queued
52	Failed to get a job exit status
53	Failed to get a job error text
54	Executable path for the debugger server is not specified
55	Failed to set debug information; unable to attach the debugger
56	Failed to get proctable; unable to attach the debugger
57	Failed while attaching to the job; unable to attach the debugger
58	Failed debugging job; unable to attach the debugger
59	Failed to begin a job
60	Failed to load a job
61	Failed to wait for job to load
62	Failed to clean up a job, partition, or both
63	Failed to cancel a job
64	Failed to destroy a partition
65	Failed to remove a partition
66	Failed to reset boot options; the rm_modify_partition() API failed
67	One or more threads died
68	Unexpected message
69	Failed to dequeue control message
70	Out of memory
71	Execution interrupted by signal

11.7 Examples

In this section, we present various examples of **mpirun** commands.

Display information

Example 11-6 shows how to display information using the -h flag.

Example 11-6 Invoking mpirun -h or -help to list all the options available

```
$ mpirun -h
Usage:
        mpirun [options]
```

```
         or
         mpirun [options] binary [arg1 arg2 ... argn]

Options:
    -h                          Provides this extended help information; can also use -help
    -version                    Display version information
    -partition <partition_id>   ID of the partition to run the job on
    -np <compute_nodes>         The number of Compute Nodes to use for the job
    -mode <SMP|DUAL|VN>         Execution mode, either SMP, DUAL, or Virtual Node Mode; the
                                default is SMP
    -exe <binary>               Full path to the binary to execute
    -cwd <path>                 Current working directory of the job, as seen by the
                                Compute Nodes; can also use -wdir
    -host <service_node_host>   Host name of the Service Node
    -port <service_node_port>   Port of the mpirun server on the Service Node
    -env <env=val>              Environment variable that should be set
    -exp_env <env vars>         Environment variable in the current environment to export
    -env_all                    Export all current enviorment variables to the job environment
    -mapfile <mapfile|mapping>  mapfile contains a user specified MPI topology;
                                mapping is a permutation of XYZT
    -args <"<arguments>">       Arguments to pass to the job; must be enclosed in double
                                quotation marks
    -timeout <seconds>          The limit of the job execution time
    -start_gdbserver <path>     Start gdbserver for the job; must specify the path to gdbserver
    -label                      Add labels (STDOUT, STDERR, and MPI rank) to the job output
    -nw                         Return mpirun job cycle status instead of the job exit status
    -nofree                     Do not deallocate the partition if mpirun allocated it
    -free <wait|nowait>         Free the partition specified by -partition; no job will be run
    -noallocate                 Do not allocate the partition; the job will only start
                                if the partition was already INITIALIZED or CONFIGURING
    -reboot                     Reboot all Compute Nodes of the specified partition before
                                running the job; the partition must be INIIALIZED prior
                                to rebooting
    -backend                    Use a specified mpirun backend binary on the Service Node
    -boot_options <options>     Low-level options used when booting a partition
    -verbose <0|1|2|3|4>        Verbosity level, default is 0
    -trace <0-7>                Trace level; output is sent to a file in the current working
                                directory; default level is 0
    -only_test_protocol         Test the mpirun frontend to backend communication;
                                no job will be run
    -strace <all|none|n>        Enable syscall trace for all, none, or node with MPI rank n
    -shape <XxYxZ>              Shape of job in XxYxZ format; if not specified, you must use
                                -partition or -np
    -psets_per_bp <n>           Number of psets per base partition required in the partition
    -connect <TORUS|MESH>       Compute Node connections; default is MESH
    -enable_tty_reporting       Correctly report tty status to the control system
    -config <path>              Specify mpirun config file path
```

Creating a partition dynamically

In Example 11-7 on page 193, a user requests a number (-np) of Compute Nodes desired for the job. The allocator API searches the machine for free resources and boots the temporary partition if enough resources are found. Upon job completion, **mpirun** deallocates the partition if the user has not specified -nofree.

Example 11-7 Dynamic allocation

```
$ mpirun -np 16 -exe /bin/hostname -verbose 1
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
```

Using -psets_per_bp

Example 11-8 illustrates the usage of `-psets_per_bp`. The number of psets per base partition is defined in the db.properties file. The value can be overridden with the `-psets_per_bp` option.

Example 11-8 psets_per_bp

```
$ mpirun -psets_per_bp 16 -shape 4x4x2 -exe /bin/hostname
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
```

```
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
dd2sys1fen3
```

Using a predefined partition and -np

Example 11-9 shows a simple script to invoke **mpirun**.

Example 11-9 csh script to invoke mpirun

```
$ ./run.pallas >& pallas_july06_2007_bgp.out
where the script run.pallas is:
#!/bin/csh
set MPIRUN="mpirun"
set MPIOPT="-np 32"
set MODE="-mode VN"
set PARTITION="-partition N01_32_1"
set WDIR="-cwd /bgusr/cpsosa/pallas"
set EXE="-exe /bgusr/cpsosa/pallas/PMB-MPI1"
#
$MPIRUN $PARTITION $MPIOPT $MODE $WDIR $EXE
#
echo "That's all folks!!"
```

Using environment variables

Example 11-10 shows use of -env to define environment variables.

Example 11-10 Use of -env

```
$ mpirun -partition N00_32_1 -np 32 -mode SMP -cwd /bgusr/cpsosa -exe a.out -env
“OMP_NUM_THREADS=4”
```

Using stdin from a terminal

In Example 11-11, the user types the user’s name `bgp user` in response to the job’s stdout. After a while, the job is terminated when the user presses Ctrl+C to send **mpirun** a SIGINT.

Example 11-11 Usage of STDIN from a terminal

```
$ mpirun -partition R00-M0-N00 -verbose 0 -exe /BGPhome/stdin.sh -np 1
What's your name?
bgp user
hello bgp user
What's your name?
<Aug 11 15:33:44.021105> FE_MPI (WARN) : SignalHandler() -
<Aug 11 15:33:44.021173> FE_MPI (WARN) : SignalHandler() -
!------------------------------------------------!
<Aug 11 15:33:44.021201> FE_MPI (WARN) : SignalHandler() - ! mpirun is now taking all the
necessary actions !
<Aug 11 15:33:44.021217> FE_MPI (WARN) : SignalHandler() - ! to terminate the job and to free
the resources !
```

```
<Aug 11 15:33:44.021233> FE_MPI (WARN) : SignalHandler() - ! occupied by this job. This might
take a while... !
<Aug 11 15:33:44.021261> FE_MPI (WARN) : SignalHandler() -
!-----------------------------------------------!
<Aug 11 15:33:44.021276> FE_MPI (WARN) : SignalHandler() -
<Aug 11 15:33:44.050365> BE_MPI (WARN) : Received a message from frontend
<Aug 11 15:33:44.050465> BE_MPI (WARN) : Execution of the current command interrupted
<Aug 11 15:33:59.532817> FE_MPI (ERROR): Failure list:
<Aug 11 15:33:59.532899> FE_MPI (ERROR):   - 1. Execution interrupted by signal (failure #71)
dd2sys1fen3:~/bgp/control/mpirun/new>
```

Using stdin from a file or pipe

Example 11-12 illustrates the use of stdin from a file or pipe.

Example 11-12 Usage of stdin from a file or pipe

```
$ cat ~/stdin.cc
#include <iostream>

using namespace std;

int main() {
    unsigned int lineno = 0;
    while (cin.good()) {
        string line;
        getline(cin, line);
        if (!line.empty()) {
            cout << "line " << ++lineno << ": " << line << endl;
        }
    }
}
$ cat stdin.txt
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Quisque id orci. Ut eleifend dui a erat varius
facilisis. Aliquam felis. Ut tincidunt, velit in pulvinar imperdiet, sem sapien sagittis neque, vitae bibendum
sapien erat vitae risus. Aenean suscipit. Aliquam molestie orci nec magna. Aliquam non enim. Integer dictum
magna quis orci. Praesent eget libero sed erat ultrices ullamcorper. Donec sodales hendrerit velit. Fusce
mattis. Suspendisse blandit ornare arcu. Pellentesque venenatis.

$ cat stdin.txt | mpirun -partition R00-M0-N00 -verbose 0 -exe /BGPhome/stdin_test -np 1
line 1: Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Quisque id orci. Ut eleifend dui a erat varius
line 2: facilisis. Aliquam felis. Ut tincidunt, velit in pulvinar imperdiet, sem sapien sagittis neque, vitae bibendum
line 3: sapien erat vitae risus. Aenean suscipit. Aliquam molestie orci nec magna. Aliquam non enim. Integer dictum
line 4: magna quis orci. Praesent eget libero sed erat ultrices ullamcorper. Donec sodales hendrerit velit. Fusce
line 5: mattis. Suspendisse blandit ornare arcu. Pellentesque venenatis.
```

Using the tee utility

To send stdout, stderr, or both to a file in addition to your terminal, use the tee utility. Give tee the `-i` argument, so that it ignores any signals that are sent, such as Ctrl+C, to terminate a job prematurely. See Example 11-13.

Example 11-13 Using tee

```
$ mpirun -partition R00-M0-N00 -verbose 1 -exe /BGPhome/datespinner.sh -np 1 | tee
-i datespinner.out
<Aug 12 10:27:10.997374> FE_MPI (Info) : Invoking mpirun backend
<Aug 12 10:27:11.155416> BRIDGE (Info) : rm_set_serial() - The machine serial
number (alias) is BGP
<Aug 12 10:27:11.194557> FE_MPI (Info) : Preparing partition
<Aug 12 10:27:11.234550> BE_MPI (Info) : Examining specified partition
```

```
<Aug 12 10:27:11.823425> BE_MPI (Info) : Checking partition R00-M0-N00 initial
state ...
<Aug 12 10:27:11.823499> BE_MPI (Info) : Partition R00-M0-N00 initial state =
READY ('I')
<Aug 12 10:27:11.823516> BE_MPI (Info) : Checking partition owner...
<Aug 12 10:27:11.823532> BE_MPI (Info) : partition R00-M0-N00 owner is 'userX'
<Aug 12 10:27:11.824744> BE_MPI (Info) : Partition owner matches the current user
<Aug 12 10:27:11.824870> BE_MPI (Info) : Done preparing partition
<Aug 12 10:27:11.864539> FE_MPI (Info) : Adding job
<Aug 12 10:27:11.864876> BE_MPI (Info) : No CWD specified ('-cwd' option)
<Aug 12 10:27:11.864903> BE_MPI (Info) :   - it will be set to
'/BGPhome/usr3/bgp/control/mpirun/new'
<Aug 12 10:27:11.865046> BE_MPI (Info) : Adding job to database...
<Aug 12 10:27:11.944540> FE_MPI (Info) : Job added with the following id: 15
<Aug 12 10:27:11.944593> FE_MPI (Info) : Starting job 15
<Aug 12 10:27:12.004492> FE_MPI (Info) : Waiting for job to terminate
<Aug 12 10:27:12.816792> BE_MPI (Info) : IO - Threads initialized
Sun Aug 12 10:27:13 CDT 2007
Sun Aug 12 10:27:18 CDT 2007
Sun Aug 12 10:27:23 CDT 2007
Sun Aug 12 10:27:28 CDT 2007
Sun Aug 12 10:27:33 CDT 2007
Sun Aug 12 10:27:38 CDT 2007
Sun Aug 12 10:27:43 CDT 2007
Sun Aug 12 10:27:48 CDT 2007
Sun Aug 12 10:27:53 CDT 2007
Sun Aug 12 10:27:58 CDT 2007
Sun Aug 12 10:28:03 CDT 2007
Sun Aug 12 10:28:08 CDT 2007
<Aug 12 10:28:11.159680> FE_MPI (Info) : SignalHandler() -
<Aug 12 10:28:11.159737> FE_MPI (Info) : SignalHandler() - ! Received signal
SIGINT
<Aug 12 10:28:11.159760> FE_MPI (WARN) : SignalHandler() -
<Aug 12 10:28:11.159773> FE_MPI (WARN) : SignalHandler() -
!------------------------------------------------!
<Aug 12 10:28:11.159788> FE_MPI (WARN) : SignalHandler() - ! mpirun is now taking
all the necessary actions !
<Aug 12 10:28:11.159801> FE_MPI (WARN) : SignalHandler() - ! to terminate the job
and to free the resources !
<Aug 12 10:28:11.159815> FE_MPI (WARN) : SignalHandler() - ! occupied by this job.
This might take a while... !
<Aug 12 10:28:11.159829> FE_MPI (WARN) : SignalHandler() -
!------------------------------------------------!
<Aug 12 10:28:11.159842> FE_MPI (WARN) : SignalHandler() -
<Aug 12 10:28:11.201498> FE_MPI (Info) : Termination requested while waiting for
backend response
<Aug 12 10:28:11.201534> FE_MPI (Info) : Starting cleanup sequence
<Aug 12 10:28:11.201794> BE_MPI (WARN) : Received a message from frontend
<Aug 12 10:28:11.201863> BE_MPI (WARN) : Execution of the current command
interrupted
<Aug 12 10:28:11.201942> BE_MPI (Info) : Starting cleanup sequence
<Aug 12 10:28:11.201986> BE_MPI (Info) : cancel_job() - Cancelling job 15
<Aug 12 10:28:11.204567> BE_MPI (Info) : cancel_job() - Job 15 state is RUNNING
('R')
```

```
<Aug 12 10:28:11.230352> BE_MPI (Info) : cancel_job() - Job 15 state is DYING
('D'). Waiting...
<Aug 12 10:28:16.249665> BE_MPI (Info) : cancel_job() - Job 15 has been moved to
the history table
<Aug 12 10:28:16.255793> BE_MPI (Info) : cleanupDatabase() - Partition was
supplied with READY ('I') initial state
<Aug 12 10:28:16.255996> BE_MPI (Info) : cleanupDatabase() - No need to destroy
the partition
<Aug 12 10:28:16.591667> FE_MPI (ERROR): Failure list:
<Aug 12 10:28:16.591708> FE_MPI (ERROR):   - 1. Execution interrupted by signal
(failure #71)
<Aug 12 10:28:16.591722> FE_MPI (Info) : ==   FE completed   ==
<Aug 12 10:28:16.591736> FE_MPI (Info) : == Exit status:   1 ==
dd2sys1fen3:~/bgp/control/mpirun/new> cat datespinner.out
Sun Aug 12 10:28:49 CDT 2007
Sun Aug 12 10:28:54 CDT 2007
Sun Aug 12 10:28:59 CDT 2007
Sun Aug 12 10:29:04 CDT 2007
Sun Aug 12 10:29:09 CDT 2007
Sun Aug 12 10:29:14 CDT 2007
Sun Aug 12 10:29:19 CDT 2007
```

Error when requesting more nodes than the partition size

Example 11-14 shows a case where the user specified a value for `-np` larger than the number provided in the partition.

Example 11-14 Error due to requesting an -np value greater than the partition size

```
$ mpirun -partition R00-M0-N00 -verbose 0 -exe /bin/hostname -np 55
<Aug 11 15:28:46.797523> BE_MPI (ERROR): Job execution failed
<Aug 11 15:28:46.797634> BE_MPI (ERROR): Job 8 is in state ERROR ('E')
<Aug 11 15:28:46.842559> FE_MPI (ERROR): Job execution failed (error code - 50)
<Aug 11 15:28:46.842738> FE_MPI (ERROR):  - Job execution failed - job switched to an error
state
<Aug 11 15:28:46.851840> BE_MPI (ERROR): The error message in the job record is as follows:
<Aug 11 15:28:46.851900> BE_MPI (ERROR):   "BG_SIZE of 55 is greater than block 'R00-M0-N00'
size of 32"
```

Job encounters RAS event

When a job encounters a system error after starting, such as bad hardware or a kernel error, a RAS event might be generated that describes the problem. RAS events are logged in the RAS database, which can be viewed by the system administrator using the Blue Gene Navigator. When a job running under **mpirun** ends with a non-zero exit status, **mpirun** checks the RAS database for events related to the job and, if any RAS events were generated, prints out the number of events found and the last RAS event for the job. The information in the RAS event might enable users to correct their application without having to request help from the system administrator. This feature was added in Blue Gene/P release V1R3M0.

RAS events that do not cause the application to fail but are useful for application developer consideration can also be generated when a job runs. The information in these RAS events can be especially useful during application development to expose problems that can cause an application to run slowly. **mpirun** will display these “APPLICATION” RAS events when the -verbose command line option is set to 2 or greater and the job ends with an exit status of zero. Additionally, when -verbose 2 or greater is specified **mpirun** automatically sets the

DCMF_DMA_VERBOSE=1 environment variable for the job. Refer to this environment variable in Appendix D, “Environment variables” on page 339 for more information. This environment variable causes the job to generate RAS events should the appropriate conditions arise in the DMA controller which is then displayed by **mpirun** when the job ends. This enhancement was added in Blue Gene/P release V1R4M0.

In Example 11-15, the job failed because it used a BG_SHAREDMEMPOOLSIZE of 3000 rather than 30 as intended. The invalid value for BG_SHAREDMEMPOOLSIZE caused the kernel to generate a RAS event, which is displayed by **mpirun**.

Example 11-15 mpirun prints out RAS event information

```
$ mpirun -partition MYPARTITION -mode dual -env "BG_SHAREDMEMPOOLSIZE=3000" -exe /bin/hostname
<Aug 11 14:34:34.266226> BE_MPI (ERROR): print_job_errtext() - Job 2109409 had 2 RAS events
<Aug 11 14:34:34.266308> BE_MPI (ERROR): print_job_errtext() - last event: KERN_1D0A
Insufficient memory to start application. Shared size=-1149239296  Persistent size=0  Text
start=0x01000000  Text size=2097152  Data start=0x01200000  Data size=1048576 Static TLB slots
used for each process in this node=0 0 0 0
<Aug 11 14:34:34.266319> BE_MPI (ERROR): print_job_errtext() - Check the Navigator's job history
for complete details
```

Killing a hung job or a running job

mpirun has the capability to kill the job and free your partition if it was booted by **mpirun**. To kill your job, we recommend that you send **mpirun** a SIGINT (kill -2) while the job is running or hung. We recommend that you do *not* use SIGKILL because subsequent jobs might experience problems.

Be aware that using SIGINT is somewhat time consuming depending on the state of the job. Therefore, do not expect it to return control instantaneously. Alternatively, if you do not want to wait, try sending **mpirun** three SIGINTs in succession. In this case, it immediately returns control to your shell. However, as the warning messages indicate, your job, partition, or both might be left in a bad state. Ensure that they are cleaned up correctly before you attempt to use them again. Example 11-16 illustrates this procedure.

Example 11-16 Proper way to kill hung or running jobs

From window 2: (open another window to kill a job)

```
ps -ef | grep cpsosa

cpsosa   23393 23379  0 13:21 pts/13   00:00:00 /bgsys/drivers/ppcfloor/bin/mpirun -partition
N04_32_1 -np 32 -mode VN -cwd /bgusr/cpsosa/red/pallas -exe /bgusr/cpsosa/red/pallas/PMB-MPI1
```

From window 1: (where the job is running)

```
.
. ! Output generated by the program
.
32768         1000         95.49        95.49        95.49       654.50
        65536          640        183.20       183.20       183.20        682.31
<Oct 18 13:22:10.804667> FE_MPI (WARN) : SignalHandler() -
<Oct 18 13:22:10.804743> FE_MPI (WARN) : SignalHandler() -
!-------------------------------------------------!
<Oct 18 13:22:10.804769> FE_MPI (WARN) : SignalHandler() - ! mpirun is now taking all the
necessary actions !
<Oct 18 13:22:10.804794> FE_MPI (WARN) : SignalHandler() - ! to terminate the job and to free
the resources !
```

```
<Oct 18 13:22:10.804818> FE_MPI (WARN) : SignalHandler() - ! occupied by this job. This might
take a while... !
<Oct 18 13:22:10.804841> FE_MPI (WARN) : SignalHandler() -
!------------------------------------------------!
<Oct 18 13:22:10.804865> FE_MPI (WARN) : SignalHandler() -
       131072          320       357.97       357.97       357.97       698.38
<Oct 18 13:21:10.936378> BE_MPI (WARN) : Received a message from frontend
<Oct 18 13:21:10.936449> BE_MPI (WARN) : Execution of the current command interrupted
<Oct 18 13:21:16.140631> BE_MPI (ERROR): The error message in the job record is as follows:
<Oct 18 13:21:16.140678> BE_MPI (ERROR):   "killed with signal 9"
<Oct 18 13:22:16.320232> FE_MPI (ERROR): Failure list:
<Oct 18 13:22:16.320406> FE_MPI (ERROR):   - 1. Execution interrupted by signal (failure #71)
```

11.8 mpirun APIs

When writing programs to the **mpirun** APIs, you must consider these requirements:

- Currently, SUSE Linux Enterprise Server (SLES) 10 for PowerPC is the only supported platform.
- C and C++ are supported with the GNU gcc V4.1.2-level compilers. For more information and downloads, refer to the following Web address:

  http://gcc.gnu.org/
- The include file is include/sched_api.h.
- Only support for both 64-bit dynamic libraries is provided, and the 64-bit dynamic library file called by **mpirun** must be called libsched_if.so.

mpirun can retrieve run-time information directly from the scheduler without using command-line parameters or environment variables. Each time **mpirun** is invoked, it attempts to load a shared library called libsched_if.so. **mpirun** looks for this library in a set of directories as described by the dlopen() API manual pages.

If the plug-in library is found and successfully loaded, **mpirun** calls the get_parameters() function within that library to retrieve information from the scheduler. The get_parameters() function returns the information in a data structure of type sched_params. This data structure contains a set of fields that describe the partition that the scheduler has allocated the job to run. Each field corresponds to one of the command-line parameters or environment variables.

mpirun complements the information that is retrieved by get_parameters() with values from its command-line parameters and environment variables. It gives precedence to the information that is retrieved by get_parameters() first, then to its command-line parameters, and finally to the environment variables, for example, if the number of processors retrieved by get_parameters() is 256, the -np command-line parameter is set to 512, and the environment variable MPIRUN_NP is set to 448, **mpirun** runs the job on 256 Compute Nodes.

If **mpirun** is invoked with the -verbose parameter with a value greater than 0, it displays information that describes the loading of the dynamically loaded library. The message Scheduler interface library loaded indicates that **mpirun** found the library, loaded it, and is using it.

The implementation of the libsched_if.so library is scheduling-system specific. In general, this library should use the scheduler's APIs to retrieve the required information and convert it to the sched_params data type for **mpirun** to use. The only requirement is that the library interface conform to the definitions in the sched_api.h header file distributed with the **mpirun** binaries. This interface can be modified with future releases of **mpirun**.

The **mpirun** plug-in interface also requires the implementer provide an mpirun_done() or mpirun_done_enhanced() function. This function is called by **mpirun** just before it exits. It is used to signal the plug-in implementer that **mpirun** is terminating.

You can find more information about the library implementation and data structures in the sched_api.h header file.

The following APIs are supported for **mpirun**:

- `int get_parameters(sched_params_t* params)`

  This function is used to provide input parameters to **mpirun** from your application. If a value of 1 (failure) is returned on the get_parameters() call, then **mpirun** proceeds to terminate. Some external resource managers use this technique to prevent standalone **mpirun** from being used. If the plug-in provider wants **mpirun** processing to continue, then they must return a 0 (success) value on the get_parameters() call.

- `void job_started(sched_info_t* info)`

  This function is called by **mpirun** when the job starts. It can be used to get information, such as the job's database ID, about the job that **mpirun** started. This optional plug-in interface does not have to be included as part of the libsched_if.so shared library. This function was introduced in Blue Gene/P release V1R3M0.

- `void mpirun_done(int res)`

  This function is called by **mpirun** just before it calls the exit() function. It can be used to signal the scheduler that **mpirun** is terminating.

- `void mpirun_done_enhanced(sched_result_t* res)`

  This function is called by **mpirun** just before it calls the exit() function. It can be used to signal the scheduler that **mpirun** is terminating. This enhanced version of the original mpirun_done() callback is intended to convey information about boot failures to a resource scheduler. This optional plug-in interface does not have to be included as part of the libsched_if.so shared library.

12

High-Throughput Computing (HTC) paradigm

In Chapter 7, “Parallel paradigms” on page 65, we described the High-Performance Computing (HPC) paradigms. Applications that run in an HPC environment make use of the network to share data among MPI tasks. In other words, the MPI tasks are tightly coupled.

In this chapter we describe a paradigm that complements the HPC environment. This mode of running applications emphasizes IBM Blue Gene/P capacity. An application runs loosely coupled; that is, multiple instances of the applications do not require data communication. The concept of High-Throughput Computing (HTC) has been defined by Condor and others (see the following URL):

http://www.cs.wisc.edu/condor/htc.html

In this chapter we cover the implementation of HTC as part of Blue Gene/P functionality. We provide an overview of how HTC is implemented and how applications can take advantage of it. We cover the following topics:

- HTC design
- Booting a partition in HTC mode
- Running a job using submit
- Checking HTC mode
- submit API
- Altering the HTC partition user list

For a more detailed description of HTC, and how it is integrated into the control system of Blue Gene/P, see *IBM System Blue Gene Solution: Blue Gene/P System Administration*, SG24-7417.

12.1 HTC design

HTC focuses on pushing a large number of relatively short jobs through the control system. Such a design is significantly different from the traditional HPC focus of Blue Gene/P where a single job runs the same executable on each node in the partition. In HTC mode, each compute node can run one, two, or four different jobs depending on the mode (SMP and Linux-SMP, DUAL, or virtual node mode) the partition was booted in. Each job can run under a different user name, with separate stdin, stdout, and stderr. The executables are compiled in the same way that HPC executables are compiled for Blue Gene/P, but they cannot use MPI. Because each node can run a different executable, and jobs start and stop independently of each other, it does not make sense to use MPI in this mode.

12.2 Booting a partition in HTC mode

When booting an HTC partition, you need to specify the job mode at boot time, either SMP, DUAL, VN, or Linux-SMP mode. Specifying the job mode at boot time is different from regular MPI partitions where you can specify the mode at the run time of the job. The following methods can be used to boot a partition in HTC mode:

htcpartition The **htcpartition** utility can be executed on a front end node or the service node. For more information, see Appendix G, “htcpartition” on page 359.

mmcs_db_console A system administrator can boot a partition in HTC mode using the mmcs_db_console; see *IBM System Blue Gene Solution: Blue Gene/P System Administration*, SG24-7417.

Bridge APIs A resource scheduler can boot a partition in HTC mode using the Bridge APIs. For more information, see Chapter 13, “Control system (Bridge) APIs” on page 209.

12.3 Running a job using submit

The interface to run an HTC job is the Blue Gene/P **submit** command. It is similar to **mpirun** in the sense that it acts as a shadow of the job running on the compute node. It transparently forwards stdin, stdout, stderr, signals, and terminates when the job is complete. However, **mpirun** can boot either predefined partitions or dynamic partitions based on size and shape requirements of a job. **submit** performs neither of these tasks; it requires a partition to already be booted in HTC mode prior to submitting the job. Other differences are less noticeable to users, such as the lack of a hybrid front end and back end design, or Bridge APIs calls in **submit** that are present in **mpirun**. The arguments supported by **submit** are also somewhat different than **mpirun**. Many of the **mpirun** arguments are not applicable when running an HTC job compared to a large parallel job.

The syntax for the **submit** command is as follows:

```
./submit [options] or
./submit [options] binary [arg1 arg2... argn]
```

Table 12-1 contains the available options for the `submit` command.

Table 12-1 Options available for the submit command

Job options (and syntax)	Description
-exe <*exe*>	Executable to run.
-args *"arg1 arg2 ... argn"*	Arguments must be enclosed in double quotes.
-env <*env=value*>	Used to add an environment variable for the job.
-exp_env <*env*>	Used to export an environment variable to the job's environment.
-env_all	Adds all current environment variables to the job's environment.
-cwd <*cwd*>	Sets the current working directory for the job.
-timeout <*seconds*>	Number of seconds to wait for before the job is killed.
-strace	Run job under system call tracing.
-start_gdbserver <*path*>	Run the job under a debugger. The default system configuration provides GDB server in /sbin.rd/gdbserver.
Resource options	
-mode <SMP or DUAL or VN or LINUX_SMP>	Job mode. The default mode is SMP.
-location <*Rxx-Mx-Nxx-Jxx-Cxx*>	Compute core location (regular expressions are supported).
-pool <*id*>	Compute node pool ID.
General options	
-port <*port*>	Listen port of the submit mux (default = 10246).
-trace <0 - 7>	Tracing level (default = 0).
-enable_tty_reporting	Disable the default line buffering of stdin, stdout, and stderr when input (stdin) or output (stdout/stderr) is not a tty.
-raise	If a job dies with a signal, `submit` raises this signal.

Environment variables

Some arguments have a corresponding environment variable. If both an environment variable and an argument are given, precedence is given to the argument:

- --pool SUBMIT_POOL
- --cwd SUBMIT_CWD
- --port SUBMIT_PORT

Using submit

This section provides selected examples on how to invoke the command and how to use the location and pool arguments.

location argument

The --location argument requests a specific compute core location to run the job; the syntax is in the form of RXX-MX-NXX-JXX-CXX. The rack numbers (RXX) can range between R00 and RFF. The midplane numbers (MX) can range between M0 (bottom) and M1 (top). The node card numbers (NXX) can range between N00-N15. The compute card numbers (JXX) can range between J04 and J35. Note that J00 and J01 are I/O nodes, and J02 and J03 are unused. The compute core numbers (CXX) can range between C00 and C03. Note that C00 is valid for SMP, DUAL, and VN mode. Core C01 is valid only for VN mode. Core 02 is valid for VN and DUAL modes. Core 03 is valid only for VN mode.

The --location argument is combined with your user ID and --mode argument to find an available location to run the job. If any of these parameters do not match the list of what is available, the job is not started and an error message is returned. See Example 12-1.

It is also possible to omit a portion of the location. If the core is omitted (for example, --location R00-M0-N14-J09), one of the cores in the compute node is chosen. If the compute card is omitted (for example, --location R00-M0-N14), a core on a compute node on the node card is chosen.

Example 12-1 Requesting specific location

```
$ submit --cwd /bgusr/tests --location R00-M0-N14-J09-C00 --mode vn --exe hello
hello world
```

If the location you request is busy (job already running), you see an error message (like that shown in Example 12-2).

Example 12-2 Job already running

```
$ submit --cwd /bgusr/tests --exe hello --location R00-M0-N14-J09-C00 --mode vn
May 07 15:05:43 (U) [4398046675584]     (submit.cc:1237:cleanup) failed to add job: location
R00-M0-N14-J09-C00 is not available
```

If the location you request was booted in a mode different than the --mode argument you give, you see an error message (Example 12-3).

Example 12-3 Node mode conflict

```
$ submit --cwd /bgusr/tests --exe hello --location R00-M0-N14-J09-C00 --mode SMP
May 07 15:06:50 (U) [4398046675584]     (submit.cc:1237:cleanup) failed to add
job: location R00-M0-N14-J09-C00 mode (VN) is incompatible with requested mode
(SMP)
```

Similarly, if your user ID does not have permission to run on the location requested, you see an error message (Example 12-4).

Example 12-4 Permission problem

```
$ whoami
bgpadmin
$ submit --cwd /bgusr/tests --exe hello --location R00-M0-N14-J09-C00 --mode vn
```

```
May 07 15:13:28 (U) [4398046675888]     (submit.cc:1237:cleanup) failed to add
job: user bgpadmin is not allowed to run at location R00-M0-N14-J09-C00
```

pool argument

A pool is a collection of compute nodes and is represented by an ID just as a partition is. A pool consists of one or more partitions. By default, each partition's pool ID is its partition ID. Outside the framework of a job scheduler, this should always be the case. Thus, Example 12-5 shows how to run a job on any available compute node in partition CHEMISTRY.

Example 12-5 Pool argument

```
$ submit --pool CHEMISTRY--exe hello_world
May 07 16:28:59 (U) [4398046675888]     (submit.cc:1237:cleanup) failed to add
job: could not find available location matching resource request
```

If no compute nodes are available, an error message is displayed as shown in Example 12-5.

12.4 Checking HTC mode

An application can check whether the node it is running on was booted in HTC mode using the personality information. Example 12-6 illustrates how to use the personality to inquire about the HTC mode.

Example 12-6 Checking whether a node was booted in HTC mode

```
#include <unistd.h>
#include <common/bgp_personality.h>
#include <common/bgp_personality_inlines.h>
#include <spi/kernel_interface.h>

int
main()
{
    // get our personality
    _BGP_Personality_t pers;
    if (Kernel_GetPersonality(&pers, sizeof(pers)) == -1) {
        fprintf(stderr, "could not get personality\n");
        exit(EXIT_FAILURE);
    }

    // check HTC mode
    if (pers.Kernel_Config.NodeConfig & _BGP_PERS_ENABLE_HighThroughput) {
        // do something HTC specific
    } else {
        // do something else
    }
}
```

12.5 submit API

When writing programs to the submit APIs, you must consider these requirements:

- Currently, SUSE Linux Enterprise Server (SLES) 10 for PowerPC is the only supported platform.
- C and C++ are supported with the GNU gcc 4.1.2 level compilers. For more information and downloads, refer to the following web address:

  http://gcc.gnu.org
- The include file is include/submit_api.h.
- Only 64-bit dynamic libraries are supported.

submit can retrieve and provide run-time information directly from the scheduler without using command-line parameters or environmental variables. Each time **submit** is invoked, it attempts to load a dynamically loaded library called libsubmit_if.so. **submit** looks for this library in a series of directories as described by the dlopen() manual page. If the plug-in library is found and successfully loaded, **submit** invokes the following three methods:

- int get_parameters(htc_sched_params *params);

  This function is used to provide input parameters to **submit** from an external scheduler. A non-zero return code is fatal and causes submit to immediately exit without running a job.
- void submit_info(const htc_sched_info* result);

  This function is called by **submit** when the job is started. The htc_sched_info structure contains details about the job being started (pool, job ID, compute node location, partition, and so on). If the job never starts for whatever reason, this function is not called.
- void submit_done(const htc_sched_result *results);

  This function is called by **submit** just before it calls the exit() function. It can be used to signal the scheduler that **submit** is terminating. The htc_sched_result structure contains job metadata (pool, job ID, compute node location, return code, and so on). The job ID is 0 if the job did not run.

The return code information in the job metadata provides additional details about the **submit** request.

12.6 Altering the HTC partition user list

A HTC partition user list defines the set of users and groups that are enabled to submit jobs to the partition. A resource scheduler typically controls which users can run a job on a partition by manipulating the user list through the rm_add_part_user() Bridge APIs described in Chapter 13, “Control system (Bridge) APIs” on page 209.

Beginning in Blue Gene/P release V1R3M0, changes to a HTC partition user list after the partition is booted take effect immediately. As a result, resource schedulers do not have to free a partition to change the user list.

Part 4

Job scheduler interfaces

In this part, we provide information about the job scheduler APIs:

13

Control system (Bridge) APIs

In this chapter, we define a list of APIs into the Midplane Management Control System (MMCS) that can be used by a job management system. The **mpirun** program that ships with the Blue Gene/P software is an application that uses these APIs to manage partitions, jobs, and other similar aspects of the Blue Gene/P system. You can use these APIs to write applications to manage Blue Gene/P partitions and control Blue Gene/P job execution, as well as other similar administrative tasks.

In this chapter, we present an overview of the support provided by the APIs and discuss the following topics:

- API requirements
- APIs
- Small partition allocation
- API examples

13.1 API requirements

The several requirements for writing programs to the Bridge APIs are as follows:

- Currently, SUSE Linux Enterprise Server (SLES) 10 for PowerPC is the only supported platform.
- C and C++ are supported with the GNU gcc 4.1.2 level compilers. For more information and downloads, refer to the following Web address:

  `http://gcc.gnu.org/`
- All required include files are installed in the /bgsys/drivers/ppcfloor/include directory. See Appendix C, “Header files and libraries” on page 335 for additional information about include files. The include file for the Bridge APIs is rm_api.h.
- The Bridge APIs support 64-bit applications that use dynamic linking using shared objects. The required library files are installed in the /bgsys/drivers/ppcfloor/lib64 directory.

  The shared object for linking to the Bridge APIs is libbgpbridge.so. The libbgpbridge.so library has dependencies on other libraries that are included with the Blue Gene/P software, including:

  – libbgpconfig.so
  – libbgpdb.so
  – libsaymessage.so
  – libtableapi.so

  These files are installed with the standard system installation procedure. They are contained in the bgpbase.rpm file.

The requirements for writing programs to the Bridge APIs are explained in the following sections.

13.1.1 Configuring environment variables

Table 13-1 provides information about the environment variables that are used to control the Bridge APIs.

Table 13-1 Environment variables that control the Bridge APIs

Environment variable	Required	Description
DB_PROPERTY	Yes	This variable must be set to the path of the db.properties file with database connection information. For default installation, the path to this file is /bgsys/local/etc/db.properties.
BRIDGE_CONFIG	Yes	This variable must be set to the path of the bridge.config file that contains the Bridge APIs configuration values. For a default installation, the path to this file is /bgsys/local/etc/bridge.config.
BRIDGE_DUMP_XML	No	When set to any value, this variable causes the Bridge APIs to dump in-memory XML streams to files in /tmp for debugging. When this variable is not set, the Bridge APIs do not dump in-memory XML streams.

For more information about the db.properties and bridge.config files, see *IBM System Blue Gene Solution: Blue Gene/P System Administration*, SG24-7417.

13.1.2 General comments

All the APIs have general considerations that apply to all calls. In the following list, we highlight the common features:

- All the API calls return a `status_t` indicating either success or an error code.
- The `get` APIs that retrieve a compound structure include accessory functions to retrieve relevant nested data.
- The `get` calls allocate new memory for the structure to be retrieved and return a pointer to the allocated memory in the corresponding argument.
- To add information to MMCS, use `new` functions as well as rm_set_data(). The `new` functions allocate memory for new data structures, and the rm_set_data() API is used to fill these structures.
- For each `get` and `new` function, a corresponding `free` function frees the memory allocated by these functions. For instance, rm_get_BG(rm_BG_t **bg) is complemented by rm_free_BG(rm_BG_t *bg).
- The caller is responsible for matching the calls to the `get` and `new` allocators to the corresponding `free` deallocators. Memory leaks result if this is not done.

Memory allocation and deallocation

Some API calls result in memory being allocated on behalf of the user. The user must call the corresponding `free` function to avoid memory leaks, which can cause the process to run out of memory.

For the rm_get_data() API, see 13.2.8, “Field specifications for the rm_get_data() and rm_set_data() APIs” on page 229 for a complete list of the fields that require calls to free memory.

Avoiding invalid pointers

Some APIs return a pointer to an offset in a data structure, or object, that was previously allocated (based on *element* in rm_get_data()) - for example, the rm_get_data() API call uses the `RM_PartListNextPart` specification. In this example, *element* is a partition list, and it returns a pointer to the first or next partition in the list. If the caller of the API frees the memory of the partition list (*element*) and *data* is pointing to a subset of that freed memory, the *data* pointer is invalid. The caller must make sure that no further calls are made against a data structure returned from an rm_get_data() call after it is freed.

First and next calls

Before a *next* call can be made against a data structure returned from an rm_get_data() call, the *first* call must have been made. Failure to do so results in an invalid pointer, either pointing at nothing or at invalid data.

Example 13-1 shows correct usage of the first and next API calls. Notice how memory is freed after the list is consumed.

Example 13-1 Correct usage of first and next API calls

```
status_t stat;
int list_size = 0;
rm_partition_list_t * bgp_part_list = NULL;
rm_partition_t * bgp_part = NULL;

// Get all information on existing partitions
stat = rm_get_partitions_info(PARTITION_ALL_FLAG, &bgp_part_list);
```

```
if (stat != STATUS_OK) {
   // Do some error handling here...
   return;
}

// How much data (# of partitions) did we get back?
rm_get_data(bgp_part_list, RM_PartListSize, &list_size);

for (int i = 0; i < list_size; i++) {
   // If this is the first time through, use RM_PartListFirstPart
   if (i == 0){
      rm_get_data(bgp_part_list, RM_PartListFirstPart, &bgp_part);
   }
   // Otherwise, use RM_PartListNextPart
   else {
      rm_get_data(bgp_part_list, RM_PartListNextPart, &bgp_part);
   }
}

// Make sure we free the memory when finished
stat = rm_free_partition_list(bgp_part_list);
if (stat != STATUS_OK) {
   // Do some error handling here...
   return;
}
```

13.2 APIs

In the following sections, we provide details about the APIs.

13.2.1 API to the Midplane Management Control System

The Bridge APIs contain an rm_get_BG() function to retrieve current configuration and status information about all the physical components of the Blue Gene/P system from the MMCS database. The Bridge APIs also include functions that add, remove, or modify information about transient entities, such as jobs and partitions.

The rm_get_BG() function returns all the necessary information to define new partitions in the system. The information is represented by three lists: a list of base partitions (BPs), a list of wires, and a list of switches. This representation does not contain redundant data. In general, it allows manipulation of the retrieved data into any desired format. The information is retrieved using a structure called `rm_BG_t`. It includes the three lists that are accessed using iteration functions and the various configuration parameters, for example, the size of a base partition in Compute Nodes.

All the data retrieved by using the `get` functions can be accessed using rm_get_data() with one of the specifications listed in 13.2.8, “Field specifications for the rm_get_data() and rm_set_data() APIs” on page 229. Additional `get` functions can retrieve information about the partitions and job entities.

The rm_add_partition() and rm_add_job() functions add and modify data in the MMCS. The memory for the data structures is allocated by the new functions and updated using the rm_set_data() function. The specifications that can be set using the rm_set_data() function are shown in 13.2.8, “Field specifications for the rm_get_data() and rm_set_data() APIs” on page 229.

13.2.2 Asynchronous APIs

Some APIs that operate on partitions or jobs are documented as being asynchronous. Asynchronous means that control returns to your application before the operation requested is complete.

Before you perform additional operations on the partition or job, make sure that it is in a valid state by using the rm_get_partition_info() or rm_get_job() APIs to check the current state of the partition or job.

13.2.3 State sequence IDs

For most Blue Gene/P objects that have a *state* field, a corresponding sequence ID field exists for the state value. MMCS guarantees that whenever the state field changes for a given object, the associated sequence ID is incremented.

The sequence ID fields can be used to determine which state value is more recent. A state value with a higher corresponding sequence ID is the more recent value. This comparison can be helpful for applications that retrieve state information from multiple sources such as the Bridge APIs and the real-time APIs.

The function to increment sequence IDs only occurs if the real-time APIs are configured for the system. For information about configuring the real-time APIs, see *IBM System Blue Gene Solution: Blue Gene/P System Administration*, SG24-7417.

13.2.4 Bridge APIs return codes

When a failure occurs, an API invocation returns an error code. You can use the error code to take corrective actions within your application. In addition, a failure always generates a log message, which provides more information for the possible cause of the problem and an optional corrective action. These log messages are used for debugging and programmed recovery of failures.

The design aims at striking a balance between the number of error codes detected and the different error paths per return code. Thus, some errors have specific return codes, while others have more generic ones. The Bridge APIs have the following return codes:

- STATUS_OK: The invocation completed successfully.
- PARTITION_NOT_FOUND: The required partition specified by the ID cannot be found in the control system.
- JOB_NOT_FOUND: The required job specified by the ID cannot be found in the control system.
- BP_NOT_FOUND: One or more of the base partitions in the `rm_partition_t` structure do not exist.
- SWITCH_NOT_FOUND: One or more of the switches in the `rm_partition_t` structure do not exist.
- JOB_ALREADY_DEFINED: A job with the same name already exists.
- PARTITION_ALREADY_DEFINED: A partition already exists with the ID specified.
- CONNECTION_ERROR: The connection with the control system has failed or could not be established.
- INVALID_INPUT: The input to the API invocation is invalid, which is due to missing required data, illegal data, and so on.

- INCOMPATIBLE_STATE: The state of the partition or job prohibits the specific action. See Figure 13-1 on page 221, Figure 13-2 on page 226, Figure 13-3 on page 227, and Figure 13-4 on page 228 for state diagrams.
- INCONSISTENT_DATA: The data retrieved from the control system is not valid.
- INTERNAL_ERROR: Such errors do not belong to any of the previously listed categories, such as a memory allocation problem or failures during the manipulation of internal XML streams.

13.2.5 Blue Gene/P hardware resource APIs

In this section, we describe the APIs that are used to manage the hardware resources in the Blue Gene/P system:

- `status_t rm_get_BG(rm_BG_t **BG);`

  This function retrieves a snapshot of the Blue Gene/P machine, held in the `rm_BG_t` data structure.

  The following return codes are possible:

  - STATUS_OK
  - CONNECTION_ERROR
  - INCONSISTENT_DATA
    - List of base partitions is empty.
    - Wire list is empty, and the number of base partitions is greater than one.
    - Switch list is empty, and the number of base partitions is greater than one.
  - INTERNAL_ERROR

- `status_t rm_get_data(rm_element_t *rme, enum RMSpecification spec, void * result);`

  This function returns the content of the requested field from a valid `rm_element_t` (Blue Gene/P object, base partition object, wire object, switch object, and so on). The specifications that are available when using rm_get_data() are listed in 13.2.8, “Field specifications for the rm_get_data() and rm_set_data() APIs” on page 229, and are grouped by the object type that is being accessed.

  The following return codes are possible:

  - STATUS_OK
  - INVALID_INPUT
    - The specification `spec` is unknown.
    - The specification `spec` is illegal (per the “rme” element).
  - INTERNAL_ERROR

- `status_t rm_get_nodecards(rm_bp_id_t bpid, rm_nodecard_list_t **nc_list);`

  This function returns all node cards in the specified base partition.

  The following return codes are possible:

  - STATUS_OK
  - CONNECTION_ERROR
  - INCONSISTENT_DATA

    The base partition was not found.
  - INTERNAL_ERROR

- `status_t rm_get_serial(rm_serial_t *serial);`

  This function gets the machine serial number that was set previously by rm_set_serial().

  The following return codes are possible:

  - STATUS_OK
  - INTERNAL_ERROR

- `status_t rm_set_data(rm_element_t *rme, enum RMSpecification spec, void * result);`

  This function sets the value of the requested field in the `rm_element_t` (Blue Gene/P object, base partition object, wire object, switch object, and so on). The specifications, which are available when using rm_set_data(), are listed in 13.2.8, "Field specifications for the rm_get_data() and rm_set_data() APIs" on page 229, and are grouped by the object type that is being accessed.

  The following return codes are possible:

  - STATUS_OK
  - INVALID_INPUT
    - The specification `spec` is unknown.
    - The specification `spec` is illegal (per the `rme` element).
  - INTERNAL_ERROR

- `status_t rm_set_serial(rm_serial_t serial);`

  This function sets the machine serial number to be used in all the API calls following this call. The database can contain more than one machine. Therefore, it is necessary to specify which machine to work with.

  The following return codes are possible:

  - STATUS_OK
  - INVALID_INPUT
    - The machine serial number `serial` is NULL.
    - The machine serial number is too long.

13.2.6 Partition-related APIs

In this section, we describe the APIs used to create and manage partitions in the Blue Gene/P system:

- `status_t rm_add_partition(rm_partition_t* p);`

  This function adds a partition record to the database. The partition structure includes an ID field that is filled by the resource manager.

  The following return codes are possible:

  - STATUS_OK
  - CONNECTION_ERROR
  - INVALID_INPUT: The data in the **`rm_partition_t`** structure is invalid:
    - No base partition nor switch list is supplied.
    - Base partition or switches do not construct a legal partition.
    - No boot images or boot image name is too long.
    - No user or user name is too long.
  - BP_NOT_FOUND:

    One or more of the base partitions in the `rm_partition_t` structure does not exist.

   - SWITCH_NOT_FOUND:

     One or more of the switches in the `rm_partition_t` structure does not exist.

   - INTERNAL_ERROR

- `status_t rm_add_part_user (pm_partition_id_t partition_id, const char *user);`

  This function adds a new user to the partition. If a partition is in "free" state any user can add users. If the partition is in any other state only the partition's owner can add users.

  The following return codes are possible:

   - STATUS_OK
   - CONNECTION_ERROR
   - INVALID_INPUT:
     - `partition_id` is NULL or the length exceeds the limitations of the control system.
     - `user` is NULL or the length exceeds the limitations of the control system.
     - `user` is already defined as the partition's user.
   - INTERNAL_ERROR

- `status_t rm_assign_job(pm_partition_id_t partition_id, db_job_id_t jid);`

  This function assigns a job to a partition. A job can be created and simultaneously assigned to a partition by calling rm_add_job() with a partition ID. If a job is created and not assigned to a specific partition, it can be assigned later by calling rm_assign_job().

> **Note:** rm_assign_job() is not supported for HTC jobs.

  The following return codes are possible:

   - STATUS_OK
   - CONNECTION_ERROR
   - INVALID_INPUT

     `partition_id` is NULL or the length exceeds control system limitations:

   - PARTITION_NOT_FOUND
   - JOB_NOT_FOUND
   - INCOMPATIBLE_STATE

     The current state of the partition is not RM_PARTITION_READY ("initialized"), the partition and job owners do not match, or the partition is in HTC mode.

   - INTERNAL_ERROR

- `status_t pm_create_partition(pm_partition_id_t partition_id);`

  This function allocates the necessary hardware for a partition, boots the partition, and updates the resulting status in the MMCS database.

  > **Note**: This API is asynchronous. Control returns to your application before the operation requested is complete.

  The following return codes are possible:

   - STATUS_OK
   - CONNECTION_ERROR
   - INVALID_INPUT

     `partition_id` is NULL or the length exceeds control system limitations.

- PARTITION_NOT_FOUND
- INCOMPATIBLE_STATE

  The current state of the partition prohibits its creation. See Figure 13-1 on page 221.
- INTERNAL_ERROR

► `status_t pm_destroy_partition(pm_partition_id_t partition_id);`

This function shuts down a currently booted partition and updates the database accordingly.

Note: This API is asynchronous. Control returns to your application before the operation requested is complete.

The following return codes are possible:

- STATUS_OK
- CONNECTION_ERROR
- INVALID_INPUT

  `partition_id` is NULL or the length exceeds the limitations of the control system.
- PARTITION_NOT_FOUND
- INCOMPATIBLE_STATE

  The state of the partition prohibits its destruction. See Figure 13-1 on page 221.
- INTERNAL_ERROR

► `status_t rm_get_partition(pm_partition_id_t partition_id, rm_partition_t **p);`

This function retrieves a partition, according to its ID.

The following return codes are possible:

- STATUS_OK
- CONNECTION_ERROR
- INVALID_INPUT

  `partition_id` is NULL or the length exceeds the limitations of the control system.
- PARTITION_NOT_FOUND
- INCONSISTENT_DATA

  The base partition or switch list of the partition is empty.
- INTERNAL_ERROR

► `status_t rm_get_partitions(rm_partition_state_t_flag_t flag, rm_partition_list_t **part_list);`

This function is useful for status reports and diagnostics. It returns a list of partitions whose current state matches the flag. The possible flags are contained in the rm_api.h include file and listed in Table 13-2 on page 218. You can use OR on these values to create a flag for including partitions with different states.

The following return codes are possible:

- STATUS_OK
- CONNECTION_ERROR
- INCONSISTENT_DATA

  At least one of the partitions has an empty base partition list.

- INTERNAL_ERROR

- `status_t rm_get_partitions_info(rm_partition_state_t_flag_t flag, rm_partition_list_t ** part_list);`

  This function is useful for status reports and diagnostics. It returns a list of partitions whose current state matches the flag. This function returns the partition information without their base partitions, switches, and node cards.

  The possible flags are contained in the rm_api.h include file and are listed in Table 13-2. You can use OR on these values to create a flag for including partitions with different states.

Table 13-2 Flags for partition states

Flag	Value
PARTITION_FREE_FLAG	0x01
PARTITION_CONFIGURING_FLAG	0x02
PARTITION_READY_FLAG	0x04
PARTITION_DEALLOCATING_FLAG	0x10
PARTITION_ERROR_FLAG	0x20
PARTITION_REBOOTING_FLAG	0x40
PARTITION_ALL_FLAG	0xFF

The following return codes are possible:

- STATUS_OK
- CONNECTION_ERROR
- INCONSISTENT_DATA

  At least one of the partitions has an empty base partition list.

- INTERNAL_ERROR

- `status_t rm_modify_partition(pm_partition_id_t partition_id, enum rm_modify_op modify_option, const void *value);`

  This function makes it possible to change a set of fields in an already existing partition. The fields that can be modified are owner, description, options, boot options, HTC pool, and the partition boot images. The `modify_option` parameter identifies the field to be modified. To change the HTC pool, the partition must be in the RM_PARTITION_READY (“initialized”) state. The other modifiable fields require the partition to be in the RM_PARTITION_FREE (“free”) state.

  The following return codes are possible:

- STATUS_OK
- CONNECTION_ERROR
- INVALID_INPUT
  - `partition_id` is NULL, or the length exceeds the limitations of the control system.
  - The value for the `modify_option` parameter is not valid.

  - PARTITION_NOT_FOUND
  - INCOMPATIBLE_STATE

    The partition's current state forbids its modification. See Figure 13-1 on page 221.

  - INTERNAL_ERROR

- `status_t pm_reboot_partition(pm_partition_id_t partition_id);`

  This function sends a request to reboot a partition and update the resulting status in the database.

  > **Note**: This API is asynchronous. Control returns to your application before the operation requested is complete.

  The following return codes are possible:

  - STATUS_OK
  - CONNECTION_ERROR
  - INVALID_INPUT
    - `partition_id` is NULL, or the length exceeds the limitations of the control system.
    - This API is not supported for HTC partitions.
  - PARTITION_NOT_FOUND
  - INCOMPATIBLE_STATE

    The partition's current state forbids it to be rebooted. See Figure 13-1 on page 221.

  - INTERNAL_ERROR

- `status_t rm_release_partition(pm_partition_id_t partition_id);`

  This function is the opposite of rm_assign_job() because it releases the partition from all jobs. Only jobs that are in an `RM_JOB_IDLE` state have their partition reference removed.

  The following return codes are possible:

  - STATUS_OK
  - CONNECTION_ERROR
  - INVALID_INPUT

    `partition_id` is NULL, or the length exceeds the limitations of the control system (configuration parameter).

  - PARTITION_NOT_FOUND
  - INCOMPATIBLE_STATE

    The current state of one or more jobs assigned to the partition prevents this release. See Figure 13-1 on page 221 and Figure 13-2 on page 226.

  - INTERNAL_ERROR

- `status_t rm_remove_partition(pm_partition_id_t partition_id);`

  This function removes the specified partition record from MMCS.

  The following return codes are possible:

  - STATUS_OK
  - CONNECTION_ERROR
  - INVALID_INPUT

    `partition_id` is NULL, or the length exceeds the limitations of the control system (configuration parameter).

  - PARTITION_NOT_FOUND
  - INCOMPATIBLE_STATE

    The partition's current state forbids its removal. See Figure 13-1 on page 221 and Figure 13-2 on page 226.

  - INTERNAL_ERROR

- `status_t rm_remove_part_user(pm_partition_id_t partition_id, const char *user);`

  This function removes a user from a partition. Removing a user from a partition can be done only by the partition owner. A user can be removed from a partition that is in any state. Once a HTC partition is booted, this API can still be used, but the submit server daemon running on the service node ignores any removed users. Those removed users are still allowed to run jobs on the partition.

  The following return codes are possible:

  - STATUS_OK
  - CONNECTION_ERROR
  - INVALID_INPUT
    - `partition_id` is NULL, or the length exceeds the limitations of the control system (configuration parameter).
    - `user` is NULL, or the length exceeds the limitations of the control system.
    - `user` is already defined as the partition's user.
    - Current user is not the partition owner.
  - INTERNAL_ERROR

- `status_t rm_set_part_owner(pm_partition_id_t partition_id, const char *user);`

  This function sets the new owner of the partition. Changing the partition's owner can be done only to a partition in the `RM_PARTITION_FREE` state.

  The following return codes are possible:

  - STATUS_OK
  - CONNECTION_ERROR
  - INVALID_INPUT
    - `partition_id` is NULL, or the length exceeds the limitations of the control system (configuration parameter).
    - `owner` is NULL, or the length exceeds the limitations of the control system.
  - INTERNAL_ERROR

- status_t rm_get_htc_pool(pm_pool_id_t pid, rm_partition_list_t **p)

  This function is useful for status reports and diagnostics. It returns a list of partitions whose HTC pool id matches the parameter.

  The following return codes are possible:

  - STATUS_OK
  - CONNECTION_ERROR
  - INCONSISTENT_DATA

    At least one of the partitions has an empty base partition list.

  - INTERNAL_ERROR

State transition diagram for partitions

Figure 13-1 illustrates the states that a partition goes through during its life cycle. For HTC partitions RM_PARTITION_REBOOTING is not a possible state.

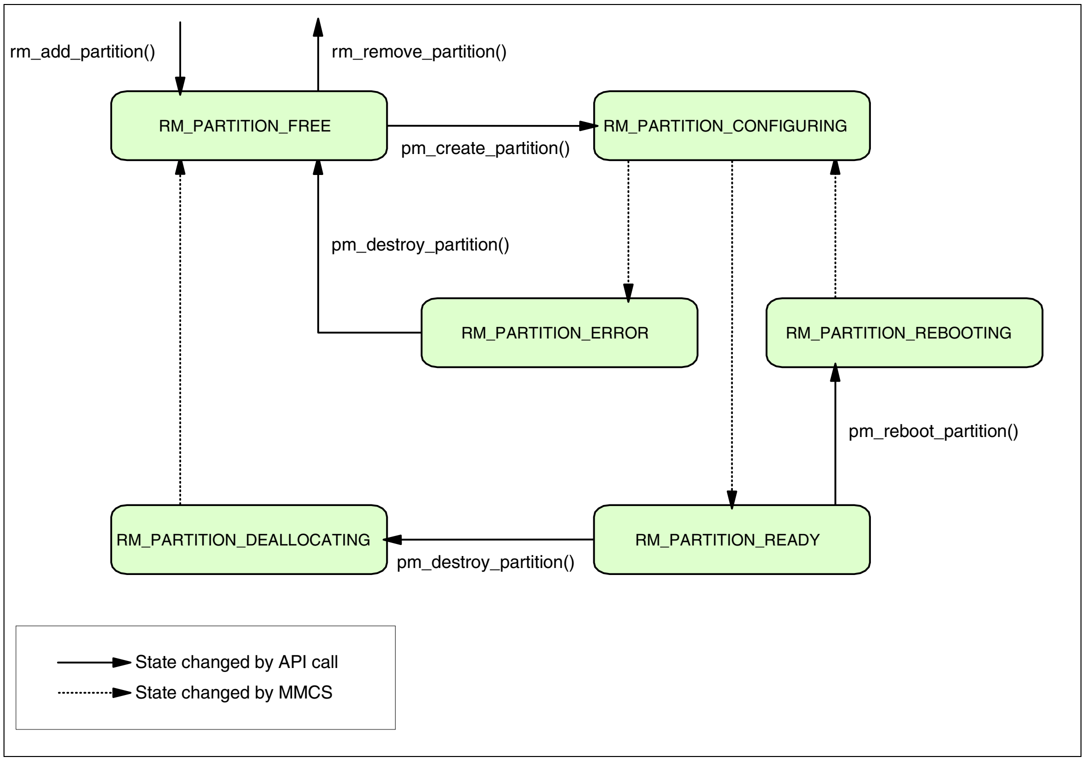


Figure 13-1 Partition state diagram

13.2.7 Job-related APIs

In this section, we describe the APIs to create and manage jobs in the Blue Gene system:

- `status_t rm_add_job(rm_job_t *job);`

  This function adds a job record to the database. The job structure includes an ID field that will be filled by the resource manager.

> **Note:** rm_add_job() is not supported for HTC jobs.

  The following return codes are possible:

  – STATUS_OK
  – CONNECTION_ERROR
  – INVALID_INPUT:

    - Data in the `rm_job_t` structure is invalid.
    - There is no job name, or a job name is too long.
    - There is no user, or the user name is too long.
    - There is no executable, or the executable name is too long.
    - The output or error file name is too long.

  - JOB_ALREADY_DEFINED

    A job with the same name already exists.

  - INTERNAL_ERROR

- `status_t jm_attach_job(db_job_id_t jid);`

  This function initiates the spawn of debug servers to a job in the `RM_JOB_LOADED` state.

Note: jm_attach_job() is not supported for HTC jobs.

  The following return codes are possible:

  - STATUS_OK
  - CONNECTION_ERROR
  - JOB_NOT_FOUND
  - INCOMPATIBLE_STATE

    The job's state prevents it from being attached. See Figure 13-2 on page 226.

  - INTERNAL_ERROR

- `status_t jm_begin_job(db_job_id_t jid);`

  This function begins a job that is already loaded.

Note: jm_begin_job() is not supported for HTC jobs.

  The following return codes are possible:

  - STATUS_OK
  - CONNECTION_ERROR
  - JOB_NOT_FOUND
  - INCOMPATIBLE_STATE

    The job's state prevents it from beginning. See Figure 13-2 on page 226.

  - INTERNAL_ERROR

- `status_t jm_cancel_job(db_job_id_t jid);`

  This function sends a request to cancel the job identified by the `jid` parameter.

  Note: This API is asynchronous. Control returns to your application before the operation requested is complete.

  The following return codes are possible:

  - STATUS_OK
  - CONNECTION_ERROR
  - JOB_NOT_FOUND
  - INCOMPATIBLE_STATE

    The job's state prevents it from being canceled. See Figure 13-2 on page 226.

  - INTERNAL_ERROR

- `status_t jm_debug_job(db_job_id_t jid);`

  This function initiates the spawn of debug servers to a job in the `RM_JOB_RUNNING` state.

Note: jm_debug_job() is not supported for HTC.

  The following return codes are possible:

  – STATUS_OK
  – CONNECTION_ERROR
  – JOB_NOT_FOUND
  – INCOMPATIBLE_STATE

    The job's state prevents it from being debugged. See Figure 13-2 on page 226.

  – INTERNAL_ERROR

- `status_t rm_get_job(db_job_id_t jid, rm_job_t **job);`

  This function retrieves the specified job object.

  The following return codes are possible:

  – STATUS_OK
  – CONNECTION_ERROR
  – JOB_NOT_FOUND
  – INTERNAL_ERROR

- `status_t rm_get_jobs(rm_job_state_flag_t flag, rm_job_list_t **job_list);`

  This function returns a list of jobs whose current state matches the flag.

  The possible flags are contained in the rm_api.h include file and are listed in Table 13-3. You can use OR on these values to create a flag for including jobs with different states.

Table 13-3 Flags for job states

Flag	Value
JOB_IDLE_FLAG	0x001
JOB_STARTING_FLAG	0x002
JOB_RUNNING_FLAG	0x004
JOB_TERMINATED_FLAG	0x008
JOB_ERROR_FLAG	0x010
JOB_DYING_FLAG	0x020
JOB_DEBUG_FLAG	0x040
JOB_LOAD_FLAG	0x080
JOB_LOADED_FLAG	0x100
JOB_BEGIN_FLAG	0x200
JOB_ATTACH_FLAG	0x400
JOB_KILLED_FLAG	0x800

The following return codes are possible:

- STATUS_OK
- CONNECTION_ERROR
- INTERNAL_ERROR

- `status_t jm_load_job(db_job_id_t jid);`

  This function sets the job state to LOAD.

> **Note:** jm_load_job() is not supported for HTC jobs.

The following return codes are possible:

- STATUS_OK
- CONNECTION_ERROR
- JOB_NOT_FOUND
- INCOMPATIBLE_STATE

  The job's state prevents it from being loaded. See Figure 13-2 on page 226.
- INTERNAL_ERROR

- `status_t rm_query_job(db_job_id_t db_job_id, MPIR_PROCDESC **proc_table, int * proc_table_size);`

  This function fills the `proc_table` with information about the specified job.

> **Note:** rm_query_job() is not supported for HTC jobs.

The following return codes are possible:

- STATUS_OK
- CONNECTION_ERROR
- JOB_NOT_FOUND
- INTERNAL_ERROR

- `status_t rm_remove_job(db_job_id_t jid);`

  This function removes the specified job record from MMCS.

> **Note:** rm_remove_job() is not supported for HTC jobs.

The following return codes are possible:

- STATUS_OK
- CONNECTION_ERROR
- JOB_NOT_FOUND
- INCOMPATIBLE_STATE

  The job's state prevents its removal. See Figure 13-2 on page 226.

- INTERNAL_ERROR

- `status_t jm_signal_job(db_job_id_t jid, rm_signal_t signal);`

  This function sends a request to signal the job identified by the `jid` parameter.

  > **Note**: This API is asynchronous. Control returns to your application before the operation requested is complete.

  The following return codes are possible:

  - STATUS_OK
  - CONNECTION_ERROR
  - JOB_NOT_FOUND
  - INCOMPATIBLE_STATE

    The job's state prevents it from being signaled.

  - INTERNAL_ERROR

- `status_t jm_start_job(db_job_id_t jid);`

  This function starts the job identified by the `jid` parameter. Note that the partition information is referenced from the job record in MMCS.

  > **Note:** jm_start_job() is *not* supported for HTC jobs.

  > **Note**: This API is asynchronous. Control returns to your application before the operation requested is complete.

  The following return codes are possible:

  - STATUS_OK
  - CONNECTION_ERROR
  - JOB_NOT_FOUND
  - INCOMPATIBLE_STATE

    The job's state prevents its execution. See Figure 13-2 on page 226.

  - INTERNAL_ERROR

- `status_t rm_get_filtered_jobs(rm_job_filter_t query_parms, rm_job_list_t **job_list);`

This function returns a list of jobs whose attributes or states (or both) that match the fields specified in the filter provided in the rm_job_filter object.

The following return codes are possible:

- STATUS_OK
- CONNECTION_ERROR
- INTERNAL_ERROR

State transition diagrams for jobs

Figure 13-2 illustrates the states that a job goes through during its life cycle. It also illustrates the order of API calls for creating, running, and canceling a job.

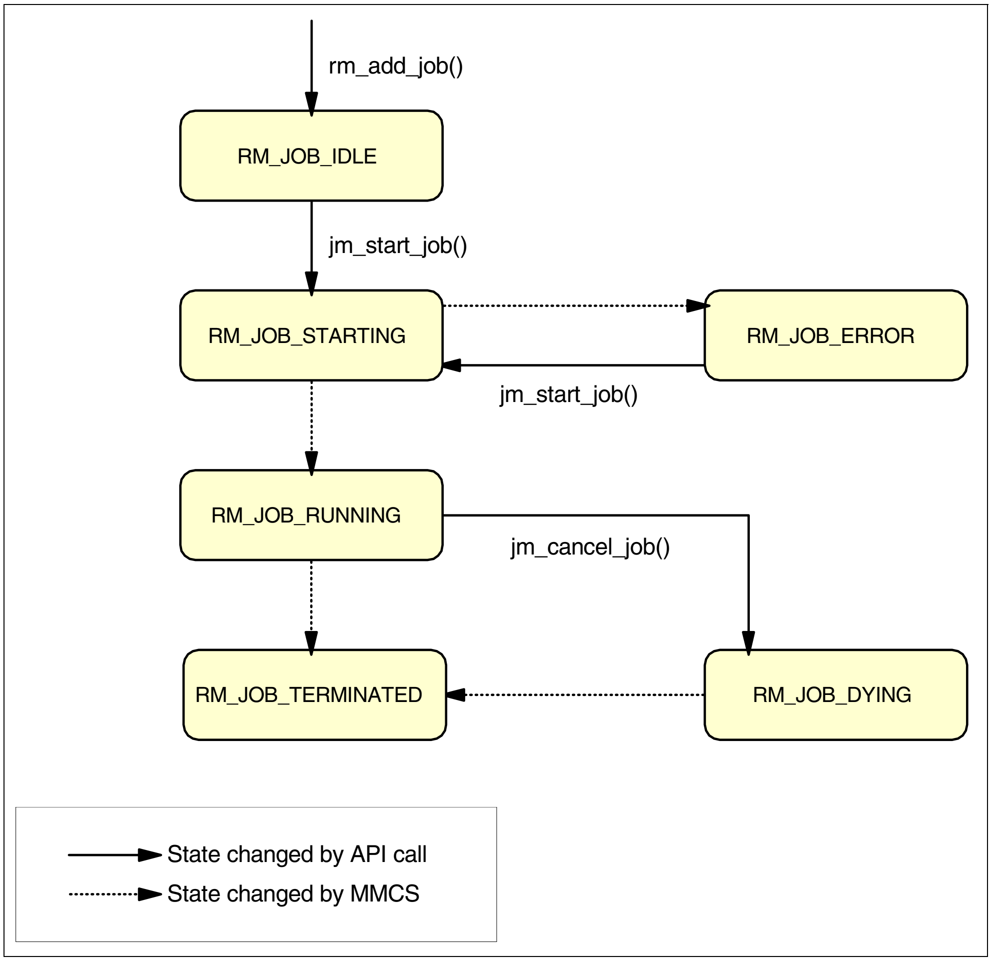


Figure 13-2 Job state diagram for running a Blue Gene/P job

Figure 13-3 illustrates the main states that a job goes through when debugging a new job.

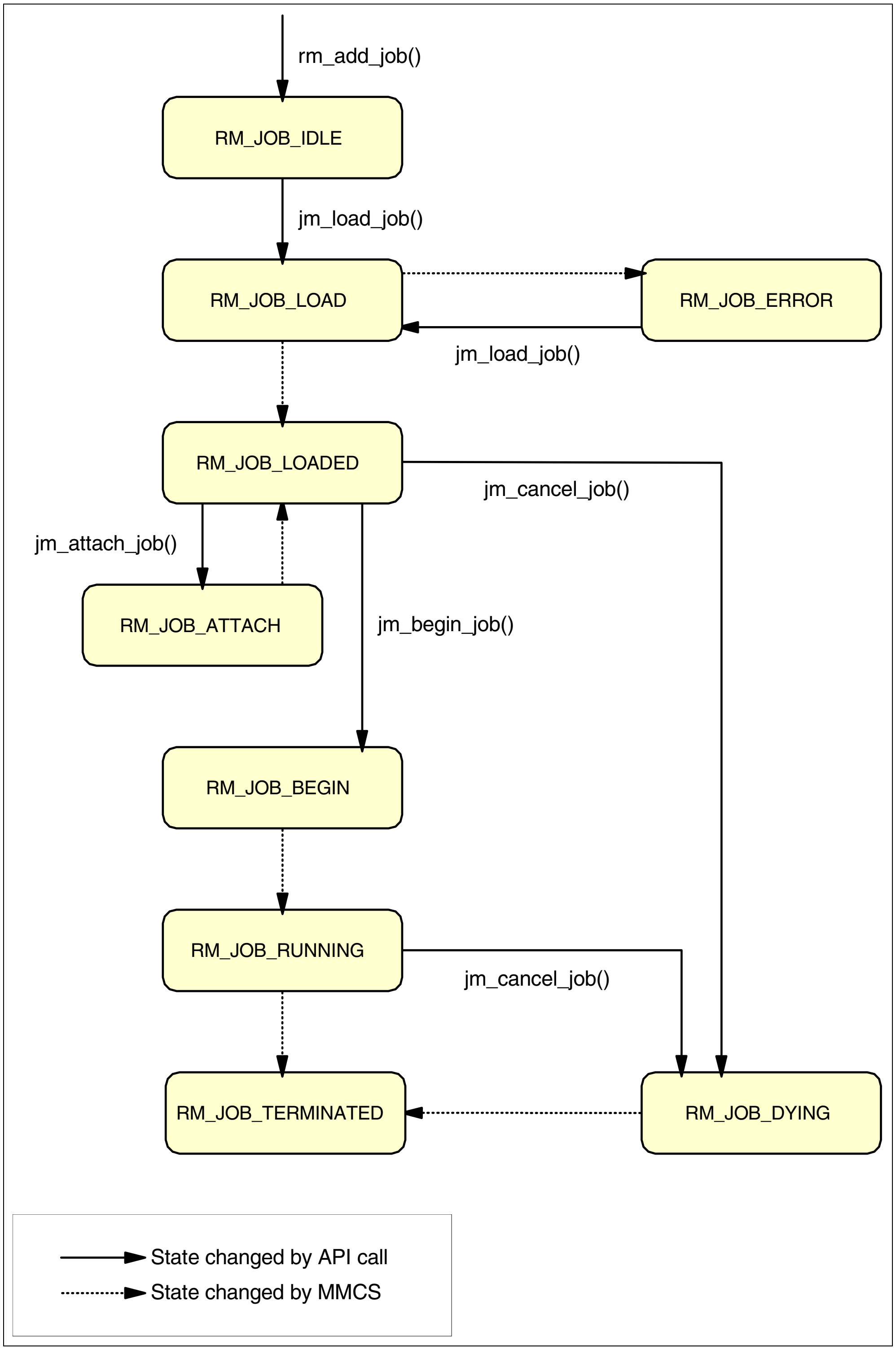


Figure 13-3 Job state diagram for debugging a running job

Figure 13-4 illustrates the states a job goes through when debugging an already running job.

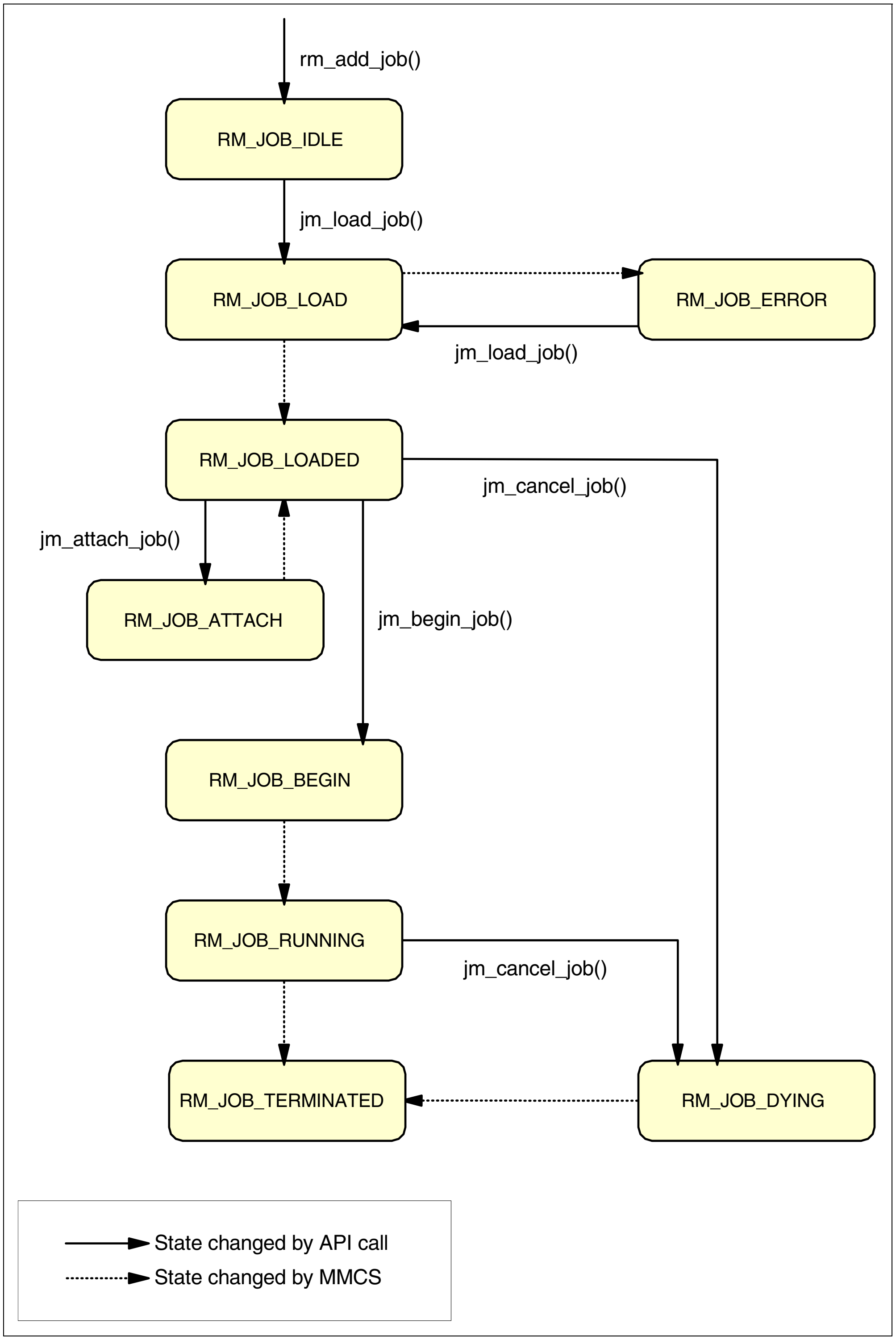


Figure 13-4 Job state diagram for debugging a new job

Figure 13-5 illustrates the states that a job goes through during its life cycle in HTC mode. It also illustrates that the **submit** command is required.

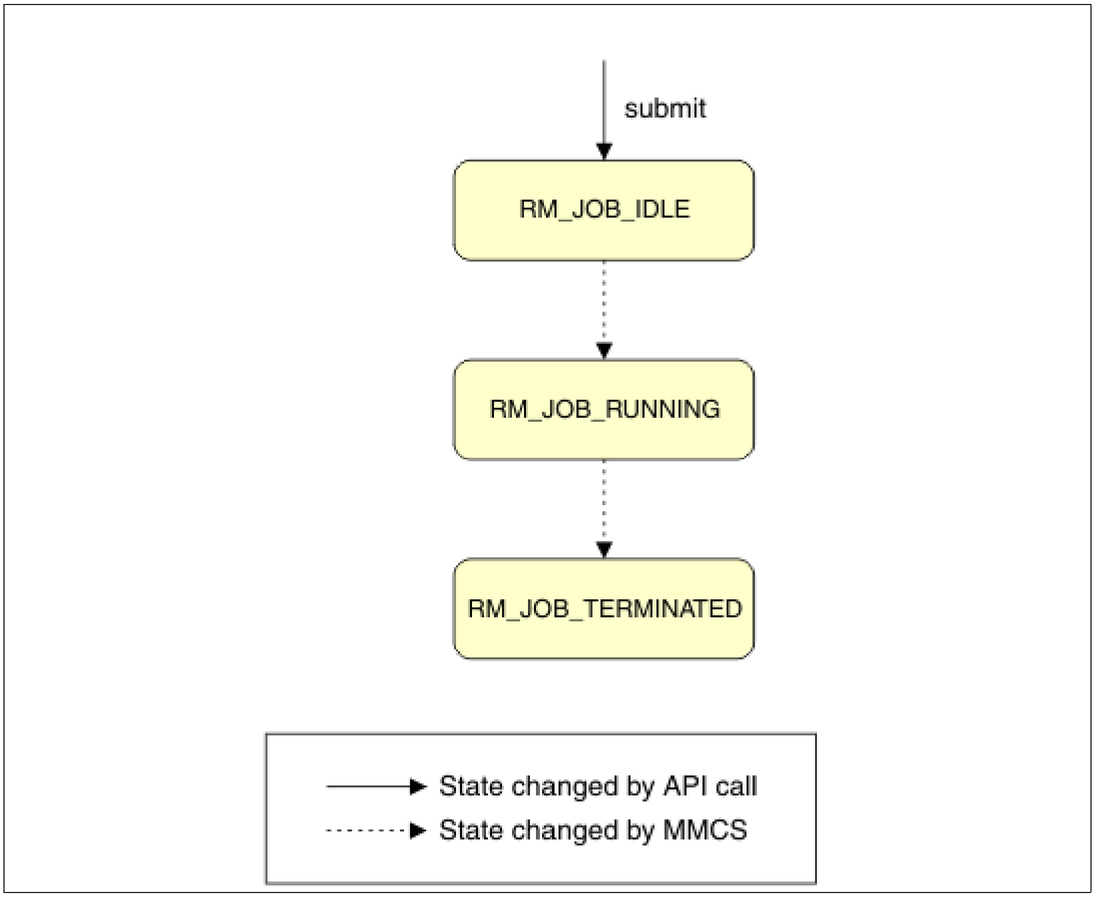


Figure 13-5 Job state diagram for HTC mode

13.2.8 Field specifications for the rm_get_data() and rm_set_data() APIs

In this section, we describe all the field specifications that can be used to get and set fields from various objects using the rm_get_data() and rm_set_data() APIs.

Blue Gene object

The Blue Gene object (rm_BG_t) represents the Blue Gene/P system. You can use this object to retrieve information and status for other components in the system, such as base partitions, node cards, I/O nodes, switches, wires, and port (see Table 13-4 on page 230). The Blue Gene object is retrieved by calling the rm_get_BG() API.

Table 13-4 Values retrieved from a Blue Gene object using rm_get_data()

Description	Specification	Argument type	Notes
Size of a base partition (in Compute Nodes) in each dimension	RM_BPsize	rm_size3D_t *	
Size of the machine in base partition units	RM_Msize	rm_size3D_t *	
Number of base partitions in the machine	RM_BPNum	int *	
First base partition in the list	RM_FirstBP	rm_BP_t **	
Next base partition in the list	RM_NextBP	rm_BP_t **	
Number of switches in the machine	RM_SwitchNum	int *	
First switch in the list	RM_FirstSwitch	rm_switch_t **	
Next switch in the list	RM_NextSwitch	rm_switch_t **	
Number of wires in the machine	RM_WireNum	int *	
First wire in the list	RM_FirstWire	rm_wire_t **	
Next wire in the list	RM_NextWire	rm_wire_t **	

Base partition object

The base partition object (rm_BP_t) represents one base partition in the Blue Gene system. The base partition object is retrieved from the Blue Gene object using either the RM_FirstBP or RM_NextBP specification. See Table 13-5.

Table 13-5 Values retrieved from a base partition object using rm_get_data()

Description	Specification	Argument type	Notes
Base partition identifier	RM_BPID	rm_bp_id_t *	free required
Base partition state	RM_BPState	rm_BP_state_t *	
Sequence ID for the base partition state	RM_BPStateSeqID	rm_sequence_id_t *	
Location of the base partition in the 3D machine	RM_BPLoc	rm_location_t *	
Identifier of the partition assocated with the base partition	RM_BPPartID	pm_partition_id_t *	free required. If no partition is associated, NULL is returned.
State of the partition assocated with the base partition	RM_BPPartState	rm_partition_state_t *	

Description	Specification	Argument type	Notes
Sequence ID for the state of the partition assocated with the base partition	RM_BPStateSeqID	rm_sequence_id_t *	
Flag indicating whether this base partition is being used by a small partition (smaller than a base partition)	RM_BPSDB	int *	0=No 1=Yes
Flag indicating whether this base partition is being divided into one or more small partitions	RM_BPSD	int *	0=No 1=Yes
Compute node memory size for the base partition	RM_BPComputeNodeMemory	rm_BP_computenode_memory_t *	
Number of available node cards	RM_BPAvailableNodeCards	int *	
Number of available I/O nodes	RM_BPNumberIONodes	int *	

Table 13-6 shows the values that are set in the base partition object using rm_set_data().

Table 13-6 Values set in a base partition object using rm_set_data()

Description	Specification	Argument type	Notes
Base partition identifier	RM_BPID	rm_bp_id_t	free required

Node card list object

The node card list object (rm_nodecard_list_t) contains a list of node card objects. The node card list object is retrieved by calling the rm_get_nodecards() API for a given base partition. See Table 13-7.

Table 13-7 Values retrieved from a node card list object using rm_get_data()

Description	Specification	Argument type	Notes
Number of node cards in the list	RM_NodeCardListSize	int *	
First node card in the list	RM_NodeCardListFirst	rm_nodecard_t **	
Next node card in the list	RM_NodeCardListNext	rm_nodecard_t **	

Node card object

The node card object (rm_nodecard_t) represents a node card within a base partition. The node card object is retrieved from the node card list object using the RM_NodeCardListFirst and RM_NodeCardListNext specifications. See Table 13-8 on page 232.

Table 13-8 Values retrieved from a node card object using rm_get_data()

Description	Specification	Argument type	Notes
Node card identifier	`RM_NodeCardID`	`rm_nodecard_id_t *`	free required; possible values: N00..N15
The quadrant of the base partition where this node card is installed	`RM_NodeCardQuarter`	`rm_quarter_t *`	
Node card state	`RM_NodeCardState`	`rm_nodecard_state_t *`	
Sequence ID for the node card state	`RM_NodeCardStateSeqID`	`rm_sequence_id_t *`	
Number of I/O nodes on the node card (can be 0, 1, or 2)	`RM_NodeCardIONodes`	`int *`	
Identifier of the partition assocated with the node card	`RM_NodeCardPartID`	`pm_partition_id_t *`	free required. If no partition is associated, NULL is returned.
State of the partition assocated with the node card	`RM_NodeCardPartState`	`rm_partition_state_t *`	
Sequence ID for the state of the partition assocated with the node card	`RM_NodeCardPartStateSeqID`	`rm_sequence_id_t *`	
Flag indicating whether the node card is being used by a partition whose size is smaller than a node card	`RM_NodeCardSDB`	`int *`	0=No 1=Yes
Number of I/O nodes in a list	`RM_NodeCardIONodeNum`	`int *`	
First I/O node in the node card	`RM_NodeCardFirstIONode`	`rm_ionode_t **`	
Next I/O node in the node card	`RM_NodeCardNextIONode`	`rm_ionode_t **`	

Table 13-9 shows the values that are set in a node card object when using `rm_set_data()`.

Table 13-9 Values set in a node card object using rm_set_data()

Description	Specification	Argument type	Notes
Node card identifier	`RM_NodeCardID`	`rm_nodecard_id_t`	
Number of I/O nodes in list	`RM_NodeCardIONodeNum`	`int *`	
First I/O node in the node card	`RM_NodeCardFirstIONode`	`rm_ionode_t *`	
Next I/O node in the node card	`RM_NodeCardNextIONode`	`rm_ionode_t *`	

I/O node object

The I/O node object (rm_ionode_t) represents an I/O node within a node card. The I/O node object is retrieved from the node card object using the RM_NodeCardFirstIONode and RM_NodeCardNextIONode specifications. See Table 13-10.

Table 13-10 Values retrieved from an I/O node object using rm_get_data()

Description	Specification	Argument type	Notes
I/O node identifier	RM_IONodeID	rm_ionode_id_t *	Possible values: J00, J01; free required
Node card identifier	RM_IONodeNodeCardID	rm_nodecard_id_t *	Possible values: N00..N15; free required
IP address	RM_IONodeIPAddress	char **	free required
MAC address	RM_IONodeMacAddress	char **	free required
Identifier of the partition assocated with the I/O node	RM_IONodePartID	pm_partition_id_t *	free required. If no partition is associated with this I/O node, NULL is returned.
State of the partition assocated with the I/O node	RM_IONodePartState	rm_partition_state_t *	
Sequence ID for the state of the partition associated with the I/O node	RM_IONodePartStateSeqID	rm_sequence_id_t *	

Table 13-11 shows the values that are set in an I/O node object by using rm_set_data().

Table 13-11 Values set in an I/O node object using rm_set_data()

Description	Specification	Argument type	Notes
I/O node identifier	RM_IONodeID	rm_ionode_id_t	Possible values: J00, J01

Switch object

The switch object (rm_switch_t) represents a switch in the Blue Gene/P system. The switch object is retrieved from the following specifications:

- The Blue Gene object using the RM_FirstSwitch and RM_NextSwitch specifications
- The partition object using the RM_PartitionFirstSwitch and RM_PartitionNextSwitch specifications

Table 13-12 shows the values that are retrieved from a switch object using rm_get_data().

Table 13-12 Values retrieved from a switch object using rm_get_data()

Description	Specification	Argument type	Notes
Switch identifier	`RM_SwitchID`	`rm_switch_id_t *`	free required
Identifier of the base partition connected to the switch	`RM_SwitchBPID`	`rm_BP_id_t *`	free required
Switch state	`RM_SwitchState`	`rm_switch_state_t *`	
Sequence ID for the switch state	`RM_SwitchStateSeqID`	`rm_sequence_id_t *`	
Switch dimension	`RM_SwitchDim`	`rm_dimension_t *`	Values: ▸ RM_DIM_X ▸ RM_DIM_Y ▸ RM_DIM_Z
Number of connections in the switch	`RM_SwitchConnNum`	`int *`	A connection is a pair of ports that are connected internally in the switch.
First connection in the list	`RM_SwitchFirstConnection`	`rm_connection_t *`	
Next connection in the list	`RM_SwitchNextConnection`	`rm_connection_t *`	

Table 13-13 shows the values that are set in a switch object using rm_set_data().

Table 13-13 Values set in a switch object using rm_set_data()

Description	Specification	Argument type	Notes
Switch identifier	`RM_SwitchID`	`rm_switch_id_t *`	
Number of connections in the switch	`RM_SwitchConnNum`	`int *`	A connection is a pair of ports that are connected internally in the switch.
First connection in the list	`RM_SwitchFirstConnection`	`rm_connection_t *`	
Next connection in the list	`RM_SwitchNextConnection`	`rm_connection_t *`	

Wire object

The wire object (`rm_wire_t`) represents a wire in the Blue Gene/P system. The wire object is retrieved from the Blue Gene/P object using the `RM_FirstWire` and `RM_NextWire` specifications. See Table 13-14 on page 235.

Table 13-14 Values retrieved from a wire object using rm_get_data()

Description	Specification	Argument type	Notes
Wire identifier	RM_WireID	rm_wire_id_t *	free required.
Wire state	RM_WireState	rm_wire_state_t *	The state can be UP or DOWN.
Source port	RM_WireFromPort	rm_port_t **	
Destination port	RM_WireToPort	rm_port_t **	
Identifier of the partition associated with the wire	RM_WirePartID	pm_partition_id_t *	free required. If no partition is associated, NULL is returned.
State of the partition associated with the wire	RM_WirePartState	rm_partition_state_t *	
Sequence ID for the state of the partition associated with the wire	RM_WirePartStateSeqID	rm_sequence_id_t *	

Port object

The port object (rm_port_t) represents a port for a switch in the Blue Gene. The port object is retrieved from the wire object using the RM_WireFromPort and RM_WireToPort specifications. See Table 13-15.

Table 13-15 Values retrieved from a port object using rm_get_data()

Description	Specification	Argument type	Notes
Identifier of the base partition or switch associated with the port	RM_PortComponentID	rm_component_id_t *	free required
Port identifier	RM_PortID	rm_port_id_t *	Possible values for base partitions: plus_x minus_x, plus_y, minus_y, plus_z minus_z. Possible values for switches: s0 . . . S5

Partition list object

The partition list object (rm_partition_list_t) contains a list of partition objects. The partition list object is retrieved by calling the rm_get_partitions() or rm_get_partitions_info() APIs. See Table 13-16.

Table 13-16 Values retrieved from a partition list object using rm_get_data()

Description	Specification	Argument type	Notes
Number of partitions in the list	RM_PartListSize	int *	
First partition in the list	RM_PartListFirstPart	rm_partition_t **	
Next partition in the list	RM_PartListNextPart	rm_partition_t **	

Partition object

The partition object (rm_partition_t) represents a partition that is defined in the Blue Gene system. The partition object is retrieved from the partition list object using the RM_PartListFirstPart and RM_PartListNextPart specifications. A new partition object is created using the rm_new_partition() API. After setting the appropriate fields in a new partition object, the partition can be added to the system using the rm_add_partition() API. See Table 13-17 on page 236.

Table 13-17 Values retrieved from a partition object using rm_get_data()

Description	Specification	Argument type	Notes
Partition identifier	`RM_PartitionID`	`pm_partition_id_t *`	free required
Partition state	`RM_PartitionState`	`rm_partition_state_t *`	
Sequence ID for the partition state	`RM_PartitionStateSeqID`	`rm_sequence_id_t *`	
Connection type of the partition	`RM_PartitionConnection`	`rm_connection_type_t *`	Values: TORUS or MESH
Partition description	`RM_PartitionDescription`	`char **`	free required
Flag indicating whether this partition is a partition smaller than the base partition	`RM_PartitionSmall`	`int *`	0=No 1=Yes
Number of used processor sets (psets) per base partition	`RM_PartitionPsetsPerBP`	`int *`	
Job identifier of the current job	`RM_PartitionJobID`	`int *`	If no job is currently on the partition, 0 is returned; for HTC partitions it always returns 0 even when HTC jobs are running.
Partition owner	`RM_PartitionUserName`	`char **`	free required
Partition options	`RM_PartitionOptions`	`char **`	free required
File name of the machine loader image	`RM_PartitionMloaderImg`	`char **`	free required
Comma-separated list of images to load on the Compute Nodes	`RM_PartitionCnloadImg`	`char **`	free required
Comma-separated list of images to load on the I/O nodes	`RM_PartitionIoloadImg`	`char **`	free required
Number of base partitions in the partition	`RM_PartitionBPNum`	`int *`	
First base partition in the partition	`RM_PartitionFirstBP`	`rm_BP_t **`	
Next base partition in the partition	`RM_PartitionNextBP`	`rm_BP_t **`	
Number of switches in the partition	`RM_PartitionSwitchNum`	`int *`	
First switch in the partition	`RM_PartitionFirstSwitch`	`rm_switch_t **`	
Next switch in the partition	`RM_PartitionNextSwitch`	`rm_switch_t **`	
Number of node cards in the partition	`RM_PartitionNodeCardNum`	`int *`	
First node card in the partition	`RM_PartitionFirstNodeCard`	`rm_nodecard_t **`	
Next node card in the partition	`RM_PartitionNextNodeCard`	`rm_nodecard_t **`	
Number of users of the partition	`RM_PartitionUsersNum`	`int *`	
First user name for the partition	`RM_PartitionFirstUser`	`char **`	free required

Description	Specification	Argument type	Notes
Next user name for the partition	`RM_PartitionNextUser`	`char **`	free required
HTC pool identifier	`RM_PartitionHTCPoolID`	`pm_pool_id_t *`	Value will be NULL for a HPC partition. free required
Partition size in compute nodes	`RM_PartitionSize`	`int *`	
Boot options	`RM_PartitionBootOptions`	`char **`	free required

Table 13-18 shows the values that are set in a partition object using rm_set_data().

Table 13-18 Values set in a partition object using rm_set_data()

Description	Specification	Argument type	Notes
Partition identifier	`RM_PartitionID`	`pm_partition_id_t`	Up to 32 characters for a new partition ID, or up to 16 characters followed by an asterisk (*) for a prefix for a unique name
Connection type of the partition	`RM_PartitionConnection`	`rm_connection_type_t *`	Values: TORUS or MESH
Partition description	`RM_PartitionDescription`	`char *`	
Flag indicating whether this partition is a partition smaller than the base partition	`RM_PartitionSmall`	`int *`	0=No 1=Yes
Number of used processor sets (psets) per base partition	`RM_PartitionPsetsPerBP`	`int *`	
Partition owner	`RM_PartitionUserName`	`char *`	
File name of the machine loader image	`RM_PartitionMloaderImg`	`char *`	
Comma-separated list of images to load on the Compute Nodes	`RM_PartitionCnloadImg`	`char *`	
Comma-separated list of images to load on the I/O nodes	`RM_PartitionIoloadImg`	`char *`	
Number of base partitions in the partition	`RM_PartitionBPNum`	`int *`	
First base partition in the partition	`RM_PartitionFirstBP`	`rm_BP_t *`	
Next base partition in the partition	`RM_PartitionNextBP`	`rm_BP_t *`	
Number of switches in the partition	`RM_PartitionSwitchNum`	`int *`	
First switch in the list in the partition	`RM_PartitionFirstSwitch`	`rm_switch_t *`	

Description	Specification	Argument type	Notes
Next switch in the partition	`RM_PartitionNextSwitch`	`rm_switch_t *`	
Number of node cards in the partition	`RM_PartitionNodeCardNum`	`int *`	
First node card in the partition	`RM_PartitionFirstNodecard`	`rm_nodecard_t *`	
Next node card in the partition	`RM_PartitionNextNodecard`	`rm_nodecard_t *`	
Boot options	`RM_PartitionBootOptions`	`char *`	

Job list object

The job list object (`rm_job_list_t`) contains a list of job objects. The job list object is retrieved by calling the rm_get_jobs() API. See Table 13-19.

Table 13-19 Values retrieved from a job list object using rm_get_data()

Description	Specification	Argument type	Notes
Number of jobs in the list	`RM_JobListSize`	`int *`	
First job in the list	`RM_JobListFirstJob`	`rm_job_t **`	
Next job in the list	`RM_JobListNextJob`	`rm_job_t **`	

Job object

The job object (`rm_job_t`) represents a job defined in the Blue Gene system. The job object is retrieved from the job list object using the `RM_JobListFirstJob` and `RM_JobListNextJob` specifications. A new job object is created using the rm_new_job() API. After setting the appropriate fields in a new job object, the job can be added to the system using the rm_add_job() API. See Table 13-20.

Table 13-20 Values retrieved from a job object using rm_get_data()

Description	Specification	Argument type	Notes
Job identifier	`RM_JobID`	`rm_job_id_t *`	free required Identifier is unique across all jobs on the system.
Identifier of the partition assigned for the job	`RM_JobPartitionID`	`pm_partition_id_t *`	free required
Job state	`RM_JobState`	`rm_job_state_t *`	
Sequence ID for the job state	`RM_JobStateSeqID`	`rm_sequence_id_t *`	
Executable file name for the job	`RM_JobExecutable`	`char **`	free required
Name of the user who submitted the job	`RM_JobUserName`	`char **`	free required
Integer containing the ID given to the job by the database	`RM_JobDBJobID`	`db_job_id_t *`	
Job output file name	`RM_JobOutFile`	`char **`	free required
Job error file name	`RM_JobErrFile`	`char **`	free required

Description	Specification	Argument type	Notes
Job output directory name	RM_JobOutDir	char **	free required This directory contains the output files if a full path is not given.
Error text returned from the control daemons	RM_JobErrText	char **	free required
Arguments for the job executable	RM_JobArgs	char **	free required
Environment parameter needed for the job	RM_JobEnvs	char **	free required
Flag indicating whether the job was retrieved from the history table	RM_JobInHist	int *	0=No 1=Yes
Job mode	RM_JobMode	rm_job_mode_t *	Indicates virtual node, SMP, or dual mode
System call trace indicator for Compute Nodes	RM_JobStrace	rm_job_strace_t *	
Job start time The format is yyyy-mm-dd-hh.mm.ss.nnnnnn. If the job never goes to running state, it will be an empty string. Data is only valid for completed jobs. The rm_get_data() specification RM_JobInHist can be used to determine whether a job has completed. If the job is an active job, then the value returned is meaningless.	RM_JobStartTime	char **	free required
Job end time Format is yyyy-mm-dd-hh.mm.ss.nnnnnn. Data is valid only for completed jobs. The rm_get_data() specification RM_JobInHist can be used to determine whether a job has completed. If the job is an active job, the value returned is meaningless.	RM_JobEndTime	char **	free required
Job run time in seconds Data is only valid for completed jobs. The rm_get_data() specification RM_JobInHist can be used to determine whether a job has completed. If the job is an active job, the value returned is meaningless.	RM_JobRunTime	rm_job_runtime_t *	

Description	Specification	Argument type	Notes
Number of Compute Nodes used by the job Data is only valid for completed jobs. The rm_get_data() specification RM_JobInHist can be used to determine whether a job has completed. If the job is an active job, the value returned is meaningless.	RM_JobComputeNodesUsed	rm_job_computenodes_used_t *	
Job exit status Data is only valid for completed jobs. The rm_get_data() specification RM_JobInHist can be used to determine whether a job has completed. If the job is an active job, the value returned is meaningless.	RM_JobExitStatus	rm_job_exitstatus_t *	
User UID	RM_JobUserUid	rm_job_user_uid_t *	Zero is returned when querying existing jobs.
User GID	RM_JobUserGid	rm_job_user_gid_t *	Zero is returned when querying existing jobs.
Job location	RM_JobLocation	rm_job_location_t *	If NULL value, then job is HPC job. Non-NULL value indicates the location of the HTC job and is of the form Rxx-Mx-Nxx-Jxx-Cxx (where C-xx is the processor core). free required
Pool ID assigned for the job	RM_JobPooID	pm_pool_id_t *	If NULL value then job is HPC job. Non-NULL value indicates the partition pool that the job is assigned to. free required

Table 13-21 shows the values that are set in a job object using rm_set_data().

Table 13-21 Values set in a job object using rm_set_data()

Description	Specification	Argument type	Notes
Job identifier	RM_JobID	rm_job_id_t	This must be unique across all jobs on the system; if not, return code JOB_ALREADY_DEFINED is returned.
Partition identifier assigned for the job	RM_JobPartitionID	pm_partition_id_t	This field can be left blank when adding a new job to the system.
Executable file name for the job	RM_JobExecutable	char *	
Name of the user who submitted the job	RM_JobUserName	char *	

Description	Specification	Argument type	Notes
Job output file name	RM_JobOutFile	char *	
Job error file name	RM_JobErrFile	char *	
Job output directory	RM_JobOutDir	char *	This directory contains the output files if a full path is not given.
Arguments for the job executable	RM_JobArgs	char *	
Environment parameter needed for the job	RM_JobEnvs	char *	
Job mode	RM_JobMode	rm_job_mode_t *	Possible values: Virtual node, SMP, or dual mode.
System call trace indicator for Compute Nodes	RM_JobStrace	rm_job_strace_t *	
User UID	RM_JobUserUid	rm_job_user_uid_t *	This value can be set when adding a job.
User GID	RM_JobUserGid	rm_job_user_gid_t *	This value can be set when adding a job.

Job filter object

The job filter object (rm_job_filter_t) represents a filter for jobs defined in the Blue Gene/P system. The job filter object is passed as a parameter to the rm_get_filtered_jobs() API. The jobs returned match all of the specified filter fields. See Table 13-22.

Table 13-22 Job filter object description for rm_job_filter_t

Description	Specification	Argument Type	Notes
Job identifier	RM_JobFilterID	rm_job_id_t*	ID is unique across all jobs on the system. free required
Partition identifier assigned for the job	RM_JobFilterPartitionID	pm_partition_id_t *	Free required.
Job state	RM_JobFilterState	rm_job_state_t *	
Executable file name for the job	RM_JobFilterExecutable	char**	free required.
Name of the user who submitted the job	RM_JobFilterUserName	char**	free required.
Integer containing the ID given to the job by the database	RM_JobFilterDBJobID	db_job_id_t*	
Job output directory name	RM_JobFilterOutDir	char**	This directory contains the output files if a full path is not given. free required.
Job mode	RM_JobFilterMode	rm_job_mode_t*	Indicates virtual node, SMP, or dual mode.

Description	Specification	Argument Type	Notes
Job start time Format is yyyy-mm-dd-hh.mm.ss.nnnnn.	RM_JobFilterStartTime	char**	free required.
Job location	RM_JobFilterLocation	rm_job_location_t *	If NULL value, then job is HPC job. Non-NULL value indicates the location of the HTC job and is of the form Rxx-Mx-Nxx-Jxx-Cxx (where Cxx is the processor core). free required.
Pool ID assigned for the job	RM_JobFilterPoolID	pm_pool_id_t *	If NULL value, then job is HPC job. Non-NULL value indicates the partition pool that the job is assigned to. free required.
Job type	RM_JobFilterType	rm_job_state_flag_t	Flag to select HPC or HTC jobs only or both. The possible flags are contained in the rm_api.h include file.

Table 13-23 shows the job modes.

Table 13-23 Job modes

Mode	Value
RM_SMP_MODE	0x0000
RM_DUAL_MODE	0x0001
RM_VIRTUAL_NODE_MODE	0x0002

Table 13-24 shows Type modes.

Table 13-24 Job types

Type	Value
JOB_TYPE_HPC_FLAG	0x0001
JOB_TYPE_HTC_FLAG	0x0002
JOB_TYPE_ALL_FLAG	0x0003

13.2.9 Object allocator APIs

In this section, we describe the APIs used to allocate memory for objects used with other API calls:

- `status_t rm_new_BP(rm_BP_t **bp);`

  Allocates storage for a new base partition object.

- `status_t rm_new_ionode(rm_ionode_t **io);`

  Allocates storage for a new I/O node object.
- `status_t rm_new_job(rm_job_t **job);`

  Allocates storage for a new job object.
- `status_t rm_new_nodecard(rm_nodecard_t **nc);`

  Allocates storage for a new node card object.
- `status_t rm_new_partition(rm_partition_t **partition);`

  Allocates storage for a new partition object.
- `status_t rm_new_switch(rm_switch_t **switch);`

  Allocates storage for a new switch object.
- `status_t rm_new_job_filter(rm_job_filter_t **jobfilter);`

  Allocates storage for a new job filter object

13.2.10 Object deallocator APIs

In this section, we describe the APIs used to deallocate memory for objects that are created by other API calls:

- `status_t rm_free_BG(rm_BG_t *bg);`

  Frees storage for a Blue Gene object.
- `status_t rm_free_BP(rm_BP_t *bp);`

  Frees storage for a base partition object.
- `status_t rm_free_ionode(rm_ionode_t *io);`

  Frees storage for an I/O node object.
- `status_t rm_free_job(rm_job_t *job);`

  Frees storage for a job object.
- `status_t rm_free_job_list(rm_job_list_t *job_list);`

  Frees storage for a job list object.
- `status_t rm_free_nodecard(rm_nodecard_t *nc);`

  Frees storage for a node card object.
- `status_t rm_free_nodecard_list(rm_nodecard_list_t *nc_list);`

  Frees storage for a node card list object.
- `status_t rm_free_partition(rm_partition_t *partition);`

  Frees storage for a partition object.
- `status_t rm_free_partition_list(rm_partition_list_t *part_list);`

  Frees storage for a partition list object.
- `status_t rm_free_switch(rm_switch_t *switch);`

  Frees storage for a switch object.
- `status_t rm_free_job_filter(rm_job_filter_t *jobfilter);`

  Frees storage for a job filter object.

13.2.11 Messaging APIs

In this section, we describe the set of thread-safe messaging APIs. These APIs are used by the Bridge APIs as well as by other components of the job management system, such as the **`mpirun`** program that ships with the Blue Gene/P software. Each message is written using the following format:

```
<Timestamp> Component (Message type): Message text
```

Here is an example:

```
<Mar 9 04:24:30> BRIDGE (Debug): rm_get_BG()- Completed Successfully
```

The message can be one of the following types:

- MESSAGE_ERROR: Error messages
- MESSAGE_WARNING: Warning messages
- MESSAGE_INFO: Informational messages
- MESSAGE_DEBUG1: Basic debug messages
- MESSAGE_DEBUG2: More detailed debug messages
- MESSAGE_DEBUG3: Very detailed debug messages

The following verbosity levels, to which the messaging APIs can be configured, define the policy:

- Level 0: Only error or warning messages are issued.
- Level 1: Level 0 messages and informational messages are issued.
- Level 2: Level 1 messages and basic debug messages are issued.
- Level 3: Level 2 messages and more debug messages are issued.
- Level 4: The highest verbosity level. All messages that will be printed are issued.

By default, only error and warning messages are written. To have informational and minimal debug messages written, set the verbosity level to 2. To obtain more detailed debug messages, set the verbosity level to 3 or 4.

In the following list, we describe the messaging APIs:

- `int isSayMessageLevel(message_type_t m_type);`

  Tests the current messaging level. Returns 1 if the specified message type is included in the current messaging level; otherwise returns 0.

- `void closeSayMessageFile();`

  Closes the messaging log file.

  Note: Any messaging output after calling this method is sent to stderr.

- `int sayFormattedMessage(FILE * curr_stream, const void * buf, size_t bytes);`

  Logs a preformatted message to the messaging output without a time stamp.

- `void sayMessage(const char * component, message_type_t m_type, const char * curr_func, const char * format, ...);`

  Logs a message to the messaging output.

  The `format` parameter is a format string that specifies how subsequent arguments are converted for output. This value must be compatible with `printf` format string requirements.

- `int sayPlainMessage(FILE * curr_stream, const char * format, ...);`

  Logs a message to the messaging output without a time stamp.

The `format` parameter is a format string that specifies how subsequent arguments are converted for output. This value must be compatible with the `printf` format string requirements.

- `void setSayMessageFile(const char* oldfilename, const char* newfilename);`

  Opens a new file for message logging.

  > **Note:** This method can be used to atomically rotate log files.

- `void setSayMessageLevel(unsigned int level);`

  Sets the messaging verbose level.

- `void setSayMessageParams(FILE * stream, unsigned int level);`

  Uses the provided file for message logging and sets the logging level.

  > **Note:** This method has been deprecated in favor of the setSayMessageFile() and setSayMessageLevel() methods.

13.3 Small partition allocation

The base allocation unit in the Blue Gene/P system is a base partition. Partitions are composed of whole numbers of base partitions, except in two special cases concerning small partitions. A *small partition* is a partition that is comprised of a fraction of a base partition. Small partitions can be created in the following sizes:

- 16 Compute Nodes

  A 16-node partition is comprised of 16 Compute Nodes from a single node card. The node card must have two installed I/O nodes in order to be used for a 16-node partition.

- 32 Compute Nodes

  A 32-node partition is comprised of all the Compute Nodes in a single node card. The node card must have at least one installed I/O node in order to be used for a 32-node partition.

- 64 Compute Nodes

  A 64-node partition is comprised of two adjacent node cards beginning with N00, N02, N04, N06, N08, N10, N12, or N14. The first node card in the pair must have at least one installed I/O node in order to be used for a 64-node partition.

- 128 Compute Nodes

  A 128-node partition is comprised of set of four adjacent node cards beginning with N00, N04, N08, or N12. The first node card in the set must have at least one installed I/O node in order to be used for a 128-node partition.

- 256 Compute Nodes

  A 256-node partition is comprised of a set of eight adjacent node cards beginning with N00 or N08. The first node card in the set must have at least one installed I/O node in order to be used for a 256-node partition.

13.3.1 Subdivided busy base partitions

It is important that you understand the concept of *subdivided busy base partitions* when working with small partitions. A base partition is considered subdivided busy if at least one partition, defined for a subset of its node cards, is busy. A partition is busy if its state is not free (RM_PARTITION_FREE).

A base partition that is subdivided busy cannot be booted as a whole because some of its hardware is unavailable. A base partition can have small partitions and full midplane partitions (multiples of 512 Compute Nodes) defined for it in the database. If the base partition has small partitions defined, they do not have to be in use, and a full midplane partition can use the actual midplane. In this case, the partition name that is using the base partition is returned on the RM_BPPartID specification.

For small partitions, multiple partitions can use the same base partition. This is the subdivided busy (SDB) example. In this situation, the value returned for the RM_BPPartID specification is meaningless. You must use the RM_BPSDB specification to determine whether the base partition is subdivided busy (small partition in use).

13.4 API examples

In this section, we provide example API calls for several common situations.

13.4.1 Retrieving base partition information

The code in Example 13-2 retrieves the Blue Gene/P hardware information and prints some information about each base partition in the system.

Example 13-2 Retrieving base partition information

```
#include "rm_api.h"
int main(int argc, char *argv[]) {
    status_t rmrc;
    rm_BG_t *rmbg;
    int bpNum;
    enum rm_specification getOption;
    rm_BP_t *rmbp;
    rm_bp_id_t bpid;
    rm_BP_state_t state;
    rm_location_t loc;
    rmrc = rm_set_serial("BGP");
    rmrc = rm_get_BG(&rmbg);
    if (rmrc) {
        printf("Error occured calling rm_get_BG: %d\n", rmrc);
        return -1;
    }
    rm_get_data(rmbg, RM_BPNum, &bpNum);
    printf("Number of base partitions: %d\n", bpNum);
    getOption = RM_FirstBP;
    for (int ii = 0; ii < bpNum; ++ii) {
        rm_get_data(rmbg, getOption, &rmbp);
        rm_get_data(rmbp, RM_BPID, &bpid);
        rm_get_data(rmbp, RM_BPState, &state);
        rm_get_data(rmbp, RM_BPLoc, &loc);
        printf("    BP %s with state %d at location <%d,%d,%d>\n", bpid, state, loc.X,
loc.Y, loc.Z);
        free(bpid);
        getOption = RM_NextBP;
    }
    rm_free_BG(rmbg);  // Deallocate memory from rm_get_BG()
}
```

The example code can be compiled and linked with the commands shown in Figure 13-6.

```
g++ -m64 -pthread -I/bgsys/drivers/ppcfloor/include -c sample1.cc -o sample1.o_64

g++ -m64 -pthread -o sample1 sample1.o_64 -L/bgsys/drivers/ppcfloor/lib64 -lbgpbridge
```

Figure 13-6 Example compile and link commands

13.4.2 Retrieving node card information

The code in Example 13-3 shows how to retrieve information about the node cards for a base partition. The rm_get_nodecards() function retrieves a list of all the node cards in a base partition. The list always contains exactly 16 node cards.

Example 13-3 Retrieving node card information

```
int getNodeCards(rm_bp_id_t bpid) {
    int rmrc;
    rm_nodecard_list_t *ncList;
    int ncNum;
    enum rm_specification getOption;
    rm_nodecard_t *rmnc;
    rm_nodecard_id_t ncid;
    rm_nodecard_state_t ncState;
    int ioNum;
    rmrc = rm_get_nodecards(bpid, &ncList);
    if (rmrc) {
        printf("Error occured calling rm_get_nodecards: %d\n", rmrc);
        return -1;
    }
    rmrc = rm_get_data(ncList, RM_NodeCardListSize, &ncNum);
    printf("    Base partition %s has %d nodecards\n", bpid, ncNum);
    getOption = RM_NodeCardListFirst;
    for (int ii = 0; ii < ncNum; ++ii) {
        rmrc = rm_get_data(ncList, getOption, &rmnc);
        rmrc = rm_get_data(rmnc, RM_NodeCardID, &ncid);
        rmrc = rm_get_data(rmnc, RM_NodeCardState, &ncState);
        rmrc = rm_get_data(rmnc, RM_NodeCardIONodes, &ioNum);
        printf("        Node card %s with state %d has %d I/O nodes\n", ncid, ncState,
ioNum);
        free(ncid);

        getOption = RM_NodeCardListNext;
    }
    rm_free_nodecard_list(ncList);
}
```

13.4.3 Defining a new small partition

Example 13-4 contains pseudo code that shows how to allocate a new small partition.

Example 13-4 Allocating a new small partition

```
int isSmall = 1;

rm_new_partition(&newpart); //Allocate space for new partition

// Set the descriptive fields
rm_set_data(newpart,RM_PartitionUserName, username);
rm_set_data(newpart,RM_PartitionMloaderImg, BGP_MLOADER_IMAGE);
rm_set_data(newpart,RM_PartitionCnloadImg, BGP_CNLOAD_IMAGE);
rm_set_data(newpart,RM_PartitionIoloadImg, BGP_IOLOAD_IMAGE);
rm_set_data(newpart,RM_PartitionSmall, &isSmall); // Mark partition as a small partition

// Add a single BP
rm_new_BP(rm_BP_t **BP);
rm_set_data(BP, RM_BPID, “R01-M0”);
rm_set_data(newpart, RM_PartitionFirstBP, BP);

// Add the node card(s) comprising the partition
ncNum = 4; // The number of node cards is 4 for 128 compute nodes
rm_set_data(newpart, RM_PartitionNodeCardNum, &ncNum); // Set the number of node cards
for (1 to ncNum) {
   // all four node cards must belong to same quarter!
   rm_new_nodecard(rm_nodecard_t **nc); // Allocate space for new node card
   rm_set_data(nc, RM_NodeCardID, ncid);
   rm_set_data(newpart, RM_PartitionFirstNodeCard, nc); // Add the node card to the
partition
            or
   rm_set_data(newpart, RM_PartitionNextNodeCard, nc);
   rm_free_nodecard(nc);
}

rm_add_partition(newpart);
```

13.4.4 Querying a small partition

Example 13-5 contains pseudo code that shows how to query a small partition for its node cards.

Example 13-5 Querying a small partition

```
rm_get_partition(part_id, &mypart); // Get the partition
rm_get_data(mypart, RM_PartitionSmall, &small); // Check if this is a “small” partition
if (small) {
   rm_get_data(mypart,RM_PartitionFirstBP, &BP); // Get the First (and only) BP
   rm_get_data(mypart,RM_PartitionNodeCardNum, &nc_num); // Get the number of node cards

   for (1 to nc_num) {
      rm_get_data(mypart, RM_PartitionFirstNodeCard, &nc);
         or
      rm_get_data(mypart, RM_PartitionNextNodeCard, &nc);

      rm_get_data(nc, RM_NodeCardID, &ncid); // Get the id
      rm_get_data(nc, RM_NodeCardQuarter, &quarter); // Get the quarter
      rm_get_data(nc, RM_NodeCardState, &state); // Get the state
      rm_get_data(nc, RM_NodeCardIONodes, &ionodes); // Get num of I/O nodes
      rm_get_data(nc, RM_NodeCardPartID, &partid); // Get the partition ID
      rm_get_data(nc, RM_NodeCardPartState, &partstate); // Get the partition state

      print node card information
   }
}
```

14

Real-time Notification APIs

With the Blue Gene/P system, two programming models can handle state transitions for jobs, blocks, base partitions, switches, wires, and node cards. The first model is based on a polling model, where the caller of the Bridge APIs is responsible for the continuous polling of state information. The second model consists of Real-time Notification APIs that allow callers to register for state transition event notifications.

The Real-time Notification APIs are designed to eliminate the need for a resource management system to constantly have to read in all of the machine state to detect changes. The APIs enable the caller to be notified in real time of state changes to jobs, blocks, and hardware, such as base partitions, switches, wires, and node cards. After a resource management application obtains an initial snapshot of the machine state using the Bridge APIs, the resource management application can then be notified only of changes using the Real-time Notification APIs.

In this chapter, we describe the Real-time Notification APIs for the Blue Gene/P system that a resource management application can use. We discuss the following specific topics:

- API support
- Real-time Notification APIs
- Real-time callback functions
- Real-time elements
- Server-side event filtering
- Real-time Notification API status codes
- Sample real-time application code

14.1 API support overview

In the following sections, we present an overview of the support that the APIs provide.

14.1.1 Requirements

The requirements for writing programs to the Real-time Notification APIs are as follows:

- Currently, SUSE Linux Enterprise Server (SLES) 10 for PowerPC is the only supported platform. The application must run on the IBM Blue Gene service node.
- When the application calls rt_init(), the API looks for the DB_PROPERTY environment variable. The corresponding db.properties file indicates the port on which the real-time server is listening and that the real-time client uses to connect to the server. The environment variable should be set to point to the actual db.properties file location as follows:
  - On a bash shell

    ```
    export DB_PROPERTY=/bgsys/drivers/ppcfloor/bin/db.properties
    ```

  - On a csh shell

    ```
    setenv DB_PROPERTY /bgsys/drivers/ppcfloor/bin/db.properties
    ```

- C and C++ are supported with the GNU gcc V4.1.2-level compilers. For more information and downloads, refer to the following Web address:

  http://gcc.gnu.org/

- The include file is /bgsys/drivers/ppcfloor/include/rt_api.h.
- Only 64-bit shared library support is provided. Link your real-time application with the file /bgsys/drivers/ppcfloor/lib64/libbgrealtime.so.

  Both the include and shared library files are installed as part of the standard system installation. They are contained in the bgpbase.rpm file.

Example 14-1 shows a possible excerpt from a makefile that you can create to help automate builds of your application. This sample is shipped in the directory /bgsys/drivers/ppcfoor/doc/realtime/sample/simple/Makefile. In this makefile, the program that is being built is `rt_sample_app`, and the source is in the rt_sample_app.cc file.

Example 14-1 Makefile excerpt

```
ALL_APPS = rt_sample_app

CXXFLAGS += -w -Wall -g -m64 -pthread
CXXFLAGS += -I/bgsys/drivers/ppcfloor/include

LDFLAGS += -L/bgsys/drivers/ppcfloor/lib64 -lbgrealtime
LDFLAGS += -pthread

.PHONY: all clean default distclean

default: $(ALL_APPS)

all: $(ALL_APPS)

clean:
   $(RM) $(ALL_APPS) *.o
```

```
distclean: clean

...
```

14.1.2 General comments

The real-time APIs have general considerations that apply to all or most calls. We highlight the following common features:

- All the API calls return an `rt_status_t`, which indicates either success or a status code. An exit code of 0 (RT_STATUS_OK) indicates that no problems were encountered. Positive status codes indicate that an expected non-error condition was encountered, whereas negative status codes indicate an error occurred.
- Most of the API calls take a pointer to a real-time handle (`rt_handle_t`), which is an opaque structure that represents a stream of real-time messages.
- The real-time APIs use `sayMessage` APIs for printing debug and error messages. The application should initialize the `sayMessage` APIs before calling the real-time APIs.

Blocking mode versus nonblocking mode

A real-time handle can be in *blocking* or *nonblocking* mode. In blocking mode, rt_request_realtime() and rt_set_server_filter() block until it can send the request, and rt_read_msgs() blocks until there is an event to receive. In nonblocking mode, rt_request_realtime() and rt_set_server_filter() return `RT_WOULD_BLOCK` if the request cannot be sent. If your application gets this return code from rt_request_realtime() or rt_set_server_filter(), you must call the function again until it returns `RT_FINISHED_PREV`. In nonblocking mode, rt_read_msgs() returns `RT_NO_REALTIME_MSGS` immediately if no real-time event is ready to be processed.

rt_get_socket_descriptor() can be used to get a file descriptor that can be used with a select()-type system API to wait for a real-time event to be available to rt_read_msgs() when a handle is in nonblocking mode.

The initial blocking or nonblocking mode is set using rt_init(). An initialized handle can be set to blocking mode by using rt_set_blocking() or set to nonblocking mode by using rt_set_nonblocking().

Filtering events

Prior to IBM Blue Gene/P release V1R3M0, filtering of real-time events was performed only on the client. With V1R3M0, filtering of real-time events can be done by the server, which is more efficient because the messages are sent only if the client wants to receive them. For more information about server-side filtering, refer to 14.5, “Server-side filtering” on page 272.

A real-time handle can be configured so that only partition events that affect certain partitions, job events, or both, are passed to the application.

Setting the client-side partition filter is done by using the rt_set_filter() API with `RT_PARTITION` as the filter_type parameter. The filter_names parameter can specify one or more partition IDs separated by spaces. When rt_get_msgs() is called, partition events are delivered only to the application if the partition ID matches any of the partition IDs in the filter. If the filter_names parameter is set to NULL, the partition filter is removed, and all partition events are delivered to the application. An example of the value to use for the filter_names parameter for partition IDs R00-M0 and R00-M1 is “R00-M0 R00-M1”.

You can set the client-side job filter by using the rt_set_filter() API with `RT_JOB` as the filter_type parameter. The filter_names parameter can specify one or more job IDs (as strings) separated by spaces. When the rt_get_msgs() API is called, job events are delivered only to the application if the job ID matches any of the job IDs in the filter. If the filter_names parameter is set to NULL, the job filter is removed, and all job events are delivered to the application. An example of the value to use for the filter_names parameter for job IDs 10030 and 10031 is “10030 10031”.

The other use of the rt_set_filter() API is to remove both types of filter by passing `RT_CLEAR_ALL` in the filter_type parameter.

14.2 Real-time Notification APIs

In this section, we describe the Real-time Notification APIs:

- `rt_status_t rt_init(rt_handle_t **handle_out, rt_block_flag_t blocking_flag, rt_callbacks_t* callbacks);`

  Initializes a real-time handle. This function gets the port of the real-time server from the db.properties file. The name of the db.properties file must be in the DB_PROPERTY environment variable, or RT_DB_PROPERTY_ERROR is returned.

  If this function is successful, `*handle_out` is set to a valid handle connected to the real-time server. The blocking state for the handle is set based on the blocking flag parameter. The callbacks for the handle are set to the callbacks parameter. If this function is not successful and `handle_out` is not NULL, then `*handle_out` is set to NULL:

- `rt_status_t rt_close(rt_handle_t **handle);`

  Closes a real-time handle. The handle must not be used after calling this function.

- `rt_status_t rt_set_blocking(rt_handle_t **handle);`

  Sets a real-time handle to blocking mode.

- `rt_status_t rt_set_nonblocking(rt_handle_t **handle);`

  Sets a real-time handle to nonblocking mode.

- `rt_status_t rt_create_server_filter(rt_filter_t **filter_out);`

  Creates a server-side filter object.

- `rt_status_t rt_server_filter_set_property(rt_filter_t *filter, rt_server_filter_property_t filter_property, void *property_value);`

  Sets a property of the server-side filter object.

- `rt_status_t rt_set_server_filter(rt_handle_t **handle, const rt_filter_t *filter, rt_filter_id_t *filter_id_out);`

  Assigns a server-side filter to a real-time handle.

- `rt_status_t rt_free_server_filter(rt_filter_t **filter_in_out);`

  Frees a server-side filter object.

- `rt_status_t rt_set_filter(rt_handle_t **handle, rt_filter_type_t filter_type, const char* filter_names);`

  Sets the client-side filter on a real-time handle. The filter names consist of a C-style string that contains a space-separated list of names to filter on. If removing filter entries, set filter_names to NULL. For filtering on partition names, consider this example of “R01-M0 R02-M1 R03”.

- `rt_status_t rt_request_realtime(rt_handle_t **handle);`

  Requests real-time events for this handle. If this function returns RT_WOULD_BLOCK, the request has not been sent. Call this function again until it returns `RT_FINISHED_PREV`, which indicates that the previous request has been sent.

  If this function returns `RT_FINISHED_PREV`, a new request was not sent.

- `rt_status_t rt_get_socket_descriptor(rt_handle_t **handle, int *sd_out);`

  Gets the socket descriptor used by the real-time APIs. You can use this socket descriptor with the select() or poll() Linux APIs to wait until a real-time message is ready to be read. Other file or socket descriptor APIs, such as close(), should not be used on the socket descriptor returned by this API.

- `rt_status_t rt_read_msgs(rt_handle_t **handle, void *data);`

  Receives real-time events on a handle. If the handle is blocking, this function blocks as long as no events are waiting. If the handle is nonblocking, the function returns immediately with `RT_NO_REALTIME_MSGS` if no events are waiting. If an event is waiting to be processed, the callback associated with the event type is called. If the callback returns `RT_CALLBACK_CONTINUE`, events continue to be processed.

- `rt_status_t rt_get_data(rt_element_t *elem, rt_specification_t field, void *data_out);`

  Gets data from a real-time element.

- `rt_status_t rt_dup_element(rt_element_t *elem, rt_element_t **elem_out);`

  Copies a real-time element.

- `rt_status_t rt_free_element(rt_element_t *elem);`

  Frees a real-time element.

- `const char* rt_convert_ras_severity_to_string(rt_ras_severity_t severity);`

  Gets a printable string for a RAS severity value.

14.3 Real-time callback functions

Developers who use the Real-time Notification APIs must write functions that are called when real-time events are received. These functions are callback functions because the application calls the rt_read_msgs() API, which then calls the function supplied by the application.

Pointers to the callback functions must be set in an `rt_callbacks_t` structure. When a real-time event is received, the corresponding function is called using that pointer. The application passes its `rt_callbacks_t` into rt_init(), which is stored for use when rt_read_msgs() is called. If the pointer to the callback function in the `rt_callbacks_t` structure is NULL, the event is discarded.

In addition to setting the callback functions in the `rt_callbacks_t` structure, the application must also set the version field to `RT_CALLBACK_VERSION_2`. With a later version of the real-time APIs, we can provide different callbacks and a different version for this field. The application can use the RT_CALLBACK_VERSION_CURRENT macro, which is the current version when the application is compiled.

From inside your callback function, you cannot call a real-time API using the same handle on which the event occurred; otherwise, your application deadlocks.

The return type of the callback functions is an indicator of whether rt_read_msgs() continues to attempt to receive another real-time event on the handle or whether it stops. If the callback function returns RT_CALLBACK_CONTINUE, rt_read_msgs() continues to attempt to receive real-time events. If the callback function returns RT_CALLBACK_QUIT, rt_read_msgs() does not attempt to receive another real-time event but returns RT_STATUS_OK.

Sequence identifiers (IDs) are associated with the state of each partition, job, base partition, node card, wire, and switch. A state with a higher sequence ID is newer. If your application gets the state for an object from the Bridge APIs in addition to the real-time APIs, you must discard any state that has a lower sequence ID for the same object.

These APIs provide the *raw state* for partitions, jobs, base partitions, node cards, wires and switches in addition to providing the state. The raw state is the status value that is stored in the Blue Gene/P database as a single character, rather than the state enumeration that the Bridge APIs use. Several raw state values map to a single state value so your application might receive real-time event notifications where the state does not change but the raw state does, for example, the partition raw states of “A” (allocating), “C” (configuring), and “B” (booting) all map to the Bridge enumerated state of RM_PARTITION_CONFIGURING.

Real-time callback structure

In this section, we describe each of the callbacks available to applications in the rt_callbacks_t structure. We list each field of the structure along with the following information:

- The description of the event that causes the callback to be invoked
- The signature of the callback function

  Your function must match the signature. Otherwise your program fails to compile.
- A description of each argument to the callback function

Field end_cb

The end_cb callback function is called when a real-time ended event occurs. Your application does not receive any more real-time events on this handle until you request real-time events from the server again by calling the rt_request_realtime API.

The function uses the following signature:

```
cb_ret_t my_rt_end(rt_handle_t **handle, void *extended_args, void *data);
```

Table 14-1 lists the arguments to the end_cb callback function.

Table 14-1 Field end_cb

Argument	Description
handle	Real-time handle on which the event occurred
extended_args	Not used; is NULL for now
data	Application data forwarded by rt_read_msgs()

Field partition_added_cb

The `partition_added_cb` function is called when a partition added event occurs.

The function uses the following signature:

```
cb_ret_t my_rt_partition_added(
     rt_handle_t **handle,
     rm_sequence_id_t seq_id,
     pm_partition_id_t partition_id,
     rm_partition_state_t partition_new_state,
     rt_raw_state_t partition_raw_new_state,
     void *extended_args,
     void *data);
```

Table 14-2 lists the arguments to the `partition_added_cb` function.

Table 14-2 Field partition_added_cb

Argument	Description
handle	Real-time handle on which the event occurred
seq_id	Sequence ID for this partition’s state
partition_id	The partition’s ID
partition_new_state	The partition’s new state
partition_raw_new_state	The partition’s new raw state
extended_args	Not used; NULL for now
data	Application data forwarded by rt_read_msgs()

Field partition_state_changed_cb

The `partition_state_changed_cb` function is called when a partition state changed event occurs.

The function uses the following signature:

```
cb_ret_t my_rt_partition_state_changed(
     rt_handle_t **handle,
     rm_sequence_id_t seq_id,
     rm_sequence_id_t previous_seq_id,
     pm_partition_id_t partition_id,
     rm_partition_state_t partition_new_state,
     rm_partition_state_t partition_old_state,
     rt_raw_state_t partition_raw_new_state,
     rt_raw_state_t partition_raw_old_state,
     void *extended_args,
     void *data);
```

Table 14-3 lists the arguments to the partition_state_changed_cb function.

Table 14-3 Field partition_state_changed_cb

Argument	Description
handle	Real-time handle on which the event occurred
seq_id	Sequence ID for this partition's new state
previous_seq_id	Sequence ID for this partition's old state
partition_id	The partition's ID
partition_new_state	The partition's new state
partition_old_state	The partition's old state
partition_raw_new_state	The partition's new raw state
partition_raw_old_state	The partition's old raw state
extended_args	Not used; NULL for now
data	Application data forwarded by rt_read_msgs()

Field partition_deleted_cb

The partition_deleted_cb function is called when a partition deleted event occurs.

The function uses the following signature:

```
cb_ret_t my_rt_partition_deleted(
     rt_handle_t **handle,
     rm_sequence_id_t previous_seq_id,
     pm_partition_id_t partition_id,
     void *extended_args,
     void *data);
```

Table 14-4 lists the arguments to the partition_deleted_cb function.

Table 14-4 Field partition_deleted_cb

Argument	Description
handle	Real-time handle on which the event occurred
previous_seq_id	Sequence ID for this partition's state when removed
partition_id	The partition's ID
extended_args	Not used; NULL for now
data	Application data forwarded by rt_read_msgs()

Field job_added_cb

The job_added_cb function is called when a job added event occurs.

Note that this function is not called if the version field is RT_CALLBACK_VERSION1 and the job_added_v1_cb field is not NULL. The job_added_v1_cb callback provides more information.

The function uses the following signature:

```
cb_ret_t my_rt_job_added(
     rt_handle_t **handle,
     rm_sequence_id_t seq_id,
     db_job_id_t job_id,
     pm_partition_id_t partition_id,
     rm_job_state_t job_new_state,
     rt_raw_state_t job_raw_new_state,
     void *extended_args,
     void *data);
```

Table 14-5 lists the arguments to the `job_added_cb` function.

Table 14-5 Field job_added_cb

Argument	Description
handle	Real-time handle on which the event occurred
seq_id	Sequence ID for the job's state
job_id	The new job's ID
partition_id	ID of the partition to which the job is assigned
job_new_state	The job's new state
job_raw_new_state	The job's new raw state
extended_args	Not used; NULL for now
data	Application data forwarded by rt_read_msgs()

Field job_state_changed_cb

The `job_state_changed_cb` function is called when a job state changed event occurs.

Note that this function is not called if the `version` field is `RT_CALLBACK_VERSION1` and the `job_state_changed_v1_cb` field is not NULL. The `job_state_changed_v1_cb` callback provides more information.

The function uses the following signature:

```
cb_ret_t my_rt_job_state_changed(
     rt_handle_t **handle,
     rm_sequence_id_t seq_id,
     rm_sequence_id_t previous_seq_id,
     db_job_id_t job_id,
     pm_partition_id_t partition_id,
     rm_job_state_t job_new_state,
     rm_job_state_t job_old_state,
     rt_raw_state_t job_raw_new_state,
     rt_raw_state_t job_raw_old_state,
     void *extended_args,
     void *data);
```

Table 14-6 lists the arguments to the job_state_changed_cb function.

Table 14-6 Field job_state_changed_cb

Argument	Description
handle	Real-time handle on which the event occurred
seq_id	Sequence ID for the job's new state
previous_seq_id	Sequence ID of the job's previous state
job_id	The job's ID
partition_id	ID of the partition to which the job is assigned
job_new_state	The job's new state
job_old_state	The job's old state
job_raw_new_state	The job's new raw state
job_raw_old_state	The job's old raw state
extended_args	Not used; NULL for now
data	Application data forwarded by rt_read_msgs()

Field job_deleted_cb

The job_deleted_cb function is called when a job-deleted event occurs.

Note that this function is not called if the version field is RT_CALLBACK_VERSION1 and the job_deleted_v1_cb field is not NULL. The job_deleted_v1_cb callback provides more information.

The function uses the following signature:

```
cb_ret_t my_rt_job_deleted(
     rt_handle_t **handle,
     rm_sequence_id_t previous_seq_id,
     db_job_id_t job_id,
     pm_partition_id_t partition_id,
     void *extended_args,
     void *data);
```

Table 14-7 lists the arguments to the job_deleted_cb function.

Table 14-7 Field job_deleted_cb

Argument	Description
handle	Real-time handle on which the event occurred
previous_seq_id	Sequence ID of the job's previous state
job_id	Deleted job's ID
partition_id	ID of the partition to which the job was assigned
extended_args	Not used; NULL for now
data	Application data forwarded by rt_read_msgs()

Field bp_state_changed_cb

The `bp_state_changed_cb` is called when a base partition state changed event occurs.

The function uses the following signature:

```
cb_ret_t my_rt_BP_state_changed_fn(
     rt_handle_t **handle,
     rm_sequence_id_t seq_id,
     rm_sequence_id_t previous_seq_id,
     rm_bp_id_t bp_id,
     rm_BP_state_t BP_new_state,
     rm_BP_state_t BP_old_state,
     rt_raw_state_t BP_raw_new_state,
     rt_raw_state_t BP_raw_old_state,
     void *extended_args,
     void *data);
```

Table 14-8 lists the arguments to the `bp_state_changed_cb` function.

Table 14-8 Field bp_state_changed_cb

Argument	Description
handle	Real-time handle on which the event occurred
seq_id	Sequence ID of the base partition's new state
previous_seq_id	Sequence ID of the base partition's previous state
bp_id	The base partition's ID
BP_new_state	The base partition's new state
BP_old_state	The base partition's old state
BP_raw_new_state	The base partition's new raw state
BP_raw_old_state	The base partition's old raw state
extended_args	Not used; NULL for now
data	Application data forwarded by rt_read_msgs()

Field switch_state_changed_cb

The `switch_state_changed_cb` is called when a switch state changed event occurs.

The function uses the following signature:

```
cb_ret_t my_rt_switch_state_changed(
     rt_handle_t **handle,
     rm_sequence_id_t seq_id,
     rm_sequence_id_t previous_seq_id,
     rm_switch_id_t switch_id,
     rm_bp_id_t bp_id,
     rm_switch_state_t switch_new_state,
     rm_switch_state_t switch_old_state,
     rt_raw_state_t switch_raw_new_state,
     rt_raw_state_t switch_raw_old_state,
     void *extended_args,
     void *data);
```

Table 14-9 lists the arguments to the `switch_state_changed_cb` function.

Table 14-9 Field switch_state_changed_cb

Argument	Description
handle	Real-time handle on which the event occurred
seq_id	Sequence ID for the switch’s new state
previous_seq_id	Sequence ID of the switch’s previous state
switch_id	The switch’s ID
bp_id	The switch’s base partition's ID
switch_new_state	The switch’s new state
switch_old_state	The switch’s old state
switch_raw_new_state	The switch’s new raw state
switch_raw_old_state	The switch’s old raw state
extended_args	Not used; NULL for now
data	Application data forwarded by rt_read_msgs()

Field nodecard_state_changed_cb

The `nodecard_state_changed_cb` is called when a node card state changed event occurs.

The function uses the following signature:

```
cb_ret_t my_rt_nodecard_state_changed(
     rt_handle_t **handle,
     rm_sequence_id_t seq_id,
     rm_sequence_id_t previous_seq_id,
     rm_nodecard_id_t nodecard_id,
     rm_bp_id_t bp_id,
     rm_nodecard_state_t nodecard_new_state,
     rm_nodecard_state_t nodecard_old_state,
     rt_raw_state_t nodecard_raw_new_state,
     rt_raw_state_t nodecard_raw_old_state,
     void *extended_args,
     void *data);
```

Table 14-10 lists the arguments to the `nodecard_state_changed_cb` function.

Table 14-10 Field nodecard_state_changed_cb

Argument	Description
handle	Real-time handle on which the event occurred
seq_id	Sequence ID for the node card's new state
previous_seq_id	Sequence ID of the node card's previous state
nodecard_id	The node card's ID
bp_id	The node card's base partition's ID
nodecard_new_state	The node card's new state
nodecard_old_state	The node card's old state
nodecard_raw_new_state	The node card's new raw state
nodecard_raw_old_state	The node card's old raw state
extended_args	Not used; NULL for now
data	Application data forwarded by rt_read_msgs()

Field job_added_v1_cb

The `job_added_v1_cb` function is called when a job added event occurs.

Note that this function is called only if the `version` field is `RT_CALLBACK_VERSION_1` or later.

The function uses the following signature:

```
cb_ret_t my_rt_job_added(
     rt_handle_t **handle,
     rm_sequence_id_t previous_seq_id,
     jm_job_id_t job_id,
     db_job_id_t db_job_id,
     pm_partition_id_t partition_id,
     rm_job_state_t job_new_state,
     rt_raw_state_t job_raw_new_state,
     void *extended_args,
     void *data);
```

Table 14-11 lists the arguments to the `job_added_v1_cb` function.

Table 14-11 Field job_added_vi_cb

Argument	Description
handle	Real-time handle on which the event occurred
seq_id	Sequence ID for the job's state
job_id	Job identifier
db_job_id	Integer containing the ID given to the job by the database

partition_id	ID of the partition to which the job is assigned
job_new_state	The job's new state
job_raw_new_state	The job's new raw state
extended_args	Not used; NULL for now
data	Application data forwarded by rt_read_msgs()

Field job_state_changed_v1_cb

The `job_state_changed_v1_cb` function is called when a job state changed event occurs.

Note that this function is called only if the `version` field is `RT_CALLBACK_VERSION_1` or later.

The function uses the following signature:

```
cb_ret_t my_rt_job_state_changed(
     rt_handle_t **handle,
     rm_sequence_id_t seq_id,
     rm_sequence_id_t previous_seq_id,
     jm_job_id_t job_id,
     db_job_id_t db_job_id,
     pm_partition_id_t partition_id,
     rm_job_state_t job_new_state,
     rm_job_state_t job_old_state,
     rt_raw_state_t job_raw_new_state,
     rt_raw_state_t job_raw_old_state,
     void *extended_args,
     void *data);
```

Table 14-12 lists the arguments to the `job_state_changed_v1_cb` function.

Table 14-12 Field job_state_changed_v1_cb

Argument	Description
handle	Real-time handle on which the event occurred
seq_id	Sequence ID for the job's new state
previous_seq_id	Sequence ID of the job's previous state
job_id	Job identifier
db_job_id	Integer containing the ID given to the job by the database
partition_id	ID of the partition to which the job is assigned
job_new_state	The job's new state
job_old_state	The job's old state
job_raw_new_state	The job's new raw state
job_raw_old_state	The job's old raw state
extended_args	Not used; NULL for now
data	Application data forwarded by rt_read_msgs()

Field job_deleted_v1_cb

The `job_deleted_v1_cb` function is called when a job-deleted event occurs.

Note that this function is called only if the `version` field is `RT_CALLBACK_VERSION_1` or later.

The function uses the following signature:

```
cb_ret_t my_rt_job_deleted(
     rt_handle_t **handle,
     rm_sequence_id_t previous_seq_id,
     jm_job_id_t job_id,
     db_job_id_t db_job_id,
     pm_partition_id_t partition_id,
     void *extended_args,
     void *data);
```

Table 14-13 lists the arguments to the `job_deleted_v1_cb` function.

Table 14-13 Field job_deleted_v1_cb

Argument	Description
handle	Real-time handle on which the event occurred
previous_seq_id	Sequence ID of the job's previous state
job_id	Job identifier
db_job_id	Integer containing the ID given to the job by the database
partition_id	ID of the partition to which the job was assigned
extended_args	Not used; NULL for now
data	Application data forwarded by rt_read_msgs()

Field wire_state_changed_cb

The `wire_state_changed_cb` function is called when a wire state changed event occurs.

Note that this function is called only if the `version` field is `RT_CALLBACK_VERSION_1` or later.

The function uses the following signature:

```
cb_ret_t my_rt_wire_state_changed(
     rt_handle_t **handle,
     rm_sequence_id_t seq_id,
     rm_sequence_id_t previous_seq_id,
     rm_wire_id_t wire_id,
     rm_wire_state_t wire_new_state,
     rm_wire_state_t wire_old_state,
     rt_raw_state_t wire_raw_new_state,
     rt_raw_state_t wire_raw_old_state,
     void *extended_args,
     void *data);
```

Table 14-14 lists the arguments to the wire_state_changed_cb function.

Table 14-14 Field wire_state_changed_cb

Argument	Description
handle	Real-time handle on which the event occurred
seq_id	Sequence ID of the wire's new state
previous_seq_id	Sequence ID of the wire's previous state
wire_id	Wire identifier
wire_new_state	The wire's new state
wire_old_state	The wire's old state
wire_raw_new_state	The wire's new raw state
wire_raw_old_state	The wire's old raw state
extended_args	Not used; NULL for now
data	Application data forwarded by rt_read_msgs()

Field filter_acknowledge_cb

The filter_acknowledge_cb function is called when a filter acknowledged event occurs.

Note that this function is called only if the version field is RT_CALLBACK_VERSION_1 or later.

The function uses the following signature:

```
cb_ret_t my_filter_acknowledged(
     rt_handle_t **handle,
     rt_filter_id_t filter_id,
     void *extended_args,
     void *data);
```

Table 14-15 lists the arguments to the filter_acknowledge_cb function.

Table 14-15 Field filter_acknowledge_cb

Argument	Description
handle	Real-time handle on which the event occurred
filter_id	Filter identifier
extended_args	Not used; NULL for now
data	Application data forwarded by rt_read_msgs()

Field htc_compute_node_failed_cb

The htc_compute_node_failed_cb function is called when a HTC compute node failed event occurs.

Note that this function is called only if the version field is RT_CALLBACK_VERSION_1 or later.

The function uses the following signature:

```
cb_ret_t my_htc_compute_node_failed(
     rt_handle_t **handle,
     rt_compute_node_fail_info_t *compute_node_fail_info,
     void *extended_args,
     void *data);
```

Table 14-16 lists the arguments to the `htc_compute_node_failed_cb` function.

Table 14-16 Field htc_compute_node_failed_cb

Argument	Description
handle	Real-time handle on which the event occurred
compute_node_fail_info	Opaque structure with information about the compute node failure
extended_args	Not used; NULL for now
data	Application data forwarded by rt_read_msgs()

Field htc_io_node_failed_cb

The `htc_io_node_failed_cb` function is called when a HTC I/O node failed event occurs.

Note that this function is called only if the `version` field is `RT_CALLBACK_VERSION_1` or later.

The function uses the following signature:

```
cb_ret_t my_htc_io_node_failed(
     rt_handle_t **handle,
     rt_io_node_fail_info_t *io_node_fail_info,
     void *extended_args,
     void *data);
```

Table 14-17 lists the arguments to the `htc_io_node_failed_cb` function.

Table 14-17 Field htc_io_node_failed_cb

Argument	Description
handle	Real-time handle on which the event occurred
`io_node_fail_info`	Opaque structure with information about the I/O node failure
extended_args	Not used; NULL for now
data	Application data forwarded by rt_read_msgs()

Field ras_event_cb

The `ras_event_cb` function is called when a RAS event occurs.

Note that this function is called only if the `version` field is `RT_CALLBACK_VERSION_2`.

The function uses the following signature:

```
cb_ret_t my_rt_ras_event(
     rt_handle_t **handle,
     rt_ras_event_t *ras_event_info,
     void *extended_args,
     void *data);
```

Table 14-18 lists the arguments to the ras_event_cb function.

Table 14-18 Field ras_event_cb

Argument	Description
handle	Real-time handle on which the event occurred
ras_event_info	Opaque structure with information about the RAS event
extended_args	Not used; NULL for now
data	Application data forwarded by rt_read_msgs()

14.4 Real-time elements

A *real-time element* is an opaque structure that contains data fields. These structures are opaque in that the application does not know the internals of the structure. The data fields available in an element depend on the type of the element. Pointers to real-time elements are passed to some of the real-time callback functions.

The value of a data field in a real-time element is retrieved using the rt_get_data() function. The element is passed into this function as the *elem* argument. The data field is identified by the *field* argument. If the element and field are valid, the *data_out* argument is set to point to the value of the data field for the element.

When an element is passed into a callback function, that element exists only for the duration of the callback. If the application wants to use the element outside of the callback, it must duplicate the element using rt_dup_element(). The application is responsible for freeing duplicated elements by using rt_free_element() to prevent a memory leak.

The following sections describe the element types and provide an example.

14.4.1 Real-time element types

In this section, we provide descriptions of each of the real-time element types, including the fields defined for each element type.

Compute node failure information element

The compute node failure information element contains information about the compute node that failed. It is used by the htc_compute_node_failed_cb callback when the client is notified that a compute node failed. The data type of the compute node failure information element is rt_compute_node_fail_info_t.

The following descriptions are of each of the fields in the compute node failure information element.

RT_SPEC_ID	
Description	ID of the compute node that failed
Data type	rm_component_id_t

RT_SPEC_DB_JOB_ID	
Description	DB ID of the job running on the compute node
Data type	db_job_id_t

RT_SPEC_REASON	
Description	Text explaining why the compute node became unavailable (might return RT_NO_DATA)
Data type	char*

I/O node failure information element

The I/O node failure information element contains information about the I/O node that failed. It is used by the htc_io_node_failed_cb callback when the client is notified that an I/O node failed. The data type of the I/O node failure information element is rt_io_node_fail_info_t.

The following descriptions are of each of the fields in the I/O node failure information element.

RT_SPEC_ID	
Description	ID of the I/O node that failed
Data type	rm_ionode_id_t

RT_SPEC_COMPUTE_NODE_INFOS	
Description	Compute nodes associated with the I/O node
Data type	rt_ionode_fail_info_compute_node_infos_t*

RT_SPEC_REASON	
Description	Text explaining why the I/O node became unavailable (might return RT_NO_DATA)
Data type	char*

I/O node failure compute node information list element

The I/O node failure compute node information list element represents the list of compute node failure information elements associated with the I/O node failure. This type of element is in the RT_SPEC_COMPUTE_NODE_INFOS field of the I/O node failure information element. The data type of the I/O node failure compute node information list element is rt_ionode_fail_info_compute_node_infos_t.

The following descriptions are of each of the fields in the I/O node failure compute node information list element.

RT_SPEC_LIST_FIRST	
Description	First compute node info. rt_get_data() returns RT_NO_DATA if there are no elements
Data type	rt_ionode_fail_info_compute_node_info_t*

RT_SPEC_LIST_NEXT	
Description	Next compute node info. rt_get_data() returns RT_NO_DATA if there are no more elements
Data type	rt_ionode_fail_info_compute_node_info_t*

I/O node failure compute node information element

The I/O node failure compute node information element contains information about a compute node that is associated with an I/O node failure. It is the type of the fields in the I/O node failure compute node information list element. The data type of this element is rt_ionode_fail_info_compute_node_info_t.

The following description is of the field in the I/O node failure compute node information element.

RT_SPEC_ID	
Description	ID of the compute node associated with the I/O node that failed
Data type	rm_component_id_t

RAS event element

The RAS event element represents a RAS event. It is used by the ras_event_cb callback. The data type of this element is rt_ras_event_t.

The following descriptions are of each of the fields in the RAS event element.

RT_SPEC_RECORD_ID	
Description	The RAS event’s record ID
Data type	rt_ras_record_id_t

RT_SPEC_MESSAGE_ID	
Description	The RAS event’s message ID
Data type	char*

RT_SPEC_SEVERITY	
Description	The RAS event’s severity

Data type	rt_ras_severity_t

14.4.2 Example

Example 14-2 illustrates the use of real-time elements in a callback function that prints out the information in the I/O node failure information element, as shown in Example 14-2.

Example 14-2 Accessing the fields of a real-time element

```
cb_ret_t rt_htc_io_node_failed_callback(
        rt_handle_t** handle,
        rt_io_node_fail_info_t* io_node_fail_info,
        void* extended_args,
        void* data
    )
{
    rt_status rc;

    rm_ionode_id_t io_node_id;
    rc = rt_get_data( (rt_element_t*) io_node_fail_info, RT_SPEC_ID, &io_node_id );

    const char *reason_buf = "";
    const char *reason_p(NULL);
    rc = rt_get_data( (rt_element_t*) io_node_fail_info, RT_SPEC_REASON, &reason_buf );
    if ( rc == RT_STATUS_OK ) {
        reason_p = reason_buf;
    } else if ( rc == RT_NO_DATA ) {
        reason_p = NULL;
        rc = RT_STATUS_OK;
    }

    ostringstream sstr;
    sstr << "[";

    rt_ionode_fail_info_compute_node_infos *cn_infos(NULL);
    rc = rt_get_data( (rt_element_t*) io_node_fail_info, RT_SPEC_COMPUTE_NODE_INFOS, &cn_infos
);

    int i(0);
    while ( true ) {

        rt_ionode_fail_info_compute_node_info *cn_info_p(NULL);

        rt_specification_t spec(i == 0 ? RT_SPEC_LIST_FIRST : RT_SPEC_LIST_NEXT);

        rc = rt_get_data( (rt_element_t*) cn_infos, spec, &cn_info_p );
        if ( rc == RT_NO_DATA ) {
            rc = RT_STATUS_OK;
            break;
        }

        rm_component_id_t compute_node_id(NULL);
        rc = rt_get_data( (rt_element_t*) cn_info_p, RT_SPEC_ID, &compute_node_id );

        if ( i++ > 0 ) {
            sstr << ",";
        }
        sstr << compute_node_id;
    }

    sstr << "]";

    string cn_ids_str(sstr.str());

    cout << "Received callback for HTC I/O node failed.\n"
            " io_node=" << io_node_id <<
            " cns=" << cn_ids_str;
    if ( reason_p ) {
        cout << " reason='" << reason_p << "'";
    }
    cout << endl;

    return RT_CALLBACK_CONTINUE;
}
```

14.5 Server-side filtering

An application using the real-time APIs might not be interested in all types of events that the real-time server sends, for example, it might be interested only in events that affect a certain partition or job. One way to handle this situation is for the client to receive all events and discard the ones it is not interested in. This approach is inefficient because the server must send messages and the client must process them. A more efficient approach has the client tell the server the types of events that it is not interested in receiving. This approach is accomplished in the real-time APIs using server-side filtering.

To use server-side filtering, first create a rt_filter_t instance using rt_create_server_filter(). A rt_filter_t instance has properties associated with it that indicate what the client is interested in. The filter created by this API is initialized with default values for its properties. To change the values of the filter's properties, call rt_server_filter_set_property() for each value to be changed. Then call rt_set_server_filter() to set the filter on the real-time handle. rt_set_server_filter() generates a filter ID unique to the real-time handle. The client's filter_acknowledge_cb callback function is called when the client receives the filter acknowledgment message from the server. After this point, the filter is in effect, and the client does not receive the events that have been filtered.

The client can set the filter on a handle before it has requested real-time messages using rt_request_realtime().

14.5.1 Pattern filter properties

Some filter properties specify patterns. The values of these properties are regular expressions, as defined in the documentation for the regcomp() function from the GNU C library, for example, the RT_FILTER_PROPERTY_PARTITION_ID property specifies the pattern for the IDs of the partitions that the application receives events for. Setting the value to "^MYPREFIX_.*$" ensures that the application only receives partition events that occur on partitions with IDs that start with MYPREFIX_.

14.5.2 Filter properties

In this section, we provide a list of the properties that are associated with a real-time filter. Each property is listed with its name, description, default argument, and argument type. The name is the constant value that is passed as the filter_property parameter. The description describes the effect that the property has on the callbacks that the application receives when the filter is applied to the real-time handle. The default value is the value of the property when the filter is created using the rt_create_server_filter() API. The argument type is the C language type expected for the property_value parameter.

Job filter properties

The following descriptions are of the job filter properties.

RT_FILTER_PROPERTY_JOBS	
Description	Indicates whether the application wants any job callbacks called. The value is an integer, where non-zero indicates that job callbacks are called, and 0 indicates that job callbacks are not called.
Default value	1, job callbacks are called
Argument type	int*

RT_FILTER_PROPERTY_JOB_ID	
Description	A pattern specifying the job IDs that the job callbacks are called for.
Default value	Jobs are not filtered by job ID
Argument type	char*, a C-style string

RT_FILTER_PROPERTY_JOB_STATES	
Description	The states that jobs are changing to that the job callbacks are called for.
Default value	Jobs are not filtered by state
Argument type	rm_job_state*, an array of rm_job_state terminated by RM_JOB_NAV

RT_FILTER_PROPERTY_JOB_DELETED	
Description	Indicates whether the application wants the job deletion callback called. The value is an integer, where non-zero indicates that the job deletion callback is called, and 0 indicates that the job deletion callback is not called.
Default value	1, job deletion callbacks are called
Argument type	int*

RT_FILTER_PROPERTY_JOB_TYPE	
Description	Indicates the type of jobs that the application wants the job callbacks called for. The value is one of the rt_filter_property_partition_type_t enum values. - RT_FILTER_PARTITION_TYPE_HPC_ONLY: only send events for HPC jobs - RT_FILTER_PARTITION_TYPE_HTC_ONLY: only send events for HTC jobs - RT_FILTER_PARTITION_TYPE_ANY: send events for any type of job
Default value	RT_FILTER_PARTITION_TYPE_HPC_ONLY
Argument type	rt_filter_property_partition_type_t*

RT_FILTER_PROPERTY_JOB_PARTITION	
Description	A pattern specifying the IDs of the partitions for the jobs that the application wants the job callbacks called for.
Default value	Jobs are not filtered by partition ID
Argument type	char*, a C-style string

Partition filter properties

The following descriptions are of the partition filter properties.

RT_FILTER_PROPERTY_PARTITIONS	
Description	Indicates whether the application wants any partition callbacks called. The value is an integer, where non-zero indicates that partition callbacks are called, and 0 indicates that partition callbacks are not called.
Default value	1, partition callbacks are called
Argument type	int*

RT_FILTER_PROPERTY_PARTITION_ID	
Description	A pattern specifying the partition IDs that the partition callbacks are called for.
Default value	Partitions are not filtered by ID
Argument type	char*, a C-style string

RT_FILTER_PROPERTY_PARTITION_STATES	
Description	The states that partitions are changing to that the partition callbacks are called for.
Default value	Partitions are not filtered by state
Argument type	rm_partition_state*, an array of rm_partition_state terminated by RM_PARTITION_NAV

RT_FILTER_PROPERTY_PARTITION_DELETED	
Description	Indicates whether the application wants the partition deletion callback called. The value is an integer, where non-zero indicates that the partition deletion callback is called, and 0 indicates that the partition deletion callback is not called.
Default value	1, partition deletion callbacks are called
Argument type	int*

RT_FILTER_PROPERTY_PARTITION_TYPE	
Description	Indicates the type of jobs that the application wants the job callbacks called for. The value is one of the rt_filter_property_partition_type_t enum values. - RT_FILTER_PARTITION_TYPE_HPC_ONLY: only send events for HPC jobs - RT_FILTER_PARTITION_TYPE_HTC_ONLY: only send events for HTC jobs - RT_FILTER_PARTITION_TYPE_ANY: send events for any type of job
Default value	RT_FILTER_PARTITION_TYPE_ANY
Argument type	rt_filter_property_partition_type_t*

Base partition (midplane) filter properties

The following descriptions are of the base partition filter properties.

RT_FILTER_PROPERTY_BPS	
Description	Indicates whether the application wants any base partition callbacks called. The value is an integer, where non-zero indicates that base partition callbacks are called, and 0 indicates that base partition callbacks are not called.
Default value	1, partition callbacks are called
Argument type	int*

RT_FILTER_PROPERTY_BP_ID	
Description	The pattern for base partition IDs.
Default value	Base partitions are not filtered by ID.
Argument type	char*, a C-style string

RT_FILTER_PROPERTY_BP_STATES	
Description	The states that base partitions are changing to that the base partition callbacks are called for.
Default value	Base partitions are not filtered by state.
Argument type	rm_BP_state*, an array of rm_BP_state terminated by RM_BP_NAV

Node card filter properties

The following descriptions are of the node card filter properties.

RT_FILTER_PROPERTY_NODE_CARDS	
Description	Indicates whether the application wants any node card callbacks called. The value is an integer, where non-zero indicates that node card callbacks are called, and 0 indicates that node card callbacks are not called.
Default value	1, node card callbacks are called
Argument type	int*

RT_FILTER_PROPERTY_NODE_CARD_ID	
Description	The pattern for node card IDs.
Default value	Node cards are not filtered by ID.
Argument type	char*, a C-style string

RT_FILTER_PROPERTY_NODE_CARD_STATES	
Description	The states that node cards are changing to that the node card callbacks are called for.
Default value	Node cards are not filtered by state.
Argument type	rm_nodecard_state_t*, an array of rm_nodecard_state_t terminated by RM_NODECARD_NAV

Switch filter properties

The following descriptions are of the switch filter properties.

RT_FILTER_PROPERTY_SWITCHES	
Description	Indicates whether the application wants any switch callbacks called. The value is an integer, where non-zero indicates that switch callbacks are called, and 0 indicates that switch callbacks are not called.
Default value	1, switch callbacks are called.
Argument type	int*

RT_FILTER_PROPERTY_SWITCH_ID	
Description	The pattern for switch IDs.
Default value	Switches are not filtered by ID.
Argument type	char*, a C-style string

RT_FILTER_PROPERTY_SWITCH_STATES	
Description	The states that switches are changing to that the switch callbacks are called for.
Default value	Switches are not filtered by state.
Argument type	rm_switch_state_t*, an array of rm_switch_state_t terminated by RM_SWITCH_NAV

Wire filter properties

The following descriptions are of the wire filter properties.

RT_FILTER_PROPERTY_WIRES	
Description	Indicates whether the application wants any wire callbacks called. The value is an integer, where non-zero indicates that wire callbacks are called, and 0 indicates that wire callbacks are not called.
Default value	1, wire callbacks are called.
Argument type	int*

RT_FILTER_PROPERTY_WIRE_ID	
Description	The pattern for wire IDs.
Default value	Wires are not filtered by ID.
Argument type	char*, a C-style string

RT_FILTER_PROPERTY_WIRE_STATES	
Description	The states that wires are changing to that the wire callbacks are called for.
Default value	Wires are not filtered by state.
Argument type	rm_wire_state_t*, an array of rm_wire_state_t terminated by RM_WIRE_NAV

Filter properties related to HTC events

The following descriptions are of the filter properties related to HTC events.

RT_FILTER_PROPERTY_HTC_EVENTS	
Description	Indicates whether the application wants any HTC callbacks called. The value is an integer, where non-zero indicates that HTC callbacks are called, and 0 indicates that HTC callbacks are not called.
Default value	1, HTC callbacks are called.
Argument type	int*

RT_FILTER_PROPERTY_HTC_COMPUTE_NODE_FAIL	
Description	Indicates whether the application wants the HTC compute node failure callback called. The value is an integer, where non-zero indicates that the HTC compute node failure callback is called, and 0 indicates that the HTC compute node failure callback is not called.
Default value	1, the HTC compute node callback is called.
Argument type	int*

RT_FILTER_PROPERTY_HTC_IO_NODE_FAIL	
Description	Indicates whether the application wants the HTC I/O node failure callback called. The value is an integer, where non-zero indicates that the HTC I/O node failure callback is called, and 0 indicates that the HTC I/O node failure callback is not called.
Default value	1, the HTC I/O node callback is called.
Argument type	int*

Filter properties related to RAS events

The following descriptions are of the filter properties related to RAS events.

RT_FILTER_PROPERTY_RAS_EVENTS	
Description	Indicates whether the application wants any RAS event callbacks called. The value is an integer, where non-zero indicates that RAS event callbacks are called, and 0 indicates that RAS event callbacks are not called.
Default value	0, RAS callbacks are not called.
Argument type	int*

RT_FILTER_PROPERTY_RAS_MESSAGE_ID	
Description	The pattern for RAS event message IDs.
Default value	RAS events aren’t filtered by message ID.
Argument type	char*, a C-style string

RT_FILTER_PROPERTY_RAS_SEVERITIES	
Description	The pattern for message severities for RAS events.
Default value	RAS events aren’t filtered by severity.
Argument type	rt_ras_severity_t*, an array or rt_ras_severity_t, terminated by RT_RAS_SEVERITY_NAV

RT_FILTER_PROPERTY_RAS_JOB_DB_IDS	
Description	The job database IDs for RAS events.
Default value	RAS events aren’t filtered by job database ID.
Argument type	db_job_id_t*, an array of db_job_id_t, terminated by -1.

RT_FILTER_PROPERTY_RAS_PARTITION_ID	
Description	The pattern for RAS event partition IDs.
Default value	RAS events aren’t filtered by partition ID.
Argument type	char*, a C-style string

14.5.3 Example

Example 14-3 illustrates use of the real-time server-side filtering APIs.

Example 14-3 Using the real-time server-side filtering APIs

```
#include <rt_api.h>
#include <iostream>
using namespace std;

cb_ret_t rt_filter_acknowledge_callback(
        rt_handle_t **handle,
        rt_filter_id_t filter_id,
        void* extended_args, void* data
    )
{
    cout << "Received callback for filter acknowledged for filter ID " << filter_id << endl;
    return RT_CALLBACK_CONTINUE;
}

int main( int argc, char *argv[] ) {
    rt_filter_t *filter_handle(NULL);
    rt_create_server_filter( &filter_handle );

    char job_name_pattern[] = "^MYPREFIX.*$";
    rt_server_filter_set_property( filter_handle, RT_FILTER_PROPERTY_JOB_ID,
        (void*) job_name_pattern );

    int filter_parts( 0 );
    rt_server_filter_set_property( filter_handle, RT_FILTER_PROPERTY_PARTITIONS,
        (void*) &filter_parts );

    rm_BP_state_t bp_states[] = { RM_BP_UP, RM_BP_ERROR, RM_BP_NAV };
    rt_server_filter_set_property( filter_handle, RT_FILTER_PROPERTY_BP_STATES,
        (void*) bp_states );

    rt_filter_id_t filter_id;
    rt_handle_t *rt_handle;

    rt_callbacks_t rt_callbacks;
    rt_callbacks.version = RT_CALLBACK_VERSION_CURRENT;
    rt_callbacks.filter_acknowledge_cb = &rt_filter_acknowledge_callback;

    rt_init( &rt_handle, RT_BLOCKING, &rt_callbacks );
    rt_set_server_filter( &rt_handle, filter_handle, &filter_id );
    rt_request_realtime( &rt_handle );
    rt_read_msgs( &rt_handle, NULL);
    rt_close( &rt_handle );
}
```

14.6 Real-time Notification APIs status codes

When a failure occurs, an API invocation returns a status code. This status code helps apply automatic corrective actions within the resource management application. In addition, a failure always generates a log message, which provides more information for the possible cause of the problem and any corrective action. These log messages are used for debugging and non-automatic recovery of failures.

The design aims at striking a balance between the number of status codes detected and the different error paths per status code. Thus, some errors have specific status codes, while others have more generic ones.

The Real-time Notification APIs use the following status codes:

- `RT_STATUS_OK`: API call completed successfully.
- `RT_NO_REALTIME_MSGS`: No events available.
- `RT_WOULD_BLOCK`: In nonblocking mode and the request blocks.
- `RT_FINISHED_PREV`: Previous request completed.
- `RT_NO_DATA`: The field in the real-time element contains no data.
- `RT_CONNECTION_ERROR`: Connection to the real-time server failed.
- `RT_INTERNAL_ERROR`: Unexpected internal error. No recovery possible.
- `RT_INVALID_INPUT_ERROR`: The input to the API is bad due to missing required data, illegal data, and so on.
- `RT_DB_PROPERTY_ERROR`: Error trying to read the db.properties file.
- `RT_PROTOCOL_ERROR`: An incorrect message was received from the real-time server.
- `RT_HANDLE_CLOSED`: The handle passed to the API was previously closed.
- `RT_UNEXPECTED_FIELD`: The field is not valid for the real-time element.

14.6.1 Status code specification

The various API functions have the following status codes:

- `rt_status_t rt_init(rt_handle_t **handle_out, rt_block_flag_t blocking_flag, rt_callbacks_t* callbacks);`

  This function initializes a real-time handle.

  The status codes are:

  – `RT_STATUS_OK`: The handle is initialized.
  – `RT_CONNECTION_ERROR`: Failed to connect to the real-time server.
  – `RT_INVALID_INPUT_ERROR`: One or more of the parameters are not valid.
  – `RT_INTERNAL_ERROR`: An unexpected internal error occurred in setting blocking or nonblocking mode on socket.
  – `RT_DB_PROPERTY_ERROR`: Problem accessing the db.properties file.

- `rt_status_t rt_close(rt_handle_t **handle);`

  This function closes a real-time handle.

  The status codes are:

  – `RT_STATUS_OK`: The handle was closed.
  – `RT_INTERNAL_ERROR`: An unexpected internal error occurred in closing the handle.
  – `RT_INVALID_INPUT_ERROR`: The handle is not valid.
  – `RT_HANDLE_CLOSED`: The handle was already closed.

- `rt_status_t rt_set_blocking(rt_handle_t **handle);`

  This function sets a real-time handle to blocking mode.

  The status codes are:

  – `RT_STATUS_OK`: Blocking mode was set for the handle.
  – `RT_INTERNAL_ERROR`: An unexpected internal error occurred in setting blocking mode.
  – `RT_INVALID_INPUT_ERROR`: The handle is not valid.
  – `RT_HANDLE_CLOSED`: The handle was closed.

- `rt_status_t rt_set_nonblocking(rt_handle_t **handle);`

  This function sets a real-time handle to nonblocking mode.

  The status codes are:

  – `RT_STATUS_OK`: Nonblocking mode was set for the handle.
  – `RT_INTERNAL_ERROR`: An unexpected internal error occurred in setting nonblocking mode.
  – `RT_INVALID_INPUT_ERROR`: The handle is not valid.
  – `RT_HANDLE_CLOSED`: The handle was closed.

- `rt_status_t rt_create_server_filter(rt_filter_t **filter_out);`

  This function creates a server-side filter object.

  The status codes are:

  – `RT_STATUS_OK`: The server-side filter was created.
  – `RT_INVALID_INPUT_ERROR`: The parameter is not valid.
  – `RT_INTERNAL_ERROR`: An unexpected internal error occurred when creating the server-side filter object.

- `rt_status_t rt_server_filter_set_property(rt_filter_t *filter, rt_server_filter_property_t filter_property, void *property_value);`

  This function sets a property of the server-side filter object.

The status codes are:

– `RT_STATUS_OK`: The server-side filter was created.
– `RT_INVALID_INPUT_ERROR`: The parameter is not valid.
– `RT_INTERNAL_ERROR`: An unexpected internal error occurred when creating the server-side filter object.

- `rt_status_t rt_set_server_filter(rt_handle_t **handle, const rt_filter_t *filter, rt_filter_id_t *filter_id_out);`

  This function assigns a server filter to a real-time handle.

  The status codes are:

  – `RT_STATUS_OK`: The server-side filter was assigned.
  – `RT_WOULD_BLOCK`: The handle is nonblocking, and this request blocks.
  – `RT_FINISHED_PREV`: A previous request finished.
  – `RT_CONNECTION_ERROR`: The connection to the server was lost.
  – `RT_INTERNAL_ERROR`: An unexpected internal error occurred when setting the server-side filter on the real-time handle.
  – `RT_INVALID_INPUT_ERROR`: A parameter is not valid.
  – `RT_PROTOCOL_ERROR`: Protocol error communicating with the server.
  – `RT_HANDLE_CLOSED`: The handle was closed.

- `rt_status_t rt_free_server_filter(rt_filter_t **filter_in_out);`

  This function frees a server-side filter object.

  The status codes are:

  – `RT_STATUS_OK`: The server-side filter was freed.
  – `RT_INTERNAL_ERROR`: An unexpected internal error occurred when freeing the server-side filter.
  – `RT_INVALID_INPUT_ERROR`: A parameter is not valid.

- `rt_status_t rt_set_filter(rt_handle_t **handle, rt_filter_type_t filter_type, const char* filter_names);`

  This function sets the client-side filter on a real-time handle.

  The status codes are:

  – `RT_STATUS_OK`: Filtering was set successfully.
  – `RT_INTERNAL_ERROR`: An unexpected internal error occurred in setting the filter.
  – `RT_INVALID_INPUT_ERROR`: An input parameter is not valid.
  – `RT_HANDLE_CLOSED`: The handle was closed.

- `rt_status_t rt_request_realtime(rt_handle_t **handle);`

  This function requests real-time events for this handle.

  The status codes are:

  – `RT_STATUS_OK`: Request to start real-time updates was successful.
  – `RT_WOULD_BLOCK`: The handle is nonblocking, and this request blocks.
  – `RT_FINISHED_PREV`: A previous request finished.
  – `RT_CONNECTION_ERROR`: The connection to the server was lost.
  – `RT_INTERNAL_ERROR`: An unexpected internal error occurred when requesting real-time.
  – `RT_INVALID_INPUT_ERROR`: The handle is not valid.
  – `RT_PROTOCOL_ERROR`: Protocol error communicating with the server.
  – `RT_HANDLE_CLOSED`: The handle was closed.

- `rt_status_t rt_get_socket_descriptor(rt_handle_t **handle, int *sd_out);`

  This function gets the socket descriptor used by the real-time APIs.

The status codes are:

- RT_STATUS_OK: Socket descriptor was retrieved successfully.
- RT_INTERNAL_ERROR: An unexpected internal error occurred when getting the socket descriptor.
- RT_INVALID_INPUT_ERROR: The handle is not valid.
- RT_HANDLE_CLOSED: The handle was closed.

- `rt_status_t rt_read_msgs(rt_handle_t **handle, void* data);`

  This function receives real-time events.

  The status codes are:

  - RT_STATUS_OK: Message or messages were read successfully.
  - RT_NO_REALTIME_MSGS: Nonblocking mode and no messages to receive.
  - RT_INVALID_INPUT_ERROR: The handle is not valid.
  - RT_CONNECTION_ERROR: The connection to the server was lost.
  - RT_INVALID_INPUT_ERROR: The handle is not valid.
  - RT_INTERNAL_ERROR: An unexpected internal error occurred when reading messages.
  - RT_PROTOCOL_ERROR: Protocol error communicating with the server.
  - RT_HANDLE_CLOSED: The handle was closed.

- `rt_status_t rt_get_data(rt_element_t *elem, rt_specification_t field, void *data_out);`

  This function gets data from a real-time element.

  The status codes are:

  - RT_STATUS_OK: The field value was retrieved successfully.
  - RT_NO_DATA: The field contains no data.
  - RT_INTERNAL_ERROR: An unexpected internal error occurred when getting data.
  - RT_INVALID_INPUT_ERROR: A parameter is not valid.
  - RT_UNEXPECTED_FIELD: The field is not valid for the element type.

- `rt_status_t rt_dup_element(rt_element_t *elem, rt_element_t **elem_out);`

  This function copies a real-time element.

  The status codes are:

  - RT_STATUS_OK: The element was copied.
  - RT_INTERNAL_ERROR: An unexpected internal error occurred when copying the element.
  - RT_INVALID_INPUT_ERROR: A parameter is not valid.

- `rt_status_t rt_free_element(rt_element_t *elem);`

  This function frees a real-time element.

  The status codes are:

  - RT_STATUS_OK: The element was freed.
  - RT_INTERNAL_ERROR: An unexpected internal error occurred when freeing the element.
  - RT_INVALID_INPUT_ERROR: A parameter is not valid.

14.7 Sample real-time application code

Example 14-4 shows basic sample code for calling the real-time APIs and programming the callback functions.

Example 14-4 Sample real-time application

```
#include <rt_api.h>
#include <sayMessage.h>

#include <stdio.h>
#include <unistd.h>

#include <iostream>
#include <sstream>

using namespace std;


// Converts partition state enum to character string for messages
string partition_state_to_msg( rm_partition_state_t state )
{
    switch ( state ) {
        case RM_PARTITION_FREE:
            return "Free";
        case RM_PARTITION_CONFIGURING:
            return "Configuring";
        case RM_PARTITION_READY:
            return "Ready";
        case RM_PARTITION_DEALLOCATING:
            return "Deallocating";
        case RM_PARTITION_ERROR:
            return "Error";
        case RM_PARTITION_NAV:
            return "Not a value (NAV)";
    }
    return "Unknown";
} // partition_state_to_msg()


// Converts job state enum to character string for messages
string job_state_to_msg( rm_job_state_t state )
{
    switch ( state ) {
        case RM_JOB_IDLE:
            return "Queued/Idle";
        case RM_JOB_STARTING:
            return "Starting";
        case RM_JOB_RUNNING:
            return "Running";
        case RM_JOB_TERMINATED:
            return "Terminated";
        case RM_JOB_ERROR:
            return "Error";
        case RM_JOB_DYING:
            return "Dying";
        case RM_JOB_DEBUG:
            return "Debug";
        case RM_JOB_LOAD:
            return "Load";
        case RM_JOB_LOADED:
            return "Loaded";
        case RM_JOB_BEGIN:
            return "Begin";
        case RM_JOB_ATTACH:
            return "Attach";
        case RM_JOB_NAV:
            return "Not a value (NAV)";
    }
    return "Unknown";
} // job_state_to_msg()


// Converts BP state enum to character string for messages
string BP_state_to_msg( rm_BP_state_t state )
{
    switch ( state ) {
        case RM_BP_UP:
            return "Available/Up";
        case RM_BP_MISSING:
            return "Missing";
        case RM_BP_ERROR:
            return "Error";
        case RM_BP_DOWN:
            return "Service/Down";
        case RM_BP_NAV:
            return "Not a value (NAV)";
    }
    return "Unknown";
} // BP_state_to_msg()


// Converts switch state enum to character string for messages
string switch_state_to_msg( rm_switch_state_t state )
{
    switch ( state ) {
        case RM_SWITCH_UP:
            return "Available/Up";
        case RM_SWITCH_MISSING:
            return "Missing";
        case RM_SWITCH_ERROR:
            return "Error";
        case RM_SWITCH_DOWN:
            return "Service/Down";
        case RM_SWITCH_NAV:
            return "Not a value (NAV)";
    }
    return "Unknown";
} // switch_state_to_msg()
```

```
// Converts nodecard state enum to character string for messages
string nodecard_state_to_msg( rm_nodecard_state_t state )
{
    switch ( state ) {
        case RM_NODECARD_UP:
            return "Available/Up";
        case RM_NODECARD_MISSING:
            return "Missing";
        case RM_NODECARD_ERROR:
            return "Error";
        case RM_NODECARD_DOWN:
            return "Service/Down";
        case RM_NODECARD_NAV:

            return "Not a value (NAV)";
    }
    return "Unknown";
} // nodecard_state_to_msg()


string wire_state_to_msg( rm_wire_state_t state )
{
    switch ( state ) {
        case RM_WIRE_UP:
            return "Available/Up";
        case RM_WIRE_MISSING:
            return "Missing";
        case RM_WIRE_ERROR:
            return "Error";
        case RM_WIRE_DOWN:
            return "Service/Down";
        case RM_WIRE_NAV:
            return "Not a value (NAV)";
    }
    return "Unknown";
} // wire_state_to_msg()


/* Definitions of the Real-time callback functions. */

cb_ret_t rt_end_callback( rt_handle_t **handle, void* extended_args, void* data )
{
    cout << "Received Real-time end message." << endl;
    return RT_CALLBACK_QUIT;
}

cb_ret_t rt_partition_added_callback( rt_handle_t **handle, rm_sequence_id_t seq_id,
        pm_partition_id_t partition_id, rm_partition_state_t partition_new_state,
        rt_raw_state_t partition_raw_new_state, void* extended_args, void* data )
{
    cout << "Received callback for add partition " << partition_id << " state of partition is "
            << partition_state_to_msg( partition_new_state ) << endl << "Raw state="
            << partition_raw_new_state << " sequence ID=" << seq_id << endl;
    return RT_CALLBACK_CONTINUE;
}
```

```
cb_ret_t rt_partition_state_changed_callback( rt_handle_t **handle, rm_sequence_id_t seq_id,
        rm_sequence_id_t prev_seq_id, pm_partition_id_t partition_id,
        rm_partition_state_t partition_new_state, rm_partition_state_t partition_old_state,
        rt_raw_state_t partition_raw_new_state, rt_raw_state_t partition_raw_old_state,
        void* extended_args, void* data )
{
    cout << "Received callback for partition " << partition_id << " state change, old state is "
            << partition_state_to_msg( partition_old_state ) << ", new state is "
            << partition_state_to_msg( partition_new_state ) << endl << "Raw old state="
            << partition_raw_old_state << " Raw new state=" << partition_raw_new_state
            << " New sequence ID=" << seq_id << " Previous sequence ID=" << prev_seq_id << endl;
    return RT_CALLBACK_CONTINUE;
}

cb_ret_t rt_partition_deleted_callback( rt_handle_t **handle, rm_sequence_id_t prev_seq_id,
        pm_partition_id_t partition_id, void* extended_args, void* data )
{
    cout << "Received callback for delete on partition " << partition_id
            << " Previous sequence ID=" << prev_seq_id << endl;
    return RT_CALLBACK_CONTINUE;
}

cb_ret_t rt_job_added_v1_callback(
        rt_handle_t **handle,
        rm_sequence_id_t seq_id,
        jm_job_id_t job_id,
        db_job_id_t db_job_id,
        pm_partition_id_t partition_id,
        rm_job_state_t job_new_state,
        rt_raw_state_t job_raw_new_state,
        void* extended_args,
        void* data
    )
{
    cout << "Received callback for add job " << job_id << " with id " << db_job_id << " on
partition " << partition_id << ","
            << " state of job is " << job_state_to_msg( job_new_state ) << endl << "Raw new
state="
            << job_raw_new_state << " New sequence ID=" << seq_id << endl;
    return RT_CALLBACK_CONTINUE;
}

cb_ret_t rt_job_state_changed_v1_callback(
        rt_handle_t **handle,
        rm_sequence_id_t seq_id,
        rm_sequence_id_t previous_seq_id,

        jm_job_id_t job_id,
        db_job_id_t db_job_id,
        pm_partition_id_t partition_id,
        rm_job_state_t job_new_state,
        rm_job_state_t job_old_state,
        rt_raw_state_t job_raw_new_state,
        rt_raw_state_t job_raw_old_state,
        void* extended_args,
        void* data
    )
{
    cout << "Received callback for job " << job_id << " with id " << db_job_id
            << " state change on partition " << partition_id
            << ", old state is " << job_state_to_msg( job_old_state ) << ", new state is "
            << job_state_to_msg( job_new_state ) << endl << "Raw old state=" <<
job_raw_old_state
            << " Raw new state=" << job_raw_new_state << " New sequence ID=" << seq_id
            << " Previous sequence ID=" << previous_seq_id << endl;
    return RT_CALLBACK_CONTINUE;
}

cb_ret_t rt_job_deleted_v1_callback(
        rt_handle_t **handle,
        rm_sequence_id_t previous_seq_id,
        jm_job_id_t job_id,
        db_job_id_t db_job_id,
        pm_partition_id_t partition_id,
        void* extended_args,
        void* data
    )
{
    cout << "Received callback for delete of job " << job_id << " with id " << db_job_id
            << " on partition " << partition_id
            << " Previous sequence ID=" << previous_seq_id << endl;
    return RT_CALLBACK_CONTINUE;
}

cb_ret_t rt_BP_state_changed_callback(
        rt_handle_t **handle,
        rm_sequence_id_t seq_id,
        rm_sequence_id_t prev_seq_id,
        rm_bp_id_t bp_id,
        rm_BP_state_t BP_new_state,
        rm_BP_state_t BP_old_state,
        rt_raw_state_t BP_raw_new_state,
        rt_raw_state_t BP_raw_old_state,
        void* extended_args,
        void* data
    )
{
    cout << "Received callback for BP " << bp_id << " state change,"
            " old state is " << BP_state_to_msg( BP_old_state ) << ","
            " new state is " << BP_state_to_msg( BP_new_state ) << "\n"
            "Raw old state=" << BP_raw_old_state <<
            " Raw new state=" << BP_raw_new_state <<
            " New sequence ID=" << seq_id <<
            " Previous sequence ID=" << prev_seq_id << endl;
    return RT_CALLBACK_CONTINUE;
}

cb_ret_t rt_switch_state_changed_callback(
        rt_handle_t **handle,
        rm_sequence_id_t seq_id,
        rm_sequence_id_t prev_seq_id,
        rm_switch_id_t switch_id,
        rm_bp_id_t bp_id,
        rm_switch_state_t switch_new_state,
        rm_switch_state_t switch_old_state,
        rt_raw_state_t switch_raw_new_state,
        rt_raw_state_t switch_raw_old_state,
        void* extended_args,
        void* data
    )
{
    cout << "Received callback for switch " << switch_id << " state change on BP " << bp_id <<
","
            " old state is " << switch_state_to_msg( switch_old_state ) << ","
            " new state is " << switch_state_to_msg( switch_new_state ) << "\n"
            "Raw old state=" << switch_raw_old_state <<
            " Raw new state=" << switch_raw_new_state <<
            " New sequence ID=" << seq_id <<
            " Previous sequence ID=" << prev_seq_id << endl;
    return RT_CALLBACK_CONTINUE;
}

cb_ret_t rt_nodecard_state_changed_callback(
        rt_handle_t **handle,
        rm_sequence_id_t seq_id,
        rm_sequence_id_t prev_seq_id,
        rm_nodecard_id_t nodecard_id,
        rm_bp_id_t bp_id,
        rm_nodecard_state_t nodecard_new_state,
        rm_nodecard_state_t nodecard_old_state,
        rt_raw_state_t nodecard_raw_new_state,
        rt_raw_state_t nodecard_raw_old_state,
        void* extended_args,
        void* data
    )
{
    cout << "Received callback for node card " << nodecard_id <<
            " state change on BP " << bp_id << ","
            " old state is " << nodecard_state_to_msg( nodecard_old_state ) << ","
            " new state is " << nodecard_state_to_msg( nodecard_new_state ) << "\n"
            "Raw old state=" << nodecard_raw_old_state <<
            " Raw new state=" << nodecard_raw_new_state <<
            " New sequence ID=" << seq_id <<
            " Previous sequence ID=" << prev_seq_id << endl;
    return RT_CALLBACK_CONTINUE;
}


cb_ret_t rt_wire_state_changed_callback(
        rt_handle_t **handle,
        rm_sequence_id_t seq_id,
        rm_sequence_id_t previous_seq_id,
        rm_wire_id_t wire_id,
        rm_wire_state_t wire_new_state,
        rm_wire_state_t wire_old_state,
        rt_raw_state_t wire_raw_new_state,
        rt_raw_state_t wire_raw_old_state,
        void* extended_args,
        void* data
    )
{
    cout << "Received callback for wire '" << wire_id << "',"
            " old state is " << wire_state_to_msg( wire_old_state ) << ","
            " new state is " << wire_state_to_msg( wire_new_state ) << "\n"
            "Raw old state=" << wire_raw_old_state <<
            " Raw new state=" << wire_raw_new_state <<
            " New sequence ID=" << seq_id <<
            " Previous sequence ID=" << previous_seq_id << endl;

    return RT_CALLBACK_CONTINUE;
}


cb_ret_t rt_filter_acknowledge_callback(
        rt_handle_t **handle,
        rt_filter_id_t filter_id,
        void* extended_args, void* data
    )
{
    cout << "Received callback for filter acknowledged for filter ID " << filter_id << endl;
    return RT_CALLBACK_CONTINUE;
}


cb_ret_t rt_htc_compute_node_failed_callback(
        rt_handle_t** handle,
        rt_compute_node_fail_info_t* compute_node_fail_info,
        void* extended_args,
        void* data
    )
{
    rt_status rc;

    rm_component_id_t compute_node_id;
    rc = rt_get_data( (rt_element_t*) compute_node_fail_info, RT_SPEC_ID, &compute_node_id );
    if ( rc != RT_STATUS_OK ) {
        cerr << "rt_get_data failed in " << __FUNCTION__ << " getting RT_SPEC_ID\n";
        return RT_CALLBACK_CONTINUE;
    }

    db_job_id_t db_job_id;
    rc = rt_get_data( (rt_element_t*) compute_node_fail_info, RT_SPEC_DB_JOB_ID, &db_job_id );
    if ( rc != RT_STATUS_OK ) {
        cerr << "rt_get_data failed in " << __FUNCTION__ << " getting RT_SPEC_DB_JOB_ID\n";
        return RT_CALLBACK_CONTINUE;
    }

    const char *reason_buf = "";
    const char *reason_p;
    rc = rt_get_data( (rt_element_t*) compute_node_fail_info, RT_SPEC_REASON, &reason_buf );
    if ( rc == RT_STATUS_OK ) {
        reason_p = reason_buf;
    } else if ( rc == RT_NO_DATA ) {
        reason_p = NULL;
        rc = RT_STATUS_OK;
    } else {
        cerr << "rt_get_data failed in " << __FUNCTION__ << " getting RT_SPEC_REASON\n";
        return RT_CALLBACK_CONTINUE;
    }

    cout << "Received callback for HTC compute node failed.\n"
            " compute_node=" << compute_node_id <<
            " db_job_id=" << db_job_id;
    if ( reason_p ) {
        cout << " reason='" << reason_p << "'";
    }
    cout << endl;

    return RT_CALLBACK_CONTINUE;
}


cb_ret_t rt_htc_io_node_failed_callback(
        rt_handle_t** handle,
        rt_io_node_fail_info_t* io_node_fail_info,
        void* extended_args,
        void* data
    )
{
    rt_status rc;

    rm_ionode_id_t io_node_id;
    rc = rt_get_data( (rt_element_t*) io_node_fail_info, RT_SPEC_ID, &io_node_id );
    if ( rc != RT_STATUS_OK ) {
        cerr << "rt_get_data failed in " << __FUNCTION__ << " getting RT_SPEC_ID.\n";
        return RT_CALLBACK_CONTINUE;
    }

    const char *reason_buf = "";
    const char *reason_p(NULL);
    rc = rt_get_data( (rt_element_t*) io_node_fail_info, RT_SPEC_REASON, &reason_buf );
    if ( rc == RT_STATUS_OK ) {
        reason_p = reason_buf;
    } else if ( rc == RT_NO_DATA ) {
        reason_p = NULL;
        rc = RT_STATUS_OK;
    } else {
        cerr << "rt_get_data failed in " << __FUNCTION__ << " getting RT_SPEC_REASON\n";
        return RT_CALLBACK_CONTINUE;
    }

    ostringstream sstr;
    sstr << "[";
    rt_ionode_fail_info_compute_node_infos *cn_infos(NULL);
    rc = rt_get_data( (rt_element_t*) io_node_fail_info, RT_SPEC_COMPUTE_NODE_INFOS, &cn_infos
);
    if ( rc != RT_STATUS_OK ) {
        cerr << "rt_get_data failed in " << __FUNCTION__ << " getting
RT_SPEC_COMPUTE_NODE_INFOS.\n";
        return RT_CALLBACK_CONTINUE;
    }

    int i(0);
    while ( true ) {

        rt_ionode_fail_info_compute_node_info *cn_info_p(NULL);

        rt_specification_t spec(i == 0 ? RT_SPEC_LIST_FIRST : RT_SPEC_LIST_NEXT);

        rc = rt_get_data( (rt_element_t*) cn_infos, spec, &cn_info_p );
        if ( rc == RT_NO_DATA ) {
            rc = RT_STATUS_OK;
            break;
        }
        if ( rc != RT_STATUS_OK ) {
            cerr << "rt_get_data failed in " << __FUNCTION__ << " getting compute node info list
element.\n";
            return RT_CALLBACK_CONTINUE;
        }

        rm_component_id_t compute_node_id(NULL);
        rc = rt_get_data( (rt_element_t*) cn_info_p, RT_SPEC_ID, &compute_node_id );
        if ( rc != RT_STATUS_OK ) {
            cerr << "rt_get_data failed in " << __FUNCTION__ << " getting compute node info
RT_SPEC_ID.\n";
            return RT_CALLBACK_CONTINUE;
        }

        if ( i++ > 0 ) {
            sstr << ",";
        }
        sstr << compute_node_id;
    }

    sstr << "]";

    string cn_ids_str(sstr.str());

    cout << "Received callback for HTC I/O node failed.\n"
            " io_node=" << io_node_id <<
            " cns=" << cn_ids_str;
    if ( reason_p ) {
        cout << " reason='" << reason_p << "'";
    }
    cout << endl;

    return RT_CALLBACK_CONTINUE;
}
```

```
/* Program entry point */


int main( int argc, char *argv[] )
{
    setSayMessageParams( stdout, verbose );

    rt_handle_t *rt_handle;
    rt_callbacks_t rt_callbacks;

    rt_callbacks.end_cb = &rt_end_callback;
    rt_callbacks.partition_added_cb = &rt_partition_added_callback;
    rt_callbacks.partition_state_changed_cb = &rt_partition_state_changed_callback;
    rt_callbacks.partition_deleted_cb = &rt_partition_deleted_callback;
    rt_callbacks.job_added_cb = NULL; // switched to using v1 callback.
    rt_callbacks.job_state_changed_cb = NULL; // switched to using v1 callback.
    rt_callbacks.job_deleted_cb = NULL; // switched to using v1 callback.
    rt_callbacks.bp_state_changed_cb = &rt_BP_state_changed_callback;
    rt_callbacks.switch_state_changed_cb = &rt_switch_state_changed_callback;
    rt_callbacks.nodecard_state_changed_cb = &rt_nodecard_state_changed_callback;
    rt_callbacks.job_added_v1_cb = &rt_job_added_v1_callback;
    rt_callbacks.job_state_changed_v1_cb = &rt_job_state_changed_v1_callback;
    rt_callbacks.job_deleted_v1_cb = &rt_job_deleted_v1_callback;
    rt_callbacks.wire_state_changed_cb = &rt_wire_state_changed_callback;
    rt_callbacks.filter_acknowledge_cb = &rt_filter_acknowledge_callback;
    rt_callbacks.htc_compute_node_failed_cb = &rt_htc_compute_node_failed_callback;
    rt_callbacks.htc_io_node_failed_cb = &rt_htc_io_node_failed_callback;

    // Get a handle, set socket to block, and setup callbacks
    if ( rt_init( &rt_handle, RT_BLOCKING, &rt_callbacks ) != RT_STATUS_OK ) {
        cout << "Failed on Real-time initialize (rt_init), exiting program." << endl;
        return -1;
    }

    // Tell Real-time server we are ready to handle messages
    if ( rt_request_realtime( &rt_handle ) != RT_STATUS_OK ) {
        cout << "Failed to connect to Real-time server, exiting program." << endl;
        rt_close( &rt_handle );
        return -1;
    }

    // Read messages
    if ( rt_read_msgs( &rt_handle, NULL) != RT_STATUS_OK ) {
        cout << "rt_read_msgs failed" << endl;
        rt_close( &rt_handle );
        return -1;
    }

    // Close the handle
    rt_close( &rt_handle );
    return 0;
} // main()
```

15

Dynamic Partition Allocator APIs

The Dynamic Partition Allocator APIs provide an easy-to-use interface for the dynamic creation of partitions. These APIs inspect the current state of the Blue Gene/P machine and attempt to create a partition based on available resources. If no resources are available that match the partition requirements, the partition is not created. It is expected that any job scheduler that uses the partition allocator does so from a centralized process to avoid conflicts in finding free resources to build the partition. Dynamic Partition Allocator APIs are thread safe. Only 64-bit shared libraries are provided.

In this chapter, we define a list of APIs into the Midplane Management Control System (MMCS) Dynamic Partition Allocator. See Chapter 13, “Control system (Bridge) APIs” on page 209, for details about the Bridge APIs.

We discuss the following specific topics in this chapter and provide a sample program:

- API support
- API details

> **Note:** The Dynamic Partition Allocator APIs changed in IBM Blue Gene/P release V1R3M0 in ways that are not compatible with previous releases. Programs using the Dynamic Partition Allocator APIs prior to V1R3M0 must be changed to use the new APIs.

15.1 Overview of API support

In the following sections, we provide an overview of the support provided by the APIs.

15.2 Requirements

When writing programs to the Dynamic Partition Allocator APIs, you must meet the following requirements:

- Operating system supported

  Currently, SUSE Linux Enterprise Server (SLES) 10 for PowerPC is the only supported platform.

- Languages supported

  C and C++ are supported with the GNU gcc 4.1.2 level compilers. For more information and downloads, refer to the following Web address:

  `http://gcc.gnu.org/`

- Include files

  All required include files are installed in the /bgsys/drivers/ppcfloor/include directory. The include file for the dynamic allocator API is allocator_api.h.

- Library files

  The Dynamic Partition Allocator APIs support 64-bit applications using dynamic linking with shared objects.

  Sixty-four bit libraries: The required library files are installed in the /bgsys/drivers/ppcfloor/lib64 directory. The shared object for linking to the Bridge APIs is libbgpallocator.so.

  The libbgpallocator.so library has dependencies on other libraries included with the IBM Blue Gene/P software, including the following objects:

  - libbgpbridge.so
  - libbgpconfig.so
  - libbgpdb.so
  - libsaymessage.so
  - libtableapi.so

  These files are installed with the standard system installation procedure. They are contained in the bgpbase.rpm file.

15.3 API details

In this section, we provide details about the APIs and return codes for dynamic partition allocation.

15.3.1 APIs

The following APIs are used for dynamic partition allocation and are all thread safe:

- `BGALLOC_STATUS rm_init_allocator(const char * caller_desc, const char * drain_list);`

  A program should call rm_init_allocator() and pass a description that will be used as the text description for all partitions used by subsequent rm_allocate_partition() calls, for example, passing in *ABC job scheduler* causes any partitions that are created by rm_allocate_partition() to have *ABC job scheduler* as the partition description.

  The caller can also optionally specify a drain list file name that identifies the base partitions (midplanes) that will be excluded from the list of resources to consider when allocating new partitions. If NULL is passed in for the drain list file name, a default drain list is set first from the following locations:

  – The path in the environment variable `ALLOCATOR_DRAIN_LIST` if it exists
  – The /etc/allocator_drain.lst file if it exists

  If no drain list file is established, no base partitions are excluded. If an invalid file name is passed in, the call fails, for example, a drain list file with the following content excludes base partitions R00-M0, R00-M1, and R01-M0 when allocating resources for a partition:

  ```
  R00-M0
  R00-M1
  R01-M0
  ```

  The list of resources can contain items separated by any white-space character (space, tab, new line, vertical tab, or form feed). Items found that do not match an existing resource are ignored, but an error message is logged.

- `BGALLOC_STATUS rm_allocate_partition(`

  ```
  const rm_size_t size,
  const rm_connection_type_t conn,
  const rm_size3D_t shape,
  const rm_job_mode_t mode,
  const rm_psetsPerBP_t psetsPerBP,
  const char * user_name,
  const char * caller_desc,
  const char * options,
  const char * ignoreBPs,
  const char * partition_id,
  char ** newpartition_id,
  const char * bootOptions);
  ```

  The caller to rm_allocate_partition() provides input parameters that describe the characteristics of the partition that should be created from available Blue Gene/P machine resources. If resources are available that match the requirements, a partition is created and allocated, and the partition name is returned to the caller along with a return code of `BGALLOC_OK`.

  If both size and shape values are provided, the allocation is based on the shape value only.

  The user_name parameter is required.

If the caller_desc value is NULL, the caller description specified on the call to rm_init_allocator() is used.

The options parameter is optional and can be NULL.

If the ignoreBPs parameter is not NULL, it must be a string of blank-separated base partition identifiers to be ignored. The base partitions listed in the parameter are ignored as though the partitions were included in the drain list file currently in effect.

If the partition_id parameter is not NULL, it can specify one of the following options:

– The name of the new partition

  The name can be from 1 to 32 characters. Valid characters are a...z, A...Z, 0...9, - (hyphen), and _ (underscore).

– The prefix to be used for generating a unique partition name

  The prefix can be from 1 to 16 characters, followed by an asterisk (*). Valid characters are the same as those for a new partition name, for example, if `ABC-Scheduler*` is specified as a prefix, the resulting unique partition name can be `ABC-Scheduler-27Sep1519514155`.

The bootOptions parameter is optional and can be NULL. Otherwise it specifies the initial boot options for the partition and typically is used when booting alternate images.

> **Important:** The returned char * value for `newpartition_id` should be freed by the caller when it is no longer needed to avoid memory leaks.

- `BGALLOC_STATUS rm_allocate_htc_pool(`

```
const unsigned int size,
const rm_job_mode_t mode,
const int psetsPerBP,
const char * user_name,
const char * caller_description,
const char * ignoreBPs,
const char * pool_id,
const char * user_list,
const char * bootOptions,
const rm_job_mode_extender_t mode_extender);
```

The caller to rm_allocate_htc_pool() provides input parameters that describe the characteristics of the pool of HTC partitions that should be created from available Blue Gene/P machine resources. If resources are available that match the requirements, a pool of partitions is created and allocated, and a return code of BGALLOC_OK is returned.

The size parameter specifies the total number of nodes to allocate for the pool. The psetsPerBP specifies the number of psets per base partition to be used. By specifying fewer psets per base partition, the I/O ratio for the allocated partition can be effectively increased. If zero is specified, use all IO nodes available. This will be the partition size used or 32, whichever is greater.

The mode and mode_extender parameters are used in conjunction to specify the HTC job mode and whether the new pool uses CNK or Linux. The mode_extender parameter specifies whether the new pool uses CNK (RM_CNK) or Linux (RM_LINUX). If the mode_extender parameter is RM_CNK, the mode parameter can be any of the job modes in Table 13-23 on page 242. If the mode_extender parameter is RM_LINUX, the mode parameter must be RM_SMP_MODE.

The user_name parameter is required. If the caller_desc value is NULL, the caller description specified on the call to rm_init_allocator is used. If the ignoreBPs parameter is not NULL, it must be a string of blank-separated base partition identifiers. The base partitions listed in the parameter are ignored as though the partitions were included in the drain list file currently in effect.

The pool_id is used as a prefix for generating unique partition names. It must be from 1 to 32 characters. Valid characters are a...z, A...Z, 0...9, -(hyphen), and _ (underscore). If the user_list parameter is not NULL, the user IDs specified are permitted to run jobs in the pool.

The bootOptions parameter is optional and can be NULL. Otherwise it specifies the initial boot options for the partition and is typically used when booting alternate images.

Multiple calls can be made to rm_allocate_htc_pool() with the same pool ID; these calls allocate additional resources to the pool. The additional resources can use different parameters such as job mode and users.

- `BGALLOC_STATUS rm_deallocate_htc_pool(`

  ```
  const unsigned int in_removed,
  const char * pool_id,
  unsigned * num_removed,
  const rm_mode_pref_t mode_pref);
  ```

  This API deallocates the specified number of nodes from a HTC pool.

  The pool_id parameter specifies the name of the pool.

  The mode_pref parameter specifies the job mode of the partitions to be deallocated from the pool. The possible values for this parameter are described in Table 15-1.

Table 15-1 rm_mode_pref_t values

Name	Description
RM_PREF_SMP_MODE	Symmetric multiprocessing mode
RM_PREF_DUAL_MODE	Dual mode
RM_PREF_VIRTUAL_NODE_MODE	Virtual node mode
RM_PREF_LINUX_SMP_MODE	Linux/SMP mode
RM_PREF_ANY_MODE	Any of the modes

The in_removed parameter specifies the number of nodes to remove from the pool. If the number of nodes to remove is not a multiple of the size of the partitions in the pool allocated using the specified mode, or if the number is greater than the number of nodes available to be removed, fewer nodes than in_removed are removed. If zero is specified, all the nodes in the pool allocated with the specified mode are deallocated.

The value returned in num_removed is the actual number of nodes removed from the pool. This number might be less than the number of nodes specified by in_removed.

15.3.2 Return codes

When a failure occurs, the API invocation returns an error code. In addition, a failure always generates a log message, which provides more information about the possible cause of the problem and an optional corrective action. These log messages are used for debugging and non-automatic recovery of failures.

The BGALLOC_STATUS return codes for the Dynamic Partition Allocator can be one of the following types:

- BGALLOC_OK: Invocation completed successfully.
- BGALLOC_ILLEGAL_INPUT: The input to the API invocation is invalid. This result is due to missing required data, illegal data, and similar problems.
- BGALLOC_ERROR: An error occurred, such as a memory allocation problem or failure on a low-level call.
- BGALLOC_NOT_FOUND: The request to dynamically create a partition failed because required resources are not available.
- BGALLOC_ALREADY_EXISTS: A partition already exists with the name specified. This error occurs only when the caller indicates a specific name for the new partition.

15.3.3 Configuring environment variables

The environment variables in Table 15-1 on page 299 are used to control the dynamic allocator and Bridge APIs.

Table 15-2 Environment variables that control the Bridge APIs

Environment variable	Required	Description
DB_PROPERTY	Yes	This variable must be set to the path of the db.properties file with database connection information. For a default installation, the path to this file is /bgsys/local/etc/db.properties.
BRIDGE_CONFIG	Yes	This variable must be set to the path of the bridge.config file that contains the Bridge APIs configuration values. For a default installation, the path to this file is /bgsys/local/etc/bridge.config.
ALLOCATOR_DRAIN_LIST	No	This variable can be set to the path of the base partition drain list to be used if one is not specified on the call to rm_init_allocator(). When this variable is not set, the file /etc/allocator_drain.lst is used as a default if it exists.
BRIDGE_DUMP_XML	No	When set to any value, this variable causes the Bridge APIs to dump in-memory XML streams to files in /tmp for debugging. When this variable is not set, the Bridge APIs do not dump in-memory XML streams.

15.4 Sample program

The sample program in Example 15-1 shows how to allocate a partition from resources on base partition R001.

Example 15-1 Sample allocator API program

```
#include <iostream>
#include <sstream>
#include <cstring>
#include "allocator_api.h"

using std::cout;
using std::cerr;
using std::endl;
```

```
int main() {
    rm_size3D_t shape;
    rm_connection_type_t conn = RM_MESH;
    char * ignoreBPs = "R00-M0";
    char* new_partition_id;
    shape.X = 0;
    shape.Y = 0;
    shape.Z = 0;
    BGALLOC_STATUS alloc_rc;

    //set lowest level of verbosity
    setSayMessageParams(stderr, MESSAGE_DEBUG1);
    alloc_rc = rm_init_allocator("test", NULL);
    alloc_rc = rm_allocate_partition(256, conn, shape, RM_SMP_MODE, 0,
                                     "user1",
                                     "New partition description",
                                     NULL,
                                     ignoreBPs,
                                     "ABC-Scheduler*",
                                     &new_partition_id, NULL);
    if (alloc_rc == BGALLOC_OK) {
        cout << "successfully allocated partition: " << new_partition_id << endl;
        free(new_partition_id);
    } else {
        cerr << "could not allocate partition: " << endl;
        if (alloc_rc == BGALLOC_ILLEGAL_INPUT) {
            cerr << "illegal input" << endl;
        } else if (alloc_rc == BGALLOC_ERROR) {
            cerr << "unknown error" << endl;
        } else if (alloc_rc == BGALLOC_NOT_FOUND) {
            cerr << "not found" << endl;
        } else if (alloc_rc == BGALLOC_ALREADY_EXISTS) {
            cerr << "partition already exists" << endl;
        } else {
            cerr << "internal error" << endl;
        }
   }
}
```

Example 15-2 shows the commands used to compile and link the sample program.

Example 15-2 compile and link commands

```
g++ -m64 -pthread -I/bgsys/drivers/ppcfloor/include -c sample1.cc -o sample1.o_64

g++ -m64 -pthread -o sample1 sample1.o_64 -L/bgsys/drivers/ppcfloor/lib64
-lbgpallocator
```

Part 5

Applications

In this part, we discuss applications that are being used on the IBM Blue Gene/L or IBM Blue Gene/P system. This part includes Chapter 16, “Performance overview of engineering and scientific applications” on page 305.

16

Performance overview of engineering and scientific applications

In this chapter, we briefly describe a series of scientific and engineering applications that are currently being used on either the Blue Gene/L or Blue Gene/P system. For a comprehensive list of applications, refer to the IBM Blue Gene Web page at:

http://www-03.ibm.com/servers/deepcomputing/bluegene/siapps.html

The examples in this chapter emphasize the benefits of using the Blue Gene supercomputer as a highly scalable parallel system. They present results for running applications in various modes that exploit the architecture of the system. We discuss the following topics:

- IBM Blue Gene/P system from an applications perspective
- Chemistry and life sciences applications

16.1 Blue Gene/P system from an applications perspective

This book has been dedicated to describing the Blue Gene/P massively parallel supercomputer from IBM. In this section, we summarize the benefits of the Blue Gene/P system from an applications point of view. At the core of the system is the IBM PowerPC (IBM PowerPC 450) processor with the addition of two floating-point units (FPU). This system uses a distributed memory, message-passing programming model.

To achieve a high level of integration and quantity of micro-processors with low power consumption, the machine was developed based on a processor with moderate frequency. The Blue Gene/P system uses system-on-a-chip (SoC) technology to allow a high level of integration, low power, and low design cost. Each processor core runs at a frequency of 850 MHz giving a theoretical peak performance of 3.4 gigaflops/core or 13.6 gigaflops/chip. The chip constitutes the Compute Node.

The next building blocks are the compute and I/O cards. A single Compute Node attached to a processor card with either 2 GB or 4 GB of memory (RAM) creates the compute and I/O cards. The compute cards and I/O cards are plugged into a node card. Two rows of sixteen compute cards are on the node card. Up to two I/O cards can be on a node card.

A midplane consists of 16 node cards stacked in a rack. A rack holds two midplanes, for a total of 32 node cards. A system with 72 racks consisting of 294,912 processor cores.

In 2005, running a real application on the Blue Gene/L system broke the barrier of 100 teraflops/second (TF/s), sustaining performance using the domain decomposition molecular-dynamics code (ddcMD) from the Lawrence Livermore National Laboratory.[34] In 2006, the first system to break the barrier of 200 TF/s was Qbox running at 207.3 TF/s.[35] Real applications are currently achieving two orders of magnitude higher performance than previously possible. Successful scaling has pushed from O(1000) processors to O(100,000) processors by the Gordon Bell Prize finalists at Supercomputing 2006.[36] Out of six finalists, three ran on the Blue Gene/L system.

In silico experimentation plays a crucial role in many scientific disciplines. It provides a fingerprint for experimentation. In engineering applications, such as automotive crash studies, numerical simulation is much cheaper than physical experimentation. In other applications, such as global climate change where experiments are impossible, simulations are used to explore the fundamental scientific issues.[37] This is certainly true in life sciences as well as in materials science. Figure 16-1 on page 307 illustrates a landscape of a few selected areas and techniques where High-Performance Computing is important to carry out simulations.

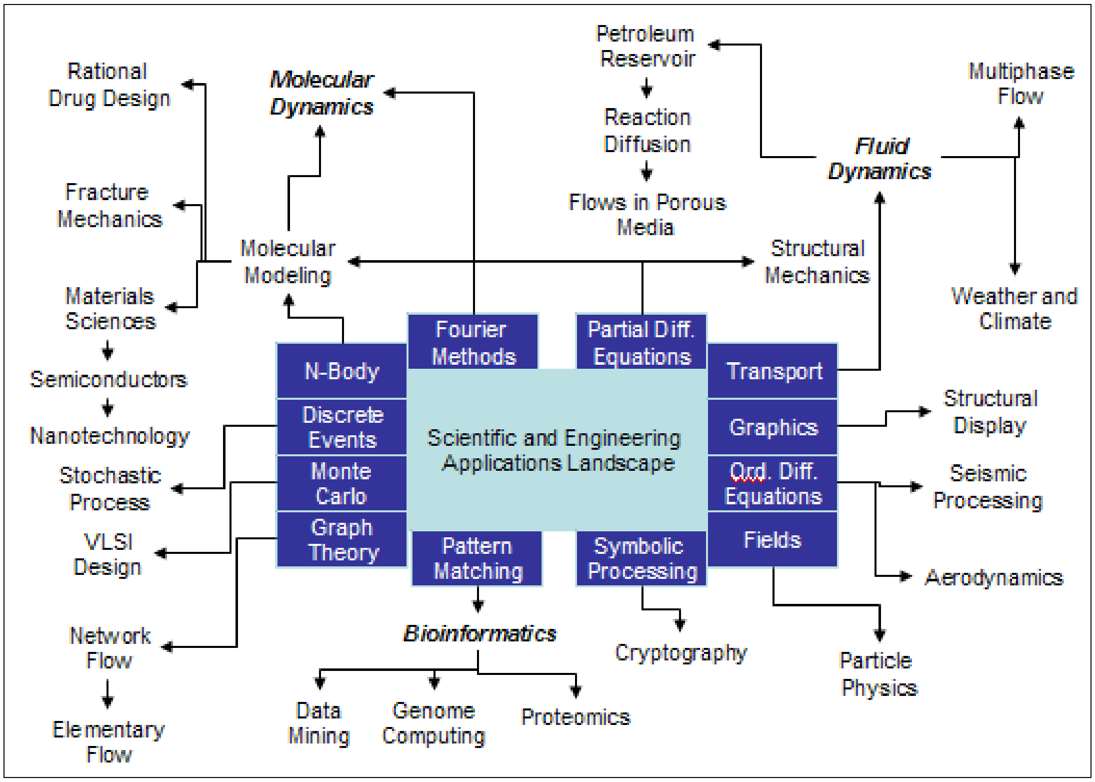


Figure 16-1 High-performance computing landscape for selected scientific and engineering applications

In the rest of this chapter, we summarize the performance that has been recorded in the literature for a series of applications in life sciences and materials science. A comprehensive list of applications is available for the Blue Gene/L and Blue Gene/P systems. For more information, see the IBM Blue Gene Applications Web page at:

http://www-03.ibm.com/servers/deepcomputing/bluegene/siapps.html

16.2 Chemistry and life sciences applications

In this section, we provide a brief overview of the performance characteristics of a selected set of chemistry and life sciences applications. In particular, we focus on what is known as *computational chemistry*. However, as other disciplines in sciences that traditionally relied almost exclusively on experimental observation began to fully incorporate Information Technology (IT) as one of their tools, the area of computational chemistry has expanded to new disciplines such as bioinformatics, systems biology, and several other areas that have emerged after the post-genomic era.

To understand or define the kind of molecular systems that can be studied with these techniques, Figure 16-2 on page 308 defines the computational chemistry landscape as a function of the size of the systems and the methodology. It illustrates that Classical Molecular Mechanics/Molecular Dynamics (MM/MD) are commonly used to simulate large biomolecules that cannot be treated with more accurate methods. The next level corresponds to semi-empirical methods. Finally *Ab Initio* methods (also called *electronic structure methods*) provide a more accurate description of the system, but the computational demands in terms of compute cycles increase rapidly.

Alternatively, bioinformatics techniques rely mainly on string manipulations in an effort to carry out data mining of large databases. These applications tend to be data intensive.

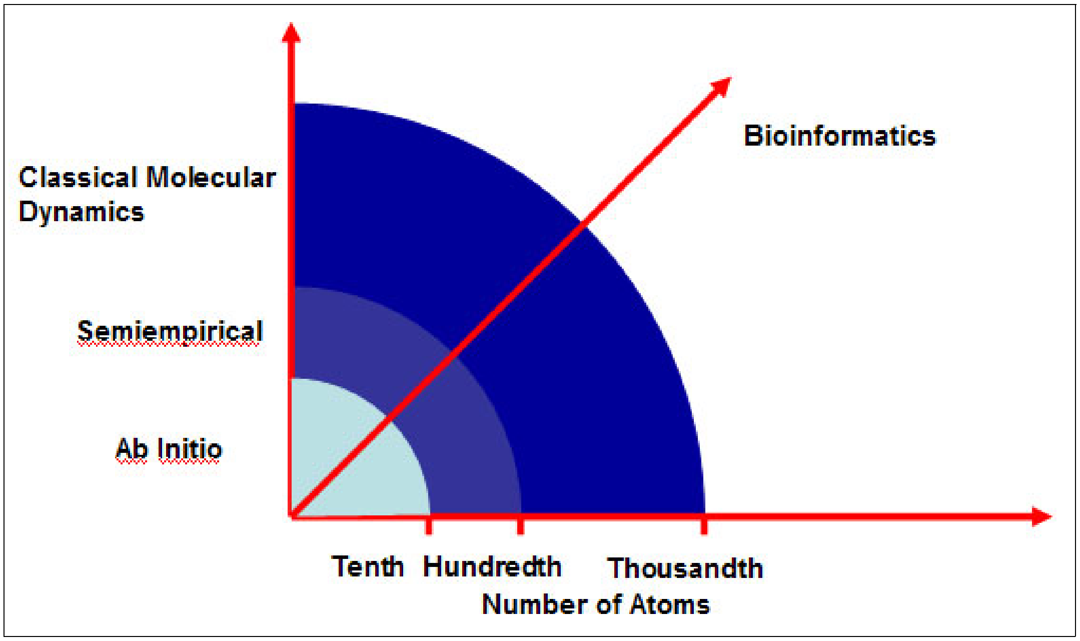


Figure 16-2 Computational methods landscape in computational chemistry

Although Density Functional Theory-based approaches are not fully represented in Figure 16-2, nowadays these types of methods are being used to simulate biologically important systems.[38] These techniques allow for the calculation of larger systems. In this chapter, we briefly describe Car-Parrinello Molecular Dynamics (CPMD).[39] In the same vein, use of mixed Quantum Mechanical/Molecular Mechanical (QM/MM) methods[40] can simulate larger systems.

16.2.1 Classical molecular mechanics and molecular dynamics applications

Applications in such areas as chemistry and life sciences can benefit from the type of architecture used in the Blue Gene supercomputer.[41] In particular, software packages based on molecular dynamics have been considered good candidates for the Blue Gene architecture. Classical MD simulations compute atomic trajectories by solving equations of motion numerically by using empirical force fields. The overall MD energy equation is broken into three components: bonded, van der Waals, and electrostatic. The first two components are local in nature and therefore do not make a significant contribution to the overall running time.

The quadratic scaling of the electrostatics force terms, however, requires a high level of optimization of the MD application.[42] To improve performance on simulations in which the solvent is modeled at the atomic level (that is, explicit solvent modeling), the four Blue Gene MD applications of AMBER,[43] Blue Matter,[44] LAMMPS,[45] and NAMD[46] employ a reciprocal-space technique called *Ewald sums*, which enables the evaluation of long-range electrostatic forces to a preselected level of accuracy. In addition to the *particle mesh Ewald (PME) method*, LAMMPS offers the particle particle/particle-mesh (PPPM) technique with characteristics that make it scale well on massively parallel processing (MPP) machines such as the Blue Gene system.

AMBER

AMBER[47] is the collective name for a suite of programs that are developed by the Scripps Research Institute. With these programs, users can carry out molecular dynamics simulations, particularly on biomolecules. The primary AMBER module, called *sander*, was designed to run on parallel systems and provides direct support for several force fields for proteins and nucleic acids. AMBER includes an extensively modified version of sander, called *pmemd* (particle mesh). For complete information about AMBER as well as benchmarks, refer to the AMBER Web site at:

http://amber.scripps.edu/

For implicit solvent (continuum) models, which rely on variations of the Poisson equation of classical electrostatics, AMBER offers the Generalized Born (GB) method. This method uses an approximation to the Poisson equation that can be solved analytically and allows for good scaling. In Figure 16-3, the experiment is with an implicit solvent (GB) model of 120,000 atoms (Aon benchmark).

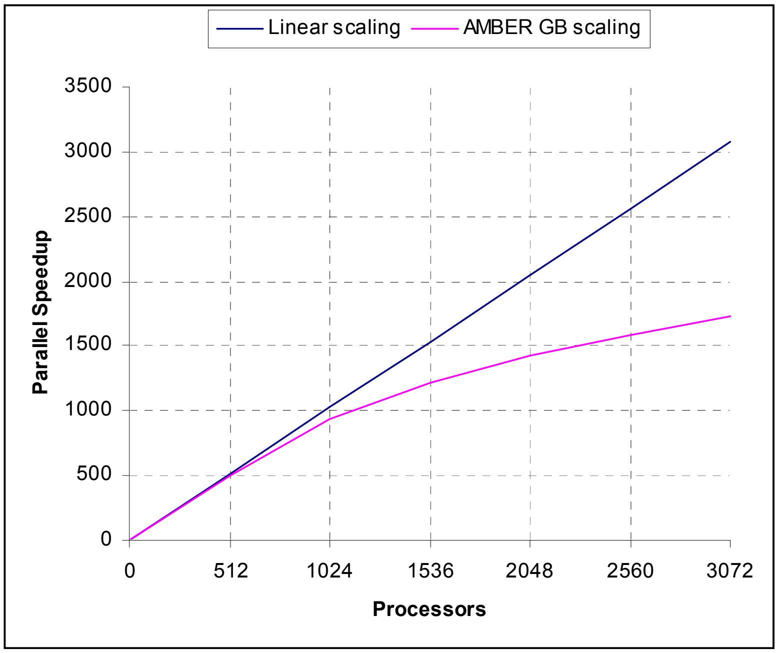


Figure 16-3 Parallel scaling of AMBER on the IBM Blue Gene/L system

AMBER also incorporates the PME algorithm, which takes the full electrostatic interactions into account to improve the performance of electrostatic force evaluation (see Figure 16-4). In Figure 16-4, the experiment is with an explicit solvent (PME) model of 290,000 atoms (Rubisco).

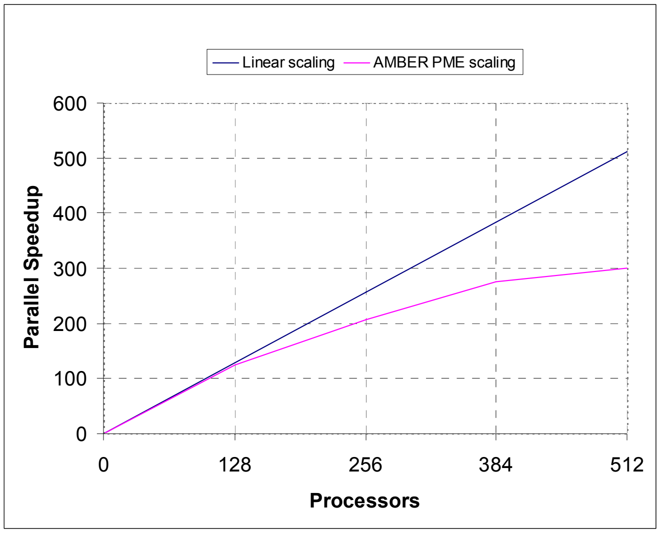


Figure 16-4 Parallel scaling of AMBER on the Blue Gene/L system

Blue Matter

Blue Matter[48] is a classical molecular dynamics application that has been under development as part of the IBM Blue Gene project. The effort serves two purposes:

- Enables scientific work in the area of biomolecular simulation that IBM announced in December 1999.
- Acts as an experimental platform for the exploration of programming models and algorithms for massively parallel machines in the context of a real application.

Blue Matter has been implemented via spatial-force decomposition for N-body simulations using the PME method for handling electrostatic interactions. The Ewald summation method and particle mesh techniques are approximated by a finite range cut-off and a reciprocal space portion for the charge distribution. This is done in Blue Matter via the Particle-Particle-Particle-Mesh (P3ME) method.[49]

The results presented by Fitch et al.[50] show impressive scalability on the Blue Gene/L system. Figure 16-5 on page 311 shows scalability as a function of the number of nodes. It illustrates that the performance in time/time step as a function of the number of processors for β-Hairpin contains a total of 5,239 atoms. SOPE contains 13,758 atoms. In this case, the timings that are reported here correspond to a size of 64^3 FFT. Rhodopsin contains 43,222 atoms, and ApoA1 contains 92,224 atoms. All runs were carried out using the P3ME method, which was implemented in Blue Matter at constant particle number, volume, and energy (NVE).[52]

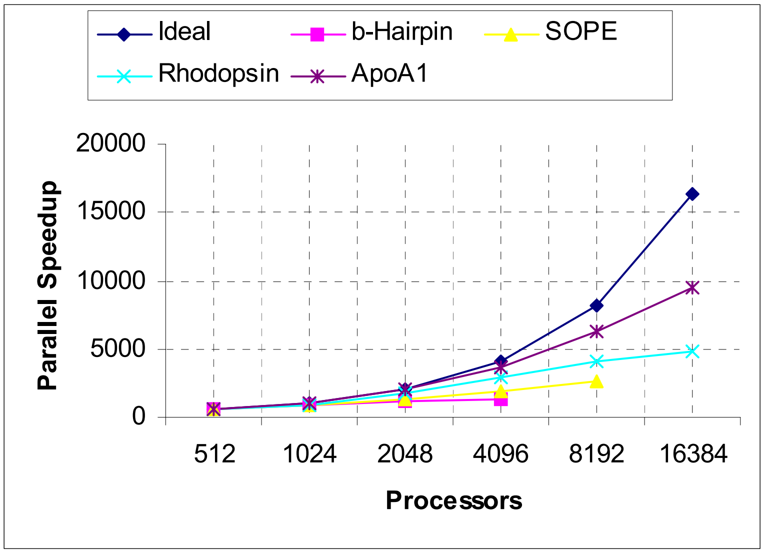


Figure 16-5 Performance in time/time step as a function of number of processors (from Fitch, et al.[51])

LAMMPS

Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS)[53] is an MD program from Sandia National Laboratories that is designed specifically for MPP. LAMMPS is implemented in C++ and is distributed freely as open-source software under the GNU Public License (GPL).[54] LAMMPS can model atomic, polymeric, biological, metallic, or granular systems using a variety of force fields and boundary conditions. The parallel efficiency of LAMMPS varies from the size of the benchmark data and the number of steps being simulated. In general, LAMMPS can scale to more processors on larger systems (see Figure 16-6).

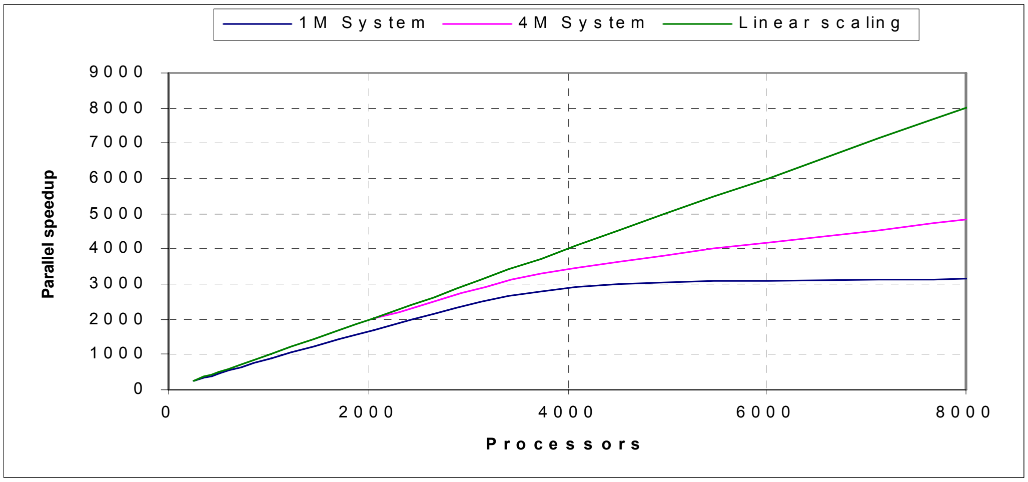


Figure 16-6 Parallel scaling of LAMMPS on Blue Gene/L (1M System: 1-million atom scaled rhodopsin; 4M System: 4-million atom scaled rhodopsin)

For a one-million atom system, LAMMPS can scale up to 4096 nodes. For a larger system, such as a four-million atom system, LAMMPS can scale up to 4096 nodes as well. As the size of the system increases, scalability increases as well.

NAMD

NAMD is a parallel molecular dynamics application that was developed for high-performance calculations of large biological molecular systems.[55] NAMD supports the force fields used by AMBER, CHARMM,[56] and X-PLOR[57] and is also file compatible with these programs. This commonality allows simulations to migrate between these four programs. The C++ source for NAMD and Charm++ are freely available from UIUC. For additional information about NAMD, see the official NAMD Web site at:

http://www.ks.uiuc.edu/Research/namd/

NAMD incorporates the PME algorithm, which takes the full electrostatic interactions into account and reduces computational complexity. To further reduce the cost of the evaluation of long-range electrostatic forces, a multiple time step scheme is employed. The local interactions (bonded, van der Waals, and electrostatic interactions within a specified distance) are calculated at each time step. The longer range interactions (electrostatic interactions beyond the specified distance) are computed less often. An incremental load balancer monitors and adjusts the load during the simulation.

Due to the good balance of network and processor speed of the Blue Gene system, NAMD is able to scale to large processor counts (see Figure 16-7). While scalability is affected by many factors, many simulations can make use of multiple Blue Gene racks. Work by Kumar et al.[58] has reported scaling up to 8192 processors. Timing comparisons often use the "benchmark time" metric instead of wall clock time to completion. The benchmark time metric omits setup, I/O, and load balance overhead. While benchmark scaling can be considered a guide to what is possible, ideal load balance and I/O parameters for each case must be found for the wall clock time to scale similarly. Careful consideration of these parameters might be necessary to achieve the best scalability.

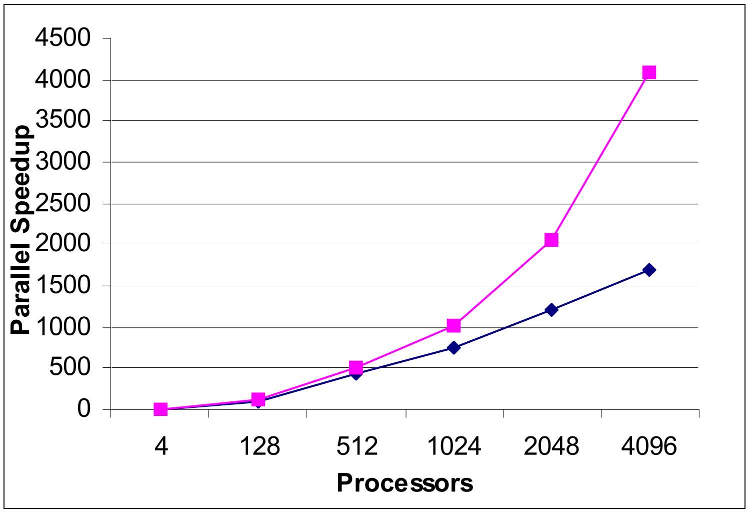


Figure 16-7 Parallel speedup on the Blue Gene/L system for the NAMD standard apoA1 benchmark

16.2.2 Molecular docking applications

Applications in the area of molecular docking are becoming important in High-Performance Computing. In particular, in *silico screening* using molecular docking has been recognized as an approach that benefits from High-Performance Computing to identify novel small molecules that can then be used for drug design.[59] This process consists of the identification or selection of compounds that show activity against a biomolecule that is of interest as a drug target.[60]

Docking programs place molecules into the active site of the receptor (or target biomolecule) in a noncovalent fashion and then rank them by the ability of the small molecules to interact with the receptor.[61] An extensive family of molecular docking software packages is available.[62]

DOCK is an open-source molecular docking software package that is frequently used in structure-based drug design.[63] The computational aspects of this program can be divided into two parts. The first part consists of the *ligand atoms* located inside the cavity or binding pocket of a receptor, which is a large biomolecule. This step is carried out by a search algorithm.[64] The second part corresponds to scoring or identifying the most favorable interactions, which is normally done by means of a scoring function.[65]

The latest version of the DOCK software package is Version 6.1. However, in our work, we used Version 6.0. This version is written in C++ to exploit code modularity and has been parallelized using the Message Passing Interface (MPI) paradigm. DOCK V6.0 is parallelized using a master-worker scheme.[66] The master handles I/O and tasks management, while each worker is given an individual molecule to perform simultaneous independent docking.[67]

Recently, Peters, et al. have shown that DOCK6 is well suited for doing virtual screening on the Blue Gene/L or Blue Gene/P system.[68] Figure 16-8 shows the receptor HIV-1 reverse transcriptase in complex with nevirapine as used and described in the Official UCSF DOCK Web site. The ligand library corresponds to a subset of 27,005 drug-like ligands from the ZINC database.[69] The scalability of the parallel version of the code is illustrated by constructing a set of ligands with 128,000 copies of nevirapine as recommended in the Official UCSF DOCK Web site to remove dependence on the order and size of the compound. You can find this Web site at:

http://dock.compbio.ucsf.edu

In Figure 16-8, the original code is the dark bar. Sorting by total number of atoms per ligand is represented by the bar with horizontal lines. Sorting by total number of rotatable bonds per ligand is represented by the white bar.[70]

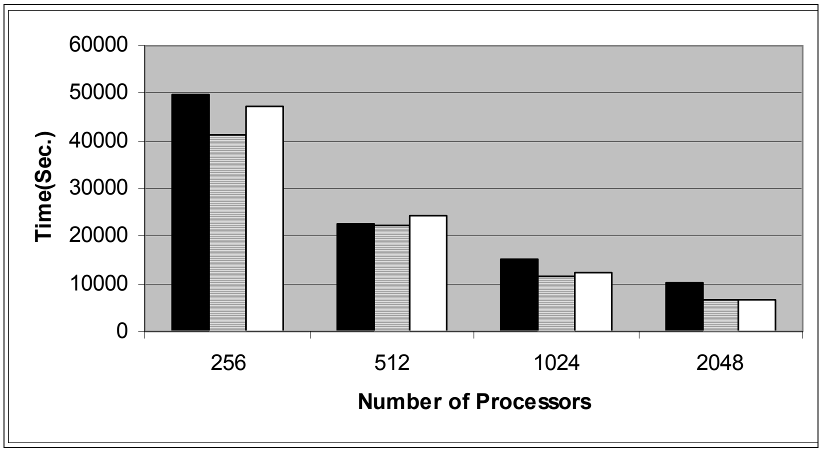


Figure 16-8 The effect of load-balancing optimization for 27,005 ligands on 2048 processors

16.2.3 Electronic structure (Ab Initio) applications

Electronic structure calculations, such as the Hartree-Fock (HF) method, represent one of the simplest techniques in this area. However, even this first approximation tends to be computationally demanding. Many types of calculations begin with a Hartree-Fock calculation and subsequently correct for electron-electron repulsion, which is also referred to as *electronic correlation*. The Møller-Plesset perturbation theory (MPn) and coupled cluster theory (CC) are examples of these post-Hartree-Fock methods.[71]

A common characteristic of these techniques is that they are used to accurately compute molecular properties. As such, they tend to be widely available in High-Performance Computing. However, in addition to traditional electronic structure methods, Density Functional Theory-based methods have proven to be an attractive alternative to include correction effects and still treat large systems.

The CPMD code is based on the original computer code written by Car and Parrinello.[72] It was developed first at the IBM Research Zurich laboratory, in collaboration with many groups worldwide. It is a production code with many unique features written in Fortran and has grown from its original size of approximately 10,000 lines to currently close to 200,000 lines of code. Since January 2002, the program has been freely available for noncommercial use.[73]

The basics of the implementation of the Kohn-Sham method using a plane-wave basis set and pseudopotentials are described in several review articles,[74] and the CPMD code follows them closely. All standard gradient-corrected density functionals are supported, and preliminary support for functionals that depend on the kinetic energy density is available. Pseudopotentials used in CPMD are either of the norm-conserving or the ultra-soft type.[75] Norm-conserving pseudopotentials have been the default method in CPMD, and only some of the rich functionality has been implemented for ultra-soft pseudopotentials.

The emphasis of CPMD on MD simulations of complex structures and liquids led to the optimization of the code for large supercells and a single k-point (the k = 0 point) approximation. Therefore, many features have only been implemented for this special case. CPMD has a rich set of features, many of which are unique. For a complete overview, refer to the CPMD manual.[76] The basic electronic structure method implemented uses fixed occupation numbers, either within a spin-restricted or an unrestricted scheme. For systems with a variable occupation number (small gap systems and metals), the free energy functional3 can be used together with iterative diagonalization methods.

16.2.4 Bioinformatics applications

The list of molecular biology databases is constantly increasing, and more scientists rely on this information. The NAR Molecular Biology Database collection reported an increase of 139 more databases for 2006 compared to the previous year. enBank doubles its size approximately every 18 months. However, the increase in microprocessor clock speed is not changing at the same rate. Therefore, scientists try to leverage the use of multiple processors. In this section, we introduce some of the applications currently running on the Blue Gene supercomputer.

HMMER

For a complete discussion of hidden Markov models, refer to the work by Krogh et al.[77] HMMER V2.3.2 consists of nine different programs: hmmalign, hmmbuild, hmmcalibrate, hmmconvert, hmmemit, hmmfetch, hmmindex, hmmpfam, and hmmsearch.[78] Out of these nine programs, hmmcalibrate, hmmpfam, and hmmsearch have been parallelized. hmmcalibrate is used to identify statistical significance parameters for profile HMM. hmmpfam is used to search a profile HMM database, and hmmsearch is used to carry out sequence database searches.[79]

The first module tested corresponds to hmmcalibrate. Figure 16-9 summarizes the performance of this module up to 2048 nodes.[80] Although this module was not optimized, the parallel efficiency is still 75% on 2048 nodes. The graph in Figure 16-9 illustrates the performance of hmmcalibrate using only the first 327 entries in the Pfam database.[81]

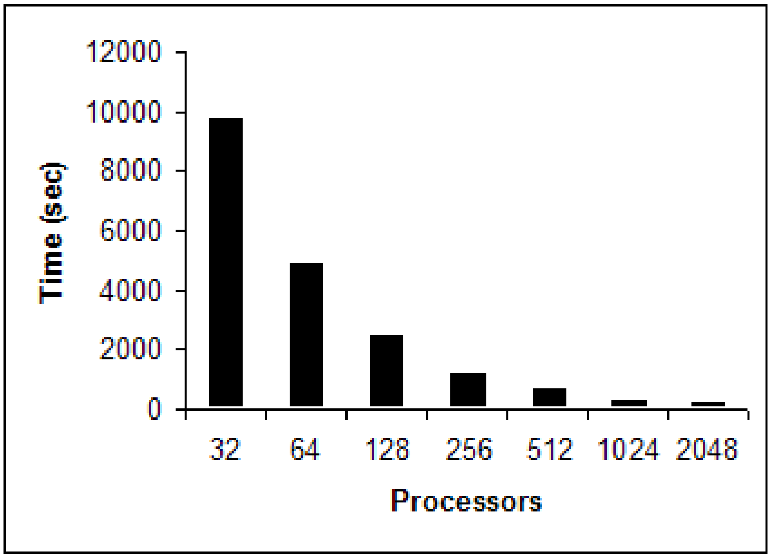


Figure 16-9 .hmmcalibrate parallel performance using the first 327 entries of the Pfam database

Figure 16-10 on page 316 illustrates the work presented by Jiang, et al.[80] for optimizing hmmsearch parallel performance using 50 proteins of the globin family from different organisms and the UniProt release 8 database. For each processor count, the left bar shows the original PVM to MPI port. Notice scaling stops at 64 nodes. The second bar shows the multiple master implementation. The third bar shows the dynamic data collection implementation, and the right bar shows the load balancing implementation.

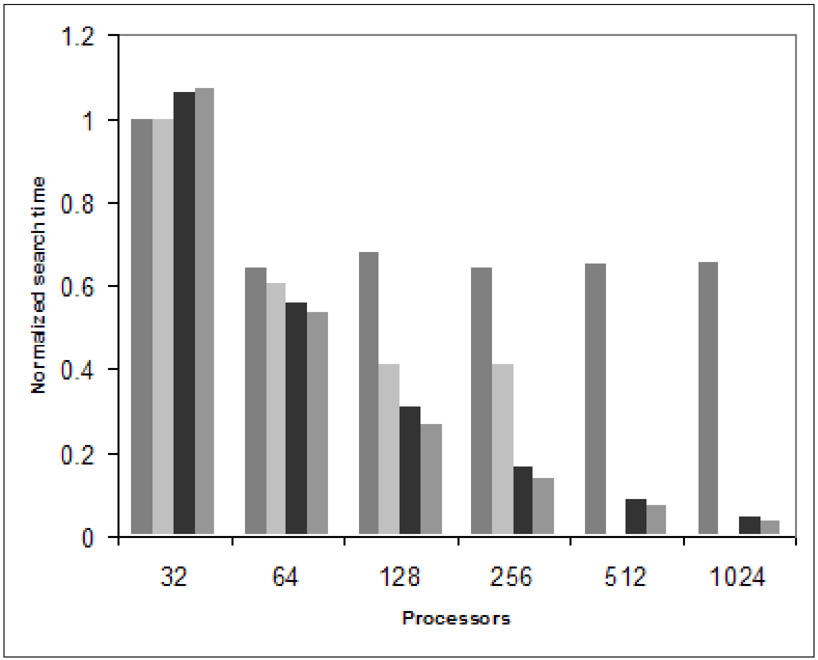


Figure 16-10 hmmsearch parallel performance

mpiBLAST-PIO

mpiBLAST is an open-source parallelization of BLAST that uses MPI.[83] One of the key features of the initial parallelization of mpiBLAST is its ability to fragment and distribute databases.

Thorsen et al.[84] have compared the query Arabidopsis thaliana, a model organism for studying plant genetics. This query was further subdivided into small, medium, and large query sets that contain 200, 1168, and 28014 sequences, respectively.

Figure 16-11 on page 317 illustrates the results of comparing three queries of three different sizes. We labeled them “small,” “medium,” and “large.” The database corresponds to NR. This figure shows that scalability is a function of the query size. The small query scales to approximately 1024 nodes in coprocessor mode with a parallel efficiency of 72% where the large query scales to 8,192 nodes with a parallel efficiency of 74%.

From the top of Figure 16-11, the thick solid line corresponds to ideal scaling. The thin solid line corresponds to the large query. The dashed line corresponds to the medium query. The dotted line corresponds to the small query.

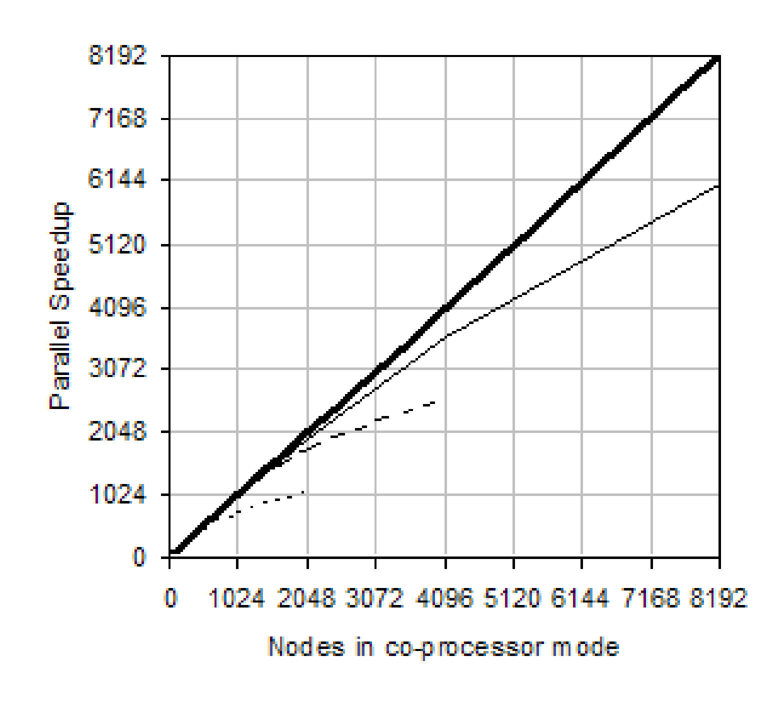


Figure 16-11 Scaling chart for queries run versus the nr database

16.2.5 Performance kernel benchmarks

Communication performance is an important aspect when running parallel applications, particularly, when running on a distributed-memory system such as the Blue Gene/P system. On both the Blue Gene/L and Blue Gene/P systems, instead of implementing a single type of network capable of transporting all protocols needed, these two systems have separate networks for different types of communications.

Usually two measurements provide information about the network and can be used to look at the parallel performance of applications:

Bandwidth The number of MB of data that can be sent from a node to another node in one second

Latency The amount of time it takes for the first byte sent from one node to reach its target node

These two values provide information about communication. In this section, we illustrate two simple cases. The first case corresponds to a benchmark that involves a single transfer. The second case corresponds to a collective as defined in the Intel MPI Benchmarks. Intel MPI Benchmarks is formerly known as “Pallas MPI Benchmarks” - PMB-MPI1 (for MPI1 standard functions only). Intel MPI Benchmarks - MPI1 provides a set of elementary MPI benchmark kernels.

For more details, see the product documentation included in the package that you can download from the Web at:

`http://www.intel.com/cd/software/products/asmo-na/eng/219848.htm`

The Intel MPI Benchmarks kernel or elementary set of benchmarks was reported as part of *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686. Here we describe and perform the same benchmarks. You can run all of the supported benchmarks, or just a subset, specified through the command line. The rules, such as time measurement, message lengths, selection of communicators to run a particular benchmark, are program parameters. For more information, see the product documentation that is included in the package, which you can download from the Web at:

http://www.intel.com/software/products/cluster/mpi/mpi_benchmarks_lic.htm

This set of benchmarks has the following objectives:

- Provide a concise set of benchmarks targeted at measuring important MPI functions: point-to-point message-passing, global data movement and computation routines, and one-sided communications and file I/O
- Set forth precise benchmark procedures: run rules, set of required results, repetition factors, and message lengths
- Avoid imposing an interpretation on the measured results: execution time, throughput, and global operations performance

16.2.6 MPI point-to-point

In the Intel MPI Benchmarks, single transfer corresponds to PingPong and PingPing benchmarks. Here we illustrate a comparison between the Blue Gene/L and Blue Gene/P system for the case of PingPong. This benchmark illustrates a single message that was transferred between two MPI tasks, which in our case, is on two different nodes.

To run this benchmark, we used the Intel MPI Benchmark Suite Version 2.3, MPI-1 part. On the Blue Gene/L system, the benchmark was run in coprocessor mode, which is defined in *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686. On the Blue Gene/P system, we used the SMP Node mode.

Example 16-1 shows how **mpirun** was invoked on the Blue Gene/L system.

Example 16-1 mpirun on the Blue Gene/L system

```
mpirun -nofree -timeout 120 -verbose 1 -mode CO -env "BGL_APP_L1_WRITE_THROUGH=0
BGL_APP_L1_SWOA=0" -partition R000 -cwd /bglscratch/pallas -exe
/bglscratch/pallas/IMB-MPI1.4MB.perf.rts -args "-msglen 4194304.txt -npmin 512
PingPong" | tee IMB-MPI1.4MB.perf.PingPong.4194304.512.out) >>
run.IMB-MPI1.4MB.perf.PingPong.4194304.512.out 2>&1
```

Example 16-2 shows how **mpirun** was invoked on the Blue Gene/P system.

Example 16-2 mpirun on the Blue Gene/P system

```
mpirun -nofree -timeout 300 -verbose 1 -np 512 -mode SMP  -partition R01-M1 -cwd
/bgusr/BGTH_BGP/test512nDD2BGP/pallas/pall512DD2SMP/bgpdd2sys1-R01-M1 -exe
/bgusr/BGTH_BGP/test512nDD2BGP/pallas/pall512DD2SMP/bgpdd2sys1-R01-M1/IMB-MPI1.4MB
.perf.rts -args "-msglen 4194304.txt -npmin 512 PingPong" | tee
IMB-MPI1.4MB.perf.PingPong.4194304.512.out) >>
run.IMB-MPI1.4MB.perf.PingPong.4194304.512.out 2>&1
```

Figure 16-12 on page 319 shows the bandwidth on the torus network as a function of the message size, for one simultaneous pair of nearest neighbor communications. The protocol switch from short to eager is visible in these two cases, where the eager to rendezvous switch is most pronounced on the Blue Gene/L system. This figure also shows the improved performance on the Blue Gene/P system. Notice also in Figure 16-12 that the diamonds corresponds to the Blue Gene/P system and the asterisks (*) correspond to the Blue Gene/L system.

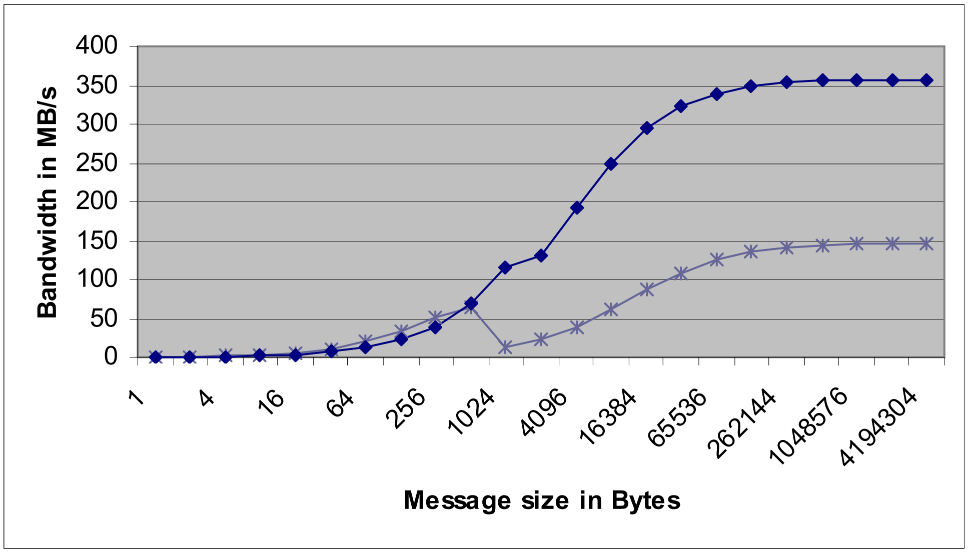


Figure 16-12 Bandwidth versus message size

16.2.7 MPI collective benchmarks

In the Intel MPI Benchmarks, collective benchmarks correspond to Bcast, Allgather, Allgatherv, Alltoall, Alltoallv, Reduce, Reduce_scatter, Allreduce, and Barrier benchmarks. Here we illustrate a comparison between the Blue Gene/L and Blue Gene/P system for the case of Allreduce, which is a popular collective used in certain scientific applications. These benchmarks measure the message-passing power of a system as well as the quality of the implementation.

To run this benchmark, we used the Intel MPI Benchmark Suite Version 2.3, MPI-1 part. On the Blue Gene/P system, the benchmark was run in coprocessor mode, which is defined in *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686. On the Blue Gene/P system, we used SMP Node mode.

Example 16-3 shows how **mpirun** was invoked on the Blue Gene/L system.

Example 16-3 mpirun on the Blue Gene/L system

```
mpirun -nofree -timeout 120 -verbose 1 -mode CO -env "BGL_APP_L1_WRITE_THROUGH=0
BGL_APP_L1_SWOA=0" -partition R000 -cwd
/bglscratch/BGTH/testsmall512nodeBGL/pallas -exe
/bglscratch/BGTH/testsmall512nodeBGL/pallas/IMB-MPI1.4MB.perf.rts -args "-msglen
4194304.txt -npmin 512 Allreduce" | tee
IMB-MPI1.4MB.perf.Allreduce.4194304.512.out) >>
run.IMB-MPI1.4MB.perf.Allreduce.4194304.512.out 2>&1
```

Example 16-4 shows how **mpirun** was invoked on the Blue Gene/P system.

Example 16-4 mpirun on the Blue Gene/P system

```
mpirun -nofree -timeout 300 -verbose 1 -np 512 -mode SMP  -partition R01-M1 -cwd
/bgusr/BGTH_BGP/test512nDD2BGP/pallas/pall512DD2SMP/bgpdd2sys1-R01-M1 -exe
```

```
/bgusr/BGTH_BGP/test512nDD2BGP/pallas/pall512DD2SMP/bgpdd2sys1-R01-M1/IMB-MPI1.4MB
.perf.rts -args "-msglen 4194304.txt -npmin 512 Allreduce" | tee
IMB-MPI1.4MB.perf.Allreduce.4194304.512.out) >>
run.IMB-MPI1.4MB.perf.Allreduce.4194304.512.out 2>&1
```

Collective operations are more efficient on the Blue Gene/P system. You should try to use these operations instead of point-to-point communication wherever possible. The overhead for point-to-point communications is much larger than those for collectives. Unless all your point-to-point communication is purely the nearest neighbor, it is also difficult to avoid network congestion on the torus network.

Alternatively, collective operations can use the barrier (global interrupt) network or the torus network. If they run over the torus network, they can still be optimized by using specially designed communication patterns that achieve optimum performance. Doing this manually with point-to-point operations is possible in theory, but in general, the implementation in the Blue Gene/P MPI library offers superior performance.

With point-to-point communication, the goal of reducing the point-to-point Manhattan distances necessitates a good mapping of MPI tasks to the physical hardware. For collectives, mapping is equally important because most collective implementations prefer certain communicator shapes to achieve optimum performance. The technique of mapping is illustrated in Appendix F, “Mapping” on page 355.

Similar to point-to-point communications, collective communications also works best if you do not use complicated derived data types and if your buffers are aligned to 16-byte boundaries.

While the MPI standard explicitly allows for MPI collective communications to occur at the same time as point-to-point communications (on the same communicator), we generally do not recommend that you allow this to happen for performance reasons.

Table 16-1 summarizes the MPI collectives that have been optimized on the Blue Gene/P system, together with their performance characteristics when executed on the various networks of the Blue Gene/P system.

Table 16-1 MPI collectives optimized on the Blue Gene/P system

MPI routine	Condition	Network	Performance
MPI_Barrier	MPI_COMM_WORLD	Barrier (global interrupt) network	1.2 µs
MPI_Barrier	Any communicator	Torus network	30 µs
MPI_Broadcast	MPI_COMM_WORLD	Collective network	817 MBps
MPI_Broadcast	Rectangular communicator	Torus network	934 MBps
MPI_Allreduce	MPI_COMM_WORLD fixed-point	Collective network	778 MBps
MPI_Allreduce	MPI_COMM_WORLD floating point	Collective network	98 MBps
MPI_Alltoall[v]	Any communicator	Torus network	84-97% peak
MPI_Allgatherv	N/A	Torus network	Same as broadcast

Figure 16-13 shows a comparison between the Blue Gene/L and Blue Gene/P systems for the MPI_Allreduce() type of communication.

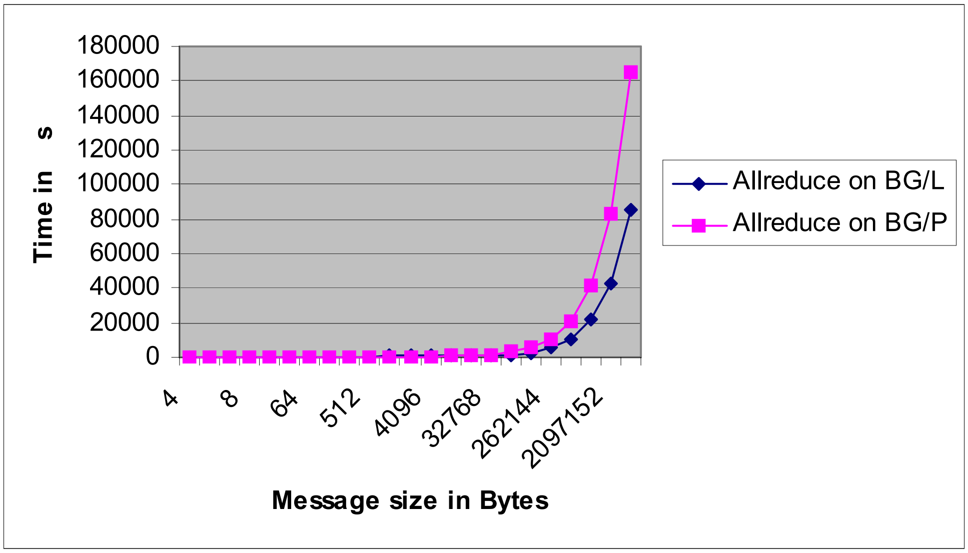


Figure 16-13 MPI_Allreduce() performance on 512 nodes

Figure 16-14 illustrates the performance of the barrier on Blue Gene/P for up to 32 nodes.

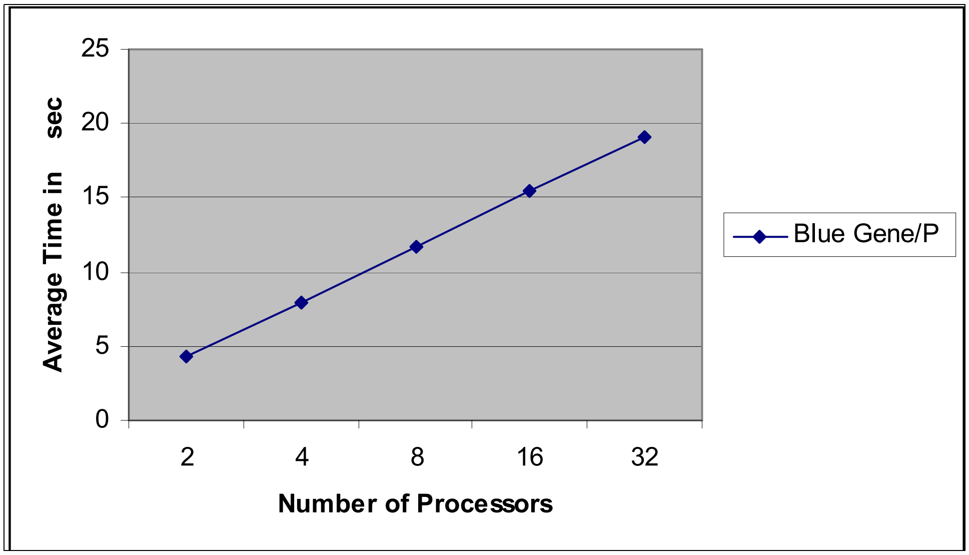


Figure 16-14 Barrier performance on the Blue Gene/P system

Part 6

Appendixes

In this part, we provide additional information about system administration for the IBM Blue Gene/P system. This part includes the following appendixes:

- Appendix A, “Blue Gene/P hardware-naming conventions” on page 325
- Appendix B, “Files on architectural features” on page 331
- Appendix C, “Header files and libraries” on page 335
- Appendix D, “Environment variables” on page 339
- Appendix E, “Porting applications” on page 353
- Appendix F, “Mapping” on page 355
- Appendix G, “htcpartition” on page 359
- Appendix H, “Use of GNU profiling tool on Blue Gene/P” on page 361
- Appendix I, “Statement of completion” on page 365

A

Blue Gene/P hardware-naming conventions

In this appendix, we present an overview of how the IBM Blue Gene/P hardware locations are assigned. These naming conventions are used consistently throughout both hardware and software.

Figure A-1 shows the conventions used when assigning locations to all hardware except the various cards in a Blue Gene/P system. Using the charts and diagrams that follow, consider an example where you have an error in the fan named R23-M1-A3-0. This naming convention tells you where to look for the error. In the upper-left corner of Figure A-1, you see that racks use the convention Rxx. Looking at our error message, we can see that the rack involved is R23. From the chart in Figure A-1, we see that R23 is the fourth rack in row two. (Remember that all numbering starts with 0). The bottom midplane of any rack is 0. Therefore, we are dealing with the top midplane (R23-M1).

In the chart, you can see in the fan assemblies description that assemblies 0-4 are on the front of the rack, bottom to top, respectively. Therefore, we check for an attention light (Amber LED) on the fan assembly second from the top, because the front-most fan is the one that is causing the error message to surface. Service, link, and node cards use a similar form of addressing.

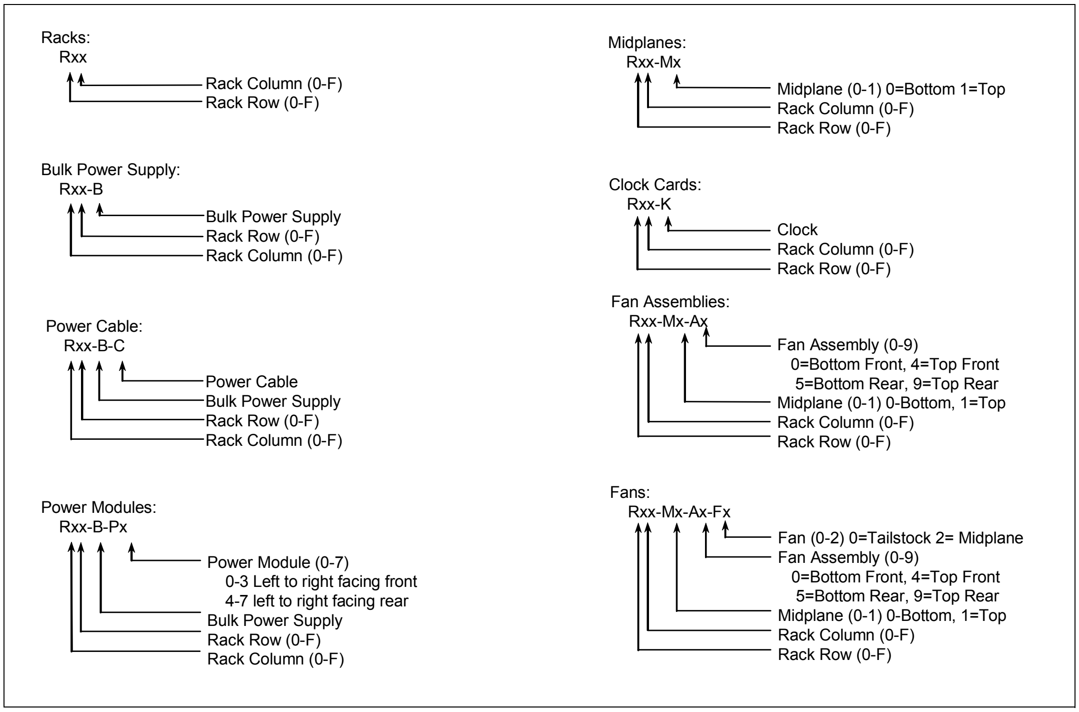


Figure A-1 Hardware-naming conventions

Figure A-2 shows the conventions used for the various card locations.

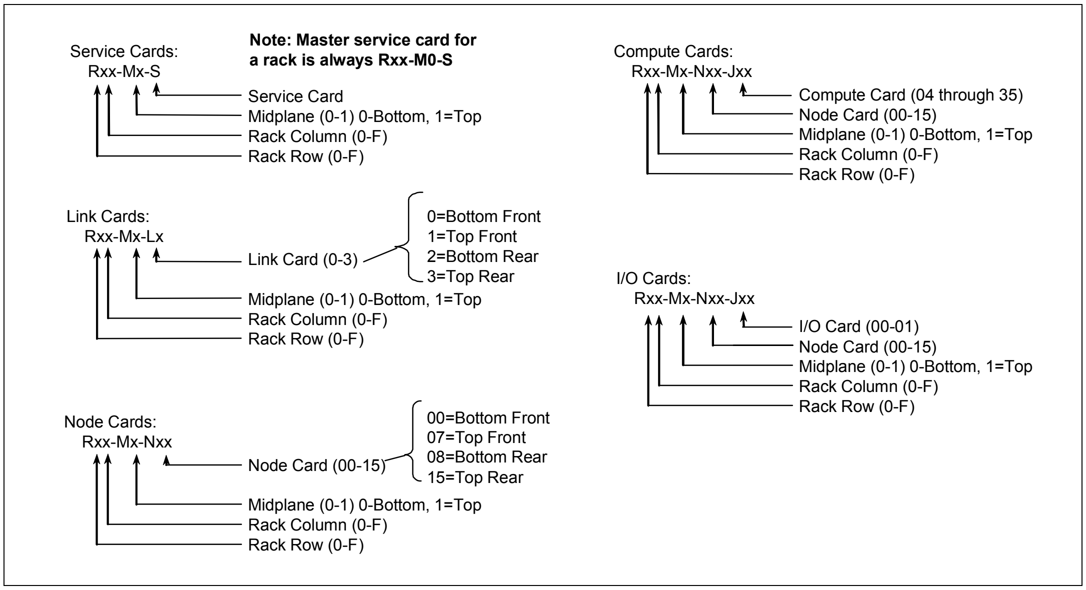


Figure A-2 Card-naming conventions

Table A-1 contains examples of various hardware conventions. The figures that follow the table provide illustrations of the actual hardware.

Table A-1 Examples of hardware-naming conventions

Card	Element	Name	Example
Compute	Card	J04 through J35	R23-M10-N02-J09
I/O	Card	J00 through J01	R57-M1-N04-J00
I/O & Compute	Module	U00	R23-M0-N13-J08-U00
Link	Module	U00 through U05 (00 leftmost, 05 rightmost)	R32-M0-L2_U03
Link	Port	TA through TF	R01-M0-L1-U02-TC
Link data cable	Connector	J00 through J15 (as labeled on link card)	R21-M1-L2-J13
Node Ethernet	Connector	EN0, EN1	R16-M1-N14-EN1
Service	Connector	Control FPGA, control network, Clock R, Clock B	R05-M0-S-Control FPGA
Clock	Connector	Input, Output 0 through Output 9	R13-K- Output 3

Figure A-3 shows the layout of a 64-rack system.

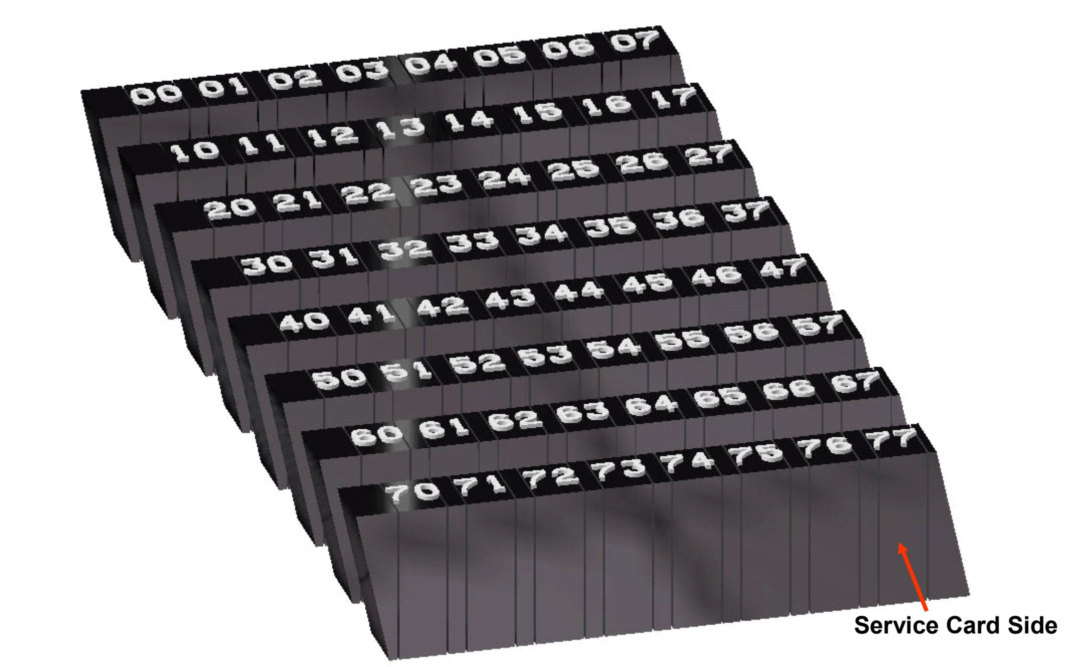


Figure A-3 Rack numbering

Note: The fact that Figure A-3 shows numbers 00 through 77 does not imply that this configuration is the largest possible. The largest configuration possible is 256 racks numbered 00 through FF.

Figure A-4 identifies each of the cards in a single midplane.

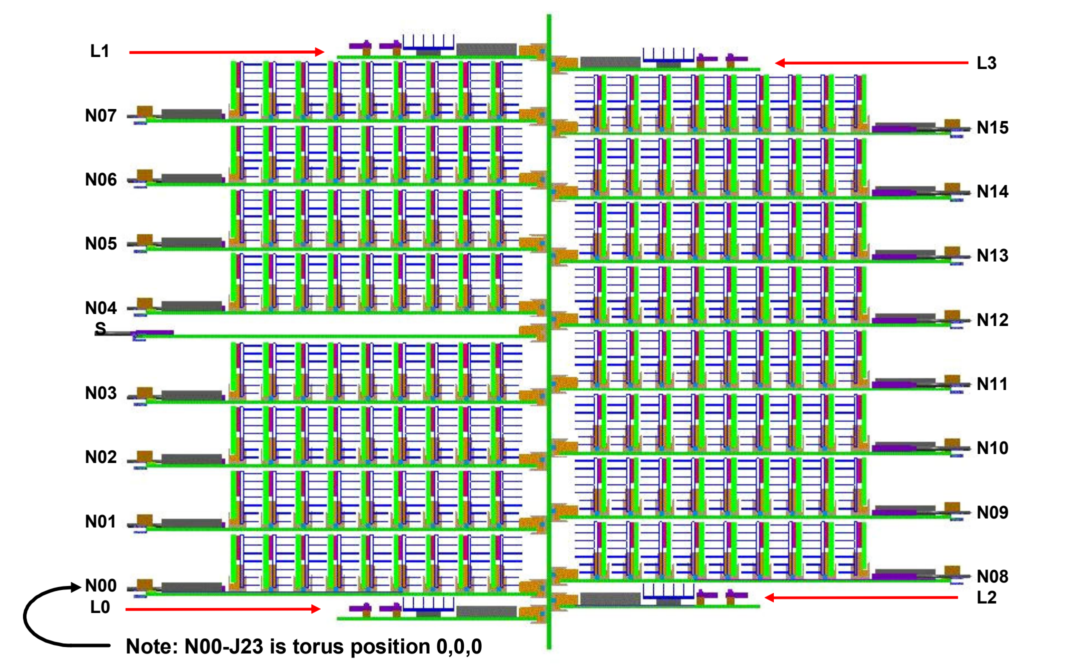


Figure A-4 Positions of the node, link, and service cards

Figure A-5 shows a diagram of a node card. On the front of the card are Ethernet ports EN0 and EN1. The first nodes behind the Ethernet ports are the I/O Nodes. In this diagram, the node card is fully populated with I/O Nodes, meaning that it has two I/O Nodes. Behind the I/O Nodes are the Compute Nodes.

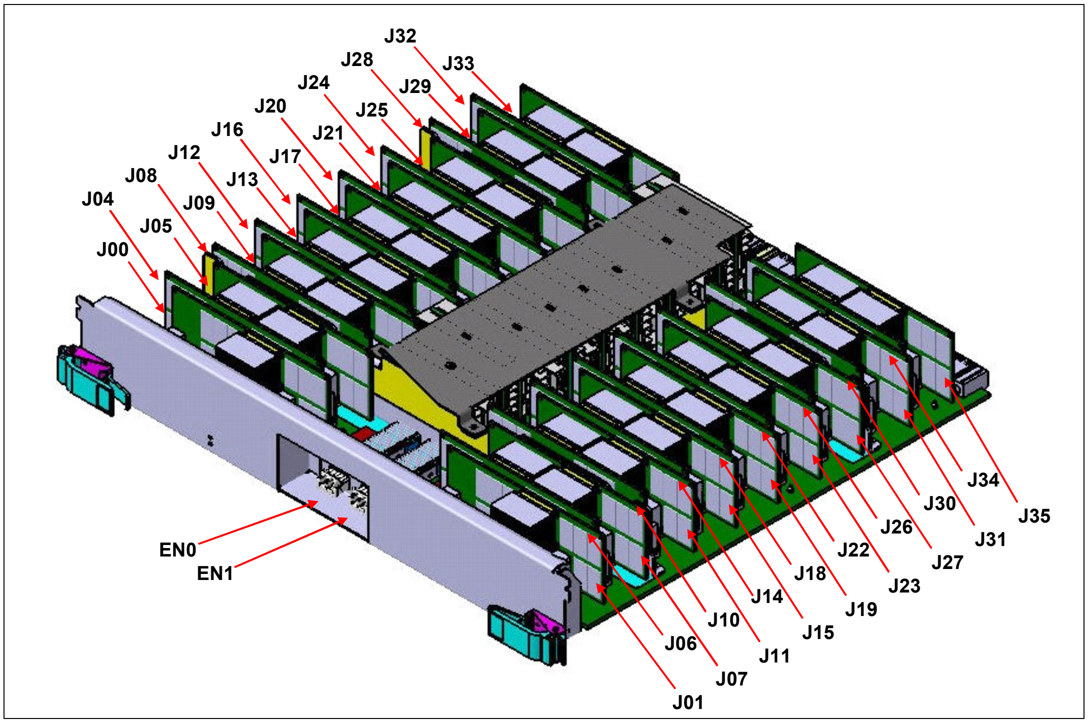


Figure A-5 Node card diagram

Figure A-6 is an illustration of a service card.

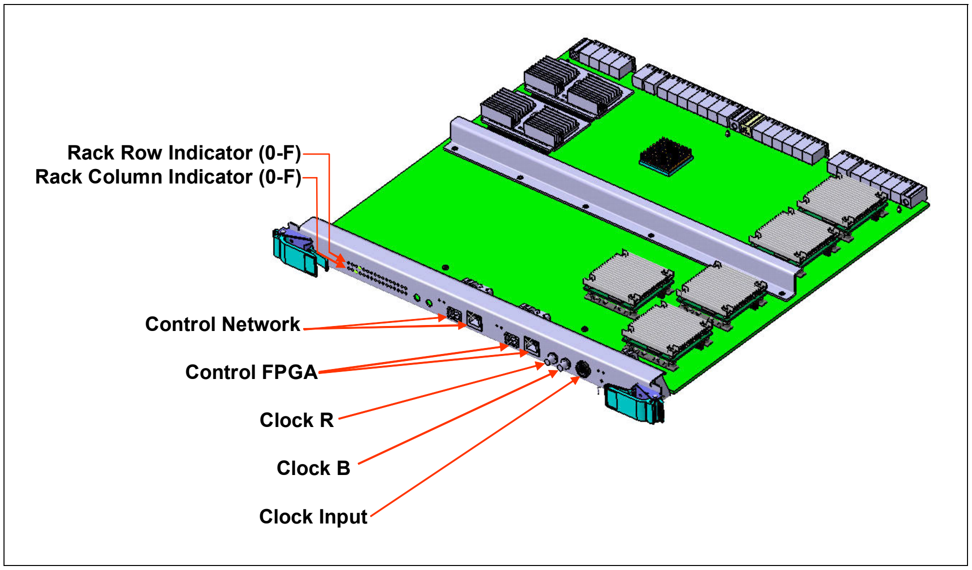


Figure A-6 Service card

Figure A-7 shows the link card. The locations identified as J00 through J15 are the link card connectors. The link cables are routed from one link card to another to form the torus network between the midplanes.

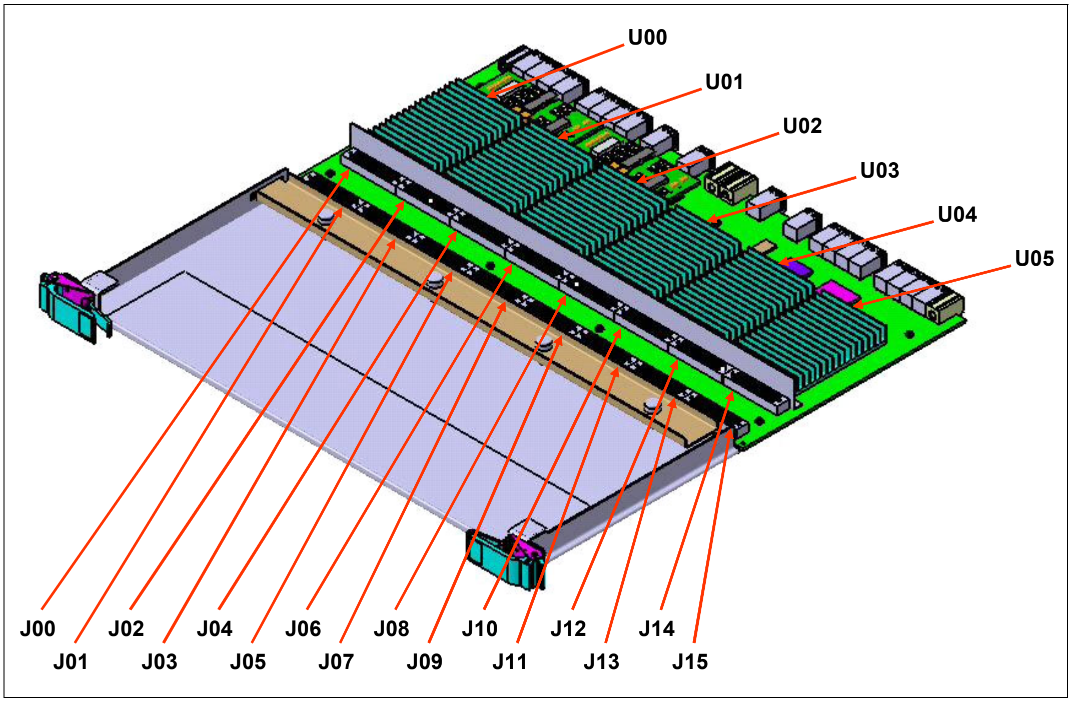


Figure A-7 Link card

Figure A-8 shows the clock card. If the clock is a secondary or tertiary clock, a cable comes to the input connector on the far right. Next to the input (just to the left) is the master and worker toggle switch. All clock cards are built with the capability of filling either role. If the clock is a secondary or tertiary clock, this must be set to *worker*. Output zero through nine can be used to send signals to midplanes throughout the system.

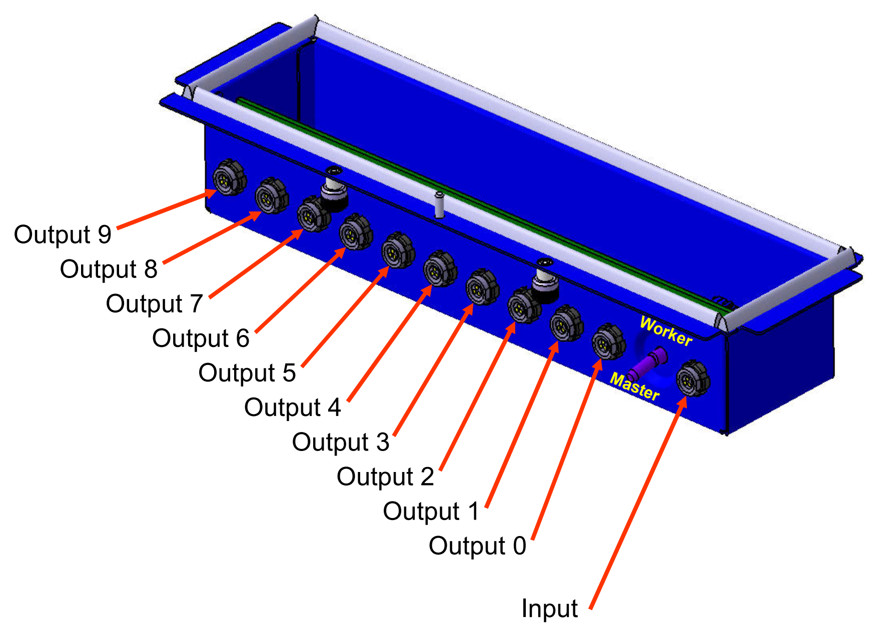


Figure A-8 Clock card

B

Files on architectural features

System calls that provide access to certain hardware or system features can be accessed by applications. In this appendix, we illustrate how to obtain hardware-related information.

Personality of Blue Gene/P

The personality of a Blue Gene/P node is static data given to every Compute Node and I/O Node at boot time by the control system. This data contains information that is specific to the node, with respect to the block that is being booted.

The personality is a set of C language structures that contain such items as the node's coordinates on the torus network. This kind of information can be useful if the application programmer wants to determine, at run time, where the tasks of the application are running. It can also be used to tune certain aspects of the application at run time, such as determining which set of tasks share the same I/O Node and then optimizing the network traffic from the Compute Nodes to that I/O Node.

Example of running personality on Blue Gene/P

Example B-1 illustrates how to invoke and print selected hardware features.

Example: B-1 personali.c architectural features program

```
/*  ---------------------------------------------------------------- */
/* Example: architectural features                                    */
/* Written by: Bob Walkup                                             */
/* IBM Watson, Yorktown, NY                                           */
/* September 17, 2007                                                 */
/*  ---------------------------------------------------------------- */

#include <mpi.h>
#include <stdio.h>

#include <spi/kernel_interface.h>
#include <common/bgp_personality.h>
#include <common/bgp_personality_inlines.h>

int main(int argc, char * argv[])
{
    int taskid, ntasks;
    int memory_size_MBytes;
    _BGP_Personality_t personality;
    int torus_x, torus_y, torus_z;
    int pset_size, pset_rank, node_config;
    int xsize, ysize, zsize, procid;
    char location[128];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
    MPI_Comm_size(MPI_COMM_WORLD, &ntasks);

    Kernel_GetPersonality(&personality, sizeof(personality));

    if (taskid == 0)
    {
        memory_size_MBytes = personality.DDR_Config.DDRSizeMB;
        printf("Memory size = %d MBytes\n", memory_size_MBytes);

        node_config = personality.Kernel_Config.ProcessConfig;
        if      (node_config == _BGP_PERS_PROCESSCONFIG_SMP) printf("SMP mode\n");
        else if (node_config == _BGP_PERS_PROCESSCONFIG_VNM) printf("Virtual-node mode\n");
        else if (node_config == _BGP_PERS_PROCESSCONFIG_2x2) printf("Dual mode\n");
        else                                                 printf("Unknown mode\n");

        printf("number of MPI tasks = %d\n", ntasks);

        xsize = personality.Network_Config.Xnodes;
        ysize = personality.Network_Config.Ynodes;
        zsize = personality.Network_Config.Znodes;

        pset_size = personality.Network_Config.PSetSize;
        pset_rank = personality.Network_Config.RankInPSet;

        printf("number of processors in the pset = %d\n", pset_size);
        printf("torus dimensions = <%d,%d,%d>\n", xsize, ysize, zsize);
    }

    torus_x = personality.Network_Config.Xcoord;
    torus_y = personality.Network_Config.Ycoord;
    torus_z = personality.Network_Config.Zcoord;

    BGP_Personality_getLocationString(&personality, location);

    procid = Kernel_PhysicalProcessorID();

    /*-----------------------------------------------*/
    /* print torus coordinates and the node location */
    /*-----------------------------------------------*/
    printf("MPI rank %d has torus coords <%d,%d,%d>  cpu = %d, location = %s\n",
           taskid, torus_x, torus_y, torus_z, procid, location);

    MPI_Finalize();
    return 0;
}
```

Example B-2 illustrates the makefile that is used to build personality.c. This particular file uses the GNU compiler.

Example: B-2   Makefile to build the personality.c program

```
BGP_FLOOR   = /bgsys/drivers/ppcfloor
BGP_IDIRS   = -I$(BGP_FLOOR)/arch/include

CC          = /bgsys/drivers/ppcfloor/comm/bin/mpicc

EXE         =  personality
OBJ         =  personality.o
SRC         =  personality.c
FLAGS       =
FLD         =

$(EXE): $(OBJ)
        ${CC} $(FLAGS) -o $(EXE) $(OBJ) $(BGP_LIBS)
$(OBJ): $(SRC)
```

```
        ${CC} $(FLAGS) $(BGP_IDIRS) -c $(SRC)

clean:
        rm personality.o personality
```

Example B-3 shows a section of the output that is generated after running **personality** using TXYZ mapping. (See Appendix F, “Mapping” on page 355.) Notice that the output has been ordered by MPI rank for readability.

Example: B-3 Output generated with TXYZ mapping

```
/bgsys/drivers/ppcfloor/bin/mpirun -partition N04_32_1 -label -env "BG_MAPPING=TXYZ" -mode VN
-np 8 -cwd `pwd` -exe personality | tee personality_VN_8_TXYZ.out

Memory size = 2048 MBytes
Virtual-node mode
number of MPI tasks = 128
number of processors in the pset = 32
torus dimensions = <4,4,2>
MPI rank 0 has torus coords <0,0,0>  cpu = 0, location = R00-M0-N04-J23
MPI rank 1 has torus coords <0,0,0>  cpu = 1, location = R00-M0-N04-J23
MPI rank 2 has torus coords <0,0,0>  cpu = 2, location = R00-M0-N04-J23
MPI rank 3 has torus coords <0,0,0>  cpu = 3, location = R00-M0-N04-J23
MPI rank 4 has torus coords <1,0,0>  cpu = 0, location = R00-M0-N04-J04
MPI rank 5 has torus coords <1,0,0>  cpu = 1, location = R00-M0-N04-J04
MPI rank 6 has torus coords <1,0,0>  cpu = 2, location = R00-M0-N04-J04
MPI rank 7 has torus coords <1,0,0> cpu = 3, location = R00-M0-N04-J04
```

Example B-4 illustrates running **personality** with XYZT mapping for a comparison. Notice that the output has been ordered by MPI rank for readability.

Example: B-4 Output generated with XYZT mapping

```
/bgsys/drivers/ppcfloor/bin/mpirun -partition N04_32_1 -label -env "BG_MAPPING=XYZT" -mode VN
-np 8 -cwd `pwd` -exe personality | tee personality_VN_8_XYZT.out

Memory size = 2048 MBytes
Virtual-node mode
number of MPI tasks = 128
number of processors in the pset = 32
torus dimensions = <4,4,2>
MPI rank 0 has torus coords <0,0,0>  cpu = 0, location = R00-M0-N04-J23
MPI rank 1 has torus coords <1,0,0>  cpu = 0, location = R00-M0-N04-J04
MPI rank 2 has torus coords <2,0,0>  cpu = 0, location = R00-M0-N04-J12
MPI rank 3 has torus coords <3,0,0>  cpu = 0, location = R00-M0-N04-J31
MPI rank 4 has torus coords <0,1,0>  cpu = 0, location = R00-M0-N04-J22
MPI rank 5 has torus coords <1,1,0>  cpu = 0, location = R00-M0-N04-J05
MPI rank 6 has torus coords <2,1,0>  cpu = 0, location = R00-M0-N04-J13
MPI rank 7 has torus coords <3,1,0>  cpu = 0, location = R00-M0-N04-J30
```

C

Header files and libraries

In this appendix, we provide information about selected header files and libraries for the IBM Blue Gene/P system. Directories that contain header files and libraries for the Blue Gene/P system are under the main system path in the /bgsys/drivers/ppcfloor directory.

Blue Gene/P applications

Blue Gene/P applications run on the Blue Gene/P compute nodes. Table C-1 describes the header files in the /bgsys/drivers/ppcfloor/comm/default/include directory and the /bgsys/drivers/ppcfloor/comm/fast/include directory. There are links to the “default” version of the header files in /bgsys/drivers/ppcfloor/comm/include for compatibility with previous releases of the Blue Gene/P software.

Table C-1 Header files in /bgsys/drivers/ppcfloor/comm/default/include

File name	Description
mpe_thread.h	Multi-processing environment (MPE) routines
mpicxx.h	MPI GCC script routine naming
mpif.h	MPI Fortran parameters
mpi.h	MPI C defines
mpiof.h	MPI I/O Fortran programs
mpio.h	MPI I/O C includes
mpix.h	Blue Gene/P extensions to the MPI specifications
mpido_properties.h	Properties used by MPIX_Get_property() and MPIX_Set_property()
mpi.mod, mpi_base.mod, mpi_constants.mod, mpi_sizeofs.mod	F90 bindings
opa_config.h, opa_primitives.h, opa_queue.h, opa_util.h	OpenPA headers used by MPICH2

Table C-2 describes the header files in the /bgsys/drivers/ppcfloor/comm/sys/include directory. There are links to the “default” version of the header files in /bgsys/drivers/ppcfloor/comm/include for compatibility with previous releases of the Blue Gene/P software.

Table C-2 Header files in /bgsys/drivers/ppcfloor/comm/sys/include

File name	Description
dcmf.h	Common BGP message layer interface
dcmf_collectives.h	Common BGP message layer interface for general collectives

Table C-3 describes the header files in the /bgsys/drivers/ppcfloor/arch/include/common directory.

Table C-3 Header files in /bgsys/drivers/ppcfloor/arch/include/common

File name	Description
bgp_personality.h	Defines personality
bgp_personality_inlines.h	Static inline for personality
bgp_personalityP.h	Defines personality processing

Table C-4 describes the 32-bit static and dynamic libraries in the /bgsys/drivers/ppcfloor/comm/default/lib directory and the /bgsys/drivers/ppcfloor/comm/fast/lib directory. There are links to the "default" version of the libraries in /bgsys/drivers/ppcfloor/comm/lib for compatibility with previous releases of the Blue Gene/P software.

Table C-4 32-bit static and dynamic libraries in /bgsys/drivers/ppcfloor/comm/default/lib/

File name	Description
libmpich.cnk.a, libmpich.cnk.so	C bindings for MPI
libcxxmpich.cnk.a, libcxxmpich.cnk.so	C++ bindings for MPI
libfmpich.cnk.a, libfmpich.cnk.so	Fortran bindings for MPI
libfmpich_.cnk.a	Fortran bindings for MPI with extra underscoring
libmpich.cnkf90.a, libmpich.cnkf90.so	Fortran 90 bindings
libopa.a	OpenPA library used by MPICH2
libtvmpich2.so	TotalView library for MPICH2 queue debugging

Table C-5 describes the 32-bit static and dynamic libraries in the /bgsys/drivers/ppcfloor/comm/sys directory. There are links to the "default" version of the libraries in /bgsys/drivers/ppcfloor/comm/lib for compatibility with previous releases of the Blue Gene/P software.

Table C-5 32-bit static and dynamic libraries in /bgsys/drivers/ppcfloor/comm/sys

File name	Description
libdcmf.cnk.a, libdcmf.cnk.so	Common BGP message layer interface in C
libdcmfcoll.cnk.a, libdcmfcoll.cnk.so	Common BGP message layer interface for general collectives in C
libdcmf-fast.cnk.a	"Fast" version of the common BGP message layer interface in C
libdcmfcoll-fast.cnk.a	"Fast" version of the common BGP message layer interface for general collectives in C

Resource management APIs

Blue Gene/P resource management applications run on the Service Node. Table C-6 describes the header files used by resource management applications. They are located in the /bgsys/drivers/ppcfloor/include directory.

Table C-6 Header files for resource management APIs

File name	Description
allocator_api.h	Available for applications using the Dynamic Partition Allocator APIs

File name	Description
attach_bg.h	The Blue Gene/P version of attach.h, which is described in the Message Passing Interface (MPI) debug specification
rm_api.h	Available for applications that use Bridge APIs
rt_api.h	Available for applications that use Real-time Notification APIs
sayMessage.h	Available for applications that use sayMessage APIs
sched_api.h	Available for applications that use the **mpirun** plug-in interface
submit_api.h	Available for applications that use the **submit** plug-in interface

Table C-7 describes the 64-bit dynamic libraries available to resource management applications. They are located in the /bgsys/drivers/ppcfloor/lib64 directory.

Table C-7 64-bit dynamic libraries for resource management APIs

File Name	Description
libbgpallocator.so	Required when using the Dynamic Partition Allocator APIs
libbgrealtime.so	Required when using the Real-time Notification APIs
libbgpbridge.so	Required when using the Bridge APIs
libsaymessage.so	Required when using the sayMessage APIs

D

Environment variables

In this appendix, we describe the environment variables that the user can change to affect the run time characteristics of a program that is running on the IBM Blue Gene/P compute nodes. Changes are usually made in an attempt to improve performance, although on occasion the goal is to modify functional attributes of the application.

In this appendix, we discuss the following topics:

- Setting environment variables
- Blue Gene/P MPI environment variables
- Compute Node Kernel environment variables

Setting environment variables

The easiest and most convenient way to set environment variables is to pass them in on the command line when running **mpirun**, for example, if you want to set environment variable “XYZ” to value “ABC,” you can call **mpirun** as the example shows:

```
$ mpirun -env "XYZ=ABC" myprogram.rts
```

Multiple environment variables can be passed by separating them by a space, for example:

```
$ mpirun -env "XYZ=ABC DEF=123" myprogram.rts
```

You can use other ways to pass environment variables with **mpirun**. For more information, see Chapter 11, “mpirun” on page 177.

Blue Gene/P MPI environment variables

The Blue Gene/P MPI implementation provides several environment variables that affect its behavior. By setting these environment variables, you can allow a program to run faster, or, if you set the variables improperly, you might cause the program not to run at all. None of these environment variables are required to be set for the Blue Gene/P MPI implementation to work.

The Blue Gene/P MPI implementation provides the following environment variables:

- DCMF_VERBOSE: Increases the amount of information dumped during an MPI_Abort() call. Possible values:
  - 0: No additional information is dumped.
  - 1: Additional information is dumped.
  - Default is 0.
- DCMF_STATISTICS: Turns on statistics printing for the message layer such as the maximum receive queue depth. Possible values:
  - 0: No statistics are printed.
  - 1: Statistics are printed.
  - Default is 0.
- DCMF_EAGER, DCMF_RZV, or DCMF_RVZ: Sets the cutoff for the switch to the rendezvous protocol. All three options are identical. This takes an argument, in bytes, to switch from the eager protocol to the rendezvous protocol for point-to-point messaging. Increasing the limit might help for larger partitions and if most of the communication is nearest neighbor:
  - Default is 1200 bytes.
- DCMF_OPTRVZ or DCMF_OPTRZV: Determines the optimized rendezvous limit. Both options are identical. This takes an argument, in bytes. The optimized rendezvous protocol will be used if: eager_limit <= message_size < (eager_limit + DCMF_OPTRZV). For sending, one of three protocols will be used depending on the message size: The eager protocol for small messages, the optimized rendezvous protocol for medium messages, and the default rendezvous protocol for large messages. The optimized rendezvous protocol generally has less latency than the default rendezvous protocol, but does not wait for a receive to be posted first. Therefore, unexpected messages in this size range might be received, consuming storage until the receives are issued. The default rendezvous protocol waits for a receive to be posted first. Therefore, no unexpected messages in this size range will be received. The optimized rendezvous protocol also avoids filling injection fifos which can cause delays while larger fifos are allocated, for

example, alltoall on large subcommunicators with thread mode multiple will benefit from optimized rendezvous.

  – Default is 0 bytes, meaning that optimized rendezvous is not used.

- DCMF_NUMREQUESTS: Sets the number of outstanding asynchronous broadcasts to have before a barrier is called. This is mostly used in allgather/allgatherv using asynchronous broadcasts. Higher numbers can help on larger partitions and larger message sizes.
  – Default is 32.
- DCMF_RMA_PENDING: Maximum outstanding RMA requests. Limits number of DCMF_Request objects allocated by MPI Onesided operations.
  – Default is 1000.
- DCMF_INTERRUPT or DCMF_INTERRUPTS: Turns on interrupt driven communications. This can be beneficial to some applications and is required if you are using Global Arrays or ARMCI. (They force this on, regardless of the environment setting). Possible values:
  – 0: Interrupt driven communications is not used.
  – 1: Interrupt driven communications is used.
  – Default is 0.
- DCMF_SENDER_SIDE_MATCHING or DCMF_SSM: Turns on sender-side matching. This can speed up point-to-point messaging in well-behaved applications, specifically those that do not do MPI_ANY_SOURCE receives. Possible values:
  – 0: Sender side matching is not used.
  – 1: Sender side matching is used.
  – Default is 0.
- DCMF_TOPOLOGY: Turns on optimized topology creation functions when using MPI_Cart_create with the reorder flag. We attempt to create communicators similar to those requested, that match physical hardware as much as possible. Possible values:
  – 0: Optimized topology creation functions are not used.
  – 1: Optimized topology creation functions are used.
  – Default is 1.
  – DCMF_COLLECTIVE or DCMF_COLLECTIVES: Controls whether optimized collectives are used. Possible values:
  – 0: Optimized collectives are not used.
  – 1: Optimized collectives are used.
  – NOTREE. Only collective network optimizations are not used.
  – Default is 1.
- DCMF_ASYNCCUTOFF: Changes the cutoff point between asynchronous and synchronous rectangular/binomial broadcasts. This can be highly application dependent:
  – Default is 128k.
- DCMF_SCATTER: Controls the protocol used for scatter. Possible values:
  – MPICH: Use the MPICH point-to-point protocol.
  – Default (or if anything else is specified) is to use a broadcast-based scatter at a 2k or larger message size.
- DCMF_SCATTERV: Controls the protocol used for scatterv. Possible values:
  – ALLTOALL: Use an all-to-all based protocol when the message size is above 2k. This is optimal for larger messages and larger partitions.
  – BCAST: Use a broadcast-based scatterv. This works well for small messages.

  - MPICH: Use the MPICH point-to-point protocol.
  - Default is ALLTOALL.
- DCMF_GATHER: Controls the protocol used for gather. Possible values:
  - MPICH: Use the MPICH point-to-point protocol.
  - Default (or if anything else is specified) is to use a reduce-based algorithm for larger message sizes.
- DCMF_REDUCESCATTER: Controls the protocol used for reduce_scatter operations. The options for DCMF_SCATTERV and DCMF_REDUCE can change the behavior of reduce_scatter. Possible values:
  - MPICH: Use the MPICH point-to-point protocol.
  - Default (or if anything else is specified) is to use an optimized reduce followed by an optimized scatterv. This works well for larger messages.
- DCMF_BCAST: Controls the protocol used for broadcast. Possible values:
  - MPICH: Turn off all optimizations for broadcast and use the MPICH point-to-point protocol.
  - TREE: Use the collective network. This is the default on MPI_COMM_WORLD and duplicates of MPI_COMM_WORLD in MPI_THREAD_SINGLE mode. This provides the fastest possible broadcast.
  - CCMI: Use the CCMI collective network protocol. This is off by default.
  - CDPUT: Use the CCMI collective network protocol with DPUT. This is off by default.
  - AR: Use the asynchronous rectangle protocol. This is the default for small messages on rectangular subcommunicators. The cutoff between async and sync can be controlled with DCMF_ASYNCCUTOFF.
  - AB: Use the asynchronous binomial protocol. This is the default for irregularly shaped subcommunicators. The cutoff between async and sync can be controlled with DCMF_ASYNCCUTOFF.
  - RECT: Use the rectangle protocol. This is the default for rectangularly shaped subcommunicators for large messages. This disables the asynchronous protocol.
  - BINOM: Use the binomial protocol. This is the default for irregularly shaped subcommunicators for large messages. This disables the asynchronous protocol.
  - Default varies based on the communicator. See above.
- DCMF_NUMCOLORS: Controls how many colors are used for rectangular broadcasts. Possible values:
  - 0: Let the lower-level messaging system decide.
  - 1, 2, or 3.
  - Default is 0.
- DCMF_SAFEALLREDUCE: The direct put allreduce bandwidth optimization protocols require the send/recv buffers to be 16-byte aligned on all nodes. Unfortunately, you can have root's buffer be misaligned from the rest of the nodes. Therefore, by default we must do an allreduce before dput allreduces to ensure all nodes have the same alignment. If you know all of your buffers are 16 byte aligned, turning on this option will skip the allreduce step and improve performance. Possible values:
  - N: Perform the allreduce
  - Y: Bypass the allreduce. If you have mismatched alignment, you will likely get weird behavior or asserts.
  - Default is N.

- DCMF_SAFEBCAST: The rectangle direct put bcast bandwidth optimization protocol requires the bcast buffers to be 16-byte aligned on all nodes. Unfortunately, you can have root's buffer be misaligned from the rest of the nodes. Therefore, by default we must do an allreduce before dput bcasts to ensure all nodes have the same alignment. If you know all of your buffers are 16 byte aligned, turning on this option will skip the allreduce step. Possible values:
  - N: Perform the allreduce
  - Y: Bypass the allreduce. If you have mismatched alignment, you will likely get weird behavior or asserts.
  - Default is N.
- DCMF_SAFEALLGATHER: The optimized allgather protocols require contiguous datatypes and similar datatypes on all nodes. To verify this is true, we must do an allreduce at the beginning of the allgather call. If the application uses *well-behaved* datatypes, you can set this option to skip over the allreduce. This is most useful in irregular subcommunicators where the allreduce can be expensive. Possible values:
  - N: Perform the allreduce.
  - Y: Skip the allreduce. Setting this with *unsafe* datatypes will yield unpredictable results, usually hangs.
  - Default is N.
- DCMF_SAFEALLGATHERV: The optimized allgatherv protocols require contiguous datatypes and similar datatypes on all nodes. Allgatherv also requires continuous displacements. To verify this is true, we must do an allreduce at the beginning of the allgatherv call. If the application uses well-behaved datatypes and displacements, you can set this option to skip over the allreduce. This is most useful in irregular subcommunicators where the allreduce can be expensive. Possible values:
  - N: Perform the allreduce.
  - Y: Skip the allreduce. Setting this with unsafe datatypes will yield unpredictable results, usually hangs.
  - Default is N.
- DCMF_SAFESCATTERV: The optimized scatterv protocol requires contiguous datatypes and similar datatypes on all nodes. It also requires continuous displacements. To verify this is true, we must do an allreduce at the beginning of the scatterv call. If the application uses well-behaved datatypes and displacements, you can set this option to skip over the allreduce. This is most useful in irregular subcommunicators where the allreduce can be expensive. Possible values:
  - N: Perform the allreduce.
  - Y: Skip the allreduce. Setting this with unsafe datatypes will yield unpredictable results, usually hangs.
  - Default is N.
- DCMF_ALLTOALL, DCMF_ALLTOALLV, or DCMF_ALLTOALLW: Controls the protocol used for alltoall/alltoallv/alltoallw. Possible values:
  - MPICH: Turn off all optimizations and use the MPICH point-to-point protocol.
  - Default (or if anything else is specified) is to use an optimized alltoall/alltoallv/alltoallw.
- DCMF_ALLTOALL_PREMALLOC, DCMF_ALLTOALLV_PREMALLOC, or DCMF_ALLTOALLW_PREMALLOC: These are equivalent options. The alltoall protocols require 6 arrays to be setup before communication begins. These 6 arrays are each of size (comm_size) so can be sizeable on large machines. If your application does not use alltoall, or you need as much memory as possible, you can turn off pre-allocating these

arrays. By default, we allocate them once per communicator creation. There is only one set, regardless of whether you are using alltoall, alltoallv, or alltoallw. Possible values:

  - Y: Premalloc the arrays.
  - N: Malloc and free on every alltoall operation.
  - Default is Y.

- DCMF_ALLGATHER: Controls the protocol used for allgather. Possible values:
  - MPICH: Turn off all optimizations for allgather and use the MPICH point-to-point protocol.
  - ALLREDUCE: Use a collective network based allreduce. This is the default on MPI_COMM_WORLD for smaller messages.
  - ALLTOALL: Use an all-to-all based algorithm. This is the default on irregular communicators. It works very well for larger messages.
  - BCAST: Use a broadcast. This will use a collective network broadcast on MPI_COMM_WORLD. It is the default for larger messages on MPI_COMM_WORLD. This can work well on rectangular subcommunicators for smaller messages.
  - ASYNC: Use an async broadcast. This will use asynchronous broadcasts to do the allgather. This is a good option for small messages on rectangular or irregular subcommunicators.
  - Default varies based on the communicator. See above.
- DCMF_ALLGATHERV: Controls the protocol used for allgatherv. Possible values:
  - MPICH: Turn off all optimizations for allgather and use the MPICH point-to-point protocol.
  - ALLREDUCE: Use a collective network based allreduce. This is the default on MPI_COMM_WORLD for smaller messages.
  - ALLTOALL: Use an all-to-all based algorithm. This is the default on irregular communicators. It works very well for larger messages.
  - BCAST: Use a broadcast. This will use a collective network broadcast on MPI_COMM_WORLD. It is the default for larger messages on MPI_COMM_WORLD. This can work well on rectangular subcommunicators for smaller messages.
  - ASYNC: Use an async broadcast. This will use asynchronous broadcasts to do the allgather. This is a good option for small messages on rectangular or irregular subcommunicators.
  - Default varies based on the communicator. See previous.
- DCMF_PREALLREDUCE: Controls the protocol used for the pre-allreducing employed by bcast, allreduce, allgather(v), and scatterv. This option is mostly independent from DCMF_ALLREDUCE. Possible values are:
  - MPIDO: Just call MPIDO_Allreduce and let the existing logic determine what allreduce to use. This can be expensive, but it is the only guaranteed option, and it is the only way to get MPICH for the pre-allreduce
  - SRECT: Use the short rectangle protocol. If you set this and do not have a rectangular (sub)communicator, you get the MPIDO option. This is the default selection for rectangular subcomms.
  - SBINOM: Use the short binomial protocol. This is the default for irregular subcomms.
  - ARING: Use the async rectangular ring protocol
  - ARECT: Use the async rectangle protocol
  - ABINOM: Use the async binomial protocol

- RING: Use the rectangular ring protocol
- RECT: Use the rectangle protocol
- TDPUT: Use the tree dput protocol. This is the default in virtual node mode on MPI_COMM_WORLD
- TREE: Use the tree. This is the default in SMP mode on MPI_COMM_WORLD
- DPUT: Use the rectangular direct put protocol
- PIPE: Use the pipelined CCMI tree protocol
- BINOM: Use a binomial protocol

- DCMF_ALLREDUCE: Controls the protocol used for allreduce. Possible values:
  - MPICH: Turn off all optimizations for allreduce and use the MPICH point-to-point protocol.
  - RING - Use a rectangular ring protocol. This is the default for rectangular subcommunicators.
  - RECT: Use a rectangular/binomial protocol. This is off by default.
  - BINOM: Use a binomial protocol. This is the default for irregular subcommunicators.
  - TREE: Use the collective network. This is the default (except for GLOBAL between 512 and 8K) for MPI_COMM_WORLD and duplicates of MPI_COMM_WORLD in MPI_THREAD_SINGLE mode.
  - GLOBAL: Use the global collective network protocol for sizes between 512 and 8K. Otherwise this defaults the same as TREE.
  - CCMI: Use the CCMI collective network protocol. This is off by default.
  - PIPE: Use the pipelined CCMI collective network protocol. This is off by default.
  - ARECT: Enable the asynchronous rectangle protocol
  - ABINOM: Enable the async binomial protocol
  - ARING: Enable the asynchronous version of the rectangular ring protocol.
  - TDPUT: Use the tree+direct put protocol. This is the default for VNM on MPI_COMM_WORLD
  - DPUT: Use the rectangular direct put protocol. This is the default for large messages on rectangular subcomms and MPI_COMM_WORLD
  - Default varies based on the communicator and message size and if the operation/datatype pair is supported on the tree hardware.
- DCMF_ALLREDUCE_REUSE_STORAGE: This allows the lower level protocols to reuse some storage instead of malloc/free on every allreduce call. Possible values:
  - Y: Does not malloc/free on every allreduce call. This improves performance, but retains malloc'd memory between allreduce calls.
  - N: Malloc/free on every allreduce call. This frees up storage for use between allreduce calls.
  - Default is Y.
- DCMF_ALLREDUCE_REUSE_STORAGE_LIMIT: This specifies the upper limit of storage to save and reuse across allreduce calls when DCMF_ALLREDUCE_REUSE_STORAGE is set to Y. (This environment variable is processed within the DCMF_Allreduce_register() API, not in MPIDI_Env_setup().):
  - Default is 1048576 bytes.

- DCMF_REDUCE: Controls the protocol used for reduce. Possible values:
  - MPICH: Turn off all optimizations and use the MPICH point-to-point protocol.
  - RECT: Use a rectangular/binomial protocol. This is the default for rectangular subcommunicators.
  - BINOM: Use a binomial protocol. This is the default for irregular subcommunicators.
  - TREE: Use the collective network. This is the default for MPI_COMM_WORLD and duplicates of MPI_COMM_WORLD in MPI_THREAD_SINGLE mode.
  - CCMI: Use the CCMI collective network protocol. This is off by default.
  - Default varies based on the communicator. See previous.
- DCMF_REDUCE_REUSE_STORAGE: This allows the lower level protocols to reuse some storage instead of malloc/free on every reduce call. Possible values:
  - Y: Does not malloc/free on every reduce call. This improves performance, but retains malloc'd memory between reduce calls.
  - N: Malloc/free on every reduce call. This frees up storage for use between reduce calls.
  - Default is Y.
- DCMF_REDUCE_REUSE_STORAGE_LIMIT: This specifies the upper limit of storage to save and reuse across allreduce calls when DCMF_REDUCE_REUSE_STORAGE is set to Y. (This environment variable is processed within the DCMF_Reduce_register() API, not in MPIDI_Env_setup().):
  - Default is 1048576 bytes.
- DCMF_BARRIER: Controls the protocol used for barriers. Possible values:
  - MPICH: Turn off optimized barriers and use the MPICH point-to-point protocol.
  - BINOM: Use the binomial barrier. This is the default for all subcommunicators.
  - GI: Use the GI network. This is the default for MPI_COMM_WORLD and duplicates of MPI_COMM_WORLD in MPI_THREAD_SINGLE mode.
  - CCMI: Use the CCMI GI network protocol. This is off by default.
  - Default varies based on the communicator. See above.
- DCMF_STAR: Turns on the STAR-MPI mechanism that tunes MPI collectives. STAR-MPI is turned off by default. Possible values:
  - 1: turn on
- DCMF_STAR_NUM_INVOCS: Sets the number of invocations that STAR-MPI uses to examine performance of each communication algorithm:
  - Possible values: any integer value > 0 (default is 10).
- DCMF_STAR_TRACEBACK_LEVEL: Sets the traceback level of an MPI collective routine, which will be used to get the address of the caller to the collective routine. The default is 3 (MPI application, MPICH, MPIDO). If users utilize or write their own collective wrappers, then they must increase the traceback level (beyond 3) depending on the extra levels they introduce by their wrappers:
  - Possible values: any integer value > 3 (default is 3).

- DCMF_STAR_CHECK_CALLSITE: Turns on sanity check that makes sure all ranks are involved in the same collective call site. This is important in root-like call sites (Bcast, Reduce, Gather...etc) where the call site of the root might be different than non root ranks (different if statements):
  - Possible values: 1 - (default is 1: on).
- DCMF_STAR_VERBOSE: Turns on verbosity of STAR-MPI by writing to an output file in the form "exec_name-star-rank#.log". This is turned off by default:
  - Possible values: 1 - (default is 0: off).
- DCMF_RECFIFO: The size, in bytes, of each DMA reception FIFO. Incoming torus packets are stored in this fifo until DCMF Messaging can process them. Making this larger can reduce torus network congestion. Making this smaller leaves more memory available to the application. DCMF Messaging uses one reception FIFO. The value specified is rounded up to the nearest 32-byte boundary:
  - Default is 8388608 bytes (8 megabytes).
- DCMF_INJFIFO: The size, in bytes, of each DMA injection FIFO. These FIFOs store 32-byte descriptors, each describing a memory buffer to be sent on the torus. Making this larger can reduce overhead when there are many outstanding messages. Making this smaller can increase that overhead. DCMF Messaging uses 15 injection FIFOs in DEFAULT and RZVANY mode, and 25 injection FIFOs in ALLTOALL mode (refer to DCMF_FIFOMODE). The value given is rounded up to the nearest 32-byte boundary:
  - Default is 32768 (32 kilobytes).
- DCMF_RGETFIFO: The size, in bytes, of each DMA remote get FIFO. These FIFOs store 32-byte descriptors, each describing a memory buffer to be sent on the torus, and are used to queue requests for data (remote gets). Making this larger can reduce torus network congestion and reduce overhead. Making this smaller can increase that congestion and overhead. DCMF Messaging uses 7 remote get FIFOs in DEFAULT and ALLTOALL mode, and 13 remote get FIFOs in RZVANY mode (refer to DCMF_FIFOMODE). The value given is rounded up to the nearest 32-byte boundary:
  - Default is 32768 (32 kilobytes).
- DCMF_POLLLIMIT: The limit on the number of consecutive non-empty polls of the reception fifo before exiting the poll function so other processing can be performed. Making this larger might help performance because polling overhead is smaller. Making this smaller might be necessary for applications that continuously send to a node that needs to perform processing. Special values:
  - 0: There is no limit.
  - Default is 16 polls.
- DCMF_INJCOUNTER: The number of DMA injection counter subgroups that DCMF will allocate during MPI_Init or DCMF_Messager_Initialize. There are 8 DMA counters in a subgroup. This is useful for applications that access the DMA directly and need to limit the number of injection counters used for messaging. Possible values:
  - 1..8: The specified value can range from 1 to 8.
  - Default is 8.
- DCMF_RECCOUNTER: The number of DMA reception counter subgroups that DCMF will allocate during MPI_Init or DCMF_Messager_Initialize. There are 8 DMA counters in a subgroup. This is useful for applications that access the DMA directly and need to limit the number of reception counters used for messaging. Possible values:
  - 1..8: The specified value can range from 1 to 8.
  - Default is 8.

- DCMF_FIFOMODE: The fifo mode to use. This determines how many injection fifos are used by messaging and what they are used for:
  - DEFAULT: The default fifo mode. Uses 22 injection fifos:
    - 6 normal fifos, each mapped to 1 torus fifo.
    - 1 local normal fifo.
    - 6 remote get fifos, each mapped to 1 torus fifo.
    - 1 local remote get fifo.
    - 6 all-to-all fifos. These can inject into any of the torus fifos.
    - 2 control message fifos.
  - RZVANY: Similar to DEFAULT, except it is optimized for sending messages that use the rendezvous protocol. It has 6 more remote get fifos optimized for sending around corners:
    - 6 normal fifos, each mapped to 1 torus fifo.
    - 1 local normal fifo.
    - 6 remote get fifos, each mapped to 1 torus fifo.
    - 6 remote get fifos, each mapped to all of the torus fifos.
    - 1 local remote get fifo.
    - 6 all-to-all fifos.
    - 2 control message fifos.
  - ALLTOALL: Optimized for All-To-All communications. Same as DEFAULT, except there are 16 All-To-All fifos that can inject into any of the torus fifos:
    - Default is DEFAULT.
- DCMF_DMA_VERBOSE: Control the output of information associated with the Direct Memory Access messaging device. Specifically, it controls whether informational RAS events are generated when remote get resources become full and are increased in size. Possible values:
  - 0: No DMA information is output.
  - 1: DMA information is output.
  - Default is 0.
- DCMF_THREADED_TREE: Bitmask indicating whether Send (1) and Recv (2) should use Comm (helper) Threads. Note, Comm threads might not be used in all cases, it depends on factors, such as run mode, message size, partition size, data operand, and so on. Possible values:
  - 0: Neither Send nor Recv will use Comm Threads.
  - 1: Only Send will use Comm Threads.
  - 2: Only Recv will use Comm Threads.
  - 3: Both Send and Recv will use Comm Threads.
  - Default is 3.
- DCMF_PERSISTENT_ADVANCE: Number of cycles to persist in the advance loop waiting for a (the first) receive packet to arrive:
  - Default is a value computed from the partition size (Collective network depth).
- DCMF_PERSIST_MAX: Upper limit on the number of cycles to persist in advance. This is only used when DCMF_PERSISTENT_ADVANCE is computed:
  - Default is 5000 cycles.
- DCMF_PERSIST_MIN: Lower limit on the number of cycles to persist in advance. This is only used when DCMF_PERSISTENT_ADVANCE is computed:
  - Default is 1000 cycles.

- DCMF_TREE_DBLSUM_THRESH: Number of doubles at which to start using the 2-Pass algorithm. Special values:
  - -1 (minus 1): Effectively disables the 2-Pass algorithm.
  - Default is 2 doubles.
- DCMF_TREE_HELPER_THRESH: Number of bytes (message size) at which to start using a helper thread. Ideally this value would be computed based on network depth and comm thread start-up time:
  - Default 16384 bytes.
- DCMF_TREE_VN_DEEP: Boolean indicating whether to use the *Deep* protocol for receiving a message in virtual node mode. Currently not used. Possible values:
  - 0 (false): The Deep protocol is not used.
  - 1 (true): The Deep protocol is used.
  - Default is 1.

Compute Node Kernel environment variables

The Compute Node Kernel (CNK) provides several environment variables that affect its run-time characteristics. If these variables are set improperly, it could cause a program to fail to run. None of these environment variables are required to be set for the CNK to work.

The CNK provides the following environment variables:

- BG_STACKGUARDENABLE

  Boolean indicating whether CNK creates guard pages. Default is 1 (YES). If the variable is specified, a value must be set to either "0" or "1".
- BG_STACKGUARDSIZE

  Size, in bytes, of the main() ’s stack guard area. Default is 4096. If the specified value is greater than zero but less than 512, 512 bytes are used.
- BG_PROCESSWINDOWS

  The number of TLB slots that are preserved for process windows. If not specified, no slots are reserved.
- BG_MAXALIGNEXP

  The maximum number of floating-point alignment exceptions that CNK can handle. If the maximum is exceeded, the application coredumps. Values:
  - 0: No alignment exceptions are processed.
  - -1: All alignment exceptions.
  - <n>: n alignment exceptions are processed (1000 is the default).
- BG_POWERMGMTPERIOD

  The number of microseconds between proactive power management idle loops.
- BG_POWERMGMTDUR

  The number of microseconds spent in one proactive power management idle loop.
- BG_SHAREDMEMPOOLSIZE

  Size, in MB, of the shared memory region. Default is 8 MB.
- BG_PERSISTMEMSIZE

  Size, in MB, of the persistent memory region. Default is 0.

- BG_PERSISTMEMRESET

  Boolean indicating that the persistent memory region must be cleared before the job starts. Default is 0. To enable, the value must be set to “1”.

- BG_COREDUMPONEXIT

  Creates a core file when the application exits. This variable is useful when the application performed an exit() operation and the cause and location of the exit() is not known.

- BG_COREDUMPONERROR

  Creates a core file when the application exits with a non-zero exit status. This variable is useful when the application performed an exit(1) operation and the cause and location of the exit(1) is not known.

- BG_COREDUMPDISABLED

  Boolean. Disables creation of core files if set.

- BG_COREDUMP_FILEPREFIX

  Sets the file name prefix of the core files. The default is “core”. The MPI task number is appended to this prefix to form the file name.

- BG_COREDUMP_PATH

  Sets the directory for the core files.

- BG_COREDUMP_FULL

  Specifies that the full core file format is to be used for coredumps.

- BG_COREDUMP_REGS

  Part of the Booleans that control whether or not register information is included in the core files. BG_COREDUMP_REGS is the master switch.

- BG_COREDUMP_GPR

  Part of the Booleans that control whether or not register information is included in the core files. BG_COREDUMP_GPR controls GPR (integer) registers.

- BG_COREDUMP_FPR

  Part of the Booleans that control whether or not register information is included in the core files. BG_COREDUMP_FPR controls output of FPR (floating-point) registers.

- BG_COREDUMP_SPR

  Part of the Booleans that control whether or not register information is included in the core files. BG_COREDUMP_SPR controls output of SPR (special purpose) registers.

- BG_COREDUMP_PERS

  Boolean that controls whether the node's personality information (XYZ dimension location, memory size, and so on) are included in the core files.

- BG_COREDUMP_INTCOUNT

  Boolean that controls whether the number of interrupts handled by the node are included in the core file.

- BG_COREDUMP_TLBS

  Boolean that controls whether the TLB layout at the time of the core is to be included in the core file.

- BG_COREDUMP_STACK

  Boolean that controls whether the application stack addresses are to be included in the core file.

- BG_COREDUMP_SYSCALL

  Boolean that controls whether a histogram of the number of system calls performed by the application is to be included in the core file.

- BG_COREDUMP_BINARY

  Specifies the MPI ranks for which a binary core file will be generated rather than a lightweight core file. This type of core file can be used with the GNU Project Debugger (GDB) but not the Blue Gene/P Core Processor utility. If this variable is not set then all ranks will generate a lightweight core file. The variable must be set to a comma-separated list of the ranks that will generate a binary core file or "*" (an asterisk) to have all ranks generate a binary core file.

- BG_APPTHREADDEPTH

  Integer that controls the number of application threads per core. Default is 1. The value can be between 1 and 3.

E

Porting applications

In this appendix, we summarize Appendix A, “BG/L prior to porting code,” in *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686. Porting applications to massively parallel systems requires special considerations to take full advantage of this specialized architecture. Never underestimate the effort required to port a code to any new hardware. The amount of effort depends on the nature of the way in which the code has been implemented.

Answer the following questions to help you in the decision-making process of porting applications and the level of effort required (answering “yes” to most of the questions is an indication that your code is already enabled for distributed-memory systems and a good candidate for Blue Gene/P):

1. Is the code already running in parallel?
2. Is the application addressing 32-bit?
3. Does the application rely on system calls, for example, **`system`**?
4. Does the code use the Message Passing Interface (MPI), specifically MPICH? Of the several parallel programming APIs, the only one supported on the Blue Gene/P system that is portable is MPICH. OpenMP is supported only on individual nodes.
5. Is the memory requirement per MPI task less than 4 GB?
6. Is the code computational intensive? That is, is there a small amount of I/O compared to computation?
7. Is the code floating-point intensive? This allows the double floating-point capability of the Blue Gene/P system to be exploited.
8. Does the algorithm allow for distributing the work to a large number of nodes?
9. Have you ensured that the code does not use `flex_lm` licensing? At present, flex_lm library support for Linux on IBM System p® is not available.

If you answered “yes” to all of these questions, answer the following questions:

- Has the code been ported to Linux on System p?

- Is the code Open Source Software (OSS)? These type of applications require the use of the GNU standard **`configure`** and special considerations are required.[85]
- Can the problem size be increased with increased numbers of processors?
- Do you use standard input? If yes, can this be changed to single file input?

F

Mapping

In this appendix, we summarize and discuss mapping of tasks with respect to the Blue Gene/P system. We define mapping as an assignment of MPI rank onto IBM Blue Gene processors. As with IBM Blue Gene/L, the network topology for IBM Blue Gene/P is a three-dimensional (3D) torus or mesh, with direct links between the nearest neighbors in the +/-x, +/-y, and +/-z directions. When communication involves the nearest neighbors on the torus network, you can obtain a large fraction of the theoretical peak bandwidth. However, when MPI ranks communicate with many hops between the neighbors, the effective bandwidth is reduced by a factor that is equal to the average number of hops that messages take on the torus network. In a number of cases, it is possible to control the placement of MPI ranks so that communication remains local. This can significantly improve scaling for a number of applications, particularly at large processor counts.

The default mapping is to place MPI ranks on the system in XYZT order, where <X,Y,Z> are torus coordinates and T is the processor number within each node (T=0,1,2,3). If the job uses symmetrical multiprocessing (SMP) Node mode on the Blue Gene/P system, only one MPI rank is assigned to each node using processor 0. For SMP Node mode and the default mapping, we get the following results:

- MPI rank 0 is assigned to <X,Y,Z,T> coordinates <0,0,0,0>.
- MPI rank 1 is assigned to <X,Y,Z,T> coordinates <1,0,0,0>.
- MPI rank 2 is assigned to <X,Y,Z,T> coordinates <2,0,0,0>.

The results continue like this, first incrementing the X coordinate, then the Y coordinate, and then the Z coordinate. In Virtual Node Mode and in Dual mode, the same XYZT order remains the default, for example, in Virtual Node Mode, the system first places one MPI rank using processor 0 on each of the nodes in XYZ order. The next MPI ranks are assigned to processor 1, again in XYZ order, and so forth. In many cases, it might be better to change this assignment so that the first four MPI ranks use processors 0,1,2,3 on the first node, then the next four ranks use processors 0,1,2,3 on the second node, where the nodes are populated in XYZ order. This ordering is called *TXYZ order* (first increment T, then X, then Y, and then Z).

The predefined mappings available on Blue Gene/P are the same as those available on Blue Gene/L: XYZT, XZYT, YZXT, YXZT, ZXYT, ZYXT, TXYZ, TXZY, TYZX, TYXZ, TZXY, TZYX.

Table F-1 illustrates this type of mapping using the output from the personality program presented in Appendix B, “Files on architectural features” on page 331.

Table F-1 Topology mapping 4x4x2 with TXYZ and XYZT

Mapping option	Topology	Coordinates	Processor
TXYZ	4x4x2	0,0,0	0
		0,0,0	1
		0,0,0	2
		0,0,0	3
		1,0,0	0
		1,0,0	1
		1,0,0	2
		1,0,0	3
XYZT	4x4x2	0,0,0	0
		1,0,0	0
		2,0,0	0
		3,0,0	0
		0,1,0	0
		1,1,0	0
		2,1,0	0
		3,1,0	0

The way to specify a mapping depends on the method that is used for job submission. The **mpirun** command for the Blue Gene/P system includes two methods to specify the mapping. You can add `-mapfile TXYZ` to request TXYZ order. Other permutations of XYZT are also permitted. You can also create a map file, and use `-mapfile my.map`, where *my.map* is the name of your map file. Alternatively, you can specify the environment variable `-env BG_MAPPING=TXYZ` to obtain one of the predefined non-default mappings.

Using customized map file provides the most flexibility. The syntax for the map file is simple. It must contain one line for each MPI rank in the Blue Gene/P partition, with four integers on each line separated by spaces, where the four integers specify the <X,Y,Z,T> coordinates for each MPI rank. The first line in the map file assigns MPI rank 0, the second line assigns MPI rank 1, and so forth. It is important to ensure that your map file is consistent, with a unique relationship between MPI rank and <X,Y,Z,T> location.

General guidance

For applications that use a 1D, 2D, 3D, or 4D (D for dimensional) logical decomposition scheme, it is often possible to map MPI ranks onto the Blue Gene/P torus network in a way that preserves locality for nearest-neighbor communication, for example, in a one-dimensional processor topology, where each MPI rank communicates with its rank +/- 1, the default XYZT mapping is sufficient at least for partitions large enough to use torus wrap-around.

Torus wrap-around is enabled for partitions that are one midplane = 8x8x8 512 nodes, or multiples of one midplane. With torus wrap-around, the XYZT order keeps communication local, except for one extra hop at the torus edges. For smaller partitions, such as a 64-node partition with a 4x4x4 mesh topology, it is better to create a map file that assigns ranks that go down the X-axis in the +x direction, and then for the next Y-value, fold the line to return in the -x direction, making a snake-like pattern that winds back and forth, filling out the 4x4x4 mesh. It is worthwhile to note that for a random placement of MPI ranks onto a 3D torus network, the average number of hops is one-quarter of the torus length, in each of the three dimensions. Thus mapping is generally more important for large or elongated torus configurations.

Two-dimensional logical processes topologies are more challenging. In some cases, it is possible to choose the dimensions of the logical 2D process mesh so that one can fold the logical 2D mesh to fit perfectly in the 3D Blue Gene/P torus network, for example, if you want to use one midplane (8x8x8 nodes) in virtual node mode, a total of 2048 CPUs are available. A 2D process mesh is 32x64 for this problem. The 32 dimension can be lined up along one edge of the torus, say the X-axis, using TX order to fill up processors (0,1,2,3) on each of the eight nodes going down the X-axis, resulting in 32 MPI ranks going down the X-axis.

The simplest good mapping, in this case, is to specify *`-mapfile TXYZ`*. This keeps nearest-neighbor communication local on the torus, except for one extra hop at the torus edges. You can do slightly better by taking the 32x64 logical 2D process mesh, aligning one edge along the X-axis with TX order and then folding the 64 dimension back and forth to fill the 3D torus in a seamless manner. It is straightforward to construct small scripts or programs to generate the appropriate map file. Not all 2D process topologies can be neatly folded onto the 3D torus.

For 3D logical process topologies, it is best to choose a decomposition or mapping that fits perfectly onto the 3D torus if possible, for example, if your application uses SMP Node mode on one Blue Gene/P rack (8x8x16 torus); then it is best to choose a 3D decomposition with 8 ranks in the X-direction, 8 ranks in the Y-direction, and 16 ranks in the Z-direction. If the application requires a different decomposition - for example, 16x8x8 - you might be able to use mapping to maintain locality for nearest-neighbor communication. In this case, ZXY order works.

Quantum chromodynamics (QCD) applications often use a 4D process topology. This can fit perfectly onto Blue Gene/P using virtual node mode, for example, with one full rack, there are 4096 CPUs in virtual node mode, with a natural layout of 8x8x16x4 (X,Y,Z,T order). By choosing a decomposition of 8x8x16x4, communication remains entirely local for nearest neighbors in the logical 4D process mesh. In contrast, a more balanced decomposition of 8x8x8x8 results in a significant amount of link sharing, and thus degraded bandwidth in one of the dimensions.

In summary, it is often possible to choose a mapping that keeps communication local on the Blue Gene/P torus network. This is recommended for cases where a natural mapping can be identified based on the parallel decomposition strategy used by the application. The mapping can be specified using the `-mapfile` argument for the **`mpirun`** command.

G

htcpartition

The **htcpartition** utility, the subject of this appendix, boots or frees a HTC partition from a Front End Node or service node. The **htcpartition** utility is similar to **mpirun** in two ways. First, both communicate with the mpirun daemon on the service node; however, **htcpartition** cannot run a job. Second, the mpirun scheduler plug-in interface is also called when **htcpartition** is executed. The plug-in interface provides a method for the resource scheduler to specify the partition to boot or free, and if the resource scheduler does not allow **mpirun** outside its framework, that policy is also enforced with **htcpartition**.

The **htcpartition** utility is located in /bgsys/drivers/ppcfloor/bin along with the other IBM Blue Gene/P executables. Its return status indicates whether or not the request succeeded; zero indicates success and non-zero means failure. Table G-1 provides a complete list of options for the `htcpartition` command.

Table G-1 htcpartition parameters

Parameter (and syntax)	Description
--help	Extended help information.
--version	Version information.
--boot | --free	Indication whether to boot or free the HTC partition. One, and only one, of these parameters must be specified.
--partition <*partition*>	Partition to boot or free. Alternatively, the partition might be supplied by the mpirun scheduler plug-in.
--mode <smp | dual | vn | linux_smp>	The mode that the HTC partition is to be booted in. This parameter applies only when the --boot option is used. The default is smp.
--userlist <*user_list* | *ALL>	A comma-separated list of users and groups that can run jobs on the HTC partition. This parameter applies only when the --boot option is used. *ALL enables any user job to run. The default is that only the user that boots the partition can submit jobs to the partition. Prior to Blue Gene/P release V1R3M0, only user names could be specified for this parameter.

Parameter (and syntax)	Description
--host <*hostname*>	Service node host name that the mpirun server listens on. If not specified, the host name must be in the MMCS_SERVER_IP environment variable.
--port <*port*>	Service node TCP/IP port number that the mpirun server listens on. The default port is 9874.
--config <*path*>	mpirun configuration file, which contains the shared secret needed for **htcpartition** to authenticate with the mpirun daemon on the service node. If not specified, the mpirun configuration file is located by looking for these files in order: /etc/mpirun.cfg or <release-dir>/bin/mpirun.cfg (where <release-dir> is the Blue Gene/P system software directory, for example, /bgsys/drivers/V1R2M0_200_2008-080513P/ppc).
--trace <0-7>	Trace level. Higher numbers provide more tracing information. The default is 0.

Example G-1 shows how to boot a partition in SMP mode.

Example: G-1 Booting in SMP mode

```
$ htcpartition --boot --partition MYPARTITION
```

By default, **htcpartition** boots a partition so only the owner can run jobs on that partition. You can use the --userlist argument to add additional users so they can run jobs on the partition (see Example G-2).

Example: G-2 Adding users

```
$ htcpartition --boot --mode DUAL --partition MYPARTITION --userlist
“sam,tom,mark,brant”
```

Freeing a partition is shown in Example G-3.

Example: G-3 Free partition

```
$ htcpartition --free --partition MYPARTITION
```

H

Use of GNU profiling tool on Blue Gene/P

In this appendix we describe the GNU profiling toolchain for IBM Blue Gene/P.

For additional information about the usage of the GNU toolchain profiling tools, visit GNU gprof:

http://sourceware.org/binutils/docs-2.16/gprof/index.html

Speed your code with the GNU profiler:

http://www.ibm.com/developerworks/library/l-gnuprof.html

Profiling with the GNU toolchain

Profiling tools provide information about potential bottlenecks in your program; they help identify functions or sections of the code that might become good candidates to optimize. When using gmon profiling, three levels of profiling information can be generated, machine instruction level, procedure level, or full level. The choice of options depends on the amount of detail desired and the amount of overhead that is acceptable. Profiling with the GNU compiler set is usually enabled by adding **`-pg`** to the gcc compile flags.

Timer tick (machine instruction level) profiling

This level of profiling provides timer tick profiling information at the machine instruction level. To enable this type of profiling, add the -p option on the link command but no additional options on the compile commands:

- This level of profiling adds the least amount of performance collection overhead.
- It does not provide call graph information.

Procedure-level profiling with timer tick information

This level of profiling provides call graph information. To enable this level of profiling, include the -p option on all compile commands and on the link command. In addition to call-level profiling, you get profiling information at the machine instruction level:

- This level of profiling adds additional overhead during performance data collection.
- When using higher levels of optimization, the entire call flow might not be available due to inlining, code movement, scheduling, and other optimizations performed by the compiler.

Full level of profiling

To enable all available profiling for a program, add the -pg options to all compiles and links. Doing so provides profiling information that can be used to create call graph information, statement-level profiling, basic block profiling, and machine instruction profiling. This level of profiling introduces the most overhead while collecting performance data. When higher levels of compiler optimization are used, the statement mappings and procedure calls might not appear as expected due to inlining, code movement, scheduling, and other optimizations performed by the compiler.

Additional function in the Blue Gene/P gmon support

The basic gmon support is described in the man pages for the GNU toolchain:

`http://gcc.gnu.org/`

On Blue Gene/P, in addition to the functionality provided in the standard GNU toolchain, profiling information can be collected on each node. An application can run on multiple nodes, in which case profiling data is collected on each node of execution. To provide data for each node, gmon on Blue Gene/P generates a gmon.out file for each node where the application runs. The files is named gmon.out.x, where x is the rank of the node where profiling information was collected.

Enabling and disabling profiling within your application

To turn profiling on and off within your application, the application must still be compiled with the -p or -pg options as described previously. By inserting the following procedures at various points in the application, the user can enable and disable profile data collection and only collect data for the significant sections of the application:

- __moncontrol(1) turns profiling on
- __moncontrol(0) turns profiling off

Collecting the gmon data as a set of program counter values

Performance data can be collected in an alternate format, as a set of instruction addresses that were executing at the time of each sampling interval, instead of a summarized histogram. To enable this type of collection, set the environment variable GMON_SAMPLE_DATA="yes" before running your program. When data is collected this way, the output files are named gmon.sample.x instead of gmon.out.x. In most cases, this file is much smaller than the gmon.out.x file and also allows the user to see the sequence of execution samples instead of the summarized profile. The gprof tool in the Blue Gene/P toolchain has been updated to read this type of file.

Enabling profile data for threads in Blue Gene/P

Because enabling profiling on threads impacts the performance of nonprofiled runs, the thread profiling function is not included in the base gmon support. To do this type of profiling, an alternate toolchain must be built.

Enhancements to gprof in the Blue Gene/P toolchain

Because Blue Gene/P is a massivelly parallel system, the GNU toolchain requires additional functionality to collect profiling information about multiple nodes.

Using gprof to read gmon.sample.x files

The version of gprof in the Blue Gene/P toolchain has been modified to recognize and process gmon.sample.x files as described previously. When using gprof on a sample file, gprof generates the same type of report as it does for gmon.out.x files. If the -sum option is added, gprof generates a gmon.sum file that is in normal gmon.out format from the data in the gmon.sample.x file(s). The -d option displays the program counter values in the order in which they were collected.

Using gprof to merge a large number of gmon.out.x files

The base version of gprof has a limit on the number of gmon.out.x files that can be merged in one command invocation, which is due to the Linux limit on input arguments to a command.

The following new option has been added to gprof to allow merging of an unlimited number of gmon.out.x files:

```
> /bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-gprof -sumbg some.pgm
```

This command searches the current directory for all gmon.out files of the form gmon.out.x where x is an integer value, starting with 0 until a file in the sequence cannot be found. The data in these files is summed in the same way as gprof normally does.

As in the previous case, the following command searches the current directory for all gmon.sample files of the form gmon.sample.x where x is an integer value, starting with 0 until a file in the sequence cannot be in. A gmon histogram is generated by summing the data found in each individual file, and the output goes to gmon.sum.

```
> /bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-gprof -sumbg=gmon.sample
pgm
```

I

Statement of completion

IBM considers the IBM Blue Gene/P installation to be complete when the following activities have taken place:

- The Blue Gene/P rack or racks have been physically placed in position.
- The cabling is complete, including power, Ethernet, and torus cables.
- The Blue Gene/P racks can be powered on.
- All hardware is displayed in the Navigator and is available.

References

1. TOP500 Supercomputer sites:

   http://www.top500.org/

2. The MPI Forum. The MPI message-passing interface standard. May 1995:

   http://www.mcs.anl.gov/mpi/standard.html

3. OpenMP application programming interface (API):

   http://www.openmp.org

4. IBM XL family of compilers:

   – XL C/C++

     http://www-306.ibm.com/software/awdtools/xlcpp/

   – XL Fortran

     http://www-306.ibm.com/software/awdtools/fortran/xlfortran/features/bg/

5. GCC, the GNU Compiler Collection:

   http://gcc.gnu.org/

6. *IBM System Blue Gene Solution: Configuring and Maintaining Your Environment*, SG24-7352.
7. *GPFS Multicluster with the IBM System Blue Gene Solution and eHPS Clusters*, REDP-4168.
8. Engineering and Scientific Subroutine Library (ESSL):

   http://www.ibm.com/systems/p/software/essl.html

9. See note 2.
10. See note 3.
11. See note 4.
12. See note 5.
13. See note 6.
14. See note 7.
15. See note 8.
16. Gropp, W. and Lusk, E. "Dynamic Process Management in an MPI Setting." *7th IEEE Symposium on Parallel and Distributed Processing*. p. 530, 1995:

    http://www.cs.uiuc.edu/homes/wgropp/bib/papers/1995/sanantonio.pdf

17. See note 2.
18. See note 3.
19. See note 5.
20. See note 8.
21. Ganier, C J. "What is Direct Memory Access (DMA)?"

    http://cnx.org/content/m11867/latest/

22. See note 2.

23. See note 3.

24. A. Faraj, X. Yuan, and D. K. Lowenthal. “STAR-MPI: Self Tuned Adaptive Routines for MPI Collective Operations.” The 20th ACM International Conference on Supercomputing (ICS’ 06), Queensland, Australia, June 28-July 1, 2006.

25. Quinn, Michael J. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, New York, 2004. ISBN 0-072-82256-2.

26. Snir, Marc, et. al. *MPI: The Complete Reference, 2nd Edition, Volume 1*. MIT Press, Cambridge, Massachusetts, 1998. ISBN 0-262-69215-5.

27. Gropp, William, et. al. *MPI: The Complete Reference, Volume 2 - The MPI-2 Extensions*. MIT Press, Cambridge, Massachusetts, 1998. ISBN 0-262-69216-3.

28. See note 3.

29. See note 25.

30. Ibid.

31. Ibid.

32. See note 3.

33. Flynn’s taxonomy in Wikipedia:

    `http://en.wikipedia.org/wiki/Flynn%27s_Taxonomy`

34. Rennie, Gabriele. “Keeping an Eye on the Prize.” *Science and Technology Review*, July/August 2006:

    `http://www.llnl.gov/str/JulAug06/pdfs/07_06.3.pdf`

35. Rennie, Gabriele. “Simulating Materials for Nanostructural Designs.” *Science and Technology Review*, January/February 2006:

    `http://www.llnl.gov/str/JanFeb06/Schwegler.html`

36. SC06 Supercomputing Web site, press release from 16 November 2006:

    `http://sc06.supercomputing.org/news/press_release.php?id=14`

37. *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686

38. Sebastiani, D. and Rothlisberger, U. “Advances in Density-functional-based Modeling Techniques of the Car-Parrinello Approach,” chapter in *Quantum Medicinal Chemistry*, P. Carloni and F. Alber, eds. Wiley-VCH, Germany, 2003. ISBN 9-783-52730-456-1.

39. Car, R. and Parrinello, M. “Unified Approach for Molecular Dynamics and Density-Functional Theory.” *Physical Review Letter* 55, 2471 (1985):

    `http://prola.aps.org/abstract/PRL/v55/i22/p2471_1`

40. See note 34.

41. Suits, F., et al. “Overview of molecular dynamics techniques and early scientific results from the Blue Gene Project.” IBM Research & Development, 2005. 49, 475 (2005):

    `http://www.research.ibm.com/journal/rd/492/suits.pdf`

42. Ibid.

43. Case, D. A., et al. “The Amber biomolecular simulation programs.” *Journal of Computational Chemistry*. 26, 1668 (2005).

44. Fitch, B. G., et al. “Blue Matter, an application framework for molecular simulation on Blue Gene.” *Journal of Parallel and Distributed Computing*. 63, 759 (2003):

http://portal.acm.org/citation.cfm?id=952903.952912&dl=GUIDE&dl=ACM

45. Plimpton, S. “Fast parallel algorithms for short-range molecular dynamics.” *Journal of Computational Physics*. 117, 1 (1995).

46. Phillips, J., et al. “Scalable molecular dynamics with NAMD.” *Journal of Computational Chemistry*. 26, 1781 (2005).

47. See note 43.

48. See note 44.

49. Ibid.

50. Ibid.

51. Ibid.

52. Ibid.

53. See note 45.

54. LAMMPS Molecular Dynamics Simulator:

http://lammps.sandia.gov/

55. See note 46.

56. Brooks, B. R., et. al. “CHARMM. A Program for Macromolecular Energy, Minimization, and Dynamics Calculations.” *Journal of Computational Chemistry*. 4, 187 (1983).

57. Brünger, A. I. “X-PLOR, Version 3.1, A System for X-ray Crystallography and NMR.” 1992: The Howard Hughes Medical Institute and Department of Molecular Biophysics and Biochemistry, Yale University. 405.

58. Kumar, S., et al. “Achieving Strong Scaling with NAMD on Blue Gene/L.” *Proceedings of IEEE International Parallel & Distributed Processing Symposium*, 2006.

59. Waszkowycz, B., et al. “Large-scale Virtual Screening for Discovering Leads in the Postgenomic Era.” *IBM Systems Journal*. 40, 360 (2001).

60. Patrick, G. L. *An Introduction to Medicinal Chemistry, 3rd Edition*. Oxford University Press, Oxford, UK, 2005. ISBN 0-199-27500-9.

61. Kontoyianni, M., et al. “Evaluation of Docking Performance: Comparative Data on Docking Algorithms.” *Journal of Medical Chemistry*. 47, 558 (2004).

62. Kuntz, D., et al. “A Geometric Approach to Macromolecule-ligand Interactions.” *Journal of Molecular Biology*. 161, 269 (1982); Morris, G. M., et al. “Automated Docking Using a Lamarckian Genetic Algorithm and Empirical Binding Free Energy Function.” *Journal of Computational Chemistry*. 19, 1639 (1998); Jones, G., et al. “Development and Validation of a Genetic Algorithm to Flexible Docking.” *Journal of Molecular Biology*. 267, 904 (1997); Rarey, M., et al. “A Fast Flexible Docking Method Using an Incremental Construction Algorithm.” *Journal of Molecular Biology*. 261, 470 (1996), Scrödinger, Portland, OR 972001; Pang, Y. P., et al. “EUDOC: A Computer Program for Identification of Drug Interaction Sites in Macromolecules and Drug Leads from Chemical Databases.” *Journal of Computational Chemistry*. 22, 1750 (2001).

63. (a) http://dock.compbio.ucsf.edu; (b) Moustakas, D. T., et al. “Development and Validation of a Modular, Extensible Docking Program: DOCK5.” *Journal of Computational Aided Molecular Design*. 20, 601 (2006).

64. Ibid.

65. Ibid.

66. Ibid.

67. Ibid.

68. Peters, A., et al., “High Throughput Computing Validation for Drug Discovery using the DOCK Program on a Massively Parallel System.” *1st Annual MSCBB*. Northwestern University, Evanston, IL, September, 2007.

69. Irwin, J. J. and Shoichet, B. K. “ZINC - A Free Database of Commercially Available Compounds for Virtual Screening.” *Journal of Chemical Information and Modeling*. 45, 177 (2005).

70. Ibid.

71. Pople, J. A. *Approximate Molecular Orbital Theory (Advanced Chemistry)*. McGraw-Hill, NY. June 1970. ISBN 0-070-50512-8.

72. See note 39.

73. (a) CPMD V3.9, Copyright IBM Corp. 1990-2003, Copyright MPI fur Festkorperforschung, Stuttgart, 1997-2001. (b) See also:

    `http://www.cpmd.org`

74. Marx, D. and Hutter, J. *Ab-initio molecular dynamics: Theory and implementation* in *Modern Methods and Algorithms of Quantum Chemistry*. J. Grotendorst (ed.), NIC Series, 1, FZ Julich, Germany, 2000. See also the following URL and references therein:

    `http://www.fz-juelich.de/nic-series/Volume3/marx.pdf`

75. Vanderbilt, D. “Soft self-consistent pseudopotentials in a generalized eigenvalue formalism.” *Physical Review B*. 1990, 41, 7892 (1990):

    `http://prola.aps.org/abstract/PRB/v41/i11/p7892_1`

76. See note 73.

77. Eddy, S. R., *HMMER User’s Guide. Biological Sequence Analysis Using Profile Hidden Markov Models*, Version 2.3.2, October 1998.

78. Ibid.

79. Ibid.

80. Jiang, K., et al. “An Efficient Parallel Implementation of the Hidden Markov Methods for Genomic Sequence Search on a Massively Parallel System.” *IEEE Transactions on Parallel and Distributed Systems*. 19, 1 (2008).

81. Bateman, A., et al. “The Pfam Protein Families Database.” *Nucleic Acids Research*. 30, 276 (2002).

82. Ibid.

83. Darling, A., et al. “The Design, Implementation, and Evaluation of mpiBLAST.” *Proceedings of 4th International Conference on Linux Clusters* (in conjunction with ClusterWorld Conference & Expo), 2003.

84. Thorsen, O., et al. “Parallel genomic sequence-search on a massively parallel system.” *Conference on Computing Frontiers: Proceedings of the 4th International Conference on Computing Frontiers*. ACM, 2007, pp. 59-68.

85. Heyman, J. “Recommendations for Porting Open Source Software (OSS) to Blue Gene/P,” white paper WP101152:

    `http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP101152`

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

For information about ordering these publications, see “How to get IBM Redbooks” on page 374. Note that some of the documents referenced here might be available in softcopy only:

- *IBM System Blue Gene Solution: Blue Gene/P Safety Considerations*, REDP-4257
- *Blue Gene/L: Hardware Overview and Planning*, SG24-6796
- *Blue Gene/L: Performance Analysis Tools*, SG24-7278
- *Evolution of the IBM System Blue Gene Solution*, REDP-4247
- *GPFS Multicluster with the IBM System Blue Gene Solution and eHPS Clusters*, REDP-4168
- *IBM System Blue Gene Solution: Application Development*, SG24-7179
- *IBM System Blue Gene Solution: Configuring and Maintaining Your Environment*, SG24-7352
- *IBM System Blue Gene Solution: Hardware Installation and Serviceability*, SG24-6743
- *IBM System Blue Gene Solution Problem Determination Guide*, SG24-7211
- *IBM System Blue Gene Solution: System Administration*, SG24-7178
- *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686

Other publications

These publications are also relevant as further information sources:

- Bateman, A., et al. “The Pfam Protein Families Database.” *Nucleic Acids Research*. 30, 276 (2002).
- Brooks, B. R.; Bruccoleri, R. E.; Olafson, B. D.; States, D. J.; Swaminathan, S.; Karplus, M. “CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations.” *Journal of Computational Chemistry*. 4, 187 (1983).
- Brünger, A. I. “X-PLOR, Version 3.1, A System for X-ray Crystallography and NMR.” 1992: The Howard Hughes Medical Institute and Department of Molecular Biophysics and Biochemistry, Yale University. 405.
- Car, R. and Parrinello, Mi. “Unified Approach for Molecular Dynamics and Density-Functional Theory.” *Physical Review Letter* 55, 2471 (1985):

  http://prola.aps.org/abstract/PRL/v55/i22/p2471_1
- Case, D. A., et al. “The Amber biomolecular simulation programs.” *Journal of Computational Chemistry*. 26, 1668 (2005).

- Darling, A., et al. “The Design, Implementation, and Evaluation of mpiBLAST.” *Proceedings of 4th International Conference on Linux Clusters* (in conjunction with ClusterWorld Conference & Expo), 2003.
- Eddy, S. R., *HMMER User’s Guide. Biological Sequence Analysis Using Profile Hidden Markov Models*, Version 2.3.2, October 1998.
- Fitch, B. G., et al. “Blue Matter, an application framework for molecular simulation on Blue Gene.” *Journal of Parallel and Distributed Computing*. 63, 759 (2003).
- Gropp, W. and Lusk, E. “Dynamic Process Management in an MPI Setting.” *7th IEEE Symposium on Parallel and Distributed Processing*. p. 530, 1995:

  http://www.cs.uiuc.edu/homes/wgropp/bib/papers/1995/sanantonio.pdf
- Gropp, William; Huss-Lederman, Steven; Lumsdaine, Andrew; Lusk, Ewing; Nitzberg, Bill; Saphir, William; Snir, Marc. *MPI: The Complete Reference, Volume 2 - The MPI-2 Extensions*. MIT Press, Cambridge, Massachusetts, 1998. ISBN 0-262-69216-3.
- Heyman, J. “Recommendations for Porting Open Source Software (OSS) to Blue Gene/P,” white paper WP101152.

  http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP101152
- Irwin, J. J. and Shoichet, B. K. “ZINC - A Free Database of Commercially Available Compounds for Virtual Screening.” *Journal of Chemical Information and Modeling*. 45, 177 (2005).
- Jiang, K., et al. “An Efficient Parallel Implementation of the Hidden Markov Methods for Genomic Sequence Search on a Massively Parallel System.” *IEEE Transactions On Parallel and Distributed Systems*. 19, 1 (2008).
- Jones, G., et al. “Development and Validation of a Genetic Algorithm to Flexible Docking.” *Journal of Molecular Biology*. 267, 904 (1997).
- Kontoyianni, M., et al. “Evaluation of Docking Performance: Comparative Data on Docking Algorithms.” *Journal of Medical Chemistry*. 47, 558 (2004).
- Kumar, S., et al. “Achieving Strong Scaling with NAMD on Blue Gene/L.” *Proceedings of IEEE International Parallel & Distributed Processing Symposium*, 2006.
- Kuntz, D., et al. “A Geometric Approach to Macromolecule-ligand Interactions.” *Journal of Molecular Biology*. 161, 269 (1982).
- Marx, D. and Hutter, J. *Ab-initio molecular dynamics: Theory and implementation, in: Modern Methods and Algorithms of Quantum Chemistry*. J. Grotendorst (ed.), NIC Series, 1, FZ Julich, Germany, 2000:

  http://www.fz-juelich.de/nic-series/Volume3/marx.pdf
- Morris, G. M., et al. “Automated Docking Using a Lamarckian Genetic Algorithm and Empirical Binding Free Energy Function.” *Journal of Computational Chemistry*. 19, 1639 (1998).
- Pang, Y. P., et al. “EUDOC: A Computer Program for Identification of Drug Interaction Sites in Macromolecules and Drug Leads from Chemical Databases.” *Journal of Computational Chemistry*. 22, 1750 (2001).
- Patrick, G. L. *An Introduction to Medicinal Chemistry, 3rd Edition*. Oxford University Press, Oxford, UK, 2005. ISBN 0-199-27500-9.
- Peters, A., et al., “High Throughput Computing Validation for Drug Discovery using the DOCK Program on a Massively Parallel System.” *1st Annual MSCBB* - Location: Northwestern University - Evanston, IL, September, 2007.
- Phillips, J., et al. “Scalable molecular dynamics with NAMD.” *Journal of Computational Chemistry*. 26, 1781 (2005).

- Plimpton, S. “Fast parallel algorithms for short-range molecular dynamics.” *Journal of Computational Physics*. 117, 1 (1995).
- Pople, J. A. *Approximate Molecular Orbital Theory (Advanced Chemistry)*. McGraw-Hill, NY. June 1970. ISBN 0-070-50512-8.
- Quinn, Michael J. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, New York, 2004. ISBN 0-072-82256-2.
- Rarey, M., et al. “A Fast Flexible Docking Method Using an Incremental Construction Algorithm.” *Journal of Molecular Biology*. 261, 470 (1996), Scrödinger, Portland, OR 972001.
- Sebastiani, D. and Rothlisberger, U. “Advances in Density-functional-based Modeling Techniques of the Car-Parrinello Approach,” chapter in *Quantum Medicinal Chemistry*, P. Carloni and F. Alber (eds.), Wiley-VCH, Germany, 2003. ISBN 9-783-52730-456-1.
- Snir, Marc; Otto, Steve; Huss-Lederman, Steven; Walker, David; Dongarra, Jack. *MPI: The Complete Reference, 2nd Edition, Volume 1*. MIT Press, Cambridge, Massachusetts, 1998. ISBN 0-262-69215-5.
- Suits, F., et al. “Overview of Molecular Dynamics Techniques and Early Scientific Results from the Blue Gene Project.” IBM Research & Development, 2005. 49, 475 (2005).

  http://www.research.ibm.com/journal/rd/492/suits.pdf
- Thorsen, O., et al. “Parallel genomic sequence-search on a massively parallel system.” *Conference on Computing Frontiers: Proceedings of the 4th International Conference on Computing Frontiers*. ACM, 2007, pp. 59-68.
- Vanderbilt, D. “Soft self-consistent pseudopotentials in a generalized eigenvalue formalism.” *Physical Review B*. 1990, 41, 7892 (1990).

  http://prola.aps.org/abstract/PRB/v41/i11/p7892_1
- Waszkowycz, B., et al. “Large-scale Virtual Screening for Discovering Leads in the Postgenomic Era.” *IBM Systems Journal*. 40, 360 (2001).

Online resources

These Web sites are also relevant as further information sources:

- Compiler-related topics:
  - XL C/C++

    http://www-306.ibm.com/software/awdtools/xlcpp/
  - XL C/C++ library

    http://www.ibm.com/software/awdtools/xlcpp/library/
  - XL Fortran Advanced Edition for Blue Gene

    http://www-306.ibm.com/software/awdtools/fortran/xlfortran/features/bg/
  - XL Fortran library

    http://www-306.ibm.com/software/awdtools/fortran/xlfortran/library/

- Debugger-related topics:
  - GDB: The GNU Project Debugger

    http://www.gnu.org/software/gdb/gdb.html
  - GDB documentation:

    http://www.gnu.org/software/gdb/documentation/
- Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL

  http://www.ibm.com/systems/p/software/essl.html
- GCC, the GNU Compiler Collection

  http://gcc.gnu.org/
- Intel MPI Benchmarks is formerly known as "Pallas MPI Benchmarks."

  http://www.intel.com/cd/software/products/asmo-na/eng/219848.htm
- Mathematical Acceleration Subsystem

  http://www-306.ibm.com/software/awdtools/mass/index.html
- Message Passing Interface Forum

  http://www.mpi-forum.org/
- MPI Performance Topics

  http://www.llnl.gov/computing/tutorials/mpi_performance/
- The OpenMP API Specification:

  http://www.openmp.org
- Danier, CJ, "What is Direct Memory Access (DMA)?"

  http://cnx.org/content/m11867/latest/

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

IBM System Blue Gene Solution: Blue Gene/P Application Development

(0.5” spine)
0.475”<->0.873”
250 <-> 459 pages

IBM System Blue Gene Solution: Blue Gene/P Application Development

Understand the Blue Gene/P programming environment

Learn how to run and debug MPI programs

Learn about Bridge and Real-time APIs

This IBM® Redbooks® publication is one in a series of IBM books written specifically for the IBM System Blue Gene/P Solution. The Blue Gene/P system is the second generation of a massively parallel supercomputer from IBM in the IBM System Blue Gene Solution series. In this book, we provide an overview of the application development environment for the Blue Gene/P system. We intend to help programmers understand the requirements to develop applications on this high-performance massively parallel supercomputer.

In this book, we explain instances where the Blue Gene/P system is unique in its programming environment. We also attempt to look at the differences between the IBM System Blue Gene/L Solution and the Blue Gene/P Solution. In this book, we do not delve into great depth about the technologies that are commonly used in the supercomputing industry, such as Message Passing Interface (MPI) and Open Multi-Processing (OpenMP), nor do we try to teach parallel programming. References are provided in those instances for you to find more information if necessary.

Prior to reading this book, you must have a strong background in high-performance computing (HPC) programming. The high-level programming languages that we use throughout this book are C/C++ and Fortran95. Previous experience using the Blue Gene/L system can help you better understand some concepts in this book that we do not extensively discuss. However, several IBM Redbooks publications about the Blue Gene/L system are available for you to obtain general information about the Blue Gene/L system. We recommend that you refer to “IBM Redbooks” on page 371 for a list of those publications.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-7287-03 ISBN 0738433330