Blue Gene/P

Recommendations for Porting Open Source Software (OSS) to Blue Gene/P

Version Date: December 4, 2007

Business Systems Enablement

Jerrold Heyman
Technical Consultant
jheyman@us.ibm.com

Trademarks

1 Introduction and Acknowledgments

Migrating applications from a Linux cluster to IBM's Blue Gene/P can be a beneficial effort, but it is not an entirely painless effort. Each of the Open Source Software (OSS) components that the application is dependent upon requires a separate port – some of which will be easy and others which might not. The degree of difficulty derives from the fact that Blue Gene/P does not come with the OSS that is installed with any standard Linux distribution. Additionally Blue Gene/P requires the developer to compile and debug the application on a POWER-based front-end node that runs Novell's SuSE Linux Enterprise Server (SLES) 10. Application development is done with a cross-compiler (GNU Compiler Collection or IBM's proprietary XL C and FORTRAN compilers) that generates Blue Gene/P output.

The Blue Gene/P super computer is a massively parallel processing (MPP) computer based around IBM's POWER architecture, and the successor to the Blue Gene/L (introduced in 2004). The cross-compilation environment creates code that will not execute in the SLES 10 environment. Software developers need to be aware that the standard *configure* based scripts will not quite support the effort.

Thank you to the following reviewers:

Carlos Sosa, IBM Computational Chemistry and Biology Software Engineer
Billy Robinson, IBM Petroleum Seismic Processing Software Engineering Consultant
Michael T Nelson, IBM Software Developer

2 Open Source Software on BG/P

The GNU standard *configure* process (generated via the **autoconf** package) attempts to determine the *platform* that the code is being compiled on and create the appropriate header and Makefile(s). The platform is composed of the operating system, the central processing unit (CPU), and the capabilities of the operating system version in use. It is this last part, the capabilities, where using the *configure* process to generate Blue Gene specific code has a problem. The logic of the *configure* process dictates that compilation (and at times execution) of simple programs occur to test each feature.

One of the first things the *configure* process does is determine the compiler in order to test for the other capabilities of the operating system. The compiler search can be overridden by either including **CC=<value>** on the command line or exporting **CC=<value>** into the environment – unfortunately it is different for each OSS project. If the developer overrides the search for the compiler by using the **CC** variable pointing to the Blue Gene cross compilers, then the *configure* process will fail (error exit) the first time that it attempts to execute the resulting binary. Unfortunately, permitting the *configure* process discover the compiler (gcc on the SLES 10 system), will result in Makefile(s) that point to the wrong compiler. Neither approach is truly viable, nor is it obvious that there is a good solution to this problem.

The configure process generates shared libraries by default for most, if not all, OSS projects. The default behavior of the Blue Gene compilers is to generate a static library. This is true of both the IBM XL compilers and of the GNU Compiler Collection. This will lead to obvious problems, and is more fully addressed in Section 4 Shared Libraries..

2.1 MPI enabled compiler solution

If the application in question uses Message Passing Interface (MPI), then the developer has the option of making use of the standard *mpi** scripts found in **/bgsys/drivers/ppcfloor/comm/bin**. Much more detail on the *mpi** scripts is available in chapter 7 of the IBM Redbook entitled *IBM System Blue Gene Solution: Blue Gene/P Application Development.* If the developer decides to avoid the *mpi** scripts, they are then required to specify the additional headers and libraries required to successfully compile and link the application. Note that use of these standard scripts precludes using them as the **CC** value during the execution of the *configure* script as they will invoke the Blue Gene cross compiler and create executables that will not run on SLES 10.

There are quite a number of *mpi** scripts; therefore understanding which compiler is invoked becomes critical. To invoke the IBM XL compilers, the developer would use (and substitute) *mpixlc*, *mpixlcxx*, and *mpixlf* to replace xlc, xlC, and xlf. Note that there are several additional xlf versions corresponding to the different FORTRAN language standard levels. The additional *mpi*_r* compiler scripts invoke (and link) with the thread safe version of the compiler and libraries. The performance of the Blue Gene/P binaries is enhanced as the XL compilers generate single instruction multiple data (SIMD) execution streams, which in turn utilize the second floating point processor contained within the CPU core.

To invoke the GNU compilers, you would use (and substitute) *mpicc*, *mpicxx*, and *mpif77* to replace gcc, g++, and gfortran.

Using the mpi* scripts has the advantage that all the headers and link libraries necessary for support of MPI applications are added to the compilation by default.

2.2 IBM XL Compiler solution

If the message passing interface (MPI) is not required, then the developer can invoke the IBM XL compilers directly. The developer could use the **CC=** and **CXX=** on the command line, or export those same variables into their environment, pointing directly to the compiler(s). As mentioned in the previous section, pointing these to the actual Blue Gene compilers will cause the *configure* process to fail. Instead, the developer should point these variables to the IBM XL compilers by using the full path to the compilers:

```
CC=/opt/ibmcmp/vacpp/bg/9.0/bin/xlc
CXX=/opt/ibmcmp/vacpp/bg/9.0/bin/xlC
F77=/opt/ibmcmp/xlf/bg/11.1/bin/xlf
```

The binaries that are created during the running of the *configure* process will execute correctly and generation of the Makefile(s) will occur as expected. The performance of the Blue Gene/P binaries is enhanced (compared to the GNU Compiler Collection) as the XL compilers generate more processor specific optimized code and also generate single instruction multiple data (SIMD) execution streams. Additionally, the correct path is appended for library searches. The necessary edits to the resulting Makefile(s) is/are minimal as the only changes need to be made are

```
xlc -> bgxlc
xlC -> bgxlC
xlf -> bgxlf
```

There are other ways to invoke the compilers (xlf90/xlf95/xlf2003 depending on the language level of FORTRAN needed, c89/c99/cc depending on the language level for C needed, and _r versions of all compilers if thread safe code is necessary). A simple *sed* script can be created for the appropriate substitutions. Compiling the code then becomes the simple exercise of invoking the appropriate make rule.

Overriding the default compiler and linker flags can also be done at the command line, utilizing the **CFLAGS**, **CXXFLAGS**, and **LDFLAGS** variables or exporting the same variables into the environment. Compiler options included in the **CFLAGS** and **CXXFLAGS** variables are used by the configure process, replacing any values normally generated. The most common of these option include optimization: *-Ox* (where x is the level of optimization) and *-qarch=450* (generate BG/P specific architecture instructions). OpenMP applications require the additional support of *-qsmp=omp:noauto –qthreaded* compiler options.

2.3 GNU Compiler Collection solution

If the application cannot be compiled with the IBM XL compilers, or there are other reasons for using the GNU Compiler Collection (gcc), allowing the *configure* script to locate the compiler(s) also allows the process to complete and generate the necessary Makefile(s). Editing of the Makefile(s) becomes necessary, with the changes being a little more extensive. References with the Makefile(s) to the **CC** variable will have to be altered from *gcc* to the fully qualified location of *powerpc-bgp-linux-gcc*.

CC=gcc becomes CC=/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-gcc

C++ changes are:

CXX=gcc or CXX=g++ or CXX=c++ becomes
CXX=/ bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-[g++ | gcc | c++]

The FORTRAN changes are:

F77=gfortran becomes
F77=/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-gfortran

3 The Configure Process

The normal process used for installation of an open source package is to:

```
$ ./configure [options]
$ ./make
$ ./make test
$ ./make install
```

The default installation location, by convention, is **/usr/local**. This particular location is of no value as Blue Gene/P compute nodes only access file systems that are mounted. A developer running an application must make sure that they have copied the application binary, configuration files, and any input data required to the Blue Gene/P mounted file system. As the default installation location is not valid for Blue Gene/P, the developer will have to decide if they actually need to do the installation. If so, then they need to install on the file system that the Blue Gene/P has mounted using the –prefix command line option to the *configure* process. If the developer makes the determination that installation is not required, then all subsequent references to the open source project will require the fully qualified path for reference.

3.1 Simple Example:

An image processing application to be ported to Blue Gene/P and has a dependency on *libtiff.a.* The open source package that creates libtiff.a is **tiff-3.8.2**, and it in turn has a dependency on *libz.a*. In this example, the package **zlib-1.2.1** will be used to resolve the libtiff.a dependency.

First download zlib-1.2.1. Then untar the source, and run the *configure* process. As the *make test* target will not be executed, point the **CC** variable on the *configure* process command line to the IBM XL C compiler for Blue Gene/P:

```
$ ./configure CC=/opt/ibmcmp/vacpp/bg/9.0/bin/bgxlc
$ make > make.out 2>&1
```

Next download the tiff-3.8.2 source. Then untar the source, and run the *configure* process. Since the tiff package creates executables, do not use the **CC** variable to point to the Blue Gene/P compiler, but point it to the IBM XL C compiler for Linux. Additionally, since the application needs to support C++ interfaces, also specify the **CXX** macro. Lastly, point the build to the libz.a that was just created and disable some features for which the application has no use.

```
 $ ./configure --disable-shared --disable-jpeg \
--enable-cxx \
--with-zlib-include-dir=/bgusr2/heymanj/OCI/zlib-1.2.1 \
--with-zlib-lib-dir=/bgusr2/heymanj/OCI/zlib-1.2.1 \
--disable-pixarlog \
CC=/opt/ibmcmp/vacpp/bg/9.0/bin/xlc \
CXX=/opt/ibmcmp/vacpp/bg/9.0/bin/xlC
```

After the *configure* process has completed, use the technique mentioned earlier in the section entitled 'IBM XL Compiler Solution' and edit all the generated Makefiles changing the xlc/xlC to bgxlc/bgxlC. Build the tiff package:

```
$ ./make > make.out 2>&1
```

After the build is completed, there should now be a working libtiff.a (by specifying –disable-shared) and a valid libz.a that can be utilized for the application.

3.2 More complicated example

In section 3.1, the simple example of building a library for use on Blue Gene/P is shown. The combination of the two packages (zlib-1.2.1 and tiff-3.8.2) includes creation of some additional binaries during the make process, and the steps in section 3.1 will build those binaries such that they can only execute on the Blue Gene/P hardware. Obviously, executing some of those binaries on Blue Gene/P would **not** be the preferred way to access the results. Creating the binaries for a given package (as opposed to the libraries) should be targeted for either the front-end node, running SuSE SLES 10, or other platform where analysis would be done.

The Hierarchical Data Format (HDF) is a perfect example of this. The HDF5 package (found at http://www.hdf-group.org/index.html) is made up of libraries that provide both serial and parallel output and a set of utilities to parse, compare, and list the objects stored within the hdf5 files. The availability of the libraries for Blue Gene/P applications is important for some applications that will execute on Blue Gene/P. The availability of the utilities executing on Blue Gene/P does not make much sense as they are interactive, command-line tools which do not take advantage of the MPP capability of Blue Gene/P. Building the libraries/tools is a two phased approach where a complete build is done for the platform where the tools will be executed, and a second build for via the Blue Gene/P cross-compiler to create the libraries for linkage to other Blue Gene/P applications.

The HDF libraries are the critical part for a Blue Gene/P application reading or writing HDF5 files. The configuration options may be different for each installation, depending on the support necessary. In the example support for the FORTRAN API and parallel I/O is compiled in. Zlib compression, the Stream Virtual File Driver, and shared library creation are disabled. Executing the *configure* process would entail the following options:

```
$ F9X=/bgsys/drivers/ppcfloo/comm/bin/mpixlf90; export F9X
$ ./configure --enable-fortran \
--enable-linux-lfs --enable-production \
--enable-parallel --disable-stream-vfd \
--without-zlib --disable-shared \
--host=powerpc-bgp-linux-gnu \
CC=/bgsys/drivers/ppcfloor/comm/bin/mpixlc \
CXX=/bgsys/drivers/ppcfloor/comm/bin/mpixlcxx
```

By specifying the *–host* option, the configure process scans the **config** subdirectory for the platform specified. If the file exists, then the information in that file is processed to include specific values that then do not have to be computed via the process. If the file does not exist, then all the computations must still be performed.

Creating this file, **powerpc-bgp-linux-gnu**, from scratch is not difficult. Once created, the same file can be used for other *configure* based projects. Each project is different, but at a minimum, the file can contain:

```
ac_cv_c_bigendian=${ac_cv_c_bigendian=  yes  }
ac_cv_header_stdc=${ac_cv_header_stdc=  yes  }
ac_cv_header_sys_ioctl_h=${ac_cv_header=  yes  }
ac_cv_func_fork=${ac_cv_func_fork=  no  }
```

Since Blue Gene/P is a 32bit machine, it is easier to supply the values of all the data types than letting the configure process figure them out.

```
ac_cv_sizeof_char=${ac_cv_sizeof_char=1}
ac_cv_sizeof_short=${ac_cv_sizeof_short=2}
ac_cv_sizeof_int=${ac_cv_sizeof_int=4}
ac_cv_sizeof_long=${ac_cv_sizeof_long=4}
ac_cv_sizeof_long_long=${ac_cv_sizeof_long_long=8}
ac_cv_sizeof___int64=${ac_cv_sizeof___int64=8}
ac_cv_sizeof_float=${ac_cv_sizeof_float=4}
ac_cv_sizeof_double=${ac_cv_sizeof_double=8}
ac_cv_sizeof_long_double=${ac_cv_sizeof_long_double=8}
ac_cv_sizeof_int8_t=${ac_cv_sizeof_int8_t=1}
ac_cv_sizeof_uint8_t=${ac_cv_sizeof_uint8_t=1}
ac_cv_sizeof_int_least8_t=${ac_cv_sizeof_int_least8_t=1}
ac_cv_sizeof_uint_least8_t=${ac_cv_sizeof_uint_least8_t=1}
ac_cv_sizeof_int_fast8_t=${ac_cv_sizeof_int_fast8_t=1}
ac_cv_sizeof_uint_fast8_t=${ac_cv_sizeof_uint_fast8_t=1}
ac_cv_sizeof_int16_t=${ac_cv_sizeof_int16_t=2}
ac_cv_sizeof_uint16_t=${ac_cv_sizeof_uint16_t=2}
ac_cv_sizeof_int_least16_t=${ac_cv_sizeof_int_least16_t=2}
ac_cv_sizeof_uint_least16_t=${ac_cv_sizeof_uint_least16_t=2}
ac_cv_sizeof_int_fast16_t=${ac_cv_sizeof_int_fast16_t=4}
ac_cv_sizeof_uint_fast16_t=${ac_cv_sizeof_uint_fast16_t=4}
ac_cv_sizeof_int32_t=${ac_cv_sizeof_int32_t=4}
ac_cv_sizeof_uint32_t=${ac_cv_sizeof_uint32_t=4}
ac_cv_sizeof_int_least32_t=${ac_cv_sizeof_int_least32_t=4}
ac_cv_sizeof_uint_least32_t=${ac_cv_sizeof_uint_least32_t=4}
ac_cv_sizeof_int_fast32_t=${ac_cv_sizeof_int_fast32_t=4}
ac_cv_sizeof_uint_fast32_t=${ac_cv_sizeof_uint_fast32_t=4}
ac_cv_sizeof_int64_t=${ac_cv_sizeof_int64_t=8}
ac_cv_sizeof_uint64_t=${ac_cv_sizeof_uint64_t=8}
ac_cv_sizeof_int_least64_t=${ac_cv_sizeof_int_least64_t=8}
ac_cv_sizeof_uint_least64_t=${ac_cv_sizeof_uint_least64_t=8}
ac_cv_sizeof_int_fast64_t=${ac_cv_sizeof_int_fast64_t=8}
ac_cv_sizeof_uint_fast64_t=${ac_cv_sizeof_uint_fast64_t=8}

ac_cv_sizeof_size_t=${ac_cv_sizeof_size_t=4}
```

```
ac_cv_sizeof_ssize_t=${ac_cv_sizeof_ssize_t=4}
ac_cv_sizeof_off_t=${ac_cv_sizeof_off_t=8}
```

Once the file is created, the –host option can now be specified on the command line. Many more things can be added to the file, but most are project related so learning to read through the *configure* script will be necessary.

4 Shared Libraries

With the release of Blue Gene/P support for using shared libraries was added. This can be both an advantage and a disadvantage – depending on the developer's perspective. Many open source packages (subsequently referred to as simply *package*) create shared libraries as the default, and either create static at the same time or require a command line option to the configure process to have the static libraries created. This runs counter to the default behavior of the Blue Gene/P compilers that create static libraries and binaries by default.

4.1 Creating a shared library

To create a shared library, the developer has to invoke the correct option with the associated compiler. For the XL compilers, the option is '-G –o <libname>.so'.
The basic syntax is:

```
$ bgxlc -G -o lib<name>.so <list of object files>
```

A specific example is:

```
$ bgxlc -G -o libjerry.so.1 *.o
```

This would take all the object files (*.o) found in the current directory and create a shared library called libjerry.so.1. These object files must have been previously compiled with the **–qpic** option which generates the necessary position independent code (**pic**) used in the creation of Linux shared libraries.

To create a shared library under the GNU compiler collection, the developer would use the basic syntax

```
$ gcc -shared -Wl,-soname,your_soname \
    -o library_name file_list [library_list]
```

A specific example

```
$ powerpc-bgp-linux-gcc -shared -Wl,-soname,libjerry.so.1 \
    -o libjerry.so.1 *.o
```

This would take all the object files (which need to be compiled with the **–fpic** and/or the **–fPIC** option) found in the current directory and create a shared library called libjerry.so. 1

4.2 Linking with shared libraries

To link a binary with a shared library, the LDFLAGS variable is set to *–dy* if linking with any of the GNU compilers. Linking with the IBM XL Compilers requires the LDFLAGS variable set to *–qnostaticlink* for xlc and xlC (C and C++), and setting LDFLAGS to *–Wl, -dy* for linking with XL FORTRAN.

Assume that the libraries built in Section 3.1 were built as shared libraries. The following is the command necessary when using the IBM XL C/C++ compiler

```
$ xlc -qnostaticlink   o <binary> \
  <list of object files> \
   -L/bgusr2/heymanj/OCI/zlib-1.2.1   lz
```

The following is the command necessary when using any of the GNU compilers

```
$ powerpc-bgp-linux-gcc   dy   o <binary> \
  <list of object files> \
  -L/bgusr2/heymanj/OCI/zlib-1.2.1 -lz
```

Both these commands have the linker look in /bgusr2/heymanj/OCI/zlib-1.2.1 for libz.so. The link will fail if the library doesn't exist. Likewise, execution of the resulting binary is contingent on libz.so being available on a shared file system for Blue Gene/P to load it.

4.3 Use shared libraries?

Paraphrasing Shakespeare, the question becomes "To use shared libraries or not use shared libraries." If shared libraries are supported by the package and by Blue Gene/P, the obvious question would be why not use them? The answer to the question is not a simple yes or no. Using shared libraries is going to depend on how large the application in question scales (how many nodes are going to be utilized) and which CPU mode the application runs under. The implication is that for each application executable instance initiated, the shared libraries must also be loaded. If executing in Virtual Node (-mode VN) mode, then the shared libraries are loaded four times per node (4 separately running cores). If executing in Symmetric Multi-Processing (-mode SMP) mode, then the shared libraries are loaded once for the entire node.

Shared libraries will also have to be located on the file system that is mounted by the Blue Gene/P partition for the same reason as discussed in the Installation section. During the invocation (via the *mpirun* command), the additional '-env' option must be specified.

This option would require the use of the LD_LIBRARY_PATH environment variable. The syntax would be:

```
$ mpirun <other options> -env \
  LD_LIBRARY_PATH=$LD_LIBRARY_PATH:<path to the shared
libraries>”
```

An example based on the image processing application mentioned in the Installation section (assuming it was compiled with shared library support):

```
$ mpirun <other options> -env \
  LD_LIBRAY_PATH=$LD_LIBRARY_PATH:/bgusr2/heymanj/OCI
```

This presumes that all the shared libraries required for the application have been copied to the /bgusr2/heymanj/OCI directory.

5 Conclusions

In summary, open source packages port rather easily to Blue Gene/P if they have already been ported to Linux on Power - and can be built rather quickly. Several of the decision points that will have to be made during the port were discussed along with enumerating the alternative solutions. These decision points include how to make configure scripts run, which compiler to use, and the use of shared libraries. The developers involved will have to be the final arbiter in determining which of the alternatives detailed in this paper are best for their given application.

6 Appendix (full example)

As mentioned, each OSS project is different and the suggestions given in this paper may or may not apply. Utilizing a small OSS project as an example should give the reader a better understanding of how much can be done by rote and how much needs to be dynamic.

Using the OSS project tcl 8.4.16 as an example, it should become apparent what steps will need to be addressed by the reader when it comes to their particular OSS project of choice.

6.1 Shared Library

1) download stable (tcl8.4.16-src.tar.gz at this time) from http://www.tcl.tk/software/tcltk and copy to frontend node
2) execute *tar –zxvf tcl8.4.16-src.tar.gz*
3) *cd tcl8.4.16/unix*
4) CC=mpixlc; export CC
5) PATH=/bgsys/drivers/ppcfloor/comm/bin:$PATH; export PATH
6) ./configure –enable-shared –enable-threads

(configure process executes)

7) edit Makefile:
8) search for xlc – replace with mpixlc
9) search for LDFLAGS and comment out existing line; make a copy of that line directly beneath it and replace *-Wl,--export-dynamic* with *–qnostaticlink*
10) seach for SHLIB_LD and comment out existing line; make a copy of that line directly beneath it and replace *–shared* with *–G*
11) search for SHLIB_CFLAGS and comment out existing line; make a copy of that line directly beneath it and replace *–fPIC* with *–qpic*
12) save Makefile
13) run make

(this should result in a successful build of libtcl8.4.so and tclsh)

6.2 Static Library

1) download stable (tcl8.4.16-src.tar.gz at this time) from http://www.tcl.tk/software/tcltk and copy to frontend node
2) execute *tar –zxvf tcl8.4.16-src.tar.gz*
3) *cd tcl8.4.16/unix*
4) CC=mpixlc; export CC
5) PATH=/bgsys/drivers/ppcfloor/comm/bin:$PATH; export PATH
6) ./configure –disable-shared –disable-threads
7) (configure process executes)
8) edit Makefile:
9) search for SHLIB_CFLAGS and comment out
10) search for LDFLAGS and comment out existing line; make a copy of that line directly beneath it and replace -*Wl,--export-dynamic* with *–qstaticlink*
11) save Makefile
12) run make

(this should result in a successful build of libtcl8.4.so and tclsh with multiple warnings messages – *warning: Using '**functioname**' in statically linked applications requires at runtime the shared libraries from the glibc version used for linking*)

7 References

Vaughan, Gary V., Ben Elliston, Tom Tromey, and Ian Lance Taylor. GNU Autoconf, Automake, and libtool. New Riders Publishing. October 2000.

LIBTIFF library website (http://www.libtiff.org)
ZLIB library website (http://www.zlib.net)

IBM Redbooks. IBM System Blue Gene Solution: Blue Gene/P Application Development (http://www.redbooks.ibm.com)

TCL/TK website (http://www.tcl.tk/software/tcltk/)