Hybrid MPI-OpenMP Programming

Pierre-Francois.Lavallee@idris.fr
Philippe.Wautelet@aero.obs-mip.fr

CNRS — IDRIS / LA

Version 3.0 — 1 December 2016
This document is subject to regular updating. The most recent version is available on the IDRIS Web server, section IDRIS Training:

http://www.idris.fr/eng

IDRIS
Institute for Development and Resources in Intensive Scientific Computing
Rue John Von Neumann
Bâtiment 506
BP 167
91403 ORSAY CEDEX
France
http://www.idris.fr
Tools
- SCALASCA
- TAU
- TotalView

Hands-on Exercises
- TP1 — Hybrid MPI and OpenMP — Global synchronization
- TP2 — Hybrid MPI and OpenMP — Parallel PingPong
- TP3 — HYDRO, from MPI to hybrid MPI and OpenMP version

Appendices
- SBPR on older architectures
- Case Study: Poisson3D
Preamble
Presentation of the Training Course

The purpose of this course is to present MPI+OpenMP hybrid programming as well as feedback from effective implementations of this model of parallelization on several application codes.

- The Introduction chapter endeavors to show, through technological evolutions of architectures and parallelism constraints, how the transition to hybrid parallelization is indispensable if we are to take advantage of the power of the latest generation of massively parallel machines.

- However, a hybrid code cannot perform well if the MPI and OpenMP parallel implementations have not been previously optimized.

- The Hybrid programming section is entirely dedicated to the MPI+OpenMP hybrid approach. The benefits of hybrid programming are numerous:
  - Memory savings
  - Improved performances
  - Better load balancing
  - Coarser granularity, resulting in improved scalability
  - Better code adequacy to the target architecture hardware specificities

However, as you will notice in the hands-on exercises, the implementation on a real application requires a large time investment and a thorough familiarity with MPI and OpenMP.
Introduction
Moore’s Law

Statement
According to Moore’s law, the number of transistors which can be placed on an integrated circuit at a reasonable cost doubles every two years.

Electric consumption
- Dissipated electric power = $frequency^3$ (for a given technology).
- Dissipated power per $cm^2$ is limited by cooling.
- Energy cost.

Moore’s law and electric consumption
- Processor frequency is no longer increasing due to prohibitive electrical consumption (maximum frequency limited to 3GHz since 2002-2004).
- Number of transistors per chip continues to double every two years.

=> Number of cores per chip is increasing: The Intel Broadwell-EP chips have up to 22 cores each and can run 44 threads simultaneously.

=> Some architectures favor low-frequency cores, but in a very large number (IBM Blue Gene).
Moore's Law

Microprocessor Transistor Counts 1971-2011 & Moore's Law

The graph shows the transistor count doubling every two years, starting from the 4004 microprocessor in 1971. The trend line indicates exponential growth in transistor count from 1971 to 2011.

CC BY-SA 3.0, http://en.wikipedia.org/wiki/Moore%27s_law
# The Memory Wall

## Causes
- Throughputs towards the memory are not increasing as quickly as processor computing power.
- Latencies (access times) of the memory are decreasing very slowly.
- Number of cores per memory module is increasing.

## Consequences
- The gap between the memory speed and the theoretical performance of the cores is increasing.
- Processors waste more and more cycles while waiting for data.
- Increasingly difficult to maximally exploit the performance of processors.

## Partial solutions
- Addition of cache memories is essential.
- Access parallelization via several memory banks as found on the vector architectures (Intel Broadwell: 4 channels).
- If the clock frequency of the cores stagnates or falls, the gap could be reduced.
Evolution of the performance in the Top500
Evolution of the number of cores in the Top500

Number of cores

Date

Evolution of the performance per core in the Top500
Technical evolution

- The computing power of supercomputers is doubling every year (faster than Moore’s Law, but electrical consumption is also increasing).
- The number of cores is increasing rapidly (massively parallel (IBM Blue Gene Q) and many-cores architectures (Intel Xeon Phi)).
- Emergence of heterogeneous accelerated architectures (standard processors coupled with GPU).
- Machine architecture is becoming more complex and the number of layers increasing (processors/cores, memory access, network and I/O).
- Memory per core has been stagnating and is beginning to decrease.
- Performance per core is stagnating and is much lower on some machines than on a simple laptop (IBM Blue Gene).
- Throughput towards the disk and memory is increasing more slowly than the computing power.
Amdahl’s Law

Statement

Amdahl’s Law predicts the theoretical maximum speedup obtained by parallelizing a code ideally, for a given problem with a fixed size:

\[
Sp(P) = \frac{T_s}{T_{//}(P)} = \frac{1}{\alpha + \frac{(1-\alpha)}{P}} < \frac{1}{\alpha} \quad (P \to \infty)
\]

with \(Sp\) the speedup, \(T_s\) the execution time of the sequential code (monoprocessor), \(T_{//}(P)\) the execution time of the ideally parallelized code on \(P\) cores and \(\alpha\) the non-parallelizable part of the application.

Interpretation

Regardless of the number of cores, the speedup is always less than the inverse of the percentage represented by the purely sequential fraction.

Example: If the purely sequential fraction of a code represents 20% of the execution time of the sequential code, then regardless of the number of cores, we will have:

\[
Sp < \frac{1}{20\%} = 5
\]
# Theoretical Maximum Speedup

<table>
<thead>
<tr>
<th>Cores</th>
<th>$\alpha$ (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>100000</td>
<td>100000</td>
</tr>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

![Graph 1](image1.png)

![Graph 2](image2.png)
The Gustafson-Barsis Law predicts the theoretical maximum speedup obtained by parallelizing a code ideally for a problem of constant size per core, in supposing that the execution time of the sequential fraction does not increase with the overall problem size:

$$Sp(P) = \alpha + P(1 - \alpha)$$

with $Sp$ the speedup, $P$ the number of cores and $\alpha$ the non-parallelizable part of the application.

This law is more optimistic than Amdahl’s because it shows that the theoretical speedup increases with the size of the problem being studied.
### Consequences for the Users

#### Consequences for the applications
- It is necessary to exploit a large number of relatively slow cores.
- Tendency for individual core memory to decrease: Necessity to not waste memory.
- Higher level of parallelism continually needed for the efficient usage of modern architectures (regarding both computing power and memory size).
- The I/O also becoming an increasingly current problem.

#### Consequences for the developers
- The time has ended when you only needed to wait a while to obtain better performance (i.e. stagnation of computing power per core).
- Increased necessity to understand the hardware architecture.
- More and more difficult to develop codes on your own (need for experts in HPC as well as multi-disciplinary teams).
Evolution of programming methods

- MPI is still predominant and it will remain so for some time (a very large community of users and the majority of current applications).
- The MPI-OpenMP hybrid approach is being used more and seems to be the preferred approach for supercomputers.
- GPU programming usage is increasing, but it requires a third level of parallelism (MPI+OpenMP+GPU).
- New parallel programming languages are appearing (UPC, Coarray-Fortran, PGAS languages, X10, Chapel, ...), but they are in experimental phases (at variable levels of maturity). Some are very promising; it remains to be seen whether they will be used in real applications.
Turing: IBM Blue Gene/Q

Ada: IBM x3750
Important numbers

**Turing: 6 racks Blue Gene/Q:**
- 6,144 nodes
- 98,304 cores
- 393,216 threads
- 96 TiB
- 1.258 Tflop/s
- 636 kW (106 kW/ rack)

**Ada: 15 racks IBM x3750M4:**
- 332 compute nodes and 4 pre-/post-processing nodes
- 10,624 Intel SandyBridge cores at 2.7 GHz
- 46 TiB
- 230 Tflop/s
- 366 kW

- 5 PiB on shared disks between BG/Q and Intel (100 GiB/s peak bandwidth)
- 1 MW for the whole configuration (not counting the cooling system)
Hybrid programming
Hybrid programming
- Definitions
- Reasons for Hybrid Programming
- Applications Which Can Benefit From Hybrid Programming
- MPI and Multithreading
- MPI and OpenMP
- Adequacy to the Architecture: Memory Savings
- Adequacy to the Architecture: the Network Aspect
- Effects of a non-uniform architecture
- Case Study: Multi-Zone NAS Parallel Benchmark
- Case Study: HYDRO
Hybrid parallel programming consists of mixing several parallel programming paradigms in order to benefit from the advantages of the different approaches.

In general, MPI is used for communication between processes, and another paradigm (OpenMP, pthreads, PGAS languages, UPC, ...) is used inside each process.

In this training course, we will talk exclusively about the use of MPI with OpenMP.
Hybrid Programming

Schematic drawing

Process 0

Shared memory

Thread 0
Private memory (stack or heap)
Execution stack

Thread 1
Private memory (stack or heap)
Execution stack

Thread n
Private memory (stack or heap)
Execution stack

Messages

Process 1

Shared memory

Thread 0
Private memory (stack or heap)
Execution stack

Thread 1
Private memory (stack or heap)
Execution stack

Thread n
Private memory (stack or heap)
Execution stack

Process n-1

Shared memory

Thread 0
Private memory (stack or heap)
Execution stack

Thread 1
Private memory (stack or heap)
Execution stack

Thread n
Private memory (stack or heap)
Execution stack

Process n

Shared memory

Thread 0
Private memory (stack or heap)
Execution stack

Thread 1
Private memory (stack or heap)
Execution stack

Thread n
Private memory (stack or heap)
Execution stack
Advantages of hybrid programming (1)

- Improved scalability through a reduction in both the number of MPI messages and the number of processes involved in collective communications (MPI_Alltoall is not very scalable), and by improved load balancing.
- More adequate to the architecture of modern supercomputers (interconnected shared-memory nodes, NUMA machines, ...), whereas MPI used alone is a flat approach.
- Optimization of the total memory consumption, thanks to the OpenMP shared-memory approach; less replicated data in the MPI processes; and less memory used by the MPI library itself.
- Reduction of the footprint memory when the size of certain data structures depends directly on the number of MPI processes.
- Can go beyond certain algorithmic limitations (for example, the maximum decomposition in one direction).
- Enhanced performance of certain algorithms by reducing the number of MPI processes (fewer domains = a better preconditioner, provided that the contributions of other domains are dropped).
Advantages of hybrid programming (2)

- Fewer simultaneous accesses in I/O and a larger average record size; fewer and more suitably-sized requests cause less load on the meta-data servers, and potentially significant time savings on a massively parallel application.

- There are fewer files to manage if each process writes its own file(s) (an approach strongly advised against, however, in a framework of massive parallelism).

- Certain architectures require executing several threads (or processes) per core in order to efficiently use the computational units.

- An MPI parallel code is a succession of computation and communication phases. The granularity of a code is defined as the average ratio between two successive computation and communication phases. The greater the granularity of a code, the more scalable it is. Compared to the pure MPI approach, the hybrid approach significantly increases the granularity and consequently, the scalability of codes.

Disadvantages of hybrid programming

- Complexity and higher level of expertise.

- Necessity of having good MPI and OpenMP performances (Amdahl’s law applies separately to the two approaches).

- Total gains in performance are not guaranteed (extra additional costs, ...).
### Applications which can benefit from hybrid programming

- Codes having limited MPI scalability (due to using calls to `MPI_Alltoall`, for example)
- Codes requiring dynamic load balancing
- Codes limited by memory size and having a large amount of replicated data in the MPI process or having data structures which depend on the number of processes for their dimension
- Inefficient local MPI implementation library for intra-node communications
- Many massively parallel applications
- Codes working on problems of fine-grain parallelism or on a mixture of fine-grain and coarse-grain parallelism
- Codes limited by the scalability of their algorithms
- ...
Thread support in MPI

The MPI standard provides a particular subroutine to replace MPI_Init when the MPI application is multithreaded: This subroutine is MPI_Init_thread.

- The standard does not require a minimum level of thread support. Certain architectures and/or implementations, therefore, could end up not having any support for multithreaded applications.
- The ranks identify only the processes; the threads cannot be specified in the communications.
- Any thread can make MPI calls (depending on the level of support).
- Any thread of a given MPI process can receive a message sent to this process (depending on the level of support).
- Blocking calls will only block the thread concerned.
- The call to MPI_Finalize must be made by the same thread that called MPI_Init_thread and only when all the threads of the process have finished their MPI calls.
The level of support requested is provided in the variable "required". The level actually obtained (which could be less than what was requested) is returned in "provided".

- **MPI_THREAD_SINGLE**: Only one thread per process can run.
- **MPI_THREAD_FUNNELED**: The application can launch several threads per process, but only the main thread (the one which made the call to MPI_Init_thread) can make MPI calls.
- **MPI_THREAD_SERIALIZED**: All the threads can make MPI calls, but only one at a time.
- **MPI_THREAD_MULTIPLE**: Entirely multithreaded without restrictions.
Other MPI subroutines

MPI_Query_thread returns the support level of the calling process:

```c
int MPI_Query_thread(int *provided)
MPI_QUERY_THREAD_PROVIDED(IERROR)
```

MPI_Is_thread_main gives the return, whether it is the main thread calling or not. (Important if the support level is MPI_THREAD_FUNNELED and also for the call MPI_Finalize.)

```c
int MPI_Is_thread_main(int *flag)
MPI_IS_THREAD_MAIN(FLAG, IERROR)
```
Restrictions on MPI collective calls (1)

In `MPI_THREAD_MULTIPLE` mode, the user must ensure that collective operations using the same communicator, memory window, or file handle are correctly ordered among the different threads.

- It is forbidden, therefore, to have several threads per process making calls with the same communicator without first ensuring that these calls are made in the same order on each of the processes.
- We cannot have at any given time, therefore, more than one thread making a collective call with the same communicator (whether the calls are different or not).
- For example, if several threads make a call to `MPI_Barrier` with `MPI_COMM_WORLD`, the application may hang (this was easily verified on Babel and Vargas).
- 2 threads, each one calling an `MPI_Allreduce` (with the same reduction operation or not), could obtain false results.
- 2 different collective calls cannot be used either (for example, an `MPI_Reduce` and an `MPI_Bcast`).
There are several possible ways to avoid these difficulties:

- Impose the order of the calls by synchronizing the different threads interior to each MPI process.
- Use different communicators for each collective call.
- Only make collective calls on one single thread per process.

Comment: In MPI_THREAD_SERIALIZED mode, the restrictions should not exist because the user must ensure that at any given moment, a maximum of only one thread per process is involved in an MPI call (collective or not). Caution: The same order of calls in all the processes must nevertheless be respected.
Implications of the different support levels

The multithreading support level provided by the MPI library imposes certain conditions and restrictions on the use of OpenMP:

- **MPI_THREAD_SINGLE**: OpenMP cannot be used.
- **MPI_THREAD_FUNNELED**: MPI calls must be made either outside of OpenMP *parallel* regions, in OpenMP master regions, or in protected zones (call to `MPI_Is_thread_main`).
- **MPI_THREAD_SERIALIZED**: In the OpenMP parallel regions, MPI calls must be made in critical sections (when necessary, to ensure that only one MPI call is made at a time)
- **MPI_THREAD_MULTIPLE**: No restriction.
### State of current implementations

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Level Supported</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPICH</td>
<td>MPI_THREAD_MULTIPLE</td>
<td></td>
</tr>
<tr>
<td>OpenMPI</td>
<td>MPI_THREAD_MULTIPLE</td>
<td>Must be compiled with –enable-mpi-threads</td>
</tr>
<tr>
<td>IBM BlueGene/Q</td>
<td>MPI_THREAD_MULTIPLE</td>
<td></td>
</tr>
<tr>
<td>IBM PEMPI</td>
<td>MPI_THREAD_MULTIPLE</td>
<td></td>
</tr>
<tr>
<td>BullxMPI</td>
<td>MPI_THREAD_FUNNELED</td>
<td></td>
</tr>
<tr>
<td>Intel - MPI</td>
<td>MPI_THREAD_MULTIPLE</td>
<td>Use -mt_mpi</td>
</tr>
<tr>
<td>SGI - MPT</td>
<td>MPI_THREAD_MULTIPLE</td>
<td>Use -mpi_mt</td>
</tr>
</tbody>
</table>
Why memory savings?

- Hybrid programming allows optimizing code adequacy to the target architecture (generally composed of shared-memory nodes [SMP] linked by an interconnection network). The advantage of shared memory inside a node is that it is not necessary to duplicate data in order to exchange them. Every thread can access (read /write) SHARED data.

- The ghost or halo cells, introduced to simplify MPI code programming using a domain decomposition, are no longer needed within the SMP node. Only the ghost cells associated with the inter-node communications are necessary.

- The memory savings associated with the elimination of intra-node ghost cells can be considerable. The amount saved largely depends on the order of the method used, the type of domain (2D or 3D), the domain decomposition (in one or multiple dimensions), and on the number of cores in the SMP node.

- The footprint memory of the system buffers associated with MPI is not negligible and increases with the number of processes. For example, for an Infiniband network with 65,000 MPI processes, the footprint memory of system buffers reaches 300 MB per process, almost 20 TB in total!
Example: 2D domain, decomposition in both directions

8 cores SMP node, flat MPI domain decomposition

- Intra-node ghost cells
- Extra-node ghost cells
- Sub-domain associated with one MPI process

8 cores SMP node, hybrid domain decomposition
What are the relative memory savings obtained by using a hybrid version (instead of a flat MPI version) of a 3D code parallelized by a technique of domain decomposition in its three dimensions? Let us try to calculate this in function of numerical method ($h$) and the number SMP node cores ($c$).

We will assume the following hypotheses:

- The order of the numerical method $h$ varies from 1 to 10.
- The number of cores $c$ of the SMP node varies from 1 to 128.
- To size the problem, we will assume that we have access to 64 GB of shared-memory on the node.

The simulation result is presented in the following slide. The isovalues 10%, 20% and 50% are represented by the white lines on the isosurface.
Extrapolation on a 3D domain

% of gain comparing hybrid and flat MPI approach

Order of the method

Number of cores
Memory savings on some real application codes (1)

- **Source:** « Mixed Mode Programming on HECToR », A. Stathopoulos, August 22, 2010, MSc in High Performance Computing, EPCC
- **Target machine:** HECToR CRAY XT6.
  1856 Compute Nodes (CN), each one composed of two processors AMD 2.1GHz, 12 cores sharing 32 GB of memory, for a total of 44544 cores, 58 GB of memory and a peak performance of 373 Tflop/s.
- **Results** (the memory per node is expressed in MB):

<table>
<thead>
<tr>
<th>Code</th>
<th>Pure MPI version</th>
<th>Hybrid version</th>
<th>Memory savings</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pure MPI version</td>
<td>Hybrid version</td>
<td>Memory savings</td>
</tr>
<tr>
<td></td>
<td>MPI prc Mem./ Node</td>
<td>MPI x threads Mem./Node</td>
<td></td>
</tr>
<tr>
<td>CPMD</td>
<td>1152 2400</td>
<td>48 x 24 500</td>
<td>4.8</td>
</tr>
<tr>
<td>BQCD</td>
<td>3072 3500</td>
<td>128 x 24 1500</td>
<td>2.3</td>
</tr>
<tr>
<td>SP-MZ</td>
<td>4608 2800</td>
<td>192 x 24 1200</td>
<td>2.3</td>
</tr>
<tr>
<td>IRS</td>
<td>2592 2600</td>
<td>108 x 24 900</td>
<td>2.9</td>
</tr>
<tr>
<td>Jacobi</td>
<td>2304 3850</td>
<td>96 x 24 2100</td>
<td>1.8</td>
</tr>
</tbody>
</table>
Memory savings on some real application codes (2)

- Executions on 4096 cores
- Supercomputers used: Fujitsu BX900 with Nehalem-EP processors at 2.93 GHz (8 cores and 24 GiB per node)
- All sizes given in TiB

<table>
<thead>
<tr>
<th>System</th>
<th>Pure MPI</th>
<th>4 threads/process</th>
<th>8 threads/process</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total (code+sys.)</td>
<td>Total (code+sys.)</td>
<td>Gain</td>
</tr>
<tr>
<td>BX900</td>
<td>5.40 (3.40+2.00)</td>
<td>2.83 (2.39+0.44)</td>
<td>1.9</td>
</tr>
</tbody>
</table>
Conclusion

- The memory savings aspect is too often forgotten when we talk about hybrid programming.
- The potential savings, however, are very significant and could be exploited to increase the size of the problems to be simulated!
- There are several reasons why the differential between the MPI and hybrid approaches will enlarge at an increasingly rapid rate for the next generation of machines:
  1. Multiplication in the total number of cores.
  2. Rapid multiplication in the number of available cores within a node as well as the generalization of hyperthreading or SMT (the possibility of running multiple threads simultaneously on one core).
  3. General use of high-order numerical methods (computing costs decreasing, thanks particularly to hardware accelerators).
- The benefits will make the transition to hybrid programming almost mandatory...
How to optimise the use of the inter-node interconnection network

- The hybrid approach aims to use the available hardware resources the most effectively (shared memory, memory hierarchy, communication network).

- One of the difficulties of hybrid programming is to generate a sufficient number of communication flows in order to make the best use of the inter-node communication network.

- In fact, the throughputs of inter-node interconnection networks of recent architectures are high (bidirectional throughput peak of 10 GB/s on Ada, for example) and one data flow alone cannot saturate it; only a fraction of the network is really used, the rest being wasted.

- IDRIS has developed a small benchmark SBPR (Saturation Bande Passante Réseau [Network Bandwidth Saturation]), a simple parallel ping-pong test aimed at determining the number of concurrent flows required to saturate the network.
MPI_THREAD_FUNNELED version of SBPR

**MPI_THREAD_FUNNELED approach:**

- We increase the network bandwidth actually used by increasing the number of MPI processes per node (i.e. we generate as many parallel communication flows as there are MPI processes per node).

- The basic solution, which consists of using as many OpenMP threads as there are cores inside a node and as many MPI processes as the number of nodes, is not generally the most efficient: The resources are not being used optimally, in particular the network.

- We look for the optimal ratio value between the number of MPI processes per node and the number of OpenMP threads per MPI process. The greater the ratio, the better the inter-node network flow rate, but the granularity is not as good. A compromise has to be found.

- The number of MPI processes (i.e. the data flow to be managed simultaneously) necessary to saturate the network varies greatly from one architecture to another.

- This value could be a good indicator of the optimal ratio of the number of MPI processes/number of OpenMP threads per node of a hybrid application.
MPI_THREAD_FUNNELED version of SBPR: Example on a 4-Core (BG/P) SMP Node


MP1_THREAD_FUNNELED version of SBPR: Example on a 4-Core (BG/P) SMP Node

2 concurrent MPI communication data flows

4 concurrent MPI communication data flows
**MPI_THREAD_MULTIPLE version of SBPR**

**MPI_THREAD_MULTIPLE approach:**
- We increase the network bandwidth actually used by increasing the number of OpenMP threads which participate in the communications.
- We have a single MPI process per node. We look for the minimum number of communication threads required to saturate the network.

---

**MPI_THREAD_MULTIPLE version of SBPR: Example on a 4-Core (BG/P) SMP Node**

[Diagram showing two nodes connected by a communication link, with MPI processes and OpenMP threads depicted.]
MPI_THREAD_MULTIPLE version of SBPR: Example on a 4-Core (BG/P) SMP Node
Optimal Use of the Interconnection Network

**SBPR: Results on Ada**

2 links in //, FDR10 Infiniband, peak throughput 10 GB/s.

![Graph showing Ada extranode bandwidth for 1MB messages](image)

- **Funneled**
- **Multiple**

**Number of streams** vs **Bandwidth (MB/s)**

At 10 GB/s, the bandwidth saturates with a growing number of streams.
SBPR: Results on Ada

2 links in //, FDR10 Infiniband, peak throughput 10 GB/s.

![Ada extranode bandwidth (100MB messages)](image)

- **Funneled**
- **Multiple**

Bandwidth (MB/s)

Number of streams
SBPR: Results on Ada

- With a single data flow, we use only a fraction of the inter-node network bandwidth.
- In `MPI_THREAD_FUNNELED` mode, saturation of Ada inter-node network links begins with only 2 parallel flows (i.e. 2 MPI processes per node).
- In `MPI_THREAD_MULTIPLE` mode, saturation of Ada inter-node network links appears with 16 parallel flows (i.e. 16 threads per node participating in communications).
- The 2 `MPI_THREAD_FUNNELED` and `MPI_THREAD_MULTIPLE` approaches are well suited to Ada with an advantage for the first method.
SBPR: Results on Turing

2 links in // (E direction of 5D torus), peak throughput 4 GB/s.
SBPR: Results on Turing

2 links in // (E direction of 5D torus), peak throughput 4 GB/s.

![Graph showing Turing extranode bandwidth - Multiple]
SBPR: Results on Turing

- The use of only one data flow (i.e. one single communication thread or MPI process per node) is sufficient to totally saturate the interconnection network between two neighboring nodes.
- The performances of the `MPI_THREAD_MULTIPLE` and `MPI_THREAD_FUNNELED` versions are comparable on Turing.
- The throughput reached is about 3.5 GB/s, which is around 85% of the peak inter-node network bandwidth (for the E direction of the 5D torus).
Non-uniform architecture

Most modern supercomputers have a non-uniform architecture:

- NUMA, Non Uniform Memory Access with the memory modules attached to different sockets inside a given node.
- Memory caches shared or not between different cores or groups of cores.
- Network cards connected to some sockets.
- Non-uniform network (for example with several layers of network switches) => see also process mapping.

Effects

- Performance of MPI communications are not the same for each core even inside a node.
- Process mapping is important inside and outside nodes.
- Performance problems and optimisation are hard due to the complexity of the modern architectures.
Non-uniform architecture on Ada

Switch IB

Noeud 0

Noeud 1

IB0

IB1

M0

M1

M2

M3

S0

S1

S2

S3

C0 C1 C2 C3

C4 C5 C6 C7

C12 C13 C14 C15

C16 C17 C18 C19

C20 C21 C22 C23

C8 C9 C10 C11

C24 C25 C26 C27

C28 C29 C30 C31

C0 C1 C2 C3

C4 C5 C6 C7

C12 C13 C14 C15

C16 C17 C18 C19

C20 C21 C22 C23

C24 C25 C26 C27

C28 C29 C30 C31
Ping-pong on Ada (POE)

- Intra-node intra-socket
- Intra-node extra-socket 1
- Intra-node extra-socket 3
- Extra-node socket 0
- Extra-node socket 1
- Extra-node socket 3

Bandwidth (MiB/s) vs. Message size (bytes)

1 December 2016
Developed by NASA, the Multi-Zone NAS Parallel Benchmark is a group of performance test programs for parallel machines.

- These codes use algorithms close to those used in certain CFD codes.
- The multi-zone version provides three different applications with eight different problem sizes.
- This benchmark is used frequently.
- The sources are available at the address:
Presentation of the Benchmark

**Selected Application: BT-MZ**

BT-MZ: block tridiagonal solver.
- The zone sizes vary widely: poor load balancing.
- The hybrid approach should improve the situation.

**Selected Application: SP-MZ**

SP-MZ: scalar pentadiagonal solver.
- All the zone sizes are identical: perfect load balancing.
- The hybrid approach should not bring any improvement.

**Selected Problem Sizes**

- Class D: 1024 zones (and therefore limited to 1024 MPI processes), 1632 x 1216 x 34 grid points (13 GiB)
- Class E: 4096 zones (and therefore limited to 4096 MPI processes), 4224 x 3456 x 92 grid points (250 GiB)
The hybrid version is equivalent to the MPI for a not very large number of processes.

When load imbalance appears in pure MPI (starting from 512 processes for class D and from 2048 for class E), the hybrid version permits maintaining a very good scalability by reducing the number of processes.

The limitation of 1024 zones in class D and of 4096 in class E limits the number of MPI processes to 1024 and 4096 respectively; however, the addition of OpenMP permits using many more cores while at the same time obtaining excellent scalability.
Analysis of Results: SP-MZ

- This benchmark benefits in certain cases from the hybrid character of the application even when there is not load imbalance.
- The limitation of 1024 zones in class D and of 4096 in class E, limits the number of MPI processes to 1024 and 4096 respectively; but the addition of OpenMP permits using many more cores while, at that same time, obtaining an excellent scalability.
Presentation of the HYDRO Code (1)

- This is the code used for the hands-on exercises of the hybrid course.
- Hydrodynamics code, 2D-Cartesian grid, finite volume method, resolution of a Riemann problem on the interfaces with a Godunov method.
- For the last few years, in the framework of the IDRIS technology watch, this code has served as a benchmark for new architectures, from the simple graphics card to the petaflops machine.
- New versions have been regularly developed over the years with new implementations (new languages, new paradigms of parallelization).
- 1500 lines of code in its F90 monoprocessor version.
Today, there are the following hydro versions:

- Original version, F90 monoprocessor (P.-Fr. Lavallée, R. Teyssier)
- Monoprocessor C version (G. Colin de Verdière)
- MPI F90 parallel version (1D P.-Fr. Lavallée, 2D Ph. Wautelet)
- MPI C parallel version (2D Ph. Wautelet)
- OpenMP Fine-Grain and Coarse-Grain F90 parallel version (P.-Fr. Lavallée)
- OpenMP Fine-Grain C parallel version (P.-Fr. Lavallée)
- MPI2D-OpenMP Fine-Grain and Coarse-Grain F90 hybrid parallel version (P.-Fr. Lavallée, Ph. Wautelet)
- MPI2D-OpenMP Fine-Grain hybrid parallel version C (P.-Fr. Lavallée, Ph. Wautelet)
- C GPGPU CUDA, HMPP, OpenCL version (G. Colin de Verdière)
- Pthreads parallel version C (D. Lecas)

Many other versions are under development: UPC, CAF, OpenACC, OpenMP4.5, ...
## Results for the nx = 10000, ny = 10000 domain

<table>
<thead>
<tr>
<th>Time in (s)</th>
<th>32 cores</th>
<th>64 cores</th>
<th>128 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>VN mode</td>
<td>53.14</td>
<td>24.94</td>
<td>12.40</td>
</tr>
<tr>
<td>DUAL mode</td>
<td>50.28</td>
<td>24.70</td>
<td>12.22</td>
</tr>
<tr>
<td>SMP mode</td>
<td>52.94</td>
<td>25.12</td>
<td>12.56</td>
</tr>
</tbody>
</table>

![Graph showing speedup comparison]
### Characteristics of the domains used for weak scaling

- **On 4096 cores, total number of points of the domain: 16 10^8**
  - 400000x4000: domain elongated in the first dimension
  - 40000x4000: square domain
  - 4000x400000: domain elongated in the second dimension

- **On 8192 cores, total number of domain points: 32 10^8**
  - 800000x4000: domain elongated in the first dimension
  - 56568x56568: square domain
  - 4000x800000: domain elongated in the second dimension

- **On 16384 cores, total number of points of the domain: 64 10^8**
  - 1600000x4000: domain elongated in the first dimension
  - 80000x80000: square domain
  - 4000x1600000: domain elongated in the second dimension

- **On 32768 cores, total number of points of the domain: 128 10^8**
  - 3200000x4000: domain elongated in the first dimension
  - 113137x113137: square domain
  - 4000x3200000: domain elongated in the second dimension

- **On 40960 cores, total number of points of the domain: 16 10^9**
  - 4000000x4000: domain elongated in the first dimension
  - 126491x126491: square domain
  - 4000x4000000: domain elongated in the second dimension
Results for the domain elongated in the first dimension

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>4096 cores</th>
<th>8192 cores</th>
<th>16384 cores</th>
<th>32768 cores</th>
<th>40960 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>VN mode</td>
<td>6.62</td>
<td>7.15</td>
<td>8.47</td>
<td>13.89</td>
<td>19.64</td>
</tr>
<tr>
<td>DUAL mode</td>
<td>6.21</td>
<td>6.46</td>
<td>6.75</td>
<td>7.85</td>
<td>8.75</td>
</tr>
<tr>
<td>SMP mode</td>
<td>6.33</td>
<td>6.38</td>
<td>6.72</td>
<td>7.00</td>
<td>7.22</td>
</tr>
</tbody>
</table>

Performances compared to the MPI version

Elapsed execution time
Results using 10 racks on Babel - Weak Scaling

Results for the square domain

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>4096 cores</th>
<th>8192 cores</th>
<th>16384 cores</th>
<th>32768 cores</th>
<th>40960 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>VN mode</td>
<td>6.17</td>
<td>6.67</td>
<td>8.00</td>
<td>13.32</td>
<td>19.57</td>
</tr>
<tr>
<td>DUAL mode</td>
<td>6.17</td>
<td>6.14</td>
<td>6.52</td>
<td>7.64</td>
<td>8.56</td>
</tr>
<tr>
<td>SMP mode</td>
<td>6.24</td>
<td>6.19</td>
<td>6.33</td>
<td>6.57</td>
<td>7.19</td>
</tr>
</tbody>
</table>

Performances compared to the MPI version

Elapsed execution time
Results using 10 racks on Babel - Weak Scaling

Results for the domain elongated in the second dimension

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>4096 cores</th>
<th>8192 cores</th>
<th>16384 cores</th>
<th>32768 cores</th>
<th>40960 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>VN mode</td>
<td>8.04</td>
<td>8.28</td>
<td>9.79</td>
<td>15.42</td>
<td>21.17</td>
</tr>
<tr>
<td>DUAL mode</td>
<td>8.22</td>
<td>8.30</td>
<td>8.20</td>
<td>9.44</td>
<td>12.08</td>
</tr>
<tr>
<td>SMP mode</td>
<td>7.33</td>
<td>8.58</td>
<td>8.61</td>
<td>8.43</td>
<td>8.64</td>
</tr>
</tbody>
</table>

Performances compared to the MPI version

Elapsed execution time
Interpretation of results

- The results of weak scaling, obtained by using up to 40960 computation cores, are very interesting. Certain phenomena become visible with this high number of cores.

- The scalability of the flat MPI version shows its limits very rapidly. It has difficulty scaling up to 16384 cores and then the elapsed time begins to explode.

- As we expected, the DUAL hybrid version, but even more the SMP version, behave very well up to 32768 cores with nearly constant elapsed times. On 40960 cores, the SMP version shows a very slight additional cost; on the DUAL version the additional cost becomes significant.

- In weak scaling, the scalability limit of the flat MPI version is 16384 cores, that of the DUAL version is 32768 cores, and that of the SMP version has not yet been reached on 40960 cores!

- On 40960 cores, the SMP hybrid version is between 2.5 and 3 times faster than the pure MPI version.

- It is clear that scaling (here over 16K cores) with this type of parallelization method (i.e. domain decomposition), requires hybrid parallelization. It is not enough to use MPI alone!
Results using 10 racks on Babel - Strong Scaling

Results for the nx = 400000, ny = 4000 domain

<table>
<thead>
<tr>
<th>Time(s)</th>
<th>4096 cores</th>
<th>8192 cores</th>
<th>16384 cores</th>
<th>32768 cores</th>
<th>40960 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>VN Mode</td>
<td>6.62</td>
<td>4.29</td>
<td>4.44</td>
<td>8.30</td>
<td>13.87</td>
</tr>
<tr>
<td>DUAL Mode</td>
<td>6.21</td>
<td>3.34</td>
<td>2.03</td>
<td>2.40</td>
<td>3.13</td>
</tr>
<tr>
<td>SMP Mode</td>
<td>6.33</td>
<td>3.18</td>
<td>1.75</td>
<td>1.24</td>
<td>1.29</td>
</tr>
</tbody>
</table>

Performances compared to the MPI version

Scalability up to 40960 cores
### Results for the nx = 40000, ny = 40000 Domain

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>4096 cores</th>
<th>8192 cores</th>
<th>16384 cores</th>
<th>32768 cores</th>
<th>40960 c.</th>
</tr>
</thead>
<tbody>
<tr>
<td>VN Mode</td>
<td>6.17</td>
<td>3.54</td>
<td>3.10</td>
<td>8.07</td>
<td>13.67</td>
</tr>
<tr>
<td>DUAL Mode</td>
<td>6.17</td>
<td>3.10</td>
<td>1.88</td>
<td>2.35</td>
<td>3.12</td>
</tr>
<tr>
<td>SMP Mode</td>
<td>6.24</td>
<td>3.10</td>
<td>1.63</td>
<td>1.20</td>
<td>1.26</td>
</tr>
</tbody>
</table>

**Performances compared to the MPI version**

**Scalability up to 40960 cores**
Results using 10 racks on Babel - Strong Scaling

Results for the nx = 4000, ny = 400000 Domain

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>4096 cores</th>
<th>8192 cores</th>
<th>16384 cores</th>
<th>32768 cores</th>
<th>40960 c.</th>
</tr>
</thead>
<tbody>
<tr>
<td>VN Mode</td>
<td>8.04</td>
<td>4.31</td>
<td>4.52</td>
<td>8.26</td>
<td>13.85</td>
</tr>
<tr>
<td>DUAL Mode</td>
<td>8.22</td>
<td>3.96</td>
<td>2.22</td>
<td>2.46</td>
<td>3.34</td>
</tr>
<tr>
<td>SMP Mode</td>
<td>7.33</td>
<td>3.94</td>
<td>1.91</td>
<td>1.29</td>
<td>1.32</td>
</tr>
</tbody>
</table>

Performances compared to the MPI version

Scalability up to 40960 cores
Interpretation of results

- The results of strong scaling, obtained by using up to 40960 computation cores, are very interesting. Here again, new phenomena emerge with this high number of cores.

- The scalability of the flat MPI version shows its limits very quickly. It has difficulty to scale up to 8192 cores and then it begins to collapse.

- As we expected, the DUAL hybrid version, but even more the SMP version, behave very well up to 16384 cores, with a perfectly linear acceleration. The SMP version continues to scale (non-linearly) up to 32768 cores; beyond this, the performances are no longer improved.

- In strong scaling, the scalability limit of the flat MPI version is 8192 cores, whereas that of the SMP hybrid version is 32768 cores. We find here a factor of 4 which corresponds to the number of cores in the BG/P node!

- The best hybrid version (32768 cores) is between 2.6 and 3.5 times faster than the best pure MPI version (8192 cores).

- It is clear that with this type of parallelization method (i.e. domain decomposition), scaling (here over 10K cores) requires recourse to hybrid parallelization. It is not enough to use MPI alone!
Results on Two Vargas Nodes

Results for the $nx = 100000$, $ny = 1000$ domain

<table>
<thead>
<tr>
<th>MPI x OMP per node</th>
<th>Time (s)</th>
<th>Mono</th>
<th>64 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 x 1</td>
<td>361.4</td>
<td>7.00</td>
<td></td>
</tr>
<tr>
<td>16 x 2</td>
<td>361.4</td>
<td>6.11</td>
<td></td>
</tr>
<tr>
<td>8 x 4</td>
<td>361.4</td>
<td>5.75</td>
<td></td>
</tr>
<tr>
<td>4 x 8</td>
<td>361.4</td>
<td>5.61</td>
<td></td>
</tr>
<tr>
<td>2 x 16</td>
<td>361.4</td>
<td>5.86</td>
<td></td>
</tr>
<tr>
<td>1 x 32</td>
<td>361.4</td>
<td>6.24</td>
<td></td>
</tr>
</tbody>
</table>

The hybrid version is always more efficient than the pure MPI version.

The maximum gain is superior to 20% for the 8MPIx4OMP, 4MPIx8OMP and 2MPIx16OMP distributions.
Results on Two Vargas Nodes

Results for the nx = 10000, ny = 10000 Domain

<table>
<thead>
<tr>
<th>MPI x OMP per node</th>
<th>Time(s)</th>
<th>64 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mono</td>
<td></td>
</tr>
<tr>
<td>32 x 1</td>
<td>449.9</td>
<td>6.68</td>
</tr>
<tr>
<td>16 x 2</td>
<td>449.9</td>
<td>6.03</td>
</tr>
<tr>
<td>8 x 4</td>
<td>449.9</td>
<td>5.64</td>
</tr>
<tr>
<td>4 x 8</td>
<td>449.9</td>
<td>5.82</td>
</tr>
<tr>
<td>2 x 16</td>
<td>449.9</td>
<td>5.87</td>
</tr>
<tr>
<td>1 x 32</td>
<td>449.9</td>
<td>6.31</td>
</tr>
</tbody>
</table>

The hybrid version is always more efficient than the pure MPI version.

The maximum gain is on the order of 20% for the 8MPIx4OMP distribution.
Results for the \(nx = 1000\), \(ny = 100000\) domain

<table>
<thead>
<tr>
<th>MPI x OMP per node</th>
<th>Time (s) Mono</th>
<th>64 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 x 1</td>
<td>1347.2</td>
<td>8.47</td>
</tr>
<tr>
<td>16 x 2</td>
<td>1347.2</td>
<td>7.75</td>
</tr>
<tr>
<td>8 x 4</td>
<td>1347.2</td>
<td>6.92</td>
</tr>
<tr>
<td>4 x 8</td>
<td>1347.2</td>
<td>7.13</td>
</tr>
<tr>
<td>2 x 16</td>
<td>1347.2</td>
<td>7.84</td>
</tr>
<tr>
<td>1 x 32</td>
<td>1347.2</td>
<td>8.53</td>
</tr>
</tbody>
</table>

- The hybrid version is always more efficient than the pure MPI version.
- The maximum gain is on the order of 20% for the 8MPIx4OMP distribution.
Results on Two Vargas Nodes

Interpretation of Results

- Whatever the domain type, the flat MPI version and the hybrid version with only one MPI process per node systematically give the least efficient results.

- The best results are obtained on the hybrid version with (a) a distribution of eight MPI processes per node and four OpenMP threads per MPI process for the two last test cases, and (b) a distribution of four MPI processes per node and sixteen OpenMP threads per MPI process for the first test case.

- We find here a ratio (i.e. number of MPI processes/number of OpenMP threads) close to the one obtained during the interconnection network saturation tests (saturation beginning with eight MPI processes per node).

- Even with a modest size in terms of the number of cores used, it is interesting to note that the hybrid approach prevails each time, sometimes even with significant gains in performance.

- Very encouraging and shows that there is a real interest in increasing the number of cores used.
Conclusions

- A sustainable approach, based on recognized standards (MPI and OpenMP): It is a long-term investment.
- The advantages of the hybrid approach compared to the pure MPI approach are many:
  - Significant memory savings
  - Gains in performance (on a fixed number of execution cores) due to better code adaptation to the target architecture
  - Gains in terms of scalability: Permits pushing the limit of code scalability with a factor equal to the number of cores of the shared-memory node
- These different gains are proportional to the number of cores in the shared-memory node, a number which will increase significantly in the short term (general use of multi-core processors)
- The only viable solution able to take advantage of the massively parallel architectures of the future (multi-peta, exascale, ...).
Tools
Tools

- SCALASCA
- TAU
- TotalView
Description

SCALASCA is a graphical tool for performance analysis of parallel applications. Principal characteristics:

- Support for MPI and multithreaded/OpenMP applications
- Profiling and tracing modes (limited to `MPI_THREAD_FUNNELED` for traces)
- Identification/automatic analysis of common performance problems (using trace mode)
- Unlimited number of processes
- Support for hardware counters (via PAPI)

Use

- Compile your application with `skin f90` (or other compiler).
- Execute with `scan mpirun`. Use the option `-t` for the trace mode.
- Visualize the results with `square`. 
## Description

TAU is a graphical tool for performance analysis of parallel applications. Principal characteristics:

- Support for MPI and multithreaded/OpenMP applications
- Profiling and tracing modes
- Unlimited number of processes
- Support for hardware counters (via PAPI)
- Automatic instrumentation of loops
- Memory allocations track
- I/O track
- Call tree
- 3D visualization (useful for comparing processes/threads to each other)
TotalView is a graphical debugging tool for parallel applications. Key features:

- Support for MPI and multithreaded/OpenMP applications
- Support for C/C++ and Fortran95
- Integrated memory debugger
- Maximum number of processes (depending on the license)

Use

- Compile your application with `-g` and a not very aggressive level of optimisation.
Hands-on Exercises
Hands-on Exercises

- TP1 — Hybrid MPI and OpenMP — Global synchronization
- TP2 — Hybrid MPI and OpenMP — Parallel PingPong
- TP3 — HYDRO, from MPI to hybrid MPI and OpenMP version
Objective
Synchronize all of the OpenMP threads located on the different MPI processes.

Statement
You are asked to complete the `barrier_hybride.f90` file so that all the OpenMP threads on the different MPI processes would be synchronized during a call to the `barrierMPIOMP` subroutine.
Objective
Measure the network sustained bandwidth between two nodes.

Statement
You are asked to write a hybrid parallel PingPong code for measuring the network sustained bandwidth between two nodes.

- In the first version, you will use the `MPI_THREAD_FUNNELED` level of thread support (the application can launch several threads per process but only the main thread can make MPI calls). In this case, the number of parallel communication flows will be equal to the number of MPI processes per node.

- In the second version, you will use the `MPI_THREAD_MULTIPLE` level of thread support (entirely multithreaded without restrictions) with one MPI process per node. In this case, the number of parallel communication flows will be equal to the number of OpenMP threads which participate in the communications.
Objective
Parallelize an application using MPI and OpenMP.

Statement
You are asked to start with the HYDRO MPI parallel version application.
1. You should implement a new level of parallelism by adding OpenMP directives to construct a hybrid parallel version but with only one parallel OpenMP region.
2. Compare the hybrid performance obtained with that of the MPI version. Does it have good scalability?
3. What improvements can be made to obtain better performances? Tests and compare.
Appendices
Appendices

- SBPR on older architectures
- Case Study: Poisson3D
**MPI_THREAD_FUNNELED** version of SBPR: Results on Vargas

4 links in //, DDR Infiniband, peak throughput 8 GB/s.

<table>
<thead>
<tr>
<th>MPI x OMP per node</th>
<th>Total throughput (MB/s) Message of 1 MB</th>
<th>Total throughput (MB/s) Message of 10 MB</th>
<th>Total throughput (MB/s) Message of 100 MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 x 32</td>
<td>1016</td>
<td>1035</td>
<td>959</td>
</tr>
<tr>
<td>2 x 16</td>
<td>2043</td>
<td>2084</td>
<td>1803</td>
</tr>
<tr>
<td>4 x 8</td>
<td>3895</td>
<td>3956</td>
<td>3553</td>
</tr>
<tr>
<td>8 x 4</td>
<td>6429</td>
<td>6557</td>
<td>5991</td>
</tr>
<tr>
<td>16 x 2</td>
<td>7287</td>
<td>7345</td>
<td>7287</td>
</tr>
<tr>
<td>32 x 1</td>
<td>7412</td>
<td>7089</td>
<td>4815</td>
</tr>
</tbody>
</table>

**Interpretations**

- With a single data flow, we only use one-eighth of the inter-node network bandwidth.
- Saturation of Vargas inter-node network links begins to appear at 8 parallel flows (i.e. 8 MPI processes per node).
- There is total saturation with 16 parallel flows (i.e. 16 MPI processes per node).
- With 16 flows in parallel, we obtain a throughput of 7.35 GiB/s, or more than 90% of the available peak inter-node network bandwidth!
## MPI_THREAD_FUNNELED version of SBPR: Results on Babel

Peak throughput: 425 MB/s

<table>
<thead>
<tr>
<th>MPI x OMP par node</th>
<th>Total throughput (MB/s) Message of 1 MB</th>
<th>Total throughput (MB/s) Message of 10 MB</th>
<th>Total throughput (MB/s) Message of 100 MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMP (1 x 4)</td>
<td>373.5</td>
<td>374.8</td>
<td>375.0</td>
</tr>
<tr>
<td>DUAL (2 x 2)</td>
<td>374.1</td>
<td>374.9</td>
<td>375.0</td>
</tr>
<tr>
<td>VN (4 x 1)</td>
<td>374.7</td>
<td>375.0</td>
<td>375.0</td>
</tr>
</tbody>
</table>

### Interpretations

- The use of a single data flow (i.e. one MPI process per node) is sufficient to totally saturate the interconnection network between two neighboring nodes.
- The throughput rate reached is 375 MB/s, or 88% of the peak inter-node network bandwidth.
The MPI_THREAD_MULTIPLE version has a very different performance on Vargas (compared to the MPI_THREAD_FUNNELED version): The throughput does not increase with the number of flows in parallel but remains constant.

Whether there is only one or several flows, we always use just one-eighth of the inter-node network bandwidth. As a result, it is never saturated!

This MPI_THREAD_MULTIPLE approach (i.e. several threads communicating simultaneously within the same MPI process) is, therefore, absolutely unsuitable to the Vargas machine; it is better to choose the MPI_THREAD_FUNNELED approach.
The performances of the `MPI_THREAD_MULTIPLE` and `MPI_THREAD_FUNNELED` versions are comparable on Babel.

The use of only one data flow (i.e. one single communication thread per node) is sufficient to totally saturate the interconnection network between two neighboring nodes.

The throughput reached is 375 MB/s, which is 88% of the peak inter-node network bandwidth.
Presentation of Poisson3D

Poisson3D is an application which resolves Poisson’s equation on the cubic domain \([0,1] \times [0,1] \times [0,1]\) using a finite difference method and a Jacobi solver.

\[
\begin{align*}
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} &= f(x, y, z) \quad \text{in } [0, 1] \times [0, 1] \times [0, 1] \\
\mathbf{u}(x, y, z) &= 0. \quad \text{on the boundaries} \\
\mathbf{f}(x, y, z) &= 2yz(y - 1)(z - 1) + 2xz(x - 1)(z - 1) + 2xy(x - 1)(y - 1) \\
\mathbf{u}_{\text{exact}}(x, y) &= xyz(x - 1)(y - 1)(z - 1)
\end{align*}
\]

Solver

The discretization is made on a regular grid in the three spatial directions (step \(h = h_x = h_y = h_z\)).

The solution is calculated using this Jacobi solver where the solution to the \(n + 1\) iteration is calculated from the immediately preceding \(n\) iteration solution.

\[
u_{ijk}^{n+1} = \frac{1}{6} (u_{i+1jk}^n + u_{i-1jk}^n + u_{ij+1k}^n + u_{ij-1k}^n + u_{ijk+1}^n + u_{ijk-1}^n - h^2 f_{ijk})
\]
3D domain decomposition

The physical domain is split into the three spatial directions.
Case Study: Poisson3D on Babel

**Versions**

Four different versions have been developed:

1. Pure MPI version without computation-communication overlap
2. Hybrid MPI + OpenMP version without computation-communication overlap
3. Pure MPI version with computation-communication overlap
4. Hybrid MPI + OpenMP version with computation-communication overlap

OpenMP versions are all using a fine-grain approach.

**Babel**

All tests have been run on Babel which was an IBM Blue Gene/P system consisting of 10,240 nodes each with 4 cores and 2 GiB of memory.

**Interesting Phenomena**

- Cache effects
- Derived datatypes
- Process mapping
### Cartesian topology and cache use

<table>
<thead>
<tr>
<th>Version</th>
<th>Topology</th>
<th>Time (s)</th>
<th>L1 read (TiB)</th>
<th>DDR read (TiB)</th>
<th>Torus send (GiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI with overlap</td>
<td>16x4x4</td>
<td>52.741</td>
<td>11.501</td>
<td>14.607</td>
<td>112.873</td>
</tr>
<tr>
<td>MPI with overlap</td>
<td>4x16x4</td>
<td>39.039</td>
<td>11.413</td>
<td>7.823</td>
<td>112.873</td>
</tr>
<tr>
<td>MPI with overlap</td>
<td>4x4x16</td>
<td>36.752</td>
<td>11.126</td>
<td>7.639</td>
<td>37.734</td>
</tr>
</tbody>
</table>

Running on 256 Blue Gene/P cores with a size of $512^3$.

- The way the Cartesian topology is split has a major effect.
- The phenomenon appears to be due to cache effects. In $512^3$, The $u$ and $u_{new}$ arrays require 8 MiB/core.
- Depending on the topology, the accesses to the central memory are very different (between 7.6 TiB and 18.8 TiB in *read*). The elapsed time appears strongly correlated with these accesses.
Case Study: Poisson3D

Sub-domain form \( (512^3) \)

- 16x4x4
  - 130 x 130 x 34

- 4x16x4
  - 130 x 130 x 34
Cache effects

- The effect of the Cartesian topology shape is explained by the layout in the caches.
- The $u$ and $u\_new$ tables are split in the $16x4x4$ topology into $(34, 130, 130)$ and in the $4x16x4$ topology into $(130, 34, 130)$.
- In the computation of the exterior domain, the computation of the $i = constant$ faces results in the use of a single $u\_new$ element per line of the L3 cache (which contains 16 doubles).
- The $i = constant$ faces are four times smaller in $4x16x4$ than in $16x4x4$; this explains a big part of the time difference.

To improve the use of caches, we can calculate more $i = constant$ plans in the exterior domain than before.

<table>
<thead>
<tr>
<th>Topology</th>
<th>Plans</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$4x16x4$</td>
<td>1</td>
<td>39.143</td>
</tr>
<tr>
<td>$4x16x4$</td>
<td>16</td>
<td>35.614</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Topology</th>
<th>Plans</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$16x4x4$</td>
<td>1</td>
<td>52.777</td>
</tr>
<tr>
<td>$16x4x4$</td>
<td>16</td>
<td>41.559</td>
</tr>
</tbody>
</table>
The hybrid version is almost always slower than the pure MPI version.

- For an equal number of cores, the communications take twice as much time in the hybrid version ($256^3$ on 16 cores).
- This loss of time comes from sending messages which use the most non-contiguous derived datatypes (plans YZ).
- The construction of these derived datatypes uses only one single element per cache line.
- In the hybrid version, the communication and the filling of the derived datatypes is made by one single thread per process.
  \[ \Rightarrow \] One single flow in memory *read* (or *write*) per computation node. The prefetch unit is capable of storing only two lines of L3 cache per flow.
- In the pure MPI version, four processes per node read or write simultaneously (on faces four times smaller than on the hybrid version).
  \[ \Rightarrow \] Four simultaneous flows which result in faster filling
Cache effects on the derived datatypes: solution

- Replacement of the derived datatypes by manually-filled arrays of 2D faces.
- The copying towards and from these faces is parallelizable in OpenMP.
- The filling is now done in parallel as in the pure MPI version.

Results of some tests ($512^3$):

<table>
<thead>
<tr>
<th></th>
<th>MPI std</th>
<th>MPI no deriv</th>
<th>MPI+OMP std</th>
<th>MPI+OMP no deriv</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 cores</td>
<td>84.837s</td>
<td>84.390s</td>
<td>102.196s</td>
<td>88.527s</td>
</tr>
<tr>
<td>256 cores</td>
<td>27.657s</td>
<td>26.729s</td>
<td>25.977s</td>
<td>22.277s</td>
</tr>
<tr>
<td>512 cores</td>
<td>16.342s</td>
<td>14.913s</td>
<td>16.238s</td>
<td>13.193s</td>
</tr>
</tbody>
</table>

Improvements also appear in the pure MPI version.
The preceding tests show the quantities of data sent on the 3D torus to be variable in function of the topology. The causes are:

- The messages sent between processes which are inside a compute node are not included. A topology in which the processes are well placed, therefore, have a diminished quantity of sent data on the network.

- In addition, the measurements include the transit traffic through each node. A message sent to a process located on a node non-adjacent to that of the sender will therefore be measured many times (generating real traffic and producing contention on the network links).

### MPI communications

<table>
<thead>
<tr>
<th>Version</th>
<th>Topology</th>
<th>Time (s)</th>
<th>L1 read (TiB)</th>
<th>DDR read (TiB)</th>
<th>Torus send (GiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI without overlap</td>
<td>16x4x4</td>
<td>42.826</td>
<td>11.959</td>
<td>9.265</td>
<td>112.873</td>
</tr>
<tr>
<td>MPI with overlap</td>
<td>8x8x4</td>
<td>45.748</td>
<td>11.437</td>
<td>10.716</td>
<td>113.142</td>
</tr>
<tr>
<td>MPI with overlap</td>
<td>16x4x4</td>
<td>52.741</td>
<td>11.501</td>
<td>14.607</td>
<td>112.873</td>
</tr>
<tr>
<td>MPI with overlap</td>
<td>32x4x2</td>
<td>71.131</td>
<td>12.747</td>
<td>18.809</td>
<td>362.979</td>
</tr>
<tr>
<td>MPI with overlap</td>
<td>4x16x4</td>
<td>39.039</td>
<td>11.413</td>
<td>7.823</td>
<td>112.873</td>
</tr>
<tr>
<td>MPI with overlap</td>
<td>4x4x16</td>
<td>36.752</td>
<td>11.126</td>
<td>7.639</td>
<td>37.734</td>
</tr>
</tbody>
</table>

P.-Fr. Lavallée — P. Wautelet (IDRIS / LA)
Case Study: Poisson3D on Babel

Comparison: Optimized versus original versions (without overlap)

Scalability of Poisson3D norecov 512

- MPI old
- MPI+OMP old
- MPI new
- MPI+OMP new

Speedup vs. Number of cores
Observations

- The Cartesian topology has an important effect on the performances because of the way in which the caches are re-used.
- The Cartesian topology effects the volume of communication and, therefore, the performances.
- The use of derived datatypes has an impact on memory access.
- Hybrid versions are (slightly) more performant than pure MPI versions as long as the work arrays does not hold in the L3 caches.
- Achieving good performances in the hybrid version is possible, but it is not always easy.
- Important gains can be achieved (also in the pure MPI version).
- A good understanding of the application and of the hardware architecture is necessary.
- The advantage of the hybrid approach is not obvious here (beyond a reduction in memory usage), probably because pure MPI Poisson3D has already an excellent scalability and because a fine-grain OpenMP approach was used.