



Hybrid MPI-OpenMP Programming

Pierre-Francois.Lavallee@idris.fr
Philippe.Wautelet@idris.fr

CNRS — IDRIS

Version 2.2 — 15 June 2015

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>

1 Preamble

2 Introduction

- Moore's Law and Electric Consumption
- The Memory Wall
- As for Supercomputers
- Amdahl's Law
- Gustafson-Barsis' Law
- Consequences for users
- Evolution of Programming Methods
- Presentation of the Machines Used

3 Advanced MPI

- Introduction
- History
- Types of MPI Communications
 - Types of MPI Communications
 - Point-to-Point Send Modes
 - Collective Communications
 - One-Sided Communications
- Computation-Communication Overlap
- Derived Datatypes
- Load Balancing
- Process Mapping

4 Advanced OpenMP

- Introduction
- Limitations of OpenMP
- The Fine-Grain (FG) Classical Approach
- The Coarse-Grain Approach (CG)
- CG vs. FG: additional costs of work-sharing
- CG — Impact on the Code
- CG — Low-Level Synchronizations
- MPI/OpenMP-FG/OpenMP-CG Compared Performances
- Conclusion

5 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: Poisson3D

- Case Study: HYDRO

6 Tools

- SCALASCA
- TAU
- TotalView

7 Appendices

- MPI
 - Factors Affecting MPI Performance
 - Ready Sends
 - Persistent Communications
- Introduction to Code Optimisation
- SBPR on older architectures

8 Hands-on Exercises

- TP1 — MPI — HYDRO
- TP2 — OpenMP — Dual-Dependency Nested Loops
- TP3 — OpenMP — HYDRO
- TP4 — Hybrid MPI and OpenMP — Global synchronization
- TP5 — Hybrid MPI and OpenMP — HYDRO

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. This will be discussed in the sections on Advanced MPI and Advanced OpenMP.
- 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

Statement

Moore's law says that 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.

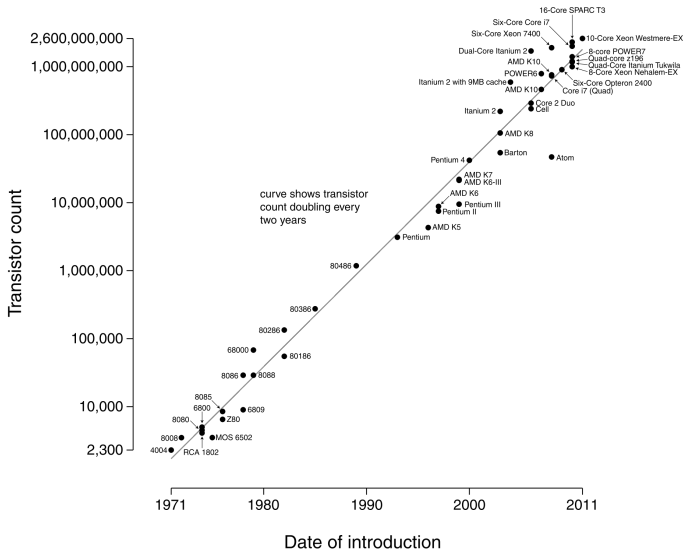
Moore's law and electric consumption

- 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 Haswell-EP chips have up to 18 cores each and can run 36 threads simultaneously; the AMD Abu Dhabi chips have 16 cores.

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

Microprocessor Transistor Counts 1971-2011 & Moore's Law



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

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.

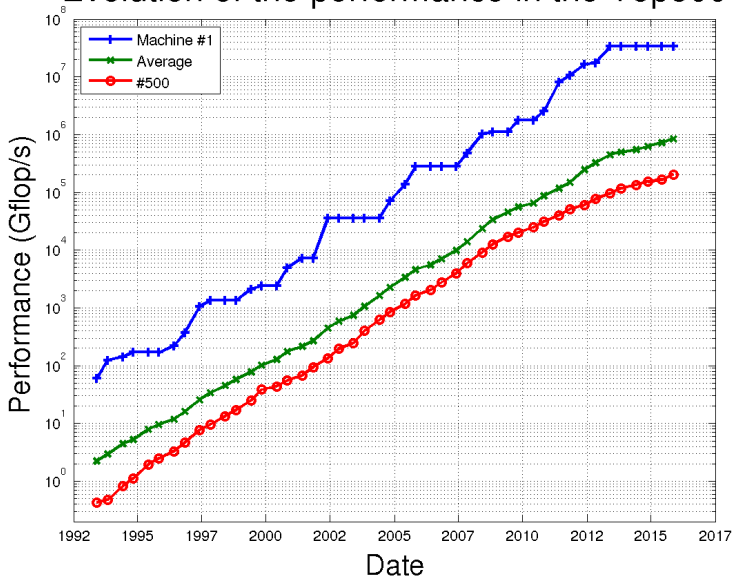
Consequences

- 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.

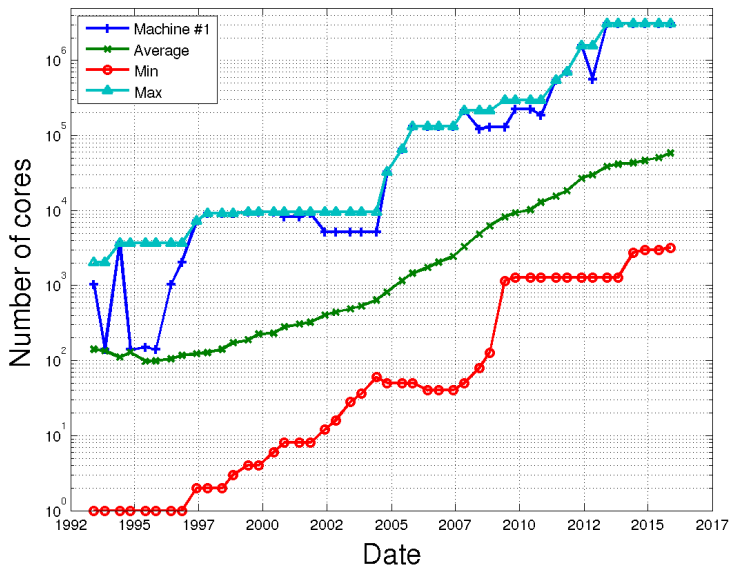
Partial solutions

- Addition of cache memories is essential.
- Access parallelization via several memory banks as found on the vector architectures (Intel Haswell: 4 channels and AMD: 4).
- If the clock frequency of the cores stagnates or falls, the gap could be reduced.

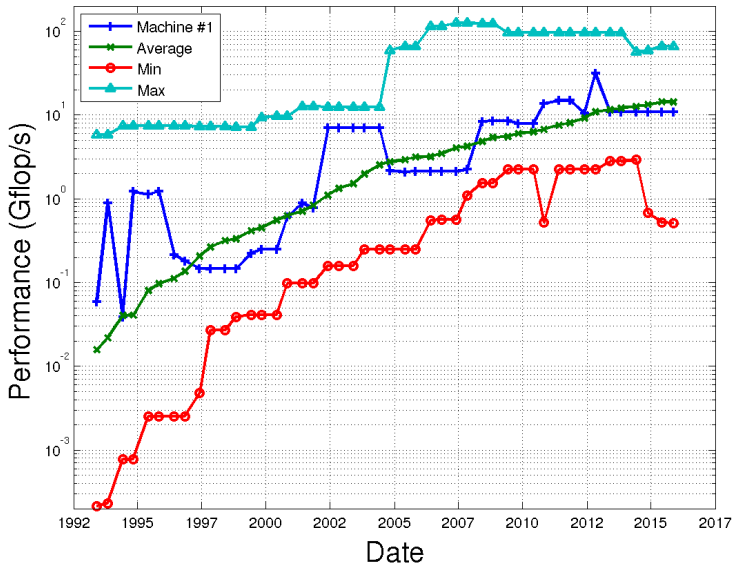
Evolution of the performance in the Top500



Evolution of the number of cores in the Top500



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 and many-cores architectures).
- Emergence of hybrid architectures (examples: GPU or Xeon Phi and standard processors).
- Machine architecture is becoming more complex and the number of layers increasing (processors/cores, memory access, network and I/O).
- Memory per core is stagnating and beginning to decrease.
- Performance per core is stagnating and much lower on some machines than on a simple laptop (IBM Blue Gene).
- Throughput towards the disk is increasing more slowly than the computing power.

Statement

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

$$Sp(P) = \frac{T_s}{T_{//}(P)} = \frac{1}{\alpha + \frac{(1-\alpha)}{P}} < \frac{1}{\alpha} \quad (P \rightarrow \infty)$$

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 α the non-parallelizable part of the application.

Interpretation

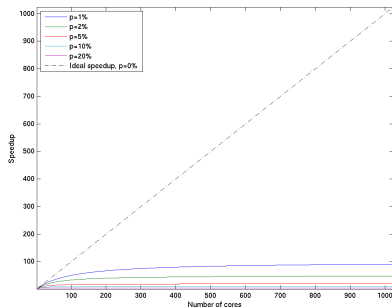
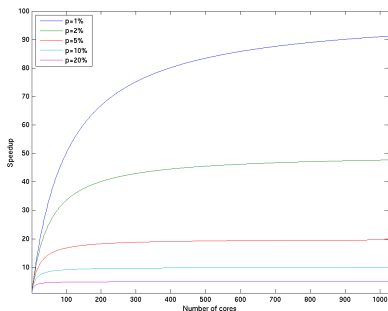
Regardless of the number of cores, the speedup is always less than the inverse of the percentage represented by the purely sequential fraction.

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

Cores	α (%)								
	0	0.01	0.1	1	2	5	10	25	50
10	10	9.99	9.91	9.17	8.47	6.90	5.26	3.08	1.82
100	100	99.0	91.0	50.2	33.6	16.8	9.17	3.88	1.98
1000	1000	909	500	91	47.7	19.6	9.91	3.99	1.998
10000	10000	5000	909	99.0	49.8	19.96	9.99	3.99	2
100000	100000	9091	990	99.9	49.9	19.99	10	4	2
∞	∞	10000	1000	100	50	20	10	4	2



Statement

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:

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

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

Interpretation

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 applications

- It is necessary to exploit a large number of relatively slow cores.
- Tendancy 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).

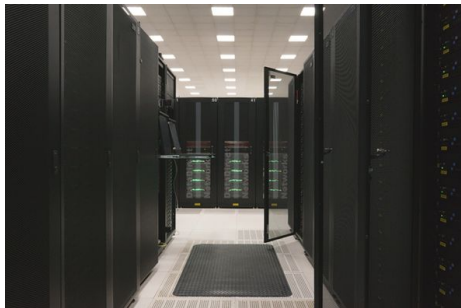
Evolution of programming methods

- 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 use increasing, but the technique is still immature.
- Other forms of hybrid programming also being tested (MPI + GPU, ...), generally with MPI as ingredient.
- 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

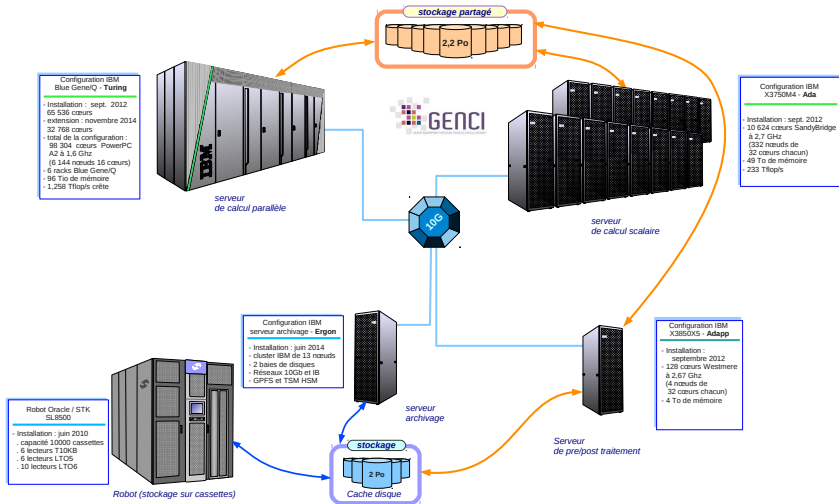


Ada: IBM x3750

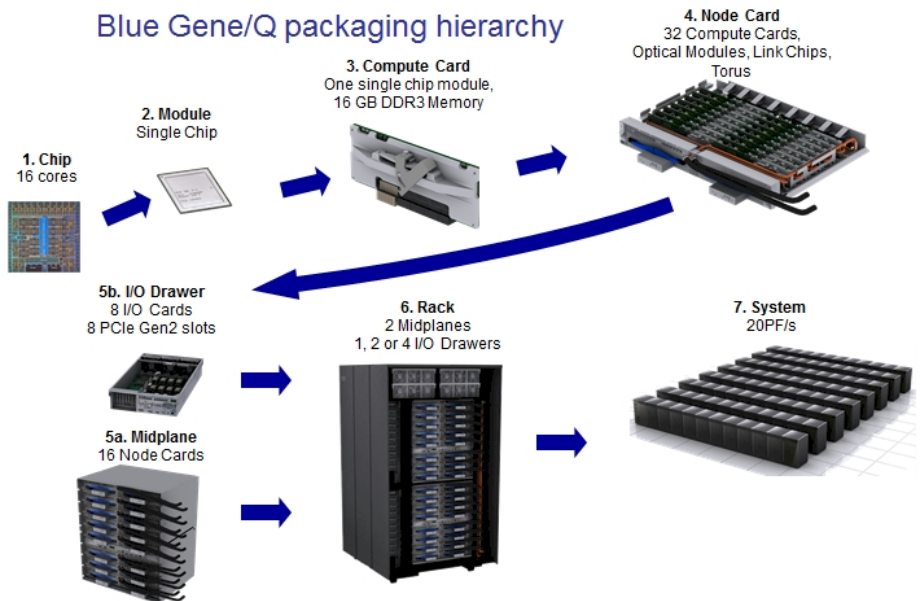


Important numbers

- **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
- 2.2 PiB on shared disks between BG/Q and Intel (50 GiB/s peak)
- 1 MW for the whole configuration (not counting the cooling system)



Blue Gene/Q packaging hierarchy



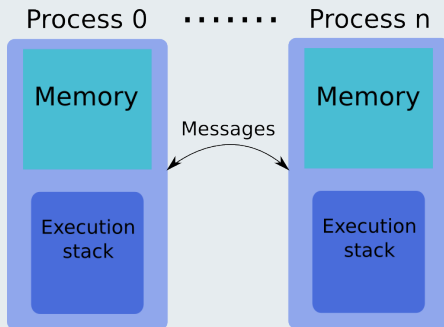
Advanced MPI

Advanced MPI

- Introduction
- History
- Types of MPI Communications
 - Types of MPI Communications
 - Point-to-Point Send Modes
 - Collective Communications
 - One-Sided Communications
- Computation-Communication Overlap
- Derived Datatypes
- Load Balancing
- Process Mapping

Presentation of MPI

- Parallelization paradigm for distributed memory architectures based on the use of a portable library.
- MPI is based on a communications approach of passing messages between processes.
- MPI provides different types of communications:
 - Point-to-point
 - Collective
 - One-sided
- MPI also provides the following functionalities (non-exhaustive list):
 - Execution environment
 - Derived datatypes
 - Communicators and topologies
 - Dynamic process management
 - Parallel I/O
 - Profiling interface



Limitations of MPI

- Final speedup limited by the purely sequential fraction of the code (Amdahl's law).
- Scalability is limited due to the additional costs related to the MPI library and load balancing management.
- Certain types of collective communications become more and more time-consuming as the number of processes increases (e.g. `MPI_Alltoall`).
- No distinction made between processes running in shared or distributed memory; yet, this has a major impact on the communications performance. Most implementations take this into account, but the MPI standard does not provide the information to know if the process communications are within a node or between nodes.
- There are few means provided in the standard to match the MPI processes with the hardware (e.g. process mapping). However, this often can be done through means which are not in the standard.

Up to now

- **Version 1.0** : in June 1994, the MPI forum, with the participation of about forty organizations, came to the definition of a set of subroutines concerning the MPI message passing library.
- **Version 1.1** : June 1995, with only minor changes.
- **Version 1.2** : in 1997, with minor changes for a better consistency of the naming of some subroutines.
- **Version 1.3** : September 2008, with clarifications in MPI 1.2, according to clarifications themselves made by MPI 2.1
- **Version 2.0** : released in July 1997, this version brought deliberately non-integrated, essential additions in MPI 1.0 (process dynamic management, one-sided communications, parallel I/O, etc.).
- **Version 2.1** : June 2008, merge of versions 1.3 and 2.0 with some clarifications in MPI 2.0 but without any changes.
- **Version 2.2** : September 2009, with only "small" additions.

MPI 3.0

- Changes and important additions to version 2.2.
- Published in September 2012.
- Principal changes:
 - Non-blocking collective communications
 - Implementation revision for one-sided communications
 - Fortran (2003-2008) bindings
 - C++ interfaces removed
 - Interfacing of external tools (for debugging and performance measurements)
 - etc.
- See http://meetings.mpi-forum.org/MPI_3.0_main_page.php
<https://svn.mpi-forum.org/trac/mpi-forum-web/wiki>.

State of current implementations

Implementation	Corresponding standard
MPICH	Standard 3.0 since version 3.0 (December 2012)
OpenMPI	Standard 2.1 (version 1.6.5, since 1.3.3) Standard 3.0 (1.8 versions)
IBM Blue Gene/Q	Standard 2.2 (without dynamic process management; based on MPICH2-1.5)
Intel MPI	Standard 3.0 (since 5.0)
IBM PEMPI	Standard 2.2
BullxMPI	Standard 2.1 (version 1.2.8.2)
Cray	Standard 3.0

Comment: Most implementations include part of the 3.0 standard.

Paradigms of MPI communications

MPI provides several approaches to achieve communications between processes:

- Blocking or non-blocking point-to-point
- Persistent point-to-point (see appendices)
- Blocking collective
- Non-blocking collective (from MPI 3.0)
- One-sided (RMA)

Point-to-Point Send Modes

Mode	Blocking	Non-blocking
Standard send	<code>MPI_Send</code>	<code>MPI_Isend</code>
Synchronous send	<code>MPI_Ssend</code>	<code>MPI_Issend</code>
Buffered send	<code>MPI_Bsend</code>	<code>MPI_Ibsend</code>
Ready send	<code>MPI_Rsend</code>	<code>MPI_Irsend</code>
Receive	<code>MPI_Recv</code>	<code>MPI_Irecv</code>

Key terms

It is important to thoroughly understand the definition of certain MPI terms.

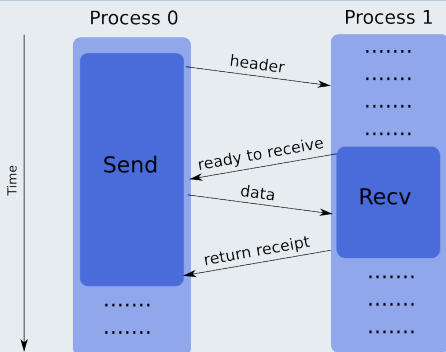
- **Blocking call:** A call is blocking if the memory space used for the communication can be re-used immediately after the call exits. The data that were or will be sent are the data that were in this space at the moment of the call. If it is a receive, the data must have already been received in this space (if the return code is `MPI_SUCCESS`).
- **Non-blocking call:** A non-blocking call returns very quickly, but it does not authorize the immediate re-use of the memory space used in the communication. It is necessary to make sure that the communication is fully completed (with `MPI_Wait`, for example) before using it again.
- **Synchronous send:** A synchronous send involves a synchronization between the concerned processes. There can be no communication before the two processes are ready to communicate. A send cannot start until its receive is posted.
- **Buffered send:** A buffered send involves the copying of data into an intermediate memory space. In this case, there is no coupling between the two processes of communication. The return of this type of send, therefore, does not mean that the receive has occurred.

Synchronous sends

A synchronous send is made by calling the subroutine `MPI_Ssend` or `MPI_Issend`.

Rendezvous protocol

The rendezvous protocol is generally the one used for synchronous sends (depending on implementation used). The return receipt is optional.



Advantages

- Few resources used (no buffer)
- Rapid if the receiver is ready (no copying into a buffer)
- Guaranteed reception through synchronization

Disadvantages

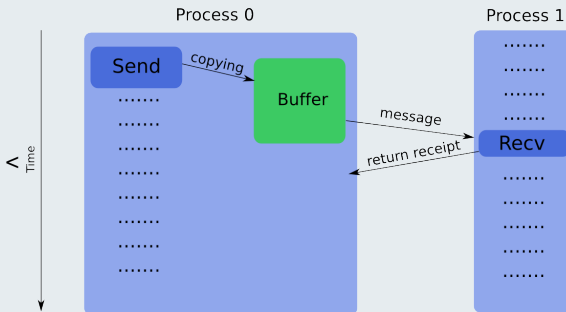
- Waiting time if the receiver is not there/not ready
- Risk of deadlock

Buffered sends

A buffered send is made by calling the `MPI_Bsend` or `MPI_Ibsend` subroutine. The buffers must be managed manually (with calls to `MPI_Buffer_attach` and `MPI_Buffer_detach`). Their allocation must take into account the message header size (by adding the constant `MPI_BSEND_OVERHEAD` for each message instance).

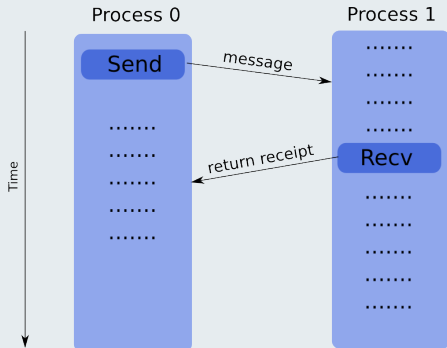
Protocol with user buffer on the sender side

This approach is the one generally used for the `MPI_Bsend` or `MPI_Ibsend`. In this approach, the buffer is located on the sender side and is managed explicitly by the application. A buffer managed by MPI can exist on the receiver side; numerous variants are possible. The return receipt is optional.



Eager protocol

The eager protocol is often used for standard mode sends of small-size messages. Eager can also be used by `MPI_Bsend` for small messages (depending on implementation used) through bypassing the user buffer on the sender side. In this approach, the buffer is on the receiver side. The return receipt is optional.



Advantages

- No need to wait for the receiver (copying into a buffer).
- No risk of deadlock.

Disadvantages

- More resources used (memory occupation by buffers with risk of saturation).
- The send buffers used in the `MPI_Bsend` or `MPI_Ibsend` calls have to be managed manually (often tricky to choose a suitable size).
- A little bit slower than the synchronous sends (when the receiver is ready).
- No guarantee of a good reception (send-receive decoupling).
- Risk of wasted memory space if the buffers are too oversized.
- The application will crash if the buffer is too small.
- There are often hidden buffers managed by the MPI implementation on the sender side and/or on the receiver side (and consuming memory resources).

Standard sends

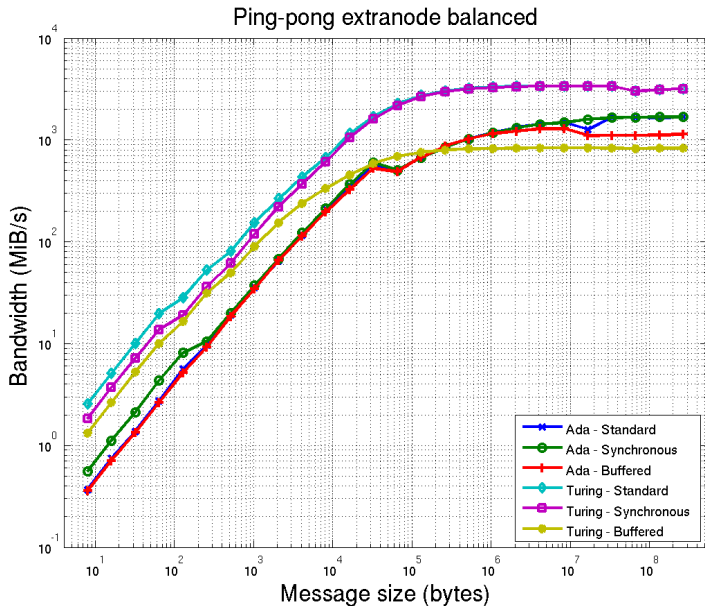
A standard send is made by calling the subroutine `MPI_Send` or `MPI_Isend`. In most implementations, this mode switches from buffered to synchronous when the message size increases.

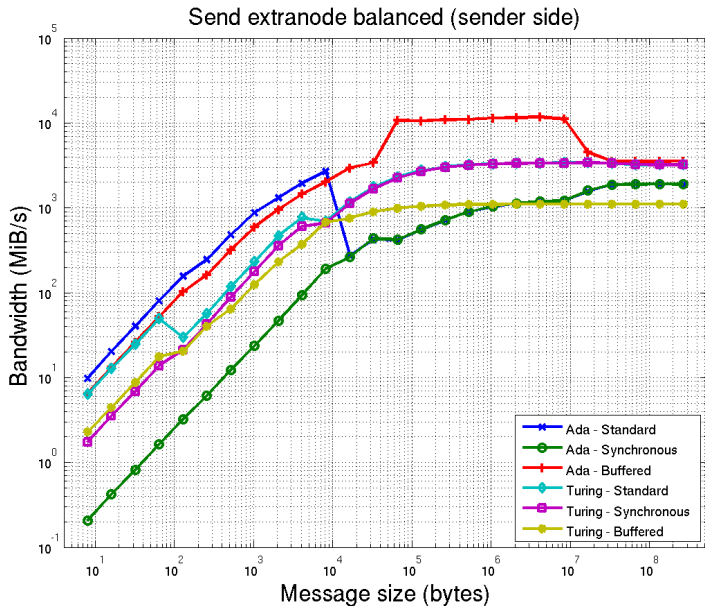
Advantages

- Often the most efficient (because it was maximally adapted by the manufacturer).
- The most portable for dependably good performances.

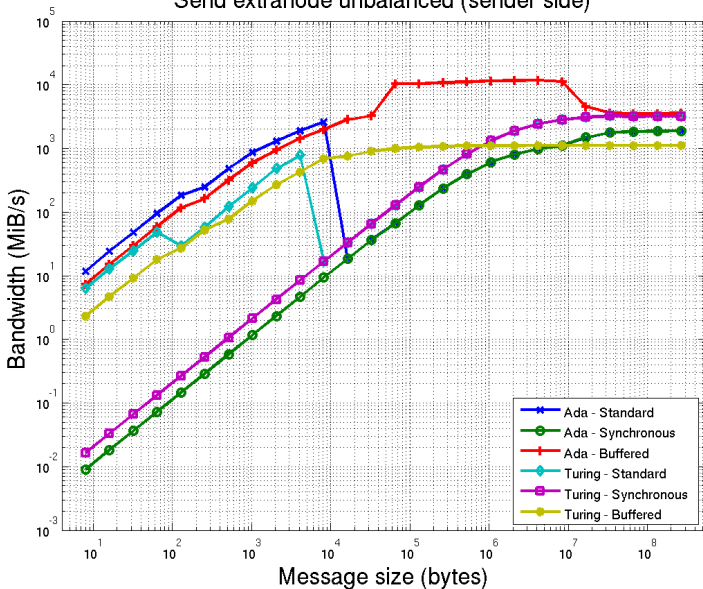
Disadvantages

- Little control over the mode actually used (but usually accessible via environment variables).
- Risk of deadlock depending on mode used.
- Behavior can vary according to the architecture and the problem size.





Send extranode unbalanced (sender side)



Definitions and general characteristics

Collective communications allow carrying out communications involving several processes.

- Collective communications can be highly optimized and some can also have reduction operations; they may be simulated by a series of point-to-point operations but this is neither efficient nor advantageous.
- A collective communication involves all the processes of the communicator used.
- Up to MPI 2.2, only blocking calls were included (i.e. a process does not return before its participation in the communication is completed). The MPI 3.0 standard has added non-blocking calls for collective communications.
- Collective communications do not include or require global synchronization (except for `MPI_Barrier`).
- They never interfere with point-to-point communications.

It is **nearly always** better to choose collective rather than point-to-point communications.

Categories

Collective communications can be divided into three categories:

- Global synchronizations (`MPI_Barrier`); not to be used unless necessary (rare)
- Transfers/data exchanges
 - data broadcasting (global with `MPI_Bcast`, selective with `MPI_Scatter`)
 - data gathering (`MPI_Gather` and `MPI_Allgather`)
 - global exchange (`MPI_Alltoall`)
- Reduction operations (`MPI_Reduce`, `MPI_Allreduce`, `MPI_Reduce_scatter` and `MPI_Scan`)

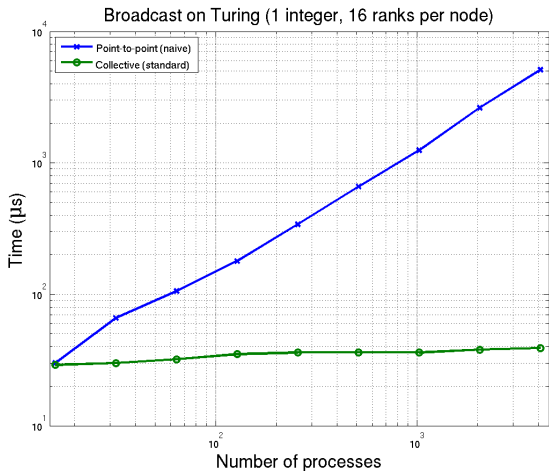
Advantages (in comparison to point-to-point communications)

- Highly optimised.
- Several point-to-point communications in only one operation.

Disadvantages (in comparison to point-to-point communications)

- Can hide very important transfer volumes from the programmer (for example, an `MPI_Alltoall` with 1024 processes involves more than 1 million point-to-point messages).
- There are no non-blocking calls (no longer the case in the MPI 3.0 standard).
- Involves all the communicator processes. Consequently, it is necessary to create sub-communicators if all the processes are not participating in the collective communication.

Collective vs Point-to-Point



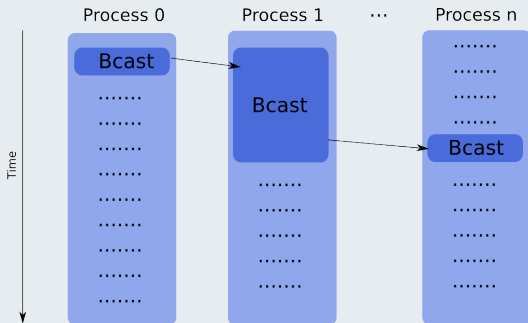
Global broadcast simulated with point-to-point communications (loop of `MPI_Send` on one process and `MPI_Recv` on the others) versus a call to `MPI_Bcast`.

Definition of global synchronization

A global synchronization (or barrier) is an operation in which, at a given moment, all the involved processes will be in the same call. A process arriving in this call cannot exit until all the other processes have also entered the call. However, they are not required to exit the call at the same time.

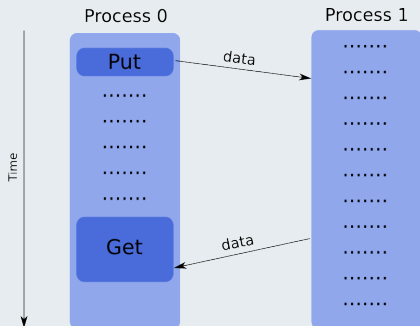
Are collective communications synchronizing?

Collective communications don't involve a global synchronization (except `MPI_Barrier`) and don't require one. Implementations are free to put global synchronization into any of the collective calls; in these cases, the developers need to ensure that their applications are working.



Definition

One-sided communications (Remote Memory Access or RMA) consists of accessing the memory of a distant process in *read* or *write* without the distant process having to manage this access explicitly. The target process does not intervene during the transfer.



General approach

- Creation of a memory window with `MPI_Win_create` to authorize RMA transfers in this zone.
- Remote access in *read* or *write* by calling `MPI_Put`, `MPI_Get` or `MPI_Accumulate`.
- Free the memory window with `MPI_Win_free`.

Synchronization methods

In order to ensure the correct functioning of the application, it is necessary to execute some synchronizations. Three methods are available:

- Active target communication with global synchronization (`MPI_Win_fence`)
- Active target communication with synchronization by pair (`MPI_Win_Start` and `MPI_Win_Complete` for the origin process; `MPI_Win_Post` and `MPI_Win_Wait` for the target process)
- Passive target communication without target intervention (`MPI_Win_lock` and `MPI_Win_unlock`)

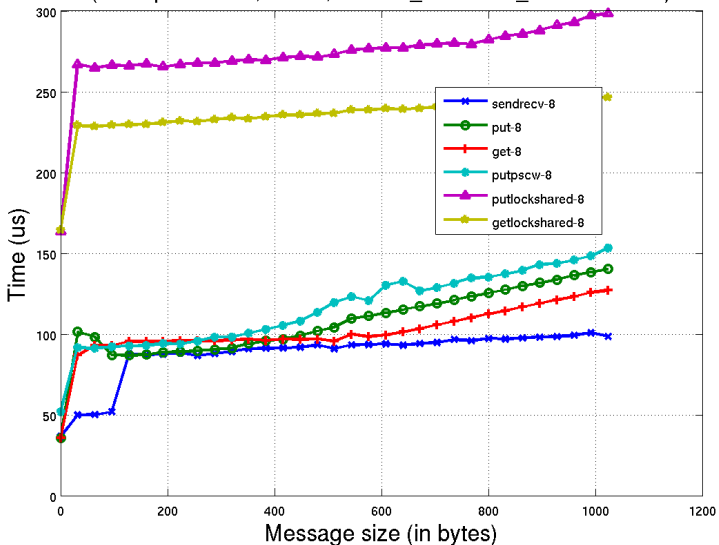
Advantages

- Certain algorithms can be written more easily.
- More efficient than point-to-point communications on certain machines (use of specialized hardware such as a DMA engine, coprocessor, specialized memory, ...).
- The implementation can group together several operations.

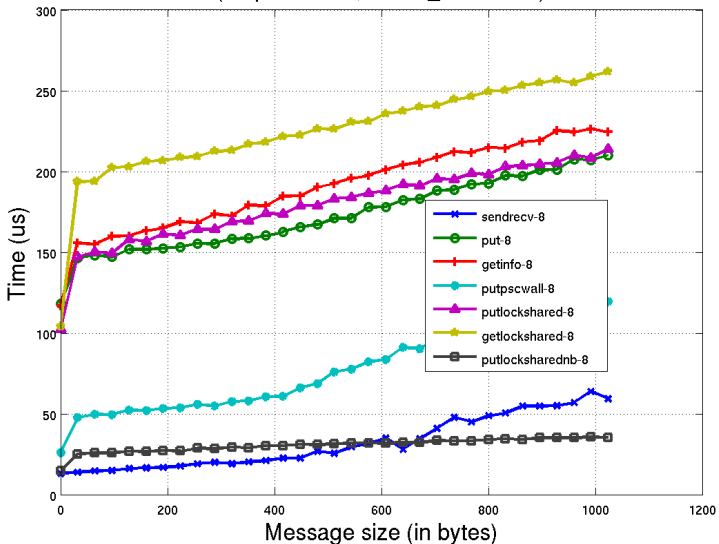
Disadvantages

- Synchronization management is tricky.
- Complexity and high risk of error.
- For passive target synchronizations, it is mandatory to allocate the memory with `MPI_Alloc_mem` which does not respect the Fortran standard (Cray pointers cannot be used with certain compilers).
- Less efficient than point-to-point communications on certain machines.

Halo on Turing with 8 neighbors
(2048 processes, 64CN, PAMID_THREAD_MULTIPLE=1)



Halo on Ada with 8 neighbors
(64 processes, BULK_XFER=no)

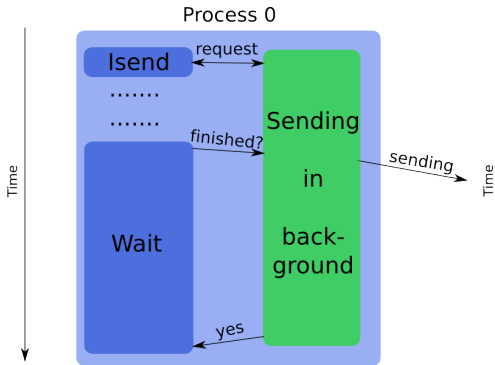


Presentation

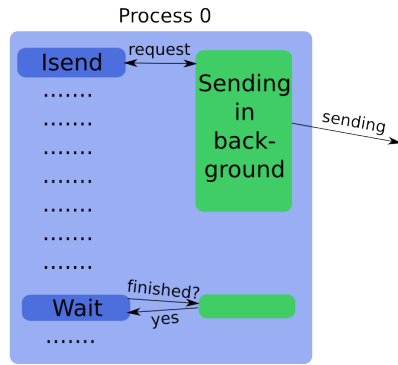
The overlap of communications by computations is a method which allows the communication operations to be carried out in the background while the program continues to run.

- It is thus possible, if the hardware and software architectures permit it, to hide all or part of the communications costs.
- The computation-communication overlap can be seen as an additional level of parallelism.
- This approach is used in MPI by the use of non-blocking subroutines (i.e. `MPI_Isend`, `MPI_Irecv` and `MPI_Wait`).

Partial overlap



Full overlap



Advantages

- Possibility of hiding all or part of communications costs (if the architecture allows it).
- No risk of deadlock.

Disadvantages

- Greater additional costs (several calls for one single send or receive; management of requests).
- More complexity and more complicated maintenance.
- Less efficient on some machines (for example, with transfer starting only at the call `MPI_Wait`).
- Risk of performance loss on the computational kernels; (for example, differentiated management between the area near the border of a domain and the interior area resulting in less efficient use of memory caches).
- Limited to point-to-point communications (was extended to collective communications in MPI 3.0.)

Usage

The message send is made in two steps:

- Initiate the send or the receive by a call to `MPI_Isend` or `MPI_Irecv` (or one of their variants).
- Wait until the end of the local contribution by calling `MPI_Wait` (or one of its variants).

The communications will overlap with all the operations that occur between these two steps. Access to data being received is not permitted before the end of the `MPI_Wait`.

Example

```
do i=1,niter
  ! Initialize communications
  call MPI_Irecv(data_ext,  sz,MPI_REAL,dest,tag,comm, &
                req(1),ierr)
  call MPI_Isend(data_bound,sz,MPI_REAL,dest,tag,comm, &
                req(2),ierr)

  ! Compute the interior domain (data_ext and data_bound
  ! not used) while communications are taking place
  call compute_interior_domain(data_int)

  ! Wait for the end of communications
  call MPI_Waitall(2,req,MPI_STATUSES_IGNORE,ierr)

  ! Compute the exterior domain
  call compute_exterior_domain(data_int,data_bound,data_ext)
end do
```

Overlap level on different machines

Machine	Level
Blue Gene/Q PAMID_THREAD_MULTIPLE=0	32%
Blue Gene/Q PAMID_THREAD_MULTIPLE=1	100%
Ada	33% extranode 0% intranode
NEC SX-8	10%
CURIE (in 2012)	0%

Measurements done by overlapping a computational kernel with a communication kernel having the same execution times and by using different communication patterns (intra/extra-nodes, by pairs, random processes, ...). Depending on the communication pattern, the results can be totally different.

An overlap of 0% means that the total execution time is twice the time of a computational (or communication) kernel.

An overlap of 100% means that the total execution time equals the time of a computational (or communication) kernel.

Presentation

- Derived datatypes can represent data structures of any degree of complexity.
- An important level of abstraction can be reached while hiding the underlying complexity.
- Can be used in all MPI communications, and also in the I/O.

Existing derived datatypes

Datatype	Multiple data	Non contiguous	Any kind of spacing	Heterogeneous
Predefined datatypes				
MPI_Type_contiguous	✓			
MPI_Type_vector	✓	✓		
MPI_Type_indexed	✓	✓	✓	
MPI_Type_struct	✓	✓	✓	✓

`MPI_Type_vector` and `MPI_Type_indexed` give the displacements in whole multiples of the base type.

The `MPI_Type_create_hvector` and `MPI_Type_create_hindexed` variants give the displacements in bytes.

There are also two other derived datatypes: `MPI_Type_create_subarray` to describe subarrays and `MPI_Type_create_darray` for arrays distributed on a group of processes.

Advantages

- Readability
- High level of abstraction
- Non-contiguous datatypes possible
- Heterogeneous datatypes possible
- Message grouping
- Also usable in the I/O

Disadvantages

- Can be difficult to implement
- Heterogeneous datatypes which can be difficult to write
- High level of abstraction => potential loss of program mastery
- Often lower performances

Manual copies in intermediate structures

Manual data management with making copies in intermediate structures:

- Often the most efficient
- Involves additional memory-to-memory copies
- Uses more memory resources
- Manual management of memory space
- Does not make use of possible communication sub-systems (scatter-gather hardware) or specialized parallel filesystems (e.g. PVFS)

Separate messages

Data send in separate messages.

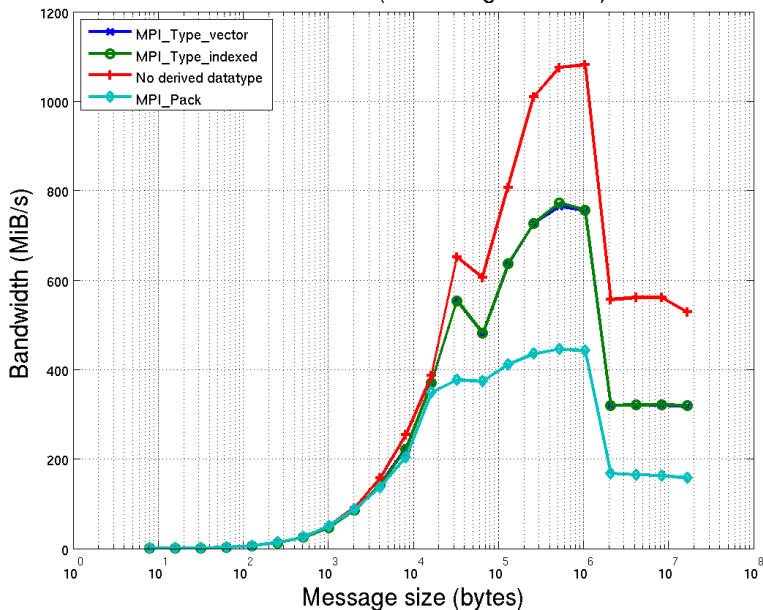
Warning: If numerous messages, **very bad performances**. To be absolutely avoided!

MPI_Pack/MPI_Unpack

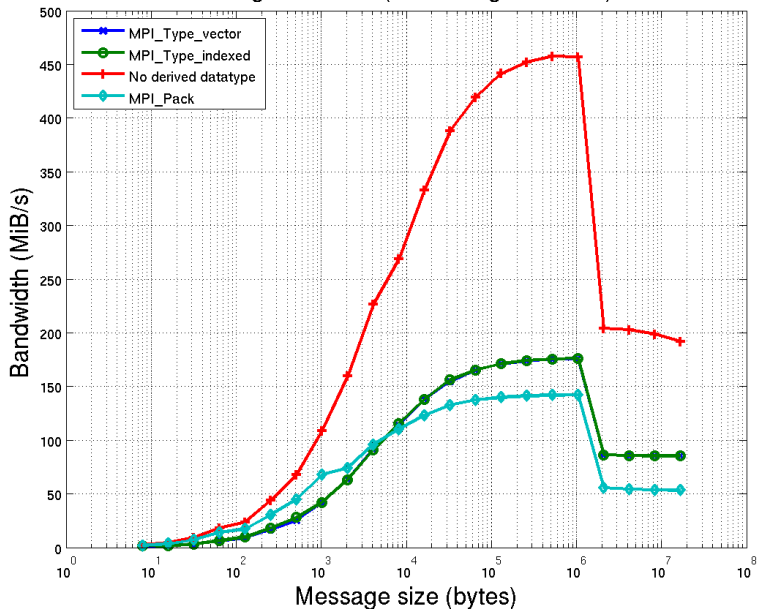
Use of `MPI_Pack/MPI_Unpack` subroutines:

- Often not efficient
- Same disadvantages as manual management with copying into intermediate structures
- Possibility of receiving or preparing a message in several parts
- Can be used with derived datatypes

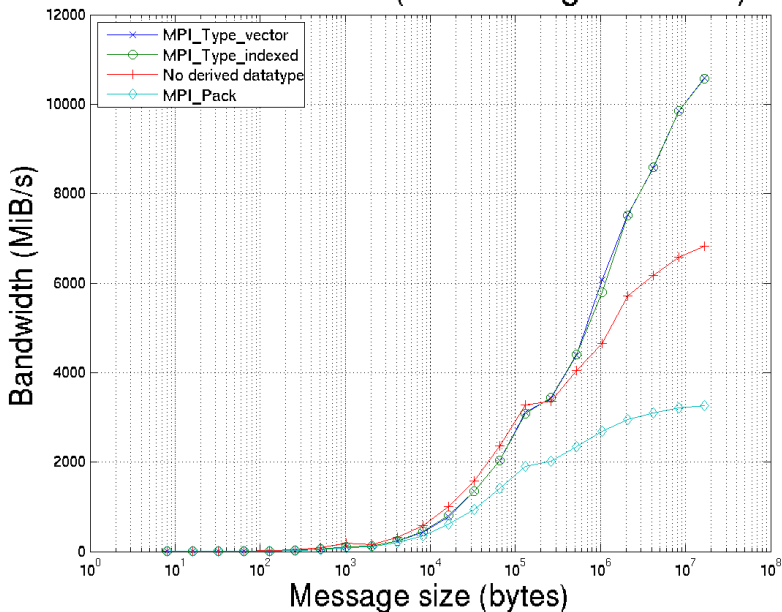
Ada extranode (non-contiguous data)



Turing extranode (non-contiguous data)



Brodie extranode (non-contiguous data)



Definitions

- A parallel MPI application has a perfect load balance when all the processes take the same time between two synchronizations (explicit or not).
- Varying work quantities for the different processes lead to load imbalance resulting in waiting times for the fastest processes, desynchronizations, and lessened scalability.
- The slowest process is the limiting factor.

Another form of imbalance is the memory imbalance between processes. This can be problematic on machines having little memory per node.

Some causes of imbalance

- Poor data partitioning from the conception
- Adaptative mesh refinement (AMR)
- Iterative processes with a different number of iterations according to the variables, cells, ...
- Sources external to the application: OS jitter, non-dedicated resources, ...

Some tips for (re)balancing

- Dynamic balancing during execution with exchanges of cells between the processes (use of the Hilbert space-filling curve, ...)
- Master-slave approach
- Use of partitioning libraries (PT-SCOTCH, ParMETIS, ...)
- Many sub-domains per process
- MPI-OpenMP hybrid approach

Definitions

- Every MPI process has a rank in the `MPI_COMM_WORLD` communicator (varying from 0 to `Nprocesses-1`).
- Every MPI process is placed on a machine node in an immovable way; there are no migrations between nodes during execution.
- On many supercomputers, each process is also placed immovably on a given core (this is called binding or affinity); there are no migrations between cores during execution.
- The mapping corresponds to the relation between the rank of the process and its position on the machine.

Importance

The higher the number of processes, the more communications there are, and the greater the average distance between each process (higher number of memory and network links to cross).

- The latency increases with the distance.
- The network or memory contention increases if messages cross several links.

A bad mapping can have a huge impact. The ideal is to communicate only between close neighbors.

Software adaptation

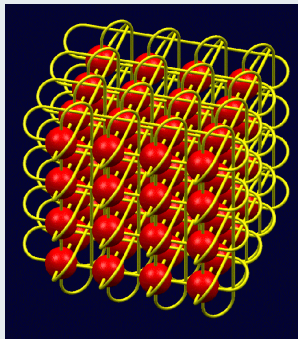
- Use `MPI_Dims_create` and `MPI_Cart_create` and leave as much freedom as possible to MPI: Do not impose dimensions; authorize the renumbering of processes.
- Know your application and its modes/patterns of communications.
- Communicate with close neighbors.
- Divide up the meshes to best match the machine characteristics.

Mapping Tools

- The use of `numactl` enables choosing the affinity on many Linux systems.
- The `hwloc` library provides access to extensive information on the machine node topology and it can determine the binding.
- On Blue Gene/Q, the process mapping is done by using the `--mapping` option of `runjob`.

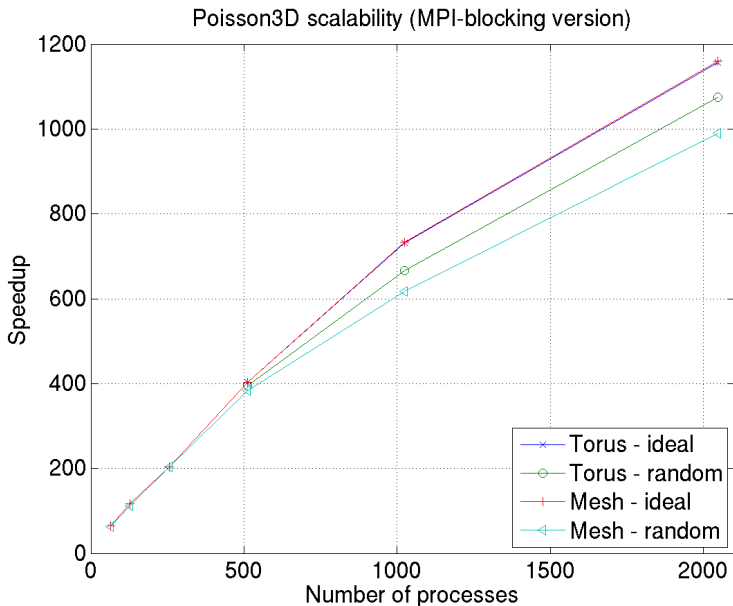
The Blue Gene/P Topology

- For point-to-point and some collective communications, the network topology is a 3D torus.
- Each compute node is connected to its 6 neighbors with bidirectional network links.
- The ideal is to communicate only between close neighbors.



Cartesian Topologies

MPI can optimally place the 3D and 4D Cartesian topologies on the 3D torus (4D if in DUAL or VN mode).



Advanced OpenMP

Advanced OpenMP

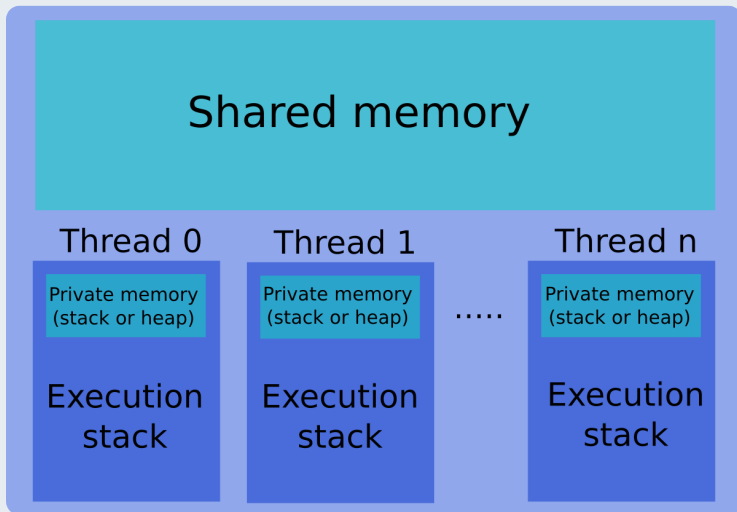
- Introduction
- Limitations of OpenMP
- The Fine-Grain (FG) Classical Approach
- The Coarse-Grain Approach (CG)
- CG vs. FG: additional costs of work-sharing
- CG — Impact on the Code
- CG — Low-Level Synchronizations
- MPI/OpenMP-FG/OpenMP-CG Compared Performances
- Conclusion

Presentation of OpenMP

- Parallelization paradigm for shared memory architecture based on directives to be inserted in the (C, C++, Fortran) code.
- OpenMP consists of a set of directives, a library of functions and a group of environment variables.
- OpenMP is an integral part of all recent Fortran/C/C++ compilers.
- OpenMP can manage:
 - Creation of threads
 - Work-sharing between these threads
 - Synchronization (explicit or implicit) between all the threads
 - Data-sharing attribute of variables (private or shared)

Schematic Drawing

Process



Limitations of OpenMP (1)

- The parallel code scalability is physically limited by the size of the shared memory node on which it is running.
- In practice, the cumulated memory bandwidth inside an SMP node can even further limit the number of cores which can be used efficiently. Contentions due to memory bandwidth limitation can often be bypassed by optimizing the use of caches.
- Pay attention to the additional costs implicit in OpenMP during the creation of threads, stemming from the synchronization between threads (implicit or explicit) or from work-sharing (for example, in the processing of parallel loops).
- Other additional costs which are directly linked to the target machine architecture can also affect performances, such as "false sharing" on shared caches, or the Non-Uniform Memory Access (NUMA) aspect.
- The binding of threads on the machine's physical cores is the responsibility of the system (accessible to the developer through the use of some libraries). The binding can have a very important impact on the performance.

Limitations of OpenMP (2)

- OpenMP does not manage the data locality aspect; this poses a real problem on highly NUMA-based architecture and prevents using any accelerator-based architecture.
- OpenMP is not suitable for dynamic problems (i.e. when the workload fluctuates rapidly during the execution).
- As for any code parallelization, the final speedup will be limited by the purely sequential fraction of a code (Amdahl's law).

Definition

Fine-Grain (FG) Open MP: Use of OpenMP directives to share the work between threads, in particular on parallel loops, by using the `DO` directive.

Advantages

- Simplicity of implementation; the parallelization does not alter the code; only one code version to manage for both the sequential and parallel versions.
- An incremental approach to code parallelization is possible.
- If we use the OpenMP work-sharing directives (`WORKSHARE`, `DO`, `SECTION`), then implicit synchronizations managed by OpenMP will enormously simplify the programming (i.e. parallel loop with reduction).
- On the parallel loops, dynamic load balancing can be done through clause options `SCHEDULE` (`DYNAMIC`, `GUIDED`).

Disadvantages

- The additional costs due to work-sharing and the creation/management of threads can become considerable, particularly when there is low granularity in the parallel code.
- Some algorithms or loop nests cannot be directly parallelizable because they require manual synchronizations, done at a "lower level" than simple barriers, mutual exclusions or single execution.
- In practice, we observe a limited scalability of codes (always inferior to the same code parallelized with MPI), even when they have been optimized well. The lower the granularity of the code, the more this phenomenon is accentuated.

Definition

Coarse-Grain (CG): Code encapsulation in only one parallel region and work distribution on the threads done manually.

Advantages

- For low-granularity codes, the additional costs of work-sharing are much less than with the fine-grain approach.
- Very good scalability when the code is predisposed to it (comparable and often even better than that obtained by parallelizing the code with MPI).
- We will see later on that this is the approach which gives the best performances on an SMP node.

Disadvantages

- Approach which is very intrusive in the code; it is no longer possible to have only one unique version of the code to manage.
- The incremental approach to code parallelization is no longer possible.
- Synchronizations (global or at thread level) are ENTIRELY the programmer's responsibility.
- Work-sharing and load balancing are also the programmer's responsibility.
- Finally, the implementation turns out to be at least as complex as a parallelization with MPI.

Example of a parallel loop

Let us consider the simple parallel loop below, repeated $nbIter = 10^6$ times.

```
do iter=1,nbIter
  do i=1,n
    B(i) = B(i) * a + iter
  enddo
enddo
```

Fine-grain parallelization of the inner loop is very simple: Work-sharing is done by using the OpenMP `DO` directive. Through the `SCHEDULE (RUNTIME)` clause, the distribution mode for the processing of iteration blocks by threads will be chosen just before the execution. Dynamic load balancing can be done with the `DYNAMIC` or `GUIDED` modes, but be careful of the potential additional costs in the execution!

```
do iter=1,nbIter
  !$OMP DO SCHEDULE (RUNTIME)
  do i=1,n
    B(i) = B(i) * a + iter
  enddo
enddo
```

Example of a parallel loop

Coarse-grain parallelization version 1 — with static load balancing, every thread executes non-consecutive iterations:

```
!$ myOMPRank    = OMP_GET_THREAD_NUM()
!$ nbOMPThreads = OMP_GET_NUM_THREADS()
do iter=1,nbIter
  do i=1+myOMPRank,n,nbOMPThreads
    B(i) = B(i) * a + iter
  enddo
enddo
```

This is actually equivalent to a parallel loop with a `STATIC` schedule and a chunk equal to 1 (i.e. only one iteration).

Example of a parallel loop (continued)

Coarse-Grain parallelization version 2 — without load balancing, each thread processes a block of consecutive iterations:

```
!$ myOMPRank    = OMP_GET_THREAD_NUM()
!$ nbOMPThreads = OMP_GET_NUM_THREADS()
nbnLoc = n/nbOMPThreads
iDeb = 1+myOMPRank*nbnLoc
iFin = iDeb+nbnLoc-1
if (myOMPRank==nbOMPThreads-1) iFin = n
do iter=1,nbIter
  do i=iDeb,iFin
    B(i) = B(i) * a + iter
  enddo
enddo
```


Example of a parallel loop (continued)

Coarse-grain parallelization version 3 — with static load balancing, every thread processes a block of consecutive iterations:

```
!$ myOMPRank    = OMP_GET_THREAD_NUM()
!$ nbOMPThreads = OMP_GET_NUM_THREADS()
iDeb = 1+(myOMPRank*n)/nbOMPThreads
iFin = ((myOMPRank+1)*n)/nbOMPThreads
do iter=1,nbIter
  do i=iDeb,iFin
    B(i) = B(i) * a + iter
  enddo
enddo
```

Coarse-Grain versus fine-grain : Additional costs due to work-sharing

	Mono	1 core	2 cores	4 cores	8 cores	16 cores	24 cores	32 cores
FG	2.75	2.65	5.44	7.26	14.75	65.97	187.2	458.4
CG V1	2.75	6.04	13.84	17.06	21.53	46.96	57.0	58.7
CG V2	2.75	6.19	3.10	1.54	0.79	0.40	0.29	0.26
CG V3	2.75	6.19	3.09	1.54	0.77	0.39	0.27	0.19

Conclusions

- The fine-grain parallel loop version gives catastrophic results; the additional costs are so important that the elapsed times increase explosively as the number of threads increases. The results are even worse if we use the `DYNAMIC` distribution mode of iterations on threads because if we do not specify the chunk size, it creates, by default, as many chunks as iterations!
- The two (and only) versions which are scalable are the CG V2 and V3, the impact of load balancing appearing only beyond 24 threads. The performances are excellent, considering the granularity of the parallelized loop (speedup of 14.5 on 32 threads).
- The coarse-grain approach limits the impact of additional costs, which permits an optimal scalability of the parallelized codes.

The coarse-grain approach: Impact on the code

- Contrary to the fine-grain approach, which intrudes very little in the code (only a few additions of OpenMP directives), the coarse-grain approach requires much code rewriting, the necessity of introducing new variables, etc.
- A simple classical example: How to parallelize a loop with a reduction in a coarse-grain version? We can no longer use the REDUCTION clause of the DO directive. It is necessary to do it manually, ...
- Every thread will calculate a local reduction in a private variable, and then it will accumulate the local reductions in the same shared variable, but one thread at a time!
- For the FG version, only one OpenMP directive (with 2 modified or added lines) is used. For the CG version, it is necessary to introduce or modify no fewer than 14 code lines and use 4 directives or function calls of the OpenMP library!!!

Example of the π computation, sequential version

```
program pi
!
! Objective: Calculation of  $\pi$  by the method of rectangles (midpoint)
!
!
!           / 1
!           |   4
!           | ----- dx =  $\pi$ 
!           | 1 + x**2
!           / 0
!
implicit none
integer, parameter :: n=30000000
real(kind=8)       :: f, x, a, h, Pi_calc
integer            :: i

! Integrand
f(a) = 4.0_8 / ( 1.0_8 + a*a )

! Length of the integration interval
h = 1.0_8 / real(n,kind=8)

! Pi calculation
Pi_calc = 0.0_8
do i = 1, n
  x = h * ( real(i,kind=8) - 0.5_8 )
  Pi_calc = Pi_calc + f(x)
end do
Pi_calc = h * Pi_calc

end program pi
```

Example of π computation, OpenMP fine-grain parallel version

```
program pi
!  
! Purpose: to calculate  $\overline{\pi}$  by the method of rectangles (midpoint)  
!  
! 
$$\int_0^1 \frac{1}{1+x^2} dx = \overline{\pi}$$
  
!  
implicit none  
integer, parameter :: n=3000000  
real(kind=8)      :: f, x, a, h, Pi_calc  
integer          :: i  
  
! Integrand  
f(a) = 4.0_8 / ( 1.0_8 + a*a )  
  
! Length of the integration interval  
h = 1.0_8 / real(n,kind=8)  
  
! Pi calculation  
Pi_calc = 0.0_8  
!$OMP PARALLEL DO PRIVATE(x) REDUCTION(+:Pi_calc)  
do i = 1, n  
    x = h * ( real(i,kind=8) - 0.5_8 )  
    Pi_calc = Pi_calc + f(x)  
end do  
!$OMP END PARALLEL DO  
Pi_calc = h * Pi_calc  
end program pi
```

Example of π computation, OpenMP coarse-grain parallel version

```
program pi
!$ use OMP_LIB
implicit none
integer, parameter :: n=3000000
real(kind=8)       :: f, x, a, h, Pi_calc, Pi_calc_loc
integer            :: i, iDeb, iFin, myOMPRank, nbOMPThreads
! Integrand
f(a) = 4.0_8 / ( 1.0_8 + a*a )

! Initialisation of myOMPRank and nbOMPThreads
myOMPRank=0
nbOMPThreads=1
! Length of the integration interval
h = 1.0_8 / real(n,kind=8)
! Pi calculation
Pi_calc = 0.0_8
Pi_calc_loc = 0.0_8
!$OMP PARALLEL PRIVATE(x,myOMPRank,iDeb,iFin) FIRSTPRIVATE(Pi_calc_loc)
!$ myOMPRank = OMP_GET_THREAD_NUM()
!$ nbOMPThreads = OMP_GET_NUM_THREADS()
iDeb = 1+(myOMPRank*n)/nbOMPThreads
iFin = ((myOMPRank+1)*n)/nbOMPThreads
do i = iDeb, iFin
  x = h * ( real(i,kind=8) - 0.5_8 )
  Pi_calc_loc = Pi_calc_loc + f(x)
end do
!$OMP ATOMIC
Pi_calc = Pi_calc + Pi_calc_loc
!$OMP END PARALLEL
Pi_calc = h * Pi_calc
end program pi
```

Low-level synchronizations between two or several threads

- The synchronizations provided by OpenMP (`BARRIER`, `SINGLE`, `ATOMIC`, etc) are not suitable to coarse-grain parallelization which requires low-level (manual) synchronizations.
- Unfortunately, nothing is available for this purpose in OpenMP. The programmer, therefore, needs to emulate these functionalities by using the shared variables and the `FLUSH` directive to exchange information between two or several other threads.
- Some algorithms are not parallelizable without resorting to this type of synchronization.
- It is complicated to code, a source of errors, and it alters the code, but it is indispensable, ...

The OpenMP `FLUSH` directive

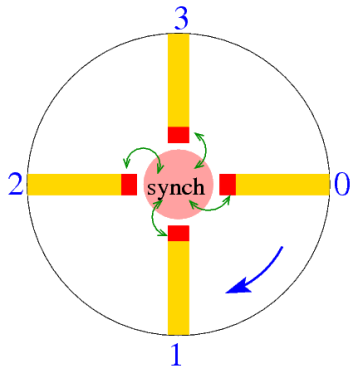
- Reminder: The OpenMP `FLUSH` directive can update all the memory hierarchy of a thread associated with the shared variables listed in argument. If there is no argument at the `FLUSH` command, all the shared variables which are visible by the thread are updated (very expensive!). More precisely, on the thread which makes the call to the `FLUSH`:
 - The shared variables listed in argument, and having been updated since the last `FLUSH`, will have their value copied in shared memory,
 - The shared variables listed in argument, and not updated since the last `FLUSH`, will have their cache line invalidated. In this way, every new reference to this variable will correspond to a *read* in the shared memory of the variable value.
- Let us assume that an algorithm composed of several parts (T_1, T_2, \dots, T_n) has the following dependencies:

$$\forall i = 1, \dots, n - 1 \quad T_{i+1} \text{ cannot start to run until } T_i \text{ is finished.}$$

Low-level synchronization between two threads, necessary to manage these dependencies, can be implemented as in the code following.

Example of dependencies managed "by hand"

```
program ring1
!$ use OMP_LIB
implicit none
integer :: rank,nb_threads,synch=0
!$OMP PARALLEL PRIVATE(rank,nb_threads)
  rank=OMP_GET_THREAD_NUM()
  nb_threads=OMP_GET_NUM_THREADS()
  do
    !$OMP FLUSH(synch)
    if(synch==mod(rank-1+nb_threads,nb_threads)) &
      exit
  end do
  print *, "Rank:", rank, ";synch:", synch
  synch=rank
  !$OMP FLUSH(synch)
!$OMP END PARALLEL
end program ring1
```



An easy trap - example

```
program ring2-wrong
  !$ use OMP_LIB
  integer :: rank,nb_threads,synch=0,counter=0
  !$OMP PARALLEL PRIVATE(rank,nb_threads)
    rank=OMP_GET_THREAD_NUM()
    nb_threads=OMP_GET_NUM_THREADS()
    if (rank == 0) then ; do
      !$OMP FLUSH(synch)
      if(synch == nb_threads-1) exit
    end do
    else ; do
      !$OMP FLUSH(synch)
      if(synch == rank-1) exit
    end do
  end if
  counter=counter+1
  print *, "Rank:", rank, ";synch:", synch, ";counter:", counter
  synch=rank
  !$OMP FLUSH(synch)
!$OMP END PARALLEL
  print *, "Counter = ", counter
end program ring2-wrong
```

A difficult and vicious trap - example

```
program ring3-wrong
  !$ use OMP_LIB
  integer :: rank,nb_threads,synch=0,counter=0
  !$OMP PARALLEL PRIVATE(rank,nb_threads)
    rank=OMP_GET_THREAD_NUM(); nb_threads=OMP_GET_NUM_THREADS()
    if (rank == 0) then ; do
      !$OMP FLUSH(synch)
      if(synch == nb_threads-1) exit
    end do
  else ; do
    !$OMP FLUSH(synch)
    if(synch == rank-1) exit
  end do
end if
print *, "Rank:", rank, ";synch:", synch, "
!$OMP FLUSH(counter)
counter=counter+1
!$OMP FLUSH(counter)
synch=rank
!$OMP FLUSH(synch)
!$OMP END PARALLEL
print *, "Counter = ",counter
end program ring3-wrong
```

Commentaries on the previous codes

- In ring2-wrong, we did not flush the shared counter variable before and after incrementing it. The end result can potentially be wrong.
- In ring3-wrong, the compiler can inverse the lines,

```
counter=counter+1  
!$OMP FLUSH(counter)
```

and the lines,

```
synch=rank  
!$OMP FLUSH(synch),
```

releasing the following thread before the counter variable has been incremented. Here also, the end result can be potentially wrong.

- To solve this problem, it is necessary to flush the two variables *counter* and *synch* just after the incrementation of the counter variable, thereby imposing an order to the compiler.
- The correct code is found below.

The correct code...

```
program ring4
  !$ use OMP_LIB
  integer :: rank, nb_threads, synch=0, counter=0
  !$OMP PARALLEL PRIVATE(rank, nb_threads)
    rank=OMP_GET_THREAD_NUM()
    nb_threads=OMP_GET_NUM_THREADS()
    if (rank == 0) then ; do
      !$OMP FLUSH(synch)
      if(synch == nb_threads-1) exit
    end do
  else ; do
    !$OMP FLUSH(synch)
    if(synch == rank-1) exit
  end do
end if
print *, "Rank:", rank, "; synch:", synch, "
!$OMP FLUSH(counter)
counter=counter+1
!$OMP FLUSH(counter, synch)
synch=rank
!$OMP FLUSH(synch)
!$OMP END PARALLEL
print *, "Counter = ", counter
end program ring4
```

Problem description and analysis

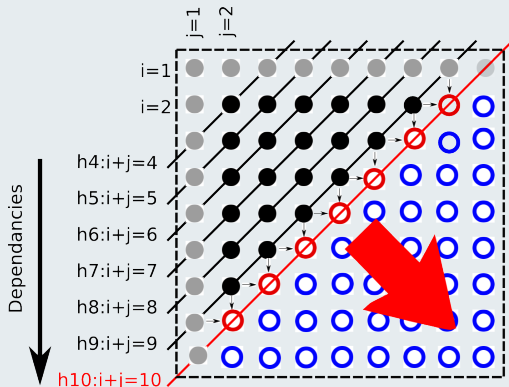
- Let us consider the following code:

```
! Nested loops with double dependencies  
do j = 2, ny  
  do i = 2, nx  
     $V(i, j) = (V(i, j) + V(i-1, j) + V(i, j-1)) / 3$   
  end do  
end do
```

- This is a classical problem in parallelism found, for example, in the NAS Parallel Benchmarks (LU application).
- Because of backward dependency in i and in j , neither the loop in i , nor the loop in j , is parallel. (Every iteration in i or j depends on the previous iteration.)
- Parallelizing the loop in i or the loop in j with the OpenMP `PARALLEL DO` directive would give wrong results.
- Nevertheless, it is still possible to expose parallelism of these nested loops by doing the calculations in an order that does not break the dependencies.
- There are at least two methods to parallelize these nested loops: the hyperplane algorithm; and software pipelining.

How to expose parallelism?

- The principle is simple: We are going to work on the hyperplanes of the equation $i + j = cst$, each corresponding to a matrix diagonal.
- On a given hyperplane, the elements are updated independently (of each other), so these operations can be carried out in parallel.
- However, there is a dependence relation between the different hyperplanes: The elements of H_n hyperplane cannot be updated until the element updating of H_{n-1} hyperplane has finished.



Code rewriting

- A code rewriting of the hyperplane algorithm is, therefore, required; with an outer loop on the hyperplanes (non-parallel because of dependencies between hyperplanes), and with an inner parallel loop on the elements belonging to the hyperplane which permits updating in any order.
- The code can be rewritten with the following form:

```
do h = 1,nb_hyperplane      ! Non // loop, dependencies between hyperplanes
  call calcul(INDI,INDJ,h)  ! compute i and j indices for the h hyperplane
  do e = 1,nb_element_hyperplane ! loop on the elements of the h hyperplane
    i = INDI(e)
    j = INDJ(e)
    V(i,j) = (V(i,j) + V(i-1,j) + V(i,j-1))/3 ! Update of V(i,j)
  enddo
enddo
```

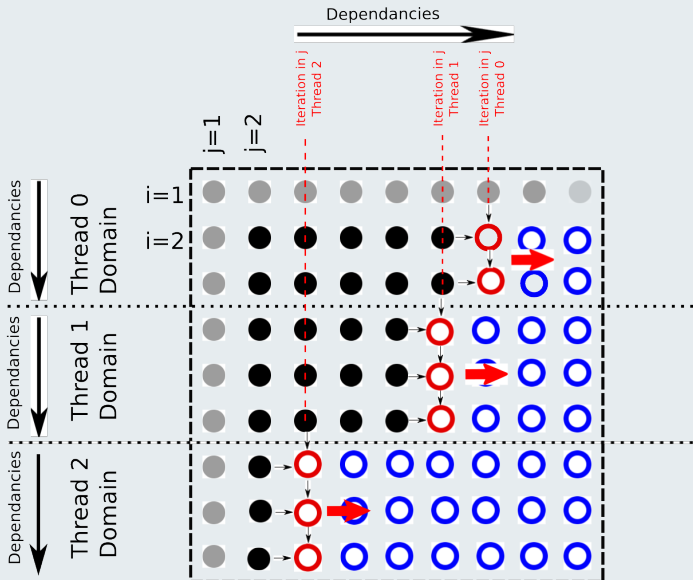
Some complementary remarks

- Once the code is rewritten, the parallelization is very simple because the inner loop is already parallel. We haven't had any need to resort to low-level synchronizations to implement the hyperplane algorithm.
- The performances obtained, unfortunately, are not optimal; the main reason being the poor use of caches due to the diagonal access (non-contiguous in memory) of the V matrix elements.

How to expose parallelism?

- The principle is simple: We are going to parallelize the innermost loop by block, first by playing with the iterations of the outer loop, followed by manually synchronizing the threads between each other, always being careful not to break the dependencies.
- We cut the matrix into horizontal slices and attribute each slice to a thread.
- The algorithm dependencies impose the following: Thread 0 processes an iteration of the outer loop j which must have a value superior to thread 1 (one), which itself must have a value superior to that of thread 2, and so on. If we do not respect this condition, we will break the dependencies!
- Specifically, when a thread has finished processing the j^{th} column of its domain, it must, before continuing, verify that the preceding thread has already finished processing the next column ($j + 1^{\text{th}}$). If this is not the case, it is necessary for it to wait until this condition has been fulfilled.
- To implement this algorithm, it is necessary to synchronize the threads constantly, in pairs, and to not release a thread until the aforementioned condition has been fulfilled.

The dependencies...



Implementation

- Finally, the implementation of this method can be done in the following way:

```
myOMPRank = ...
nbOMPThrds = ...
call calcul_borne(iDeb,iFin)
do j= 2,n
  ! Thread is blocked (except 0) as long
  ! as the previous has not finished
  ! the treatment of the j+1 iteration
  call sync(myOMPRank,j)
  ! // loop distributed on the threads
  do i = iDeb,iFin
    ! Update of V(i,j)
     $V(i,j) = (V(i,j) + V(i-1,j) + V(i,j-1))/3$ 
  enddo
enddo
```

Characteristics of the code and target machine

- The performance tests were carried out with the hydrodynamic code HYDRO (the one used in the hands-on exercises)
- The target machine is a shared memory node of Vargas (IBM SP6, 32 cores per node)
- We compared three parallelized versions of HYDRO:
 - 1 a parallelized version with MPI, 2D domain decomposition, use of derived datatypes, and no computation-communication overlap;
 - 2 a parallelized version with fine-grain OpenMP, STATIC scheduling; and
 - 3 a parallelized version with coarse-grain OpenMP, 2D domain decomposition, and thread-to-thread synchronization to manage the dependencies.

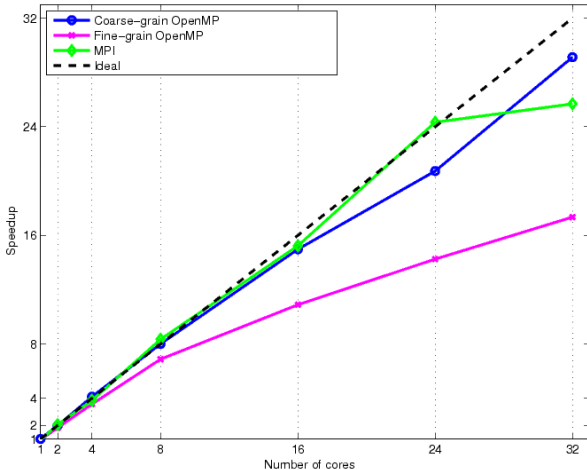
Characteristics of data sets used

We used three data sets, each having the same total size (i.e., the same total number of points), but having different distributions with two (x and y) directions:

- 1 Elongated domain in the y direction: $n_x = 1000$ and $n_y = 100000$,
- 2 Square domain: $n_x = n_y = 10000$,
- 3 Elongated domain in the x direction: $n_x = 100000$ and $n_y = 1000$.

Results for the domain $n_x = 100000, n_y = 1000$

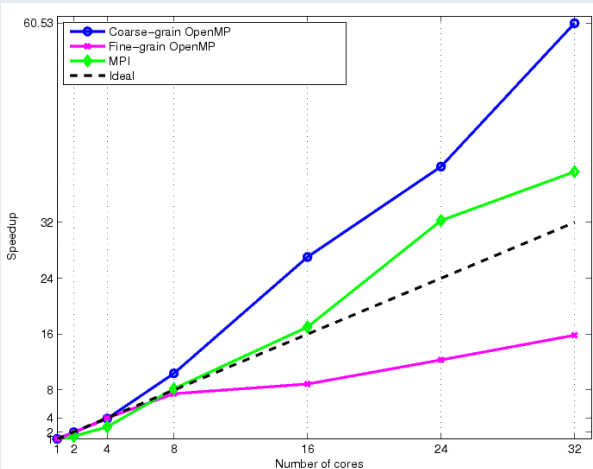
Time (s)	Mono	1 core	2 cores	4 cores	8 cores	16 cores	24 cores	32 co.
mpi2D	361.4	383.8	170.9	91.4	42.0	23.0	14.4	13.6
ompgf	361.4	371.6	193.2	98.4	51.0	32.2	24.6	20.2
ompcg	361.4	350.4	177.3	85.3	43.8	23.4	16.9	12.0



MPI/OpenMP-FG/OpenMP-CG Performances

Results for $n_x = 1000, n_y = 100000$

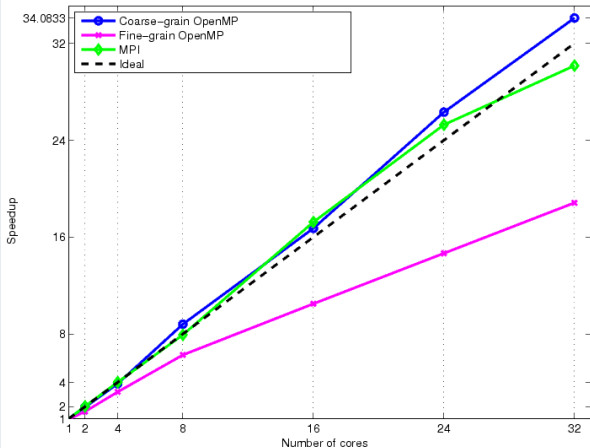
Time (s)	Mono	1 core	2 cores	4 cores	8 cores	16 cores	24 cores	32 co.
mpi2D	1347.2	1310.6	638.3	318.7	106.1	51.0	26.9	22.1
ompgf	1347.2	879.0	461.0	217.9	116.2	98.0	70.5	54.8
ompcg	1347.2	868.0	444.1	222.7	83.7	32.1	21.7	14.3



MPI/OpenMP-FG/OpenMP-CG Performances

Results for $nx = 10000, ny = 10000$

Time (s)	Mono	1 core	2 cores	4 cores	8 cores	16 cores	24 cores	32 co.
mpi2D	449.9	456.0	223.7	112.2	56.8	26.1	17.8	14.9
ompgf	449.9	471.9	283.4	140.5	71.8	42.9	30.7	23.9
ompcg	449.9	455.7	230.8	115.9	51.1	26.9	17.1	13.2



OpenMP fine-grain version

- Although we are working on identically-sized domains, the performances obtained on a single core can vary by as much as triple. The reason can be found in whether the caches are used well or not.
- Up to 4 cores, the three versions give more or less similar performances regardless of the data set. Beyond 4 cores, the OpenMP FG version suffers from a problem of degraded scalability compared to the MPI and OpenMP CG versions.
- Consequently, except for when using a limited number of cores, the OpenMP FG version is always largely dominated by the MPI or OpenMP CG versions, even if it is scalable in a regular but limited way, up to 32 cores.

MPI and OpenMP coarse-grain versions

- Up to 24 cores, the MPI and OpenMP CG versions have comparable scalabilities, both of which are perfect and sometimes even super-linear. (The re-use of caches progressively improves as the size of the local sub-domains decreases.)
- Beyond 24 cores, the MPI version seems to slow down, while the OpenMP CG version continues being perfectly scalable.
- Over 32 cores, it is always the OpenMP CG version which gives the best results.
- Nevertheless, it is important to note that the MPI version can still be optimized; for example, implementing computation-communication overlap could enable it to be scalable beyond 24 cores.

- The growing use of shared memory machines, as well as the increase in the number of cores available inside a node, require us to reconsider the way in which we program applications.
- If we are seeking maximal performance inside a node, it is the OpenMP Coarse-Grain approach that must be used. This requires extensive means (time and technical competence); it is at least as complicated to implement as an MPI version. Debugging is particularly complex. This approach is reserved for specialists who skillfully master parallelism and its traps.
- The usage simplicity and implementation rapidity of an OpenMP Fine-Grain version are its main advantages. On condition of being coded well (minimization of synchronization barriers and parallel regions), the performances, according to the type of algorithm (especially according to the granularity), can go from medium to relatively good. Debugging, with this version also, remains particularly complex. This approach, however, can be used by everyone.
- MPI obtains good performances on a shared memory node but the OpenMP CG version outclasses it in terms of scalability. Nevertheless, it can be optimized by implementing computation-communication overlap. In any case, it remains indispensable when it is necessary to surpass the use of a single node.

Hybrid programming

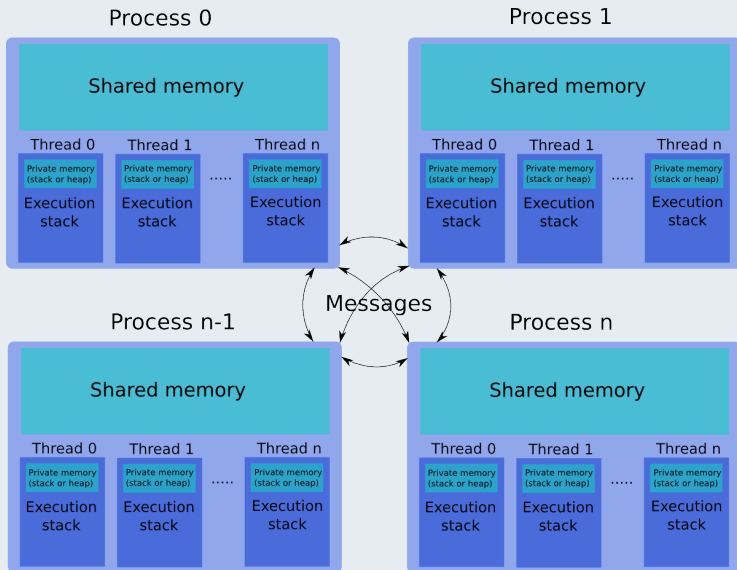
5 Hybrid programming

- 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: Poisson3D
- Case Study: HYDRO

Definitions

- 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.

Schematic drawing



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 launching 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.

MPI_Init_thread

```
int MPI_Init_thread(int *argc, char *((*argv)[]),  
                   int required, int *provided)  
MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
```

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:

```
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`.)

```
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`).

Restrictions on MPI collective calls (2)

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 the OpenMP *parallel* regions, in the OpenMP master regions, or in protected zones, and by calling `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

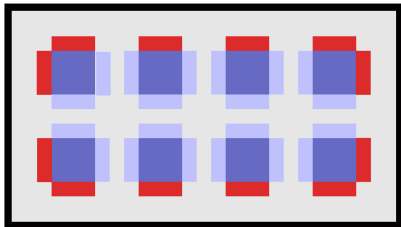
Implementation	Level Supported	Remarks
MPICH	<code>MPI_THREAD_MULTIPLE</code>	
OpenMPI	<code>MPI_THREAD_MULTIPLE</code>	Must be compiled with <i>-enable-mpi-threads</i>
IBM BlueGene/Q	<code>MPI_THREAD_MULTIPLE</code>	
IBM PEMPI	<code>MPI_THREAD_MULTIPLE</code>	
BullxMPI	<code>MPI_THREAD_FUNNELED</code>	
Intel - MPI	<code>MPI_THREAD_MULTIPLE</code>	Use <i>-mt_mpi</i>
SGI - MPT	<code>MPI_THREAD_MULTIPLE</code>	Use <i>-lmpi_mt</i>

Why memory savings?

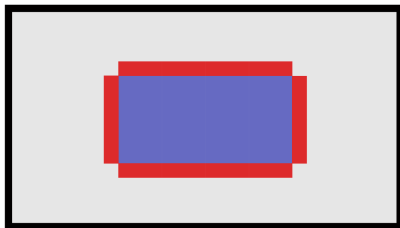
- Hybrid programming allows optimizing code adequacy to the target architecture (generally composed of shared-memory nodes [SMP] linked by an interconnection network). The advantage of shared memory inside a node is that it is not necessary to duplicate data in order to exchange them. Every thread can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



8 cores SMP node, hybrid domain decomposition

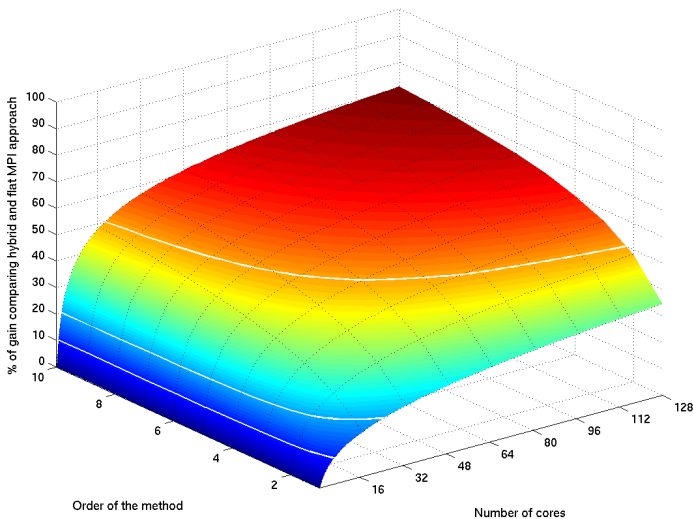


-  Intra-node ghost cells
-  Extra-node ghost cells
-  Sub-domain associated with one MPI process

Extrapolation on a 3D domain

- 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



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):

Code	Pure MPI version		Hybrid version		Memory savings
	MPI prc	Mem./ Node	MPI x threads	Mem./Node	
CPMD	1152	2400	48 x 24	500	4.8
BQCD	3072	3500	128 x 24	1500	2.3
SP-MZ	4608	2800	192 x 24	1200	2.3
IRS	2592	2600	108 x 24	900	2.9
Jacobi	2304	3850	96 x 24	2100	1.8

Memory savings on some real application codes (2)

- Source: « Performance evaluations of gyrokinetic Eulerian code GT5D on massively parallel multi-core platforms », Y. Idomura and S. Jolliet, SC11
- 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

System	Pure MPI	4 threads/process		8 threads/process	
	Total (code+sys.)	Total (code+sys.)	Gain	Total (code+sys.)	Gain
BX900	5.40 (3.40+2.00)	2.83 (2.39+0.44)	1.9	2.32 (2.16+0.16)	2.3

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 general use of hyperthreading or SMT (the possibility of running multiple threads simultaneously on one core).
 - 3 General use of high-order numerical methods (gross computing costs becoming less, 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

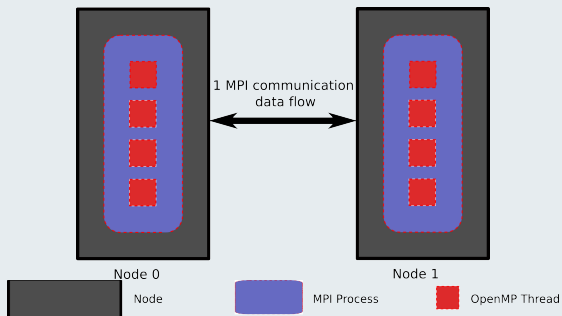
- The hybrid approach aims to use the available hardware resources the most effectively (shared memory, memory hierarchy, communication network).
- One of the difficulties of hybrid programming is to generate a sufficient number of communication flows in order to make the best use of the inter-node communication network.
- In fact, the throughputs of inter-node interconnection networks of recent architectures are high (bidirectional throughput peak of 8 GB/s on Vargas, 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

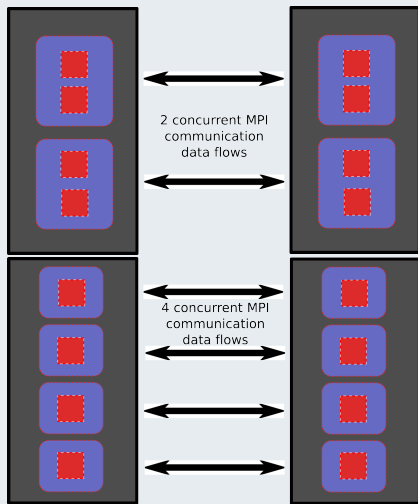
MPI_THREAD_FUNNELED approach:

- 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



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

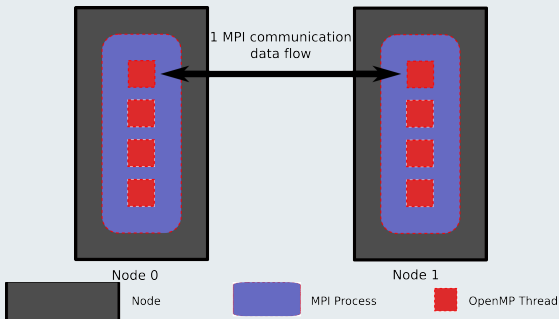


MPI_THREAD_MULTIPLE version of SBPR

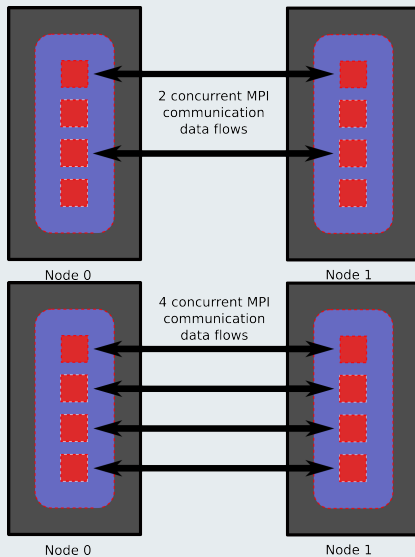
MPI_THREAD_MULTIPLE approach:

- 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

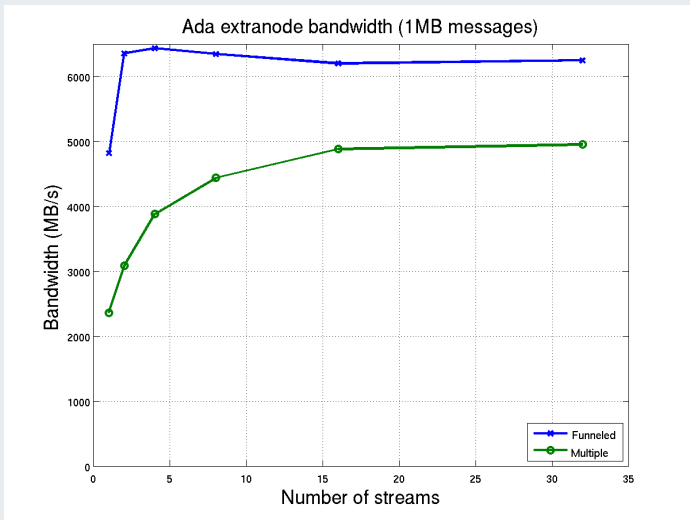


MPI_THREAD_MULTIPLE version of SBPR: Example on a 4-Core (BG/P) SMP Node



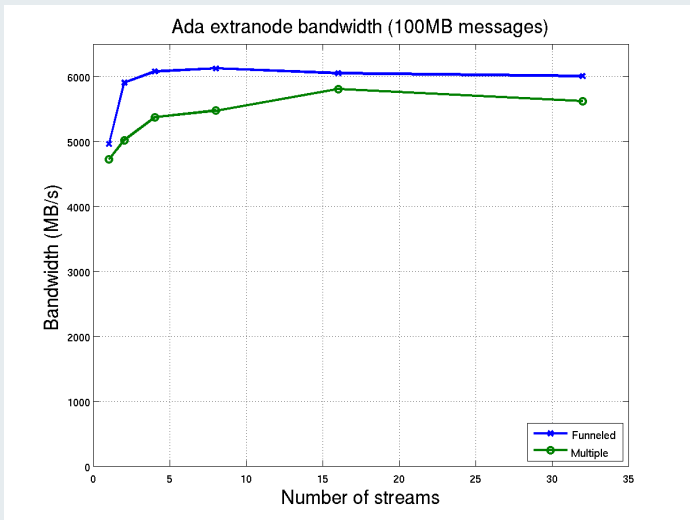
SBPR: Results on Ada

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



SBPR: Results on Ada

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

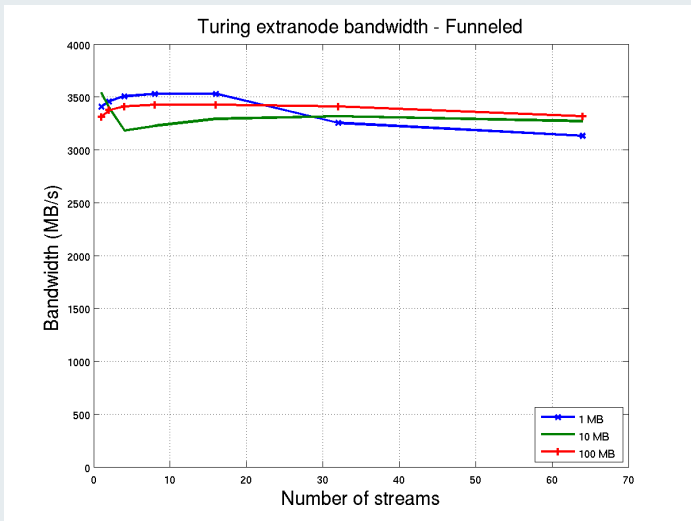


SBPR: Results on Ada

- 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.

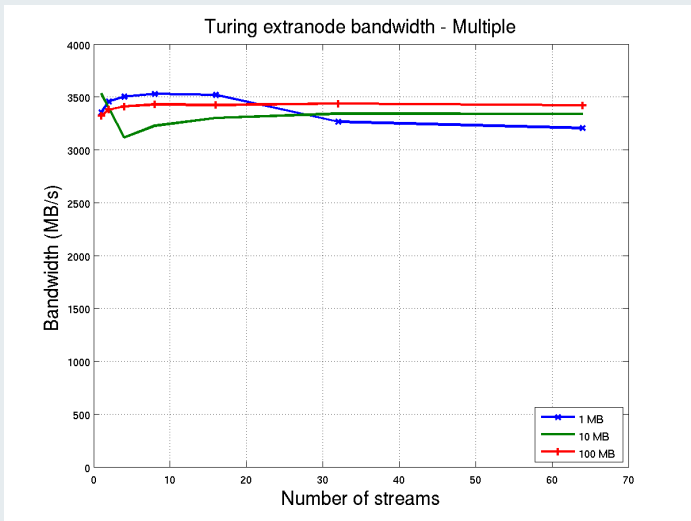
SBPR: Results on Turing

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.



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

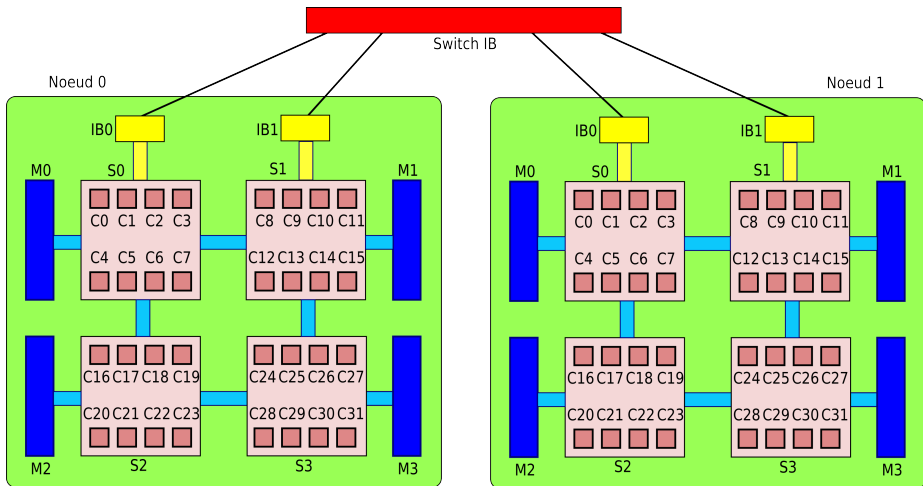
Most modern supercomputers have a non-uniform architecture :

- 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.

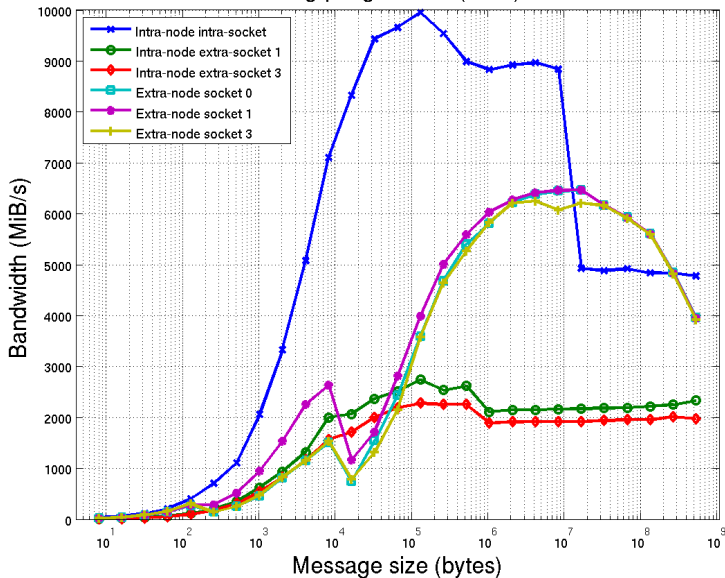
Effects

- 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



Ping-pong on Ada (POE)



Description of the Multi-Zone NAS Parallel Benchmark

- 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:
<http://www.nas.nasa.gov/Resources/Software/software.html>.

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

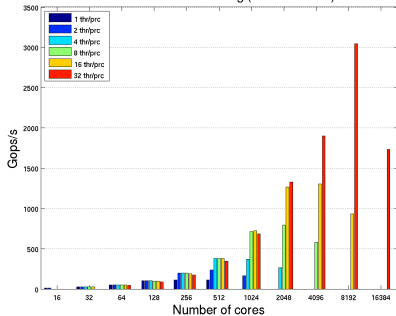
SP-MZ: scalar pentadiagonal solver.

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

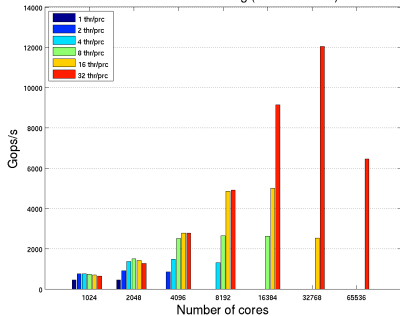
Selected Problem Sizes

- 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)

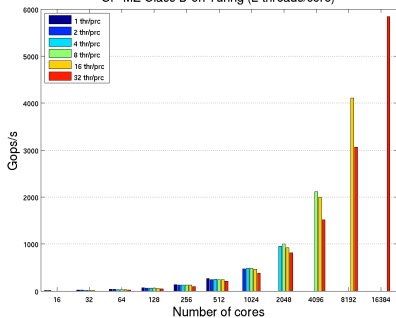
BT-MZ Class D on Turing (2 threads/core)



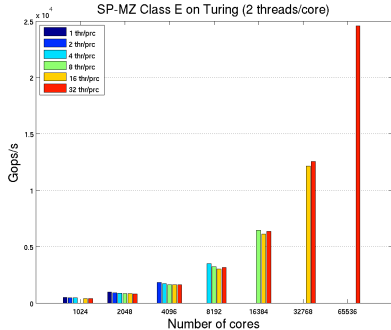
BT-MZ Class E on Turing (2 threads/core)



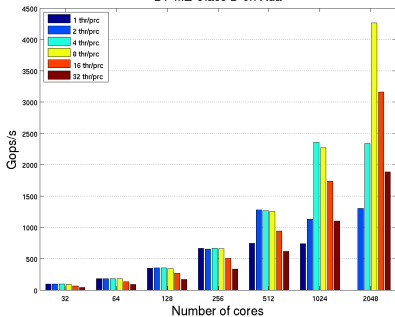
SP-MZ Class D on Turing (2 threads/core)



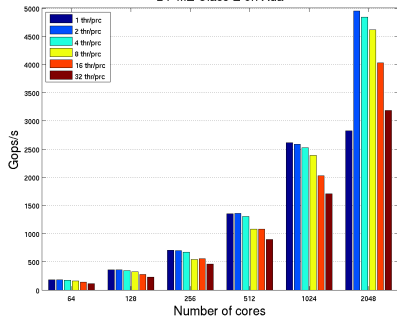
SP-MZ Class E on Turing (2 threads/core)



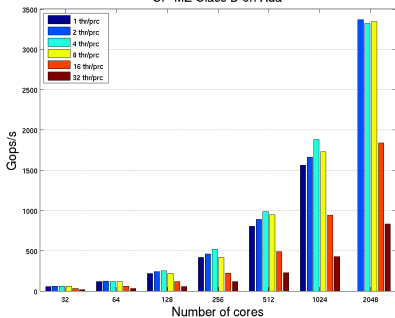
BT-MZ Class D on Ada



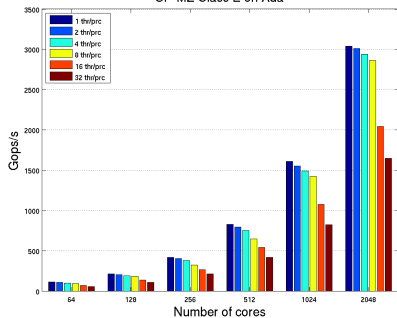
BT-MZ Class E on Ada



SP-MZ Class D on Ada



SP-MZ Class E on Ada



Analysis of Results: BT-MZ

- 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 because 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 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.

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

Solver

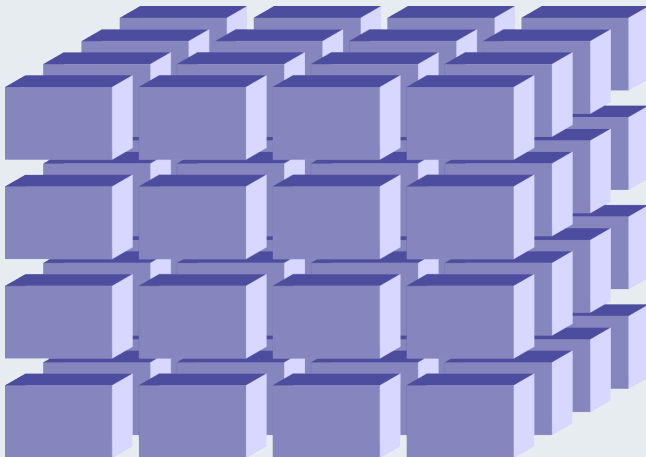
The discretization is made on a regular grid in the three spatial directions (step $h = h_x = h_y = h_z$).

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.



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 a 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

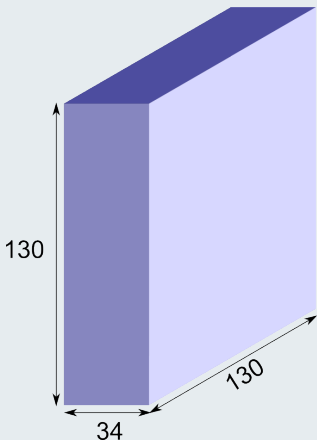
Version	Topology	Time (s)	L1 read (TiB)	DDR read (TiB)	Torus send (GiB)
MPI with overlap	16x4x4	52.741	11.501	14.607	112.873
MPI with overlap	4x16x4	39.039	11.413	7.823	112.873
MPI with overlap	4x4x16	36.752	11.126	7.639	37.734

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

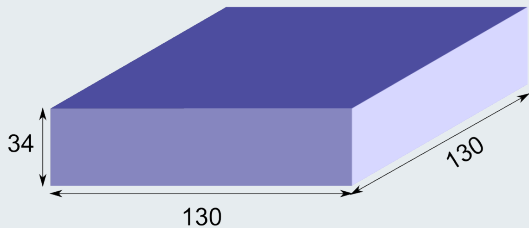
- The way the Cartesian topology is split has a major effect.
- The phenomenon appears to be due to cache effects. In 512^3 , The *u* and *u_new* arrays require 8 MiB/core.
- Depending on the topology, the accesses to the central memory are very different (between 7.6 TiB and 18.8 TiB in *read*). The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

Sub-domain form (512^3)

16x4x4



4x16x4



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 $16 \times 4 \times 4$ topology into (34, 130, 130) and in the $4 \times 16 \times 4$ 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 $4 \times 16 \times 4$ than in $16 \times 4 \times 4$; 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.

Topology	Plans	Time (s)
4x16x4	1	39.143
4x16x4	16	35.614

Topology	Plans	Time (s)
16x4x4	1	52.777
16x4x4	16	41.559

Cache effects on the derived datatypes: analysis

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):

	MPI std	MPI no deriv	MPI+OMP std	MPI+OMP no deriv
64 cores	84.837s	84.390s	102.196s	88.527s
256 cores	27.657s	26.729s	25.977s	22.277s
512 cores	16.342s	14.913s	16.238s	13.193s

Improvements also appear in the pure MPI version.

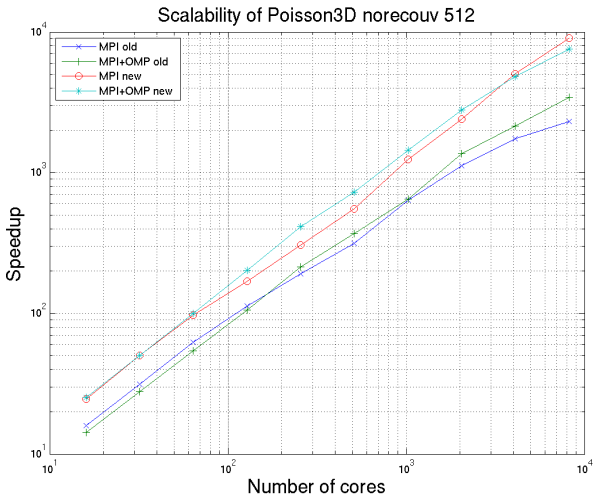
MPI communications

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).

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

Comparison: Optimized versus original versions (without overlap)



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.

Presentation of the HYDRO Code (1)

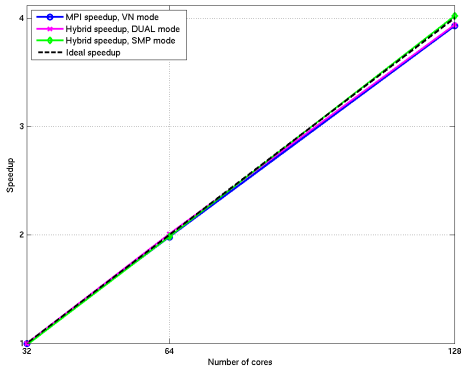
- This is the code used for the hands-on exercises of the hybrid course.
- Hydrodynamics code, 2D-Cartesian grid, finite volume method, resolution of a Riemann problem on the interfaces with a Godunov method.
- For the last few years, in the framework of the IDRIS technology watch, this code has served 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.

Presentation of the HYDRO Code (2)

- Today, there are the following hydro versions:
 - Original version, F90 monprocessor (P.-Fr. Lavallée, R. Teyssier)
 - Monprocessor 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, PGI accelerator, CUDA Fortran, ...

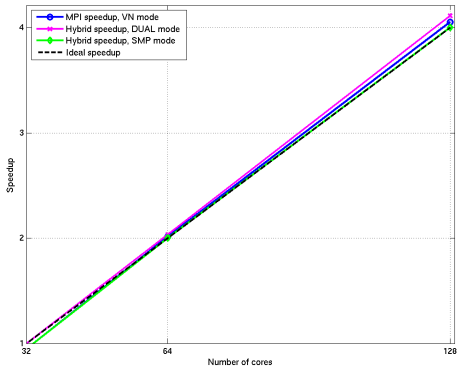
Results for the $n_x = 100000$, $n_y = 1000$ domain

Times (s)	32 cores	64 cores	128 cores
VN mode	49.12	24.74	12.47
DUAL mode	49.00	24.39	12.44
SMP mode	49.80	24.70	12.19



Results for the $n_x = 10000$, $n_y = 10000$ domain

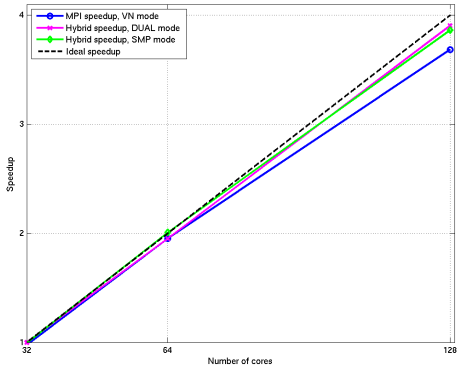
Time in (s)	32 cores	64 cores	128 cores
VN mode	53.14	24.94	12.40
DUAL mode	50.28	24.70	12.22
SMP mode	52.94	25.12	12.56



Results using 128 cores on Babel

Results for the $n_x = 1000$, $n_y = 100000$ domain

Time (s)	32 cores	64 cores	128 cores
VN mode	60.94	30.40	16.11
DUAL mode	59.34	30.40	15.20
SMP mode	59.71	29.58	15.36



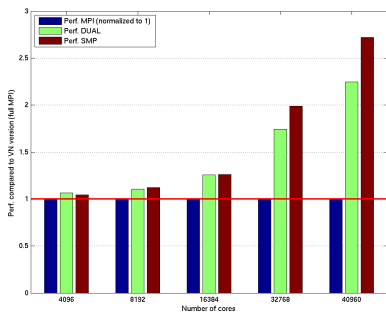
Characteristics of the domains used for weak scaling

- On 4096 cores, total number of points of the domain: $16 \cdot 10^8$
 - 400000x4000: domain elongated in the first dimension
 - 40000x40000: square domain
 - 4000x400000: domain elongated in the second dimension
- On 8192 cores, total number of domain points: $32 \cdot 10^8$
 - 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: $64 \cdot 10^8$
 - 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: $128 \cdot 10^8$
 - 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: $16 \cdot 10^9$
 - 4000000x4000: domain elongated in the first dimension
 - 126491x126491: square domain
 - 4000x4000000: domain elongated in the second dimension

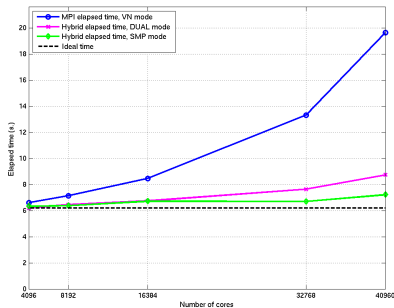
Results using 10 racks on Babel - Weak Scaling

Results for the domain elongated in the first dimension

Time (s)	4096 cores	8192 cores	16384 cores	32768 cores	40960 c.
VN mode	6.62	7.15	8.47	13.89	19.64
DUAL mode	6.21	6.46	6.75	7.85	8.75
SMP mode	6.33	6.38	6.72	7.00	7.22



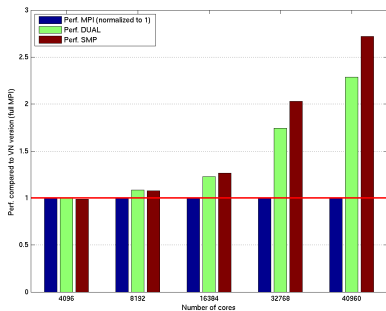
Performances compared to the MPI version



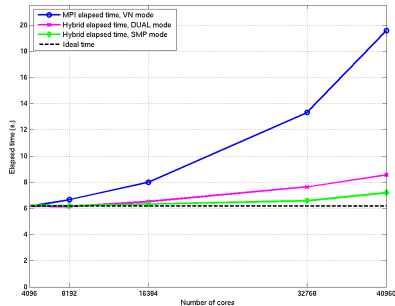
Elapsed execution time

Results for the square domain

Time (s)	4096 cores	8192 cores	16384 cores	32768 cores	40960 c.
VN mode	6.17	6.67	8.00	13.32	19.57
DUAL mode	6.17	6.14	6.52	7.64	8.56
SMP mode	6.24	6.19	6.33	6.57	7.19



Performances compared to the MPI version

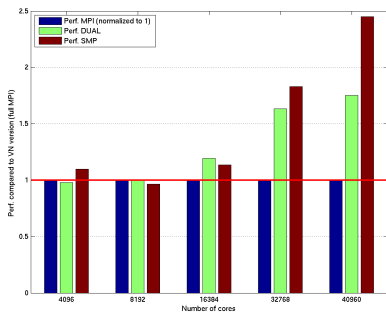


Elapsed execution time

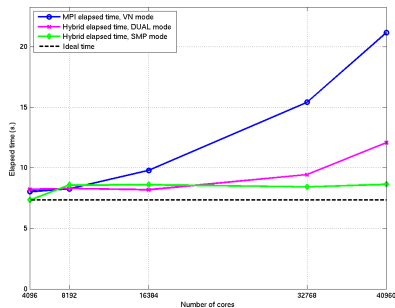
Results using 10 racks on Babel - Weak Scaling

Results for the domain elongated in the second dimension

Time (s)	4096 cores	8192 cores	16384 cores	32768 cores	40960 c.
VN mode	8.04	8.28	9.79	15.42	21.17
DUAL mode	8.22	8.30	8.20	9.44	12.08
SMP mode	7.33	8.58	8.61	8.43	8.64



Performances compared to the MPI version



Elapsed execution time

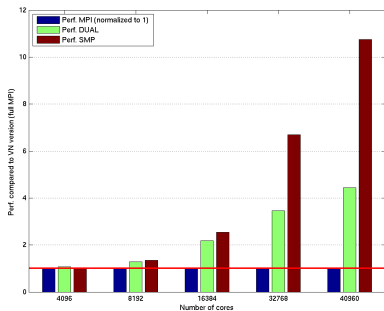
Interpretation of results

- 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. No sooner does it scale to 16384 cores when 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 recourse to hybrid parallelization. It is not enough to use MPI alone !

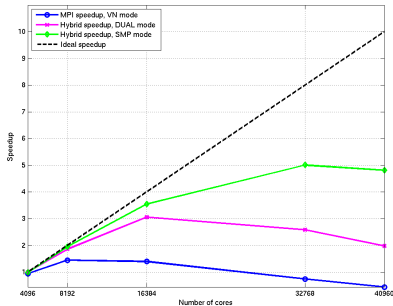
Results using 10 racks on Babel - Strong Scaling

Results for the $n_x = 40000$, $n_y = 4000$ domain

Time(s)	4096 cores	8192 cores	16384 cores	32768 cores	40960 c.
VN Mode	6.62	4.29	4.44	8.30	13.87
DUAL Mode	6.21	3.34	2.03	2.40	3.13
SMP Mode	6.33	3.18	1.75	1.24	1.29



Performances compared to the MPI version

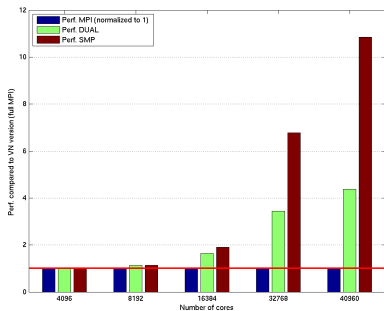


Scalability up to 40960 cores

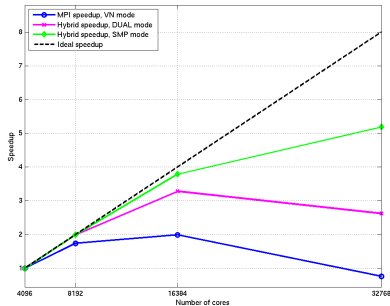
Results using 10 racks on Babel - Strong Scaling

Results for the $n_x = 40000$, $n_y = 40000$ Domain

Time (s)	4096 cores	8192 cores	16384 cores	32768 cores	40960 c.
VN Mode	6.17	3.54	3.10	8.07	13.67
DUAL Mode	6.17	3.10	1.88	2.35	3.12
SMP Mode	6.24	3.10	1.63	1.20	1.26



Performances compared to the MPI version

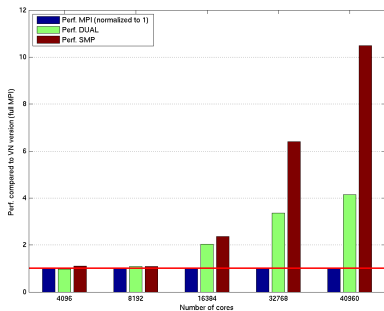


Scalability up to 40960 cores

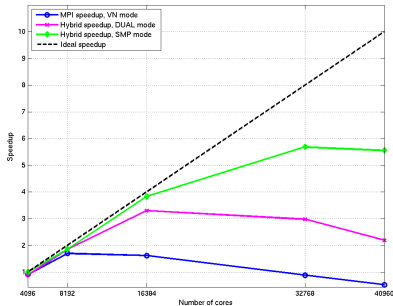
Results using 10 racks on Babel - Strong Scaling

Results for the $n_x = 4000$, $n_y = 400000$ Domain

Time (s)	4096 cores	8192 cores	16384 cores	32768 cores	40960 c.
VN Mode	8.04	4.31	4.52	8.26	13.85
DUAL Mode	8.22	3.96	2.22	2.46	3.34
SMP Mode	7.33	3.94	1.91	1.29	1.32



Performances compared to the MPI version



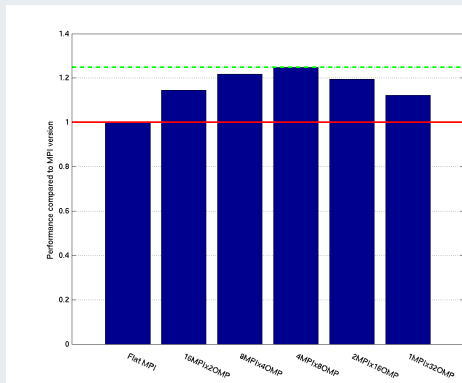
Scalability up to 40960 cores

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 no more than scales up to 8192 cores when 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 for the $n_x = 100000$, $n_y = 1000$ domain

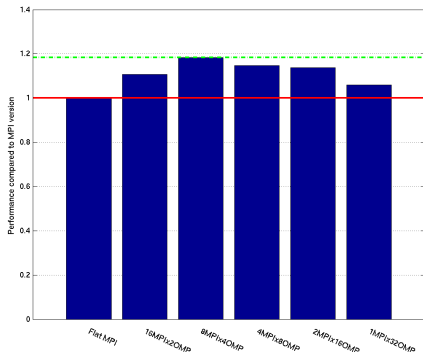
MPI x OMP per node	Time (s)	
	Mono	64 cores
32 x 1	361.4	7.00
16 x 2	361.4	6.11
8 x 4	361.4	5.75
4 x 8	361.4	5.61
2 x 16	361.4	5.86
1x 32	361.4	6.24



- 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 for the $n_x = 10000$, $n_y = 10000$ Domain

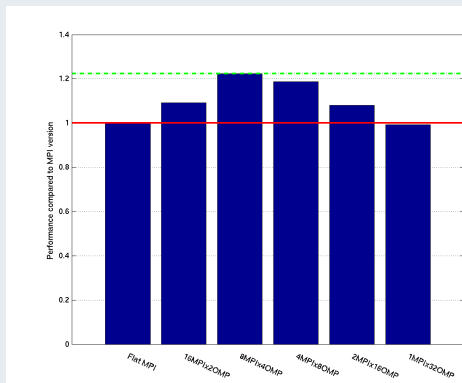
MPI x OMP per node	Time(s)	
	Mono	64 cores
32 x 1	449.9	6.68
16 x 2	449.9	6.03
8 x 4	449.9	5.64
4 x 8	449.9	5.82
2 x 16	449.9	5.87
1 x 32	449.9	6.31



- 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 $n_x = 1000$, $n_y = 100000$ domain

MPI x OMP per node	Time (s)	
	Mono	64 cores
32 x 1	1347.2	8.47
16 x 2	1347.2	7.75
8 x 4	1347.2	6.92
4 x 8	1347.2	7.13
2 x 16	1347.2	7.84
1 x 32	1347.2	8.53



- 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.

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

6

Tools

- SCALASCA
- TAU
- TotalView

Description

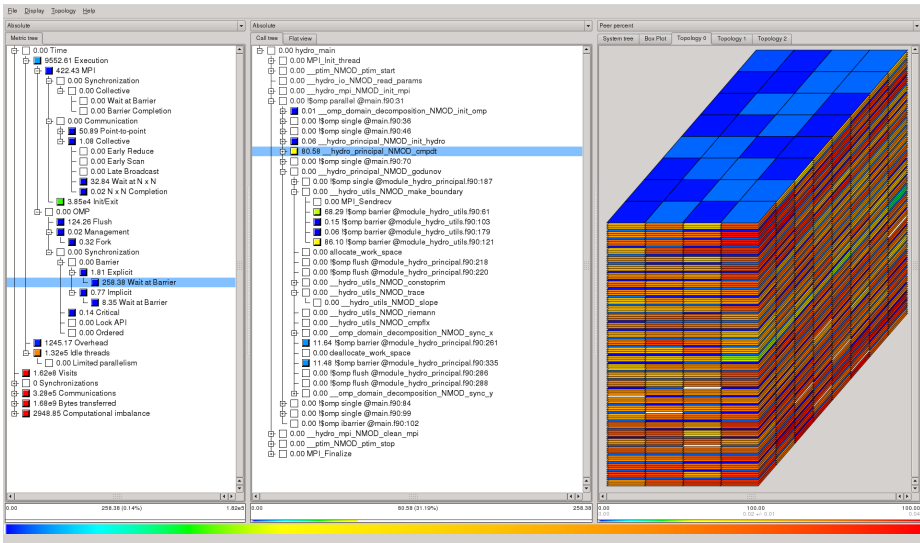
SCALASCA is a graphical tool for performance analysis of parallel applications.

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

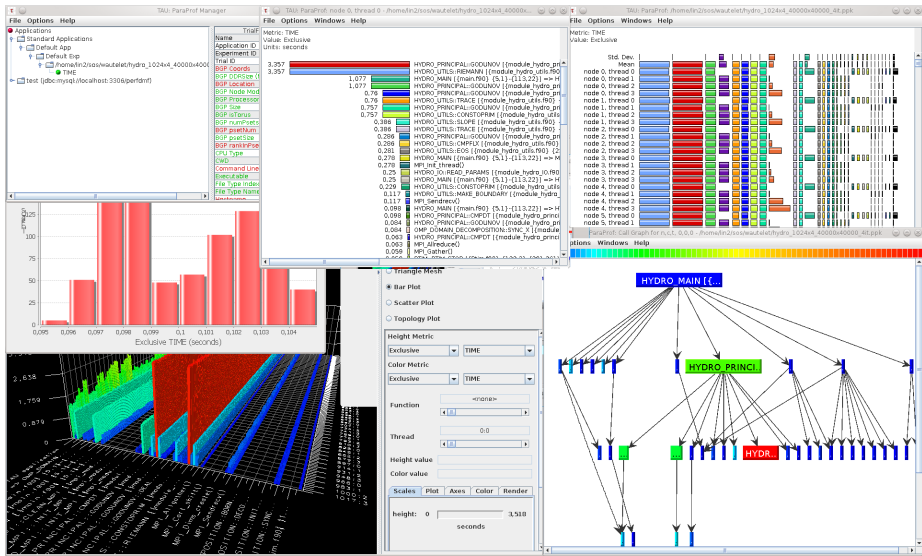
- Compile your application with *skin f90* (or other compiler).
- Execute with *scan mpirun*. Use the option *-t* for the trace mode.
- Visualize the results with *square*.



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)



Description

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.

TotalView 8.81.0 - @vargas013

ID	Rank	Host	Status	Description
2.1	0	lscabo	04	in hydro_util wake_boundary
3.1	1	lscabo	04	in hydro_util wake_boundary
4.1	2	lscabo	04	in hydro_util wake_boundary
5.1	3	lscabo	04	in hydro_util wake_boundary
6.1	4	lscabo	04	in hydro_util wake_boundary
7.1	5	lscabo	04	in hydro_util wake_boundary
8.1	6	lscabo	04	in hydro_util wake_boundary
9.1	7	lscabo	04	in hydro_util wake_boundary
2.2	0	lscabo	K	in kernel
2.5	0	lscabo	K	in kernel
2.6	0	lscabo	K	in kernel
2.7	0	lscabo	K	in kernel
2.9	0	lscabo	K	in kernel
2.10	0	lscabo	K	in kernel
2.11	0	lscabo	K	in kernel

pos-hydro_std - @vargas013

Group (Control)

Rank 2: pos-hydro_std-2 (At Breakpoint 4)
Thread 1 (1): hydro_std (At Breakpoint 4)

Stack Trace

```

000 hydro_util wake_boundary FP=ffffffffff
001 hydro_principal godswar FP=ffffffffff
002 hydro_main hydro_main@l1 FP=ffffffffff
003 _lscabo@hydro@lscabo_pos FP=ffffffffff
004 hydro_main FP=ffffffffff290
005 _start FP=ffffffffff300
    
```

Function "hydro_util wake_boundary":
Local variables:
idw: 0 (0x00000000)
sign: 56842455 (0x00000000)
code: 32642455 (0x00000000)
i: 1 (0x00000001)
sign: 5 (0x00000005)
code: 51 (0x00000033)
block_size: 52 (0x00000034)

Expression Monitor

Expression: `countor` Address: `0x11000200`

Type: `int`

Process: `pos-hydro_std-0` Value: `0.00226643191323905`

Message Queue Graph - 4.1 [hydro_std] - @varg...

Call Graph - pos-Control Group - @vargas013

Action Points

Action Points	Processes	Threads
1 module_hydro_principal.F90153	hydro_principal	cap0t@lscabo
2 module_hydro_principal.F90152	hydro_principal	cap0t@lscabo
4 module_hydro_util.F9049	hydro_util	wake_boundary@lscabo
5 module_hydro_util.F9056	hydro_util	wake_boundary@lscabo
6 module_hydro_util.F9073	hydro_util	wake_boundary@lscabo

Appendices

7 Appendices

- MPI
 - Factors Affecting MPI Performance
 - Ready Sends
 - Persistent Communications
- Introduction to Code Optimisation
- SBPR on older architectures

Machine nodes

- Cores (frequency, number per node, ...)
- Memory (throughputs, latencies, caches, number of channels, sharing between the different cores, ...)
- Network cards (throughputs, latencies, type, node connection, number of links, ...)
- Availability of an RDMA engine for the communications
- OS (light kernel, ...)
- Configuration/tuning of the machine

Network

- Card type (proprietary, InfiniBand, Myrinet, Ethernet, ...)
- Network topology (star, torus 3D or more, fat tree, hypercube, ...)
- Protocol (low level, TCP/IP, ...)
- Contention (between processes of the same job or different jobs)
- Configuration/tuning of network

Application

- Algorithms, memory access, ...
- Computations/communications, granularity
- Data-partitioning method
- Load balancing
- Process mapping
- Process-core binding
- Input/output
- Message size
- Types of communications, use of communicators, ...

MPI implementation

- Adaptation to the machine architecture (manufacturer's implementation, open source, ...)
- Use of buffers
- Communication protocols (Eager, Rendezvous, ...)
- Influence of environment variables
- Implementation quality and algorithms used

Ready sends

A Ready send is made by calling the subroutine `MPI_Rsend` or `MPI_Irsend`.

Attention: It is obligatory that these calls be made only when the receive has already been posted.

The use of Ready send is strongly advised against.

Advantages

- Slightly more efficient than the synchronous mode because the synchronization protocol can be simplified.

Disadvantages

- Errors if the receiver is not ready during the send.

Characteristics

- Persistent communications permits communication repetitions an unlimited number of times in the same memory space, but using different values.
- The benefit is to avoid re-initializing these communications (and the associated data structures) at each call (theoretically, less additional costs).
- A communication channel is thereby created for the sending of messages.
- These are non-blocking point-to-point communications.
- MPI Persistent communications do not add any significant benefit to the current machines. The performances are generally very close to the performances of standard non-blocking point-to-point communications.

The use of persistent communications is not recommended because they don't offer any particular advantages at this time.

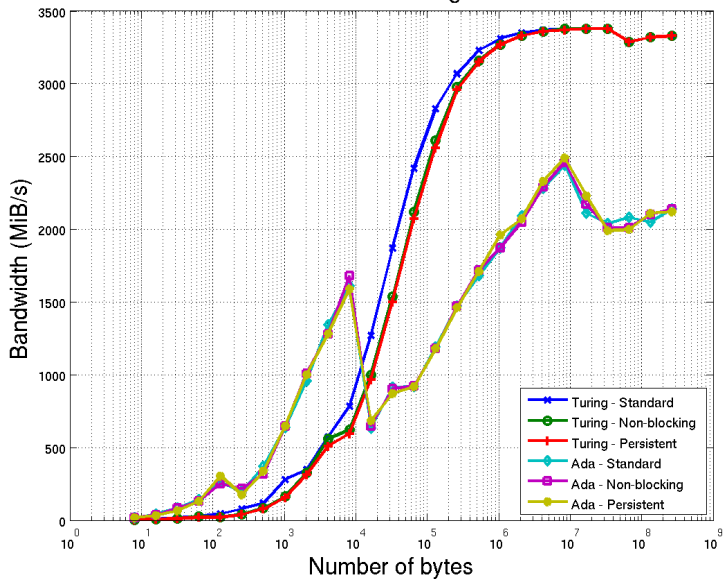
Usage

- Initialization of persistent communications with `MPI_Send_init/MPI_Recv_init` (or variants).
- Repetition of the communication sequence – for example, in a loop:
 - Beginning of the communication with a call to `MPI_Start`
 - End of the communication (non-blocking) with `MPI_Wait`
- Freeing of resources with `MPI_Request_free`.

Example

```
call MPI_Send_init (data, sz, MPI_REAL, dest, tag, comm, req, ierr)
do i=1, niter
  call MPI_Start (req, ierr)
  call kernel ()
  call MPI_Wait (req, MPI_STATUS_IGNORE, ierr)
end do
call MPI_Request_free (req, ierr)
```

Persistent communications Turing and Ada extranode



Definition

- Optimising a code consists of reducing its resource needs.
- The resource needs are diverse but we are generally referring to elapsed time.
- Memory consumption and disk-space usage also fall into this category.

Why optimise?

Optimizing an application can bring a number of advantages:

- Obtain results faster through a reduction in elapsed time.
- Possibility of obtaining more results during your attributed hours.
- Possibility of carrying out larger computations.
- Better understanding of the code, the machine architecture and their interactions.
- Competitive advantage in comparison to other teams;
- Detection and correction of bugs through the re-reading of code sources.

Improved application performance also permits:

- Reduction in computing energy consumption.
- Making better use of the machine (the cost of purchasing, maintaining, and using a supercomputer is not negligible). At IDRIS, each attributed hour represents an expense for the whole scientific community.
- Liberating resources for other research groups.

Why to NOT optimise?

- Insufficient resources or means (lack of staff, time, skills, ...)
- Decrease in code portability (Many optimizations are specific to the machine architecture.)
- Risk of performance losses on other machines
- Decreased source readability and more difficult maintenance
- Risk of unintentionally introducing bugs
- Limited life span for an optimised code
- Sufficient code speed already (It is useless to optimise a code which already provides results in an acceptable time.)
- Already optimised code

When to optimise?

- An application should not be optimised before it is working correctly for the following reasons:
 - Risk of unintentionally introducing new bugs.
 - Decrease in readability and code understanding.
 - Risk of optimising procedures which could be abandoned; used very little (if at all); or totally rewritten.
- Do not launch into optimisation unless the application is (a) too slow, or (b) does not allow making large computations in an acceptable period of time.
- If you are fully satisfied with the performances of your application, the investment may not be necessary.
- You must have enough available time ahead of you.

How to optimise?

- First and foremost, make a sequential and parallel profiling with a realistic test set in order to identify the critical zones.
- Optimize in the places where the resources are most highly consumed.
- Verify every optimisation: Are the results always correct? Have the performances really improved?
- Question: If there is little improvement, should you keep the optimisation?

What to optimise?

- Sequential performances
- MPI communications, OpenMP performances, and scalability
- The I/O

Sequential optimisation

- Algorithms and conception
- Libraries
- Compiler
- Caches
- Specialized processing units (SIMD, SSE, AVX, QPX...)
- Other

Optimization of MPI communications, OpenMP and scalability

- Algorithms and conception
- Load balancing
- Computation-communication overlap
- Process mapping
- Hybrid programming
- Other

Optimization of I/O

- Only read and write what is necessary
- Reduce the precision (simple precision instead of double)
- Parallel I/O
- Libraries (MPI-I/O, HDF5, NetCDF, Parallel-NetCDF...)
- MPI-I/O hints
- Other

MPI_THREAD_FUNNELED version of SBPR: Results on Vargas

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

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

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

MPI x OMP par node	Total throughput (MB/s) Message of 1 MB	Total throughput (MB/s) Message of 10 MB	Total throughput (MB/s) Message of 100 MB
SMP (1 x 4)	373.5	374.8	375.0
DUAL (2 x 2)	374.1	374.9	375.0
VN (4 x 1)	374.7	375.0	375.0

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.

MPI_THREAD_MULTIPLE version of SBPR: Results on Vargas

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

MPI x OMP per node	Total throughput (MB/s) Message of 1 MB	Total throughput (MB/s) Message of 10 MB	Total thrpt (MB/s) Message of 100 MB
1 x 32 (1 flow)	548.1	968.1	967.4
1 x 32 (2 flows)	818.6	1125	1016
1 x 32 (4 flows)	938.6	1114	1031
1 x 32 (8 flows)	964.4	1149	1103
1 x 32 (16 flows)	745.1	1040	1004
1 x 32 (32 flows)	362.2	825.1	919.9

Interpretations

- 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.

MPI_THREAD_MULTIPLE version of SBPR: Results on Babel

Peak throughput 425 Mo/s

MPI x OMP per node	Total throughput (MB/s) Message of 1 MB	Total throughput (MB/s) Message of 10 MB	Total throughput (MB/s) Message of 100 MB
SMP (1 flow)	372.9	374.7	375.0
SMP (2 flows)	373.7	374.8	375.0
SMP (4 flows)	374.3	374.9	375.0

Interpretations

- 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.

Hands-on Exercises

Objective

Parallelize an application using MPI.

Statement

You are asked to start with the HYDRO sequential version application. It should be parallelized by using MPI.

- 1 Firstly, the parallelization can be done in only one direction (north-south in Fortran to avoid using derived datatypes).
- 2 Secondly, you are asked to do a parallelized version in both spatial directions (north-south and east-west).

Some Tips

- The grid cells used to impose the boundary conditions in the sequential version can serve as ghost cells for the communications between processes.
- A boundary between the domains can be seen as a particular boundary condition.
- The use of a Cartesian topology type communicator is highly recommended, especially for the 2D decomposition.

Objective

Parallelize the following computation kernel using OpenMP.

```
! Dual-dependency nested loops
do j = 2, ny
  do i = 2, nx
    V(i, j) = (V(i, j) + V(i-1, j) + V(i, j-1)) / 3
  end do
end do
```

Statement

You are asked to start with a computation kernel in the sequential version. It should be parallelized by using OpenMP.

- 1 Using the pipeline method.
- 2 Optional: Using the hyperplane method.
- 3 Make a scalability curve of your parallel version(s).

Objective

Parallelize an application using OpenMP.

Statement

You are asked to begin with the HYDRO sequential version application to construct a Fine-Grain OpenMP parallel version, but with only one parallel region.

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

Parallelize an application using MPI and OpenMP.

Statement

- 1 Integrate the changes added to HYDRO in the previous hands-on exercises to obtain a hybrid version of the code.
- 2 Compare the performances obtained with the different versions. Is the scalability good?
- 3 What improvements can be added to obtain better performances? Make tests and compare.