


Hybrid MPI-OpenMP Programming

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

CNRS — IDRIS / LA

Version 3.0.1 — 1 December 2017

Availability and Updating

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

Table of Contents I

Table of Contents II

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 indispensible if we are to take advantage of the power of the latest generation of massively parallel machines.
- However, a hybrid code cannot perform well if the MPI and OpenMP parallel implementations have not been previously optimized.
- The Hybrid programming section is entirely dedicated to the MPI+OpenMP hybrid approach. The benefits of hybrid programming are numerous:
  - Memory savings
  - Improved performances
  - Better load balancing
  - Coarser granularity, resulting in improved scalability
  - Better code adequacy to the target architecture hardware specificities

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

Introduction

Moore's Law

Statement

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

Electric consumption

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

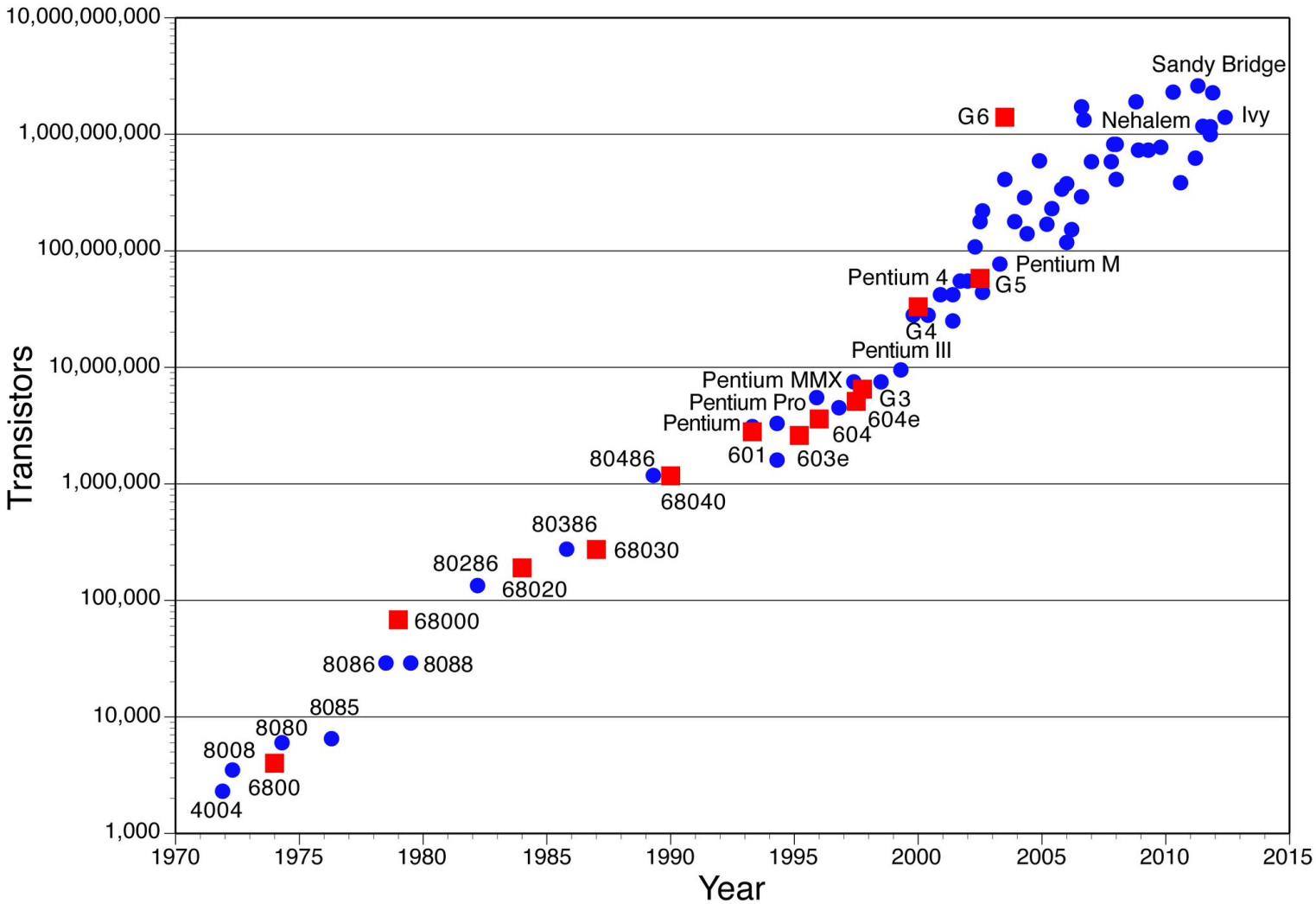
Moore's law and electric consumption

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

=> Number of cores per chip is increasing: The Intel Skylake chips have up to 28 cores each and can run 56 threads simultaneously, the AMD EPYC chips have up to 32 cores each and can run 64 threads simultaneously.

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

Moore's Law



http://en.wikipedia.org/wiki/Moore%27s_law

The Memory Wall

Causes

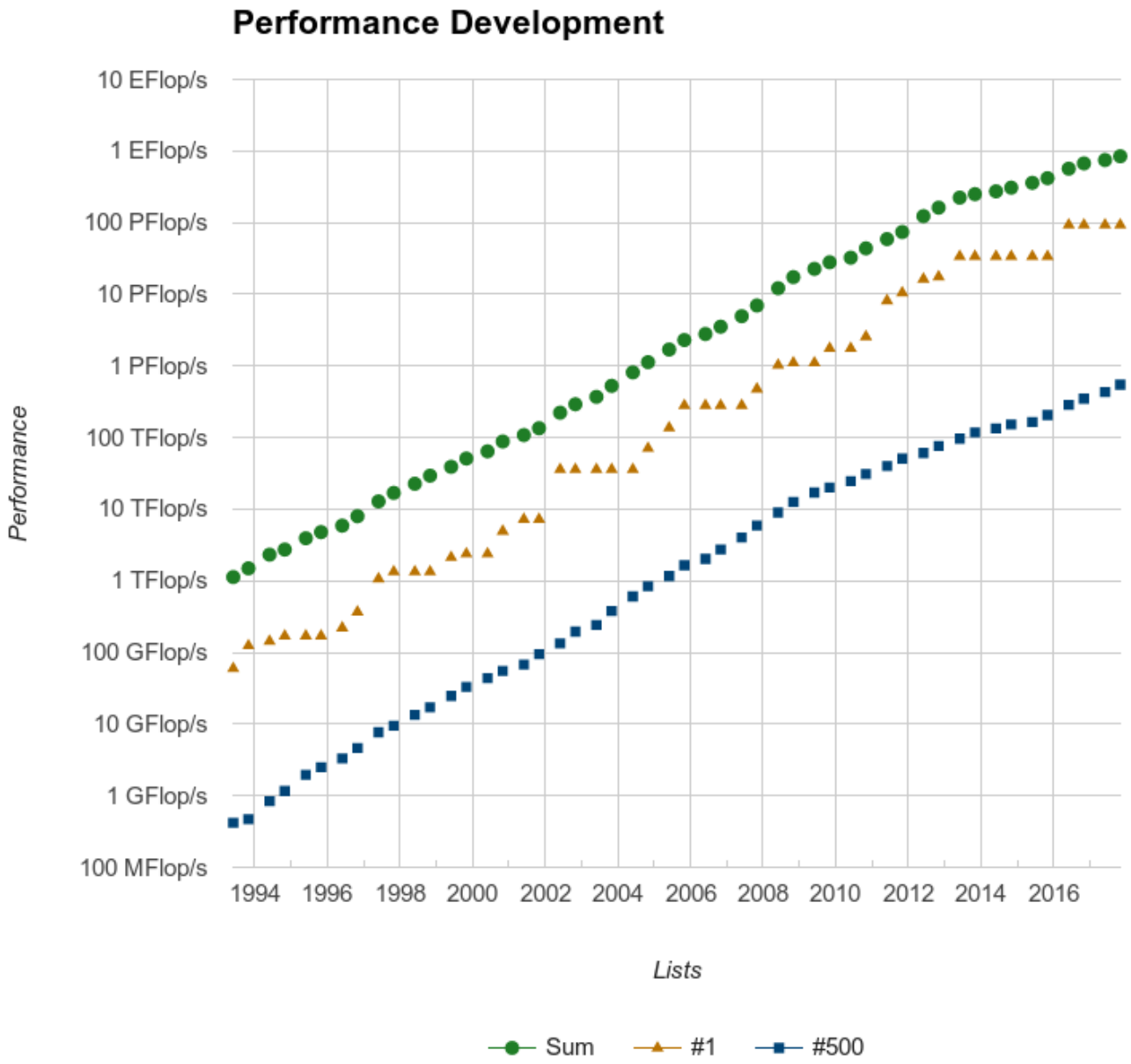
- Throughputs towards the memory are not increasing as quickly as processor computing power.
- Latencies (access times) of the memory are decreasing very slowly.
- Number of cores per memory module is increasing.

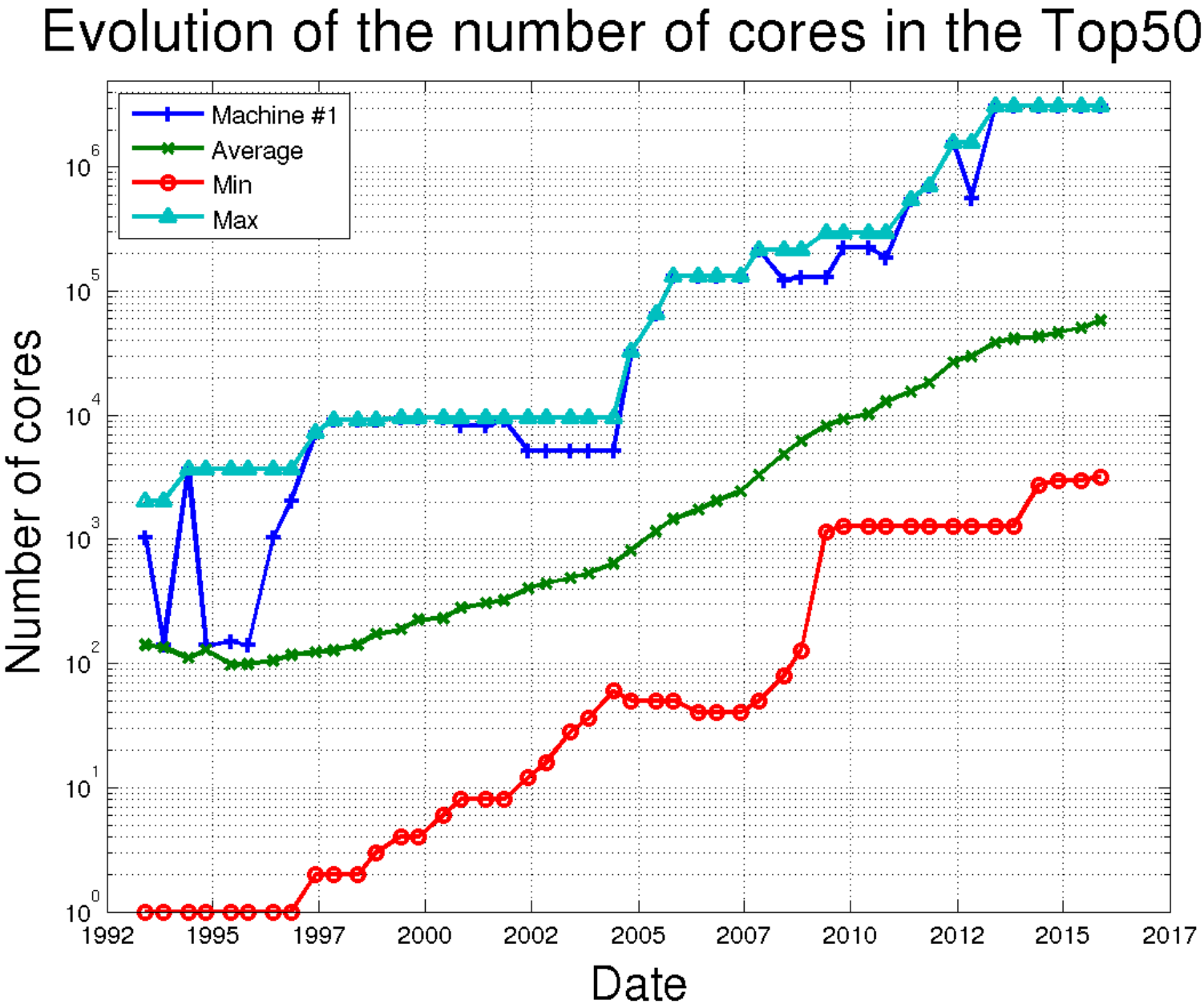
Consequences

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

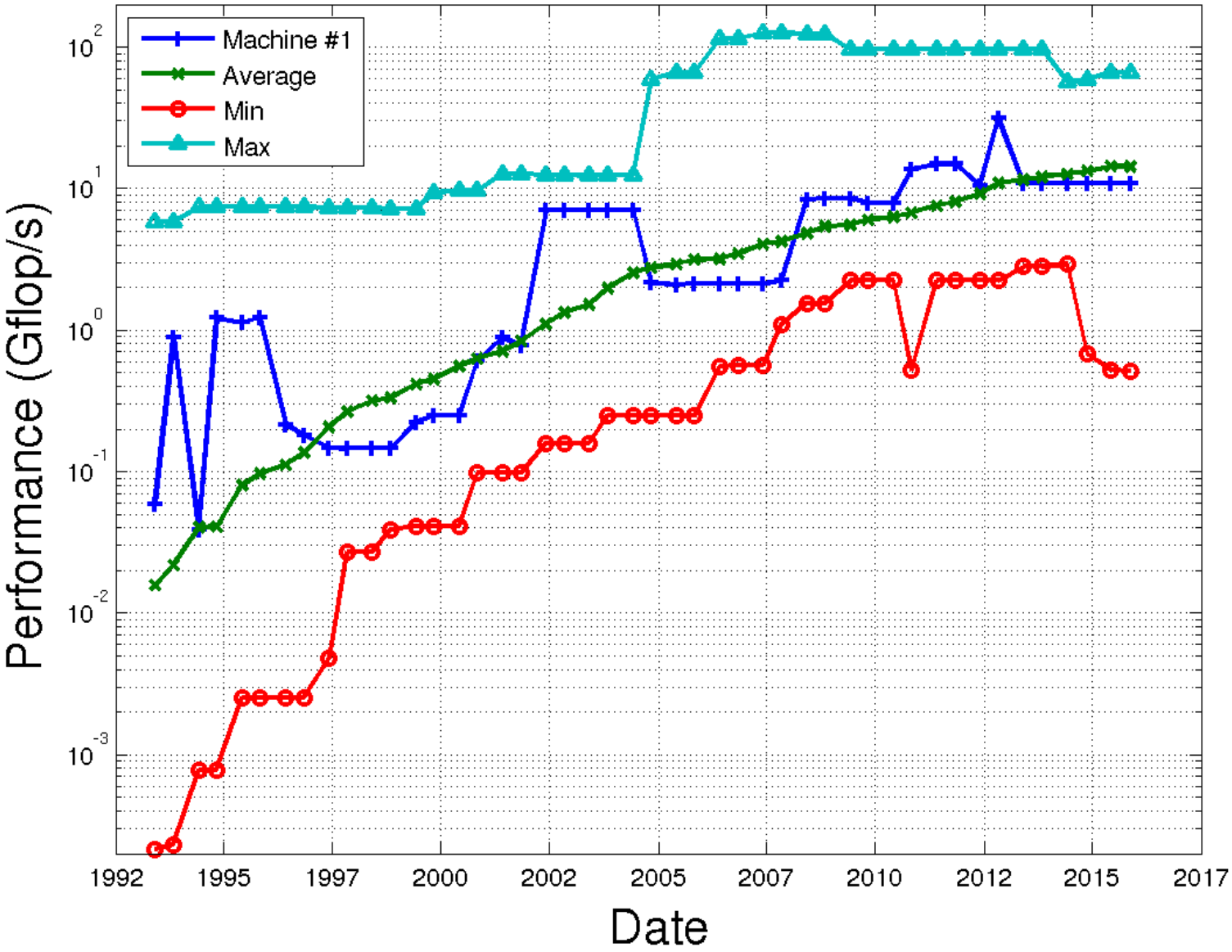
Partial solutions

- Addition of cache memories is essential.
- Access parallelization via several memory banks as found on the vector architectures (Intel Skylake: 6 channels, AMD EPYC: 8 channels, ARM ThunderX2: 8 channels).
- If the clock frequency of the cores stagnates or falls, the gap could be reduced.


Performance Development
10 EFlop/s
1 EFlop/s
100 PFlop/s
10 PFlop/s
1 PFlop/s
100 TFlop/s
10 TFlop/s
1 TFlop/s
100 GFlop/s
10 GFlop/s
1 GFlop/s
100 MFlop/s
Performance
1994
1996
1998
2000
2002
2004
2006
2008
2010
2012
2014
2016
Lists
Sum
#1
#500


Evolution of the number of cores in the Top500
Machine #1
Average
Min
Max
Number of cores
10^0
10^1
10^2
10^3
10^4
10^5
10^6
1992
1995
1997
2000
2002
2005
2007
2010
2012
2015
2017
Date

Evolution of the performance per core in the Top500



Technical evolution

- The computing power of supercomputers is doubling every year (faster than Moore's Law, but electrical consumption is also increasing).
- The number of cores is increasing rapidly (massively parallel (IBM Blue Gene Q) and many-cores architectures (Intel Xeon Phi)).
- Emergence of heterogeneous accelerated architectures (standard processors coupled with GPU, FPGA or PEZY-SC2).
- Machine architecture is becoming more complex at all levels (processors/cores, memory hierarchy, network and I/O).
- Memory per core has been stagnating and is beginning to decrease.
- Performance per core is stagnating and is much lower on some machines than on a simple laptop (IBM Blue Gene, Intel Xeon Phi).
- Throughput towards the disk and memory is increasing more slowly than the computing power.

Amdahl's Law

Statement

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

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

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

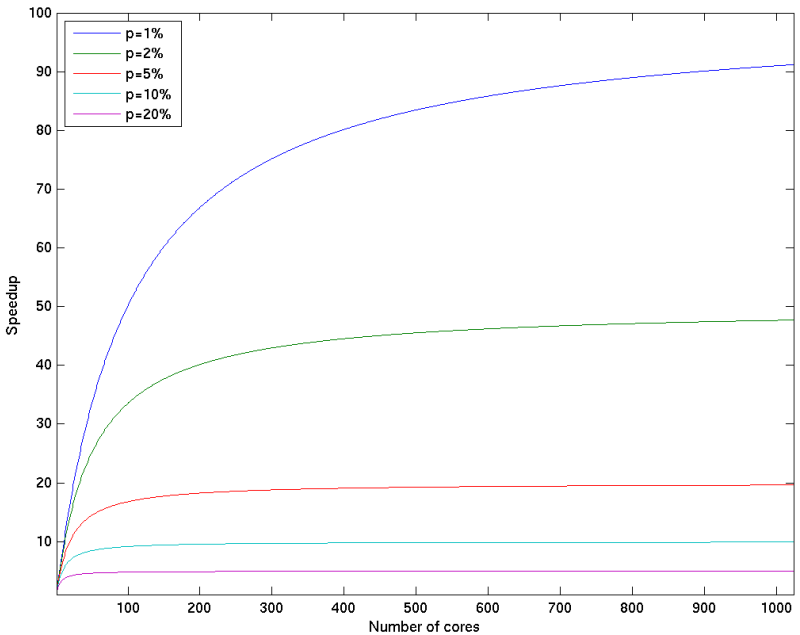
Interpretation

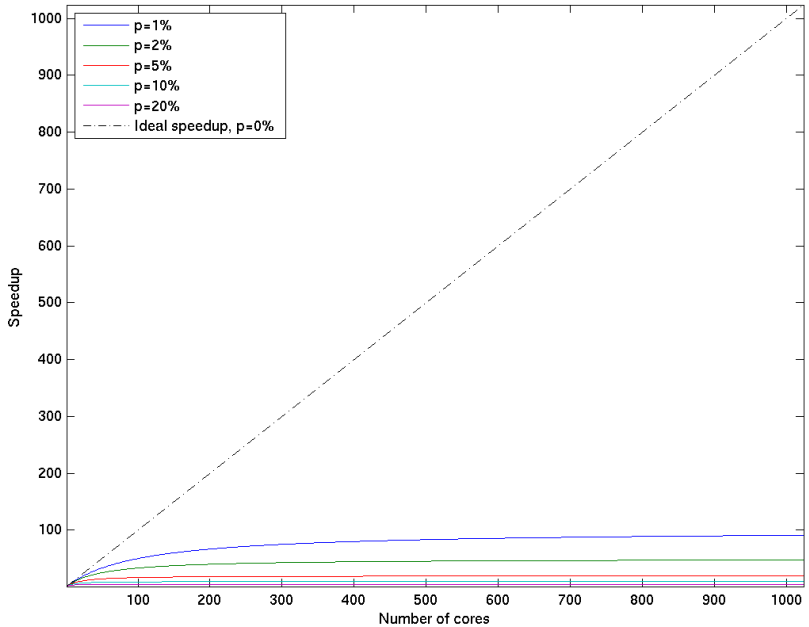
Regardless of the number of cores, the speedup is always less than the inverse of the percentage represented by the purely sequential fraction.

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





The Gustafson-Barsis Law

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:

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

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 Users

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

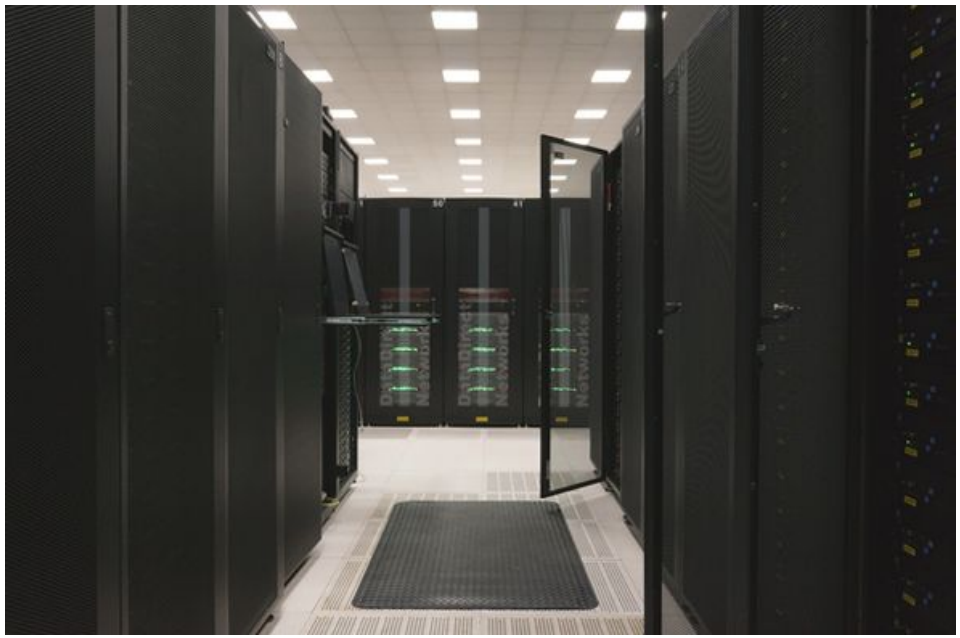
- The time has ended when you only needed to wait a while to obtain better performance (i.e. stagnation of computing power per core).
- Increased necessity to understand the hardware architecture.
- More and more difficult to develop codes on your own (need for experts in HPC as well as multi-disciplinary 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 usage is increasing, but it is still complex and requires a third level of parallelism (MPI+OpenMP+GPU).
- New parallel programming languages are appearing (UPC, Coarray- Fortran, PGAS languages, X10, Chapel, ...), but they are in experimental phases (at variable levels of maturity). Some are very promising; it remains to be seen whether they will be used in real applications.

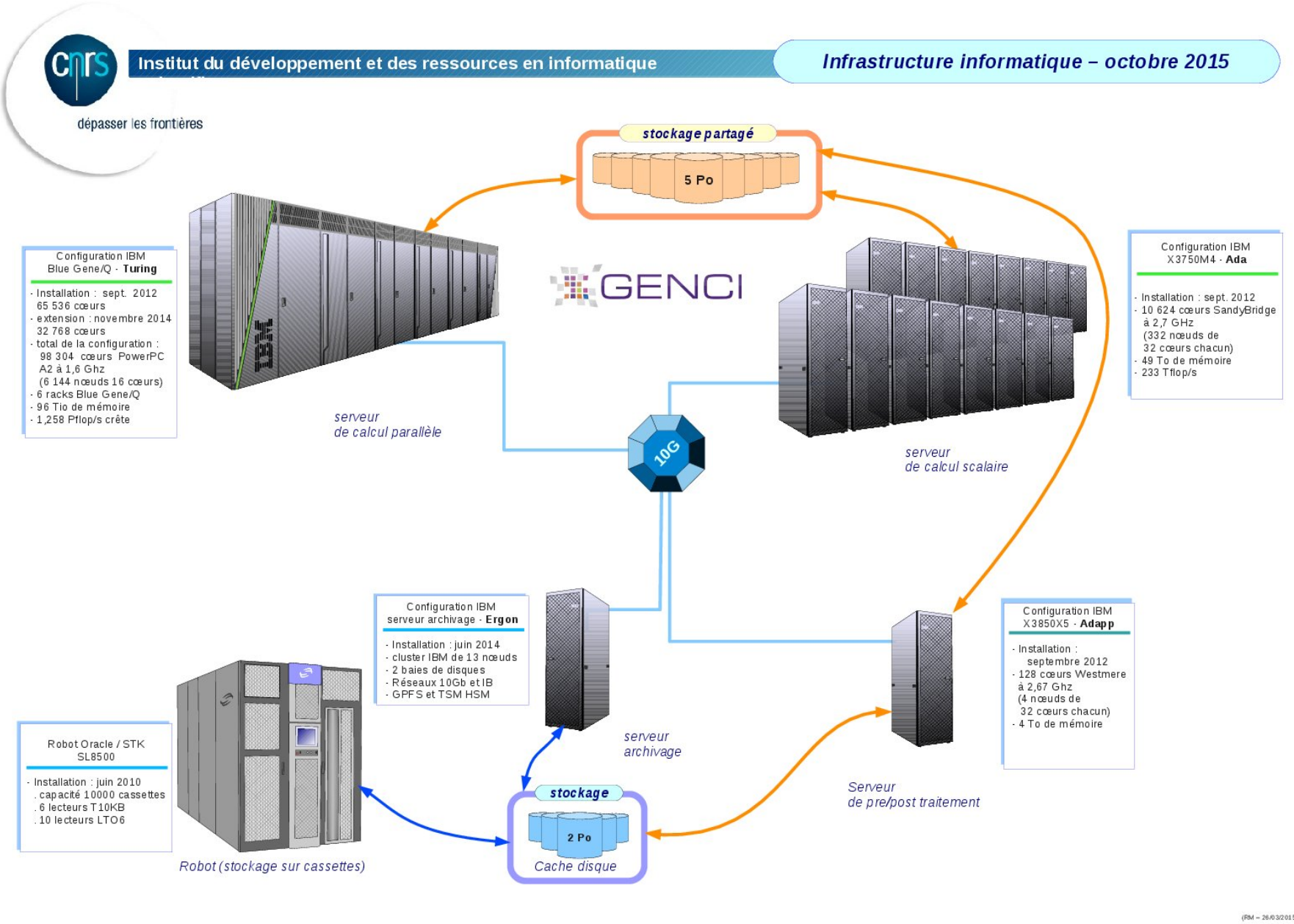
Turing: IBM Blue Gene/Q

Ada: IBM x3750

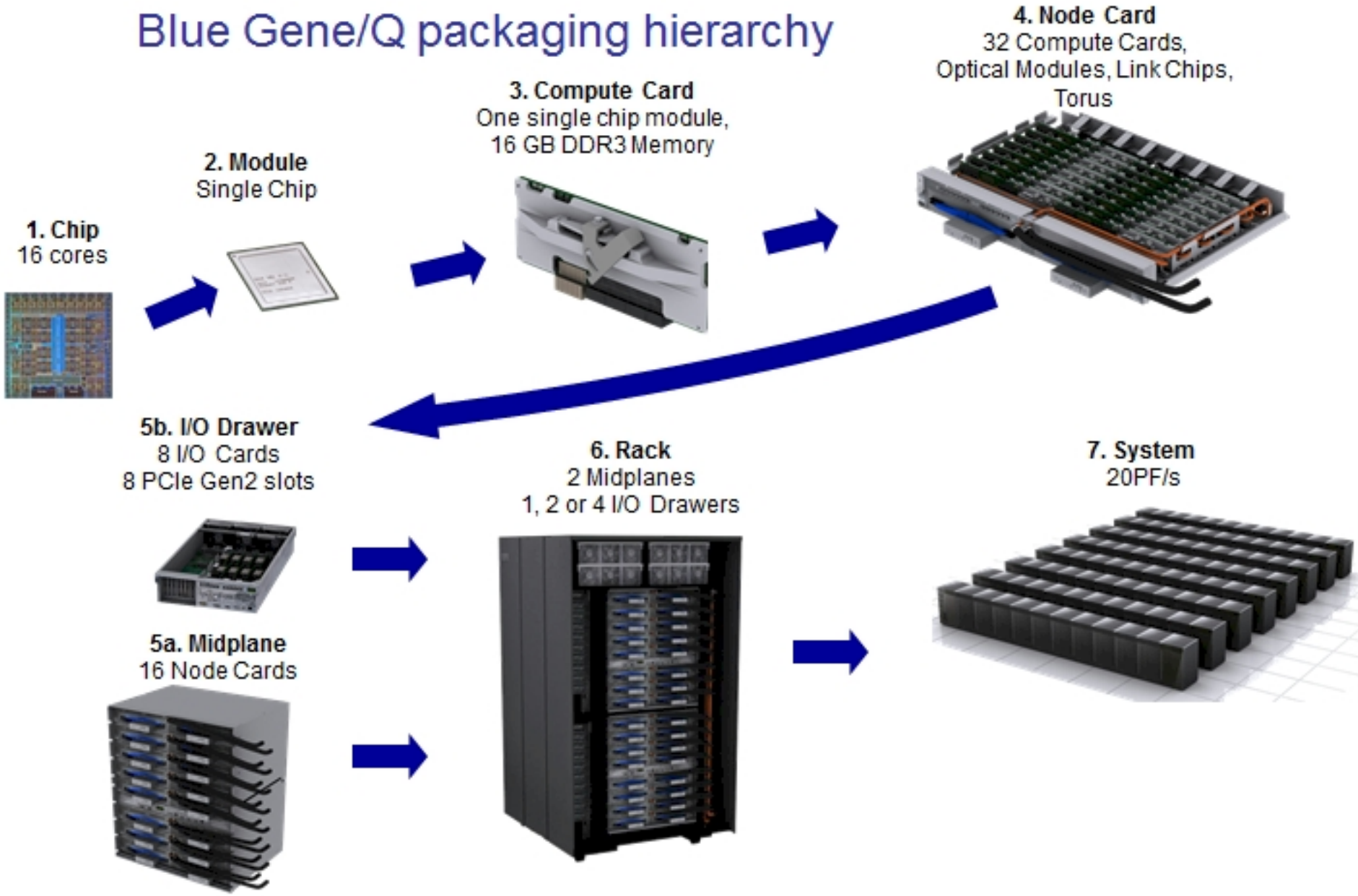


Important numbers

- **Turing: 6 racks Blue Gene/Q**:
  - 6,144 nodes
  - 98,304 cores
  - 393,216 threads
  - 96 TiB
  - 1.258 Tflop/s
  - 636 kW (106 kW/ rack)
- **Ada: 15 racks IBM x3750M4**:
  - 332 compute nodes and 4 pre-/post-processing nodes
  - 10,624 Intel SandyBridge cores at 2.7 GHz
  - 46 TiB
  - 230 Tflop/s
  - 366 kW
- 5 PiB on shared disks between BG/Q and Intel (100 GiB/s peak bandwidth)
- 1 MW for the whole configuration (not counting the cooling system)


Institut du développement et des ressources en informatique
dépasser les frontières
Infrastructure informatique – octobre 2015
stockage partagé
5 Po
GENCI
Configuration IBM
Blue Gene/Q - Turing
· Installation : sept. 2012
65 536 cœurs
· extension : novembre 2014
32 768 cœurs
· total de la configuration :
98 304 cœurs PowerPC
A2 à 1,6 Ghz
(6 144 nœuds 16 cœurs)
· 6 racks Blue Gene/Q
· 96 Tio de mémoire
· 1,258 Pflop/s crête
serveur
de calcul parallèle
10G
Configuration IBM
X3750M4 - Ada
· Installation : sept. 2012
· 10 624 cœurs SandyBridge
à 2,7 GHz
(332 nœuds de
32 cœurs chacun)
· 49 To de mémoire
· 233 Tflop/s
serveur
de calcul scalaire
Configuration IBM
serveur archivage - Ergon
· Installation : juin 2014
· cluster IBM de 13 nœuds
· 2 baies de disques
· Réseaux 10Gb et IB
· GPFS et TSM HSM
serveur
archivage
Configuration IBM
X3850X5 - Adapp
· Installation :
septembre 2012
· 128 cœurs Westmere
à 2,67 Ghz
(4 nœuds de
32 cœurs chacun)
· 4 To de mémoire
Serveur
de pre/post traitement
Robot Oracle / STK
SL8500
· Installation : juin 2010
. capacité 10000 cassettes
. 6 lecteurs T10KB
. 10 lecteurs LTO6
Robot (stockage sur cassettes)
stockage
2 Po
Cache disque

Blue Gene/Q Architecture

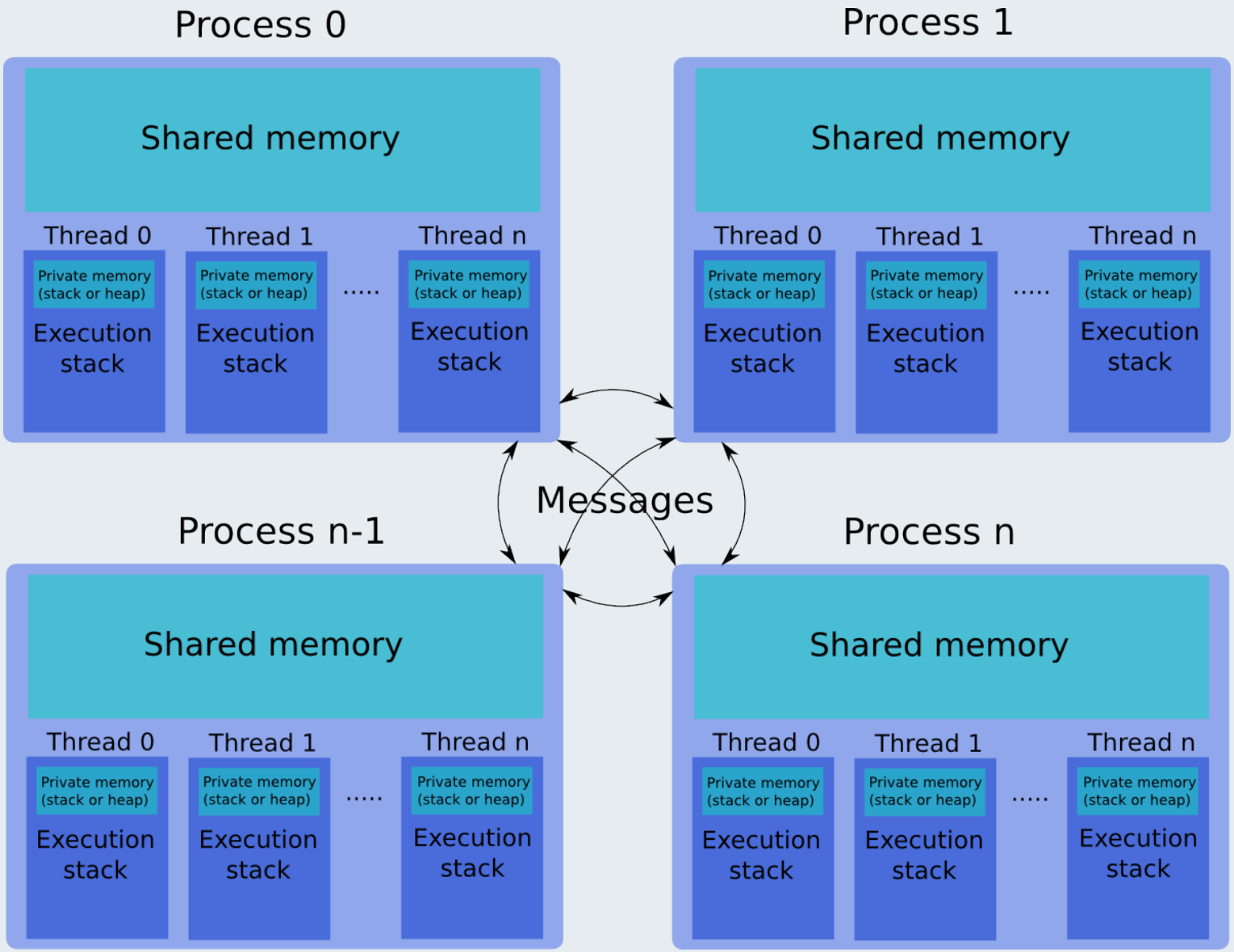


Hybrid programming

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 executing several threads (or processes) per core in order to efficiently use the computational units.
- An MPI parallel code is a succession of computation and communication phases. The granularity of a code is defined as the average ratio between two successive computation and communication phases. The greater the granularity of a code, the more scalable it is. Compared to the pure MPI approach, the hybrid approach significantly increases the granularity and consequently, the scalability of codes.

Disadvantages of hybrid programming

- Complexity and higher level of expertise.
- Necessity of having good MPI and OpenMP performances (Amdahl's law applies separately to the two approaches).
- Total gains in performance are not guaranteed (extra additional costs, ...).

Applications which can benefit from hybrid programming

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

Thread support in MPI

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

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

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 OpenMP *parallel* regions, in OpenMP master regions, or in protected zones (call to `MPI_Is_thread_main`).
- `MPI_THREAD_SERIALIZED`: In the OpenMP parallel regions, MPI calls must be made in critical sections (when necessary, to ensure that only one MPI call is made at a time)
- `MPI_THREAD_MULTIPLE`: No restriction.

State of current implementations

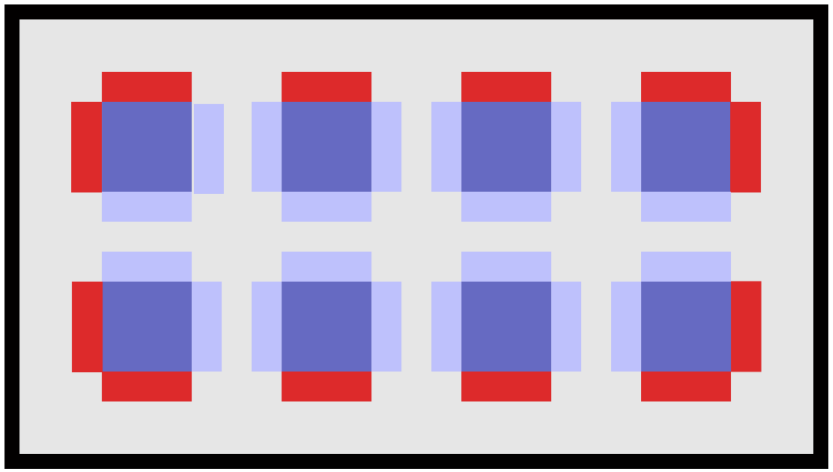
Implementation	Level Supported	Remarks
MPICH	`MPI_THREAD_MULTIPLE`	
OpenMPI	`MPI_THREAD_MULTIPLE`	Must be compiled with *–enable-mpi-threads*
IBM BlueGene/Q	`MPI_THREAD_MULTIPLE`	
IBM PEMPI	`MPI_THREAD_MULTIPLE`	
BullxMPI	`MPI_THREAD_FUNNELED`	
Intel - MPI	`MPI_THREAD_MULTIPLE`	Use *-mt_mpi*
SGI - MPT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

Why memory savings?

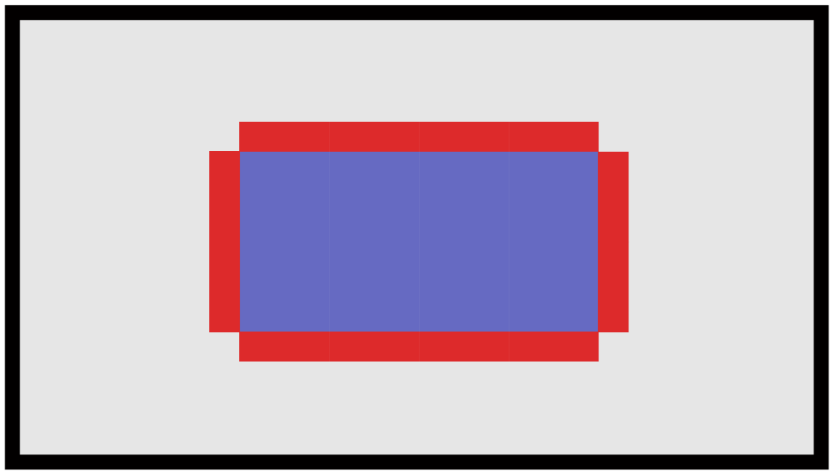
- Hybrid programming allows optimizing code adequacy to the target architecture (generally composed of shared-memory nodes [SMP] linked by an interconnection network). The advantage of shared memory inside a node is that it is not necessary to duplicate data in order to exchange them. Every thread can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

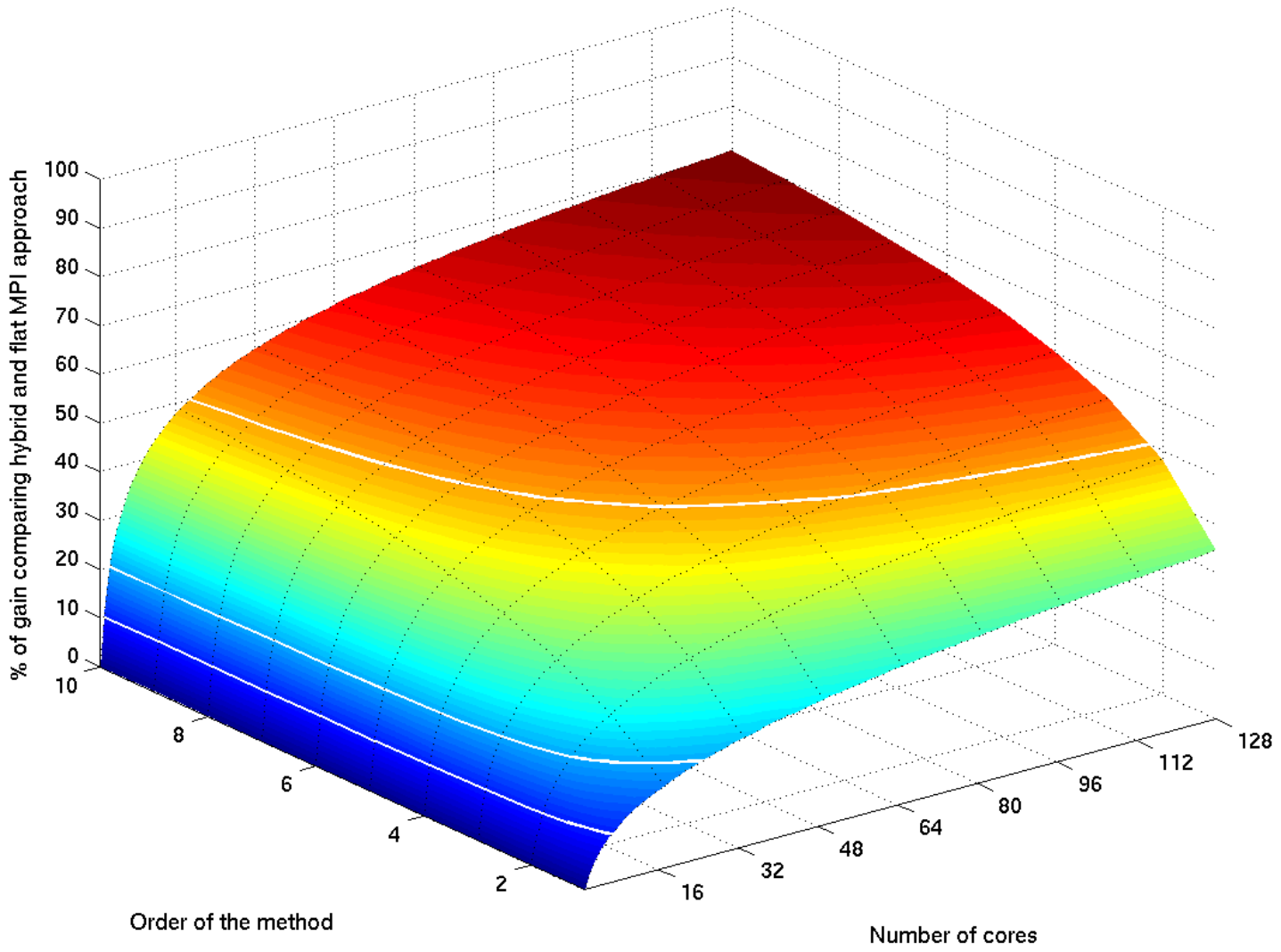
Extra-node ghost cells

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 mutiplication in the number of available cores within a node as well as the generalization of hyperthreading or SMT (the possibility of running multiple threads simultaneously on one core).
  3. General use of high-order numerical methods (computing costs decreasing, thanks particularly to hardware accelerators).
- The benefits will make the transition to hybrid programming almost mandatory...

Optimal Use of the Interconnection Network

How to optimise the use of the inter-node interconnection network

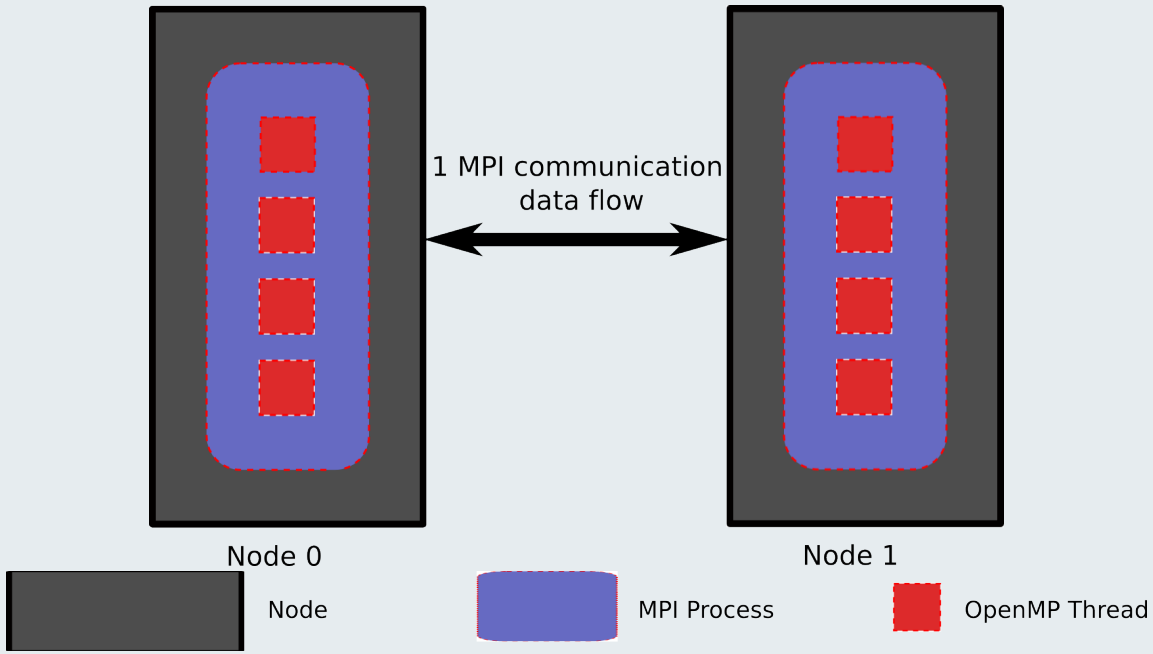
- The hybrid approach aims to use the available hardware resources the most effectively (shared memory, memory hierarchy, communication network).
- One of the difficulties of hybrid programming is to generate a sufficient number of communication flows in order to make the best use of the inter-node communication network.
- In fact, the throughputs of inter-node interconnection networks of recent architectures are high (bidirectional throughput peak of 10 GB/s on Ada, for example) and one data flow alone cannot saturate it; only a fraction of the network is really used, the rest being wasted.
- IDRIS has developed a small benchmark SBPR (*Saturation Bande Passante Réseau* [Network Bandwidth Saturation]), a simple parallel ping-pong test aimed at determining the number of concurrent flows required to saturate the network.

MPI_THREAD_FUNNELED version of SBPR

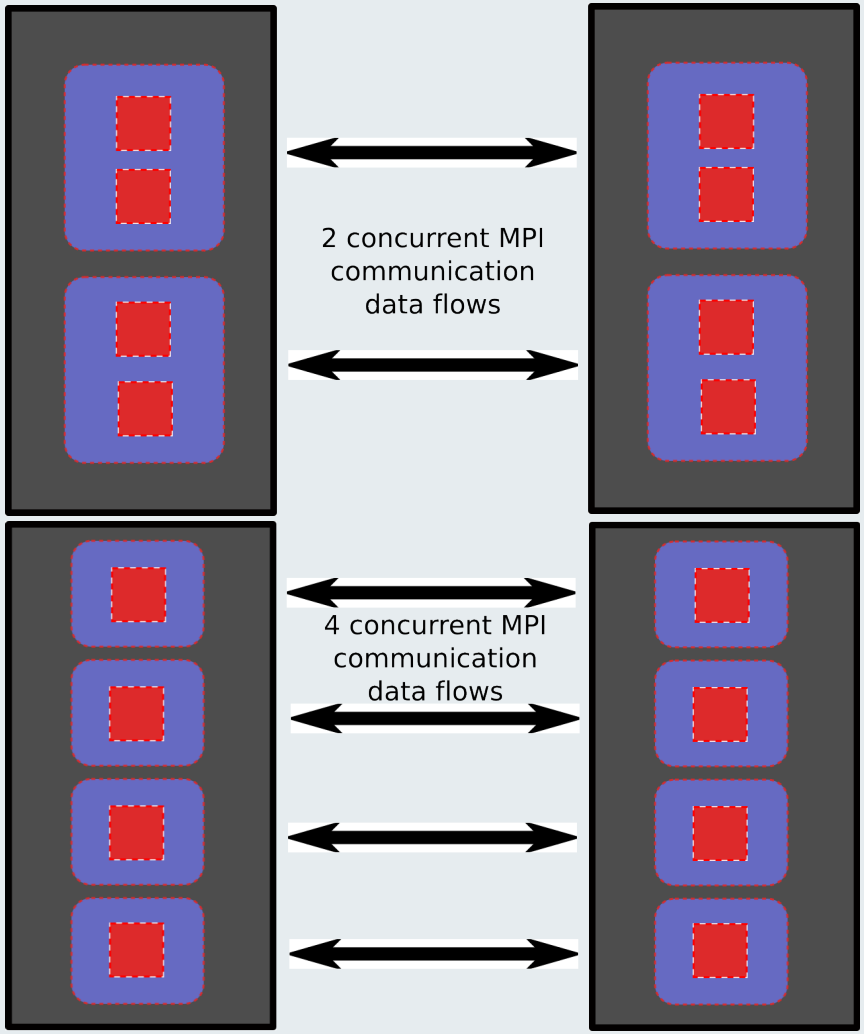
MPI_THREAD_FUNNELED approach:

- We increase the network bandwidth actually used by increasing the number of MPI processes per node (i.e. we generate as many parallel communication flows as there are MPI processes per node).
- The basic solution, which consists of using as many OpenMP threads as there are cores inside a node and as many MPI processes as the number of nodes, is not generally the most efficient: The resources are not being used optimally, in particular the network.
- We look for the optimal ratio value between the number of MPI processes per node and the number of OpenMP threads per MPI process. The greater the ratio, the better the inter-node network flow rate, but the granularity is not as good. A compromise has to be found.
- The number of MPI processes (i.e. the data flow to be managed simultaneously) necessary to saturate the network varies greatly from one architecture to another.
- This value could be a good indicator of the optimal ratio of the number of MPI processes/number of OpenMP threads per node of a hybrid application.

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



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

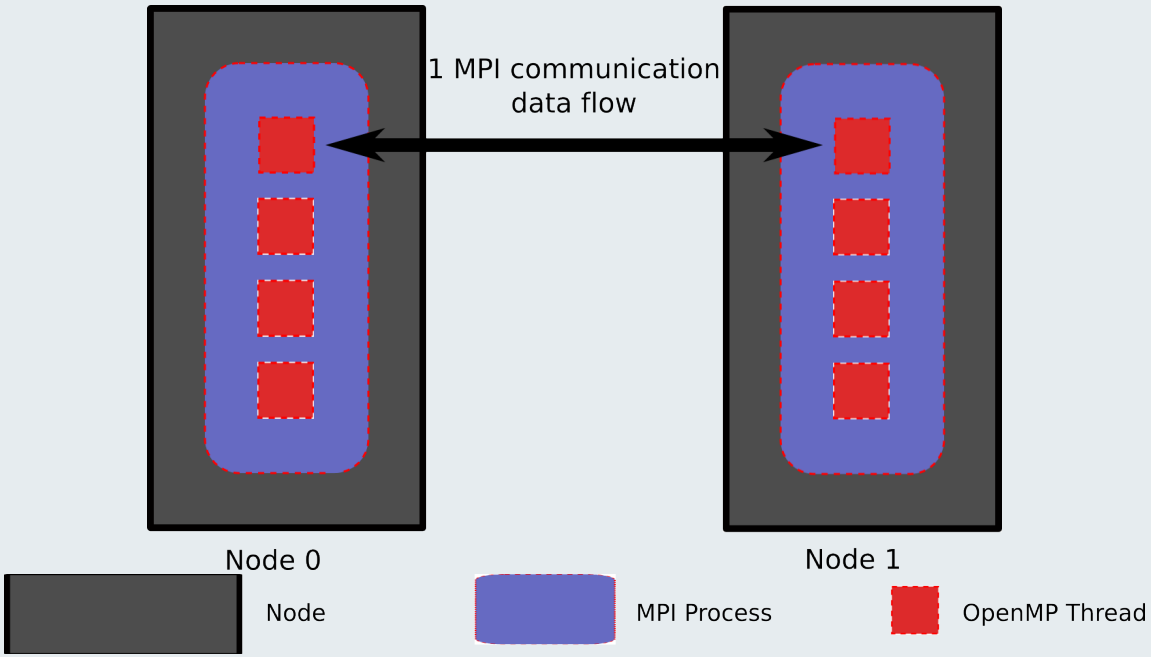


MPI_THREAD_MULTIPLE version of SBPR

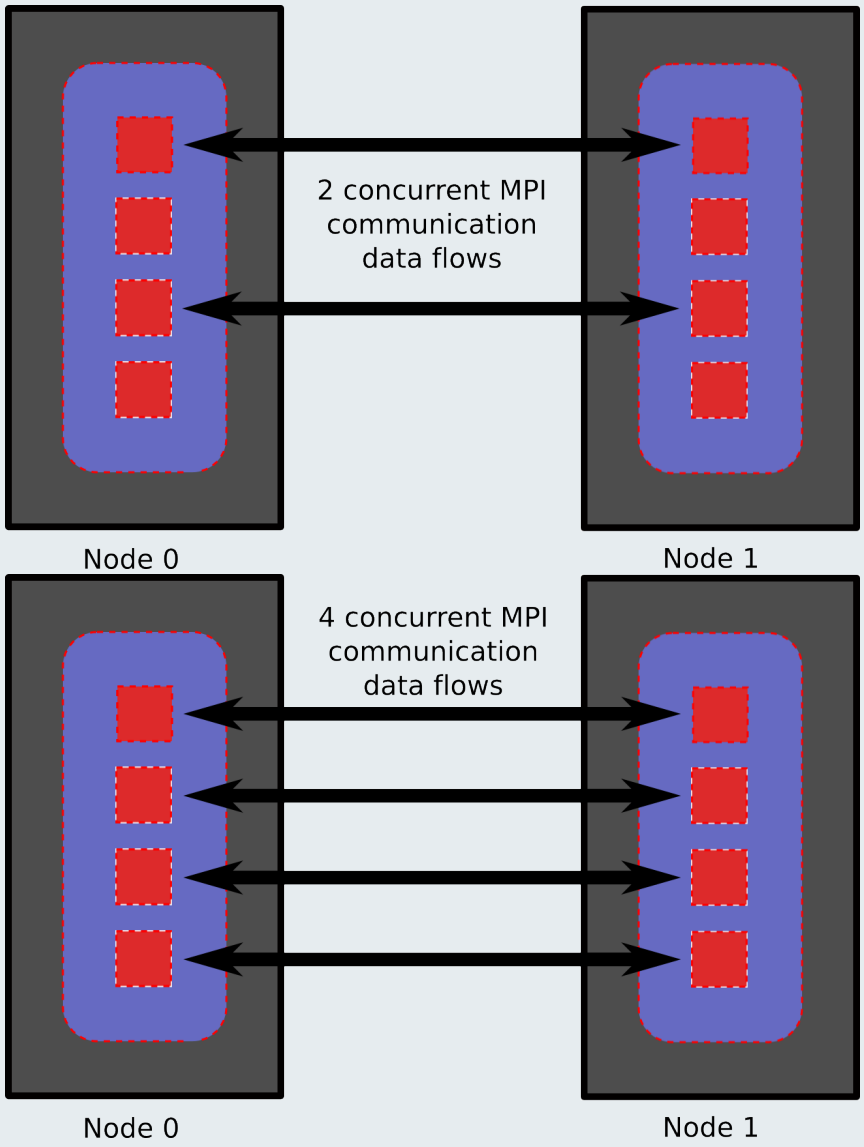
MPI_THREAD_MULTIPLE approach:

- 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

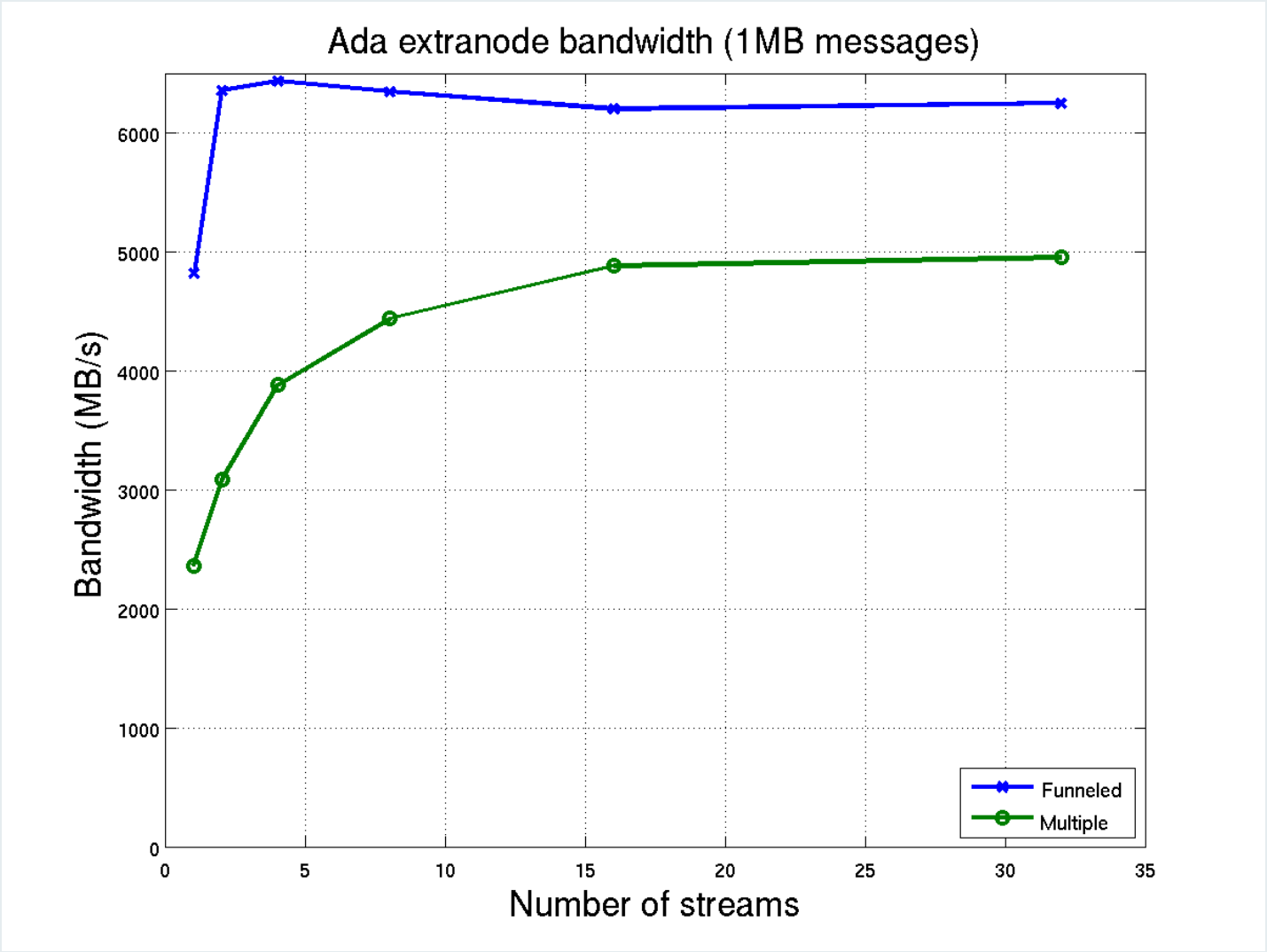


MPI_THREAD_MULTIPLE version of SBPR: Example on a 4-Core (BG/P) SMP Node



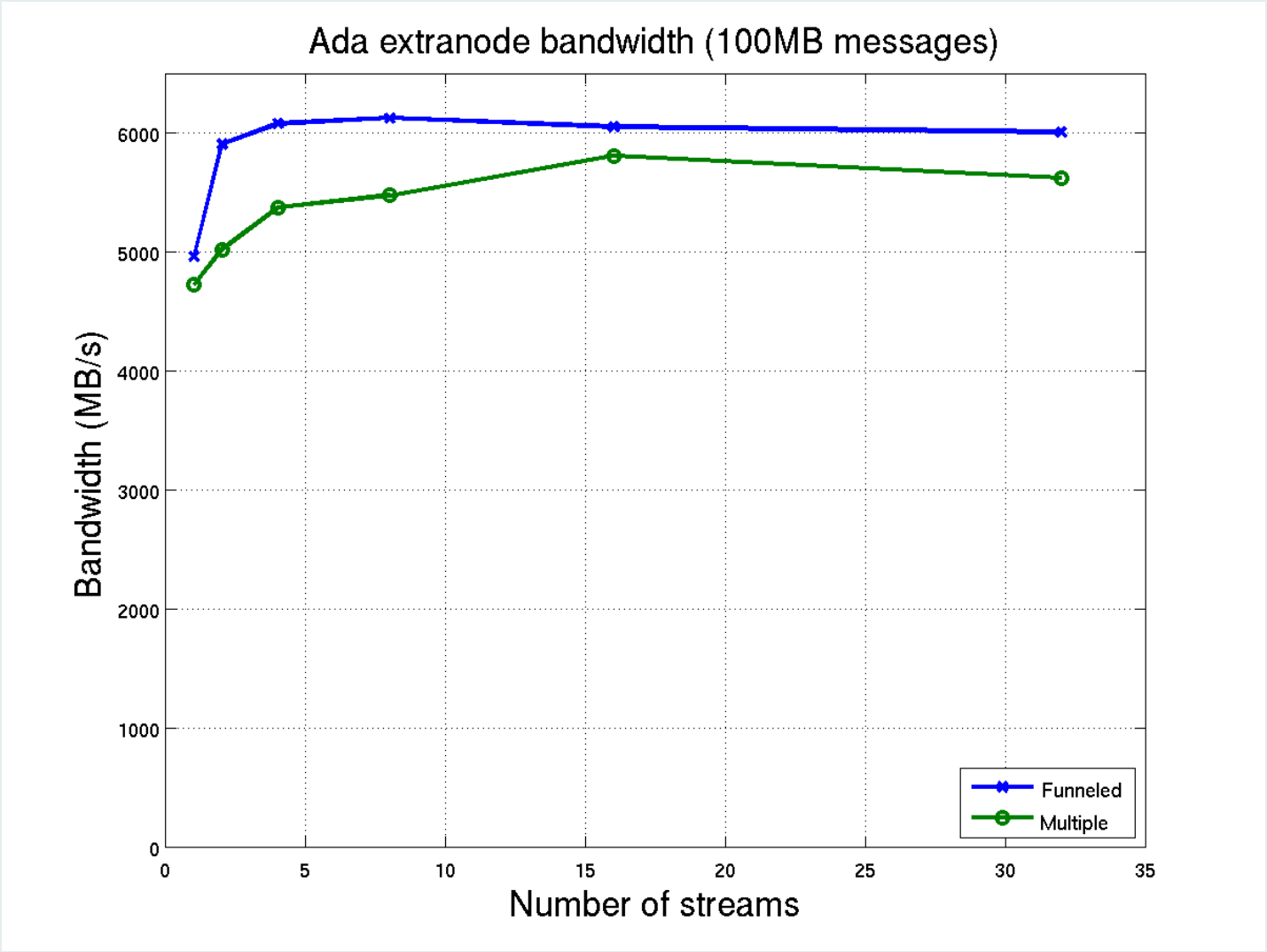
SBPR: Results on Ada

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



SBPR: Results on Ada

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

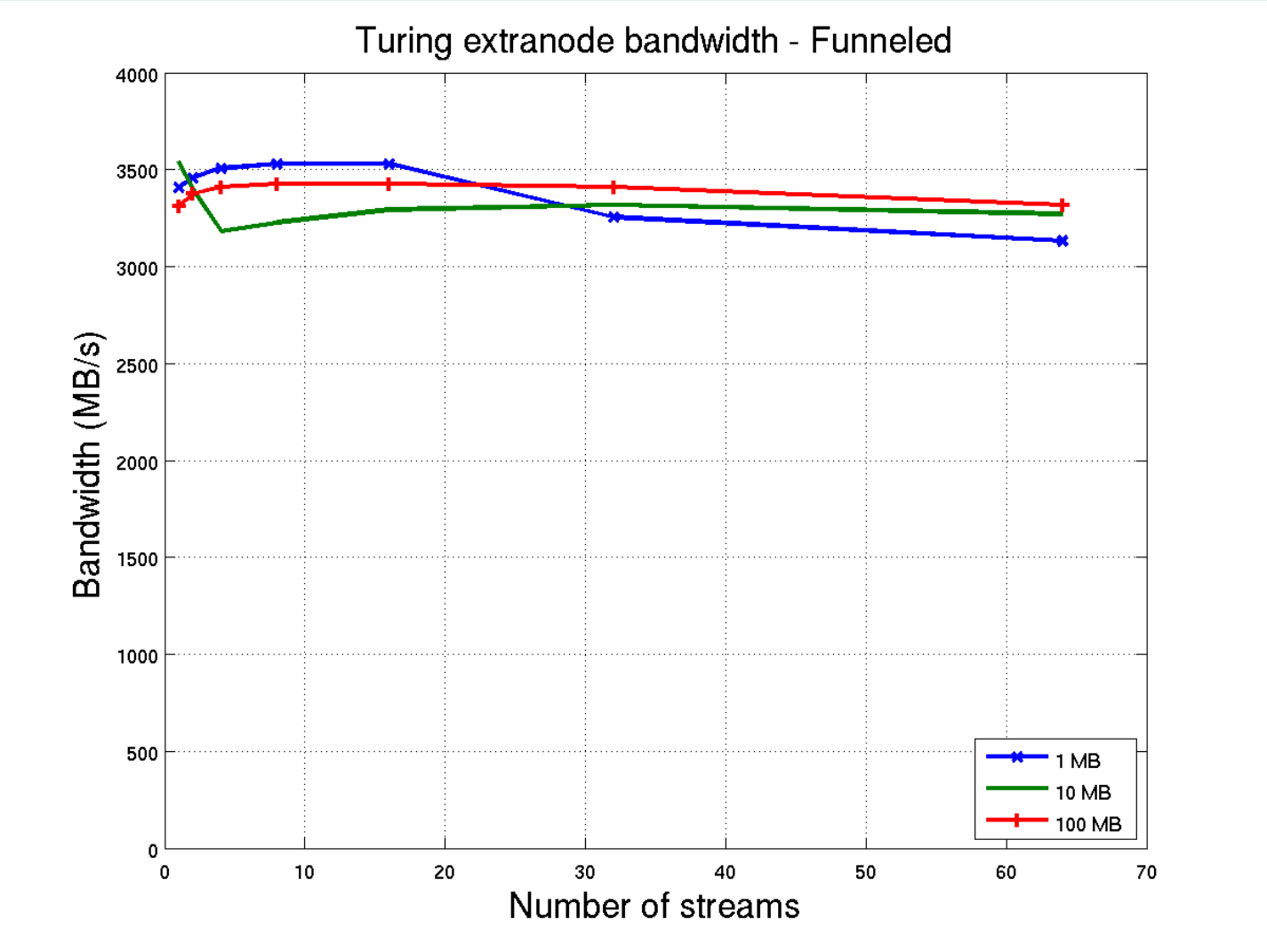


SBPR: Results on Ada

- 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.

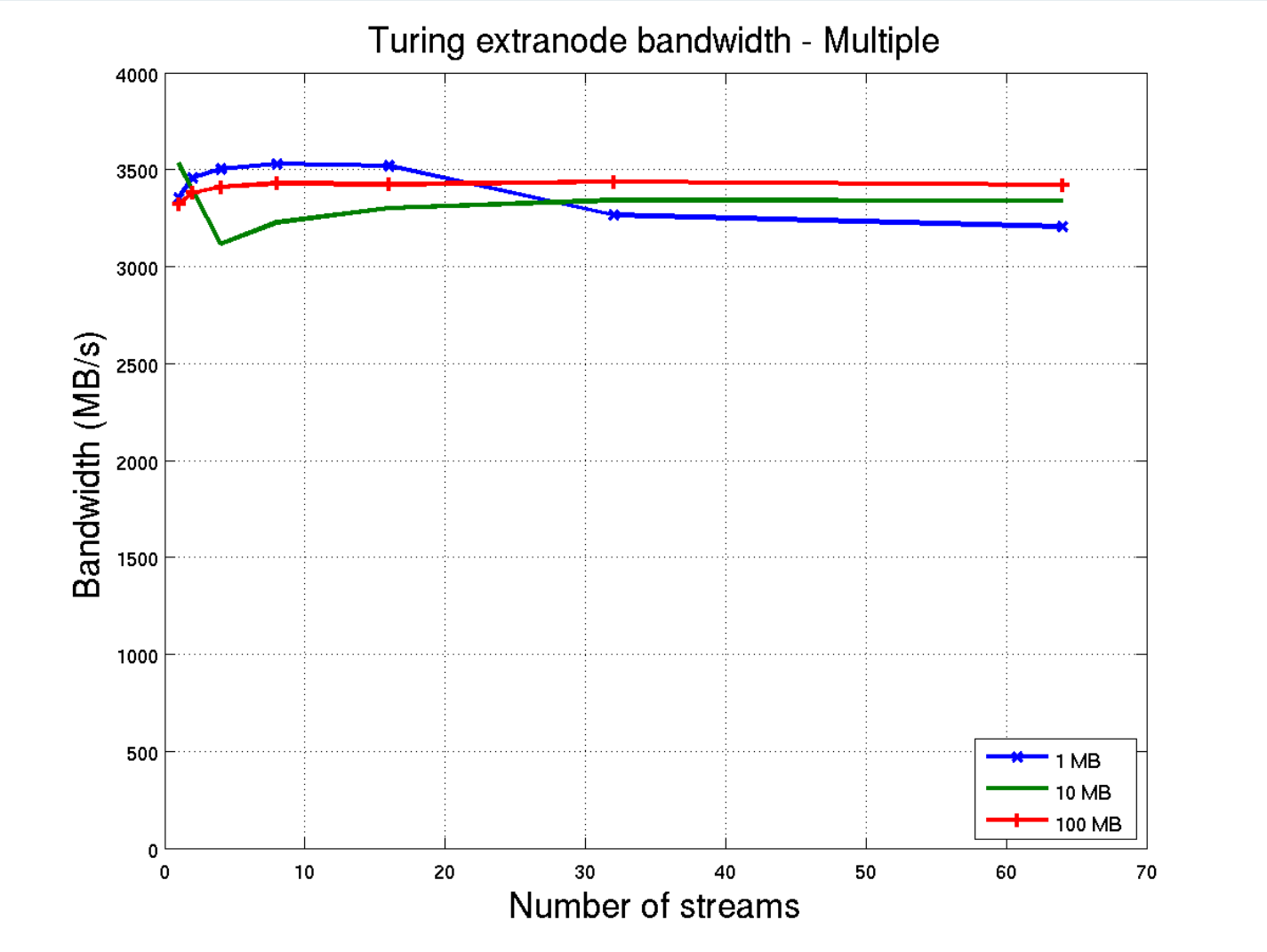
SBPR: Results on Turing

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).

Effects of a non-uniform architecture

Non-uniform architecture

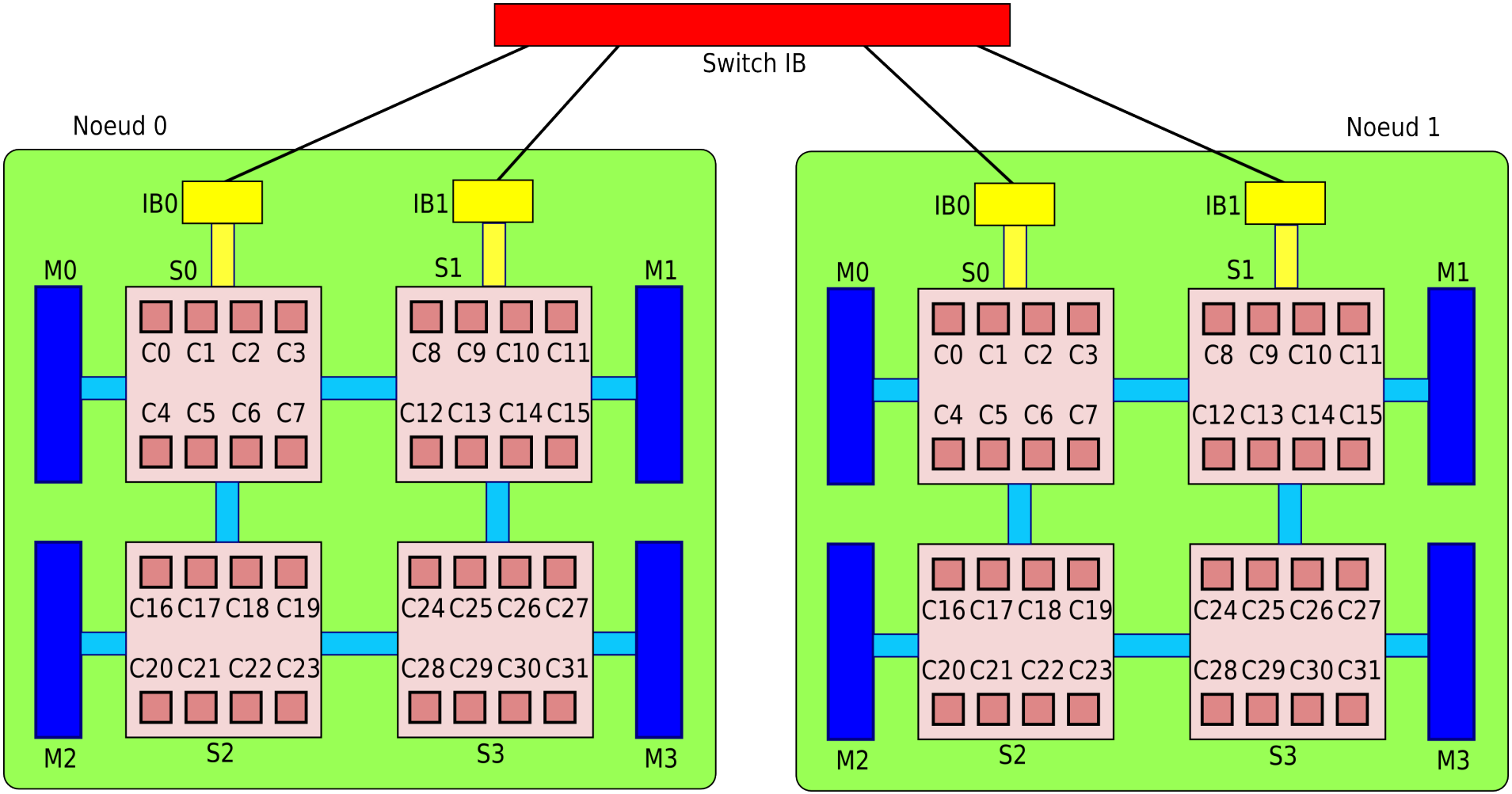
Most modern supercomputers have a non-uniform architecture :

- 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.

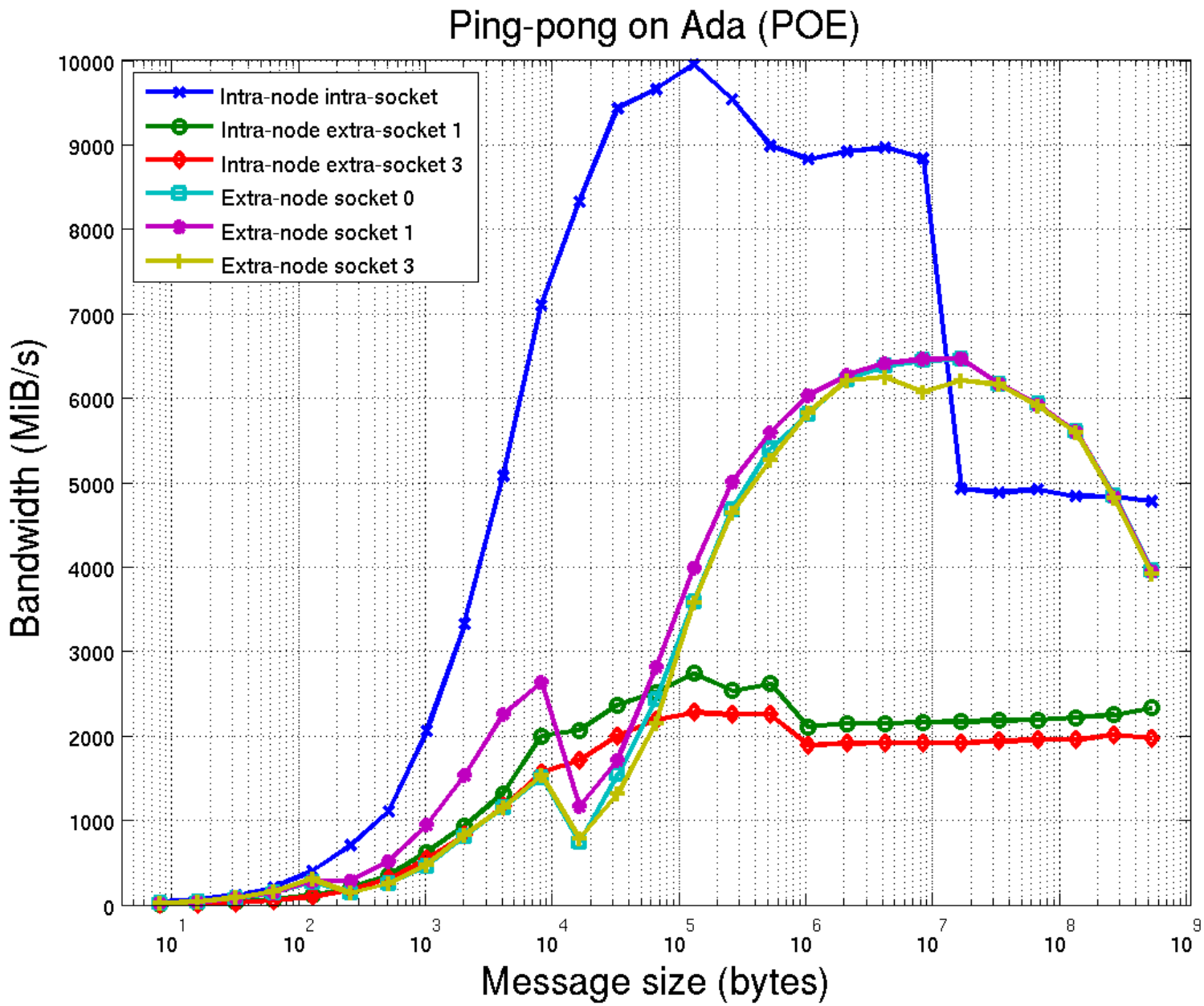
Effects

- 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



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.

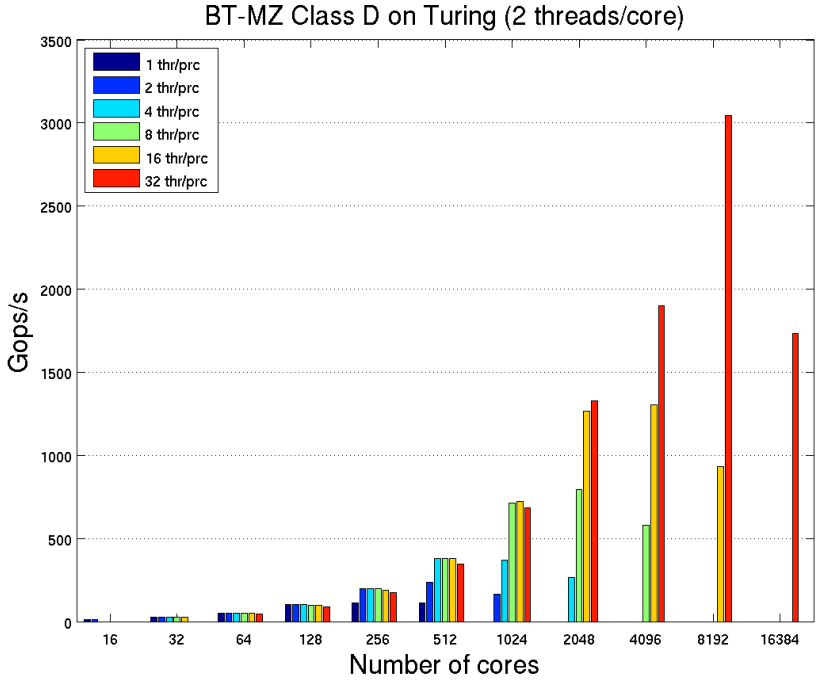
Selected Application: SP-MZ

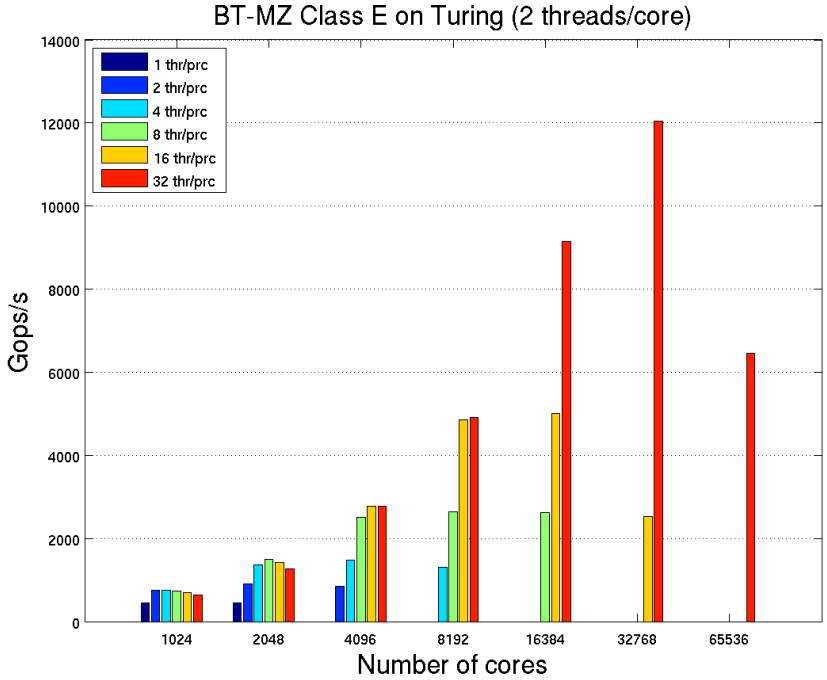
SP-MZ: scalar pentadiagonal solver.

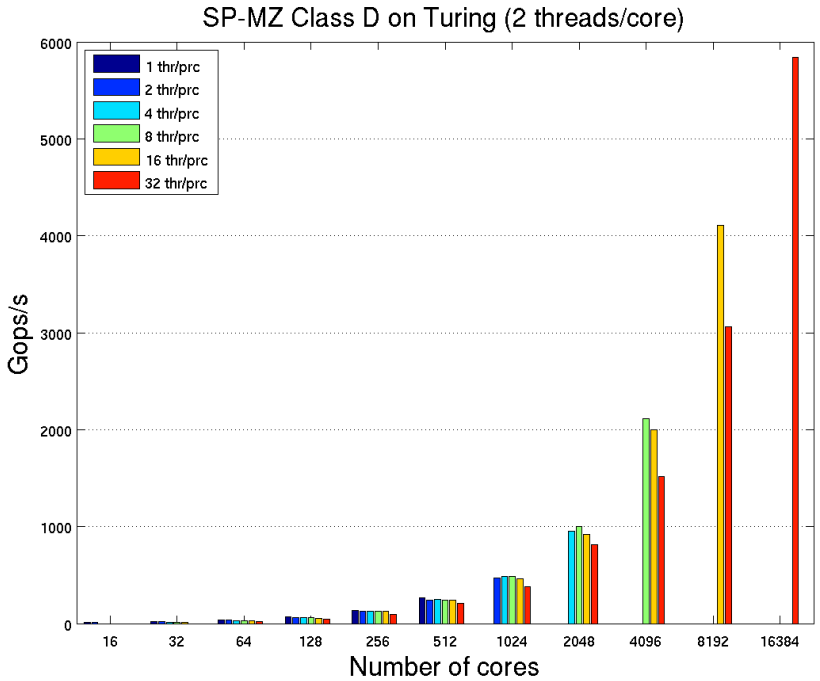
- All the zone sizes are identical: perfect load balancing.
- The hybrid approach should not bring any improvement.

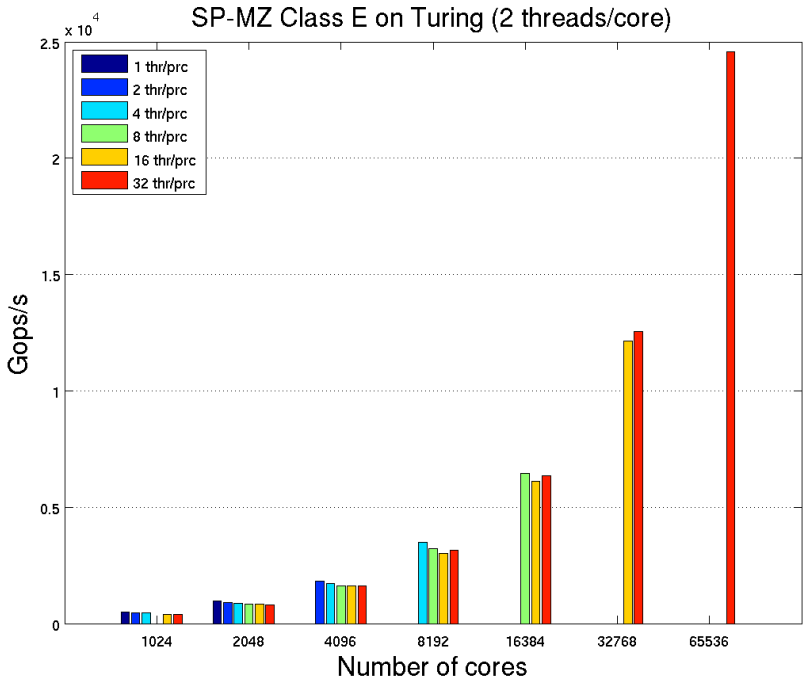
Selected Problem Sizes

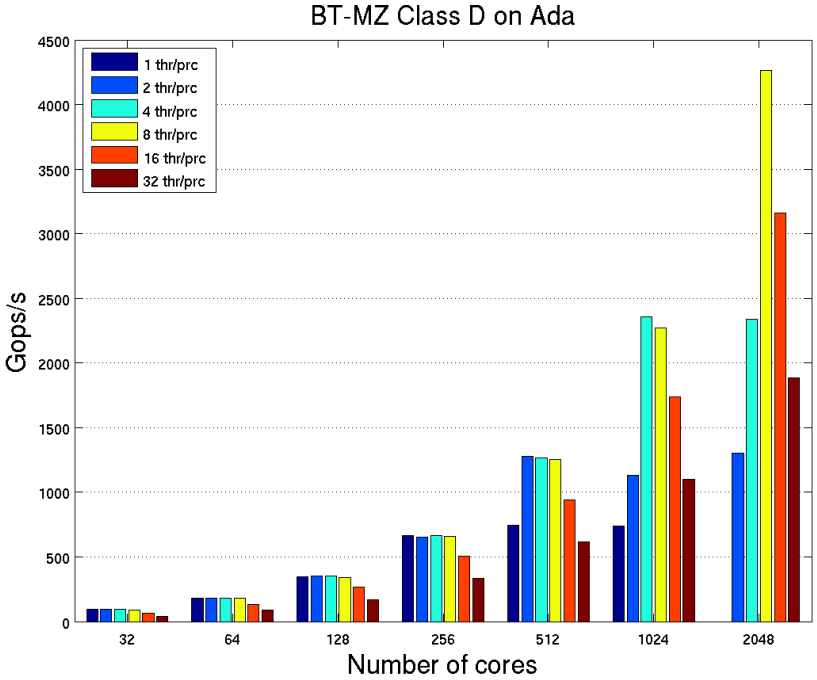
- Class D: 1024 zones (and therefore limited to 1024 MPI processes), 1632 x 1216 x 34 grid points (13 GiB)
- Class E: 4096 zones (and therefore limited to 4096 MPI processes), 4224 x 3456 x 92 grid points (250 GiB)

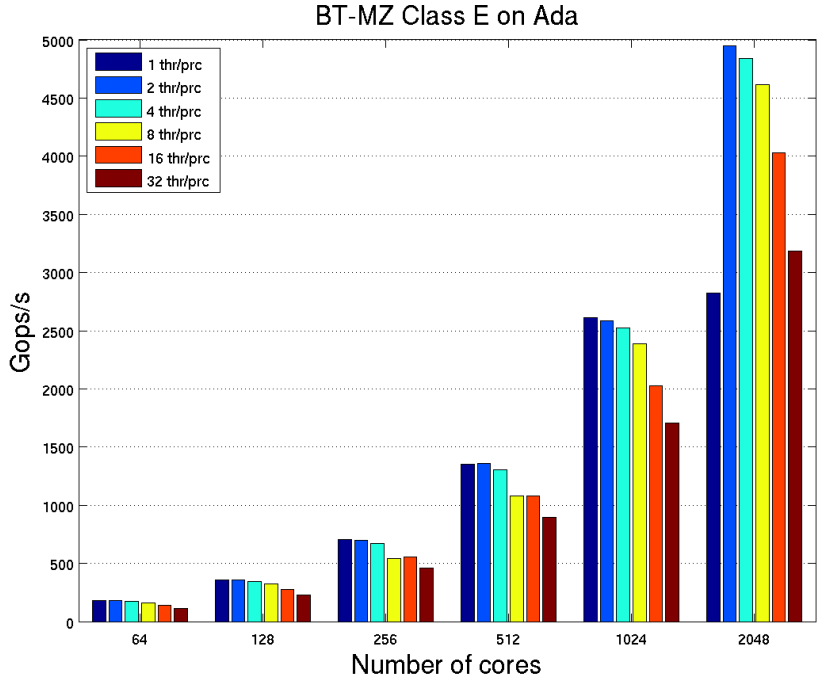

BT-MZ Class D on Turing (2 threads/core)
1 thr/prc
2 thr/prc
4 thr/prc
8 thr/prc
16 thr/prc
32 thr/prc
Gops/s
Number of cores
16
32
64
128
256
512
1024
2048
4096
8192
16384

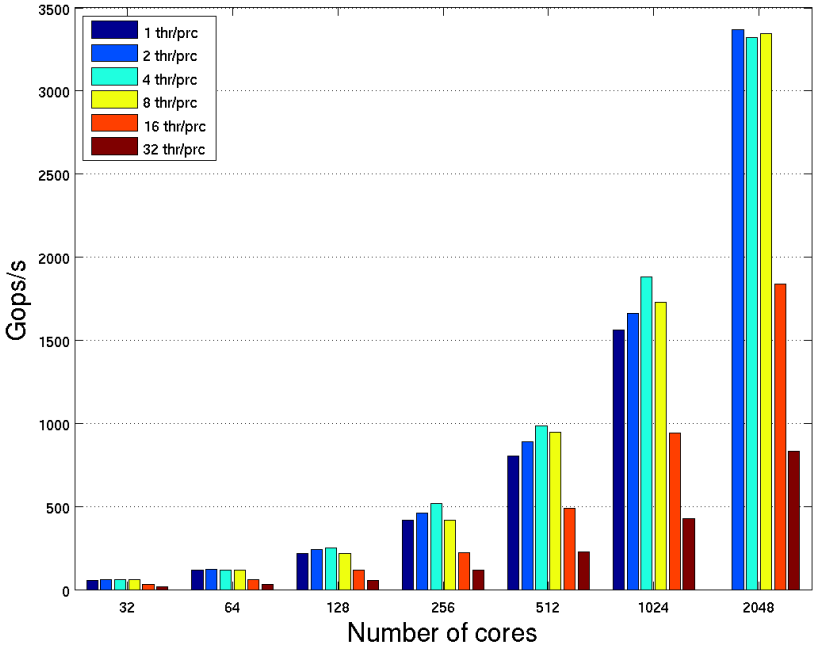

BT-MZ Class E on Turing (2 threads/core)
1 thr/prc
2 thr/prc
4 thr/prc
8 thr/prc
16 thr/prc
32 thr/prc
Gops/s
Number of cores
1024
2048
4096
8192
16384
32768
65536

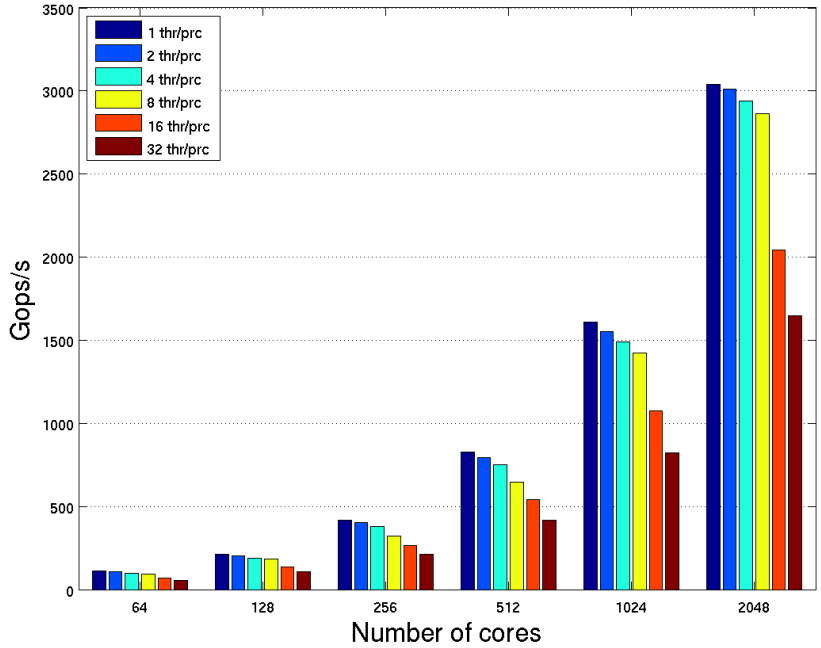

SP-MZ Class D on Turing (2 threads/core)
1 thr/prc
2 thr/prc
4 thr/prc
8 thr/prc
16 thr/prc
32 thr/prc
Gops/s
Number of cores
16
32
64
128
256
512
1024
2048
4096
8192
16384


SP-MZ Class E on Turing (2 threads/core)
x 10^4
1 thr/prc
2 thr/prc
4 thr/prc
8 thr/prc
16 thr/prc
32 thr/prc
Gops/s
Number of cores
1024
2048
4096
8192
16384
32768
65536


BT-MZ Class D on Ada
1 thr/prc
2 thr/prc
4 thr/prc
8 thr/prc
16 thr/prc
32 thr/prc
Gops/s
Number of cores
0
500
1000
1500
2000
2500
3000
3500
4000
4500
32
64
128
256
512
1024
2048


BT-MZ Class E on Ada
1 thr/prc
2 thr/prc
4 thr/prc
8 thr/prc
16 thr/prc
32 thr/prc
Gops/s
Number of cores
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
64
128
256
512
1024
2048


SP-MZ Class D on Ada
1 thr/prc
2 thr/prc
4 thr/prc
8 thr/prc
16 thr/prc
32 thr/prc
Gops/s
Number of cores
0
500
1000
1500
2000
2500
3000
3500
32
64
128
256
512
1024
2048


SP-MZ Class E on Ada
1 thr/prc
2 thr/prc
4 thr/prc
8 thr/prc
16 thr/prc
32 thr/prc
Gops/s
Number of cores
0
500
1000
1500
2000
2500
3000
3500
64
128
256
512
1024
2048

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

Analysis of Results: SP-MZ

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

Presentation of the HYDRO Code (1)

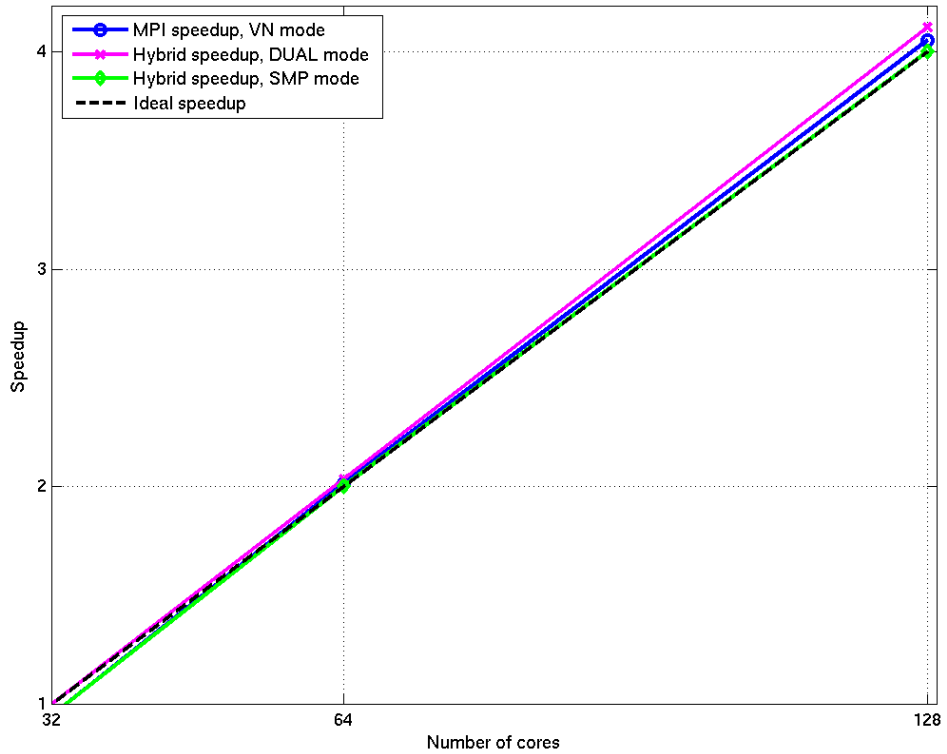
- This is the code used for the hands-on exercises of the hybrid course.
- Hydrodynamics code, 2D-Cartesian grid, finite volume method, resolution of a Riemann problem on the interfaces with a Godunov method.
- For the last few years, in the framework of the IDRIS technology watch, this code has served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

Presentation of the HYDRO Code (2)

- Today, there are the following hydro versions:
  - Original version, F90 monoprocessor (P.-Fr. Lavallée, R. Teyssier)
  - Monoprocessor C version (G. Colin de Verdière)
  - MPI F90 parallel version (1D P.-Fr. Lavallée, 2D Ph. Wautelet)
  - MPI C parallel version (2D Ph. Wautelet)
  - OpenMP Fine-Grain and Coarse-Grain F90 parallel version (P.-Fr. Lavallée)
  - OpenMP Fine-Grain C parallel version (P.-Fr. Lavallée)
  - MPI2D-OpenMP Fine-Grain and Coarse-Grain F90 hybrid parallel version (P.-Fr. Lavallée, Ph. Wautelet)
  - MPI2D-OpenMP Fine-Grain hybrid parallel version C (P.-Fr. Lavallée, Ph. Wautelet)
  - C GPGPU CUDA, HMPP, OpenCL version (G. Colin de Verdière)
  - Pthreads parallel version C (D. Lecas)
- Many other versions are under development: UPC, CAF, OpenACC, OpenMP4.5, ...

Results for the nx = 10000, ny = 10000 domain

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 10 racks on Babel - Weak Scaling

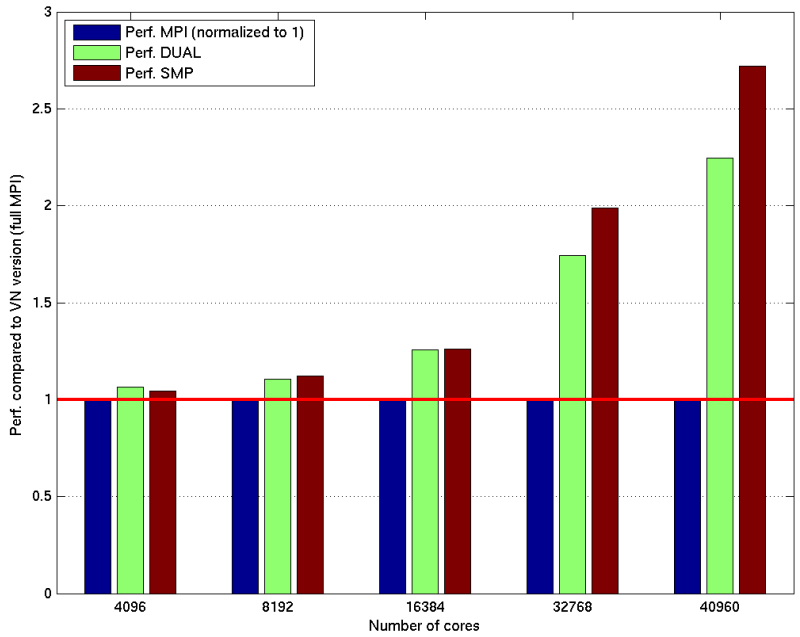
Characteristics of the domains used for weak scaling

- On 4096 cores, total number of points of the domain: 16 10^8
  - 400000x4000: domain elongated in the first dimension
  - 40000x40000: square domain
  - 4000x400000: domain elongated in the second dimension
- On 8192 cores, total number of domain points: 32 10^8
  - 800000x4000: domain elongated in the first dimension
  - 56568x56568: square domain
  - 4000x800000: domain elongated in the second dimension
- On 16384 cores, total number of points of the domain: 64 10^8
  - 1600000x4000: domain elongated in the first dimension
  - 80000x80000: square domain
  - 4000x1600000: domain elongated in the second dimension
- On 32768 cores, total number of points of the domain: 128 10^8
  - 3200000x4000: domain elongated in the first dimension
  - 113137x113137: square domain
  - 4000x3200000: domain elongated in the second dimension
- On 40960 cores, total number of points of the domain: 16 10^9
  - 4000000x4000: domain elongated in the first dimension
  - 126491x126491: square domain
  - 4000x4000000: domain elongated in the second dimension

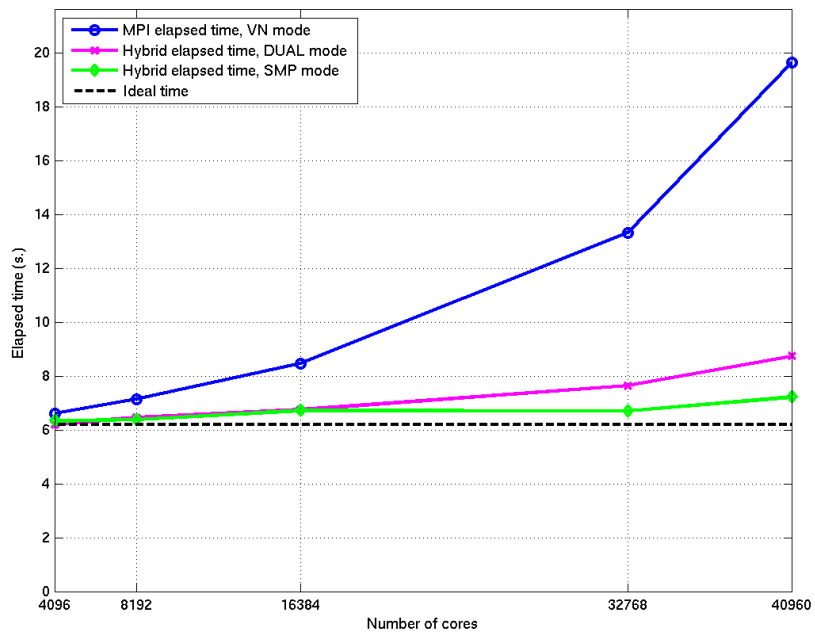
Results using 10 racks on Babel - Weak Scaling

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

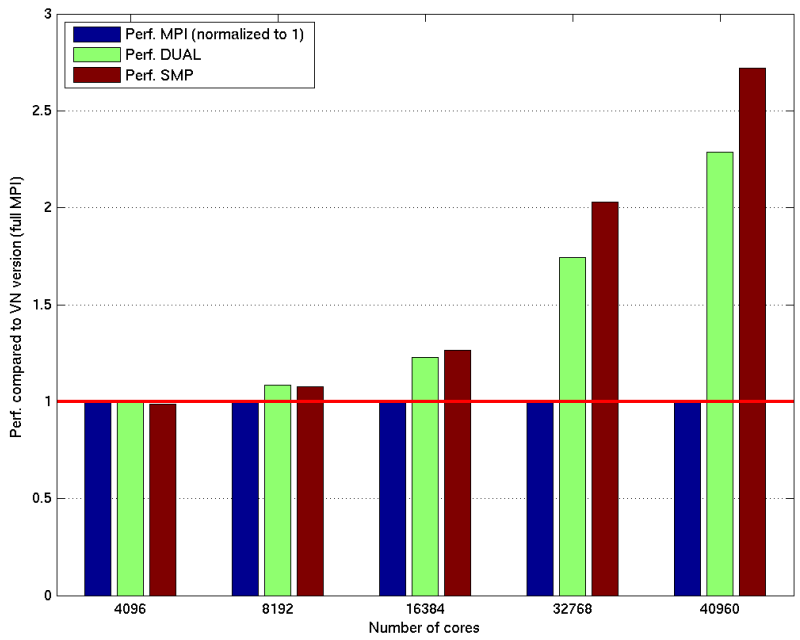


Elapsed execution time

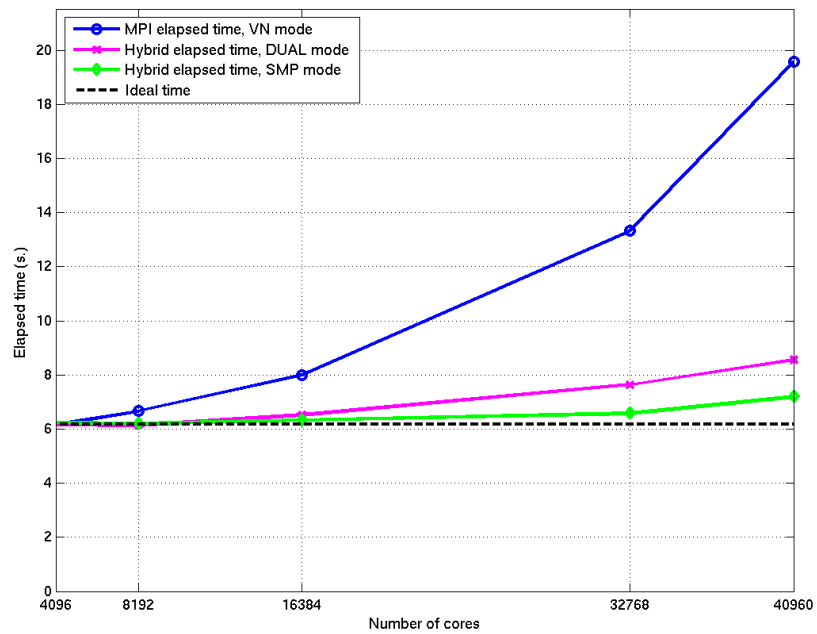
Results using 10 racks on Babel - Weak Scaling

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

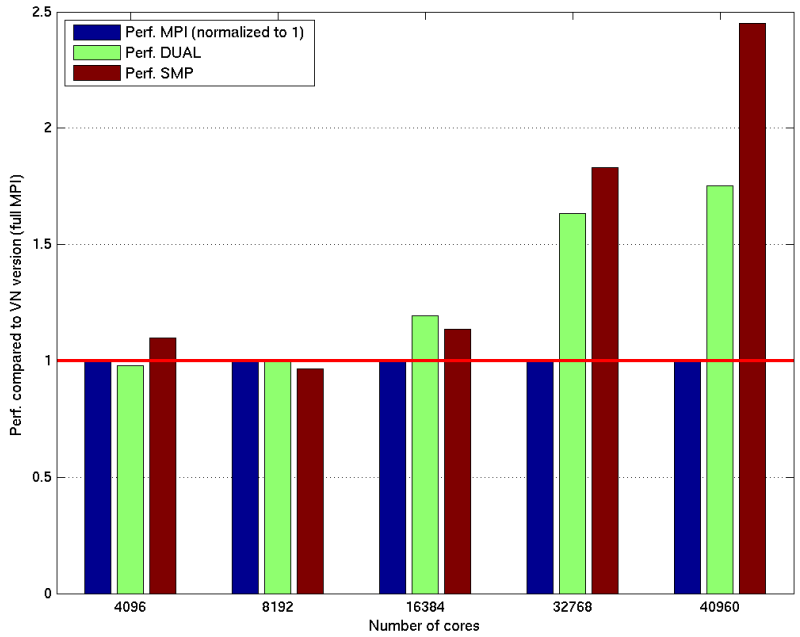


Elapsed execution time

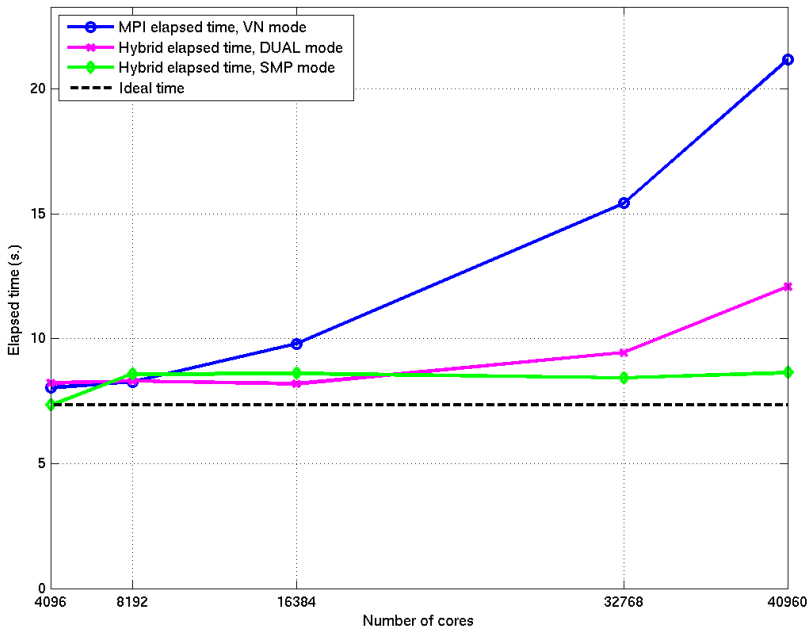
Results using 10 racks on Babel - Weak Scaling

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

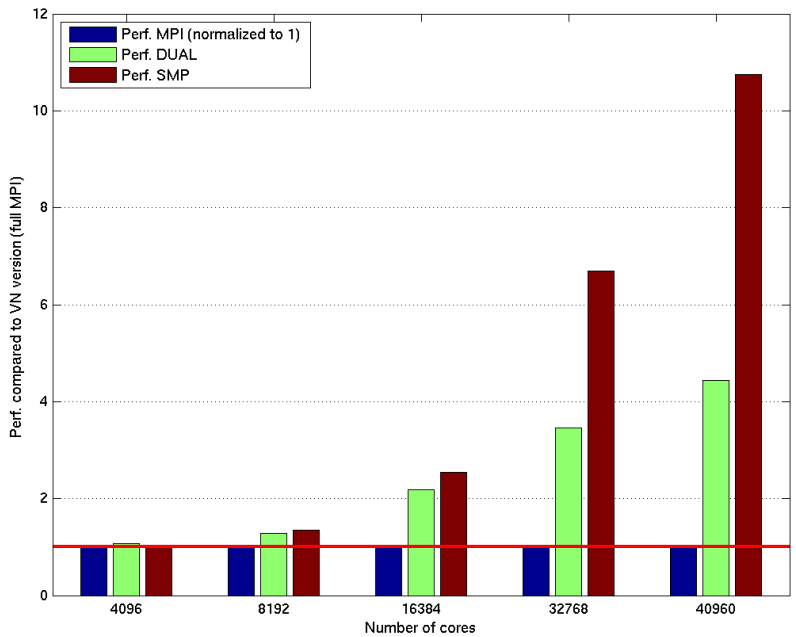
Interpretation of results

- 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. It has difficulty scaling up to 16384 cores and then the elapsed time begins to explode.
- As we expected, the DUAL hybrid version, but even more the SMP version, behave very well up to 32768 cores with nearly constant elapsed times. On 40960 cores, the SMP version shows a very slight additional cost; on the DUAL version the additional cost becomes significant.
- In weak scaling, the scalability limit of the flat MPI version is 16384 cores, that of the DUAL version is 32768 cores, and that of the SMP version has not yet been reached on 40960 cores!
- On 40960 cores, the SMP hybrid version is between 2.5 and 3 times faster than the pure MPI version.
- It is clear that scaling (here over 16K cores) with this type of parallelization method (i.e. domain decomposition), requires hybrid parallelization. It is not enough to use MPI alone !

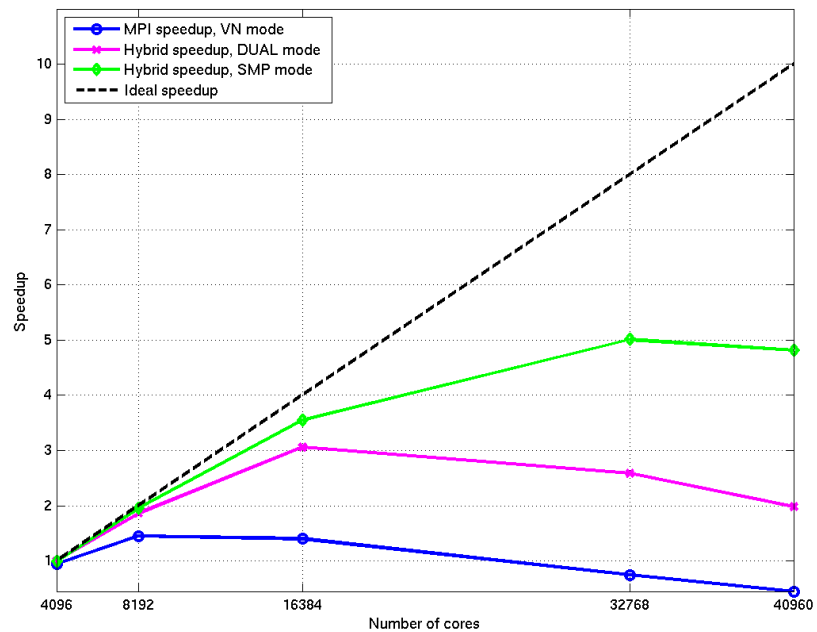
Results using 10 racks on Babel - Strong Scaling

Results for the nx = 400000, ny = 4000 domain

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

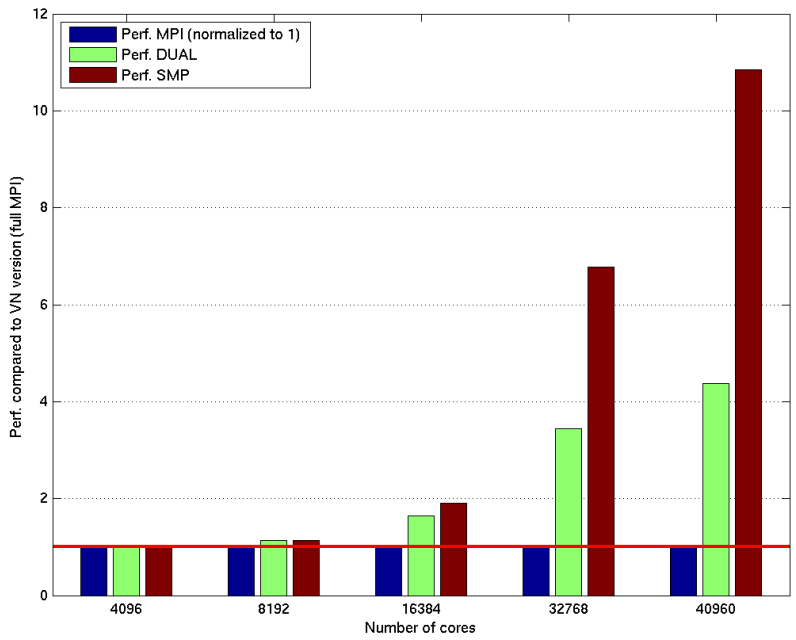


Scalability up to 40960 cores

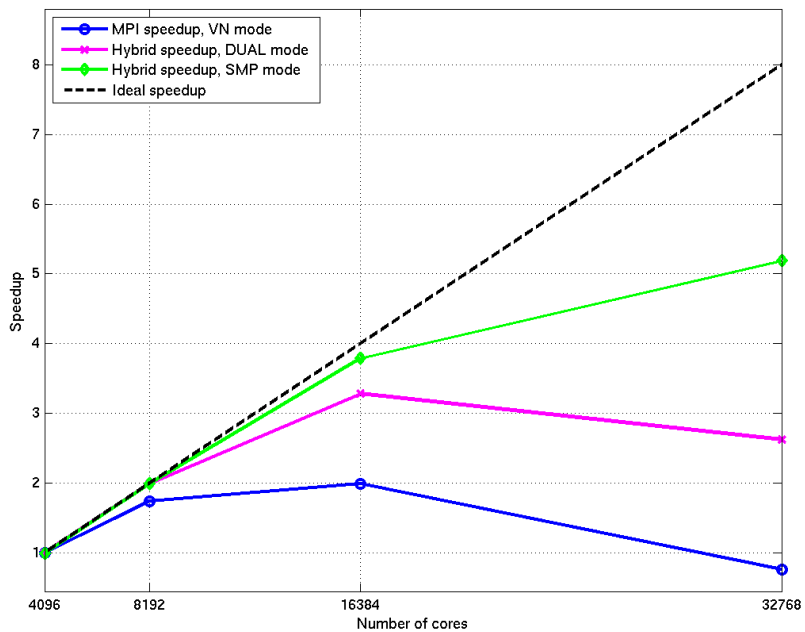
Results using 10 racks on Babel - Strong Scaling

Results for the nx = 40000, ny = 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

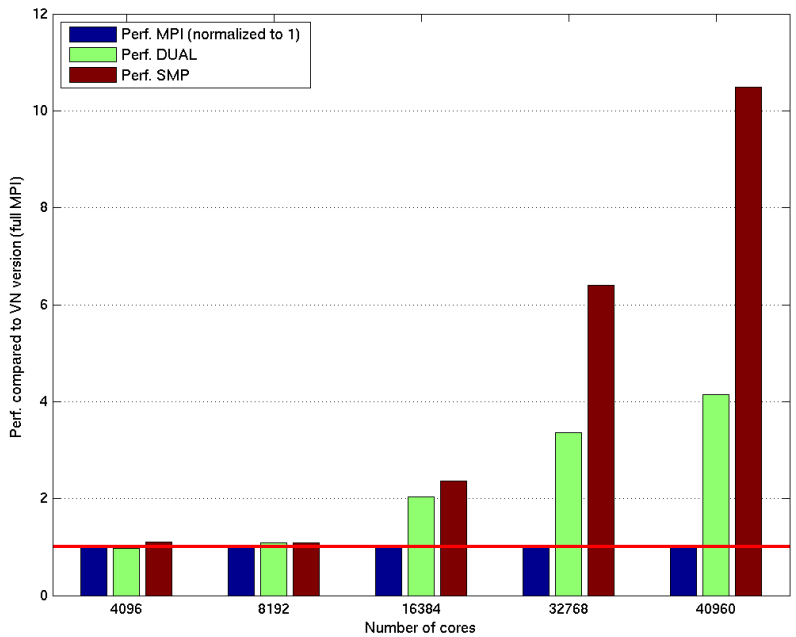


Scalability up to 40960 cores

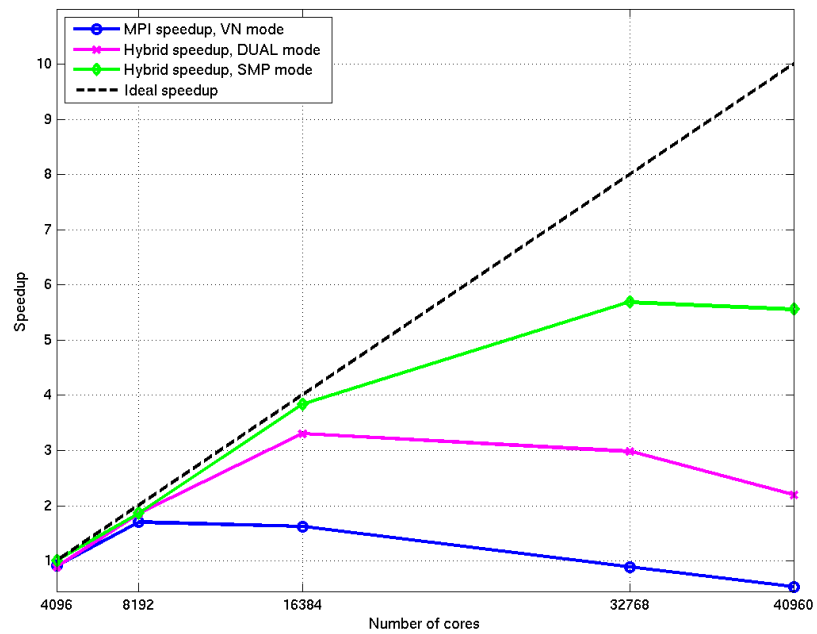
Results using 10 racks on Babel - Strong Scaling

Results for the nx = 4000, ny = 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

Results using 10 racks on Babel - Strong Scaling

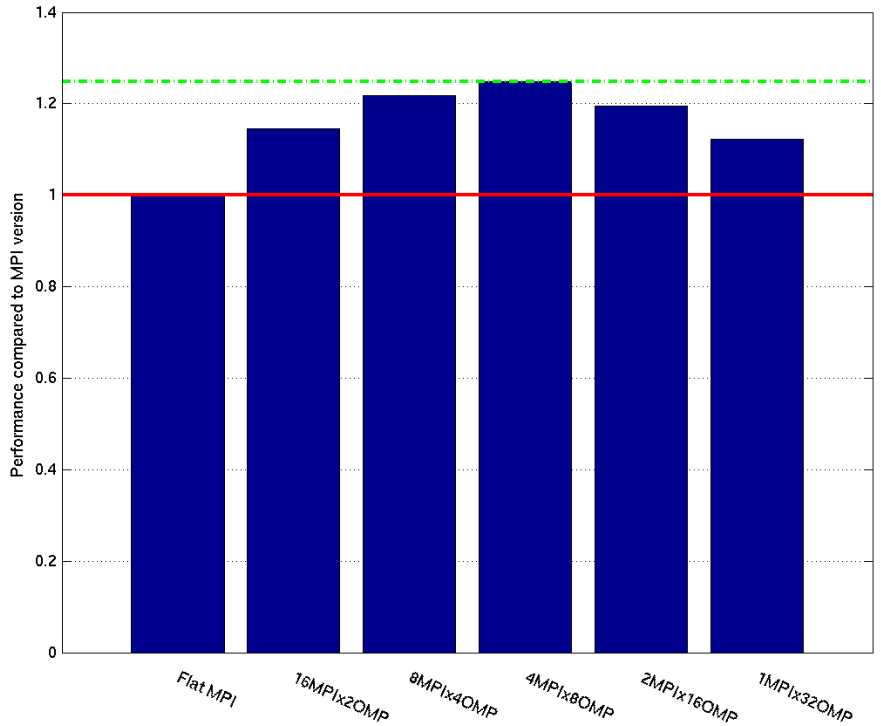
Interpretation of results

- The results of strong scaling, obtained by using up to 40960 computation cores, are very interesting. Here again, new phenomena emerge with this high number of cores.
- The scalability of the flat MPI version shows its limits very quickly. It has difficulty to scale up to 8192 cores and then it begins to collapse.
- As we expected, the DUAL hybrid version, but even more the SMP version, behave very well up to 16384 cores, with a perfectly linear acceleration. The SMP version continues to scale (non-linearly) up to 32768 cores; beyond this, the performances are no longer improved.
- In strong scaling, the scalability limit of the flat MPI version is 8192 cores, whereas that of the SMP hybrid version is 32768 cores. We find here a factor of 4 which corresponds to the number of cores in the BG/P node !
- The best hybrid version (32768 cores) is between 2.6 and 3.5 times faster than the best pure MPI version (8192 cores).
- It is clear that with this type of parallelization method (i.e. domain decomposition), scaling (here over 10K cores) requires recourse to hybrid parallelization. It is not enough to use MPI alone!

Results on Two Vargas Nodes

Results for the nx = 100000, ny = 1000 domain

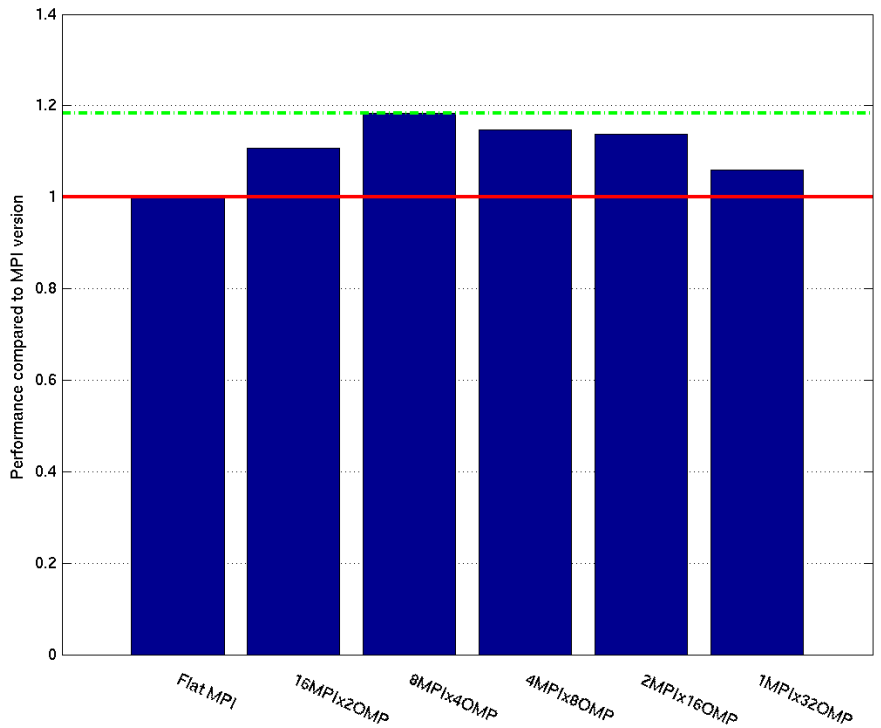
MPI x OMP per node	Time (s)	
	Mono	64 cores
32 x 1	361.4	7.00
16 x 2	361.4	6.11
8 x 4	361.4	5.75
4 x 8	361.4	5.61
2 x 16	361.4	5.86
1x 32	361.4	6.24



- 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 nx = 10000, ny = 10000 Domain

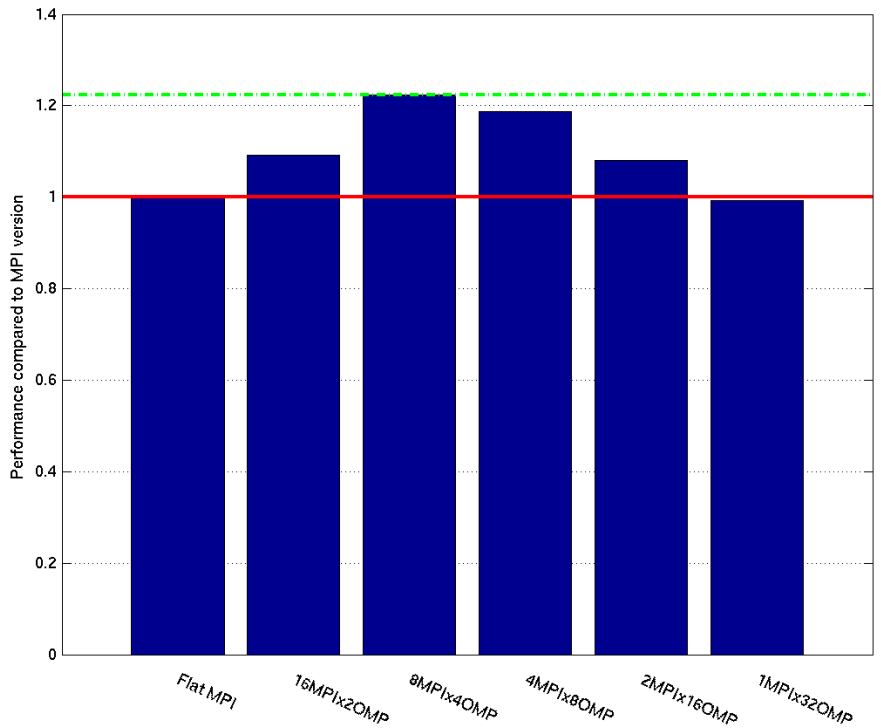
MPI x OMP per node	Time(s)	
	Mono	64 cores
32 x 1	449.9	6.68
16 x 2	449.9	6.03
8 x 4	449.9	5.64
4 x 8	449.9	5.82
2 x 16	449.9	5.87
1 x 32	449.9	6.31



- 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 nx = 1000, ny = 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 ditsribution.

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

Sommaire I

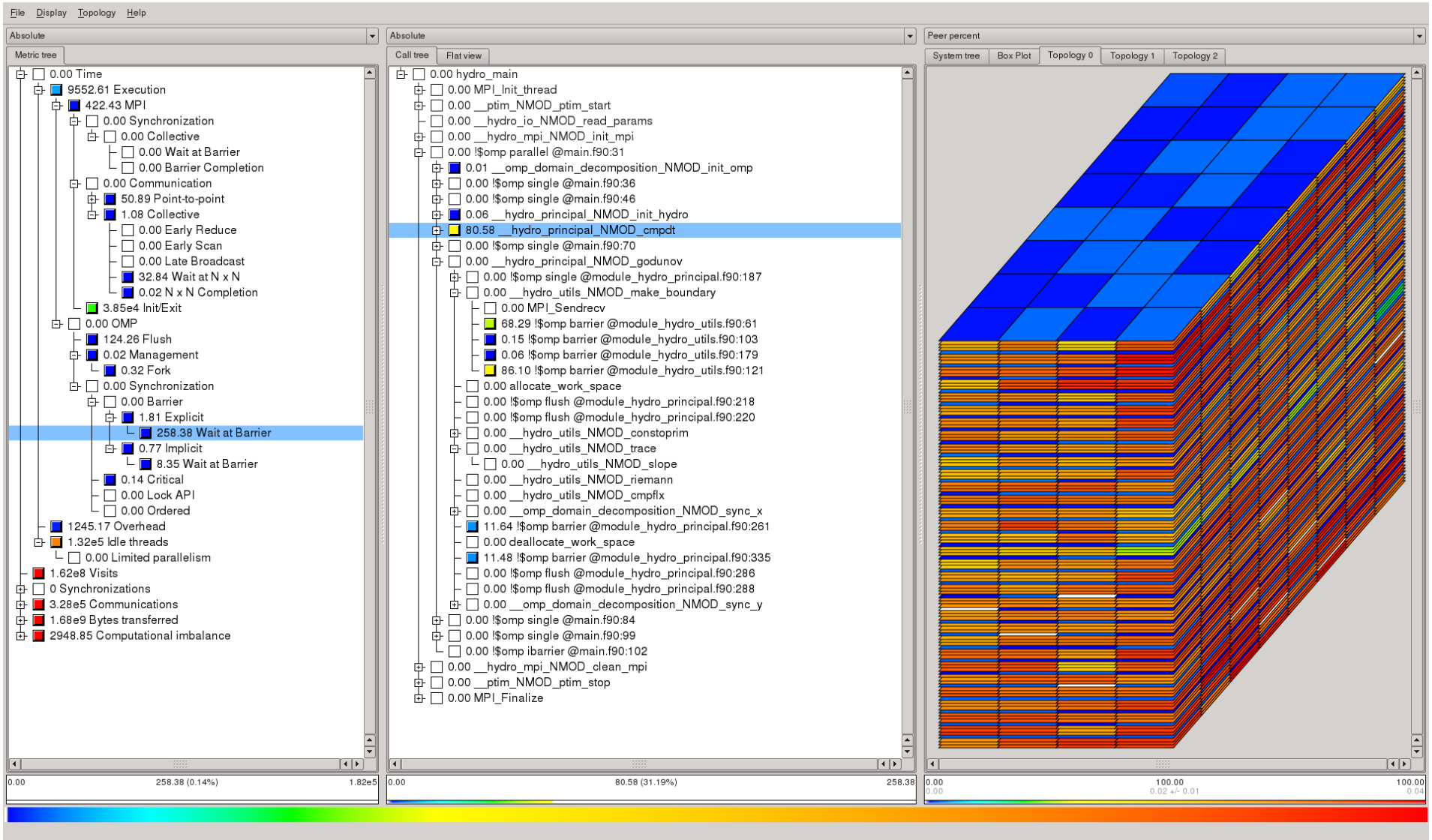
Description

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

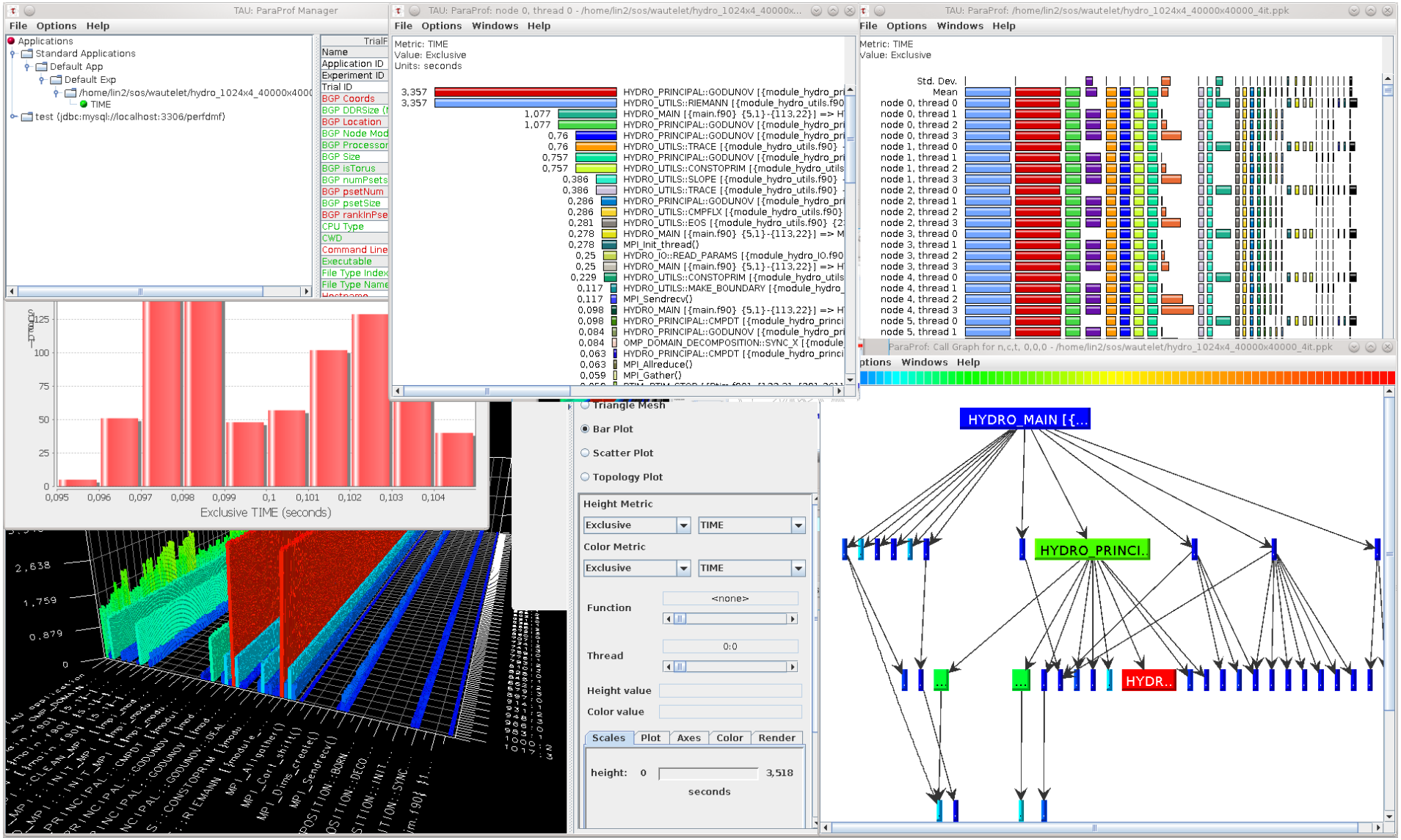
- Compile your application with *skin f90* (or other compiler).
- Execute with *scan mpirun*. Use the option *-t* for the trace mode.
- Visualize the results with *square*.


File Display Topology Help
Absolute
Metric tree
0.00 Time
9552.61 Execution
422.43 MPI
0.00 Synchronization
0.00 Collective
0.00 Wait at Barrier
0.00 Barrier Completion
0.00 Communication
50.89 Point-to-point
1.08 Collective
0.00 Early Reduce
0.00 Early Scan
0.00 Late Broadcast
32.84 Wait at N x N
0.02 N x N Completion
3.85e4 Init/Exit
0.00 OMP
124.26 Flush
0.02 Management
0.32 Fork
0.00 Synchronization
0.00 Barrier
1.81 Explicit
258.38 Wait at Barrier
0.77 Implicit
8.35 Wait at Barrier
0.14 Critical
0.00 Lock API
0.00 Ordered
1245.17 Overhead
1.32e5 Idle threads
0.00 Limited parallelism
1.62e8 Visits
0 Synchronizations
3.28e5 Communications
1.68e9 Bytes transferred
2948.85 Computational imbalance
0.00
258.38 (0.14%)
1.82e5
Absolute
Call tree
Flat view
0.00 hydro_main
0.00 MPI_Init_thread
0.00 __ptim_NMOD_ptim_start
0.00 __hydro_io_NMOD_read_params
0.00 __hydro_mpi_NMOD_init_mpi
0.00 !$omp parallel @main.f90:31
0.01 __omp_domain_decomposition_NMOD_init_omp
0.00 !$omp single @main.f90:36
0.00 !$omp single @main.f90:46
0.06 __hydro_principal_NMOD_init_hydro
80.58 __hydro_principal_NMOD_cmpdt
0.00 !$omp single @main.f90:70
0.00 __hydro_principal_NMOD_godunov
0.00 !$omp single @module_hydro_principal.f90:187
0.00 __hydro_utils_NMOD_make_boundary
0.00 MPI_Sendrecv
68.29 !$omp barrier @module_hydro_utils.f90:61
0.15 !$omp barrier @module_hydro_utils.f90:103
0.06 !$omp barrier @module_hydro_utils.f90:179
86.10 !$omp barrier @module_hydro_utils.f90:121
0.00 allocate_work_space
0.00 !$omp flush @module_hydro_principal.f90:218
0.00 !$omp flush @module_hydro_principal.f90:220
0.00 __hydro_utils_NMOD_constoprim
0.00 __hydro_utils_NMOD_trace
0.00 __hydro_utils_NMOD_slope
0.00 __hydro_utils_NMOD_riemann
0.00 __hydro_utils_NMOD_cmpflx
0.00 __omp_domain_decomposition_NMOD_sync_x
11.64 !$omp barrier @module_hydro_principal.f90:261
0.00 deallocate_work_space
11.48 !$omp barrier @module_hydro_principal.f90:335
0.00 !$omp flush @module_hydro_principal.f90:286
0.00 !$omp flush @module_hydro_principal.f90:288
0.00 __omp_domain_decomposition_NMOD_sync_y
0.00 !$omp single @main.f90:84
0.00 !$omp single @main.f90:99
0.00 !$omp ibarrier @main.f90:102
0.00 __hydro_mpi_NMOD_clean_mpi
0.00 __ptim_NMOD_ptim_stop
0.00 MPI_Finalize
0.00
80.58 (31.19%)
258.38
Peer percent
System tree
Box Plot
Topology 0
Topology 1
Topology 2
0.00
100.00
0.02 +/- 0.01
100.00
0.04

Description

TAU is a graphical tool for performance analysis of parallel applications. Prinicipal 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)


TAU: ParaProf Manager
File Options Help
Applications
Standard Applications
Default App
Default Exp
TIME
test (jdbc:mysql://localhost:3306/perfdmf)
TAU: ParaProf: node 0, thread 0
Metric: TIME
Value: Exclusive
Units: seconds
TAU: ParaProf: /home/lin2/sos/wautelet/hydro_1024x4_40000x40000_4it.ppk
Std. Dev.
Mean
Exclusive TIME (seconds)
Triangle Mesh
Bar Plot
Scatter Plot
Topology Plot
Height Metric
Color Metric
Function
Thread
Height value
Color value
Scales
Plot
Axes
Color
Render
HYDRO_MAIN
HYDRO_PRINCI..
HYDR..

TotalView

Description

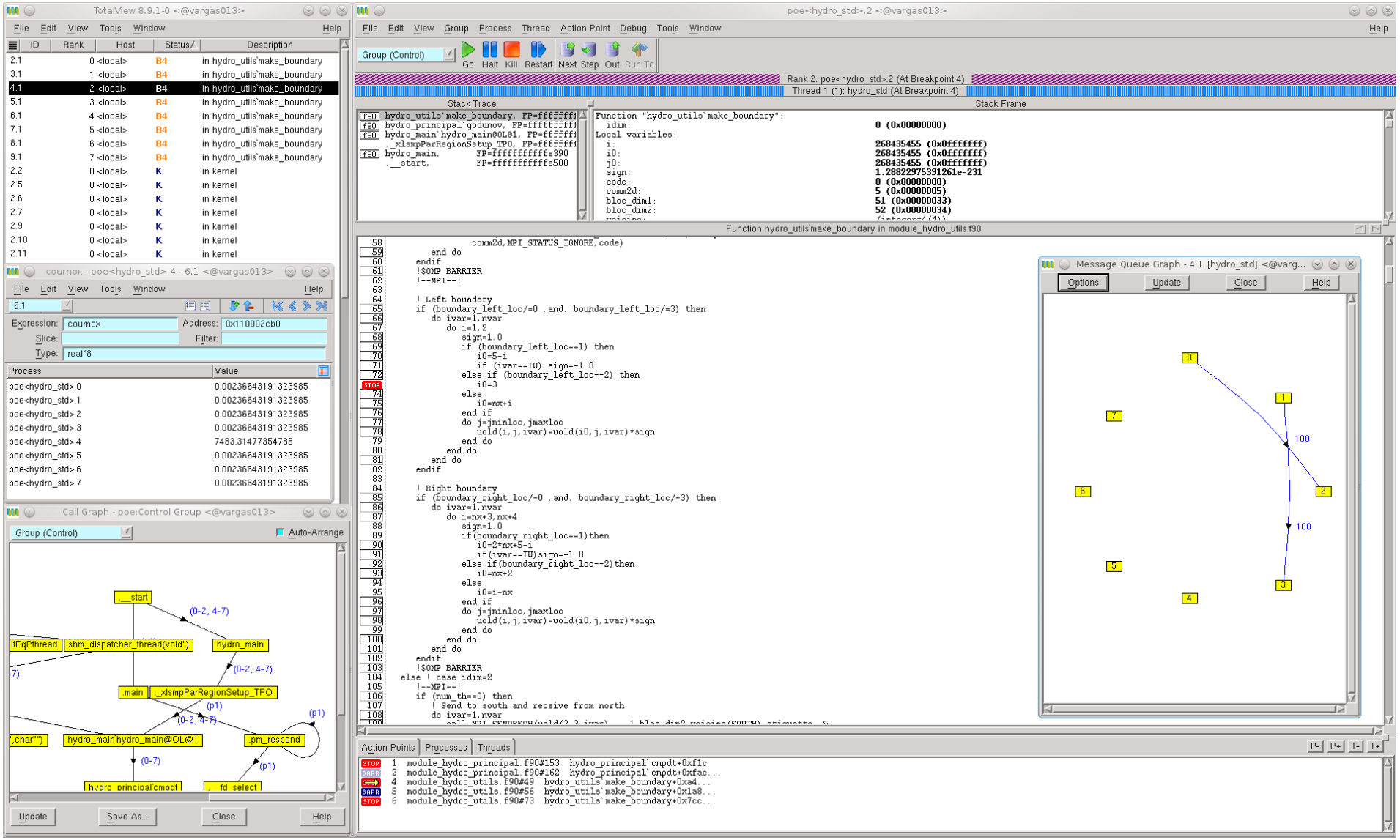
TotalView is a graphical debugging tool for parallel applications. Key features:

- 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 ion.

TotalView


TotalView 8.9.1-0 <@vargas013>
File Edit View Tools Window Help
ID Rank Host Status/ Description
2.1 0 <local> B4 in hydro_utils`make_boundary
3.1 1 <local> B4 in hydro_utils`make_boundary
4.1 2 <local> B4 in hydro_utils`make_boundary
5.1 3 <local> B4 in hydro_utils`make_boundary
6.1 4 <local> B4 in hydro_utils`make_boundary
7.1 5 <local> B4 in hydro_utils`make_boundary
8.1 6 <local> B4 in hydro_utils`make_boundary
9.1 7 <local> B4 in hydro_utils`make_boundary
2.2 0 <local> K in kernel
2.5 0 <local> K in kernel
2.6 0 <local> K in kernel
2.7 0 <local> K in kernel
2.9 0 <local> K in kernel
2.10 0 <local> K in kernel
2.11 0 <local> K in kernel
cournox - poe<hydro_std>.4 - 6.1 <@vargas013>
Expression: cournox Address: 0x110002cb0
Slice: Filter:
Type: real*8
Process Value
poe<hydro_std>.0 0.00236643191323985
poe<hydro_std>.1 0.00236643191323985
poe<hydro_std>.2 0.00236643191323985
poe<hydro_std>.3 0.00236643191323985
poe<hydro_std>.4 7483.31477354788
poe<hydro_std>.5 0.00236643191323985
poe<hydro_std>.6 0.00236643191323985
poe<hydro_std>.7 0.00236643191323985
Call Graph - poe:Control Group <@vargas013>
Group (Control) Auto-Arrange
__start
(0-2, 4-7)
ifEqPthread
shm_dispatcher_thread(void*)
hydro_main
(0-2, 4-7)
.main
_xlsmpParRegionSetup_TPO
(p1)
(0-2, 4-7)
(p1)
,char**)
hydro_main`hydro_main@OL@1
.pm_respond
(0-7)
(p1)
hydro_principal`cmpdt
fd_select
Update Save As... Close Help
poe<hydro_std>.2 <@vargas013>
File Edit View Group Process Thread Action Point Debug Tools Window Help
Group (Control)
Go Halt Kill Restart Next Step Out Run To
Rank 2: poe<hydro_std>.2 (At Breakpoint 4)
Thread 1 (1): hydro_std (At Breakpoint 4)
Stack Trace
f90 hydro_utils`make_boundary, FP=fffffff
f90 hydro_principal`godunov, FP=fffffff
f90 hydro_main`hydro_main@OL@1, FP=fffffff
._xlsmpParRegionSetup_TPO, FP=fffffff
f90 hydro_main, FP=fffffffffffe390
.__start, FP=fffffffffffe500
Stack Frame
Function "hydro_utils`make_boundary":
idim: 0 (0x00000000)
Local variables:
i: 268435455 (0x0fffffff)
i0: 268435455 (0x0fffffff)
j0: 268435455 (0x0fffffff)
sign: 1.2882297539l261e-231
code: 0 (0x00000000)
comm2d: 5 (0x00000005)
bloc_dim1: 51 (0x00000033)
bloc_dim2: 52 (0x00000034)
Function hydro_utils`make_boundary in module_hydro_utils.f90
comm2d,MPI_STATUS_IGNORE,code)
end do
endif
!$OMP BARRIER
!--MPI--!
! Left boundary
if (boundary_left_loc/=0 .and. boundary_left_loc/=3) then
do ivar=1,nvar
do i=1,2
sign=1.0
if (boundary_left_loc==1) then
i0=5-i
if (ivar==IU) sign=-1.0
else if (boundary_left_loc==2) then
i0=3
else
i0=nx+i
end if
do j=jminloc,jmaxloc
uold(i,j,ivar)=uold(i0,j,ivar)*sign
end do
end do
end do
endif
! Right boundary
if (boundary_right_loc/=0 .and. boundary_right_loc/=3) then
do ivar=1,nvar
do i=nx+3,nx+4
sign=1.0
if(boundary_right_loc==1)then
i0=2*nx+5-i
if(ivar==IU)sign=-1.0
else if(boundary_right_loc==2)then
i0=nx+2
else
i0=i-nx
end if
do j=jminloc,jmaxloc
uold(i,j,ivar)=uold(i0,j,ivar)*sign
end do
end do
end do
endif
!$OMP BARRIER
else ! case idim=2
!--MPI--!
if (num_th==0) then
! Send to south and receive from north
do ivar=1,nvar
Message Queue Graph - 4.1 [hydro_std] <@varg...
Options Update Close Help
0
1
7
100
6
2
100
5
3
4
Action Points Processes Threads
P- P+ T- T+
1 module_hydro_principal.f90#153 hydro_principal`cmpdt+0xf1c
2 module_hydro_principal.f90#162 hydro_principal`cmpdt+0xfac...
4 module_hydro_utils.f90#49 hydro_utils`make_boundary+0xa4...
5 module_hydro_utils.f90#56 hydro_utils`make_boundary+0x1a8...
6 module_hydro_utils.f90#73 hydro_utils`make_boundary+0x7cc...

Hands-on Exercises

Objective

Synchronize all of the OpenMP threads located on the different MPI processes.

Statement

You are asked to complete the *barrier_hybride.f90* file so that all the OpenMP threads on the different MPI processes would be synchronized during a call to the *barrierMPIOMP* subroutine.

Objective

Measure the network sustained bandwidth between two nodes.

Statement

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

- In the first version, you will use the `MPI_THREAD_FUNNELED` level of thread support (the application can launch several threads per process but only the main thread can make MPI calls). In this case, the number of parallel communication flows will be equal to the number of MPI processes per node.
- In the second version, you will use the `MPI_THREAD_MULTIPLE` level of thread support (entirely multithreaded without restrictions) with one MPI process per node. In this case, the number of parallel communication flows will be equal to the number of OpenMP threads which participate in the communications.

TP3 — HYDRO, from MPI to hybrid MPI and OpenMP version

Objective

Parallelize an application using MPI and OpenMP.

Statement

You are asked to start with the HYDRO MPI parallel version application.

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

Appendices

`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.

Presentation of Poisson3D

Poisson3D is an application which resolves Poisson's equation on the cubic domain[0,1]x[0,1]x[0,1] using a finite difference method and a Jacobi solver.

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

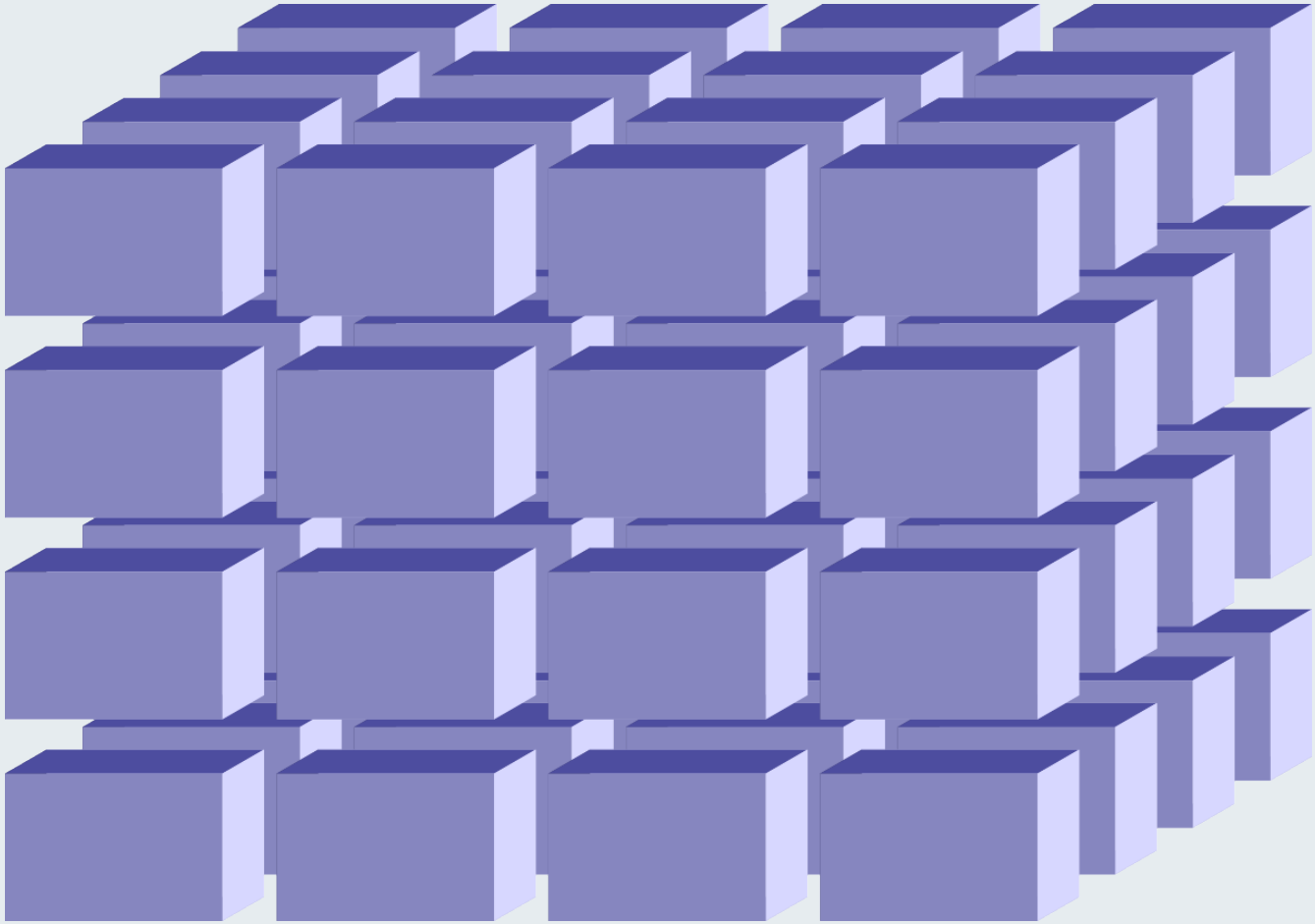
Solver

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

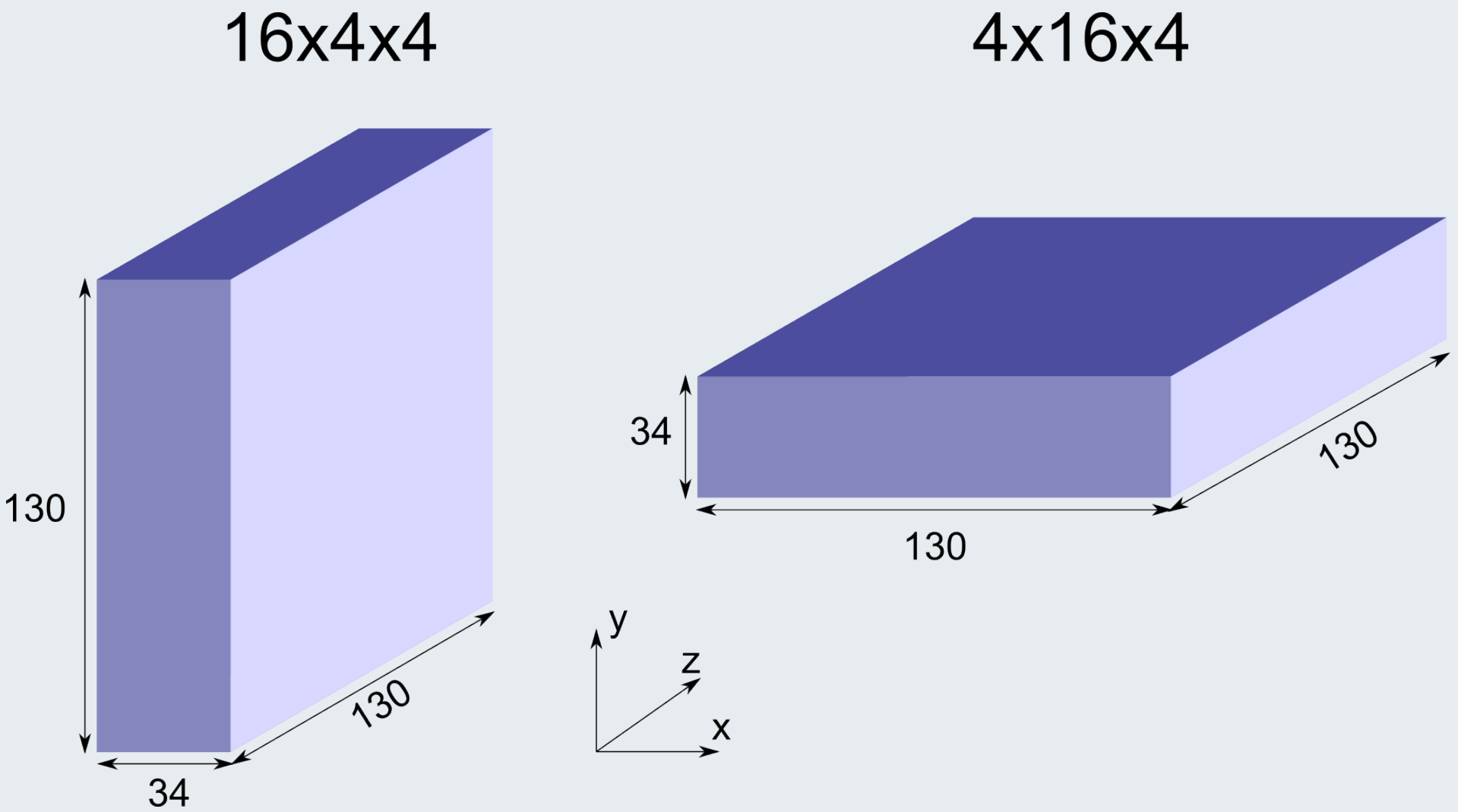
Cartesian topology and cache use

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)



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*x*4*x*4 topology into $(34, 130, 130)$ and in the 4*x*16*x*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 4x16x4 than in 16x4x4; this explains a big part of the time difference.

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

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.

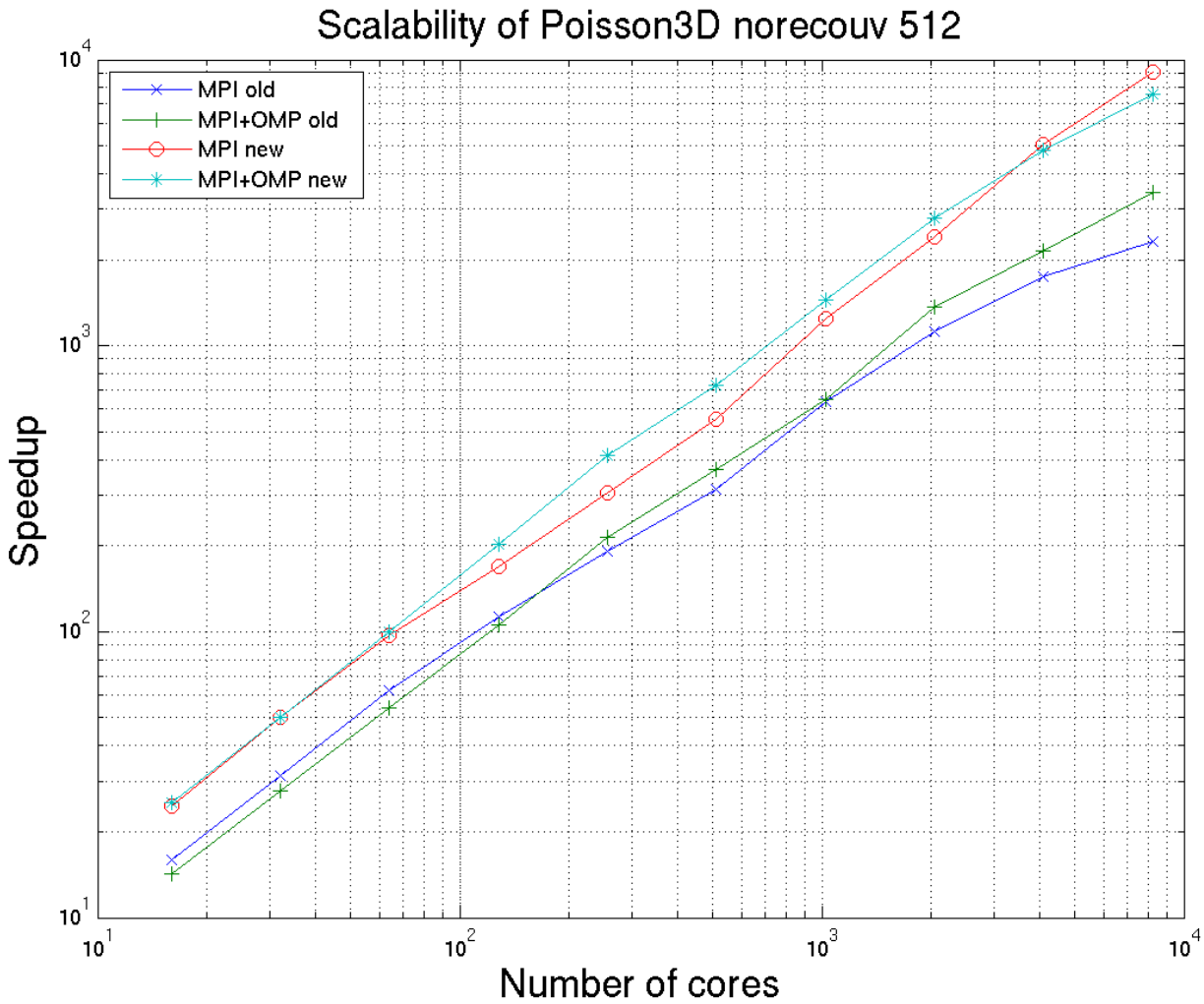
MPI communications

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 *read* (TiB)	Torus *send* (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.