

CEMRACS - Hybrid
D. Lecas

20 juillet 2012

1	Hybrid programming	3
---	--------------------------	---

1 – Hybrid programming

1.1 – Definitions

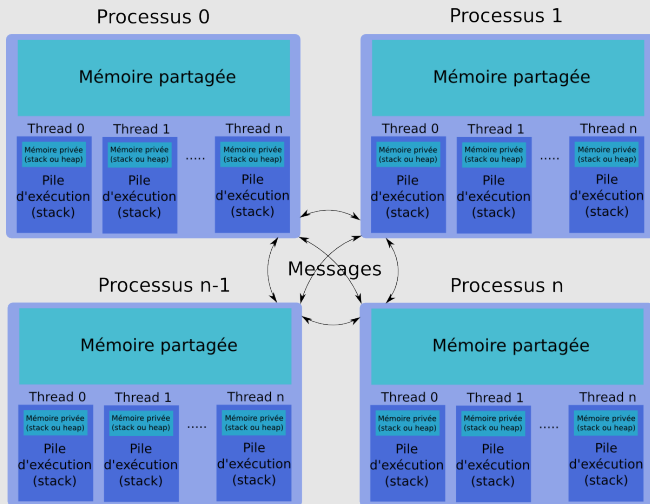
Definitions

- Hybrid parallel programming consists of mixing some parallel programming paradigms in order to benefit from the advantages of the different approaches.
- In general, MPI is used on processes and another paradigm (OpenMP, pthreads, langage PGAS, UPC...) inside each process.
- In this training course, we will deal exclusively with the use of OpenMP with MPI.

1 – Hybrid programming

1.1 – Definitions

Schematic Drawing



1 – Hybrid programming

1.2 – *Raison d'Être* for Hybrid Programming

Advantages of Hybrid Programming (1)

- Better scalability by a reduction of both the number of MPI messages, and the number of processes involved in collective communications (`MPI_Alltoall` is not very scalable) and by a better load balancing.
- Optimization of the total memory consumption (through the OpenMP shared-memory approach, savings in replicated data in the MPI processes and in the used memory by the MPI library itself).
- Reduction of the footprint memory when the size of some data structures depends directly on the number of MPI processes.
- It can remove some algorithmic limitations (maximum decomposition in one direction for example).
- Performance-enhancement of some algorithms by reducing the number of MPI processes (fewer domains = a better preconditioner if we drop the contributions of other domains).
- Fewer simultaneous accesses in I/O and larger average record size. This causes less load on the meta-data servers with requests of more suitable size. The potential savings on a massively parallel application can be significant.

1 – Hybrid programming

1.2 – *Raison d'Être* for Hybrid Programming

Advantages of Hybrid Programming (2)

- Fewer files to manage than if we use an approach where the number of files is proportional to the number of MPI processes (an approach not at all recommended in a context of massive parallelism).
- 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 computation and communication successive phases. The more the granularity of a code is significant, the more it is scalable. Compared to the pure MPI approach, the hybrid approach increases significantly the granularity and consequently the scalability of codes.

Disadvantages of Hybrid Programming

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

1 – Hybrid programming

1.3 – Applications That Can Benefit From It

Applications That Can Benefit From It

- Codes having limited MPI scalability (through the use of `MPI_Alltoall` for example).
- Codes requiring dynamic load balancing.
- Codes limited by memory size and having many replicated data between MPI processes or having data structures that depend on the number of processes for their dimension.
- Inefficient MPI implementation library for intra-node communications.
- ...

1 – Hybrid programming

1.4 – MPI and Multithreading

Thread Support in MPI

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

- The standard does not impose any minimum level of thread support. Some architectures and/or implementations may not have therefore any support for multithreaded applications.
- The ranks identify only the processes, not the threads which 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 sent message to this process (depending on the level of support).
- The blocking calls block only the calling thread.
- The `MPI_Finalize` call has to be made by the thread that called `MPI_Init_thread` and when all the threads of the process have finished their MPI calls.

1 – Hybrid programming

1.4 – MPI and Multithreading

MPI_Init_thread

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

The required level of support is provided in the *required* variable. The actually obtained level (that can be less than required) is available in *provided*.

- `MPI_THREAD_SINGLE` : only one thread per process can operate
- `MPI_THREAD_FUNNELED` : the application can launch multiple threads per process, but only the main thread (the one which made the `MPI_Init_thread` call) can make MPI calls
- `MPI_THREAD_SERIALIZED` : all the threads can make MPI calls, but only one at a time
- `MPI_THREAD_MULTIPLE` : completely multithreaded without restrictions

1 – Hybrid programming

1.4 – MPI and Multithreading

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` returns if the calling thread is the main thread or not :

```
int MPI_Is_thread_main(int *flag)
MPI_IS_THREAD_MAIN(FLAG, IERROR)
```

1 – Hybrid programming

1.4 – MPI and Multithreading

Restrictions on MPI collective calls

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

- This implies that it is forbidden to have multiple threads per process making calls on the same communicator without ensuring that the calls are in the same order on each of the processes.
- We can not have at any given time two threads that each do a collective call on the same communicator (the calls can be different or not).
- For example, if multiple threads call `MPI_Barrier` with `MPI_COMM_WORLD`, the application may hang (this is easily verified on Babel and Vargas).
- 2 threads calling each a `MPI_Allreduce` (with the same operation or not) may get false results.
- 2 different collective calls can not be used (a `MPI_Reduce` and a `MPI_Bcast` for example).

1 – Hybrid programming

1.4 – MPI and Multithreading

Restrictions on MPI collective calls

To avoid these difficulties, there are several possibilities :

- Impose the order of the calls by synchronizing the different threads within each MPI process,
- Use different communicators for each collective call,
- Do collective calls only on 1 thread by process.

Note that, in `MPI_THREAD_SERIALIZED` mode, the problem should not exist because the user must ensure that at any given time at most one thread per process is involved in a MPI call (collective or not). Be careful, however, the order of the calls must be respected.

1 – Hybrid programming

1.5 – MPI and OpenMP

Implications of the Different Support Levels

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

- `MPI_THREAD_SINGLE` : OpenMP cannot be used
- `MPI_THREAD_FUNNELED` : the MPI calls have to be made outside of the OpenMP *parallel* regions or by the main thread (the one which made the `MPI_Init_thread` call)
- `MPI_THREAD_SERIALIZED` : in the OpenMP parallel regions, the MPI calls have to be made in *critical* sections (if necessary to ensure only one MPI call at a time)
- `MPI_THREAD_MULTIPLE` : no restriction

1 – Hybrid programming

1.5 – MPI and OpenMP

State of Current Implementations

<i>Implementation</i>	<i>Supported Level</i>	<i>Remarks</i>
MPICH2	MPI_THREAD_MULTIPLE	
OpenMPI	MPI_THREAD_MULTIPLE	Must be compiled with <i>-enable-mpi-threads</i>
IBM BlueGene/P	MPI_THREAD_MULTIPLE	
IBM Power	MPI_THREAD_MULTIPLE	
BullxMPI	MPI_THREAD_SERIALIZED	
Intel - MPI	MPI_THREAD_MULTIPLE	Use <i>-mt_mpi</i>
SGI - MPT	MPI_THREAD_MULTIPLE	Use <i>-lmpi_mt</i>

1 – Hybrid programming

1.6 – Adequacy to the Architecture : Memory Savings

Why Memory Savings ?

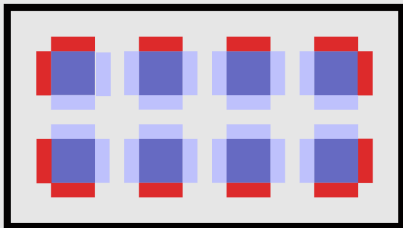
- The hybrid programming allows optimizing the code to the target architecture. The latter is generally composed of shared-memory nodes (SMP) linked by an interconnect network. The interest of the shared memory inside a node is that it is not necessary to duplicate data in order to exchange them. Every thread can access (read /write) SHARED data.
- The ghost or halo cells, introduced in order to simplify the programming of MPI codes using domain decomposition, are no longer mandatory within the SMP node. Only the ghost cells associated with the inter-node communications are mandatory.
- The savings associated with the absence of intra-node ghost cells is far from being negligible. It depends heavily on the order of the method, on the domain type (2D or 3D), on the domain decomposition (on one or multiple dimensions) and on the number of cores of the SMP node.
- The footprint memory of system buffers associated with MPI is not negligible and increases with the number of processes. For example, for an Infiniband network with 65000 MPI processes, the footprint memory of system buffers reaches 300MB per process, almost 20TB in total!

1 – Hybrid programming

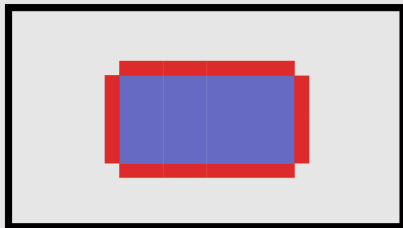
1.6 – Adequacy to the Architecture : Memory Savings


Exemple : 2D domain, décomposition on both directions


Noeud SMP à 8 cœurs, décomposition de domaine flat MPI




Noeud SMP à 8 cœurs, décomposition de domaine hybride



 Mailles fantômes intra-noeud

 Mailles fantômes inter-noeud

 Sous-domaine associé à un processus MPI

1 – Hybrid programming

1.6 – Adequacy to the Architecture : Memory Savings

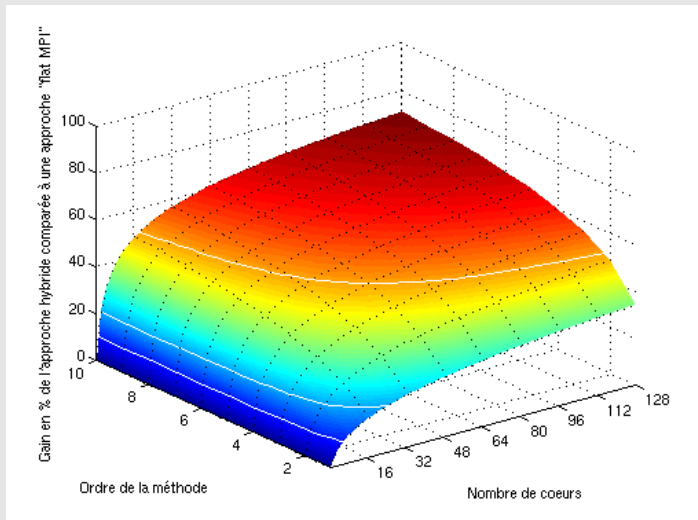
Extrapolation on a 3D Domain

- Let us try to calculate, according to the order of the numerical method (h) and the number of cores of the SMP node (c), the relative memory savings obtained by using a hybrid version instead of a flat MPI version of a parallelized 3D code by a technique of domain decomposition following its three dimensions.
- 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 64 GBi of shared-memory on the node.
- The simulation result is presented in the following slide. The isovalues 10%, 20% and 50% are represented by white lines on the isosurface.

1 - Hybrid programming

1.6 - Adequacy to the Architecture : Memory Savings

Extrapolation on a 3D Domain



1 – Hybrid programming

1.6 – Adequacy to the Architecture : Memory Savings

Effective Memory Savings On Some Real Application Codes

- Source : « *Mixed Mode Programming on HECToR* », Anastasios 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 32GiB of memory, for a total of 44544 cores, 58TiB of memory and a peak performance of 373Tflop/s.
- Results (the memory per node is expressed in MiB) :

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

1 – Hybrid programming

1.6 – Adequacy to the Architecture : Memory Savings

Conclusion on Memory Savings

- Too often, this aspect is forgotten when we talk about hybrid programming.
- However, the potential gains are very significant and could be exploited to increase the size of the problems to be simulated!
- For the following reasons, the gap between the MPI and hybrid approaches will grow more rapidly for the next generation of machines :
 - ① The increase in the total number of cores,
 - ② The rapid increase in the number of available cores within a node as well as the general use of hyperthreading or SMT (the possibility to run simultaneously multiple threads on one core),
 - ③ The general use of high-order numerical methods (nearly free computational cost thanks to hardware accelerators)
- This will make the transition to hybrid programming almost mandatory...

1 – Hybrid programming

1.7 – Case Study : HYDRO

Presentation of the HYDRO Code (1)

- It 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 to the interfaces by a Godunov method.
- In recent years, in the framework of the technology watch of IDRIS, this code serves as a benchmark for the new architectures, from the simple graphics card to the petaflops machine.
- It has grown over the years with new implementations (new languages, new paradigms of parallelization).
- 1500 lines of code in its F90 monoprocessor version.

1 - Hybrid programming

1.7 - Case Study : HYDRO

Presentation of the HYDRO Code (2)

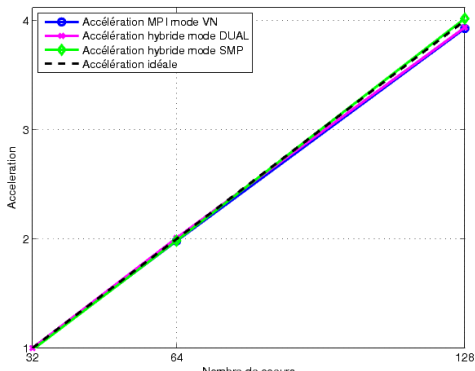
- Today, there are the following hydro versions :
 - Original version, F90 monoprocessor (P.-Fr. Lavallée, R. Teyssier)
 - Monoprocessor C version (G. Colin de Verdière)
 - MPI F90 parallel version (1D P.-Fr. Lavallée, 2D Ph. Wautelet)
 - MPI C parallel version (2D Ph. Wautelet)
 - OpenMP Fine-Grained and Coarse-Grained parallel version F90 (P.-Fr. Lavallée)
 - OpenMP Fine-Grained C parallel version (P.-Fr. Lavallée)
 - MPI2D-OpenMP Fine-Grained and Coarse-Grained hybrid parallel version F90 (P.-Fr. Lavallée, Ph. Wautelet)
 - MPI2D-OpenMP Fine-Grained 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, ...

1 - Hybrid programming

1.7 - Case Study : HYDRO

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

Times (s)	32 <i>cores</i>	64 <i>cores</i>	128 <i>cores</i>
VN Mode	49.12	24.74	12.47
DUAL Mode	49.00	24.39	12.44
SMP Mode	49.80	24.70	12.19

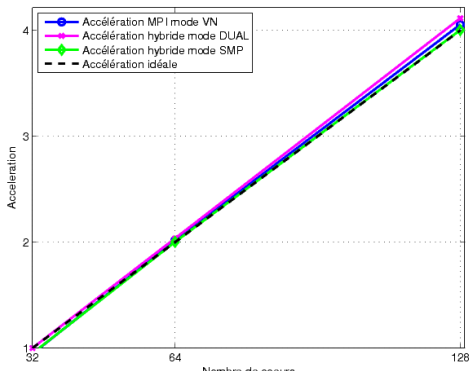


1 - Hybrid programming

1.7 - Case Study : HYDRO

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

Time in (s)	32 <i>cores</i>	64 <i>cores</i>	128 <i>cores</i>
VN Mode	53.14	24.94	12.40
DUAL Mode	50.28	24.70	12.22
SMP Mode	52.94	25.12	12.56

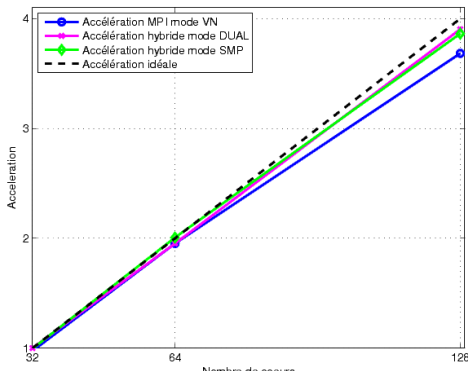


1 - Hybrid programming

1.7 - Case Study : HYDRO

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



1 – Hybrid programming

1.7 – Case Study : HYDRO

Characteristics of the Used Domains for the Weak Scaling

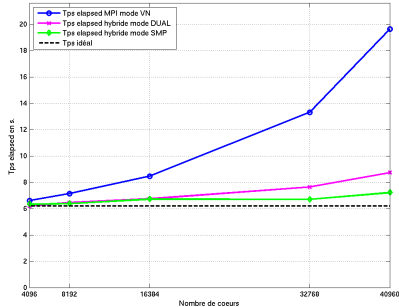
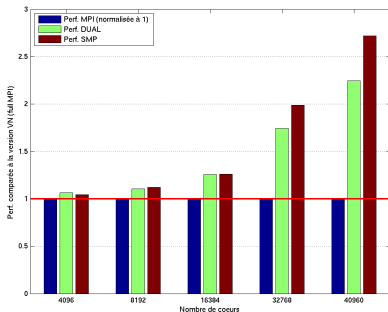
- On 4096 cores, total number of points of the domain : $16 \cdot 10^8$
 - 400000×4000 : stretched domain along the first dimension
 - 40000×40000 : square domain
 - 4000×400000 : stretched domain along the second dimension
- On 8192 cores, total number of points of the domain : $32 \cdot 10^8$
 - 800000×4000 : stretched domain along the first dimension
 - 56568×56568 : square domain
 - 4000×800000 : stretched domain along the second dimension
- On 16384 cores, total number of points of the domain : $64 \cdot 10^8$
 - 1600000×4000 : stretched domain along the first dimension
 - 80000×80000 : square domain
 - 4000×1600000 : stretched domain along the second dimension
- On 32768 cores, total number of points of the domain : $128 \cdot 10^8$
 - 3200000×4000 : stretched domain along the first dimension
 - 113137×113137 : square domain
 - 4000×3200000 : stretched domain along the second dimension
- On 40960 cores, total number of points of the domain : $16 \cdot 10^9$
 - 4000000×4000 : stretched domain along the first dimension

1 – Hybrid programming

1.7 – Case Study : HYDRO

Results for the Stretched Domain along 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

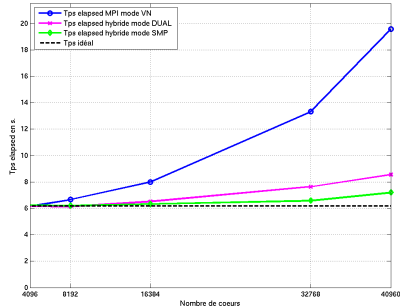
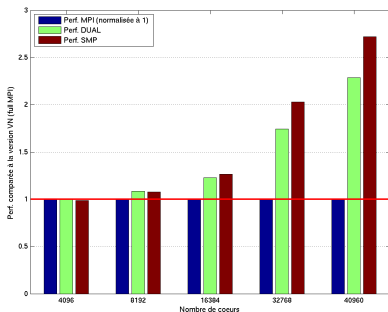


1 - Hybrid programming

1.7 - Case Study : HYDRO

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

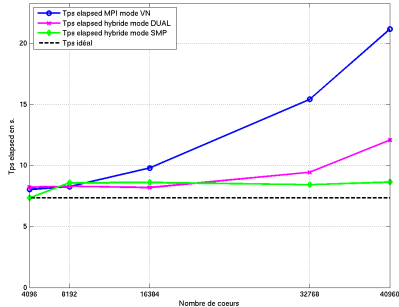
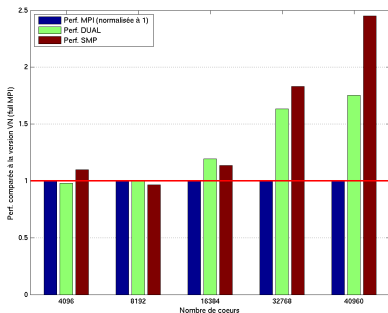


1 - Hybrid programming

1.7 - Case Study : HYDRO

Results for the Stretched Domain along 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



1 – Hybrid programming

1.7 – Case Study : HYDRO

Interpretation of Results

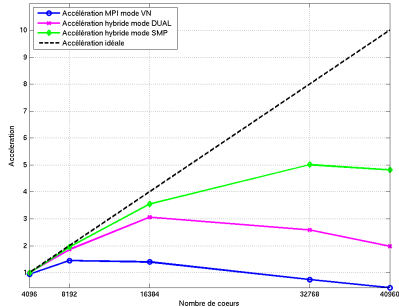
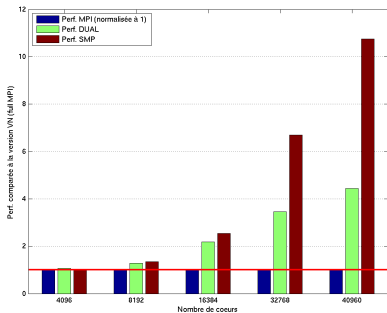
- The results of the weak scaling, obtained by using up to 40960 computation cores, are very interesting. New phenomena emerge with this high number of cores.
- The scalability of the MPI flat version shows its limits. It hardly scales up to 16384 cores, then the elapsed time explode beyond that.
- As we expected, the DUAL hybrid version, but, even more, the SMP version has a very well behavior up to 32768 cores, with nearly constant elapsed time. On 40960 cores, the SMP version shows a very slight additional cost, which becomes significant for the DUAL version.
- In weak scaling, the scalability limit of the MPI flat version is 16384 cores. The limit of the DUAL version is 32768 cores and the limit of the SMP version is not yet 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 with this type of parallelization method (i.e. domain decomposition), the scaling (here over 16K cores) requires clearly the recourse to hybrid parallelization. There is no others solutions only with MPI!

1 - Hybrid programming

1.7 - Case Study : HYDRO

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

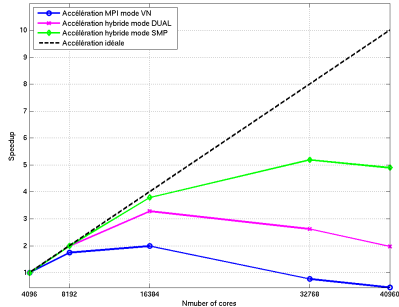
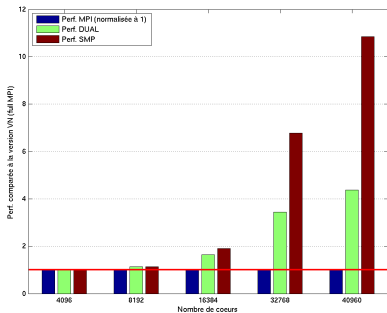


1 - Hybrid programming

1.7 - Case Study : HYDRO

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

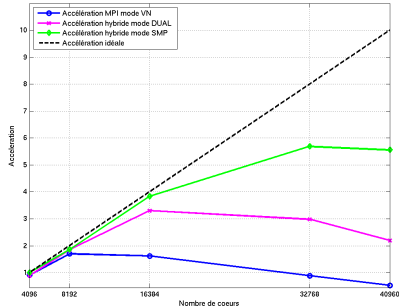
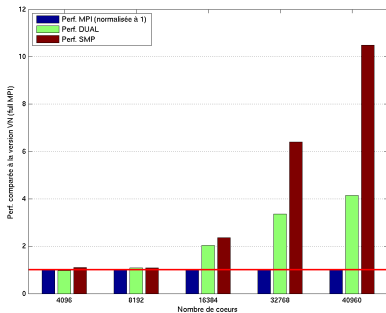


1 - Hybrid programming

1.7 - Case Study : HYDRO

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



1 – Hybrid programming

1.7 – Case Study : HYDRO

Interpretation of results

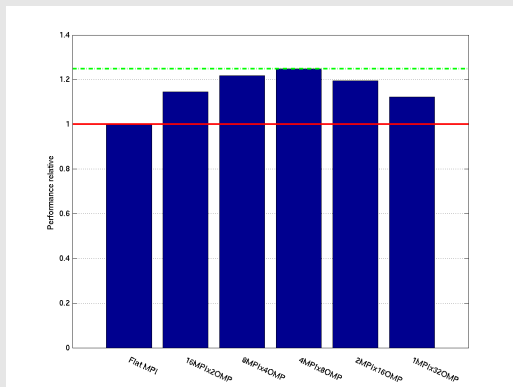
- The results of strong scaling, obtained by using up to 40960 computation cores, are very interesting. Here also, new phenomena emerge with this high number of cores.
- The scalability of the flat MPI version shows very quickly its limits. It hardly scales up to 8192 cores, and then it collapses beyond that.
- As we expected, the DUAL hybrid version, but, even more, the SMP version has a very good behavior up to 16384 cores, with a perfectly linear acceleration. The SMP version continues to scale up (non-linearly) up to 32768 cores. Beyond that, we cannot improve the performances...
- In strong scaling, the scalability limit of the flat MPI version is 8192 cores, whereas the scaling limit of the SMP hybrid version is 32768 cores. We find here the 4 factor which corresponds to the number of cores of 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), the scaling (here over 10K cores) requires clearly the recourse to hybrid parallelization. There is no others solutions only with MPI!

1 - Hybrid programming

1.7 - Case Study : HYDRO

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

MPI x OMP per node	Time (s)	
	Mono	64 <i>cores</i>
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



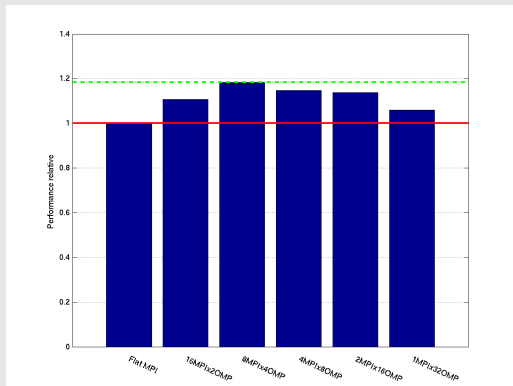
- The hybrid version is always more efficient than the pure MPI version.
- The maximum gain is superior to 20% for the 8MPIx4OMP, 4MPIx8OMP and 2MPIx16OMP distributions.

1 – Hybrid programming

1.7 – Case Study : HYDRO

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

MPI x OMP per node	Time(s)	
	Mono	64 <i>cores</i>
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



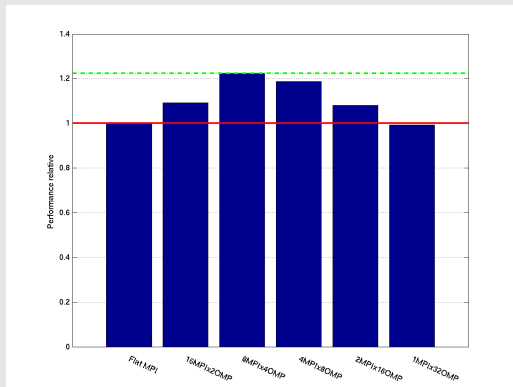
- The hybrid version is always more efficient than the pure MPI version.
- The maximum gain is on the order of 20% for the 8MPIx4OMP distribution.

1 – Hybrid programming

1.7 – Case Study : HYDRO

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.

1 – Hybrid programming

1.7 – Case Study : HYDRO

Interpretation of Results

- Whatever the domain type, the flat MPI version and the hybrid version with one MPI process per node give systematically the least good results.
- The best results are therefore obtained with the hybrid version and a distribution of eight MPI processes per node and four OpenMP threads per MPI process for the two last test cases, and 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) near the one obtained during tests of saturation of the interconnection network (beginning of saturation with eight MPI processes per node).
- Even with a modest size in terms of the number of used cores, it is interesting to notice that the hybrid approach prevails each time, sometimes even with significant gains in performance.
- It is a very good sign and that encourages the increase of the number of used cores.

1 – Hybrid programming

1.7 – Case Study : HYDRO

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), through a better adaptation of the code to the target architecture
 - Gains in terms of scalability, allowing pushing the limit of scalability of a code of an equal factor to the number of cores of the shared-memory node
- These different gains are proportional to the number of cores of the shared-memory node, a number that will increase significantly in the short term (general use of multi-core processors)
- The only durable solution that allows profiting of the next massively parallel architectures (multi-peta, exascale, ...)