High Performance Computing – Course 4: OpenMP – Task-based parallel runtime for shared memory machines

Jonathan Rouzaud-Cornabas

LIRIS / Insa de Lyon – Inria Beagle

Références

- Cours OpenMP, F. Roch (Grenoble)
- Cours OpenMP, J. Chergui & P.-F. Lavallee (IDRIS)
- http://www.openmp.org
- http://www.openmp.org/mp-documents/spec30.pdf
- http://www.idris.fr
- http://ci-tutor.ncsa.illinois.edu/login.php
- Using OpenMP , Portable Shared Memory Model, Barbara Chapman

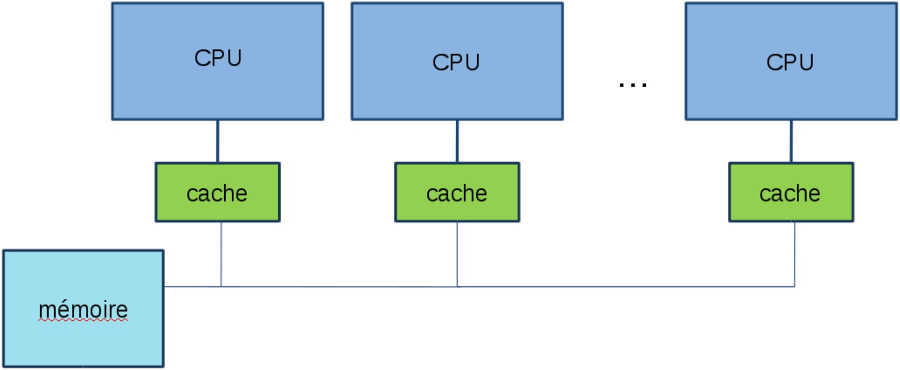
A bit of History

- Multitasking has been around for a long time, but specific libraries / languages
- Increase of multi-core computers (shared memory machines) $\rightarrow$ requirement for a standard
- OpenMP-1 (1997), OpenMP-2 (2000), OpenMP-3 (2008), OpenMP-4.0 (2013), OpenMP-4.5 (2015), OpenMP-5.0 (2018), OpenMP-5.1 (2020+)
- GPU support in OpenMP ($\approx$ 10% of loss)

Multi-tasking programming model on shared memory architecture

- Several tasks are executed in parallel
- Memory is shared (physically or virtually)
- Communication between tasks is done by reads and writes in the shared memory.
- Example: general multi-core processors share a common memory, tasks can be assigned to separate "cores"

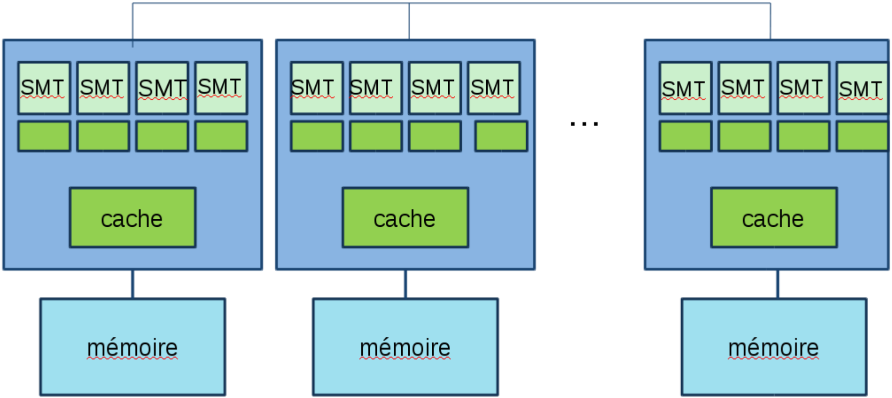
Multitasking programming on UMA architectures



Memory is shared

- Uniform memory access (UMA) architectures
- An inherent problem: memory constraints

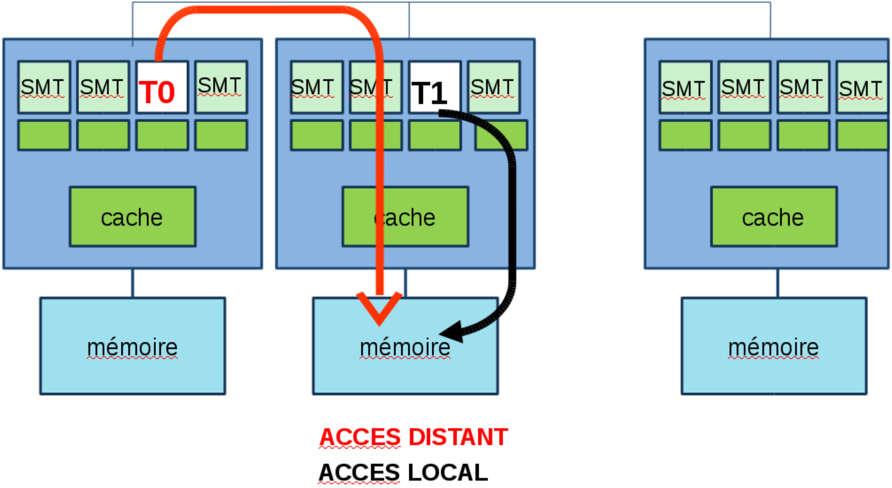
Multitasking programming on NUMA architectures



Memory is directly attached to multi-core chips

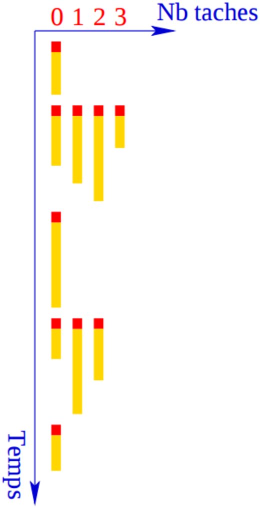
- Non-uniform memory access (NUMA) architectures

Multitasking programming on NUMA architectures



General Concepts

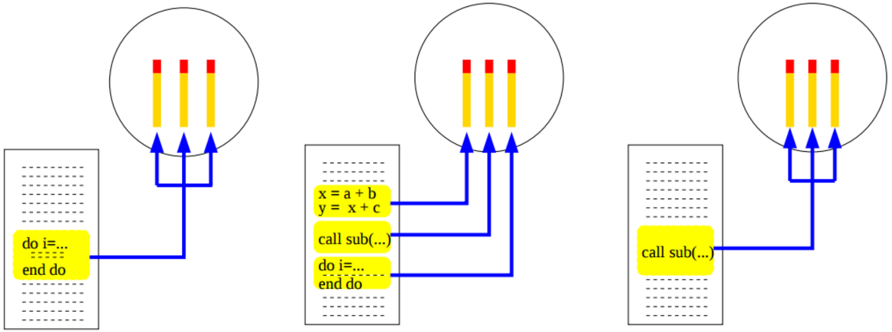
- A OpenMP program = A process with threads
- Consequence: Variable shared or not
- Sequential (always on thread 0) and parallel (multiple threads) composition



Type of Parallelization

Work sharing essentially consists of:

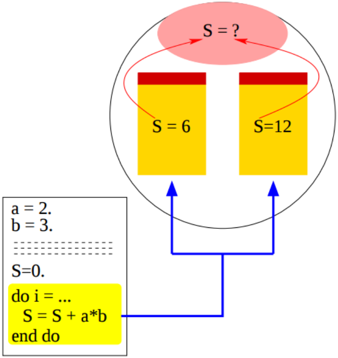
- execute a loop by distributing the iterations between the tasks
- execute multiple sections of code but only one per task
- execute multiple occurrences of the same procedure by different tasks (orphaning)



Boucle parallèle Sections parallèles Procédure parallèle (orphaning)

Synchronization

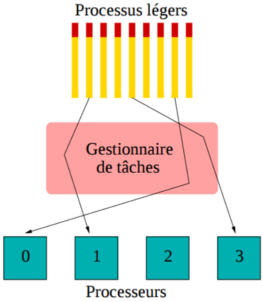
- It is sometimes necessary to introduce a synchronization between the concurrent tasks to avoid, for example, that these modify in any order the value of the same shared variable (case of reduction operations).



Distribution of Tasks

Tasks are assigned to processors by the operating system. Different cases can occur:

- at best, at each moment, there is one task per processor with as many tasks as dedicated processors during the whole execution time
- at worst, all tasks are processed sequentially by one and only one processor
- in reality, for essentially reasons of operation on a machine whose processors are not dedicated, the situation is generally intermediate.



Features of the OpenMP model

- **Benefits**
  - Transparent and portable “ thread ” management
  - Easy programming
- **Drawbacks**
  - Data locality problem
  - Shared but not hierarchical memory
  - Efficiency not guaranteed (impact of the material organization of the machine)
  - Limited scaling, moderate parallelism

OpenMP VS Threads

Two methods of reflecting the distribution

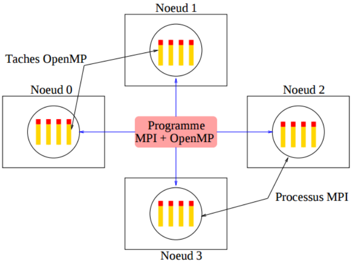
- A task: a set of calculations (common concept)
- **Thread**: We explicitly define the distribution of tasks between threads
- **Thread**: More complex code (we must write thread management)
- **OpenMP**: We don't define the distribution
- **Thread**: We can choose the order of execution
- **OpenMP**: Non-deterministic execution order

- **Thread**: Static scheduling / at the expense of the developer
- **OpenMP**: Dynamic scheduling / at the expense of the system

OpenMP VS MPI

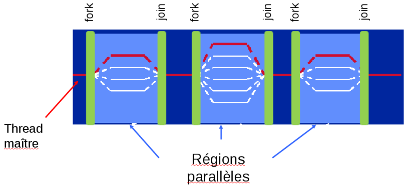
These are two complementary models of parallelization.

- OpenMP and MPI has a C and C ++ interface
- MPI is a multiprocess model whose communication mode between the processes is explicit (the management of communications is the responsibility of the user)
- OpenMP is a multitasking model whose communication mode between tasks is implicit (communication management is the responsibility of the compiler).



How it works ?

- It is the developer's responsibility to introduce OpenMP directives into his code (at least in the absence of automatic parallelization tools)
- When the program is executed, the operating system builds a parallel region on the "fork and join" model
- When entering a parallel region, the master task creates / activates (fork) "child" processes (light processes) which disappear / doze off at the end of the parallel region (join) while the task master continues the execution of the program alone until the entry of the next parallel region



Directives/pragmas format

- Sentinelle directive [clause[clause]..]

```
#pragma omp parallel private(a,b) firstprivate
{
   ...
}
```

- The line is interpreted if the OpenMP flag is activated when calling the compiler otherwise it is a comment → portability

Construction of a parallel region

```
#include <omp.h>

int main(int argc, char** argv) {
  int a,b,c;

  /*  Sequential code executed by the master */

  #pragma omp parallel private(a,b)  \
    shared(c)
  {
        /* Parallel area executed by all
           threads */
  }

  /*  Sequential code */
}
```

IF clause of the PARALLEL directive

- Conditional creation of a parallel clause region **IF(logic_expression)**

```
#include <omp.h>

int main(int argc, char** argv) {
  int a,b,c;

 /*  Sequential code executed by the master */

  #pragma omp parallel for if(para_low)
  {
        /* Parallel area executed by all
           threads */
  }

  /*  Sequential code */
}
```

- L’expression logique sera évaluée avant le début de la région parallèle.

OpenMP Threads

- **Defining the number of threads**
  - Through an environment variable `OMP_NUM_THREADS`
  - Through a function : `OMP_SET_NUM_THREADS()`
  - Through a clause `NUM_THREADS()` of the PARALLEL directive
- **Threads are numbered**
  - The number of threads does not necessarily equal the number of physical cores
  - The thread of index 0 is the master task
  - `OMP_GET_NUM_THREADS()` : number of threads
  - `OMP_GET_THREAD_NUM()` : index of a thread
  - `OMP_GET_MAX_THREADS()` : maximum number of threads

Include, Compilation and Execution

- #include <omp.h>
- gcc/g++ -fopenmp -o prog prog.c[pp]
- export OMP_NUM_THREADS=2 ; ./prog

Status of a variable

- The status of a variable in a parallel area is either
  - be SHARED, it is in the global memory
  - is PRIVATE, it is in the stack of each thread, its value is undefined at the entry of the zone
  - or FIRSTPRIVATE, it is in the stack of each thread, its value is defined at the entry of the zone (initial value of the variable)
  - or LASTPRIVATE, it is in the stack of each thread, its value is kept at the output (the value of the last task)
- Declare the status of a variable
  - `#pragma omp parallel PRIVATE(list)`
  - `#pragma omp parallel FIRSTPRIVATE(list)`
  - `#pragma omp parallel SHARED(list)`
- Declare default status `#pragma omp default(SHARED|NONE)`

PARALLEL directive clauses

- `NONE` : Any variable must have an explicitly defined status
- `SHARED (variables_list)` : Variables shared between threads
- `PRIVATE (variables_list)` : Private variables for each of the threads, undefined outside the PARALLEL block
- `FIRSTPRIVATE (variables_list)` : Variable initialized with the value that the original variable had just before the parallel section

- `THREADPRIVATE` :
  - A global variable, a file descriptor or static variables (in C)
  - The variable instance persists from one region parallel to the other (unless dynamic mode is active)
  - The instance of the variable in the sequential area is also that of thread 0

- `COPYIN` : allows to transmit the value of the shared variable to all tasks

Memory allocation

- The default compiler option is generally `PRIVATE`: local variables allocated in the private $\Rightarrow$ stack, but some options allow you to change this default and it is recommended not to use these options for OpenMP
- A memory allocation or deallocation operation on a private variable will be local to each task
- If a memory allocation / deallocation operation relates to a shared variable, the operation must be performed by a single task.

Memory allocation

- The size of the stack is limited, different environment variables or functions can act on this size
- The stack has a size limit for the shell (variable depending on the machine). (`ulimit --s`) (`ulimit --s unlimited`), values expressed in KB.
- OpenMP: OMP environment variable_STACKSIZE: defines the number of bytes that each OpenMP thread can use for its private stack

Work sharing

- Distribution of a loop between threads (loop //)
- Distribution of several sections of code between threads, one section of code per thread (sections //)
- Execution of a portion of code by a single thread
- Execution of several occurrences of the same procedure by different threads

Scope of a parallel region

- The reach of a parallel region extends
  - to the code contained lexically in this region (static scope)
  - to the code of subroutines called
- The union of the two represents the dynamic range

Work Sharing

Guidelines for controlling the distribution of work, data and the synchronization of tasks within a parallel region :

- `FOR`
- `SECTIONS`
- `SINGLE`
- `MASTER`
- `WORKSHARE` (only in Fortran)

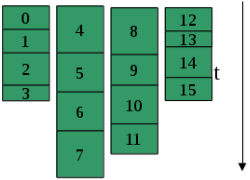
Work sharing : Parallel loop

FOR directive: parallelism by distribution of iterations of a loop.

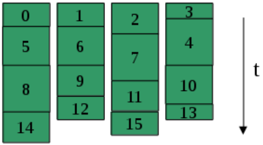
- The mode of distribution of the iterations can be specified in the SCHEDULE clause (coded in the program or thanks to an environment variable)
- A global synchronization is performed at the end of construction END FOR (except if NOWAIT)
- Possibility of having several FOR constructions in a parallel region.
- Loop indices are whole and private
- Infinite and do while loops cannot be parallelized

Work sharing : SCHEDULE clause

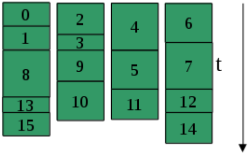
- `#pragma OMP FOR SCHEDULE(STATIC, packet_size)`
  - By default *packet_size* $= \frac{nb\_iterations}{nb\_threads}$
  - Example : 16 iterations (0 à 15), 4 threads : default packet size is 4



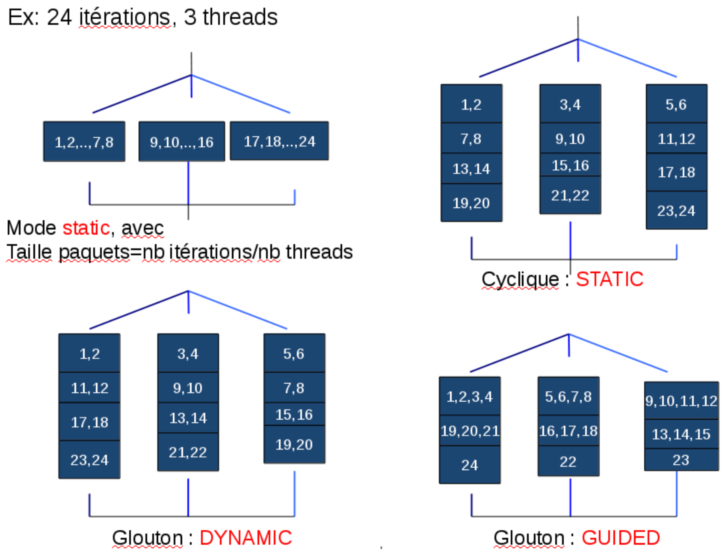
- `#pragma OMP FOR SCHEDULE(DYNAMIC, packet_size)`
  - Packets are dynamically distributed to free threads
  - All packages have the same size except possibly the last one, by default the size of the packages is 1.



- `#pragma OMP FOR SCHEDULE(GUIDED, packet_size)`
  - *packet_size* : minimum packet size (1 by default) except the last.
  - Maximum packet size at the start of the loop (here 2) then decreases to balance the load.

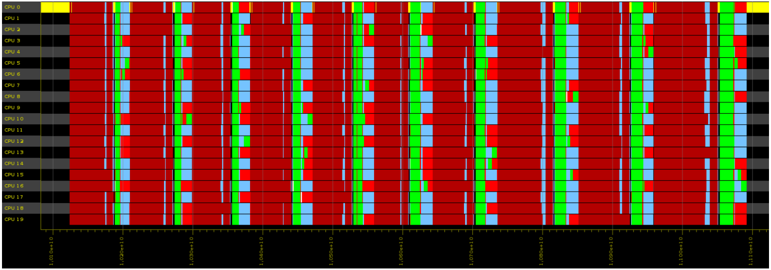


Worksharing : SCHEDULE clause



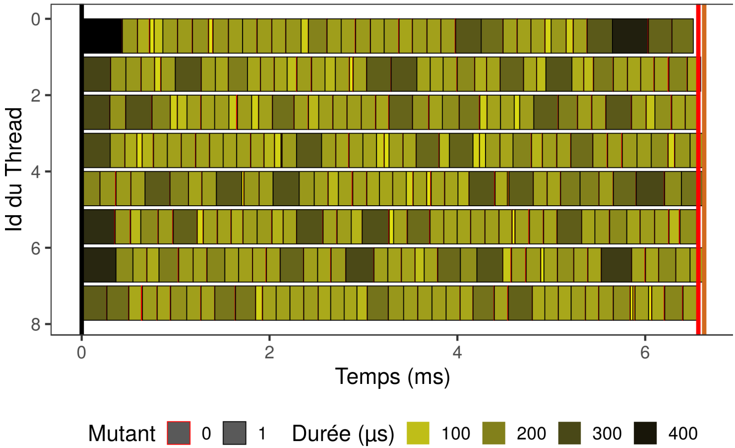
Aevol and OpenMP

- 3 parallel loop pragma, 2 atomics (counter)

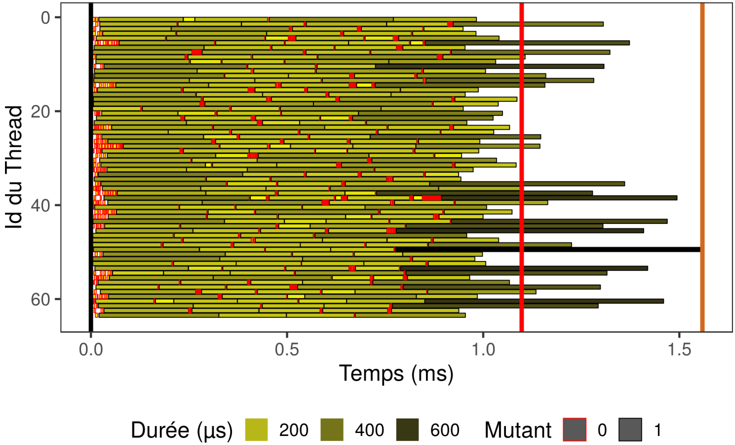


- We also parallelize statistics and checkpointing using task with dependencies
- We try to use task with dependencies everywhere, performance were very bad

Worksharing : SCHEDULE

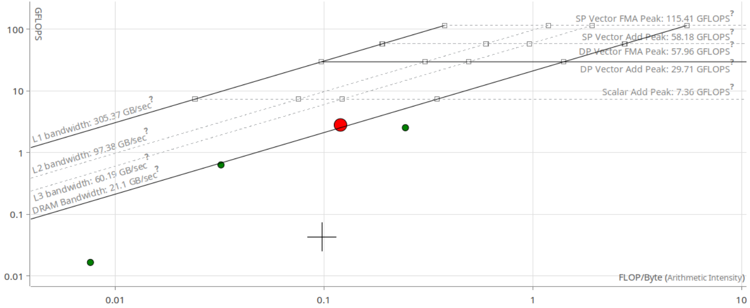


Worksharing : SCHEDULE



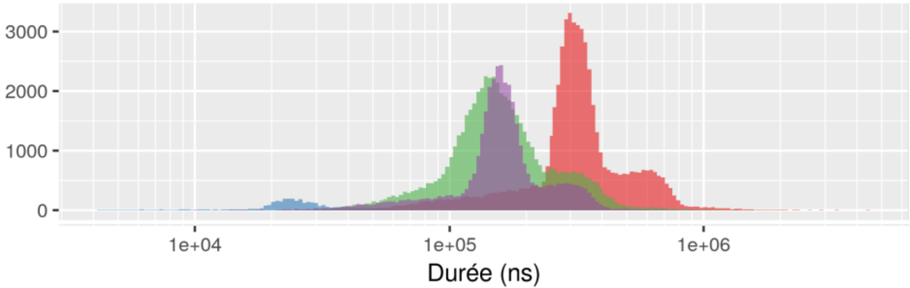
Limitations of the Aevol OpenMP implementation

- The speedup is pretty bad
- Aevol is memory bound (looking for motifs)

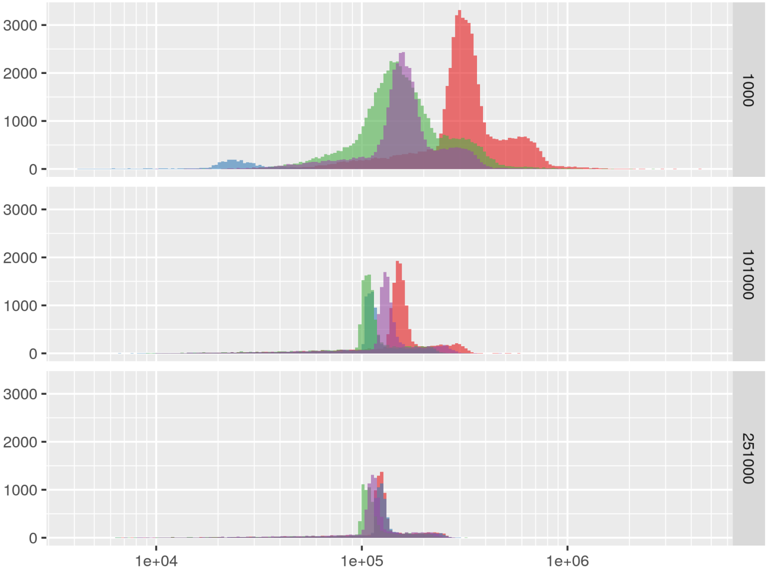


- But there is another issue

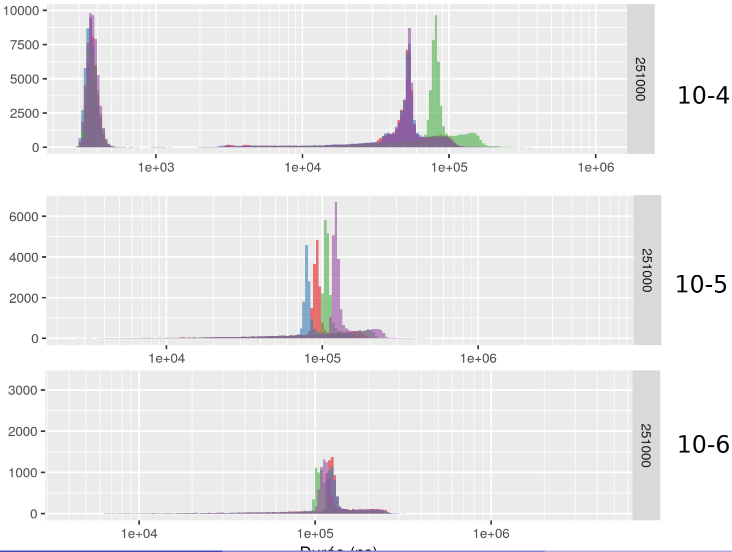
Task runtime distribution at a given timestep



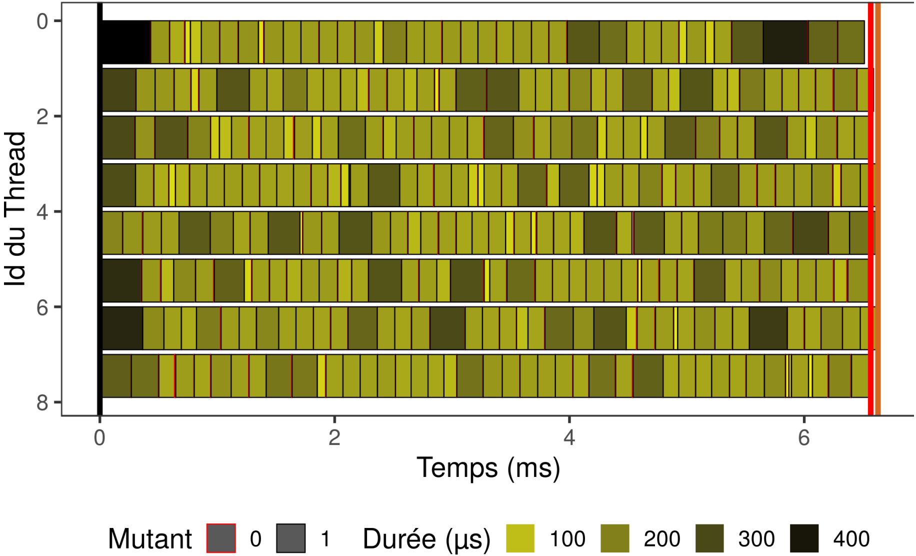
Task runtime during the evolution



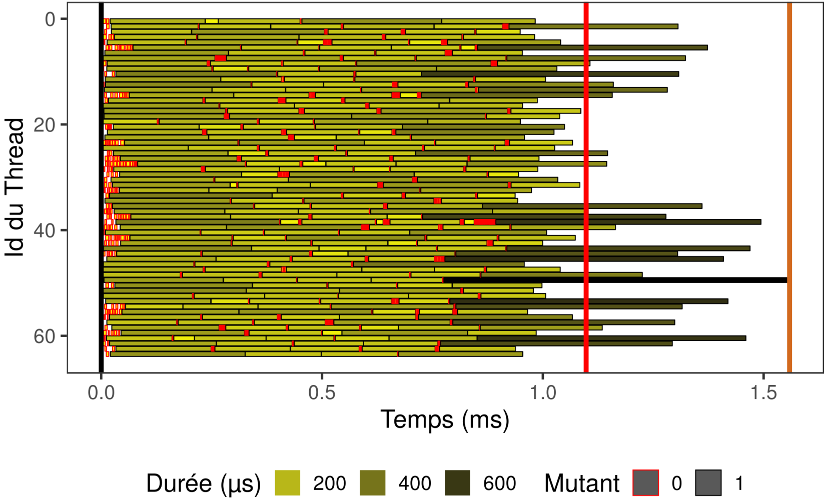
Task runtime with different simulation parameters



Irregularity and OpenMP

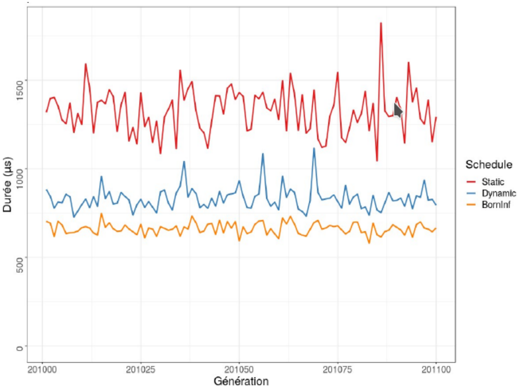


Irregularity and OpenMP



8x more cores, 4.4x speedup

OpenMP scheduling methods



Ordonnancement	Speed Up	Inactivité
Static	33.2	48.1 %
Dynamic	51.2	20 %
Borne Inf	64	0 %

A “simple” list-scheduling issue

- List scheduling (OpenMP dynamic) : (2-1/m)-approximation, O(n log m)
- LPT (Longest Processing Time first) : (4/3 -1/(3m))-approximation, O(n log n + n log m)

A "simple" list-scheduling issue

- List scheduling (OpenMP dynamic) : (2-1/m)-approximation, O(n log m)
- LPT (Longest Processing Time first) : (4/3 -1/(3m))-approximation, O(n log n + n log m)
- But can we model task runtime ?

A “simple” list-scheduling issue

- List scheduling (OpenMP dynamic) : (2-1/m)-approximation, O(n log m)
- LPT (Longest Processing Time first) : (4/3 -1/(3m))-approximation, O(n log n + n log m)
- But can we model task runtime ?
- We try but too difficult and some hardware artifacts

A "simple" list-scheduling issue

- List scheduling (OpenMP dynamic) : (2-1/m)-approximation, O(n log m)
- LPT (Longest Processing Time first) : (4/3 -1/(3m))-approximation, O(n log n + n log m)
- But can we model task runtime ?
- We try but too difficult and some hardware artifacts

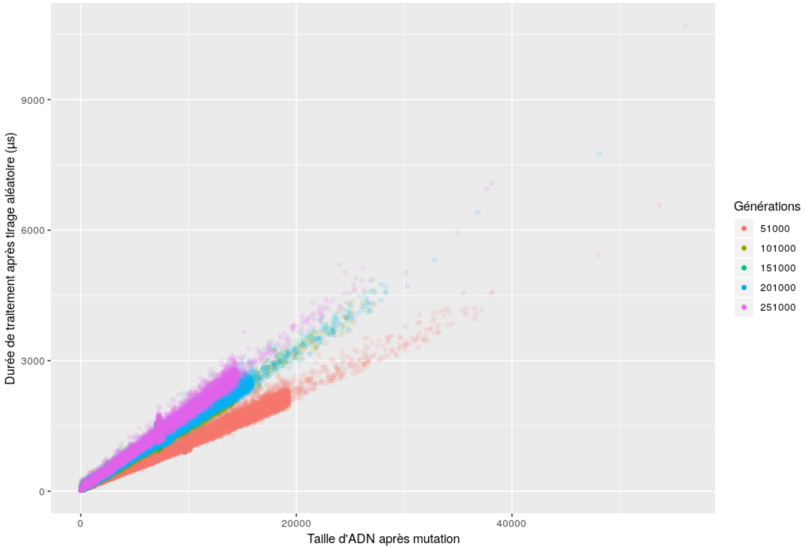
A “simple” list-scheduling issue

- List scheduling (OpenMP dynamic) : (2-1/m)-approximation, O(n log m)

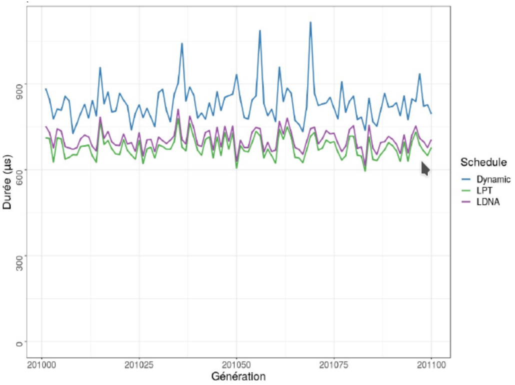
- LPT (Longest Processing Time first) : (4/3 -1/(3m))-approximation, O(n log n + n log m)

- But can we model task runtime ?
- We try but too difficult and some hardware artifacts
- We do not need to predict runtime only to predict tasks order

How to predict task order with Aevol



Preliminary results



Ordonnancement	Speed Up	Inactivité
Dynamic	51.2	20 %
LDNA	59.8	3.4 %*
LPT	62.8	1.9 %

Worksharing : SCHEDULE clause

- The choice of the distribution method can be postponed during the execution of the code with `SCHEDULE(RUNTIME)`
- Taking into account the environment variable `OMP_SCHEDULE`
- Example : `export OMP_SCHEDULE=‘‘DYNAMIC,400‘‘`

Reduction : why ?

In sequential

```
for (int i = 0;
      i < N; i++) {
   X=X+a(i)
}
```

In parallel

```
#pragma omp for shared(X)
 for (int i = 0;
       i < N; i++) {
    X=X+a(i)
 }
```

Reduction : associative operation applied to shared scalar variables

- Each task calculates a partial result independently of the others. Intermediate reductions on each thread are visible locally.
- Then the tasks are synchronized to update the final result in a global variable, by applying the same operator to the partial results.
- Attention, no guarantee of identical results from one execution to another, the intermediate values can be combined in a random order

Reduction in pratice

- Example: `#pragma omp for reduction (op:  list)` (op is an operator or an intrinsic function)
- The variables in the list must be shared in the area surrounding the directive!
- A local copy of each variable in the list is assigned to each thread and initialized according to the operation (e.g. 0 for $+$, 1 for *)
- The clause will apply to the variables in the list if the instructions are one of the following types:
  - $x = x$ operator expr
  - $x =$ expr operator x
  - $x =$ intrinsic (x, expr)
  - $x =$ intrinsic (expr,x)
- x is a scalar variable
- expr is a scalar expression not referencing x
- intrinsic $=$ MAX, MIN, IAND, IOR, IEOR
- operator $= +$,*, .AND., .OR., .EQV., .NEQV.

Reduction in pratice

```
int factorial(int number)
{
  int fac = 1;
  #pragma omp parallel for reduction(*:fac)
  for(int n=2; n<=number; ++n)
    fac *= n;
  return fac;
}
```

Ordered execution : ORDERED

- Run a zone sequentially
  - For debugging purpose
  - For having ordered I/O
- Clause and Directive : `ORDERED`
- The order of execution of instructions in the area framed by the directive will be identical to that of sequential execution, i.e. in the order of iterations

```
#pragma omp for ordered \\
            schedule(dynamic)
for(int n=0; n<100; ++n)
{
  files[n].compress();

  #pragma omp ordered
  send(files[n]);
}
```

Unfolding nested loops : COLLAPSE directive

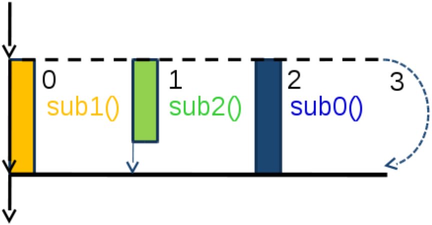
- The `COLLAPSE(N)` clause allows you to specify a number of loops to unfold to create a large space of iterations
- Loops must be perfectly nested
- Example: If the loops in i and j can be parallelized, and if N and M are small, we can thus parallelize on the whole work corresponding to the 2 loops

```
#pragma omp parallel for\\
           collapse(2)
for(int i=0; i<N; ++i)
  for(int j=0; j<M; ++j)
  {
    tick(i,j);
  }
```

Work sharing : parallel SECTIONS parallèles

- **Goal** : Distribute the execution of several independent pieces of code on different tasks
- **A section**: a portion of code executed by one and only one task
- Directive `SECTION` within a construction `SECTIONS`

```
#pragma omp sections
{
  #pragma omp section
  { sub0() }
  #pragma omp section
  { sub1() }
  #pragma omp section
  { sub2() }
}
```



Work sharing : exclusive execution

Construction SINGLE

- Execution of a portion of code by one and only one task (in general, the first which arrives on construction)
- clause NOWAIT : allows not to block the other tasks which by default await its completion
- accepted clauses : PRIVATE, FIRSTPRIVATE
- COPYPRIVATE(var): update private copies of the variable (*var*) on all tasks

```
#pragma omp parallel
{
  Work1();
  #pragma omp single
  {
    Work2();
  }
  Work3();
}
```

Work sharing : exclusive execution

Structure SINGLE

- Execution of a portion of code by the master task alone
- No synchronization, neither at the beginning nor at the end (unlike SINGLE)
- Beware of variable updates that would be used by other threads
- No clause

```
#pragma omp parallel
{
  Work1();

  #pragma omp master
  {
    Work2();
  }

  Work3();
}
```

OpenMP and SIMD

Since OpenMP 4.x, it is possible to give hint to compiler through OpenMP pragma to improve automatic vectorization

```
#pragma omp simd reduction(+:sum) aligned(a:64)
for (i = 0; i < num; i++) {
    a[i] = b[i] * c[i];
    sum = sum + a[i];
}
```

OpenMP and SIMD: WARNING !!!

- Using OpenMP SIMD pragma will bypass the compiler analysis !!! (On Intel Compiler, it can refuse/desactivate OpenMP SIMD if it predicts too much performance prediction)
- Use with caution !
- Incorrect results possible !
- Poor performance possible !
- Memory errors possible !

OpenMP and SIMD: Aligned Memory

- C

```
_mm_malloc(8*sizeof(float),64);
```

```
posix_memalign(pointer,size,align);
```

- C++
  - Write a custom allocator
  - You need to overload new and delete method within your class

OpenMP and SIMD

You can mix parallel threads (classic OpenMP) with OpenMP SIMD to use both (you should whenever it is possible)

```
#pragma omp parallel for simd
for (i = 0; i < num; i++) {
    sum = sum + a[i];
}
```

OpenMP, SIMD and functions

You can declare that a function can be used within a vectorized loop

```
#pragma omp declare simd
float myfunction(float a, float b, float c) {
 return a*b+c;
}

int main(int argc, char** argv) {
...
#pragma omp simd
for(i=0; i < num; i++) {
  OUT[i] = myfunction(array_a[i], array_b[i], array_c[i]
}
...
}
```

Work sharing : the TASK construct

- A `TASK` in the OpenMP sense is a unit of work whose execution can be deferred (or start immediately)
  - Allow dynamic generation of tasks
- Allows you to parallelize irregular problems
  - unbounded loops
  - recursive algos
  - producer / consumer diagrams
- A `TASK` is composed
  - of a code to execute
  - of an associated data environment

Work sharing : the TASK construct

- The TASK construct explicitly defines an OpenMP TASK
- If a thread encounters a TASK construction, a new instance of the TASK is created (code package + associated data)
- The thread can either execute the task or delay its execution. The task can be assigned to any thread on the team.

```
void increment_list_items
                (node* head)
{
  #pragma omp parallel
  {
    #pragma omp single
    {
      for(node*
        p = head; p;
        p = p->next)
        {
          #pragma omp task
          process(p);
// p: firstprivate by default
        }
    }
  }
}
```

Work sharing : the TASK construct

- **Note**: the concept existed before construction

- A thread that encounters a construction texttt PARALLEL
  - creates a set of implicit `TASKs` (code + data package)
  - creates a team of threads
  - implicit `TASKs` are tied to threads, one for each team thread

- When is the `TASK` executed?
  - The execution of a generated `TASK` can be assigned, by the scheduler, to a team thread that has finished its work.
  - Standard 3.0 imposes rules on the execution of `TASK`
  - Example: The pending `TASK` in the region //, must all be executed by the team threads that meet
    - A `BARRIER` (implicit or explicit)
    - A directive `TASKWAIT`

Synchronization

- Synchronization of all tasks on the same level of education (global barrier)
- Scheduling of concurrent tasks for the consistency of shared variables (mutual exclusion)
- Synchronization of several tasks among a set (lock mechanism)

Synchronization: global barrier

- By default at the end of parallel constructions, in the absence of NOWAIT
- Directive BARRIER: Explicitly imposes a synchronization barrier: each task waits for the end of all the others

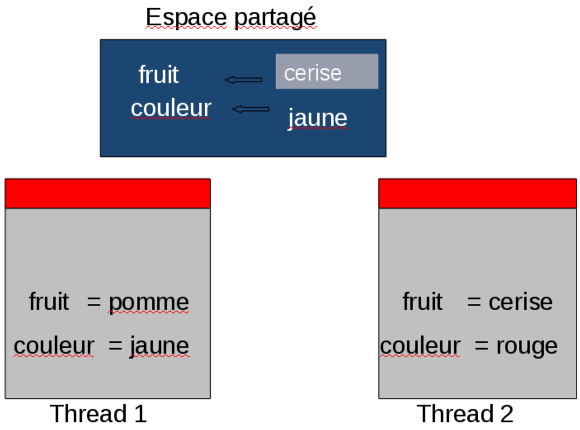
```
#pragma omp parallel
{
  /* All threads are running. */
  SomeCode();
  #pragma omp barrier
  // All threads are running but
  // they wait for everyone else before
  SomeMoreCode();
}

#pragma omp parallel
{
  #pragma omp for
  for(int n=0; n<10; ++n)
   Work();
  // Implicit barrier, all threads
  // are waiting for the end of FOR
  SomeMoreCode();
}
```

Synchronization

It may be necessary to introduce a synchronization between concurrent tasks to prevent them from modifying the value of a variable in any order

Ex :

2 threads ont un espace de mémoire partagé



Synchronization: critical regions

Directive CRITICAL

- It applies to a portion of code
- Tasks execute the critical region in a non-deterministic order, one at a time
- Guarantee threads mutual exclusion access
- Its scope is dynamic

```
int main(int argc, char** argv) {
int x;
x = 0;
#pragma omp parallel shared(x)
{
  #pragma omp critical
  x = x + 1;
}  /* end of parallel section */
}
```

Synchronization : atomic update

The ATOMIC directive only applies when updating a memory location

- A shared variable is read or modified in memory by only one thread at a time

- Acts on the instruction immediately following if it is in the form :
  - x = x (op) exp
  - or x = exp (op) x
  - or x = f(x,exp)
  - or x = f(exp,x)
  - op : +, -, *, /, .AND., .OR., .EQV., .NEQV.
  - f : MAX, MIN, IAND, IOR, IEOR
  - x is a scalar variable

```
#pragma omp atomic
        count = count+1;
```

FLUSH Directive

- Shared variable values may temporarily stay in registers for performance reasons
- The FLUSH directive guarantees that each thread has access to the values of the shared variables modified by the other threads.

```
    /* presumption: int a = 0

    /* First thread */
/* Second thread */
    b = 1;
a = 1;
    #pragma omp flush(a,b)
#pragma omp flush(a,b)
    if(a == 0)
if(b == 0)
    {
{
      /* Critical section */
/* Critical section */
    }
}
```

FLUSH Directive

- FLUSH implicit in certain regions
  - At the level of a BARRIER
  - At the entry and exit of a region PARALLEL, CRITICAL, or ORDERED, and of a region PARALLEL of work sharing
  - When calling the " lock " functions
  - Immediately before or after each TASK scheduling point
  - When entering and leaving regions ATOMIC (applies to variables updated by ATOMIC)
- No implicit FLUSH
  - At the entrance to a work sharing region
  - When entering or leaving a region MASTER

Performance and work sharing

- Minimize the number of parallel regions
- Avoid leaving a parallel region to immediately recreate it

```
#pragma omp parallel
{
 #pragma omp for
 for(int n=0; n<10; ++n)
   printf("␣%d", n);
}
printf(".\n");
```

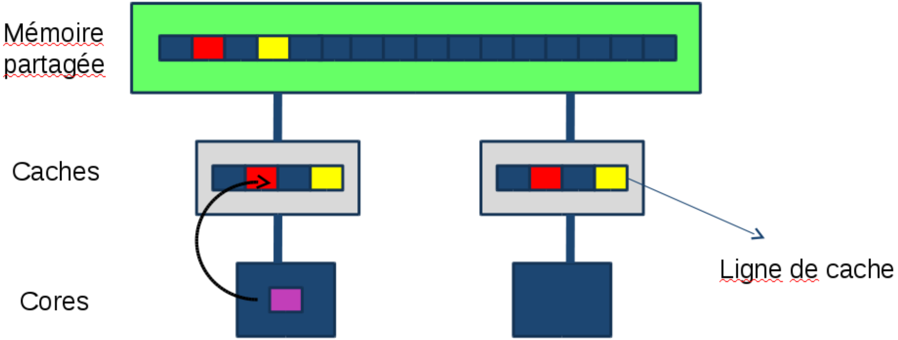
```
#pragma omp parallel for
for(int n=0; n<10; ++n)
 printf("␣%d", n);
printf(".\n");
```

Performance and work sharing

- Introduce a `PARALLEL FOR` in loops capable of executing iterations in //
- If there are dependencies between iterations, try to remove them by modifying the algorithm
- If there are dependent iterations, introduce constructions `CRITICAL` around the variables concerned by the dependencies
- If possible, group several structures in a single parallel region `FOR`
- If possible, parallel the outermost loop
- Adapt the number of tasks to the size of the problem to be treated in order to minimize the additional costs of task management by the system
- Use `SCHEDULE (RUNTIME)` if necessary
- `ATOMIC REDUCTION` is more efficient than `CRITICAL`

Performance : False Sharing effects

The consistency of the caches and the negative effects of “false sharing” can have a strong impact on performance



An operation to load a shared cache line invalidates the other copies of this line.

Performance : False Sharing effects

- The use of structures in shared memory can induce a decrease in performance and a strong limitation of extensibility
    - For performance reasons, use of the cache
    - If several processors handle different but adjacent data in memory, the update of individual elements can cause a complete loading of a cache line, so that the caches are consistent with the memory
- False sharing degrades performance when all of the following conditions are met
    - Shared data is changed on # cores
    - Multiple threads on # cores update data in the same cache line
    - These updates occur very frequently and simultaneously

Performance : False Sharing effects

- When shared data is only read, it does not generate false sharing
- In general, the phenomenon of false sharing can be reduced by
  - By possibly privatizing variables
  - Sometimes by increasing the size of the tables (problem size or artificial increase) or by doing padding
  - Sometimes by modifying the way the iterations of a loop are shared between threads (increase the size of the packets)

Performance : False Sharing effects

```
int a[nthreads];

#pragma parallel for shared(nthreads,a) schedule(static,1)
for (int i=0; i < nthreads; i++) a[i] = i
```

- **Nthreads** : number of threads executing the loop
- Suppose each thread has a copy of textbf a in its local cache. The packet size of 1 causes a phenomenon of false sharing with each update
- If a cache line can contain C elements of the vector textbf a, we can solve the problem by artificially extending the dimensions of the array ("array padding"): we declare an array **a (C , n)** and we replace **a (i)** with **a (1, i)**

Performance on multi-core architectures: efficiently schedule threads

- Maximizing the performance of each CPU is highly dependent on the location of memory accesses
- Variables are stored in a memory (and cache) area when they are initialized
  - parallelize the initialization of elements (*e.g.*, array) in the same way (same mode and same packet size) as the job
- The ideal would be to be able to specify thread placement constraints according to thread / memory affinities

Performance on multi-core architectures: problematic of the diversity of architectures

- types of processors
- numbers of cores
- cache levels
- memory architecture
- types of interconnection of the different components
- Optimization strategies which may differ from one configuration to another

Performance: conditional parallelization

- Use the `IF` clause to set up conditional parallelization
- Example: only parallelize a loop if its size is large enough

OpenMP

- Requires a multi-processor shared memory machine
- Relatively easy to use, even in a sequential program
- Allows progressive parallelization of a sequential program
- The full potential of parallel performance lies in parallel regions
- Within these parallel regions, work can be shared using loops and parallel sections. But we can also differentiate a task for a particular job
- Explicit global or point-to-point synchronizations are sometimes necessary in parallel regions
- Special care must be taken when defining the status of the variables used in a construction

OpenMP versus MPI

- OpenMP uses memory common to all processes. All communication is done by reading and writing in this memory (and using synchronization mechanisms)
- MPI is a library of routines allowing communication between different processes (located on different machines or not), communications are made by sending or receiving explicit messages
- For the programmer: rewrite the code with MPI, simple addition of directives in the sequential code with OpenMP
- Possibility of mixing the two approaches in the case of a machine cluster with multi-core nodes
- Choice strongly dependent: on the machine, the code, the time that the developer wants to devote to it and the gain sought
- Superior extensibility with MPI