OpenMP

Multithreaded Parallelization for Shared-Memory Machines

Speakers:
Rémi Lacroix
Thibaut Véry
<first-name.surname@idris.fr>

Authors:
Jalel Chergui
Pierre-François Lavallée

Reproduction Rights

Copyright © 2001-2020 CNRS/IDRIS

OpenMP: Contents

1 – Introduction 8
1.1 – History 9
1.2 – `OpenMP` Specifications 10
1.3 – Terminology and definitions 11
1.4 – General concepts 12
1.4.1 – Execution model 12
1.4.2 – Threads (*light-weight processes*) 13
1.5 – Functionalities 16
1.6 – OpenMP versus MPI 17
1.7 – Bibliography 19
2 – Principles 20
2.1 – Programming interface 20
2.1.1 – Format of a directive 21
2.1.2 – Compilation 23
2.2 – Parallel construct 24
2.3 – Data-sharing attribute of variables 26
2.3.1 – Private variables 26
2.3.2 – The DEFAULT clause 28

OpenMP: Contents

2.3.3 – Dynamic allocation 29
2.3.4 – Equivalence between Fortran variables 31
2.4 – Extent of a parallel region 32
2.5 – Transmission by arguments 34
2.6 – Static variables 35
2.7 – Complementary information 37
3 – Worksharing 39
3.1 – Introduction 39
3.2 – Parallel loop 40
3.2.1 – The SCHEDULE clause 41
3.2.2 – An ordered execution 45
3.2.3 – A reduction operation 46
3.2.4 – Fusion of loop nests 47
3.2.5 – Additional clauses 49
3.3 – The WORKSHARE construct 51
3.4 – Parallel sections 54
3.4.1 – Construction SECTIONS 55
3.4.2 – Complementary information 56

OpenMP: Contents

3.5 – Exclusive execution 57
3.5.1 – The SINGLE construct 58
3.5.2 – The MASTER construct 60
3.6 – Orphaned routines 61
3.7 – Summary 63
4 – Synchronisation 64
4.1 – Barrier 66
4.2 – Atomic update 67
4.3 – Critical regions 69
4.4 – The FLUSH directive 71
4.4.1 – Example: An easy trap 72
4.4.2 – Example: A difficult trap 73
4.4.3 – Commentaries on the previous codes 74
4.4.4 – The correct code 75
4.4.5 – Nested loops with double dependencies 76
4.5 – Summary 82
5 – SIMD Vectorisation 83
5.1 – Introduction 83

5.2 – SIMD loop vectorisation 84
5.3 – Parallelisation and SIMD vectorisation of a loop 85
5.4 – SIMD vectorisation of scalar functions 86
6 – OpenMP tasks 87
6.1 – Introduction 87
6.2 – The concept bases 88
6.3 – The task execution model 89
6.4 – Some examples 91
6.5 – Dependency between tasks 95
6.6 – Data-sharing attributes of variables in the tasks 97
6.7 – Example of updating elements of a linked list 98
6.8 – Example of a recursive algorithm 99
6.9 – The `FINAL` and `MERGEABLE` clauses 100
6.10 – The `TASKGROUP` synchronisation 101
7 – Affinities 103
7.1 – Thread affinity 103
7.1.1 – The *cpuinfo* command 104
7.1.2 – Use of the *KMP_AFFINITY* environment variable 106

OpenMP: Contents

7.1.3 – Thread affinity with `OpenMP` 4.0 108
7.2 – Memory affinity 110
7.3 – A « *First Touch* » strategy 113
7.4 – Examples of impact on performance 114
8 – Performance 118
8.1 – Good performance rules 119
8.2 – Time measurements 122
8.3 – Speedup 123
9 – Conclusion 124
10 – Annexes 125
10.1 – Subjects not addressed here 125
10.2 – Some traps 126

1 – Introduction

- OpenMP is a parallel programming model which initially only targeted shared memory architectures. Today, it also targets accelerators, integrated systems and real-time systems.
- The calculation tasks can access a common memory space. This limits data redundancy and simplifies information exchanges between tasks.
- In practice, parallelization is based on the use of light-weight processes (*threads*). We are speaking, therefore, of a *multithreaded* program.

1.1 – History

- Multithreaded parallelization has existed for a long time at certain manufacturers (e.g. CRAY, NEC, IBM, ...) but each one had its own set of directives.
- The resurgence of shared memory multiprocessors made it compelling to define a standard.
- The standardisation attempt of the PCF (*Parallel Computing Forum*) was never adopted by the official standardisation authorities.
- On the 28th October 1997, a large majority of industry researchers and manufacturers adopted `OpenMP` (*Open Multi-Processing*) as an « industrial standard ».
- Today, the `OpenMP` specifications belong to the ARB (*Architecture Review Board*), the only organisation responsible for its development.

1.2 – OpenMP Specifications

- The OpenMP 2 version was finalised in November 2000. Most importantly, it provided parallelization extensions to certain Fortran 95 constructions.
- The OpenMP 3 version of May 2008 primarily introduced the concept of tasks.
- The version OpenMP 4 of July 2013 followed by version 4.5 of November 2015 brought numerous innovations, notably accelerator support, dependencies between tasks, SIMD (vectorisation) programming and management of thread placement.
- The version OpenMP 5 of November 2018 followed by version 5.1 of November 2020 focused mainly on improving accelerator support. It also brought improvements for task programing, handling of non-uniform memory and support for the latest versions of C (11), C++ (17) and Fortran (2008) languages.

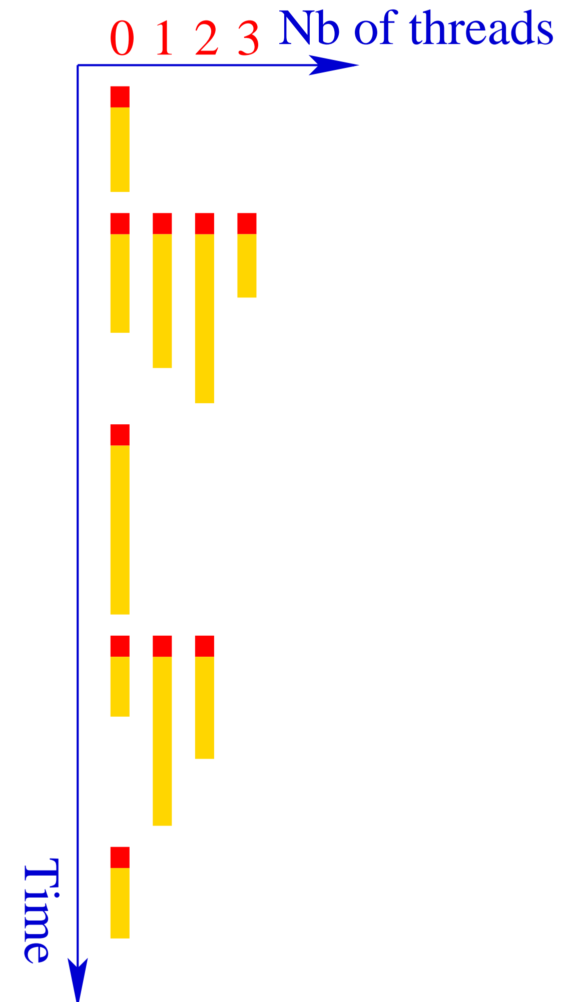
1.3 – Terminology and definitions

- *Thread*: An execution entity with a local memory (*stack*).
- *Team* : A set of one or several *threads* which participate in the execution of a parallel region.
- *Task* : An instance of executable code and its associated data. These are generated by the `PARALLEL` or `TASK` constructs.
- *Shared variable*: A variable for which the name provides access to the same block of storage shared by the tasks inside a parallel region.
- *Private variable*: A variable for which the name provides access to a different block of storage for each task inside a parallel region.
- *Host device* : Hardware (usually an SMP node) on which `OpenMP` begins its execution.
- *Target device* : Hardware (accelerator card such as GPU or Xeon Phi) on which a portion of code and the associated data can be transferred and then executed.

1.4 – General concepts

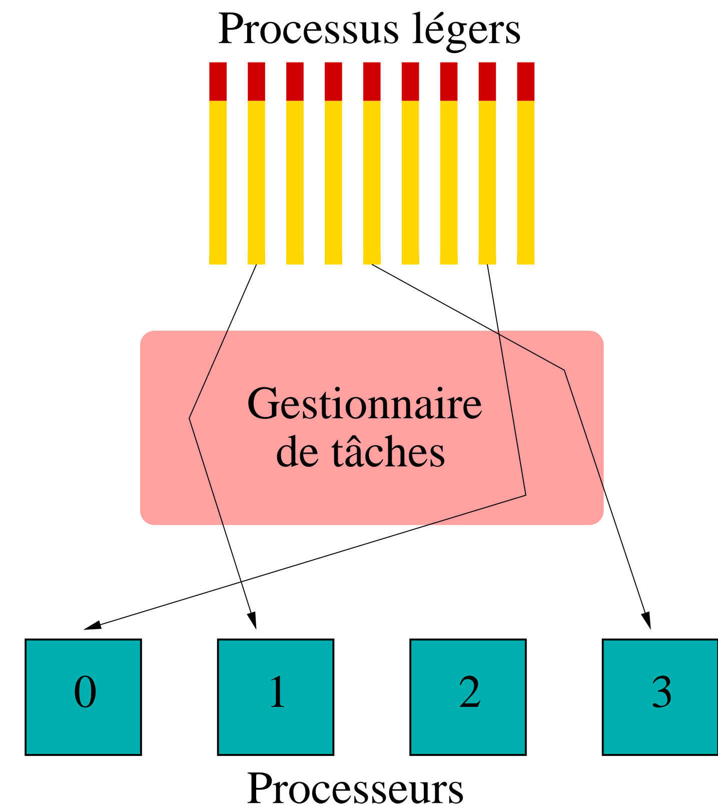
1.4.1 – Execution model

- When it begins, an OpenMP program is sequential. It has only one process, the master thread with rank 0, which executes the initial implicit task.
- OpenMP allows defining **parallel regions** which are code portions destined to be executed in parallel.
- At the entry of a parallel region, new threads and new implicit tasks are created. Each thread executes its implicit task concurrently with the others in order to share the work.
- An OpenMP program consists of an alternation between sequential regions and parallel regions.



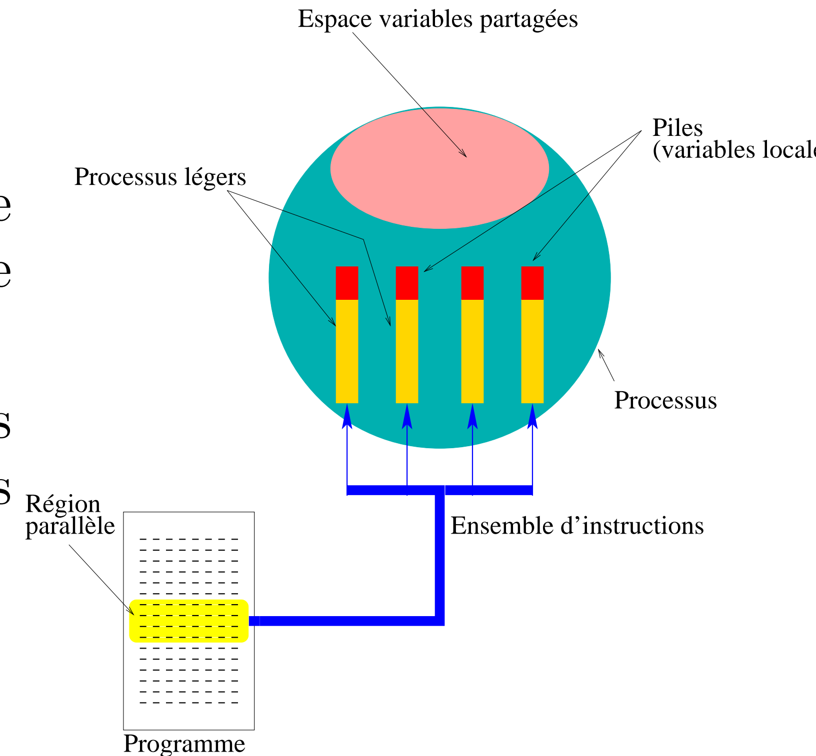
1.4.2 – Threads (*light-weight processes*)

- ☞ Each thread executes its own sequence of instructions corresponding to its task.
- ☞ The operating system chooses the execution order of the processes (light-weight or not). It assigns them to the available computing units (processor cores).
- ☞ There is no guarantee of the overall order in which the parallel program instructions will be executed.

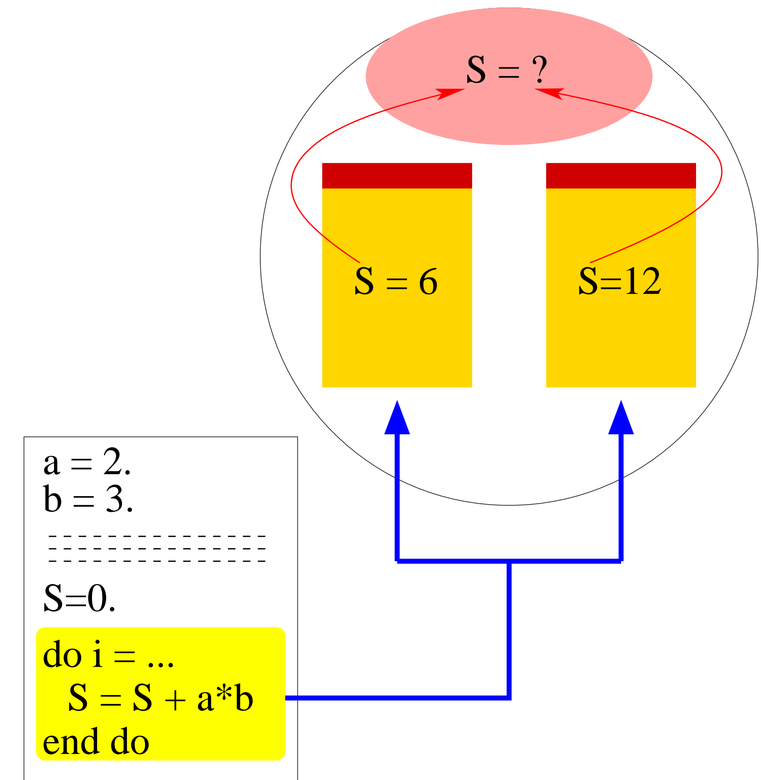


☞ Tasks of the same program share the memory space of the initial task (shared memory) but also dispose of a local memory space: the stack.

☞ Therefore, it is possible to define the shared variables (stored in the shared memory) or the private variables (stored in the stack of each one of the tasks).



- ☞ In shared memory, it is sometimes necessary to introduce synchronisation between concurrent tasks.
- ☞ Synchronisation ensures that 2 threads do not modify the value of the same shared variable in a random order (reduction operations).



1.5 – Functionalities

OpenMP facilitates the writing of parallel algorithms in shared memory by proposing mechanisms to:

- ☞ Share the work between tasks. For example, it is possible to distribute the iterations of a loop between the tasks. Then, when the loop acts on arrays, it can easily distribute the data processing between the threads.
- ☞ Share or privatise the variables.
- ☞ Synchronise the threads.

Starting with the 3.0 version, OpenMP has also allowed expressing parallelism in the form of a group of explicit tasks to be performed. OpenMP 4.0 allows offloading a part of the work to an accelerator.

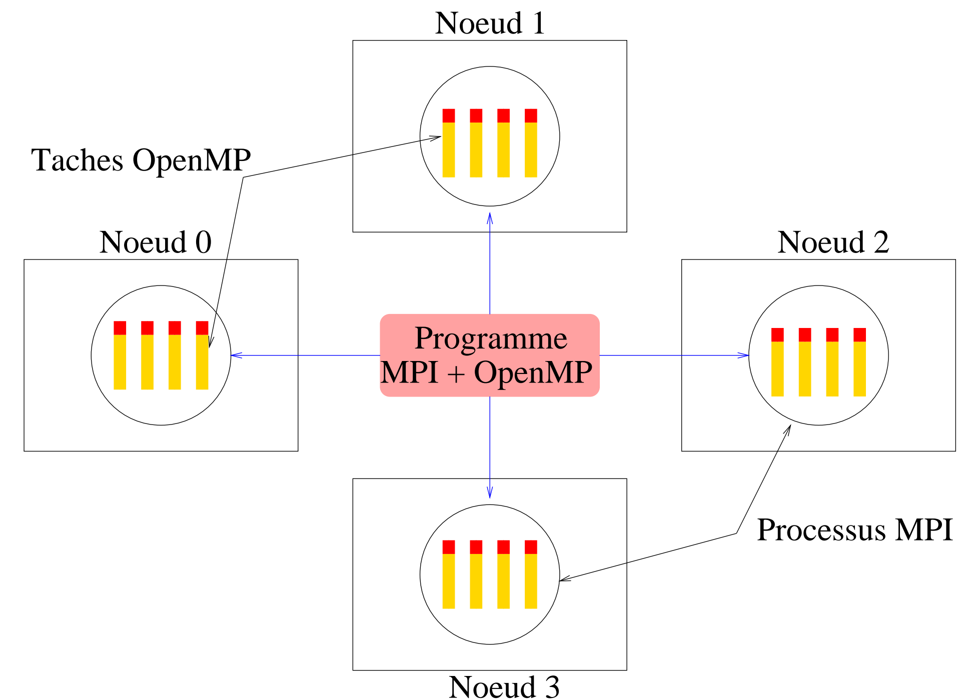
1.6 – OpenMP versus MPI

These two programming models are adapted to two different parallel architectures:

☞ `MPI` is a distributed memory programming model: Communication between the processes is explicit and the user is responsible for its management.

☞ `OpenMP` is a shared memory programming model: Each thread has a global scope of the memory.

☞ On a cluster of independant shared memory multiprocessor machines (compute nodes), the implementation of parallelization at two levels (`MPI` and `OpenMP`) in the same program can be a major advantage for the parallel performance or the memory footprint of the code.



1.7 – Bibliography

- The first book about OpenMP: R. Chandra & *al.*, *Parallel Programming in OpenMP*, ed. Morgan Kaufmann Publishers, Oct. 2000.
- Another book about OpenMP: B. Chapman & *al.*, *Using OpenMP*, MIT Press, 2008.
- A more recent one : R. van der Pas, E. Stotzer & Ch. Terboven, *USING OPENMP - THE NEXT STEP Affinity, Accelerators, Tasking, and SIMD*, MIT Press, 2017.
- Specifications of the OpenMP standard: https://www.openmp.org/

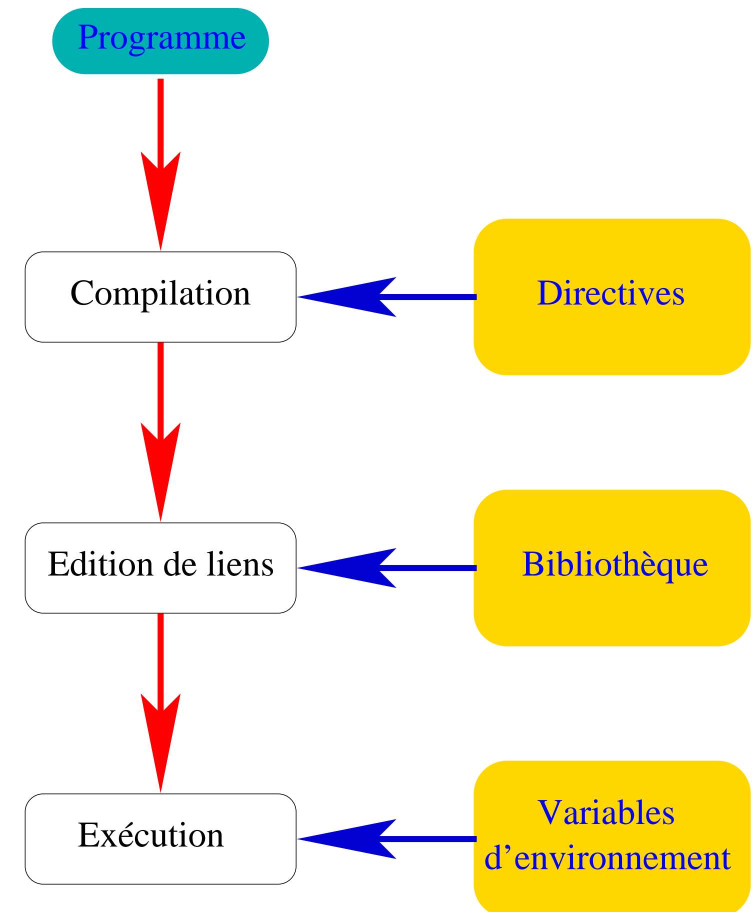
2 – Principles

2.1 – Programming interface

❶ Compilation directives and clauses: For defining work-sharing, synchronisation or data-sharing attributes (private or shared).

❷ Functions and subprograms: These are part of a library loaded during the link of the program.

❸ Environment variables: Once they are set, these values are taken into account at the execution.



2.1.1 – Format of a directive

☞ An OpenMP directive has the following general form :

```
sentinelle directive [clause[ clause]...]
```

☞ It is a comment line which is ignored by the compiler if the option that allows the interpretation of OpenMP directives is not specified.

☞ The sentinel is a character string whose value depends on the language used.

☞ There is an OMP_LIB Fortran 95 module and an omp.h C/C++ include file which define the prototype of all the OpenMP functions. It is mandatory to include them in any OpenMP program unit which uses these functions.

☞ For Fortran, in free format:

```
  !$ use OMP_LIB
  ...
  !$OMP PARALLEL PRIVATE(a,b) &
        !$OMP FIRSTPRIVATE(c,d,e)
       ...
  !$OMP END PARALLEL ! This is a comment
```

☞ For Fortran, in fixed format:

```
      !$ use OMP_LIB
      ...
C$OMP PARALLEL PRIVATE(a,b)
C$OMP1 FIRSTPRIVATE(c,d,e)
       ...
C$OMP END PARALLEL
```

☞ For C and C++:

```
#ifdef _OPENMP
#include <omp.h>
#endif
...
#pragma omp parallel private(a,b) firstprivate(c,d,e)
{ ... }
```

2.1.2 – Compilation

Compilation options for activating the interpretation of OpenMP directives by some compilers are as follows:

- The GNU compiler: -fopenmp

```
gfortran -fopenmp prog.f90                               # Fortran compiler
```

- The Intel compiler: -fopenmp or -qopenmp

```
ifort -fopenmp prog.f90                                  # Fortran compiler
```

- The PGI/NVIDIA compiler : -mp

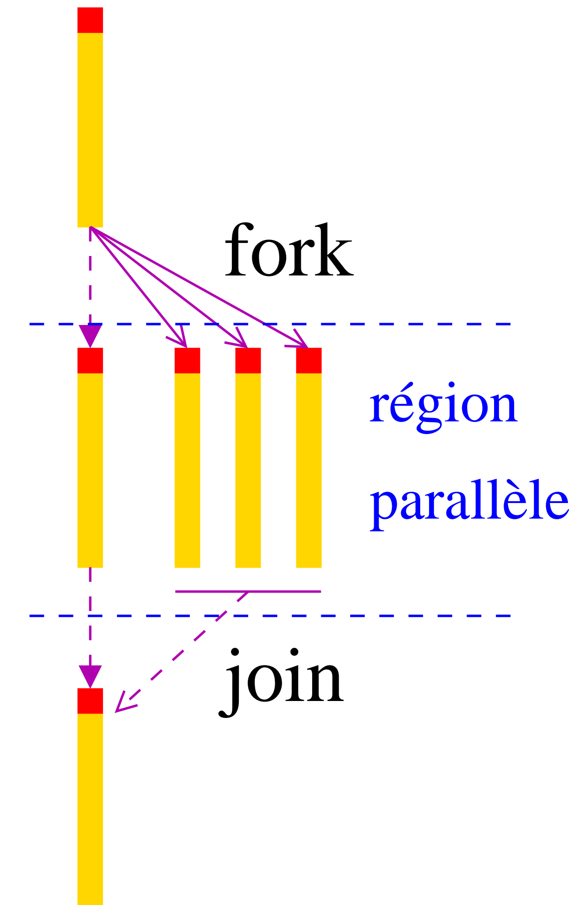
```
pgfortran/nvfortran -mp prog.f90                         # Fortran compiler
```

Execution example:

```
export OMP_NUM_THREADS=4                                 # Number of desired threads
./a.out                                                  # Execution
```

2.2 – Parallel construct

- An OpenMP program is an alternation of sequential and parallel regions (« *fork and join* » model)
- At the entry of a parallel region, the master thread (rank 0) creates/activates (*forks*) the «child» processes (light-weight processes or threads) and an equal number of implicit tasks. Each child thread executes its implicit task, then disappears or hibernates at the end of the parallel region (*joins*).
- At the end of the parallel region, the execution becomes sequential again with only the master thread executing.



- Within the same parallel region, each thread executes a separate implicit task but the tasks are composed of the same duplicated code.
- The data-sharing attribute (DSA)[a] of the variables are shared, by default.
- There is an implicit synchronisation barrier at the end of the parallel region.

[a] The initials DSA will be used henceforth to represent "data-sharing attribute".

```
program parallel
 !$ use OMP_LIB
 implicit none
 real    :: a
 logical :: p

 a = 92290; p=.false.
 !$OMP PARALLEL
  !$ p = OMP_IN_PARALLEL()
  print *,"A vaut : ",a
 !$OMP END PARALLEL
 print*,"Parallel ?:", p

end program parallel
```

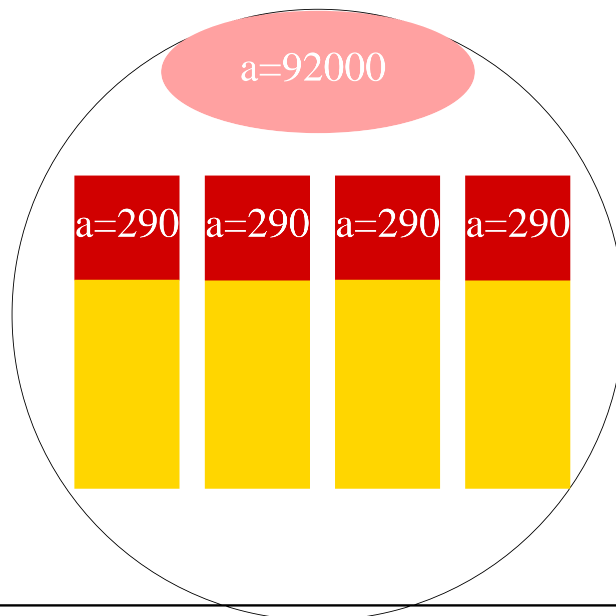
```
> ifort -fopenmp prog.f90
> export OMP_NUM_THREADS=4
> a.out
```

```
 A vaut : 92290
 A vaut : 92290
 A vaut : 92290
 A vaut : 92290
 Parallele ? : T
```

2.3 – Data-sharing attribute of variables

2.3.1 – Private variables

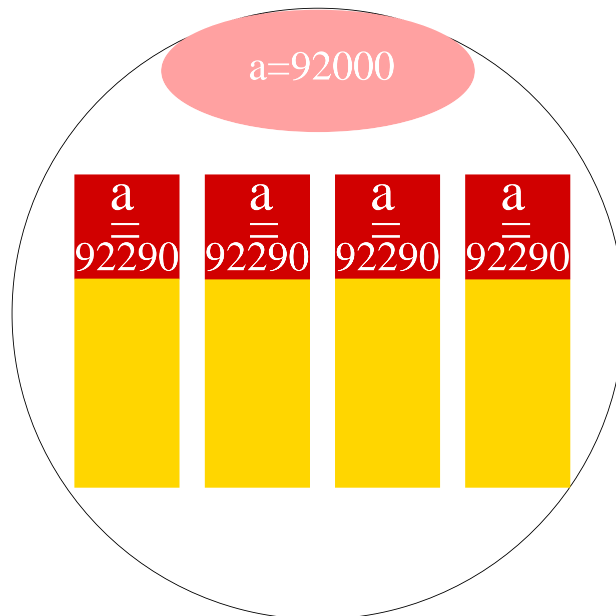
- The PRIVATE clause allows changing the DSA of a variable to private.
- If a variable has a private DSA, it is allocated in the stack of each task.
- The private variables are not initialised on entry to the parallel region.




```
program parallel
 !$ use OMP_LIB
 implicit none
 real     :: a
 integer :: rang

 a = 92000
 !$OMP PARALLEL PRIVATE(rang,a)
   !$ rang = OMP_GET_THREAD_NUM()
   a = a + 290
   print *,"Rang : ",rang, &
           "; A vaut : ",a
 !$OMP END PARALLEL
 print*,"Hors region, A vaut :",a

end program parallel
```

```
 Rang : 1 ; A vaut : 290
 Rang : 2 ; A vaut : 290
 Rang : 0 ; A vaut : 290
 Rang : 3 ; A vaut : 290
 Hors region, A vaut : 92000
```

☞ With the FIRSTPRIVATE clause, however, it is possible to force the initialisation of a private variable to the last value it had before entry to the parallel region.




```
program parallel
 implicit none
 real :: a

 a = 92000.
!$OMP PARALLEL FIRSTPRIVATE(a)
   a = a + 290
   print *,"A vaut : ",a
!$OMP END PARALLEL
print*,"Hors region, A vaut :",a

end program parallel
```

```
> ifort -fopenmp prog.f90
> export OMP_NUM_THREADS=4
> a.out
```

```
 A vaut :   92290
 A vaut :   92290
 A vaut :   92290
 A vaut :   92290
 Hors region, A vaut : 92000
```

☞ After exiting the parallel region, the private variables are lost.

2.3.2 – The DEFAULT clause

- The variables are shared by default but to avoid errors, it is recommended to define the DSA of each variable explicitly.
- Using the DEFAULT(NONE) clause requires the programmer to specify the status of each variable.
- In Fortran, it is also possible to change the implicit DSA of variables by using the DEFAULT(PRIVATE) clause.

```
program parallel
 !$ use OMP_LIB
 implicit none
 logical :: p

 p=.false.
 !$OMP PARALLEL DEFAULT(NONE) &
       !$OMP SHARED(p)
  !$ p = OMP_IN_PARALLEL()
 !$OMP END PARALLEL
 print*,"Parallel ?:", p
end program parallel
```

```
Parallele ? : T
```

2.3.3 – Dynamic allocation

The dynamic memory allocation/deallocation operation can be done inside the parallel region.

- If the operation concerns a private variable, this local variable will be created/destroyed on each task.
- If the operation concerns a shared variable, it would be more prudent if only one thread (for example, the master thread) does this operation. Because of the data locality, it is recommended to initialise the variables inside the parallel region (« first touch »).

```
program parallel
  !$ use OMP_LIB
  implicit none
  integer                              :: n,debut,fin,rang,nb_taches,i
  real, allocatable, dimension(:) :: a

  n=1024
  allocate(a(n))
  !$OMP PARALLEL DEFAULT(NONE) PRIVATE(debut,fin,nb_taches,rang,i) &
        !$OMP SHARED(a,n) IF(n .gt. 512)
     nb_taches=OMP_GET_NUM_THREADS() ; rang=OMP_GET_THREAD_NUM()
     debut=1+(rang*n)/nb_taches
     fin=((rang+1)*n)/nb_taches
     do i = debut, fin
        a(i) = 92290. + real(i)
     end do
     print *,"Rang : ",rang,"; A(",debut,"),...,A(",fin,") : ",a(debut),",...,",a(fin)
  !$OMP END PARALLEL
  deallocate(a)
end program parallel
```

```
> export OMP_NUM_THREADS=4;a.out
```

```
Rang : 3 ; A( 769 ), ... , A( 1024 ) : 93059., ... , 93314.
Rang : 2 ; A( 513 ), ... , A(  768 ) : 92803., ... , 93058.
Rang : 1 ; A( 257 ), ... , A(  512 ) : 92547., ... , 92802.
Rang : 0 ; A(   1 ), ... , A(  256 ) : 92291., ... , 92546.
```

2.3.4 – Equivalence between Fortran variables

- Only variables with the same DSA should be put in equivalence.
- If this is not the case, the result will be undefined.
- These remarks are also true for a `POINTER` association.

```
program parallel
  implicit none
  real :: a, b
  equivalence(a,b)

  a = 92290.
  !$OMP PARALLEL PRIVATE(b) &
        !$OMP SHARED(a)
     print *,"B vaut : ",b
  !$OMP END PARALLEL
end program parallel
```

```
> ifort -fopenmp prog.f90
> export OMP_NUM_THREADS=4;a.out
```

```
B vaut : -0.3811332074E+30
B vaut : 0.0000000000E+00
B vaut : -0.3811332074E+30
B vaut : 0.0000000000E+00
```

2.4 – Extent of a parallel region

- The extent of an `OpenMP` construct represents its scope in the program.
- The influence (or scope) of a parallel region includes the code lexically contained in this region (the static extent) as well as the code of the called routines. The union of these two represents the «dynamic extent».

```
program parallel
 implicit none
 !$OMP PARALLEL
    call sub()
 !$OMP END PARALLEL
end program parallel
subroutine sub()
 !$ use OMP_LIB
 implicit none
 logical :: p
 !$ p = OMP_IN_PARALLEL()
 !$ print *,"Parallele ?:", p
end subroutine sub
```

```
> ifort -fopenmp prog.f90
> export OMP_NUM_THREADS=4;a.out
```

```
 Parallele ? : T
 Parallele ? : T
 Parallele ? : T
 Parallele ? : T
```

- In a routine called in a parallel region, the local variables and automatic arrays are implicitly private for each task. (They are defined in the stack.)
- In C/C++, the variables declared inside a parallel region are private.

```
program parallel
 implicit none
 !$OMP PARALLEL DEFAULT(SHARED)
    call sub()
 !$OMP END PARALLEL
end program parallel
subroutine sub()
 !$ use OMP_LIB
 implicit none
 integer :: a
 a = 92290
 a = a + OMP_GET_THREAD_NUM()
 print *,"A vaut : ",a
end subroutine sub
```

```
> ifort -fopenmp prog.f90
> export OMP_NUM_THREADS=4;a.out
```

```
 A vaut :   92290
 A vaut :   92291
 A vaut :   92292
 A vaut :   92293
```

2.5 – Transmission by arguments

- In a subroutine or function, all the variables transmitted by argument (*dummy parameters*) inherit the DSA defined in the lexical (static) extent of the region.

```
program parallel
 implicit none
 integer :: a, b

 a = 92000
 !$OMP PARALLEL SHARED(a) PRIVATE(b)
    call sub(a, b)
    print *,"B vaut : ",b
 !$OMP END PARALLEL
end program parallel

subroutine sub(x, y)
 !$ use OMP_LIB
 implicit none
 integer :: x, y

 y = x + OMP_GET_THREAD_NUM()
end subroutine sub
```

```
> ifort -fopenmp prog.f90
> export OMP_NUM_THREADS=4
> a.out
```

```
 B vaut : 92002
 B vaut : 92003
 B vaut : 92001
 B vaut : 92000
```

2.6 – Static variables

- A static variable is accessible during the entire lifespan of a program.
  - In Fortran, this is the case with variables appearing in `COMMON`, in a `MODULE`, declared `SAVE`, or initialised in the declaration (instruction `DATA` or symbol =).
  - In C/C++, these are variables declared with the keyword `static`.
- In an `OpenMP` parallel region, a static variable is **shared** by default.

```
module var_stat
  real :: c
end module var_stat
```

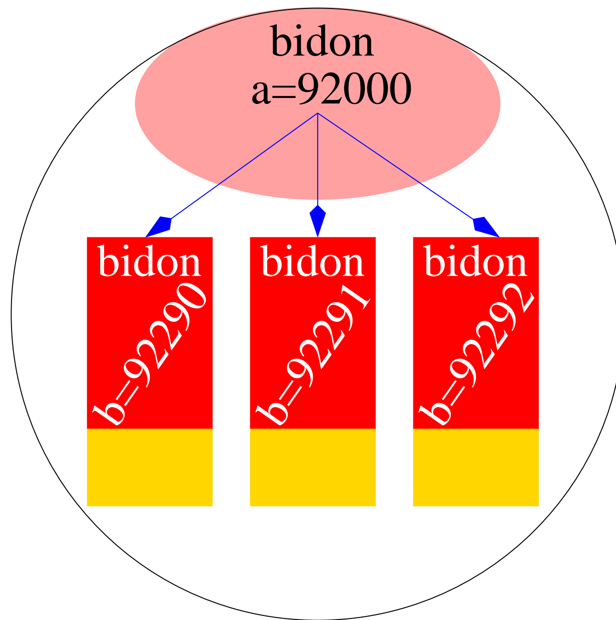
```
program parallel
  use var_stat
  implicit none
  real :: a
  common /bidon/a
  !$OMP PARALLEL
    call sub()
  !$OMP END PARALLEL
end program parallel
subroutine sub()
  use var_stat
  use OMP_LIB
  implicit none
  real    :: a, b=10.
  integer :: rang
  common /bidon/a
  rang = OMP_GET_THREAD_NUM()
  a=rang;b=rang;c=rang
  !$OMP BARRIER
  print *,"valeurs de A, B et C : ",a,b,c
end subroutine sub
```

```
> ifort -fopenmp var_stat.f90 prog.f90
> export OMP_NUM_THREADS=2;a.out
```

A possible result is:

```
valeurs de A, B et C : 0.0 1.0 1.0
valeurs de A, B et C : 0.0 1.0 1.0
```

- The THREADPRIVATE directive allows privatising a static instance (for the threads and not the tasks) and makes this persistant from one parallel region to another.
- If the COPYIN clause is specified, the initial value of the static instance is transmitted to all the threads.




```
program parallel
 !$ use OMP_LIB
 implicit none
 integer :: a
 common/bidon/a
 !$OMP THREADPRIVATE(/bidon/)
 a = 92000
 !$OMP PARALLEL COPYIN(/bidon/)
   a = a + OMP_GET_THREAD_NUM()
   call sub()
 !$OMP END PARALLEL
 print*,"Hors region, A vaut:",a
end program parallel
subroutine sub()
 implicit none
 integer :: a, b
 common/bidon/a
 !$OMP THREADPRIVATE(/bidon/)
 b = a + 290
 print *,"B vaut : ",b
end subroutine sub
```

```
B vaut : 92290
B vaut : 92291
B vaut : 92292
B vaut : 92293
Hors region, A vaut : 92000
```

2.7 – Complementary information

- A parallel region construct accepts two other clauses:
  - REDUCTION: For reduction operations with implicit synchronisation between the threads.
  - NUM_THREADS : Allows specifying the number of desired threads at the entry of a parallel region in the same way as the OMP_SET_NUM_THREADS routine would do this.
- The number of concurrent threads can vary, if desired, from one parallel region to another.

```
program parallel
  implicit none

    !$OMP PARALLEL NUM_THREADS(2)
      print *,"Bonjour !"
    !$OMP END PARALLEL

    !$OMP PARALLEL NUM_THREADS(3)
      print *,"Coucou !"
    !$OMP END PARALLEL
end program parallel
```

```
> ifort -fopenmp prog.f90
> export OMP_NUM_THREADS=4
> a.out
```

```
Bonjour !
Bonjour !
Coucou !
Coucou !
Coucou !
```

☞ It is possible to nest parallel regions but this will have no effect if it isn't activated by a call to the OMP_SET_NESTED routine or by setting the OMP_NESTED environment variable at `true`.

```
program parallel
  !$ use OMP_LIB
  implicit none
  integer :: rang

  !$OMP PARALLEL NUM_THREADS(3) &
        !$OMP PRIVATE(rang)
    rang=OMP_GET_THREAD_NUM()
    print *,"Mon rang dans region 1 :",rang
    !$OMP PARALLEL NUM_THREADS(2) &
          !$OMP PRIVATE(rang)
      rang=OMP_GET_THREAD_NUM()
      print *," Mon rang dans region 2 :",rang
    !$OMP END PARALLEL
  !$OMP END PARALLEL
end program parallel
```

```
> ifort ... -fopenmp prog.f90
> export OMP_NESTED=true; a.out
```

```
Mon rang dans region 1 : 0
  Mon rang dans region 2 : 1
  Mon rang dans region 2 : 0
Mon rang dans region 1 : 2
  Mon rang dans region 2 : 1
  Mon rang dans region 2 : 0
Mon rang dans region 1 : 1
  Mon rang dans region 2 : 0
  Mon rang dans region 2 : 1
```

3 – Worksharing

3.1 – Introduction

- The creation of a parallel region and the use of some OpenMP directives/functions should be sufficient to parallelize a part of the code. However, in this case, it is the responsibility of the programmer to distribute the work and to manage the data inside a parallel region.
- Fortunately, there are directives which facilitate this distribution (DO, WORKSHARE, SECTIONS)
- Furthermore, it is possible for some portions of code located in a parallel region to be executed by only one thread (SINGLE, MASTER).
- Synchronisation between threads will be addressed in the following chapter.

3.2 – Parallel loop

- A loop is parallel if all of its iterations are independent of each other.
- This is a parallelism by distribution of loop iterations.
- The parallelized loop is the one which immediately follows the `DO` directive.
- The « infinite » and `do while` loops are not parallelisable with this directive but they can be parallelized via the explicit tasks.
- The distribution mode of the iterations can be specified with the `SCHEDULE` clause.
- Being able to choose the distribution mode allows better control of load-balancing between the threads.
- The loop indices are private integer variables by default, so it is not necessary to specify their DSA.
- A global synchronisation is done, by default, at the end of an `END DO` construct unless the `NOWAIT` clause was specified.
- It is possible to introduce as many `DO` constructs as desired (one after another) in a parallel region.

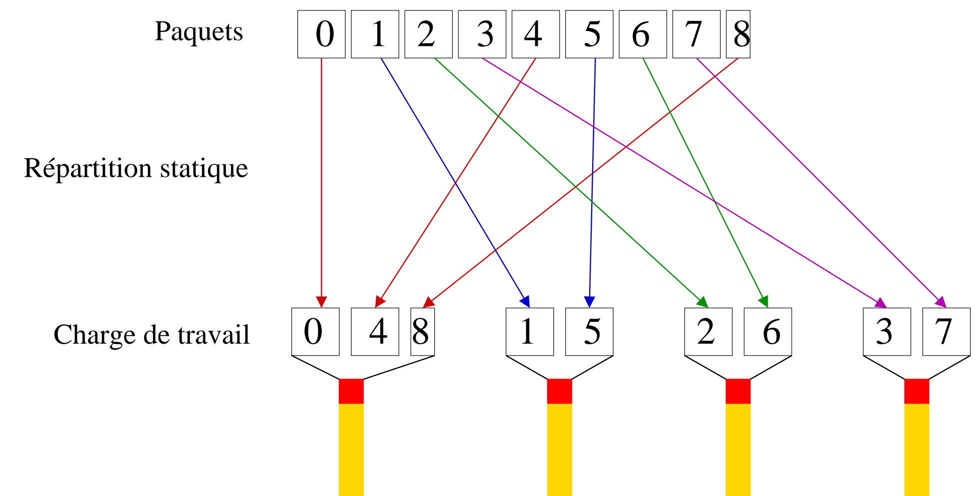
3.2.1 – The SCHEDULE clause

```
program parallel
 !$ use OMP_LIB
 implicit none
 integer, parameter :: n=4096
 real, dimension(n) :: a
 integer            :: i, i_min, i_max, rang, nb_taches
 !$OMP PARALLEL PRIVATE(rang,nb_taches,i_min,i_max)
   rang=OMP_GET_THREAD_NUM() ; nb_taches=OMP_GET_NUM_THREADS() ; i_min=n ; i_max=0
   !$OMP DO SCHEDULE(STATIC,n/nb_taches)
     do i = 1, n
       a(i) = 92290. + real(i) ; i_min=min(i_min,i) ; i_max=max(i_max,i)
     end do
   !$OMP END DO NOWAIT
   print *,"Rang : ",rang,"; i_min :",i_min,"; i_max :",i_max
 !$OMP END PARALLEL
end program parallel
```

```
> ifort ... -fopenmp prog.f90 ; export OMP_NUM_THREADS=4 ; a.out
```

```
Rang : 1 ; i_min : 1025 ; i_max : 2048
Rang : 3 ; i_min : 3073 ; i_max : 4096
Rang : 0 ; i_min :    1 ; i_max : 1024
Rang : 2 ; i_min : 2049 ; i_max : 3072
```

☞ STATIC distribution consists of dividing the iterations into a set of chunks of a given size (except perhaps for the last one). The chunks are then assigned to the threads in a cyclical manner (round-robin), following the order of the threads, until all the chunks have been distributed.



- The choice of iteration distribution mode can be deferred until the execution by using the OMP_SCHEDULE environment variable; however, doing this can sometimes result in performance degradation.
- The chosen distribution mode for loop iterations can be an important factor in load balancing on a machine which has multiple users.
- Be careful: For vector or scalar performance reasons, avoid parallelizing the loops that iterate over the first dimension of a multi-dimensional array (in Fortran).

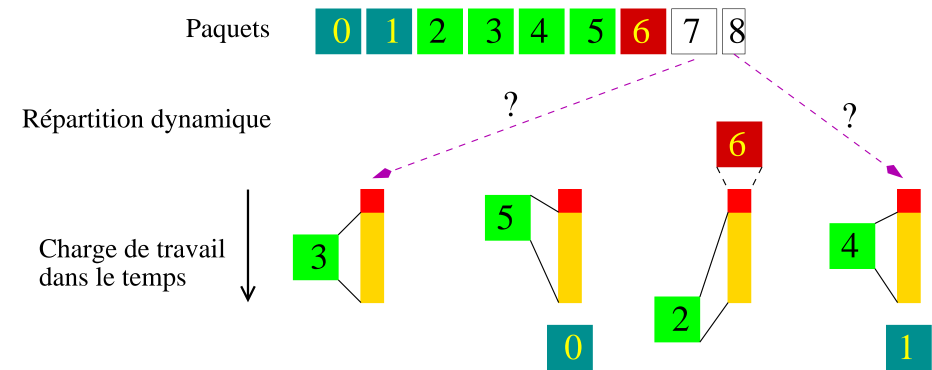
```
program parallel
 !$ use OMP_LIB
 implicit none
 integer, parameter :: n=4096
 real, dimension(n) :: a
 integer :: i, i_min, i_max
 !$OMP PARALLEL DEFAULT(PRIVATE) SHARED(a)
   i_min=n ; i_max=0
   !$OMP DO SCHEDULE(RUNTIME)
      do i = 1, n
        a(i) = 92290. + real(i)
        i_min=min(i_min,i)
        i_max=max(i_max,i)
      end do
   !$OMP END DO
   print*,"Rang:",OMP_GET_THREAD_NUM(), &
          ";i_min:",i_min,";i_max:",i_max
 !$OMP END PARALLEL
end program parallel
```

```
> export OMP_NUM_THREADS=2
> export OMP_SCHEDULE="STATIC,1024"
> a.out
```

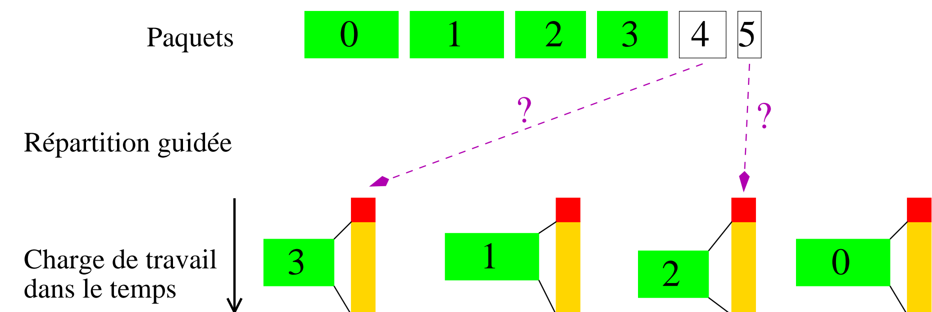
```
Rang: 0 ; i_min:    1 ; i_max: 3072
Rang: 1 ; i_min: 1025 ; i_max: 4096
```

- In addition to the STATIC mode, there are three other ways to distribute the loop iterations:
  - DYNAMIC : The iterations are divided into chunks of a given size; as soon as a thread finishes the iterations of its chunk, another chunk is assigned to it.
  - GUIDED : The iterations are divided into chunks of exponentially decreasing sizes. All the chunks have a size equal or superior to a given value with the exception of the last chunk for which the size can be smaller. As soon as a thread finishes the iterations of its chunk, another chunk is allocated to it.
  - AUTO: The distribution choice for loop iterations is delegated to the compiler or to the execution system (i.e. « *runtime* »).

```
> export OMP_SCHEDULE="DYNAMIC,480"
> export OMP_NUM_THREADS=4 ; a.out
```




```
> export OMP_SCHEDULE="GUIDED,256"
> export OMP_NUM_THREADS=4 ; a.out
```



3.2.2 – An ordered execution

- It is sometimes useful to execute a loop in an ordered way (example: debugging).
- The iteration order will then be identical to that of a sequential execution.

```
program parallel
 !$ use OMP_LIB
 implicit none
 integer, parameter :: n=9
 integer            :: i,rang
 !$OMP PARALLEL DEFAULT(PRIVATE)
   rang = OMP_GET_THREAD_NUM()
   !$OMP DO SCHEDULE(RUNTIME) ORDERED
     do i = 1, n
      !$OMP ORDERED
        print *,"Rang :",rang,"; iteration :",i
      !$OMP END ORDERED
     end do
   !$OMP END DO NOWAIT
 !$OMP END PARALLEL
end program parallel
```

```
> export OMP_SCHEDULE="STATIC,2"
> export OMP_NUM_THREADS=4 ; a.out
```

```
Rang :  0  ; iteration :  1
Rang :  0  ; iteration :  2
Rang :  1  ; iteration :  3
Rang :  1  ; iteration :  4
Rang :  2  ; iteration :  5
Rang :  2  ; iteration :  6
Rang :  3  ; iteration :  7
Rang :  3  ; iteration :  8
Rang :  0  ; iteration :  9
```

3.2.3 – A reduction operation

- A reduction is an associative operation applied to a shared variable.
- The operation can be:
  - Arithmetic: $+$, $-$, $\times$
  - Logical: .AND., .OR., .EQV., .NEQV.
  - An intrinsic function: MAX, MIN, IAND, IOR, IEOR
- Each thread calculates a partial result independently from the others, followed by synchronising with each other to obtain the final result.

```
program parallel
 implicit none
 integer, parameter :: n=5
 integer            :: i, s=0, p=1, r=1
 !$OMP PARALLEL
 !$OMP DO REDUCTION(+:s) REDUCTION(*:p,r)
    do i = 1, n
      s = s + 1
      p = p * 2
      r = r * 3
    end do
 !$OMP END PARALLEL
 print *,"s =",s, "; p =",p, "; r =",r
end program parallel
```

```
> export OMP_NUM_THREADS=4
> a.out
```

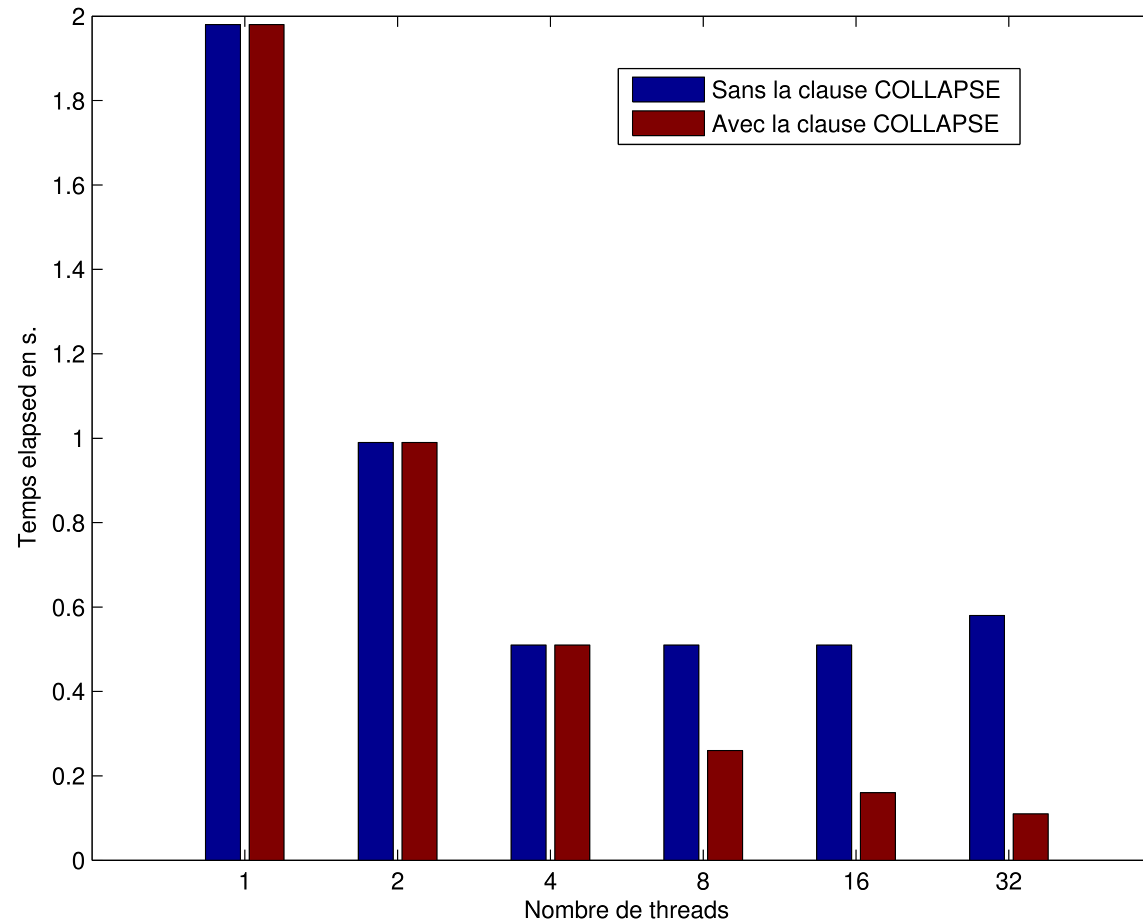
```
s = 5 ; p = 32 ; r = 243
```

3.2.4 – Fusion of loop nests

- When loops are perfectly nested and without dependencies, it can be beneficial to fuse them to obtain a unique loop with a larger iteration space.
- In this way, the granularity of each thread's work is increased and this can sometimes significantly improve the performance.
- The `COLLAPSE(n)` clause allows fusing the `n` nested loops which immediately follow the directive. The new iteration space is then shared by the threads according to the chosen distribution mode.

```
program boucle_collapse
implicit none
integer, parameter :: n1=4, n2=8, &
                      n3=1000000
real, dimension(:,:,:) :: A(n1,n2,n3)
integer :: i, j, k

...
!$OMP PARALLEL
!$OMP DO SCHEDULE(STATIC) COLLAPSE(2)
do i=1,n1
   do j=1,n2
      do k=2,n3
         A(i,j,k)=exp(sin(A(i,j,k-1))+ &
                  cos(A(i,j,k)))/2
      enddo
   enddo
enddo
!$OMP END DO
!$OMP END PARALLEL
end program boucle_collapse
```

- Execution of the preceding program with and without the COLLAPSE clause.
- Evolution of the execution elapsed time (in s.) function of the number of threads, varying from 1 to 32.



3.2.5 – Additional clauses

- The other clauses accepted in the DO directive:
  - PRIVATE: To declare the private DSA of a variable.
  - FIRSTPRIVATE: To privatise a shared variable throughout the DO construct and assign it the last value it had before entering this region.
  - LASTPRIVATE: To privatise a shared variable throughout the DO construct. This allows conserving, at the exit of this construct, the value calculated by the thread executing the last iteration of the loop.

```
program parallel
 !$ use OMP_LIB
 implicit none
 integer, parameter :: n=9
 integer            :: i, rang
 real               :: temp

 !$OMP PARALLEL PRIVATE(rang)
   !$OMP DO LASTPRIVATE(temp)
     do i = 1, n
       temp = real(i)
     end do
   !$OMP END DO
   rang=OMP_GET_THREAD_NUM()
   print *,"Rang:",rang,";temp=",temp
 !$OMP END PARALLEL
end program parallel
```

```
> export OMP_NUM_THREADS=4 ; a.out
```

```
Rang : 2 ; temp= 9.0
Rang : 3 ; temp= 9.0
Rang : 1 ; temp= 9.0
Rang : 0 ; temp= 9.0
```

- The PARALLEL DO directive is a combination of the PARALLEL and DO directives with the union of their respective clauses.
- The END PARALLEL DO termination directive includes a global synchronisation barrier and cannot accept the NOWAIT clause.

```
program parallel
 implicit none
 integer, parameter :: n=9
 integer            :: i
 real               :: temp

 !$OMP PARALLEL DO LASTPRIVATE(temp)
   do i = 1, n
     temp = real(i)
   end do
 !$OMP END PARALLEL DO
end program parallel
```

3.3 – The WORKSHARE construct

- It can only be specified inside a parallel region.
- It is useful for distributing work of certain Fortran 95 constructs, such as:
  - Assignments of Fortran 90 type arrays (i.e. notation A(:,:)).
  - Intrinsic functions applied to array type variables (MATMUL, DOT_PRODUCT, SUM, PRODUCT, MAXVAL, MINVAL, COUNT, ANY, ALL, SPREAD, PACK, UNPACK, RESHAPE, TRANSPOSE, EOSHIFT, CSHIFT, MINLOC and MAXLOC).
  - FORALL and WHERE instructions or blocks.
  - User-defined « ELEMENTAL » functions.
- NOWAIT is the only clause admitted at the end of the construct (END WORKSHARE).

- Only the instructions or `Fortran` 95 blocks specified in the lexical extent will see their operations distributed among the threads.
- The work unit is one element of an array. There is no way to change this default behaviour.
- The additional costs linked to such a work distribution can sometimes be significant.

```
program parallel
 implicit none
 integer, parameter   :: m=4097, n=513
 integer              :: i, j
 real, dimension(m,n) :: a, b

 call random_number(b)
 a(:,:) = 1.
 !$OMP PARALLEL
   !$OMP DO
     do j=1,n
       do i=1,m
          b(i,j) = b(i,j) - 0.5
       end do
     end do
   !$OMP END DO
   !$OMP WORKSHARE
     WHERE(b(:,:) >= 0.) a(:,:)=sqrt(b(:,:))
   !$OMP END WORKSHARE NOWAIT
 !$OMP END PARALLEL
end program parallel
```

☞ The PARALLEL WORKSHARE construct is a combination of the PARALLEL and WORKSHARE constructs, unifying their clauses and their respective constraints, with the exception of the NOWAIT clause at the end of the construct.

```
program parallel
 implicit none
 integer, parameter   :: m=4097, n=513
 real, dimension(m,n) :: a, b

 call random_number(b)
 !$OMP PARALLEL WORKSHARE
   a(:,:) = 1.
   b(:,:) = b(:,:) - 0.5
   WHERE(b(:,:) >= 0.) a(:,:)=sqrt(b(:,:))
 !$OMP END PARALLEL WORKSHARE
end program parallel
```

3.4 – Parallel sections

- A section is a portion of code executed by one and only one thread.
- Several code portions can be defined by the user by using the `SECTION` directive within a `SECTIONS` construct.
- The goal is to be able to distribute the execution of several independant code portions to different threads.
- The `NOWAIT` clause is accepted at the end of the construct (`END SECTIONS`) to remove the implicit synchronisation barrier.

3.4.1 – Construction SECTIONS

```
program parallel
 implicit none
 integer, parameter    :: n=513, m=4097
 real, dimension(m,n) :: a, b
 real, dimension(m)    :: coord_x
 real, dimension(n)    :: coord_y
 real                  :: pas_x, pas_y
 integer               :: i

 !$OMP PARALLEL
  !$OMP SECTIONS
   !$OMP SECTION
   call lecture_champ_initial_x(a)
   !$OMP SECTION
   call lecture_champ_initial_y(b)
   !$OMP SECTION
   pas_x      = 1./real(m-1)
   pas_y      = 2./real(n-1)
   coord_x(:) = (/ (real(i-1)*pas_x,i=1,m) /)
   coord_y(:) = (/ (real(i-1)*pas_y,i=1,n) /)
  !$OMP END SECTIONS NOWAIT
 !$OMP END PARALLEL
end program parallel
```

```
subroutine lecture_champ_initial_x(x)
 implicit none
 integer, parameter    :: n=513, m=4097
 real, dimension(m,n) :: x

 call random_number(x)
end subroutine lecture_champ_initial_x


subroutine lecture_champ_initial_y(y)
 implicit none
 integer, parameter    :: n=513, m=4097
 real, dimension(m,n) :: y

 call random_number(y)
end subroutine lecture_champ_initial_y
```

3.4.2 – Complementary information

- All the `SECTION` directives must appear in the lexical extent of the `SECTIONS` construct.
- The clauses accepted in the `SECTIONS` construct are those we already know:
  - `PRIVATE`
  - `FIRSTPRIVATE`
  - `LASTPRIVATE`
  - `REDUCTION`
- The `PARALLEL SECTIONS` directive is a fusion of the `PARALLEL` and `SECTIONS` directives, unifying their respective clauses.
- The `END PARALLEL SECTIONS` termination directive includes a global synchronisation barrier and cannot admit the `NOWAIT` clause .

3.5 – Exclusive execution

- It may occur that we want to exclude all the threads except one to execute certain code portions included in a parallel region.
- To do this, OpenMP offers two directives: SINGLE and MASTER.
- Although the desired goal is the same, the behaviour induced by these two constructs is fundamentally different.

3.5.1 – The SINGLE construct

- The SINGLE construct allows executing a portion of code by only one thread without being able to indicate which one.
- In general, it's the thread which arrives first on the SINGLE construct but this is not specified in the standard.
- All the threads which are not executing in the SINGLE region wait at the end of the construct END SINGLE until the thread executing has terminated, unless a NOWAIT clause was specified.

```
program parallel
 !$ use OMP_LIB
 implicit none
 integer :: rang
 real    :: a

 !$OMP PARALLEL DEFAULT(PRIVATE)
  a = 92290.

  !$OMP SINGLE
    a = -92290.
  !$OMP END SINGLE

  rang = OMP_GET_THREAD_NUM()
  print *,"Rang :",rang,"; A vaut :",a
 !$OMP END PARALLEL
end program parallel
```

```
> ifort ... -fopenmp prog.f90
> export OMP_NUM_THREADS=4 ; a.out
```

```
Rang : 1 ; A vaut :  92290.
Rang : 2 ; A vaut :  92290.
Rang : 0 ; A vaut :  92290.
Rang : 3 ; A vaut : -92290.
```

- A supplementary clause accepted only by the END SINGLE termination directive is the COPYPRIVATE clause.
- It allows the thread which is charged with executing the SINGLE region, to broadcast the value of a list of private variables to other threads before exiting this region.
- The other clauses accepted by the SINGLE directive are PRIVATE and FIRSTPRIVATE.

```
program parallel
  !$ use OMP_LIB
  implicit none
  integer :: rang
  real    :: a

  !$OMP PARALLEL DEFAULT(PRIVATE)
   a = 92290.

   !$OMP SINGLE
     a = -92290.
   !$OMP END SINGLE COPYPRIVATE(a)

   rang = OMP_GET_THREAD_NUM()
   print *,"Rang :",rang,"; A vaut :",a
  !$OMP END PARALLEL
end program parallel
```

```
> ifort ... -fopenmp prog.f90
> export OMP_NUM_THREADS=4 ; a.out
```

```
Rang : 1 ; A vaut : -92290.
Rang : 2 ; A vaut : -92290.
Rang : 0 ; A vaut : -92290.
Rang : 3 ; A vaut : -92290.
```

3.5.2 – The MASTER construct

- The MASTER construct allows the execution of a portion of code by the master thread only.
- This construct does not accept any clauses.
- No synchronisation barrier exists, neither at the beginning (MASTER) nor at the termination (END MASTER).

```
program parallel
 !$ use OMP_LIB
 implicit none
 integer :: rang
 real    :: a

 !$OMP PARALLEL DEFAULT(PRIVATE)
  a = 92290.

  !$OMP MASTER
    a = -92290.
  !$OMP END MASTER

  rang = OMP_GET_THREAD_NUM()
  print *,"Rang :",rang,"; A vaut :",a
 !$OMP END PARALLEL
end program parallel
```

```
> ifort ... -fopenmp prog.f90
> export OMP_NUM_THREADS=4 ; a.out
```

```
Rang : 0 ; A vaut : -92290.
Rang : 3 ; A vaut :  92290.
Rang : 2 ; A vaut :  92290.
Rang : 1 ; A vaut :  92290.
```

3.6 – Orphaned routines

- A routine (function or subprogram) called in a parallel region is executed by all the threads.
- In general, there is no interest in doing this if the work of the routine is not distributed.
- Distributing the work of a routine called in a parallel region requires the introduction of OpenMP directives (DO, SECTIONS, etc.) into the body of the routine.
- These directives are called « orphans » and, by linguistic extension, we speak of orphaned procedures (*orphaning*).
- A multithreaded scientific library parallelized with OpenMP will contain a set of orphaned routines.

```
> ls
> mat_vect.f90 prod_mat_vect.f90
```

```
program mat_vect
 implicit none
 integer,parameter   :: n=1025
 real,dimension(n,n) :: a
 real,dimension(n)   :: x, y
 call random_number(a)
 call random_number(x) ; y(:)=0.
 !$OMP PARALLEL IF(n.gt.256)
 call prod_mat_vect(a,x,y,n)
 !$OMP END PARALLEL
end program mat_vect
```

```
subroutine prod_mat_vect(a,x,y,n)
 implicit none
 integer,intent(in) :: n
 real,intent(in),dimension(n,n) :: a
 real,intent(in),dimension(n)   :: x
 real,intent(out),dimension(n)  :: y
 integer :: i
 !$OMP DO
 do i = 1, n
   y(i) = SUM(a(i,:) * x(:))
 end do
 !$OMP END DO
end subroutine prod_mat_vect
```

- Be careful: There are three execution contexts which are based on the compilation mode of the calling and called program units:
  - At compilation, the PARALLEL directive of the calling unit is interpreted (the execution can be Parallel) as well as the directives of the called unit (the work can be Distributed).
  - At compilation, the PARALLEL directive of the calling unit is interpreted (the execution can be Parallel) but not the directives contained in the called unit (the work can be Replicated).
  - At compilation, the PARALLEL directive of the calling unit is not interpreted. The execution is Sequential everywhere, even if the directives contained in the called unit were interpreted at the compilation.

compiled calling unit \ compiled called unit	with OpenMP	without OpenMP
with OpenMP	P + D	P + R
without OpenMP	S	S

Table 1 – Execution context based on compilation mode

3.7 – Summary

	default	shared	private	firstprivate	lastprivate	copyprivate	if	reduction	schedule	ordered	copyin	nowait
parallel	✓	✓	✓	✓			✓	✓			✓	
do			✓	✓	✓			✓	✓	✓		✓
sections			✓	✓	✓			✓				✓
workshare												✓
single			✓	✓		✓						✓
master												
threadprivate												

4 – Synchronisation

Synchronisation becomes necessary in the following situations:

❶ To ensure that all the concurrent threads have reached the same instruction point in the program (global barrier).

❷ To order the execution of all the concurrent threads when they need to execute the same code portion affecting one or more shared variables whose memory coherence (in read or write) must be guaranteed (mutual exclusion).

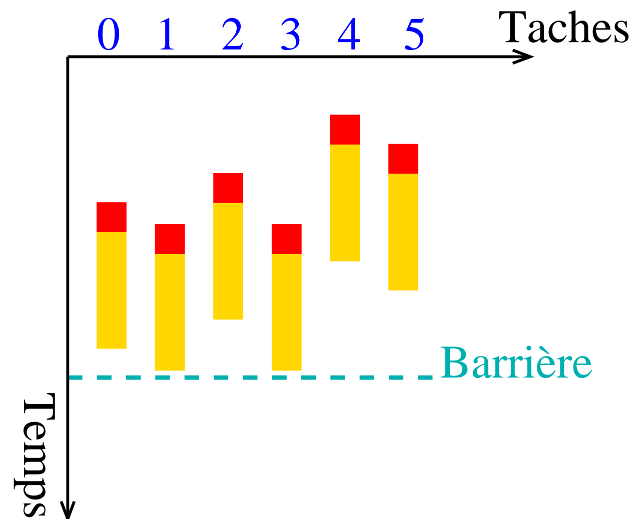
❸ To synchronise at least two concurrent threads among all the others (lock mechanism)

.

- As we have already indicated, the absence of a NOWAIT clause means that a global synchronisation barrier is implicitly applied at the end of the OpenMP construct. However, it is possible to explicitly impose a global synchronisation barrier by using the BARRIER directive.
- The mutual exclusion mechanism (one task at a time) is found, for example, in the reduction operations (REDUCTION clause) or in the ordered execution of a loop (DO ORDERED directive). This mechanism is also implemented in the ATOMIC and CRITICAL directives.
- Finer synchronisations can be done either by the implementation of lock mechanisms (requiring a call to OpenMP library subroutines) or by using the FLUSH directive.

4.1 – Barrier

- The BARRIER directive synchronises all the concurrent threads within a parallel region.
- Each thread waits until all the other threads reach this synchronisation point in order to continue, together, the execution of the program.




```
program parallel
 implicit none
 real,allocatable,dimension(:) :: a, b
 integer                       :: n, i
 n = 5
 !$OMP PARALLEL
  !$OMP SINGLE
    allocate(a(n),b(n))
  !$OMP END SINGLE
  !$OMP MASTER
    read(9) a(1:n)
  !$OMP END MASTER
  !$OMP BARRIER
  !$OMP DO SCHEDULE(STATIC)
    do i = 1, n
      b(i) = 2.*a(i)
    end do
  !$OMP SINGLE
    deallocate(a)
  !$OMP END SINGLE NOWAIT
 !$OMP END PARALLEL
 print *, "B vaut : ", b(1:n)
end program parallel
```

4.2 – Atomic update

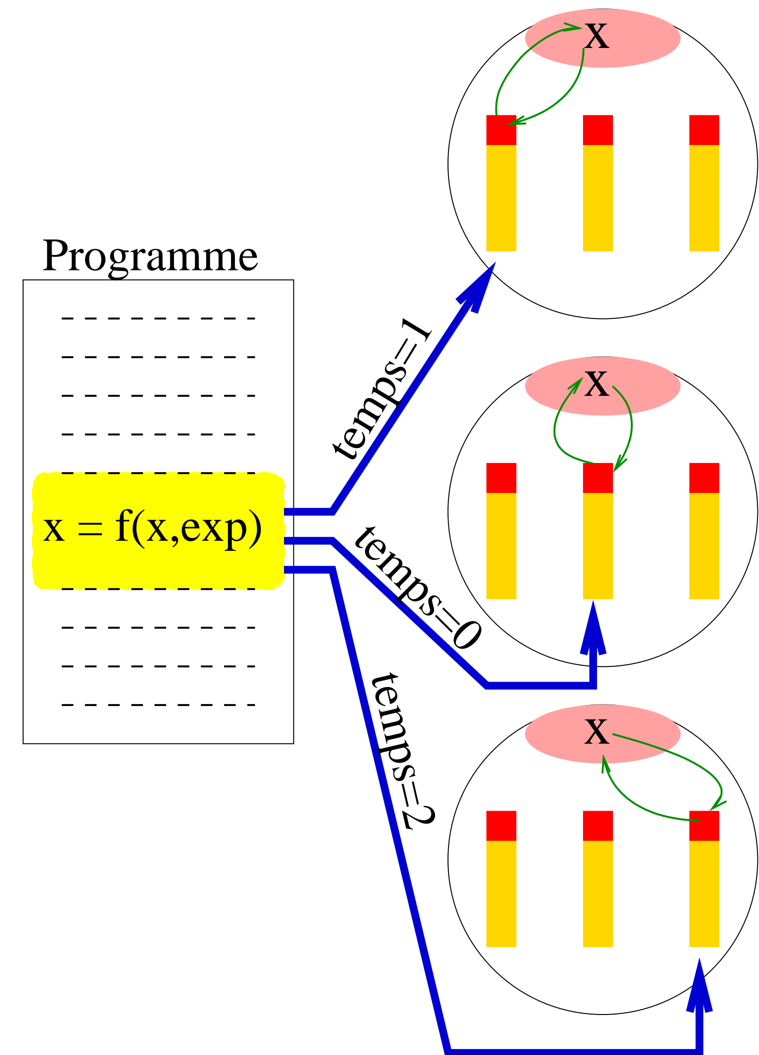
- The ATOMIC directive ensures that a shared variable is read and modified in memory by only one thread at a time.
- Its effect is limited to the instruction immediately following the directive.

```
program parallel
 !$ use OMP_LIB
 implicit none
 integer :: compteur, rang
 compteur = 92290
 !$OMP PARALLEL PRIVATE(rang)
  rang = OMP_GET_THREAD_NUM()
  !$OMP ATOMIC
   compteur = compteur + 1

  print *,"Rang :",rang, &
     "; compteur vaut :",compteur
 !$OMP END PARALLEL
 print *,"Au total, compteur vaut :", &
         compteur
end program parallel
```

```
Rang : 1 ; compteur vaut : 92291
Rang : 0 ; compteur vaut : 92292
Rang : 2 ; compteur vaut : 92293
Rang : 3 ; compteur vaut : 92294
Au total, compteur vaut : 92294
```

4 – Synchronisation: atomic update

- The instruction in question must have one of the following forms:
  - `x=x (op) exp`
  - `x=exp (op) x`
  - `x=f(x,exp)`
  - `x=f(exp,x)`
- `(op)` represents one of the following operations: `+`, `-`, $\times$, `/`, `.AND.`, `.OR.`, `.EQV.`, `.NEQV.`.
- `f` represents one of the following intrinsic functions: `MAX`, `MIN`, `IAND`, `IOR`, `IEOR`.
- `exp` is any arithmetic expression independent of `x`.



4.3 – Critical regions

- A critical region can be seen as a generalization of the `ATOMIC` directive although the underlying mechanisms are distinct.
- All the threads execute this region in a non-deterministic order but only one at a time.
- A critical region is delimited by the `CRITICAL`/`END CRITICAL` directives.
- Its extent is dynamic.
- For performance reasons, it is not recommended to emulate an atomic instruction by a critical region.
- An optional name can be given to a critical region.
- All critical regions which are not explicitly named are considered as having the same non-specified name.
- If several critical regions have the same name, the mutual exclusion mechanism considers them as being one and the same critical region.

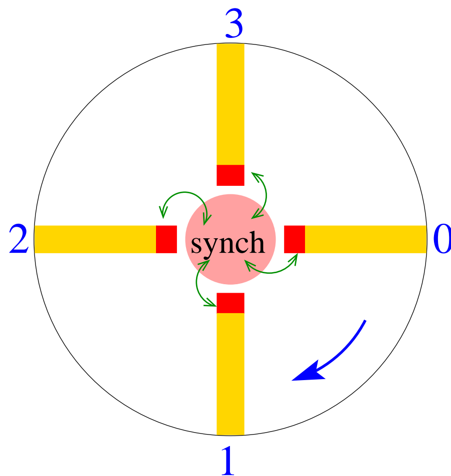
```fortran
program parallel
  implicit none
  integer :: s, p

  s=0
  p=1

  !$OMP PARALLEL
    !$OMP CRITICAL
      s = s + 1
    !$OMP END CRITICAL
    !$OMP CRITICAL (RC1)
      p = p * 2
    !$OMP END CRITICAL (RC1)
    !$OMP CRITICAL
      s = s + 1
    !$OMP END CRITICAL
  !$OMP END PARALLEL

  print *, "s= ",s, " ; p= ",p

end program parallel
```

```
> ifort ... -fopenmp prog.f90
> export OMP_NUM_THREADS=4 ; a.out
```

```
s= 8 ; p= 16
```

4.4 – The FLUSH directive

- This directive is useful in a parallel region to refresh the value of a shared variable in the global memory.
- It is even more useful if the machine has a hierarchical memory with multiple levels of caches.
- It can also serve to establish a synchronisation point mechanism between threads.




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

```
Rank :  1  ; synch :  0
Rank :  2  ; synch :  1
Rank :  3  ; synch :  2
Rank :  0  ; synch :  3
```

4.4.1 – Example: An easy trap

```
program ring2-wrong
 !$ use OMP_LIB
 implicit none
 integer :: rank,nb_threads,synch=0,counter=0
 !$OMP PARALLEL PRIVATE(rank,nb_threads)
   rank=OMP_GET_THREAD_NUM()
   nb_threads=OMP_GET_NUM_THREADS()
   if (rank == 0) then ; do
       !$OMP FLUSH(synch)
       if(synch == nb_threads-1) exit
     end do
   else ; do
       !$OMP FLUSH(synch)
       if(synch == rank-1) exit
     end do
   end if
   counter=counter+1
   print *,"Rank:",rank,";synch:",synch
   synch=rank
   !$OMP FLUSH(synch)
 !$OMP END PARALLEL
 print *,"Counter = ",counter
end program ring2-wrong
```

4.4.2 – Example: A difficult trap

```
program ring3-wrong
 !$ use OMP_LIB
 implicit none
 integer :: rank,nb_threads,synch=0,counter=0
 !$OMP PARALLEL PRIVATE(rank,nb_threads)
   rank=OMP_GET_THREAD_NUM(); nb_threads=OMP_GET_NUM_THREADS()
   if (rank == 0) then ; do
       !$OMP FLUSH(synch)
       if(synch == nb_threads-1) exit
     end do
   else ; do
       !$OMP FLUSH(synch)
       if(synch == rank-1) exit
     end do
   end if
   print *,"Rank:",rank,";synch:",synch
   !$OMP FLUSH(counter)
   counter=counter+1
   !$OMP FLUSH(counter)
   synch=rank
   !$OMP FLUSH(synch)
 !$OMP END PARALLEL
 print *,"Counter = ",counter
end program ring3-wrong
```

4.4.3 – Commentaries on the previous codes

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

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

and the lines,

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

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

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

4.4.4 – The correct code

```
program ring4
 !$ use OMP_LIB
 implicit none
 integer :: rank,nb_threads,synch=0,counter=0
 !$OMP PARALLEL PRIVATE(rank,nb_threads)
   rank=OMP_GET_THREAD_NUM()
   nb_threads=OMP_GET_NUM_THREADS()
   if (rank == 0) then ; do
       !$OMP FLUSH(synch)
       if(synch == nb_threads-1) exit
     end do
   else ; do
       !$OMP FLUSH(synch)
       if(synch == rank-1) exit
     end do
   end if
   print *,"Rank:",rank,";synch:",synch
   !$OMP FLUSH(counter)
   counter=counter+1
   !$OMP FLUSH(counter,synch)
   synch=rank
   !$OMP FLUSH(synch)
 !$OMP END PARALLEL
 print *,"Counter = ",counter
end program ring4
```

4.4.5 – Nested loops with double dependencies

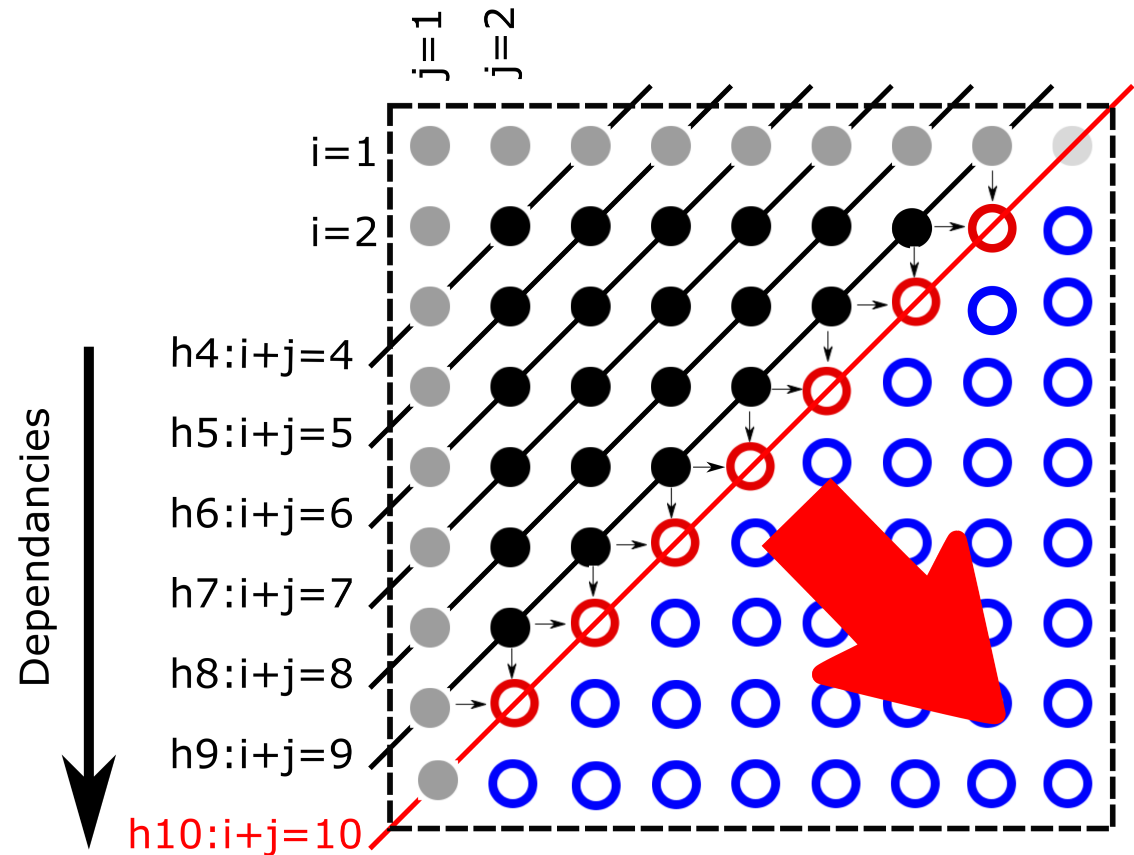
- Let us consider the following code:

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

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

Hyperplane Algorithm

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



Hyperplane Algorithm (2)

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

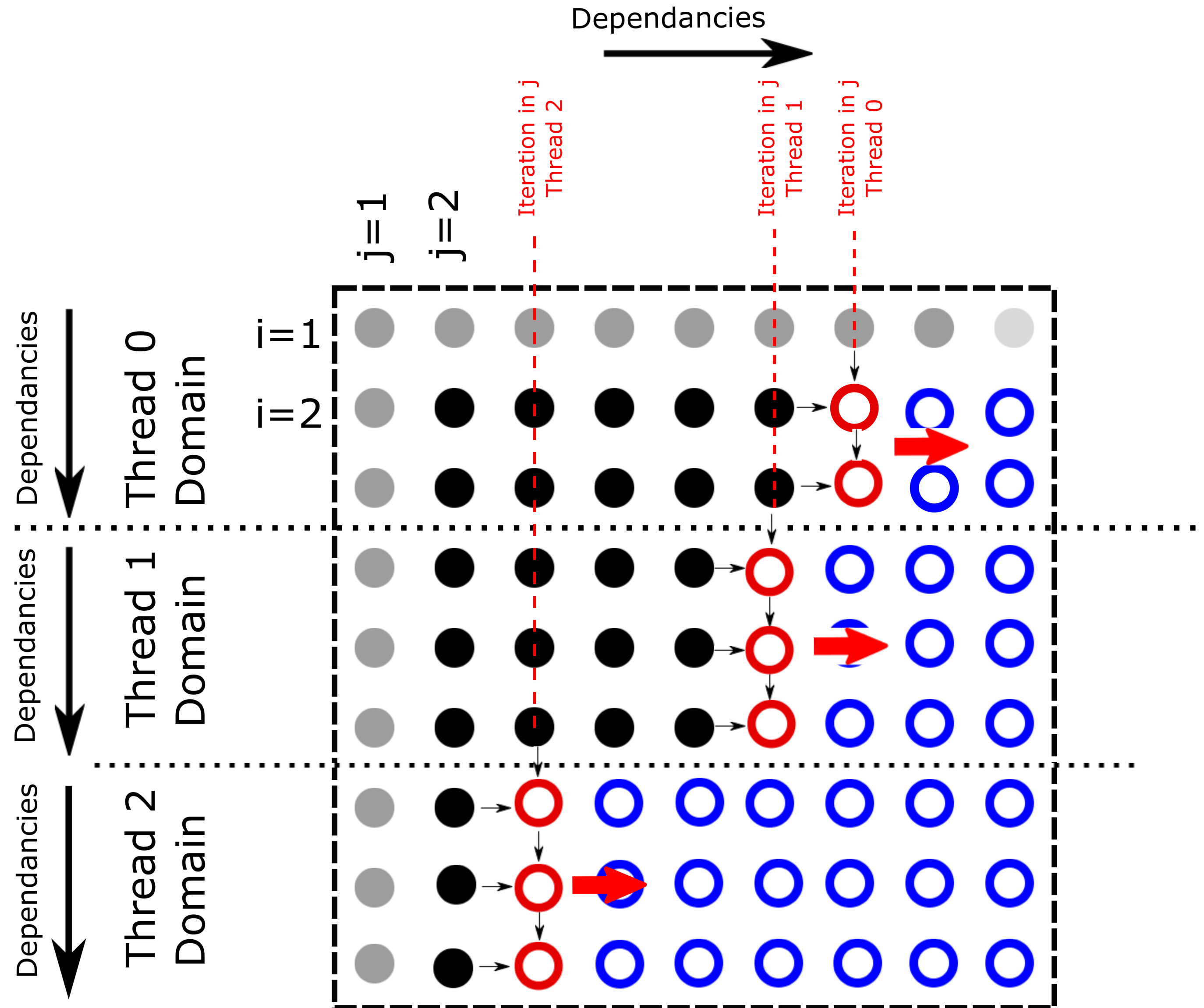
```
! Non // loop, dependencies between hyperplanes
do h = 1,nb_hyperplane
   ! compute i and j indices for the h hyperplane
   call calcul(INDI,INDJ,h)

   ! loop on the elements of the h hyperplane
   do e = 1,nb_element_hyperplane
      i = INDI(e)
      j = INDJ(e)
      V(i,j) =(V(i,j) + V(i-1,j) + V(i,j-1))/3  ! Update of V(i,j)
   enddo
enddo
```

- Once the code is rewritten, the parallelization is very simple and does not require to resort to low-level synchronizations.
- The performances obtained, unfortunately, are not optimal (poor use of caches due to the diagonal accesses, non-contiguous in memory).

Software Pipelining Algorithm

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

Software Pipelining Algorithm (2)



Software Pipelining Algorithm (3)

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

```
myOMPRank = ...
nbOMPThrds = ...

call compute_boundaries(iStart,iEnd)

do j= 2,n
   ! Thread is blocked (except 0) as long
   ! as the previous has not finished
   ! the treatment of the j+1 iteration
   call sync(myOMPRank,j)

   ! // loop distributed on the threads
   do i = iStart,iEnd
      ! Update of V(i,j)
      V(i,j) =(V(i,j) + V(i-1,j) + V(i,j-1))/3
   enddo
enddo
```

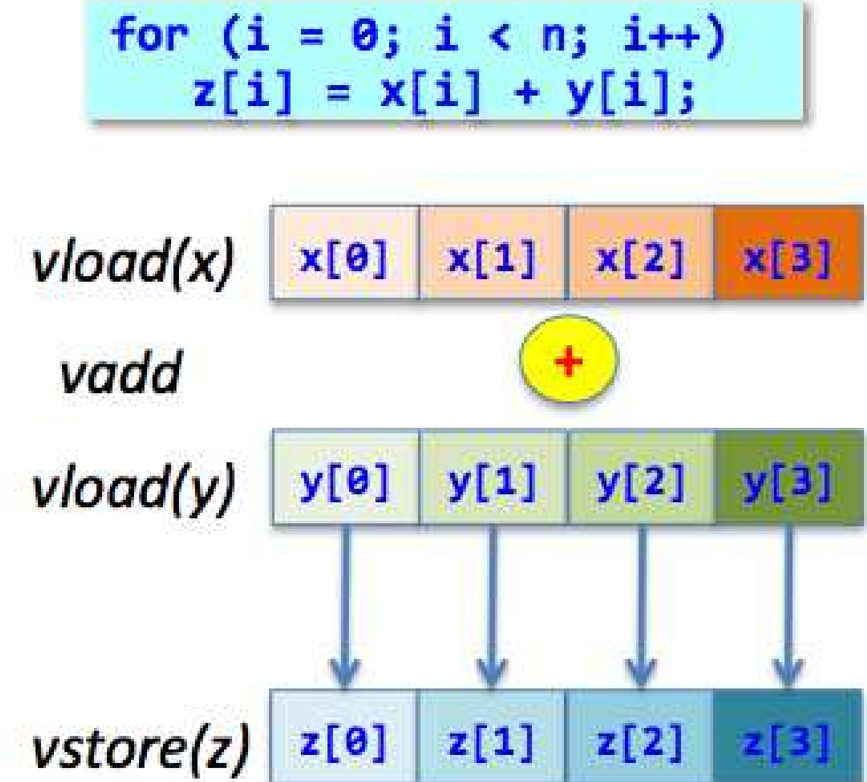
4.5 – Summary

	default	shared	private	firstprivate	lastprivate	copyprivate	if	reduction	schedule	ordered	copyin	nowait
parallel	✓	✓	✓	✓			✓	✓			✓	
do			✓	✓	✓			✓	✓	✓		✓
sections			✓	✓	✓			✓				✓
workshare												✓
single			✓	✓		✓						✓
master												
threadprivate												
atomic												
critical												
flush												

5 – SIMD Vectorisation

5.1 – Introduction

- SIMD = Single Instruction Multiple Data
- Only one instruction/operation acts in parallel on several elements.
- Before OpenMP 4.0, developers had to either rely on the know-how of the compiler or use proprietary extensions (directives or intrinsic functions).
- OpenMP 4.0 offers the possibility of managing the SIMD vectorisation in a portable and effective way by using the vector instructions available on the target architecture.



5.2 – SIMD loop vectorisation

- The SIMD directive allows cutting up the loop which immediately follows it into pieces of the same size as the vector registers available on the target architecture.
- The SIMD directive does not cause loop parallelisation.
- The SIMD directive can, therefore, be used inside or outside a parallel region.

```
program boucle_simd
implicit none
integer(kind=8) :: i
integer(kind=8), parameter :: n=500000
real(kind=8), dimension(n) :: A, B
real(kind=8)               :: somme
...
somme=0
!$OMP SIMD REDUCTION(+:somme)
do i=1,n
   somme=somme+A(i)*B(i)
enddo
...
end program boucle_simd
```

5.3 – Parallelisation and SIMD vectorisation of a loop

- The DO SIMD construct is a fusion of the DO and SIMD directives, unifying their respective clauses.
- This construct allows worksharing and vectorising the processing of loop iterations.
- The iteration packets are distributed to the threads according to the chosen distribution mode. Each thread vectorises the processing of its packet by subdividing it into iteration blocks of the same size as the vector registers; the blocks will be treated one after another with the vector instructions.
- The PARALLEL DO SIMD directive also allows creating the parallel region.

```
program boucle_simd
implicit none
integer(kind=8) :: i
integer(kind=8), parameter :: n=500000
real(kind=8), dimension(n) :: A, B
real(kind=8)               :: somme
...
somme=0
!$OMP PARALLEL DO SIMD REDUCTION(+:somme)
do i=1,n
   somme=somme+A(i)*B(i)
enddo
...
end program boucle_simd
```

5.4 – SIMD vectorisation of scalar functions

- The goal is to automatically create a vector version of scalar functions. Functions generated in this way can be called inside the vectorised loops without breaking the vectorisation.
- The vector version of the function allows processing the iterations by block instead of one after another.
- The DECLARE SIMD directive allows generating a vector version in addition to the scalar version of the function in which it is declared.

```
program boucle_fonction_simd
implicit none
integer, parameter :: n=1000
integer :: i
real, dimension(n) :: A, B
real :: dist_max
...
dist_max=0
!$OMP PARALLEL DO SIMD REDUCTION(max:dist_max)
do i=1,n
   dist_max=max(dist_max,dist(A(i),B(i)))
enddo
!$OMP END PARALLEL DO SIMD

print *,"Distance maximum = ",dist_max

contains

real function dist(x,y)
  !$OMP DECLARE SIMD (dist)
  real, intent(in) :: x, y
  dist=sqrt(x*x+y*y)
end function dist

end program boucle_fonction_simd
```

6 – OpenMP tasks

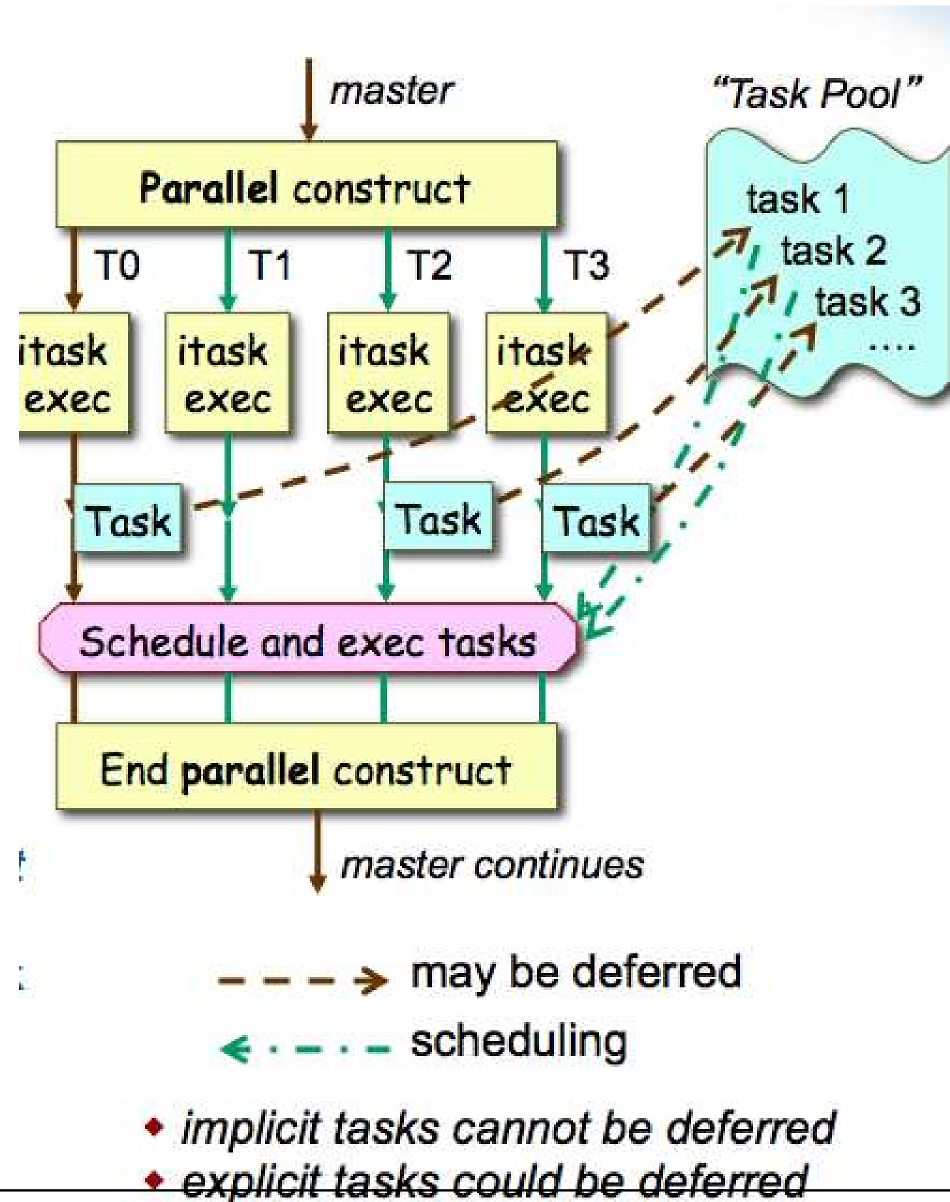
6.1 – Introduction

- The « *fork and join* » model associated with the worksharing constructs is limiting.
- In particular, it is not adapted to dynamic problems (while-loops, parallel research in a tree, etc.) or to recursive algorithms.
- A new model based on the notion of tasks was introduced with the `OpenMP` 3.0 version. It is complementary to the model which is uniquely based on threads.
- It allows the expression of parallelism for recursive or pointer-based algorithms, commonly used in C/C++.
- The `OpenMP` 4.0 version allows managing constructs of creation and synchronisation of explicit tasks (with or without dependencies).

6.2 – The concept bases

- An `OpenMP` task consists of an executable code instance and its associated data. It is executed by one thread.
- Two types of tasks exist:
  - Implicit tasks generated by the `PARALLEL` directive
  - Explicit tasks generated by the `TASK` directive
- Several types of synchronisation are available:
  - For a given task, the `TASKWAIT` directive allows waiting for the termination of all its child tasks (first generation).
  - The `TASKGROUP/END TASKGROUP` directive allows waiting for the termination of all the descendants of a group of tasks.
  - Implicit or explicit barriers allow waiting for the termination of all the explicit tasks already created.
- The DSA of a variable is relative to one task except for the `THREADPRIVATE` directive which is associated with the thread notion.

6.3 – The task execution model



- Execution begins with only the master thread.
- On encountering a parallel region (PARALLEL) :
  - Creation of a team of threads
  - Creation of the implicit tasks, one per thread, each thread executing its implicit task
- On encountering a workshare construct:
  - Distribution of the work to threads (or to implicit tasks)
- On encountering a TASK construct:
  - Creation of explicit tasks
  - The execution of these explicit tasks can be deferred.
- Execution of explicit tasks:
  - At the task scheduling points (TASK, TASKWAIT, BARRIER), the available threads begin executing the waiting tasks.
  - A thread can switch from the execution of one task to another one.
- At the end of the parallel region:
  - All the tasks finish their execution.
  - Only the master thread continues with the execution of the sequential part.

6.4 – Some examples

```
program task_print
implicit none

print *,"Un "
print *,"grand "
print *,"homme "

end program task_print
```

```
> ifort ... -fopenmp task_print.f90
> export OMP_NUM_THREADS=2 ; a.out
```

```
Un
grand
homme
```

```
program task_print
implicit none

!$OMP PARALLEL
print *,"Un "
print *,"grand "
print *,"homme "
!$OMP END PARALLEL

end program task_print
```

```
> ifort ... -fopenmp task_print.f90
> export OMP_NUM_THREADS=2 ; a.out
```

```
Un
grand
Un
homme
grand
homme
```

```
program task_print
implicit none
!$OMP PARALLEL
!$OMP SINGLE
print *,"Un "
print *,"grand "
print *,"homme "
!$OMP END SINGLE
!$OMP END PARALLEL
end program task_print
```

```
> ifort ... -fopenmp task_print.f90
> export OMP_NUM_THREADS=2 ; a.out
```

```
Un
grand
homme
```

```
program task_print
implicit none
!$OMP PARALLEL
!$OMP SINGLE
print *,"Un "
!$OMP TASK
print *,"grand "
!$OMP END TASK
!$OMP TASK
print *,"homme "
!$OMP END TASK
!$OMP END SINGLE
!$OMP END PARALLEL
end program task_print
```

```
> ifort ... -fopenmp task_print.f90
> export OMP_NUM_THREADS=2 ; a.out; a.out
```

```
Un
grand
homme
```

```
Un
homme
grand
```

- The tasks can be executed in any order...
- How to always terminate the phrase with « a marche sur la lune » ?

- If we add *print* just before the end of the SINGLE region, it doesn't work!
- In fact, the explicit tasks are only executable at the task scheduling points of the code (TASK, TASKWAIT, BARRIER), ...

```
program task_print
implicit none
!$OMP PARALLEL
!$OMP SINGLE
print *,"Un "
!$OMP TASK
print *,"grand "
!$OMP END TASK
!$OMP TASK
print *,"homme "
!$OMP END TASK
print *,"a marche sur la lune"
!$OMP END SINGLE
!$OMP END PARALLEL
end program task_print
```

```
> ifort ... -fopenmp task_print.f90
> export OMP_NUM_THREADS=2 ; a.out; a.out
```

```
Un
a marche sur la lune
homme
grand
```

```
Un
grand
a marche sur la lune
homme
```

- The solution consists of introducing a task scheduling point with the TASKWAIT directive to execute the explicit tasks, then waiting for them to terminate before continuing.
- If you want to impose an order between "grand" and "homme", you need to use the DEPEND clause introduced in OpenMP 4.0.

```
program task_print
implicit none
!$OMP PARALLEL
!$OMP SINGLE
print *,"Un "
!$OMP TASK
print *,"grand "
!$OMP END TASK
!$OMP TASK
print *,"homme "
!$OMP END TASK
!$OMP TASKWAIT
print *,"a marche sur la lune"
!$OMP END SINGLE
!$OMP END PARALLEL
end program task_print
```

```
> ifort ... -fopenmp task_print.f90
> export OMP_NUM_THREADS=2 ; a.out; a.out
```

```
Un
homme
grand
a marche sur la lune
```

```
Un
grand
homme
a marche sur la lune
```

6.5 – Dependency between tasks

- The `DEPEND(type_dependance:list)` clause allows managing the dependencies between explicit tasks having the same parent (i.e. generated by the same task).
- A *T1* task which depends on the *T2* task cannot begin to execute until the *T2* task has terminated.
- There are three types of dependencies:
  - `IN` : The generated task will be dependent on all the preceding tasks generated by the same parent and which have at least one element in common in the `OUT` or `INOUT` dependency lists.
  - `INOUT` and `OUT`: The generated task will be dependent on all the preceding tasks generated by the same parent which have at least one element in common in the `IN`, `OUT` or `INOUT` dependency lists.
- The variables of the `DEPEND` directive list are associated with a memory address and can be an element of an array or a section of an array.

- Here we introduce a dependency between explicit tasks so that the *T1 : print *,"grand"* task executes before the *T2 : print *,"homme"* task.
- We can, for example, use the `DEPEND(OUT:T1)` clause for the *T1* task and `DEPEND(IN:T1)` for the *T2* task.

```
program task_print
implicit none
integer :: T1
!$OMP PARALLEL
!$OMP SINGLE
print *,"Un "
!$OMP TASK DEPEND(OUT:T1)
print *,"grand "
!$OMP END TASK
!$OMP TASK DEPEND(IN:T1)
print *,"homme "
!$OMP END TASK
!$OMP TASKWAIT
print *,"a marche sur la lune"
!$OMP END SINGLE
!$OMP END PARALLEL
end program task_print
```

```
> ifort ... -fopenmp task_print.f90
> export OMP_NUM_THREADS=2 ; a.out
```

```
Un
grand
homme
a marche sur la lune
```

6.6 – Data-sharing attributes of variables in the tasks

- The default DSA of the variables is:
  - `SHARED` for the implicit tasks.
  - For the explicit tasks:
    - If the variable is `SHARED` in the parent task, then it inherits the `SHARED` DSA.
    - In the other cases, the default DSA is `FIRSTPRIVATE`.
- When creating the task, you can use the clauses `SHARED(list)`, `PRIVATE(list)`, `FIRSTPRIVATE(list)` or `DEFAULT(PRIVATE|FIRSTPRIVATE|SHARED|NONE)` (`DEFAULT(PRIVATE|NONE)` in C/C++ only) to explicitly specify the DSA of the variables which lexically appear in the task.

6.7 – Example of updating elements of a linked list

- How to update all the elements of a linked list in parallel ...

```
type element
integer :: valeur
type(element), pointer :: next
end type element

subroutine increment_lst_ch(debut)
type(element), pointer :: debut, p
p=>debut
do while (associated(p))
p%valeur=p%valeur+1
p=>p%next
end do
end subroutine increment_lst_ch
```

- Producer/consumer pattern (thread which executes the `single` region/the other threads).

```
subroutine increment_lst_ch(debut)
type(element), pointer :: debut, p
!$OMP PARALLEL PRIVATE(p)
!$OMP SINGLE
p=>debut
do while (associated(p))
!$OMP TASK
p%valeur=p%valeur+1
!$OMP END TASK
p=>p%next
end do
!$OMP END SINGLE
!$OMP END PARALLEL
end subroutine increment_lst_ch
```

- The DSA of the p variable inside the explicit task is `FIRSTPRIVATE` by default and this is the desired DSA.

6.8 – Example of a recursive algorithm

- The Fibonacci sequence is defined by: f(0)=0; f(1)=1; f(n)=f(n-1)+f(n-2).
- The code builds a binary tree. The parallelism comes from processing the leaves of this tree in parallel.
- Only one thread will generate the tasks but all the threads will participate in the execution.
- Pay attention to the DSA of the variables in this example: The default DSA (i.e. FIRSTPRIVATE) would give false results. It is necessary for i and j to be shared in order to recover the result in the parent task.
- Attention, the TASKWAIT directive is also mandatory to ensure that the i and j calculations will be terminated before returning the result.
- This version is not efficient.

```
program fib_rec
integer, parameter :: nn=10
integer :: res_fib
!$OMP PARALLEL
!$OMP SINGLE
res_fib=fib(nn)
!$OMP END SINGLE
!$OMP END PARALLEL
print *,"res_fib = ",res_fib
contains
recursive integer function fib(n) &
result(res)
integer, intent(in) :: n
integer :: i, j
if (n<2) then res = n
else
!$OMP TASK SHARED(i)
i=fib(n-1)
!$OMP END TASK
!$OMP TASK SHARED(j)
j=fib(n-2)
!$OMP END TASK
!$OMP TASKWAIT
res=i+j
endif
end function fib
end program fib_rec
```

6.9 – The FINAL and MERGEABLE clauses

- In the case of « *Divide and Conquer* » recursive algorithms, the work volume of each task (i.e. the granularity) decreases during the execution. This is the main reason why the preceding code is not efficient.
- The FINAL and MERGEABLE clauses are, therefore, very useful: They allow the compiler to fusion the newly created tasks.
- Unfortunately, these functionalities are very rarely implemented in an efficient way, so it would be better to manually implement « *cut off* » in the code.

6.10 – The TASKGROUP synchronisation

- The TASKGROUP construct allows defining a group of tasks and waiting at the end of the construct for all the tasks and their descendants to have finished executing.
- In this example, we will customise a task which will run a background computation while multiple iterations of the traversal of a binary tree are launched in parallel. At the end of each iteration, we synchronise the tasks which were generated for the tree traversal, and only those tasks.

```
module arbre_mod
type type_arbre
type(type_arbre), pointer :: fg, fd
end type
contains
subroutine traitement_feuille(feuille)
type(type_arbre), pointer :: feuille
! Traitement...
end subroutine traitement_feuille
recursive subroutine traverse_arbre(arbre)
type(type_arbre), pointer :: arbre
if (associated(arbre%fg)) then
!$OMP TASK
call traverse_arbre(arbre%fg)
!$OMP END TASK
endif
if (associated(arbre%fd)) then
!$OMP TASK
call traverse_arbre(arbre%fd)
!$OMP END TASK
endif
!$OMP TASK
call traitement_feuille(arbre)
!$OMP TASK
end subroutine traverse_arbre
end module arbre_mod
```

```
program principal
use arbre_mod
type(type_arbre), pointer :: mon_arbre
integer, parameter :: niter=100
call init_arbre(mon_arbre)
!$OMP PARALLEL
!$OMP SINGLE
!$OMP TASK
call travail_tache_de_fond()
!$OMP END TASK
do i=1, niter
!$OMP TASKGROUP
!$OMP TASK
call traverse_arbre(mon_arbre)
!$OMP END TASK
!$OMP END TASKGROUP
enddo
!$OMP END SINGLE
!$OMP END PARALLEL
end program principal
```

7 – Affinities

7.1 – Thread affinity

- The exploitation system chooses the execution core of a thread by default. The execution core can change during the execution but not without a significant penalty.
- To remedy this problem, it is possible to explicitly associate a thread to a core (*binding*) during the entire duration of the execution.
- With the GNU compilers, execution with thread/core binding can be done with the *GOMP_CPU_AFFINITY* environment variable.
- With the Intel compilers, execution with thread/core binding can be done with the *KMP_AFFINITY* (cf. Intel Thread Affinity Interface) environment variable.
- Since `OpenMP`4.0, execution with thread/core binding can be achieved in a portable way using the *OMP_PROC_BIND* and *OMP_PLACES* environment variables.

7.1.1 – The *cpuinfo* command

☞ The *cpuinfo* command provides much information about the topology of the execution node (the number and numbering of the sockets, the physical and logical cores, activation of hyper-threading or not, etc.).

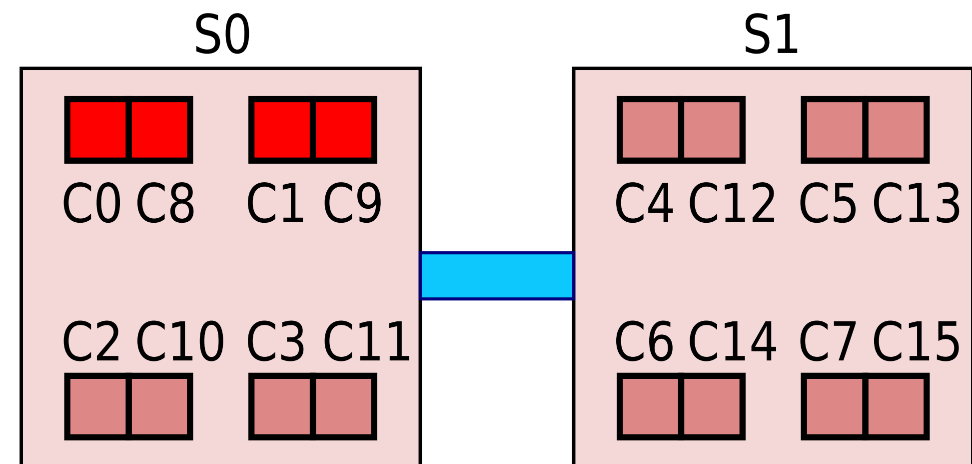
```
> cpuinfo     <= Example on an SMP node without hyper-threading activated
Intel(R) Processor information utility, Version 4.1.0 Build 20120831
Copyright (C) 2005-2012 Intel Corporation.  All rights reserved.

=====  Processor composition  =====
Processor name    : Intel(R) Xeon(R)  E5-4650 0
Packages(sockets) : 4      <= Nb de sockets du noeud
Cores             : 32     <= Nb de coeurs physiques du noeud
Processors(CPUs)  : 32     <= Nb de coeurs logiques du noeud
Cores per package : 8      <= Nb de coeurs physiques par socket
Threads per core  : 1      <= Nb de coeurs logiques par coeur physique,hyperthreading actif si valeur >1

=====  Processor identification  =====
Processor       Thread Id.      Core Id.        Package Id.
0               0               0               0
1               0               1               0
2               0               2               0
3               0               3               0
4               0               4               0
5               0               5               0
6               0               6               0
7               0               7               0
8               0               0               1
9               0               1               1
...             ...             ...             ...
30              0               6               3
31              0               7               3
=====  Placement on packages  =====
Package Id.     Core Id.        Processors
0               0,1,2,3,4,5,6,7         0,1,2,3,4,5,6,7
1               0,1,2,3,4,5,6,7         8,9,10,11,12,13,14,15
2               0,1,2,3,4,5,6,7         16,17,18,19,20,21,22,23
3               0,1,2,3,4,5,6,7         24,25,26,27,28,29,30,31

=====  Cache sharing  =====
Cache   Size            Processors
L1      32  KB          no sharing
L2      256 KB          no sharing
L3      20  MB          (0,1,2,3,4,5,6,7)(8,9,10,11,12,13,14,15)(16,17,18,19,20,21,22,23)(24,25,26,27,28,29,30,31)
```

```
> cpuinfo     <= Example on an SMP node with hyper-threading activated
Intel(R) Processor information utility, Version 4.1.0 Build 20120831
Copyright (C) 2005-2012 Intel Corporation.  All rights reserved.

=====  Processor composition  =====
Processor name    : Intel(R) Xeon(R)  E5-4650 0
Packages(sockets) : 4     <= Nb de sockets du noeud
Cores             : 32    <= Nb de coeurs physiques du noeud
Processors(CPUs)  : 64    <= Nb de coeurs logiques du noeud
Cores per package : 8     <= Nb de coeurs physiques par socket
Threads per core  : 2     <= Nb de coeurs logiques par coeur physique,hyperthreading actif si valeur >1

=====  Processor identification  =====
Processor       Thread Id.      Core Id.        Package Id.
0               0               0               0
1               0               1               0
2               0               2               0
3               0               3               0
4               0               4               0
5               0               5               0
6               0               6               0
7               0               7               0
8               0               0               1
9               0               1               1
10              0               2               1
...
54              1               6               2
55              1               7               2
56              1               0               3
57              1               1               3
58              1               2               3
59              1               3               3
60              1               4               3
61              1               5               3
62              1               6               3
63              1               7               3
=====  Placement on packages  =====
Package Id.     Core Id.        Processors
0               0,1,2,3,4,5,6,7         (0,32)(1,33)(2,34)(3,35)(4,36)(5,37)(6,38)(7,39)
1               0,1,2,3,4,5,6,7         (8,40)(9,41)(10,42)(11,43)(12,44)(13,45)(14,46)(15,47)
2               0,1,2,3,4,5,6,7         (16,48)(17,49)(18,50)(19,51)(20,52)(21,53)(22,54)(23,55)
3               0,1,2,3,4,5,6,7         (24,56)(25,57)(26,58)(27,59)(28,60)(29,61)(30,62)(31,63)

=====  Cache sharing  =====
Cache   Size            Processors
L1      32  KB          (0,32)(1,33)(2,34)(3,35)(4,36)(5,37)(6,38)(7,39)(8,40)(9,41)(10,42)(11,43)(12,44)(13,45)(14,46)(15,47)
(16,48)(17,49)(18,50)(19,51)(20,52)(21,53)(22,54)(23,55)(24,56)(25,57)(26,58)(27,59)(28,60)(29,61)(30,62)(31,63)
L2      256 KB          (0,32)(1,33)(2,34)(3,35)(4,36)(5,37)(6,38)(7,39)(8,40)(9,41)(10,42)(11,43)(12,44)(13,45)(14,46)(15,47)
(16,48)(17,49)(18,50)(19,51)(20,52)(21,53)(22,54)(23,55)(24,56)(25,57)(26,58)(27,59)(28,60)(29,61)(30,62)(31,63)
L3      20  MB          (0,1,2,3,4,5,6,7,32,33,34,35,36,37,38,39)(8,9,10,11,12,13,14,15,40,41,42,43,44,45,46,47)
(16,17,18,19,20,21,22,23,48,49,50,51,52,53,54,55)(24,25,26,27,28,29,30,31,56,57,58,59,60,61,62,63)
```

7.1.2 – Use of the *KMP_AFFINITY* environment variable

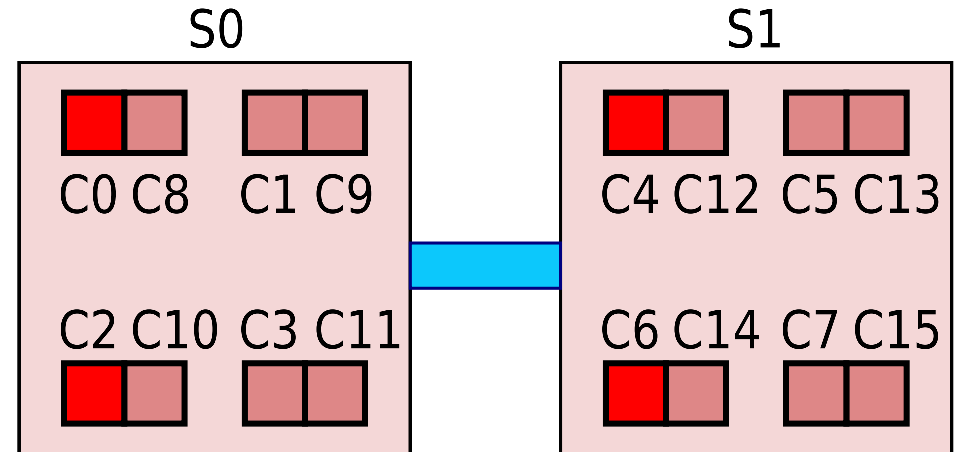
The principal modes of thread/core binding are the following:

- *Compact* mode: The threads of consecutive numbers are binded on logical or physical cores (depending on if hyper-threading is activated or not) and they are as close as possible to each other, thereby reducing cache and TLB (Translation lookaside buffer) misses.

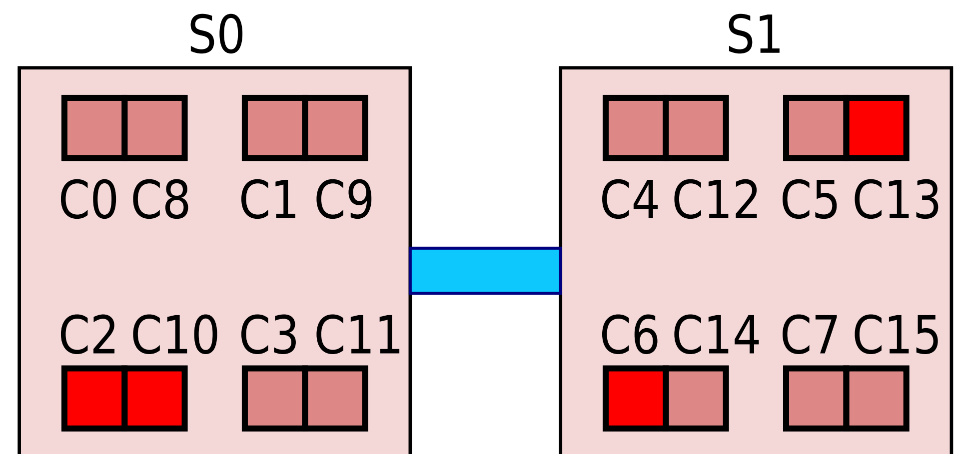


Example on a bi-socket, quad-core architecture with hyper-threading activated.

☞ *Scatter* mode: Contrary to the *compact* mode, threads of consecutive numbers are binded on logical or physical cores (depending on if hyper-threading is activated or not) which are as far away as possible from each other.



☞ *Explicit* mode: You need to explicitly define the binding of the threads on the logical or physical cores.



7.1.3 – Thread affinity with OpenMP 4.0

- OpenMP 4.0 introduces the notion of *places* which specifies groups of logical or physical cores which will be binded to a thread execution.
- *Places* can be specified explicitly by the intermediary of a list, or directly by using the following keywords:
  - Threads : Each *place* is associated with one logical core of the machine.
  - Cores : Each *place* is associated with one physical core of the machine.
  - Sockets : Each *place* is associated with one socket of the machine.
- Examples for a dual-socket quad-core architecture with hyper-threading:
  - OMP_PLACES= threads : 16 *places* associated with one logical core.
  - OMP_PLACES= "threads(4)" : 4 *places* associated with one logical core.
  - OMP_PLACES= "{0,8,1,9},{6,14,7,15}" : 2 *places*, the first place on the first socket and the second place on the second socket.

☞ The PROC_BIND clause of the PARALLEL construct or the OMP_PROC_BIND environment variable allows choosing from the following affinities:

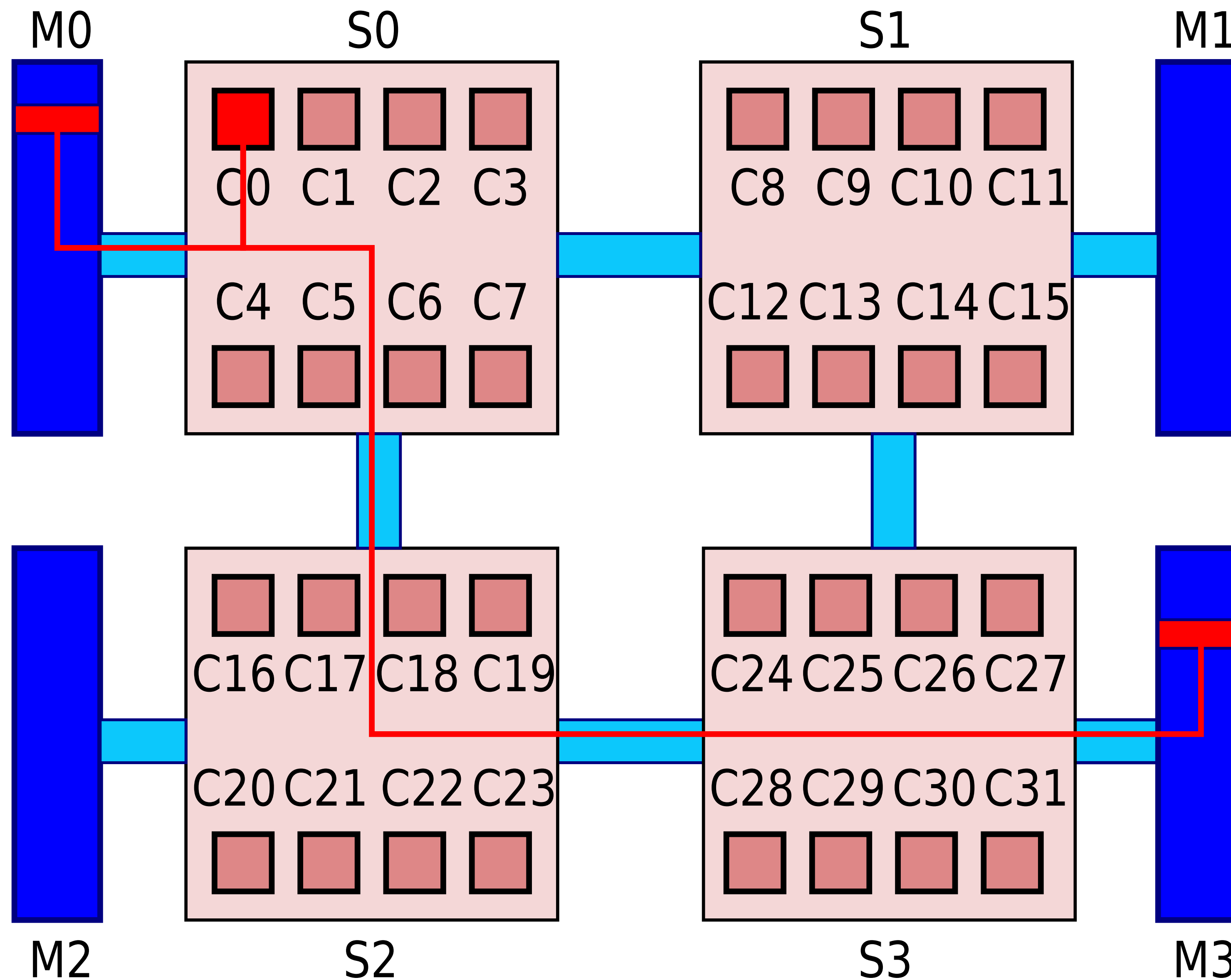
- MASTER (PRIMARY in OpenMP 5): The threads run on the same *place* as the master thread.
- CLOSE: Distribute the threads on partitions as close as possible to the *place* of the master thread.
- SPREAD: Equitable thread distribution on the defined *places*.

```
export OMP_PLACES="{0,8,1,9},{2,10,3,11},{4,12,5,13},{6,14,7,15}"
Soit 4 places p0={0,8,1,9}, p1={2,10,3,11}, p2={4,12,5,13} et p3={6,14,7,15}
!$OMP PARALLEL PROC_BIND(SPREAD) NUM_THREADS(2)
   !$OMP PARALLEL PROC_BIND(CLOSE) NUM_THREADS(4)
....
Dans la premiere region parallele
Th0 s'executera sur p0 avec une partition de place =p0p1
Th1 s'executera sur p2 avec une partition de place =p2p3
Dans la seconde region parallele
Th00 et Th01 s'executeront sur p0
Th02 et Th03 s'executeront sur p1
Th10 et Th11 s'executeront sur p2
Th12 et Th13 s'executeront sur p3
```

7.2 – Memory affinity

- The modern multi-socket nodes have strong `NUMA` (*Non-Uniform Memory Access*) effects; access time to data varies according to the memory bank location where it is stored.
- The memory storage location of the shared variables (in the local memory of the socket executing the thread or the distant memory of another socket) will strongly influence the code performance.
- The operating system tries to optimise this memory allocation process by favouring, whenever possible, allocation to the local memory of the socket responsible for the thread execution. This is what is called *memory affinity*.

Machine architecture having strong `NUMA` effects (quad-sockets, octo-cores):



- ☞ For the arrays, memory allocation is actually done at the execution, page by page, during the first access to an element of this array.
- ☞ Depending on the code characteristics (memory bound, CPU bound, random memory access, memory access by a frequently used dimension, etc.), you should either regroup all the threads within the same socket (*compact* distribution mode) or, to the contrary, distribute them on various available sockets (*scatter* distribution mode).
- ☞ In general, we try to regroup threads which are working on the same shared data on the same socket.

7.3 – A « *First Touch* » strategy

- To optimise memory affinity in an application, it is very strongly recommended to implement a « *First Touch* » strategy: Each thread initialises the part of the shared data on which it will subsequently work.
- If the threads are bound, memory access is optimised by improving data locality.
- Advantage: There are substantial performance gains.
- Disadvantages:
  - No gain can be expected with `DYNAMIC` or `GUIDED` scheduling, or with the `WORKSHARE` directive.
  - No gain can be expected if the parallelisation uses the concept of explicit tasks.

7.4 – Examples of impact on performance

☞ « *Memory Bound* » code running with 4 threads on private data

```
program SBP
...
!$OMP PARALLEL PRIVATE(A,B,C)
do i=1,n
   A(i) = A(i)*B(i)+C(i)
enddo
!$OMP END PARALLEL
...
end program SBP
```

```
> export OMP_NUM_THREADS=4
> export KMP_AFFINITY=compact
> a.out
```

```
> export OMP_NUM_THREADS=4
> export KMP_AFFINITY=scatter
> a.out
```

```
Temps elapsed = 116 s.
```

```
Temps elapsed = 49 s.
```

☞ To optimise the use of the 4 memory buses, it is preferable to bind one thread per socket. In this case, the *scatter* mode is 2.4 times more efficient than the *compact* mode!

☞ Example without « *First Touch* »

```
program NoFirstTouch
 implicit none
 integer, parameter   :: n = 30000
 integer              :: i, j
 real, dimension(n,n) :: TAB

 ! Initialisation of TAB
 TAB(1:n,1:n)=1.0

 !$OMP PARALLEL
 ! Calcul sur TAB
 !$OMP DO SCHEDULE(STATIC)
 do j=1,n
    do i=1,n
       TAB(i,j)=TAB(i,j)+i+j
    enddo
 enddo
 !$OMP END PARALLEL
end program NoFirstTouch
```

```
> export OMP_NUM_THREADS=32 ; a.out
```

```
Elapsed time = 98.35 s.
```

☞ Example with « *First Touch* »

```
program FirstTouch
 implicit none
 integer, parameter   :: n = 30000
 integer              :: i, j
 real, dimension(n,n) :: TAB
 !$OMP PARALLEL
 ! Initialisation de TAB
 !$OMP DO SCHEDULE(STATIC)
 do j=1,n
   TAB(1:n,j)=1.0
 enddo
 ! Calcul sur TAB
 !$OMP DO SCHEDULE(STATIC)
 do j=1,n
    do i=1,n
       TAB(i,j)=TAB(i,j)+i+j
    enddo
 enddo
 !$OMP END PARALLEL
end program FirstTouch
```

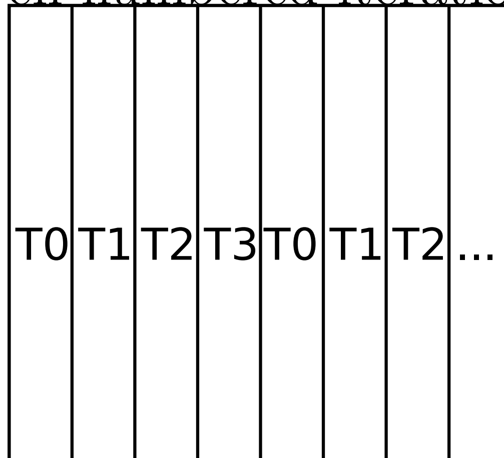
```
> export OMP_NUM_THREADS=32 ; a.out
```

```
Elapsed time = 10.22 s.
```

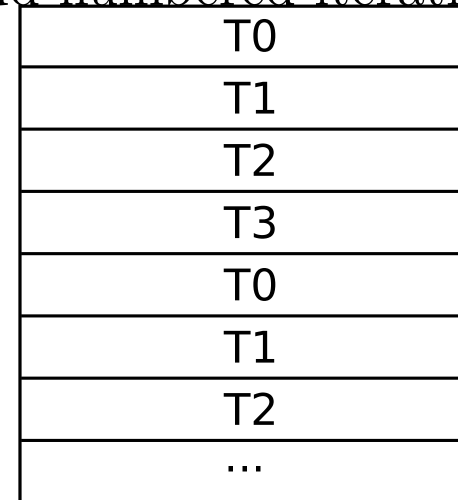
☞ Using the « *First Touch* » strategy allows a gain of about a factor of 10 in this ex.

- An « alternating directions » code runs with 4 threads on a 2D shared array, fitting into the L3 cache of a socket. This is an example for which there is no *execution thread/data* locality.
  - In even-numbered iterations, each thread works on the columns of the shared array.
  - In odd-numbered iterations, each thread works on the lines of the shared array.

Even-numbered iterations:

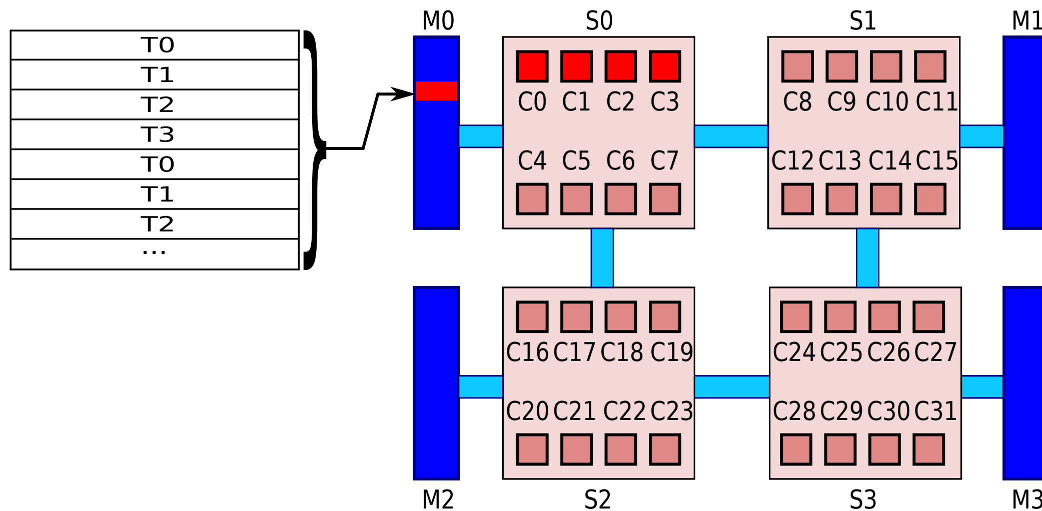


Odd-numbered iterations:

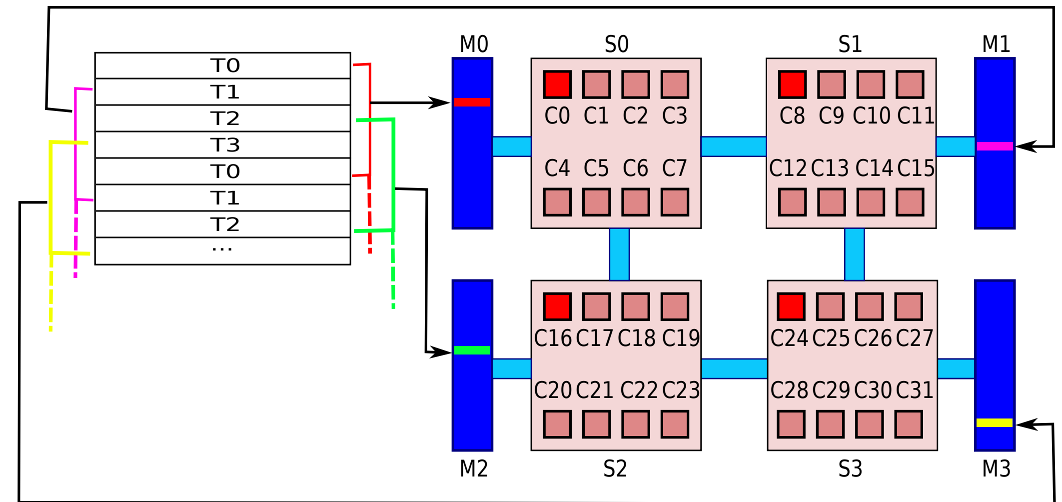


- The « *First Touch* » strategy is used.
- We will compare a *compact* binding with a *scatter* binding.

☞ A *compact* binding



☞ A *scatter* binding




```
> export OMP_NUM_THREADS=4 ; a.out
```

```
Temps elapsed = 33.46 s.
```

```
> export OMP_NUM_THREADS=4 ; a.out
```

```
Temps elapsed = 171.52 s.
```

☞ In this example, the *compact* mode is more than 5 times more efficient than the *scatter* mode!

8 – Performance

- In general, performance depends on the machine architecture (processors, interconnects and memory) and on the `OpenMP` implementation used.
- Nevertheless, there are some rules of « good performance » which are independent of the architecture.
- In the `OpenMP` optimisation phase, the objective is to reduce the elapsed time of the code and to estimate its speedup compared to a sequential execution.

8.1 – Good performance rules

- ☞ Verify that the thread binding mechanism on the execution cores is operating.
- ☞ Minimise the number of parallel regions in the code.
- ☞ Adjust the number of threads requested to the size of the specific problem to minimise the cost of thread management by the system.
- ☞ Parallelize the outermost loop whenever possible.
- ☞ During the development of your application, use the `SCHEDULE(RUNTIME)` clause so that you can dynamically change the scheduling and size of the iteration chunks in a loop.
- ☞ The `SINGLE` directive and the `NOWAIT` clause can decrease the elapsed time but most often this requires an explicit synchronisation.
- ☞ The `ATOMIC` directive and the `REDUCTION` clause are more restrictive in their usage but are more efficient than the `CRITICAL` directive.

- Use the IF clause to implement a conditional parallelization. (Usage example: On a vector architecture, do not parallelize a loop unless there is a sufficient number of iterations.)
- Avoid parallelizing the loop which iterates over the first dimension of arrays (in Fortran) because this is the one which makes reference to the contiguous elements in memory.

```
program parallel
 implicit none
 integer, parameter    :: n=1025
 real, dimension(n,n) :: a, b
 integer               :: i, j

 call random_number(a)

!$OMP PARALLEL DO SCHEDULE(RUNTIME)&
      !$OMP IF(n.gt.514)
   do j = 2, n-1
     do i = 1, n
        b(i,j) = a(i,j+1) - a(i,j-1)
     end do
   end do
!$OMP END PARALLEL DO
end program parallel
```

- Inter-task conflicts can noticeably degrade performance (memory bank conflicts on a vector machine or cache errors on a scalar machine).
- On `NUMA` machines, the affinity memory should be optimised by using the « *First Touch* » strategy.
- Independent of the machine architecture, the quality of the `OpenMP` implementation can significantly affect the scalability of the parallel loops.

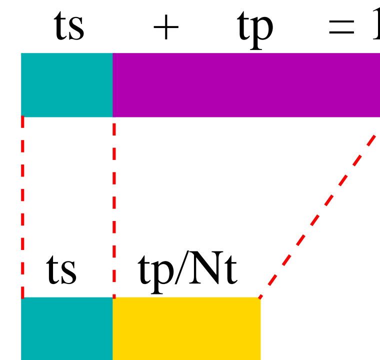
8.2 – Time measurements

- OpenMP offers two functions:
  - OMP_GET_WTIME, to measure the elapsed time in seconds.
  - OMP_GET_WTICK, to know the number of seconds between clock ticks.
- What we measure is the elapsed time from an arbitrary reference point of the code.
- This measurement can vary from one execution to another depending on the machine workload and the task distribution on the cores.

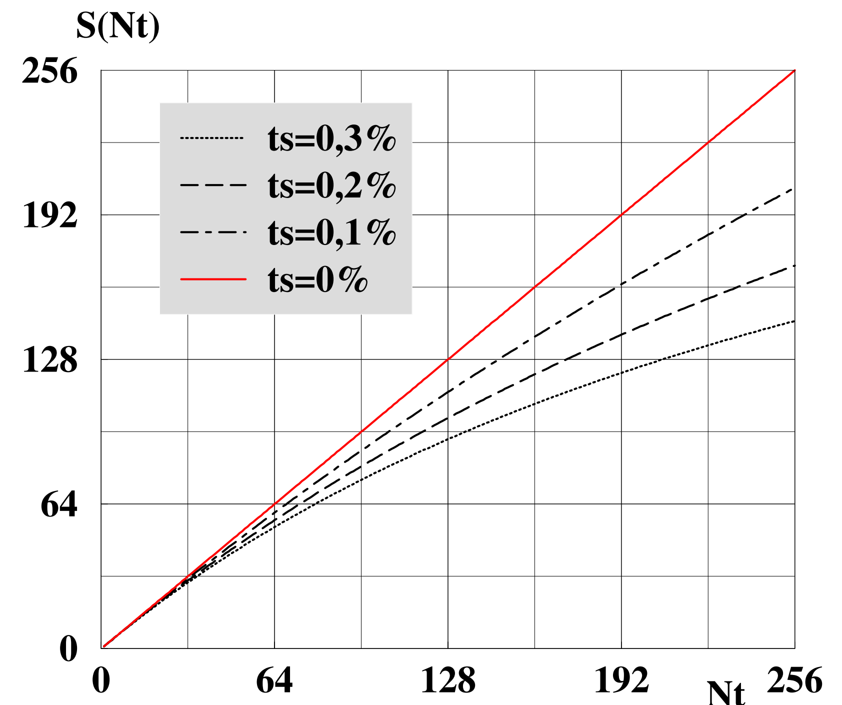
```
program mat_vect
 !$ use OMP_LIB
 implicit none
 integer,parameter    :: n=1025
 real,dimension(n,n) :: a
 real,dimension(n)    :: x, y
 real(kind=8) :: t_ref, t_final
 integer :: rang
 call random_number(a)
 call random_number(x) ; y(:)=0.
 !$OMP PARALLEL &
  !$OMP PRIVATE(rang,t_ref,t_final)


  rang = OMP_GET_THREAD_NUM()
  t_ref=OMP_GET_WTIME()
  call prod_mat_vect(a,x,y,n)
  t_final=OMP_GET_WTIME()
  print *,"Rang :",rang, &
          "; Temps :",t_final-t_ref
 !$OMP END PARALLEL
end program mat_vect
```

8.3 – Speedup

- The performance gain of a parallel code is estimated by comparing it to a sequential execution.
- The ratio between the sequential time T_s and the parallel time T_p on a dedicated machine is already a good indicator of the performance gain. It defines the speedup $S(N_t)$ of the code which depends on the number of threads N_t.
- If we consider $T_s = t_s + t_p = 1$ (t_s represents the time of the inherently sequential part and t_p is the time of the parallelizable part of the code), Amdhal's Law, $S(N_t) = \frac{1}{t_s + \frac{t_p}{N_t}}$, indicates that $S(N_t)$ is limited by the sequential fraction $\frac{1}{t_s}$ of the program.



$$S(Nt) = \frac{1}{ts + \frac{tp}{Nt}}$$



9 – Conclusion

- Requires a multi-processor computer with shared memory.
- Relatively easy to implement, even when starting from a sequential program.
- Allows progressive parallelization of a sequential program.
- The full potential of parallel performance is found in the parallel regions.
- Work can be shared within the parallel regions thanks to the loops, parallel sections and tasks. However, a thread can also be singled out for a particular work.
- Orphan directives allow the development of parallel routines.
- Explicit global or point-to-point synchronisations are sometimes necessary in the parallel regions.
- Careful attention must be given to defining the DSA of the variables used in a construct.
- Speedup measures code scalability. It is limited by the sequential fraction of the program and decreased by the costs linked to thread management.

10 – Annexes

10.1 – Subjects not addressed here

What was not (or only briefly) covered in this course:

- ☞ The « lock » procedures for point-to-point synchronisation
- ☞ Other service subroutines
- ☞ Hybrid MPI + OpenMP parallelization
- ☞ The new features of OpenMP 4.0 related to the use of accelerators

10.2 – Some traps

☞ In the first example shown here, the DSA of « s » variable is incorrect and this produces an inderminate result. In fact, the DSA of « s » must be SHARED in the lexical extent of the parallel region if the LASTPRIVATE clause is specified in the DO directive (there are also other clauses in this situation). Here, the IBM and NEC implementations give two different results. Although neither one is in contradiction with the standard, only one of the results is correct.

```
program faux_1
 ...
 real                   :: s
 real, dimension(9) :: a
 a(:) = 92290.
 !$OMP PARALLEL DEFAULT(PRIVATE) &
       !$OMP SHARED(a)
   !$OMP DO LASTPRIVATE(s)
     do i = 1, n
       s = a(i)
     end do
   !$OMP END DO
   print *, "s=",s,"; a(9)=",a(n)
 !$OMP END PARALLEL
end program faux_1
```

```
IBM SP> export OMP_NUM_THREADS=3;a.out
s=92290. ; a( 9 )=92290.
s=0.     ; a( 9 )=92290.
s=0.     ; a( 9 )=92290.
```

```
NEC SX-5> export OMP_NUM_THREADS=3;a.out
s=92290. ; a( 9 )=92290.
s=92290. ; a( 9 )=92290.
s=92290. ; a( 9 )=92290.
```

☞ In the second example shown here, a race condition occurs between the threads. As a result, the « `print` » instruction does not display the expected result of the « s » variable whose DSA is `SHARED`. Here, we find that `NEC` and `IBM` give identical results but it is possible to obtain a different but legitimate result on another platform. One solution is to add a `BARRIER` directive just after the « `print` » instruction.

```
program faux_2
 implicit none
 real    :: s
 !$OMP PARALLEL DEFAULT(NONE) &
       !$OMP SHARED(s)
  !$OMP SINGLE
   s=1.
  !$OMP END SINGLE
  print *,"s = ",s
  s=2.
 !$OMP END PARALLEL
end program faux_2
```

```
IBM SP> export OMP_NUM_THREADS=3;a.out
s =   1.0
s =   2.0
s =   2.0
```

```
NEC SX-5> export OMP_NUM_THREADS=3;a.out
s =   1.0
s =   2.0
s =   2.0
```

☞ In the third example shown here, a deadlock can occur due to thread desynchronisation. (A thread arriving late can exit the loop while the other threads, having arrived before, continue to wait at the implicit synchronisation barrier of the SINGLE construct.) The solution is to add a barrier either before the SINGLE construct or after the « `if` » test.

```
program faux_3
  implicit none
  integer :: iteration=0

  !$OMP PARALLEL
  do
     !$OMP SINGLE
     iteration = iteration + 1
     !$OMP END SINGLE
     if( iteration >= 3 ) exit
  end do
  !$OMP END PARALLEL
  print *,"Outside // region"
end program faux_3
```

```
 Intel> export OMP_NUM_THREADS=3;a.out
... rien ne s'affiche à l'écran ...
```