OpenMP

Multithreaded Parallelization for Shared-Memory Machines

Jalel Chergui
Pierre-François Lavallée*
*<Prénom.Nom@idris.fr>

OpenMP : plan

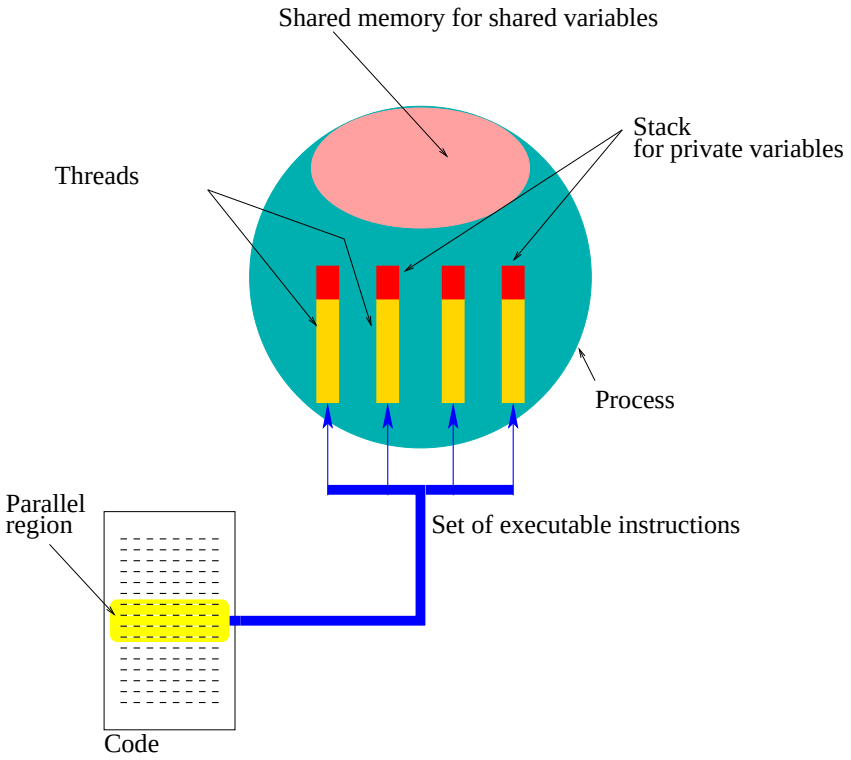
1 – Introduction

1.1 – History

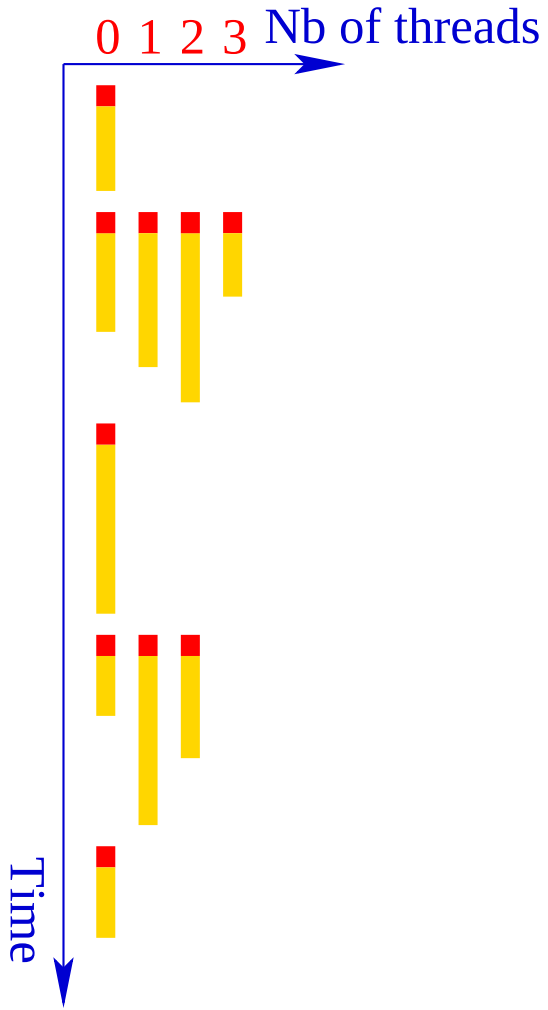
- Directive-based multithreaded parallelization existed for a long time at some manufacturers (ex. CRAY, NEC, IBM, ...), but each one had its own set of directives.
- The resurgence of shared-memory multiprocessor machines made it compelling to define a standard.
- The standardization attempt of PCF (Parallel Computing Forum) was never adopted by the official authorities of standardization.
- On the 28th of October 1997, a vast majority of industry researchers and manufacturers adopted `OpenMP` (Open Multi Processing) as an " industrial " standard.
- The specifications of `OpenMP` now rest with the ARB (Architecture Review Board), the only organization in charge of its development.
- An `OpenMP-2` version was finalized in November 2000. In particular, it brought extensions related to the parallelization of certain `Fortran` 95 constructions.
- The latest version, `OpenMP-3` dating from May 2008, essentially introduces the concept of tasks (this will be dealt with in a forthcoming version of this course).

1.2 – General concepts

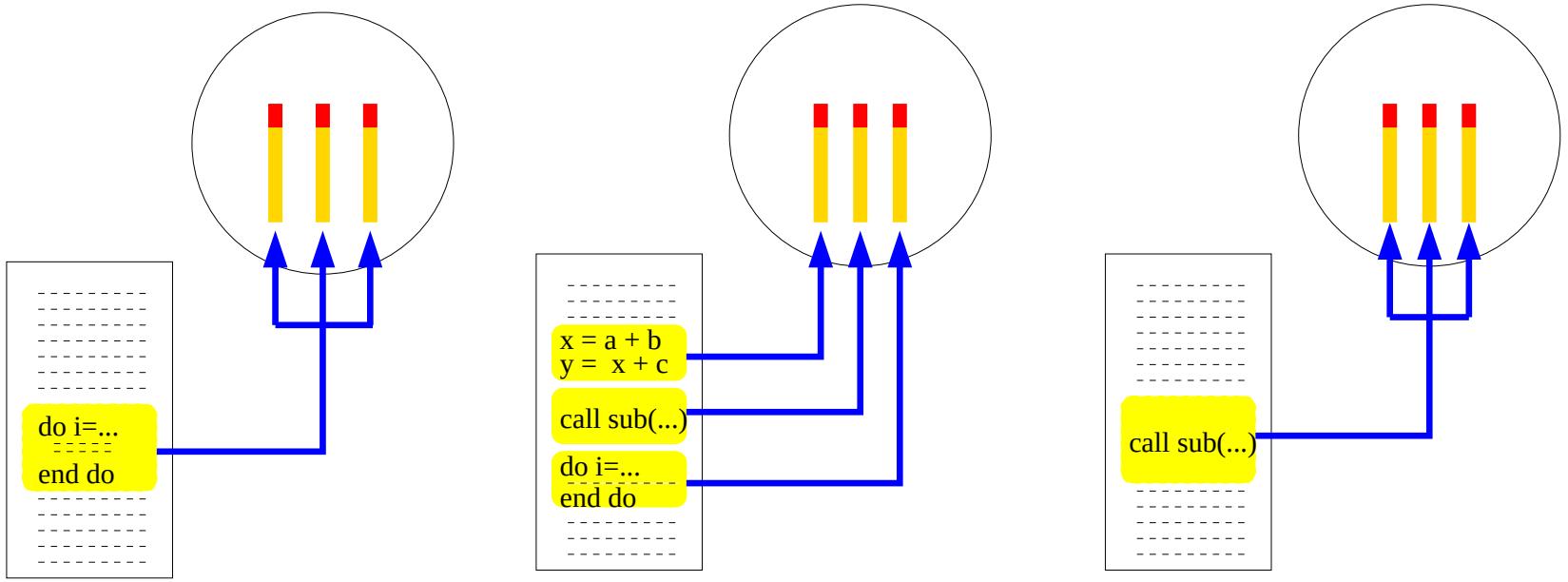
- An OpenMP program is executed by one process.
- This process activates light-weight processes (threads) at the entry of a parallel region.
- Each thread executes a task comprised of a group of instructions.
- During the execution of a task by a thread, a variable can be read and/or updated in memory.
  - It can be defined either in the stack (local memory space) of a thread : we refer to this as a private variable,
  - or in a shared-memory space accessible by all the threads : we refer to this as a shared variable.



- An `OpenMP` program is an alternation of sequential regions and parallel regions.
- A sequential region is always executed by the MASTER thread, the one whose rank equals 0.
- A parallel region can be executed by many threads at once.
- Threads can share the work contained in the parallel region.



- The work-sharing consists mainly of :
  - executing a loop by dividing up iterations between the threads ;
  - executing many code sections but only one per thread ;
  - executing many occurrences of the same procedure by different threads (orphaning).

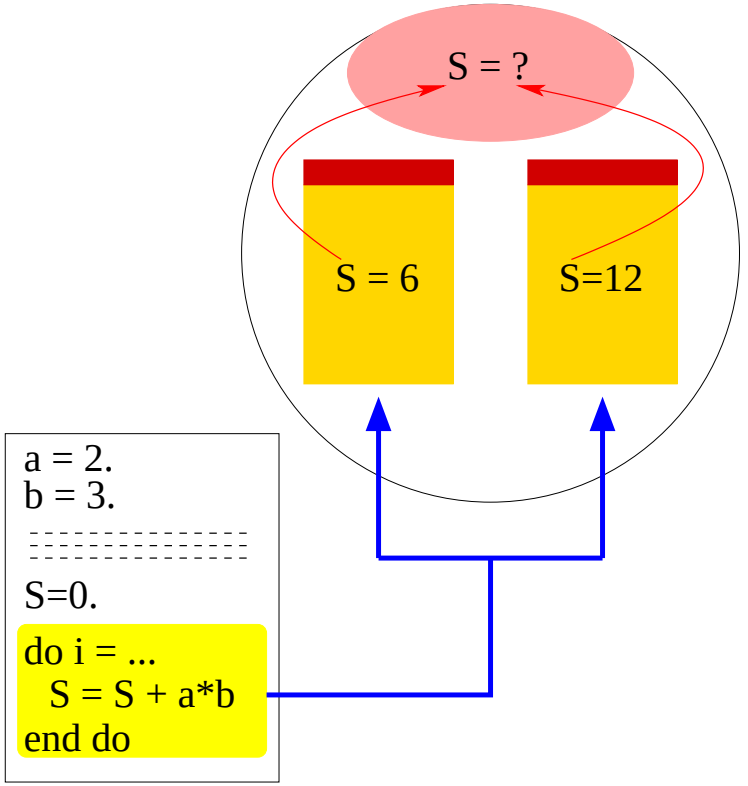


Parallel loop
(Looplevel parallelism)

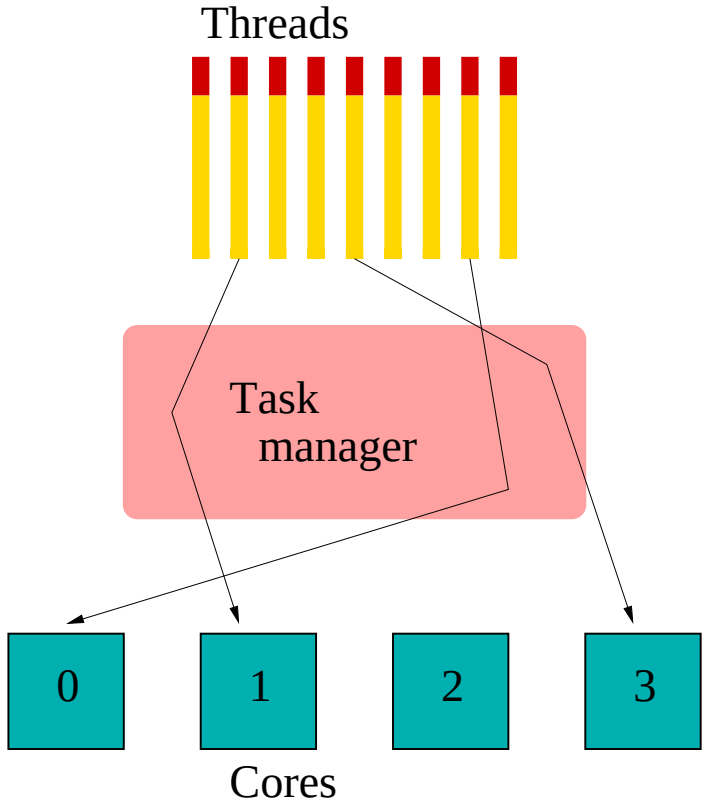
Parallel section

Orphaning

☞ It is sometimes compulsory to introduce a synchronization between the concurrent threads, for example to avoid that they update in an unspecified order the value of the same shared variable (case of reduction operations).

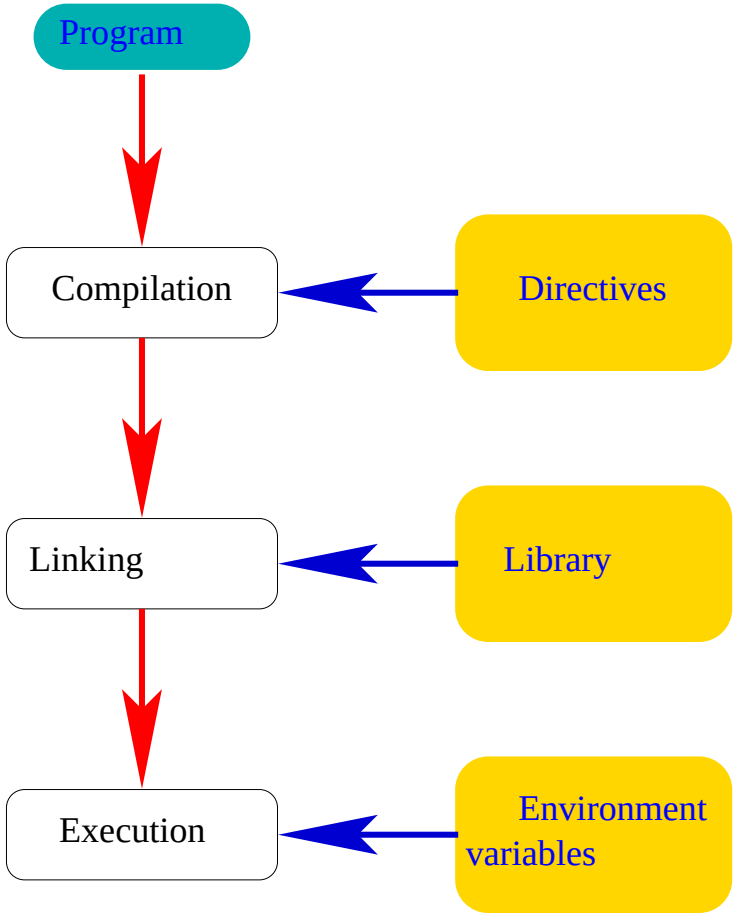


- The threads are mapped onto the execution cores by the operating system. Different cases can occur :
  - At best, at each moment, there is a thread per execution core with as many threads as dedicated execution cores during all the work time ;
  - At worst, all the threads are processed sequentially by only one execution core ;
  - In reality, for operational reasons on a machine whose execution cores are not dedicated, the situation is generally intermediate.



1.3 – OpenMP structure

1. Compilation Directives and Clauses :
   - ☞ They serve to create the threads, define the work-sharing, the synchronization and the data-sharing attributes of variables (shared or private) ;
   - ☞ They are considered by the compiler as comment lines unless a suitable compiler option is specified so that they may be interpreted.
2. Functions and Routines : They are part of a loaded library at link.
3. Environment Variables : Once set, their values are taken into account at execution.



Here are the compilation options which trigger the interpretation of OpenMP directives by certain Fortran compilers :

- On an IBM machine : -qsmp=omp

```
xlf_r -qsuffix=f=f90 -qnosave -qsmp=omp prog.f90  # Compilation and loading
export OMP_NUM_THREADS=4                          # Number of threads
a.out                                             # Execution
```

- On a NEC machine : -Popenmp

```
f90 -Popenmp prog.f90                             # Compilation and loading
export OMP_NUM_THREADS=4                          # Number of threads
a.out                                             # Execution
```

- With an Intel compiler : -openmp

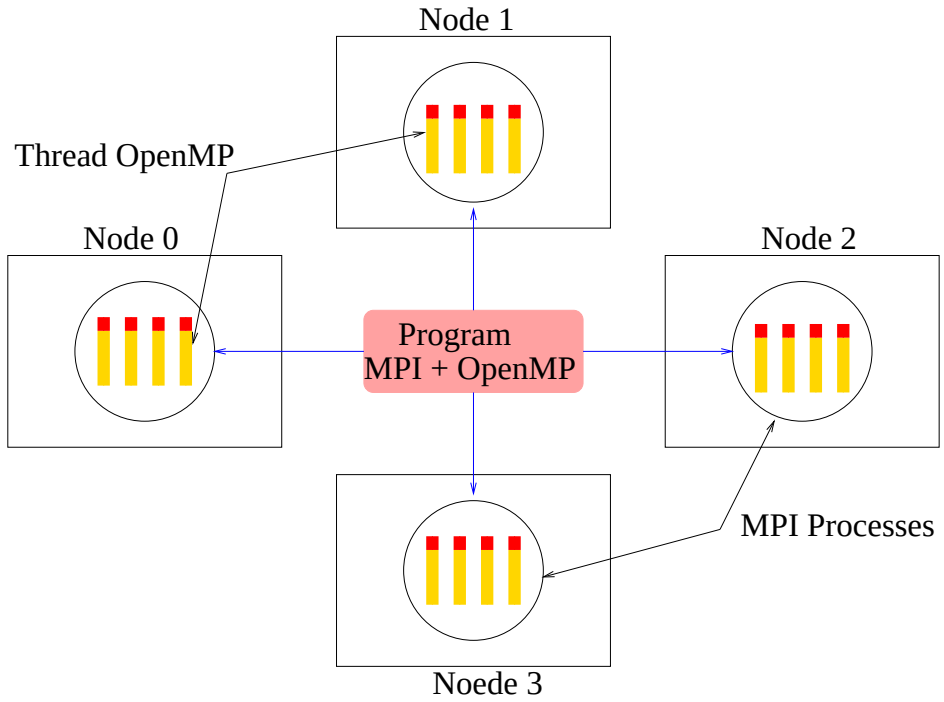
```
ifort -openmp prog.f90                            # Compilation and loading
export OMP_NUM_THREADS=4                          # Number of threads
a.out                                             # Execution
```

1.4 – OpenMP versus MPI

These are two complementary models of parallelization.

- `OpenMP`, as `MPI`, has a `Fortran`, `C` and `C++` interface.
- `MPI` is a multiprocess model for which the communication mode between the processes is explicit (the management of communications is the responsibility of the user).
- `OpenMP` is a multithreaded model for which the communication mode between the threads is implicit (the management of communications is the responsibility of the compiler).

- `MPI` is generally used on distributed-memory multiprocessor machines.
- `OpenMP` is used on shared-memory multiprocessor machines.
- On a cluster of independent shared-memory multiprocessor machines (SMP nodes), the implementation of parallelization with both `MPI` and `OpenMP` in the same program can be a major advantage for the parallel performances of the code.

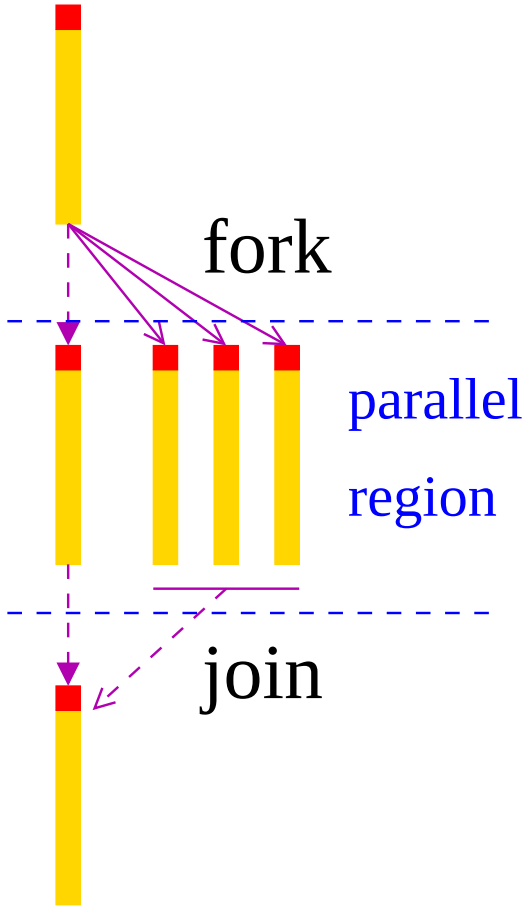


1.5 – Bibliography

- First book on OpenMP : R. Chandra & *al.*, *Parallel Programming in OpenMP*, éd. Morgan Kaufmann Publishers, oct. 2000.
- More recent book on OpenMP : B. Chapman & *al.*, *Using OpenMP*, MIT Press, 2008.
- Validated 3.0 specifications of the OpenMP standard : http://www.openmp.org/
- Website dedicated to the OpenMP users : http://www.compunity.org/

2 – Principles

- It is the responsibility of the developer to introduce `OpenMP` directives in its code (at least in the absence of automatic parallelization tools).
- At the program execution, the operating system creates a parallel region on the "fork-join" model.
- At the entry of a parallel region, the Master thread creates/activates (fork) a team of "child" threads which disappear/hibernate at the end of the parallel region (join) while the Master thread alone continues the execution of the program until the entry of the next parallel region.



2.1 – General syntax of a directive

☞ An OpenMP directive has the following general form :

```
sentinel directive-name [clause[ clause]...]
```

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

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

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

☞ For Fortran, in free format :

```
!$ use OMP_LIB
...
!$OMP PARALLEL PRIVATE(a,b) &
      !$OMP FIRSTPRIVATE(c,d,e)
      ...
!$OMP END PARALLEL ! It's 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++ :

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

2.2 – Parallel region building

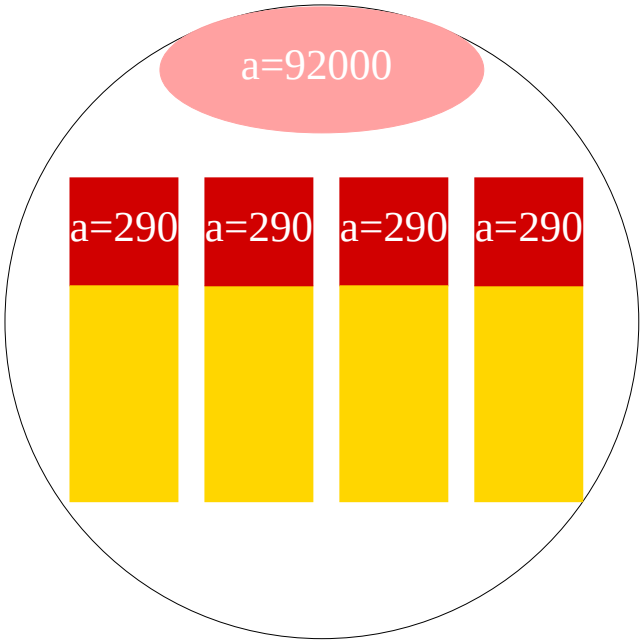
- In a parallel region, by default, the data-sharing attribute of the variables is shared.
- Within a single parallel region, all the concurrent threads execute the same code.
- There is an implicit synchronization barrier at the end of the parallel region.
- A program that branches (e.g. *GOTO*, *CYCLE*) into or out of a parallel region is non-conforming.

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

 a = 92290. ; p=.false.
 !$OMP PARALLEL
  !$ p = OMP_IN_PARALLEL()
  print *,"A = ",a, &
          "; p = ",p
 !$OMP END PARALLEL
end program parallel
```

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4
> a.out
```

```
A = 92290. ; p = T
A = 92290. ; p = T
A = 92290. ; p = T
A = 92290. ; p = T
```

- It is possible, with the DEFAULT clause, to change the default data-sharing attribute of the variables in a parallel region.
- If a variable has a private data-sharing attribute (PRIVATE), it will be stored in the stack of each thread. Its value in this case is indeterminate at the entry of a parallel region.

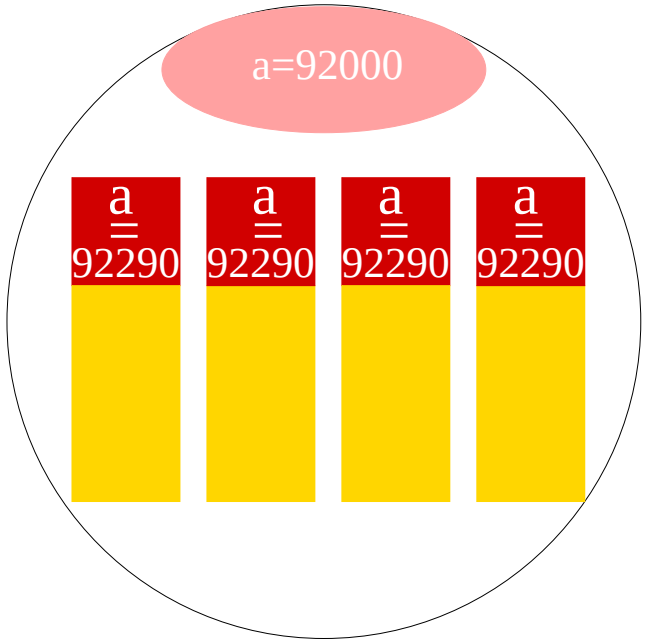



```
program parallel
 implicit none
 real :: a

 a = 92000.
 !$OMP PARALLEL DEFAULT(PRIVATE)
   a = a + 290.
   print *,"A = ",a
 !$OMP END PARALLEL
end program parallel
```

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4
> a.out
```

```
 A =  290.
 A =  290.
 A =  290.
 A =  290.
```

☞ However, with the FIRSTPRIVATE clause, it is possible to force the initialization of this private variable with the last value it had before the entry in the parallel region.




```
program parallel
 implicit none
 real :: a

 a = 92000.
!$OMP PARALLEL DEFAULT(NONE) &
        !$OMP FIRSTPRIVATE(a)
   a = a + 290.
   print *,"A = ",a
!$OMP END PARALLEL
print*,"Out of region, A =",a
end program parallel
```

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4
> a.out
```

```
 A =  92290.
 A =  92290.
 A =  92290.
 A =  92290.
 Hors region, A = 92000.
```

2.3 – Parallel region extent

- The extent of an OpenMP construct is the range of its influence in the program.
- The influence (or the scope) of a parallel region extends to the code lexically contained in this region (static extent), as well as to the code of the called routines. The union of the 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
```

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4;a.out
```

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

☞ In a called routine inside a parallel region, the local and automatic variables are implicitly private to each thread (they are stored in the stack of each thread).

```
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 = ",a
end subroutine sub
```

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4;a.out
```

```
 A =  92290
 A =  92291
 A =  92292
 A =  92293
```

2.4 – Argument-passing case

☞ In a called routine, all the dummy arguments passed by reference inherit the data-sharing attribute of the actual associated argument.

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

 a = 92000
 !$OMP PARALLEL SHARED(a) PRIVATE(b)
    call sub(a, b)
    print *,"B = ",b
 !$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
```

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4
> a.out
```

```
 B = 92002
 B = 92003
 B = 92001
 B = 92000
```

2.5 – Static variables case

- A variable is static if its location in memory is defined at declaration by the compiler.
- This is the case of variables appearing in COMMON or contained in a MODULE or declared with SAVE or initialized when declared (ex. PARAMETER, DATA, etc.).
- By default, a static variable is a shared variable.

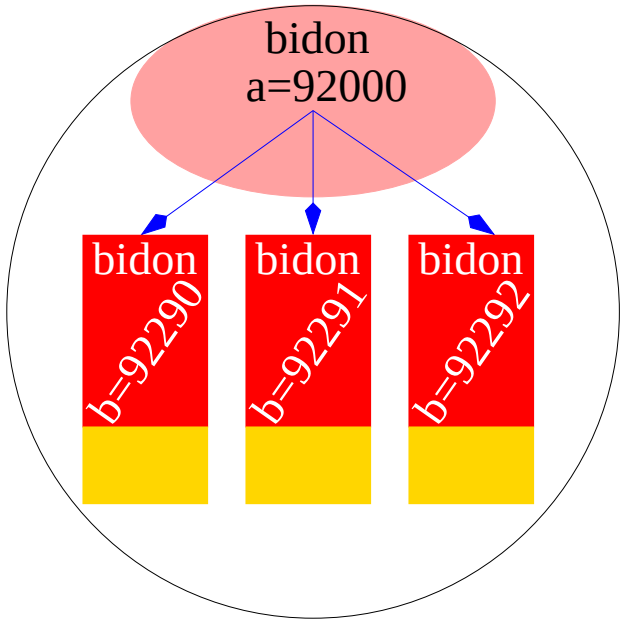
```
module var_stat
  real :: c
end module var_stat
```

```
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 :: rank
  common /bidon/a
  rank = OMP_GET_THREAD_NUM()
  a=rank;b=rank;c=rank
  !$OMP BARRIER
  print *,"values of A, B and C : ",a,b,c
end subroutine sub
```

```
> xlf_r ... -qsmp=omp var_stat.f90 prog.f90
> export OMP_NUM_THREADS=2;a.out
```

```
 values of A, B and C : 0.0 1.0 1.0
 values of A, B and C : 0.0 1.0 1.0
```

- The use of the THREADPRIVATE directive can privatize a static instance (variables are replicated, each thread having its own copy) and make it persistant from one parallel region to another.
- If, in addition, the COPYIN clause is specified then the initial value of the static instance is sent 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()
 !$OMP END PARALLEL
 print*,"Out of region, A =",a
end program parallel
subroutine sub()
 implicit none
 integer :: a, b
 common/bidon/a
 !$OMP THREADPRIVATE(/bidon/)
 b = a + 290
 print *,"B = ",b
end subroutine sub
```

```
B = 92290
B = 92291
B = 92292
B = 92293
Out of region, A = 92000
```

2.6 – The dynamic memory allocation case

- The dynamic memory allocation/deallocation operations can be made within a parallel region.
- If the operation involves a private variable, this variable will be local for each thread.
- If the operation involves a shared variable, then it is wiser for only one thread (ex. the master thread) to take the responsibility for this operation.

```
program parallel
  !$ use OMP_LIB
  implicit none
  integer                              :: n,i_start,i_end,rank,nb_threads,i
  real, allocatable, dimension(:) :: a

  n=1024 ; nb_threads=4
  allocate(a(n*nb_threads))
  !$OMP PARALLEL DEFAULT(NONE) PRIVATE(i_start,i_end,rank,i) &
        !$OMP SHARED(a,n) IF(n .gt. 512)
     rank=OMP_GET_THREAD_NUM()
     i_start=rank*n+1
     i_end=(rank+1)*n
     do i = i_start, i_end
        a(i) = 92290. + real(i)
     end do
     print *,"Rank : ",rank,"; A(",i_start,"),...,A(",i_end,") : ",a(i_start),",...,",a(i_
  !$OMP END PARALLEL
  deallocate(a)
end program parallel
```

```
Rank : 3 ; A( 3073 ), ... , A( 4096 ) : 95363., ... , 96386.
Rank : 2 ; A( 2049 ), ... , A( 3072 ) : 94339., ... , 95362.
Rank : 1 ; A( 1025 ), ... , A( 2048 ) : 93315., ... , 94338.
Rank : 0 ; A(    1 ), ... , A( 1024 ) : 92291., ... , 93314.
```

2.7 – The equivalence case

- Only variables having the same data-sharing attribute can be put into equivalence.
- If they do not, the result is undefined.
- These remarks remain valid in the case of an association between variables by POINTER.

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

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

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4;a.out
```

```
B = -0.3811332074E+30
B = 0.0000000000E+00
B = -0.3811332074E+30
B = 0.0000000000E+00
```

2.8 – Additions

- The construct of a parallel region allows two other clauses :
  - `REDUCTION` : For the reduction operations with implicit synchronization between the threads ;
  - `NUM_THREADS` : It can be used to specify the required number of threads at the entry of a parallel region in the same way as a call to `OMP_SET_NUM_THREADS` routine would do.
- From one parallel region to another, the number of concurrent threads can be changed if you wish. In order to do this, just use the `OMP_SET_DYNAMIC` routine or set the `OMP_DYNAMIC` environment variable to `true`.

```
program parallel
  implicit none

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

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

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4;
> export OMP_DYNAMIC=true; a.out
```

```
Hello !
Hello !
Hi !
Hi !
Hi !
```

☞ Nested parallel regions are possible, but this has no effect unless this mode has been activated by a call to the OMP_SET_NESTED routine or by setting the OMP_NESTED environment variable to true.

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

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

```
> xlf_r ... -qsmp=nested_par prog.f90
> export OMP_DYNAMIC=true
> export OMP_NESTED=true; a.out
```

```
My rank in region 1 : 0
  My rank in region 2 : 1
  My rank in region 2 : 0
My rank in region 1 : 2
  My rank in region 2 : 1
  My rank in region 2 : 0
My rank in region 1 : 1
  My rank in region 2 : 0
  My rank in region 2 : 1
```

Position in the Course

3 – Worksharing

- In principle, the parallel region construct and the use of some OpenMP routines are enough alone to parallelize a code portion.
- But it is, in this case, the responsibility of the programmer to distribute both the work and the data and to manage the synchronization of threads.
- Fortunately, OpenMP proposes three directives (DO, SECTIONS and WORKSHARE) which make it easily possible to finely control the work and data distribution simultaneously with the synchronization within a parallel region.
- In addition, there are other OpenMP constructs which enable the exclusion of all the threads except for one thread, in order to execute a code portion located in a parallel region.

3.1 – Parallel loop

- A parallel loop is a loop for which all iterations are independent of each other.
- It's a parallelism by distribution of loop iterations.
- The parallelized loop is the one which comes immediately after the DO directive.
- The "infinite" loops and `do while` are not parallelizable with `OpenMP`.
- The distribution mode of the iterations can be specified with the SCHEDULE clause.
- The choice of the distribution mode allows a better control of the load-balancing between the threads.
- The loop indices are private integer variables.
- By default, a global synchronization is done at the end of a END DO construct unless the NOWAIT clause is specified.
- It is possible to introduce as many DO constructs as we want (one after another) in a parallel region.

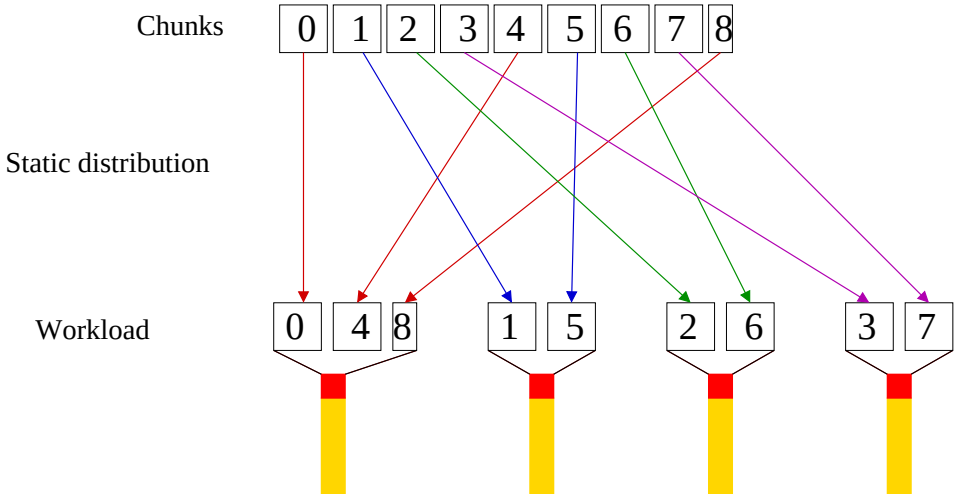
3.1.1 – SCHEDULE clause

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

```
> xlf_r ... -qsmp=omp prog.f90 ; export OMP_NUM_THREADS=4 ; a.out
```

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

☞ The `STATIC` distribution consists of dividing the iterations into chunks according to a given size (except perhaps for the last one). The chunks are then assigned to each of the threads in a cyclical manner (round-robin) following the order of the threads to be done.



- We could have postponed the scheduling of iteration distribution at runtime by using the OMP_SCHEDULE environment variable.
- The scheduling of loop iteration distribution can be a major contribution for the load-balancing on a machine whose processors are not dedicated.
- It is important to avoid using the first dimension of a multi-dimensional array as a reference when parallelizing the loops in Fortran ; doing this would heavily lower the performance.

```
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
   !$OMP END DO
   print*,"Rank:",OMP_GET_THREAD_NUM(), &
            ";i_min:",i_min,";i_max:",i_max
 !$OMP END PARALLEL
end program parallel
```

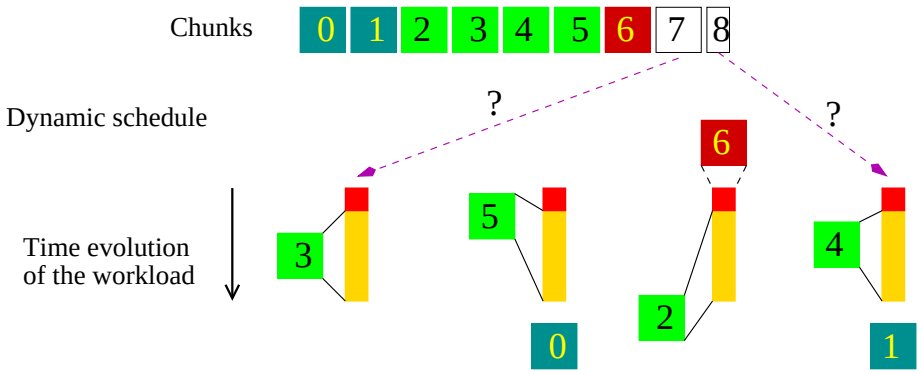
```
> export OMP_NUM_THREADS=2
> export OMP_SCHEDULE="STATIC,1024"
> a.out
```

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

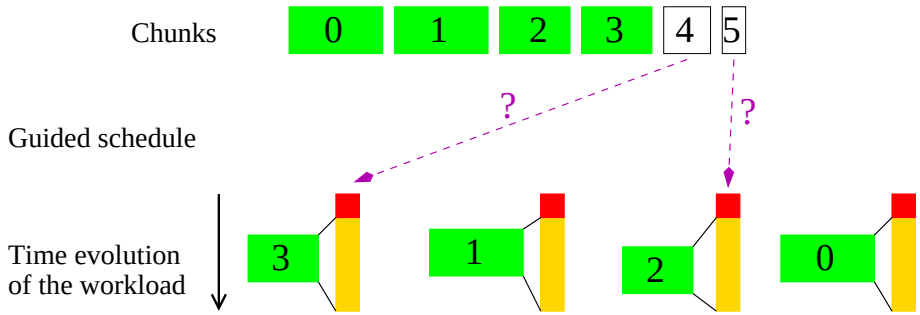
```
> export OMP_SCHEDULE="DYNAMIC,480"
> export OMP_NUM_THREADS=4 ; a.out
```

☞ In addition to the STATIC schedule, there are two other ways to distribute the loop iterations :

- ➳ DYNAMIC : The iterations are divided into chunks according to a given size. Once a thread completes its chunk of iterations, another chunk is assigned to it.
- ➳ GUIDED : The iterations are divided into chunks which decrease in size exponentially. All the chunks have a size superior or equal to a given value except for the last one whose size can be inferior. Once a thread finishes its chunk, another chunk of iterations is assigned to it.




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



3.1.2 – An ordered execution case

- It is sometimes useful (when debugging) to execute a loop in an orderly way.
- The order of iterations will then be identical to the one occuring during a sequential execution.

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

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

3.1.3 – A reduction case

- 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 computes a partial result independently of the others. They are then synchronized in order to update 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
```

3.1.4 – Additions

- The other clauses of the DO directive are :
  - PRIVATE : to assign a private data-sharing attribute to a variable ;
  - FIRSTPRIVATE : privatizes a shared variable in the extent of the DO construct and assigns to it the last affected value before the entry of the loop ;
  - LASTPRIVATE : privatizes a shared variable in the extent of the DO construct and allows to return, at the exit of this construct, the value computed by the thread executing the last loop iteration.

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

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

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

```
Rank : 2 ; res= 9.0
Rank : 3 ; res= 9.0
Rank : 1 ; res= 9.0
Rank : 0 ; res= 9.0
```

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

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

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

3.2 – Parallel sections

- The `SECTIONS` construct is a worksharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads. Each structured block is executed once by one of the threads in the team.
- Several structured blocks 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 many independent code portions (structured blocks) on different threads.
- The `NOWAIT` clause is allowed at the end of the `END SECTIONS` construct to remove the implicit synchronization barrier.

3.2.1 – SECTIONS construct

```
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                  :: step_x, step_y
 integer               :: i

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

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

 call random_number(x)
end subroutine read_initial_x


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

 call random_number(y)
end subroutine read_initial_y
```

3.2.2 – Additions

- All the `SECTION` directives have to appear in the lexical extent of the `SECTIONS` construct.
- The clauses allowed for the `SECTIONS` construct are the ones which we already know :
  - `PRIVATE` ;
  - `FIRSTPRIVATE` ;
  - `LASTPRIVATE` ;
  - `REDUCTION`.
- The `PARALLEL SECTIONS` directive is a combination of `PARALLEL` and `SECTIONS` directives with the union of their respective clauses.
- The termination directive `END PARALLEL SECTIONS` includes a global synchronization barrier and cannot admit the `NOWAIT` clause.

3.3 – WORKSHARE construct

- The `WORKSHARE` construct divides the execution of the enclosed structured block into separate units of work, each one being executed only once by one thread.
- It can only be specified within a parallel region.
- It is useful to distribute the work of `Fortran` 95 constructs such as :
  - array and scalar assignments ;
  - `FORALL` and `WHERE` constructs ;
  - transformational array intrinsic functions : `MATMUL`, `DOT_PRODUCT`, `SUM`, `PRODUCT`, `MAXVAL`, `MINVAL`, `COUNT`, `ANY`, `ALL`, `SPREAD`, `PACK`, `UNPACK`, `RESHAPE`, `TRANSPOSE`, `EOSHIFT`, `CSHIFT`, `MINLOC` and `MAXLOC` ;
  - user-defined functions called "`ELEMENTAL`".
- It admits the `NOWAIT` clause only, at the end of the construct (`END WORKSHARE`).

- Only the instructions or Fortran 95 blocks specified in the lexical extent will have their work distributed among the threads.
- The work unit is the element of an array. There is no other way to change this default behavior.
- The additional costs caused by such a work distribution can 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 PARALLEL and WORKSHARE constructs with the union of 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 – Exclusive execution

- Sometimes, in a parallel region, we want to execute some code portions excluding all the threads except one.
- In order to do this, `OpenMP` offers two directives : `SINGLE` and `MASTER`.
- Although the purpose is the same, the behavior induced by these two constructs remains basically different.

3.4.1 – SINGLE construct

- The SINGLE construct makes it possible to execute a code portion by one and only one thread without being able to specify which one.
- In general, it is the thread which comes first on the SINGLE construct, but this is not specified in the standard.
- All the threads that do not execute the SINGLE region wait, at the end of the construct (END SINGLE), for the termination of the one which is in charge of the work, unless a NOWAIT clause is specified.

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

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

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

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

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4 ; a.out
```

```
Rank : 1 ; A = 92290.
Rank : 2 ; A = 92290.
Rank : 0 ; A = 92290.
Rank : 3 ; = -92290.
```

- An additional clause allowed only by the END SINGLE termination directive is the COPYPRIVATE clause.
- It allows the thread responsible for the SINGLE region to distribute to other threads the value of a list of private variables before leaving this region.
- The other clauses of the SINGLE construct are PRIVATE and FIRSTPRIVATE.

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

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

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

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

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4 ; a.out
```

```
Rank : 1 ; A = -92290.
Rank : 2 ; A = -92290.
Rank : 0 ; A = -92290.
Rank : 3 ; A = -92290.
```

3.4.2 – MASTER construct

- The MASTER construct enables the execution of a code portion by the master thread alone.
- This construct does not allow any clause.
- There is no synchronization barrier neither at the start (MASTER) nor at the end of the construct (END MASTER).

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

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

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

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

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4 ; a.out
```

```
Rank : 0 ; A = -92290.
Rank : 3 ; A =  92290.
Rank : 2 ; A =  92290.
Rank : 1 ; A =  92290.
```

3.5 – Orphaned procedures

- A procedure (function or routine) called in a parallel region is executed by all the threads.
- In general, it is useless unless the work of the procedure is distributed.
- This requires the introduction of OpenMP (DO, SECTIONS, etc.) directives in the body of the procedure if it is called in a parallel region.
- These directives are called "orphans" and, by excess of language, they are referred to as orphaned procedures (*orphaning*).
- A multi-threaded scientific library parallelized with OpenMP will be constituted of a set of orphaned procedures.

```
> 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 according to the calling compilation mode and the 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 is 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 have been interpreted at compilation.

compiled called unit / compiled calling unit	with `-qsmp=omp`	without `-qsmp=omp`
with `-qsmp=omp`	P + D	P + R
without `-qsmp=omp`	S	S

TABLE 1 – Execution context according to the compilation mode (example with `IBM Fortran` compiler options).

3.6 – Summary

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

Position in the Course

4 – Synchronizations

Synchronization becomes necessary in the following situations :

❶ to make sure that all the concurrent threads have reached the same instruction point in the program (global barrier) ;

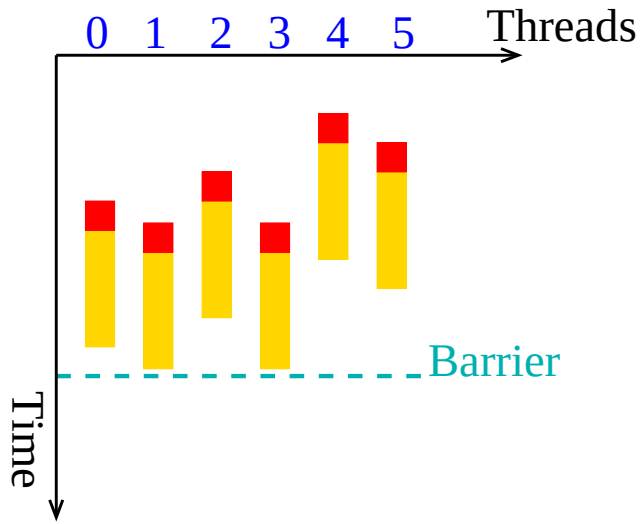
❷ to order the execution of all the concurrent threads when they have to execute the same code portion affecting one or many shared variables whose coherence in memory (read or write) has to be guaranteed (mutual exclusion).

❸ to synchronize at least two concurrent threads among the others (lock mechanism).

- As we have already indicated, the absence of a NOWAIT clause means that a global synchronization barrier is implicitly applied at the end of some OpenMP constructs. However, it is possible to explicitly impose a global synchronization barrier through the BARRIER directive.
- The mutual exclusion mechanism (one thread at a time) is found, for example, in the reduction operations (REDUCTION clause) or in the ordered loop execution (ORDERED clause). This mechanism is also implemented in the ATOMIC and CRITICAL directives.
- Finer synchronizations can also be done either by the implementation of lock mechanisms (this requires the call of OpenMP library routines), or by the use of the FLUSH directive.

4.1 – Barrier

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




```
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 = ", b(1:n)
end program parallel
```

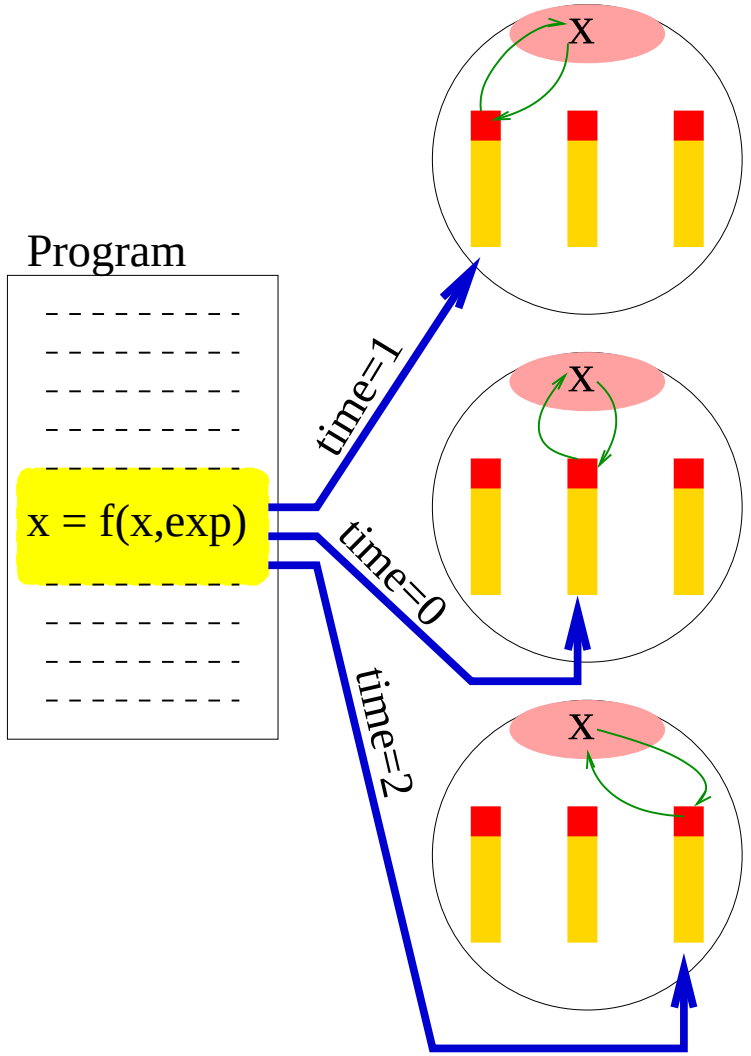
4.2 – Atomic updating

- The ATOMIC update ensures that a shared variable is read and modified in memory by one and only one thread at a time.
- It has a local effect on the instruction which immediately follows the directive.

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

  print *,"Rank :",rank, &
     "; counter =",counter
 !$OMP END PARALLEL
 print *,"In total, counter =", &
          counter
end program parallel
```

```
Rank : 1 ; counter = 92291
Rank : 0 ; counter = 92292
Rank : 2 ; counter = 92293
Rank : 3 ; counter = 92294
In total, counter = 92294
```

- The instruction 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.
- The threads execute this region in a non-deterministic order, but only one at a time.
- A critical structured block (region) is defined through the `CRITICAL` directive and applies to a code portion terminated by `END CRITICAL`.
- Its extent is dynamic.
- For performance reasons, it is not recommended to emulate an atomic instruction by a critical construct.
- An optional name can be used to name a critical construct.
- All the non-explicitly named critical constructs are considered as having the same non-specified name.
- If many critical constructs have the same name, they are considered for the mutual exclusion mechanism as the same and unique critical construct.

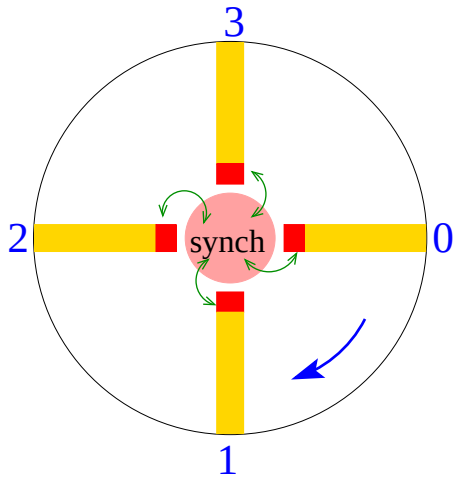
```
program parallel
  implicit none
  integer :: s, p

  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
```

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS=4 ; a.out
```

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

4.4 – The FLUSH Directive

- It is useful in a parallel region to refresh the value of a shared variable in global memory.
- It is even more useful if the memory hierarchy of a machine has multiple levels of caches.
- It can be used to establish a mechanism of synchronization between two 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
```

4.5 – Summary

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

Position in the Course

5 – Traps

☞ In the first example shown here, the data-sharing attribute of the variable "s" is incorrect, which produces an indeterminate result. In fact, the data-sharing attribute of "s" has to be SHARED in the lexical extent of the parallel region if the LASTPRIVATE clause is specified in the DO directive (it is not the only clause in that case). Here, the two implementations, IBM and NEC, provide two different results. However, neither one of them is in contradiction with the standard, whereas one of the results is correct.

```
program false_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 false_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, there is a data race between the threads. The "`print`" instruction does not print the expected result of the variable "s" whose data-sharing attribute is SHARED. It turns out here that NEC and IBM provide identical results but it is possible and legitimate to obtain a different result on another platform. One solution is to add, for example, a BARRIER directive just after the "`print`" instruction.

```
program false_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 faulse_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, there is a possibility of a deadlock occurring due to a desynchronization of the threads (a thread arriving much later than the other is able to exit the loop whereas the other threads continue to wait at the implicit synchronization barrier of the construct SINGLE). The solution consists of adding a global synchronization barrier, either just before the construct SINGLE, or just after the test to exit the loop.

```
program false_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 false_3
```

```
 Intel> export OMP_NUM_THREADS=3;a.out
... nothing appears on the screen ...
```

Position in the Course

6 – Performance

- In general, the performance depends on the architecture (processors, networks and memory) of the machine and on the `OpenMP` implementation.
- Nevertheless, there are some rules of "good performance" which are independent of the architecture.
- In the optimization phase with `OpenMP`, the goal is to reduce the elapsed time of the code and to estimate its speedup relative to a sequential execution.

6.1 – Good performance rules

- Minimize the number of parallel regions in the code.
- Adjust the number of threads to the size of the problem in order to minimize the additional costs of thread management by the system.
- Whenever possible, parallelize the outermost loop.
- Use the `SCHEDULE(RUNTIME)` clause in order to be able to dynamically change the scheduling and the size of the chunk of a parallel loop.
- The `SINGLE` directive and the `NOWAIT` clause can help to decrease the elapsed time at the expense, very often, of an explicit synchronization.
- The `ATOMIC` directive and the `REDUCTION` clause are more restrictive, but by far more efficient than the `CRITICAL` directive.

- Use the IF clause to do a conditionial parallelization (ex. do not parallelize a loop unless its length is sufficiently large).
- Avoid parallelizing the loop which refers to the first dimension of arrays (in Fortran) because it is the one which refers to 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
```

- The conflicts between threads can lead to "false sharing" (explosion of cache-misses), which may significantly diminish the performance.
- On a `NUMA` (Non-Uniform Memory Access) machine (ex. `SGI-O2000`), local memory references are faster than non-local references ; this can lead to different program behaviour depending on how data are mapped on memory by the system.
- Regardless of the architecture of machines, quality of the `OpenMP` implementation can quite significantly affect the scalability of parallel loops and the overall performance.

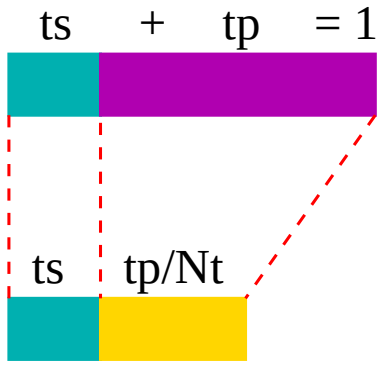
6.2 – Time measurements

- OpenMP provides two functions :
  - OMP_GET_WTIME in order to measure the elapsed time in seconds ;
  - OMP_GET_WTICK in order to get the precision of measurements in seconds.
- What we measure is the elapsed time from an arbitrary reference point of the code.
- This measure can vary from one execution to another according to the machine workload and the distribution of threads on the cores.

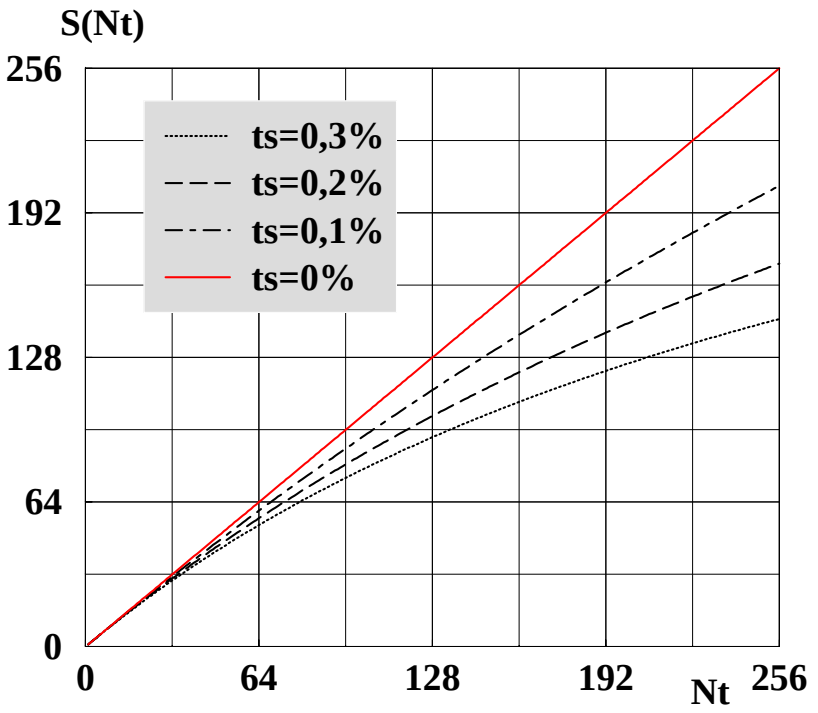
```
program mat_vect
 !$ use OMP_LIB
 implicit none
 integer,parameter   :: n=1025
 real,dimension(n,n) :: a
 real,dimension(n)   :: x, y
 real(kind=8) :: t_ref, t_final
 integer :: rank
 call random_number(a)
 call random_number(x) ; y(:)=0.
 !$OMP PARALLEL &
  !$OMP PRIVATE(rank,t_ref,t_final)
  rank = OMP_GET_THREAD_NUM()
  t_ref=OMP_GET_WTIME()
  call prod_mat_vect(a,x,y,n)
  t_final=OMP_GET_WTIME()
  print *,"Rank :",rank, &
          "; res :",t_final-t_ref
 !$OMP END PARALLEL
end program mat_vect
```

6.3 – Speedup

- The performance gain of a parallel code is estimated by comparing a parallel execution to a sequential one.
- 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 relative time of the sequential part and t_p the relative time of the parallelizable part of the code), the "AMDHAL" 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.



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



Position in the Course

7 – Conclusion

- ☞ OpenMP requires a multi-processor computer with shared memory.
- ☞ Parallelization is relatively easy to implement, even when starting from a sequential program.
- ☞ OpenMP allows incremental parallelization.
- ☞ The full potential of parallel performance lies in the parallel regions.
- ☞ Within these parallel regions, the work can be shared through the parallel loops and the parallel sections. However, a thread can also be singled out for a particular work.
- ☞ The orphaned directives allow the development of parallel routines and libraries.
- ☞ Point-to-point or global explicit synchronizations are sometimes necessary in the parallel regions.
- ☞ Special attention must be payed to the definition of the data-sharing attribute of used variables in a construct.
- ☞ The "speedup" measures the code scalability. It is limited by the sequential fraction of the program and is lowered by the additional costs due to thread management.

8 – Annexes

What we did not (or only briefly) cover in this course :

- The lock procedures for point-to-point synchronization ;
- Other service routines ;
- The MPI & OpenMP hybrid parallelization ;
- The new features of OpenMP 3.0 (the tasking concept, the improvements related to the parallel loops and nesting, etc.).