Shared Memory - OpenMP

Dept. Arquitectura de Computadors i Sistemes Operatius
Universitat Autònoma de Barcelona

Universitat Autònoma de Barcelona

Shared Memory

- OpenMP is currently the programming standard for the share memory model on multicore systems.

l'Autònoma

Shared Memory

- Thread based model.
- Threads read and write shared variables.
- Synchronization mechanism are offered.
- It is possible to change the attributes of threads and data for minimizing synchronization.

l'Autònoma

OpenMP

- **OpenMP is not an automatic parallelization tool:**
  - Programmers must specify parallelism explicitly.
- **OpenMP is not only for exploiting loop parallelism:**
  - It also offers functionalities for other forms of parallelism.

l'Autònoma

OpenMP

- **OpenMP is not a programming language:**
  - It is structured as extensions using directives to base languages like Fortran or C.
- **OpenMP is not only a research project:**
  - Many commercial compilers support OpenMP.

l'Autònoma

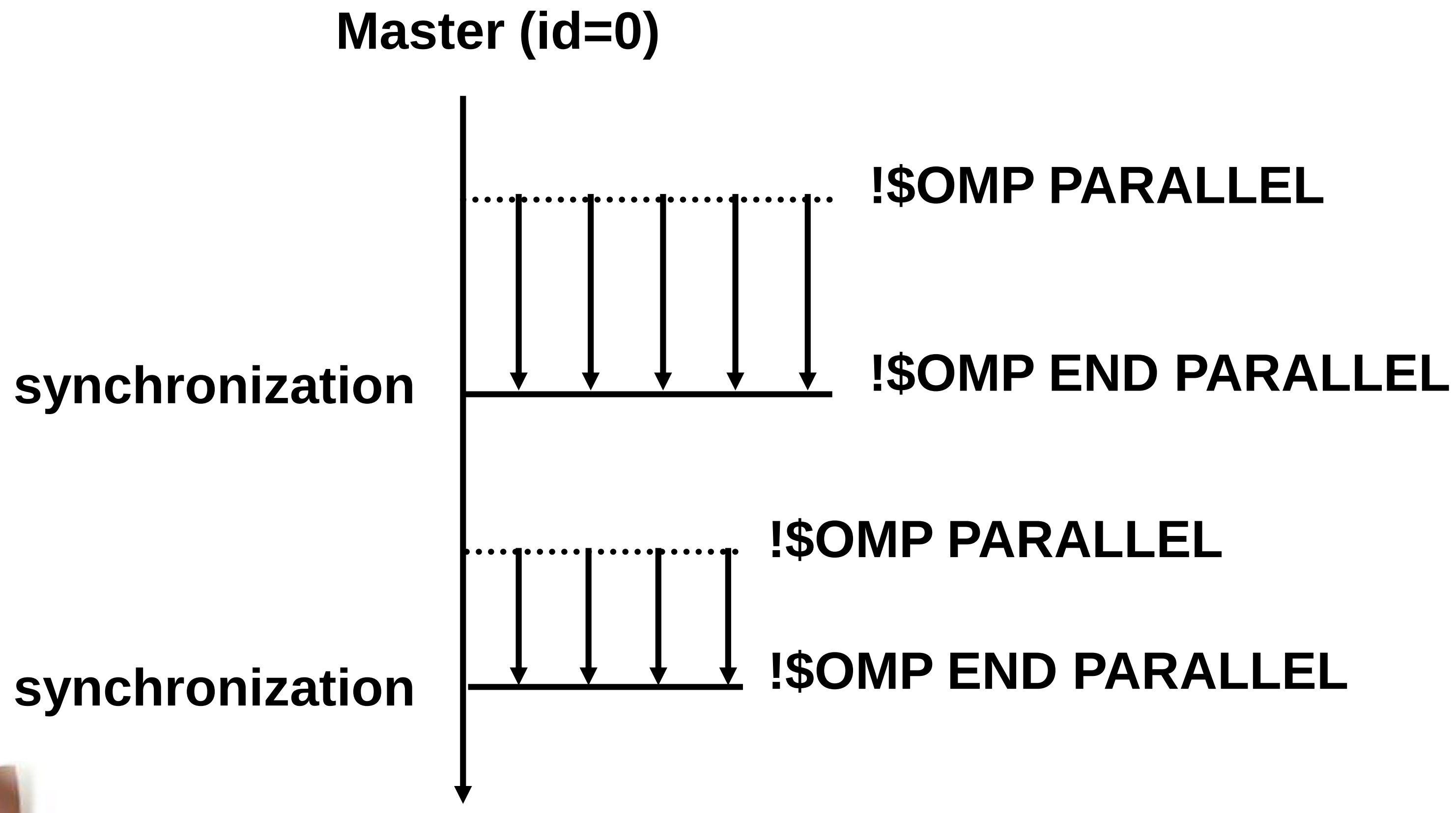
OpenMP

- OpenMP is an API (Application Program Interface) for parallel programming in shared memory systems.
- It’s main objective is to easily parallelize existing applications.
- It’s based in directives introduced in the program as special comments (pragmas).

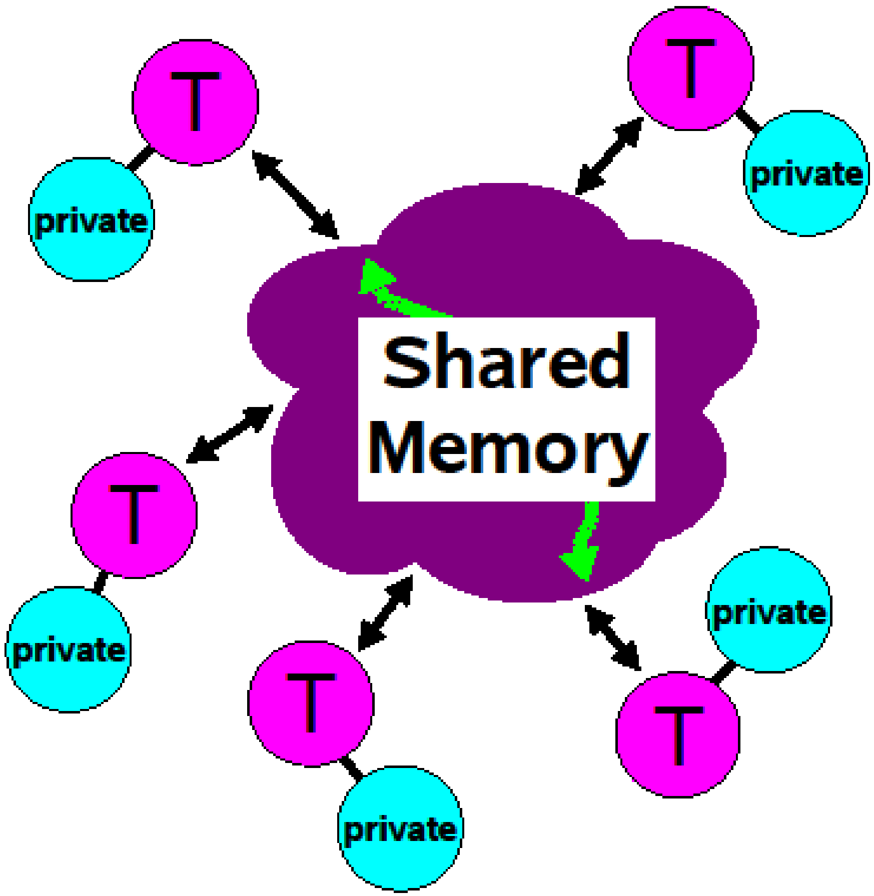
l'Autònoma

OpenMP Execution Model

- Initially based on a FORK-JOIN model



Shared Memory Model



Programming Model

- *All threads have access to the same, globally shared, memory*
- *Data can be shared or private*
- *Shared data is accessible by all threads*
- *Private data can be accessed only by the threads that owns it*
- *Data transfer is transparent to the programmer*
- *Synchronization takes place, but it is mostly implicit*

About Data

- *In a shared memory parallel program variables have a "label" attached to them:*
  - *Labelled "Private" ⇨ Visible to one thread only*
    - ✔ *Change made in local data, is not seen by others*
    - ✔ *Example - Local variables in a function that is executed in parallel*
  - *Labelled "Shared" ⇨ Visible to all threads*
    - ✔ *Change made in global data, is seen by all others*
    - ✔ *Example - Global data*

Components of OpenMP

Directives

- *Parallel regions*
- *Work sharing*
- *Synchronization*
- *Data scope attributes*
  - *private*
  - *firstprivate*
  - *lastprivate*
  - *shared*
  - *reduction*
- *Orphaning*

Environment variables

- *Number of threads*
- *Scheduling type*
- *Dynamic thread adjustment*
- *Nested parallelism*

Runtime environment

- *Number of threads*
- *Thread ID*
- *Dynamic thread adjustment*
- *Nested parallelism*
- *Timers*
- *API for locking*

User Interface

- **Compiler directives**
  - There are control structures and data attributes structures.
  - Compilers ignore these directives (they are just comments) unless the proper options are used when compiling (“-mp” or “-fopenmp“).

l'Autònoma

User Interface

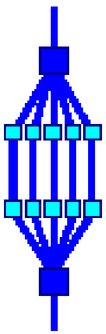
- **Library**
  - Set of functions for controlling some parameters, such as the number of threads to be

```
call omp_set_num_threads (128)
```

- **But also environment variables**
  - Another way of doing the same

```
setenv OMP_NUM_THREADS 8
```

The parallel region



A parallel region is a block of code executed by multiple threads simultaneously

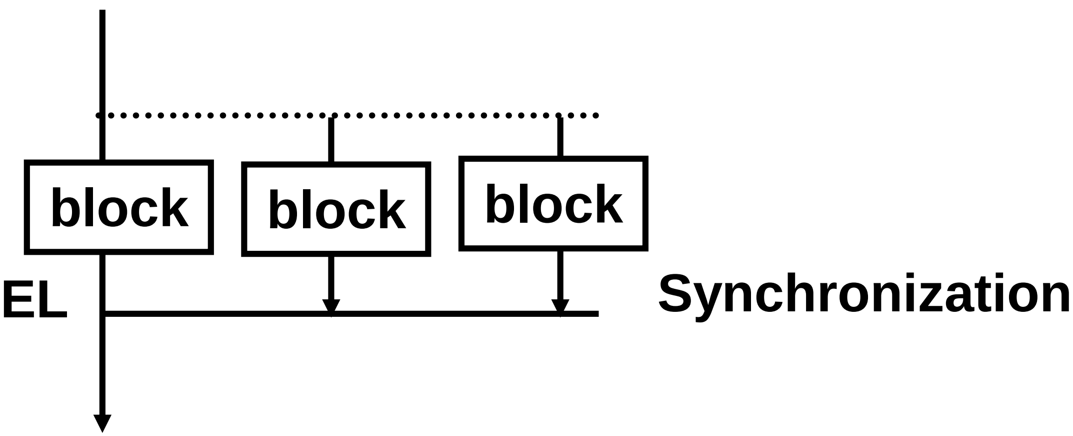
```
#pragma omp parallel [clause[[,] clause] ...]
{
    "this will be executed in parallel"

} (implied barrier)
```

```
!$omp parallel [clause[[,] clause] ...]

    "this will be executed in parallel"

!$omp end parallel (implied barrier)
```

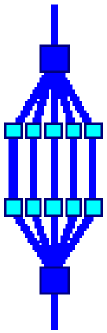
Directives

› **PARALLEL / END PARALLEL**
  - Define a parallel region
  - It does the “fork” and “join”.
  - The number of threads is constant in the parallel regions

```
!$OMP PARALLEL
      block
!$OMP END PARALLEL
```



The parallel region - clauses



A parallel region supports the following clauses:

if	*(scalar expression)*	
private	*(list)*	
shared	*(list)*	
default	*(none|shared)*	*(C/C++)*
default	*(none|shared|private)*	*(Fortran)*
reduction	*(operator: list)*	
copyin	*(list)*	
firstprivate	*(list)*	
num_threads	*(scalar_int_expr)*	

lastprivate(list)

A more elaborate example

```
#pragma omp parallel if (n>limit) default(none) \
        shared(n,a,b,c,x,y,z) private(f,i,scale)
{
    f = 1.0;
#pragma omp for nowait

    for (i=0; i<n; i++)
       z[i] = x[i] + y[i];

#pragma omp for nowait

    for (i=0; i<n; i++)
       a[i] = b[i] + c[i];

#pragma omp barrier
          ....
    scale = sum(a,0,n) + sum(z,0,n) + f;
          ....
} /*-- End of parallel region --*/
```

Statement is executed by all threads

parallel loop (work will be distributed)

parallel loop (work will be distributed)

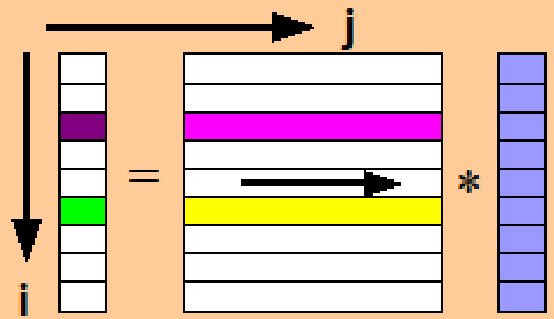
synchronization

Statement is executed by all threads

parallel region

Example - Matrix times vector

```
#pragma omp parallel for default(none) \
            private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
   sum = 0.0;
   for (j=0; j<n; j++)
     sum += b[i][j]*c[j];
   a[i] = sum;
 }
```



TID = 0

```
for (i=0,1,2,3,4)
 i = 0
   sum = Σ b[i=0][j]*c[j]
   a[0] = sum
 i = 1
   sum = Σ b[i=1][j]*c[j]
   a[1] = sum
```

TID = 1

```
for (i=5,6,7,8,9)
 i = 5
   sum = Σ b[i=5][j]*c[j]
   a[5] = sum
 i = 6
   sum = Σ b[i=6][j]*c[j]
   a[6] = sum
```

... etc ...

The default clause

default (none | shared | private)	Fortran
default (none | shared)	C/C++

Note: default(private) is not supported in C/C++

none

- ✔ *No implicit defaults*
- ✔ *Have to scope all variables explicitly*

shared

- ✔ *All variables are shared*
- ✔ *The default in absence of an explicit "default" clause*

private

- ✔ *All variables are private to the thread*
- ✔ *Includes common block data, unless THREADPRIVATE*

The reduction clause

reduction ([operator | intrinsic]) : list) *Fortran*

reduction (operator : list) *C/C++*

- ✔ *Reduction variable(s) must be shared variables*
- ✔ *A reduction is defined as:*

Check the docs for details

Fortran

```
x = x operator expr
x = expr operator x
x = intrinsic (x, expr_list)
x = intrinsic (expr_list, x)
```

C/C++

```
x = x operator expr
x = expr operator x
x++, ++x, x--, --x
x <binop> = expr
```

- ✔ *Note that the value of a reduction variable is undefined from the moment the first thread reaches the clause till the operation has completed*
- ✔ *The reduction can be hidden in a function call*

The reduction clause - example

```
      sum = 0.0
!$omp parallel default(none) &
!$omp shared(n,x) private(i)
!$omp do reduction (+:sum)
      do i = 1, n
         sum = sum + x(i)
      end do
!$omp end do
!$omp end parallel
      print *,sum
```

Variable SUM is a shared variable

- *Care needs to be taken when updating shared variable SUM*
- *With the reduction clause, the OpenMP compiler generates code such that a race condition is avoided*

The nowait clause

- *To minimize synchronization, some OpenMP directives/pragmas support the optional* ***nowait*** *clause*
- *If present, threads will not synchronize/wait at the end of that particular construct*
- *In Fortran the nowait is appended at the closing part of the construct*
- *In C, it is one of the clauses on the pragma*

```
#pragma omp for nowait
{
        :
}
```

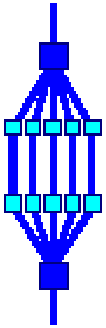
```
!$omp do
       :
       :
!$omp end do nowait
```

Directives

- **Omp for - DO / END DO**
  - It’s for classical parallel loops
  - It must be in a parallel region
  - Loop iterations are distributed among available threads
  - Loop index is by default private to each thread

l'Autònoma

The omp for/do directive



The iterations of the loop are distributed over the threads

```
#pragma omp for [clause[[,] clause] ...]
   <original for-loop>
```

```
!$omp do [clause[[,] clause] ...]
        <original do-loop>
!$omp end do [nowait]
```

Clauses supported:

private **firstprivate**
lastprivate **reduction**
ordered* ***schedule*** ← *covered later*
nowait

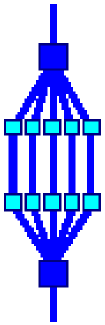
**) Required if ordered sections are in the dynamic extent of this construct*

DO / END DO

```
Program example
dimension A(100),B(100)
Integer i
!$OMP PARALLEL
  !$OMP DO
      Do i=2,100
           B(i)= (A(i)+A(i-1))/2.0
      End Do
  !$OMP END DO
!$OMP END PARALLEL
Return
End
```

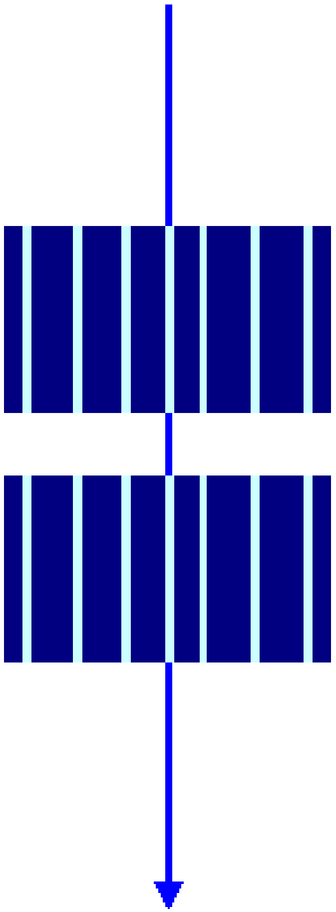
l'Autònoma

The omp for directive - example



```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp for nowait
      for (i=0; i<n-1; i++)
          b[i] = (a[i] + a[i+1])/2;
    #pragma omp for nowait
      for (i=0; i<n; i++)
          d[i] = 1.0/c[i];

  } /*-- End of parallel region --*/
```

(implied barrier)



Another OpenMP example

```
 1 void mxv_row(int m,int n,double *a,double *b,double *c)
 2 {
 3  int i, j;
 4  double sum;
 5
 6 #pragma omp parallel for default(none) \
 7             private(i,j,sum) shared(m,n,a,b,c)
 8  for (i=0; i<m; i++)
 9  {
10     sum = 0.0;
11     for (j=0; j<n; j++)
12       sum += b[i*n+j]*c[j];
13      a[i] = sum;
14  } /*-- End of parallel for --*/
15 }
```

```
% cc -c -fast -xrestrict -xopenmp -xloopinfo mxv_row.c
"mxv_row.c", line  8: PARALLELIZED, user pragma used
"mxv_row.c", line 11: not parallelized
```

#pragma omp for ordered *[clauses...]*

(loop region)

#pragma omp ordered

/*This code is executed in the same order than the sequential execution (1 thread at a time)*/

structured_block

(endo of loop region)

l'Autònoma

Load balancing

- *Load balancing is an important aspect of performance*
- *For regular operations (e.g. a vector addition), load balancing is not an issue*
- *For less regular workloads, care needs to be taken in distributing the work over the threads*
- *Examples of irregular worloads:*
  - *Transposing a matrix*
  - *Multiplication of triangular matrices*
  - *Parallel searches in a linked list*
- *For these irregular situations, the* schedule *clause supports various iteration scheduling algorithms*

Do Scheduling

- **SCHEDULE**
  - Controls how the iterations are assigned to threads.

!$OMP SCHEDULE (type,[number])

- type: Determines how
- number: Determines how many

l'Autònoma

The schedule clause/1

schedule (static | dynamic | guided [, chunk])
schedule (runtime)

static [, chunk]

- ✔ *Distribute iterations in blocks of size "chunk" over the threads in a round-robin fashion*
- ✔ *In absence of "chunk", each thread executes approx. N/P chunks for a loop of length N and P threads*

Example: *Loop of length 16, 4 threads:*

TID	0	1	2	3
no chunk	1-4	5-8	9-12	13-16
chunk = 2	1-2 9-10	3-4 11-12	5-6 13-14	7-8 15-16

The schedule clause/2

dynamic [, chunk]

- *Fixed portions of work; size is controlled by the value of chunk*
- *When a thread finishes, it starts on the next portion of work*

guided [, chunk]

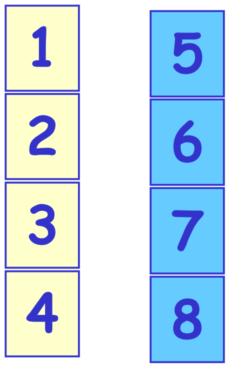
- *Same dynamic behaviour as "dynamic", but size of the portion of work decreases exponentially*

runtime

- *Iteration scheduling scheme is set at runtime through environment variable OMP_SCHEDULE*

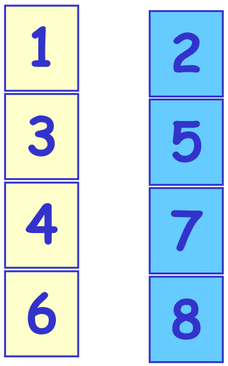
DO / END DO

```
Program example
dimension A(10),B(10)
Integer i
!$OMP PARALLEL
  !$OMP DO
        !$OMP SCHEDULE (STATIC)
              Do i=1,8
                    B(i)= A(i)/2.0
              End Do
  !$OMP END DO
!$OMP END PARALLEL
Return
End
```

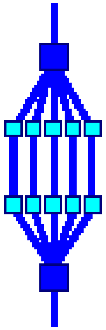


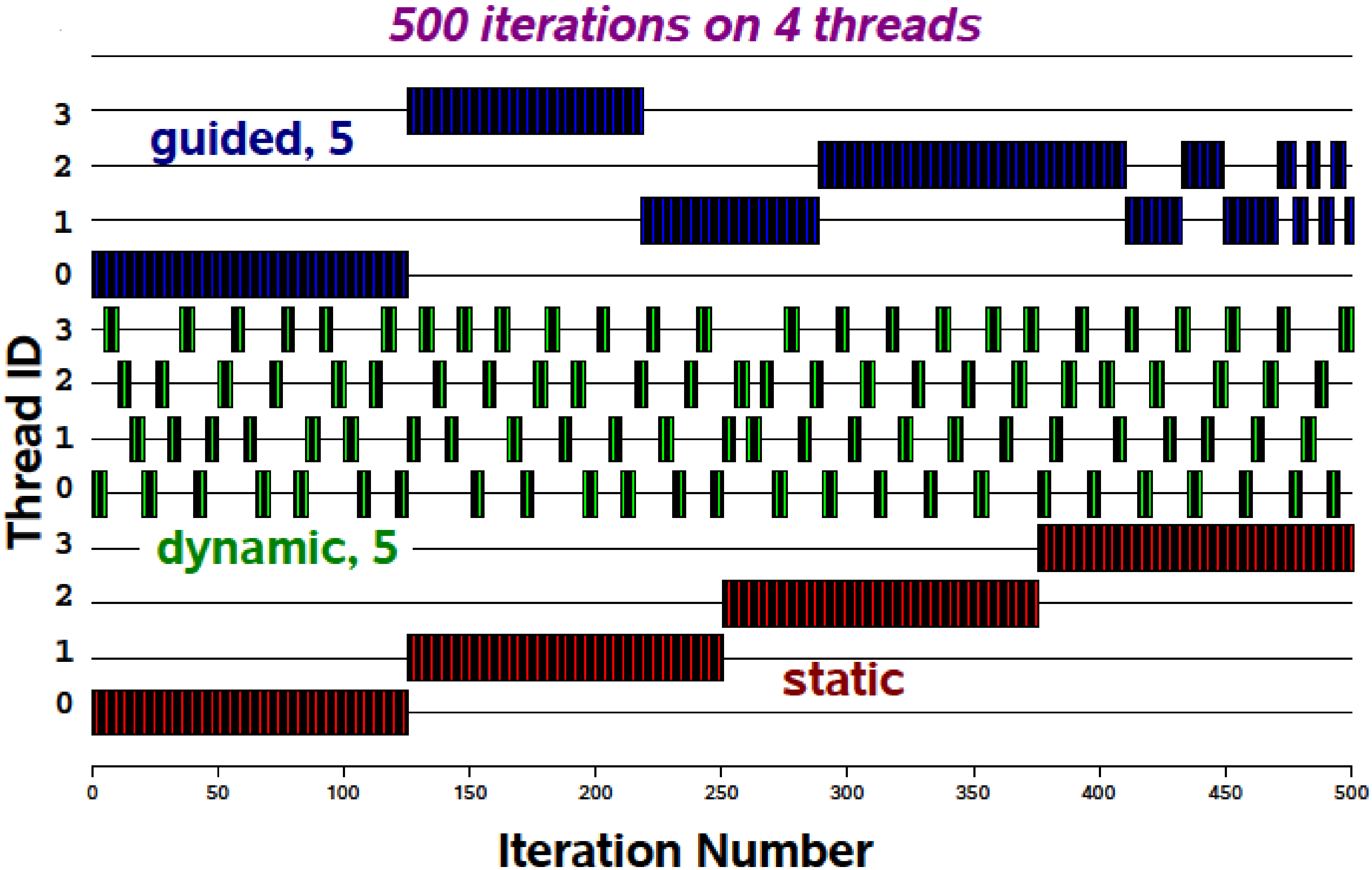
DO / END DO

```
Program example
dimension A(10),B(10)
Integer i
!$OMP PARALLEL
  !$OMP DO
      !$OMP SCHEDULE (DYNAMIC,1)
           Do i=1,8
                B(i)= A(i)/2.0
           End Do
  !$OMP END DO
!$OMP END PARALLEL
Return
End
```



The experiment





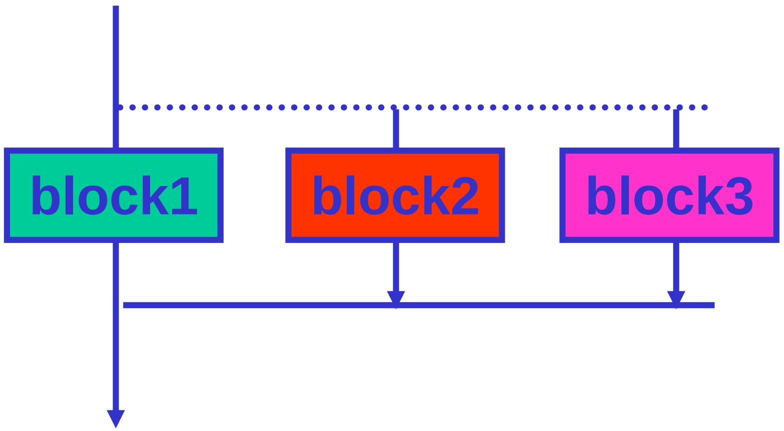
Directives

- **SECTIONS / END SECTIONS**
  - Most be in a parallel region
  - Sections are distributed among threads
  - Each thread executes a different section
  - Allows task level parallelism
  - SECTION pragma defines each section

l'Autònoma

SECTIONS / END SECTIONS

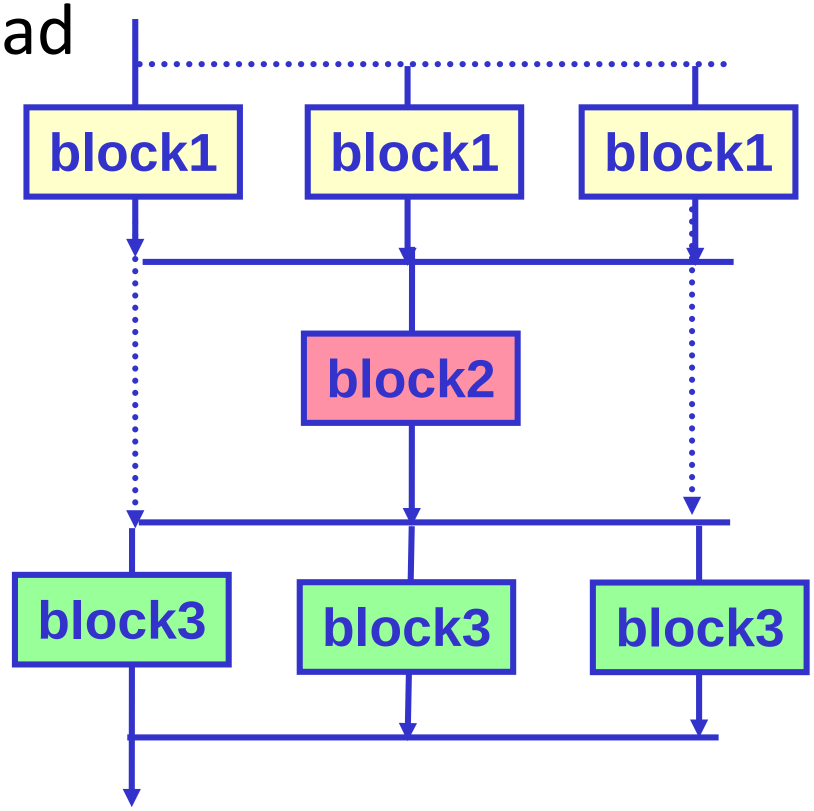
```
!$OMP PARALLEL
  !$OMP SECTIONS
            BLOCK1
        !$OMP SECTION
            BLOCK2
        !$OMP SECTION
            BLOCK3
  !$OMP END SECTIONS
!$OMP END PARALLEL
```



Directives

› **SINGLE / END SINGLE**
  - The code included in the single section will be executed by only one thread

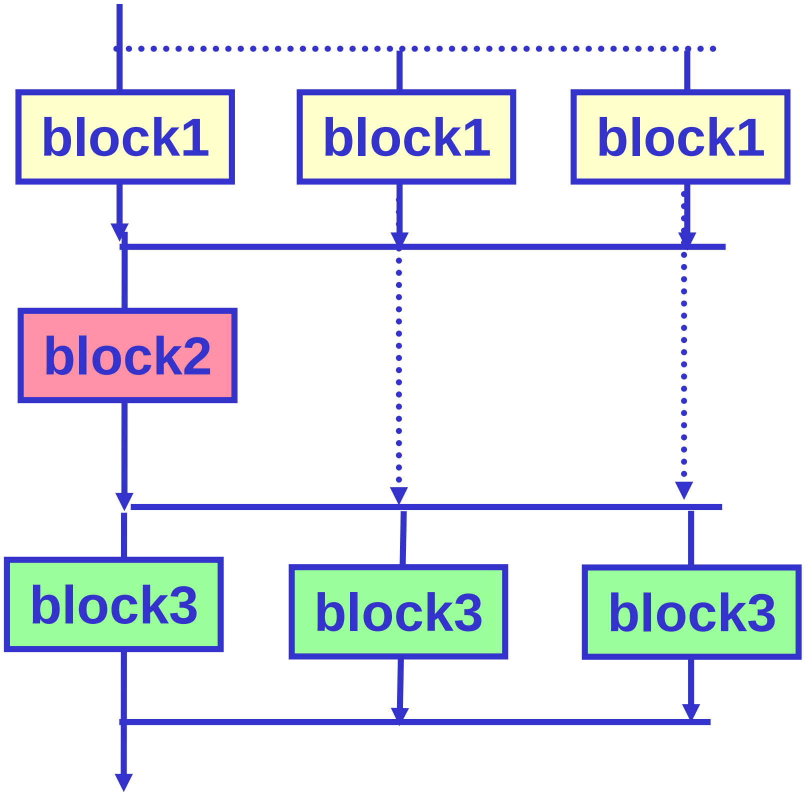
```
!$OMP PARALLEL
            BLOCK1
      !$OMP SINGLE
            BLOCK2
      !$OMP END SINGLE
            BLOCK3
!$OMP END PARALLEL
```



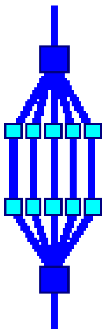
Directives

- **MASTER / END MASTER**
  - Code included in the master section will be only executed by the master thread

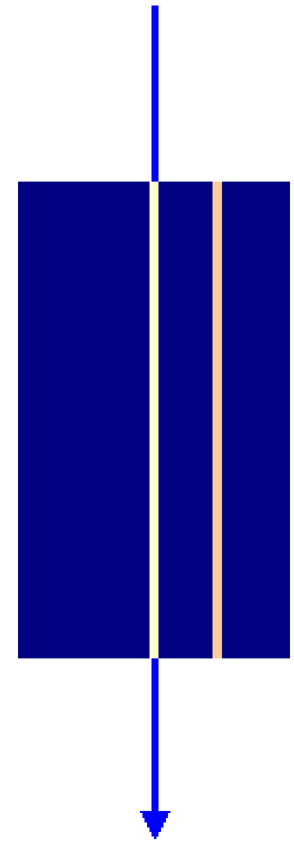
```
!$OMP PARALLEL
            BLOCK1
      !$OMP MASTER
            BLOCK2
      !$OMP END MASTER
            BLOCK3
!$OMP END PARALLEL
```



The sections directive - example



```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
       for (i=0; i<n-1; i++)
           b[i] = (a[i] + a[i+1])/2;

      #pragma omp section
       for (i=0; i<n; i++)
           d[i] = 1.0/c[i];

    } /*-- End of sections --*/

  } /*-- End of parallel region --*/
```



Short-cuts

Single PARALLEL loop

```
#pragma omp parallel
#pragma omp for
   for (...)
```

```
#pragma omp parallel for
for (....)
```

```
!$omp parallel
!$omp do
      ...
!$omp end do
!$omp end parallel
```

```
!$omp parallel do
      ...
!$omp end parallel do
```

Single PARALLEL sections

```
#pragma omp parallel
#pragma omp sections
{ ...}
```

```
#pragma omp parallel sections
{ ... }
```

```
!$omp parallel
!$omp sections
      ...
!$omp end sections
!$omp end parallel
```

```
!$omp parallel sections
      ...
!$omp end parallel sections
```

Single WORKSHARE loop

```
!$omp parallel
!$omp workshare
      ...
!$omp end workshare
!$omp end parallel
```

```
!$omp parallel workshare
      ...
!$omp end parallel workshare
```

Barrier/1

Suppose we run each of these two loops in parallel over i:

```
for (i=0; i < N; i++)
   a[i] = b[i] + c[i];
```

```
for (i=0; i < N; i++)
   d[i] = a[i] + b[i];
```

This may give us a wrong answer (one day)

Why ?

Barrier/2

We need to have <u>updated all of a[ ]</u> first, before using a[ ]

```
for (i=0; i < N; i++)
   a[i] = b[i] + c[i];
```

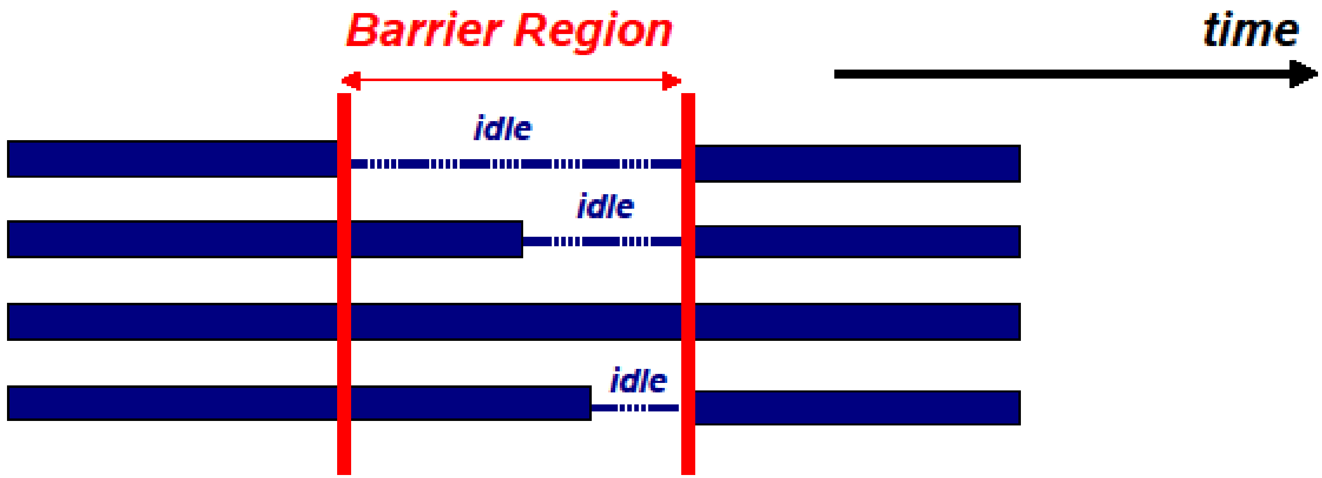
wait !

barrier

```
for (i=0; i < N; i++)
   d[i] = a[i] + b[i];
```

All threads wait at the barrier point and only continue when all threads have reached the barrier point

Barrier/3



Each thread waits until all others have reached this point:

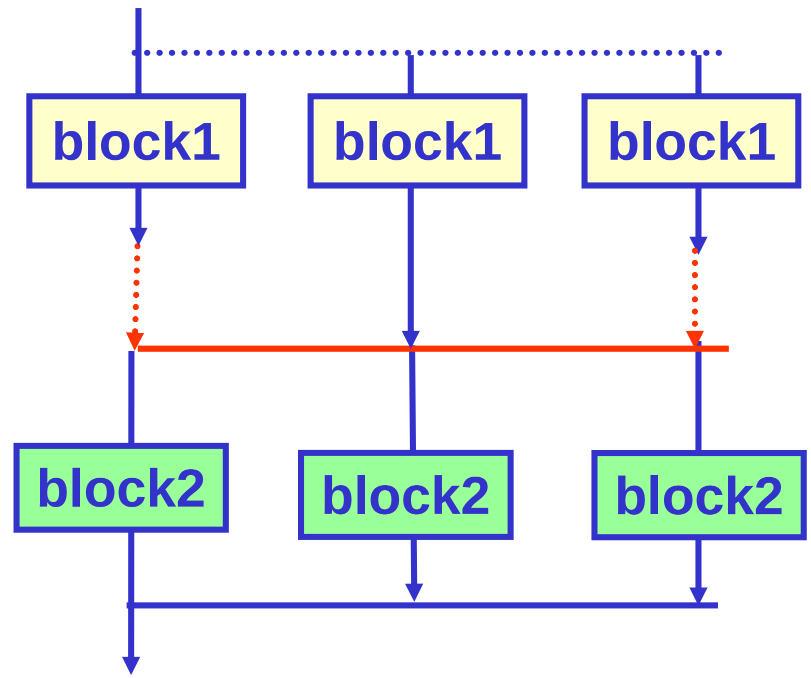
```
#pragma omp barrier
```

```
!$omp barrier
```

Barrier

- **BARRIER**
  - All threads wait until the last arrives to the barrier

```
!$OMP PARALLEL
        BLOCK1
    !$OMP BARRIER
        BLOCK2
!$OMP END PARALLEL
```



l'Autònoma

Critical region/1

If sum is a shared variable, this loop can not be run in parallel

```
for (i=0; i < N; i++){
      .....
   sum += a[i];
      .....
}
```

We can use a critical region for this:

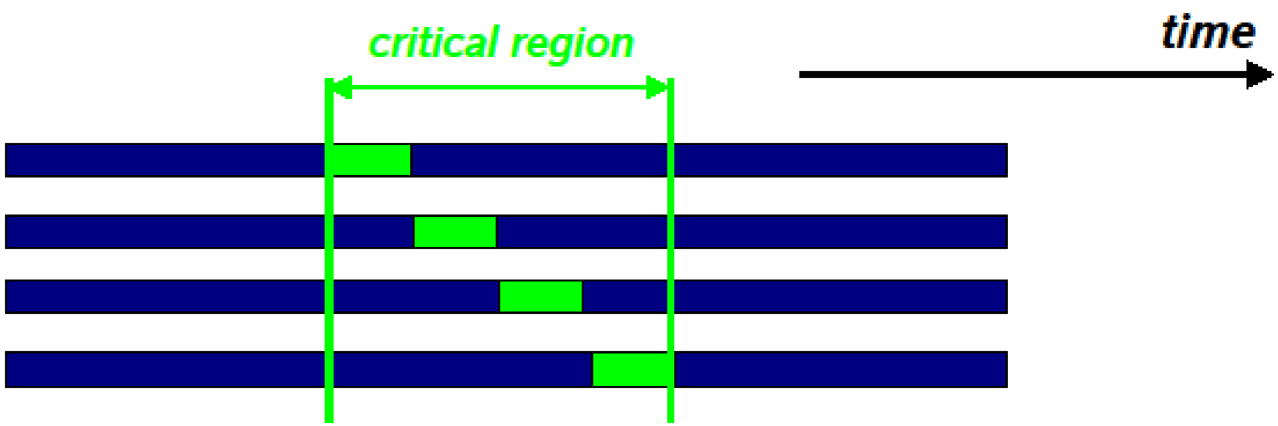
```
for (i=0; i < N; i++){
      .....
   sum += a[i];
      .....
}
```

one at a time can proceed

next in line, please

Critical region/2

- ***Useful to avoid a race condition, or to perform I/O (but which still will have random order)***
- ***Be aware that your parallel computation may be serialized and so this could introduce a scalability bottleneck (Amdahl's law)***



CRITICAL / END CRITICAL

```
!$OMP PARALLEL
        BLOCK1
   !$OMP CRITICAL
        BLOCK2
   !$OMP END CRITICAL
        BLOCK3
   !$OMP CRITICAL
        BLOCK4
   !$OMP END CRITICAL
        BLOCK5
!$OMP END PARALLEL
```

l'Autònoma

Synchronization

- **There are implicit barriers in:**
  - PARALLEL / END PARALLEL
  - DO / END DO
  - SECTIONS / END SECTIONS
  - SINGLE / END SINGLE
- **The NOWAIT pragma avoids this barrier**

OpenMP 3.0

- **TASK Construct**
  - The TASK construct defines an explicit task, which may be executed by the encountering thread, or deferred for execution by any other thread in the team.
  - The data environment of the task is determined by the data sharing attribute clauses.

OpenMP 3.0

#pragma omp task *[clause ...]*
if *(scalar expression)*
final *(scalar expression)*
untied
default (shared | none)
mergeable
private *(list)*
firstprivate *(list)*
shared *(list)*
structured_block

l'Autònoma

OpenMP 3.0

- The TASKWAIT construct specifies a wait on the completion of child tasks generated since the beginning of the current task.

#pragma omp taskwait

```
struct node {
struct node *left;
struct node *right;
};
extern void process(struct node *);
void traverse( struct node *p ) {
        if (p->left)
                #pragma omp task // p is firstprivate by default
                traverse(p->left);
        if (p->right)
                #pragma omp task // p is firstprivate by default
                traverse(p->right);
        #pragma omp taskwait
         process(p);
}
```

l'Autònoma

```c
#include <stdio.h>
int main()
{
	int x = 1;
	#pragma omp parallel
	#pragma omp single
	{
		#pragma omp task shared(x) depend(out: x)
		x = 2;
		#pragma omp task shared(x) depend(in: x)
		printf("x = %d\n", x);
	}
	return 0;
}
```

l'Autònoma

OpenMP 3.0

- **TASKGROUP Construct**
  - The TASKGROUP construct specifies a wait on completion of child tasks of the current task and their descendent tasks
  - A TASKGROUP region binds to the current task region. The binding thread set of the taskgroup region is the current team
- **TASKYIELD Construct**
  - The TASKYIELD construct specifies that the current task can be suspended in favor of execution of a different task

l'Autònoma

OpenMP 4.0

- **Support thread affinity policies** (proc_bind, get_proc_bin, OMP_PLACES)
- **Support execution on devices** (accelerators) (omp_set_default_device, omp_get_default_device, omp_get_num_devices, omp_get_num_teams, omp_get_team_num, and omp_is_initial_device)
- **Reduction clause extended to support user defined reductions**
- **The concept of cancellation is added**

l'Autònoma

Run-time Library

- **OMP_SET_NUM_THREADS (SCALAR)**
  - Sets the number of threads that will be used in the next parallel region
  - Only works if called from a sequential portion of the program

l'Autònoma

Run-time Library

- **OMP_GET_NUM_THREADS ()**
  - Returns the number of threads in the parallel region where it's called
  - The default number of threads depends on the application

Run-time Library

- **`OMP_GET_THREAD_NUM ()`**
  - Returns the thread id of the thread that calls it
  - Master thread has id 0

l'Autònoma

Run-time Library

- omp_in_parallel
- omp_set_dynamic
- omp_get_dynamic
- omp_get_cancellation
- omp_set_nested
- omp_get_nested
- omp_set_schedule
- omp_get_schedule
- omp_get_thread_limit
- omp_set_max_active_levels
- omp_get_max_active_levels
- **And so on and so forth**

l'Autònoma

References

- Ruud van der Pas "An Introduction Into OpenMP". Sun Microsystems. iWOMP 2005.
- OpenMP Architecture Review Board. OpenMP Application Program Interface. Version 2.5. May 2005.
- OpenMP Application Program Interface. Version 4.0. July 2013
- www.openmp.org
- computing.llnl.gov/tutorials/openMP/

l'Autònoma