

# OpenMP basics: Communication within a node

**Gian Franco Marras**

[g.marras@cineca.it](mailto:g.marras@cineca.it)

Dipartimento Supercalcolo, Applicazioni e Innovazione (SCAI)

CINECA - High Performance System Group

Casalecchio di Reno (BO) - Italy

[www.cineca.it](http://www.cineca.it)



## Shared Memory System

Shared memory refers to a large block of RAM that can be accessed by several different CPUs in a multiple-processor computer system.

Usually the system is a Symmetric MultiProcessor (SMP). SMP involves a multiprocessor computer hardware architecture where two or more identical processors are connected to a single shared main memory.



The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms.

OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.



## **Pros of OpenMP**

- easier to program and debug;
- directives can be added incrementally;
- gradual parallelization;
- can still run the program as a serial code;
- serial code statements usually don't need modification.

## **Cons of OpenMP**

- can only be run in shared memory computers;
- Traffic between CPU and memory increases with the number of CPUs;



OpenMP consists of a set of:

- **Compiler directives**
- **Runtime library routines**
- **Environment variables**



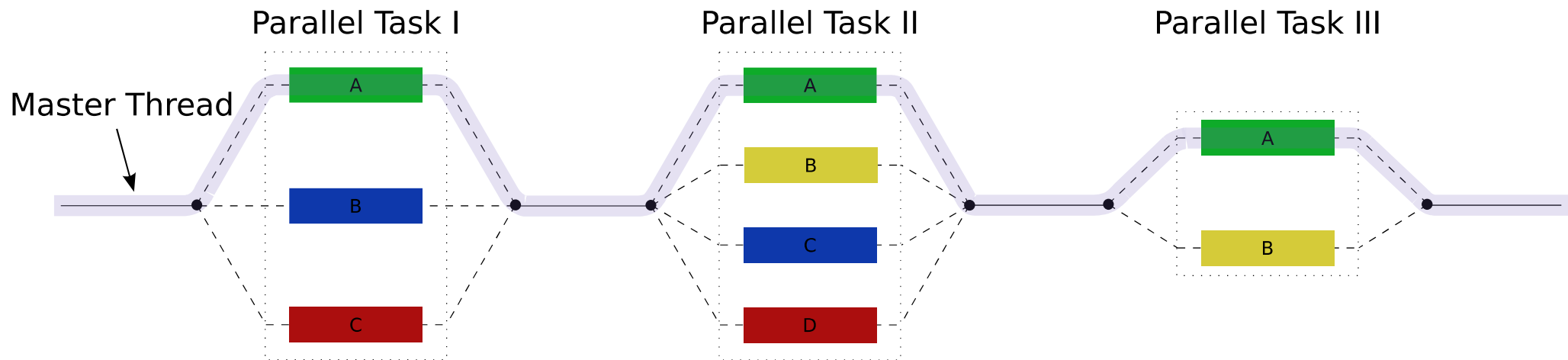
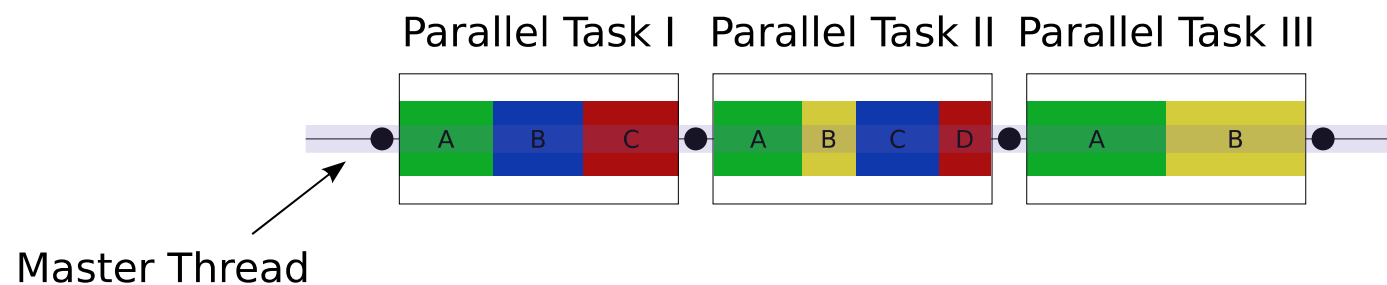
The OpenMP API uses the fork-join model of parallel execution.

An OpenMP program begins as a single thread of execution, called the initial thread. The initial thread executes sequentially until encounters a parallel construct.

The thread creates a team of tasks and becomes the master of the new team. Each task is assigned to a different thread in the team and becomes tied. Beyond the end of the parallel construct, only the master thread resume execution.



OpenMP supports dynamic adjustment of the number of thread:





## Conditional Compilation

With OpenMP compilation, the **\_OPENMP** macro is defined.

C:

```
#ifdef _OPENMP
```

```
printf("Compiled by OpenMP);
```

```
#else
```

```
printf("Compiled by an Serial-compliant  
implementation.");
```

```
#endif
```

Fortran:

```
!$ print *, "Compiled by OpenMP "
```





## Parallel construct

- Start parallel execution;
- A team of threads is created to execute the parallel region;
- The thread that encountered the parallel construct become the master thread of the new team with a thread number zero.
- There is an implicit barrier at the end of the construct;
- Any number of parallel constructs can be specified in a single program;



## A first program in Fortran:

```
PROGRAM HELLO
```

```
INTEGER VAR1, VAR2, VAR3
```

```
!Serial code
```

```
!Beginning of parallel region.
```

```
!Fork a team of threads.
```

```
!Specify variable scoping.
```

```
!$OMP PARALLEL PRIVATE(VAR1, VAR2) SHARED(VAR3)
```

```
Print *, "Hello World!!!"
```

```
!$OMP END PARALLEL
```

```
!Resume serial code
```

```
END
```



## A first program in C:

```
#include <omp.h>
int main ()
{
    int var1, var2, var3;
    Serial code
    ! Beginning of parallel region. Fork a team of threads.
    ! Specify variable scoping
    #pragma omp parallel private(var1, var2) shared(var3)
    {
        Parallel section executed by all threads
        All threads join master thread and disband
    }
    !Resume serial code
}
```



## OpenMP Memory Model

OpenMP provides a consistency shared-memory model. All threads have access to the **main memory** to retrieve shared variables.

Each thread also has access to another type of memory that must not be accessed by another threads, called **threadprivate memory**.

A directive that accepts data-sharing attribute clauses determines two kinds of access to variables used in the directive's associated structured block: **shared** and **private**.



## Data-Sharing Attribute Clauses

- **Shared**: declares one or more list items to be shared by tasks generated by a parallel construct. All changes made are visible to all threads.
- **Private**: declares one or more list items to be private to a task.  
No other thread can access this data.  
Changes can only visible to the thread owning the data.



## Worksharing construct

A worksharing construct distributes the execution of the associated region among the members of the team that encounters it.

A worksharing region has no barrier on entry; however, an implied barrier exists at the end of the worksharing region.

If a **nowait clause** is present, an implementation may omit the barrier at the end of the worksharing region.

- *Loop construct (DO / for)*
- *SECTIONS construct*
- *SINGLE construct*
- *WORKSHARE construct (only Fortran90)*



## Loop construct (DO/for)

The loop construct specifies that the iterations of one or more associated loops will be executed in parallel by threads in the team in the context of their implicit tasks. The iterations are distributed across threads that already exist in the team executing the parallel region to which the loop region binds.



## Loop construct (DO/for)

Fortran:

**!\$OMP PARALLEL**

integer :: i=5,n=200

real :: tmp

**!\$OMP DO PRIVATE(tmp)**

*do* *i*=1, *n*

*tmp* = *func* (*b* (*i*))

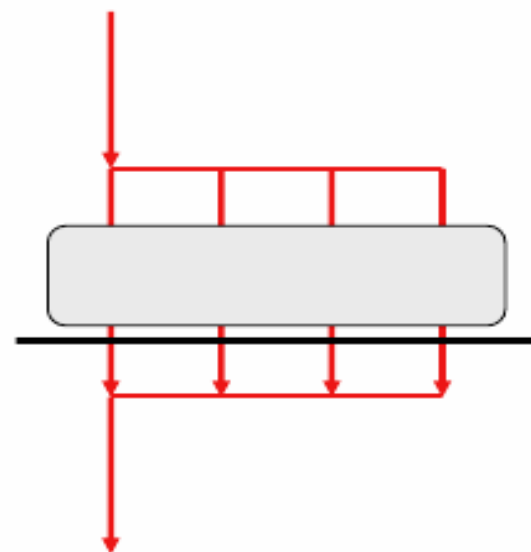
*a* (*i*) = *b* (*i*) + *tmp*

*end do*

**!\$OMP END DO [NOWAIT]**

print \*, i      !(→ 5)

**!\$OMP END PARALLEL**







# OpenMP Memory Model

Fortran:

**!\$OMP PARALLEL**

integer :: i=5,n=200

real :: tmp

**!\$OMP DO PRIVATE(tmp)**

*do* *i*=1, *n*

*tmp* = *func*(*b*(*i*))

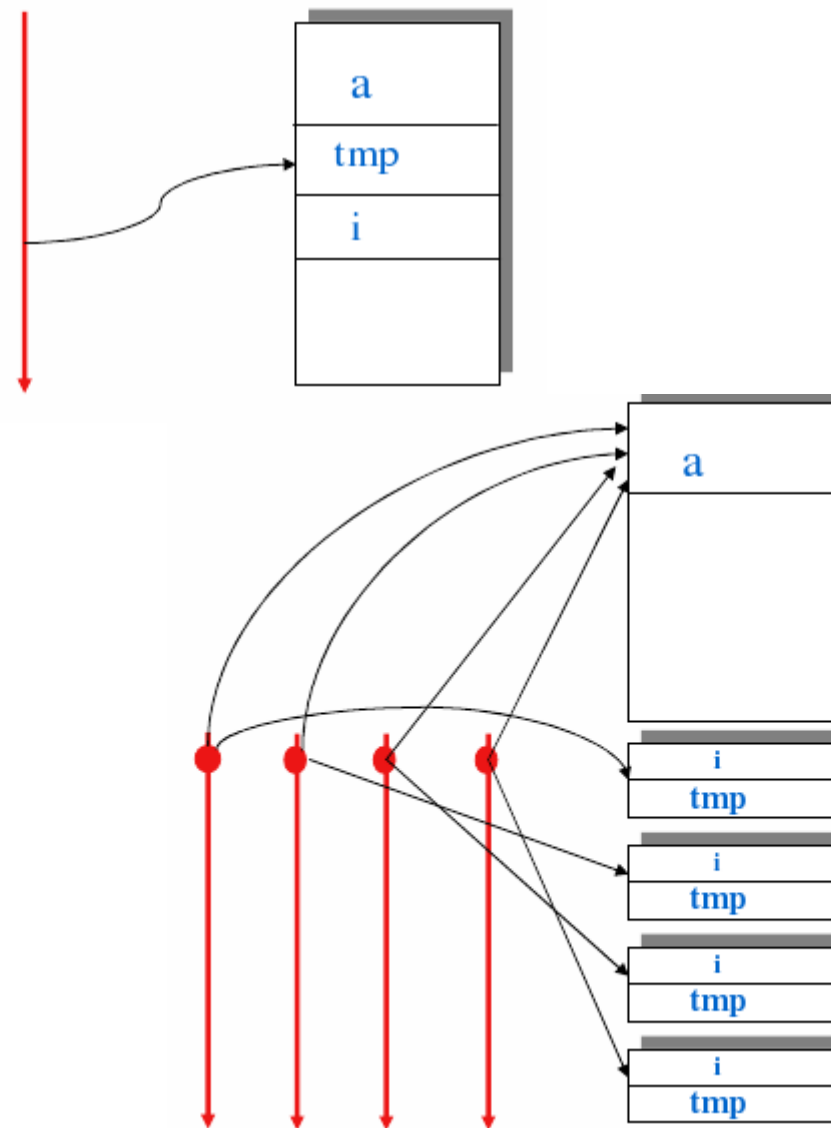
*a*(*i*) = *b*(*i*) + *tmp*

*end do*

**!\$OMP END DO [NOWAIT]**

print \*, i      !(→ 5)

**!\$OMP END PARALLEL**





## Loop construct (DO/for)

C/C++:

```
#pragma omp parallel
```

```
{
```

```
...
```

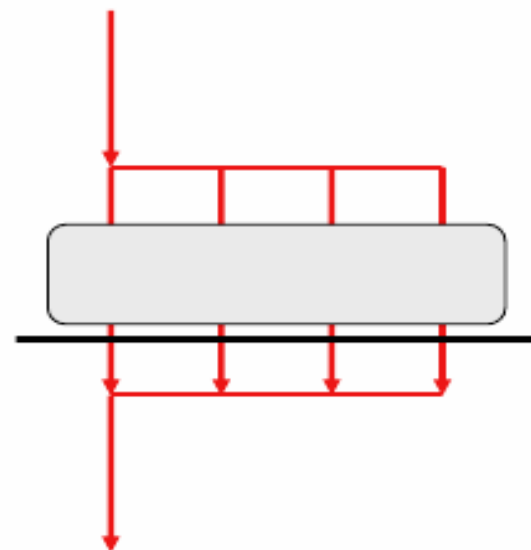
```
#pragma omp for [nowait]
```

```
for (i=0; i<n; ++i)
```

```
    a[i] = b[i] + 1.0
```

```
...
```

```
}
```





## Schedule clauses

Specifies how iterations of the associated loops are divided into contiguous non-empty subsets, called chunks, and how these chunks are distributed among threads of the team.

- *Static*: iterations are divided into chunks of size `chunk_size`, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number.
- *Dynamic*: iterations are distributed to threads in the team in chunks as the threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed.

`$omp parallel do schedule (type, [chunk])`



## Schedule clauses

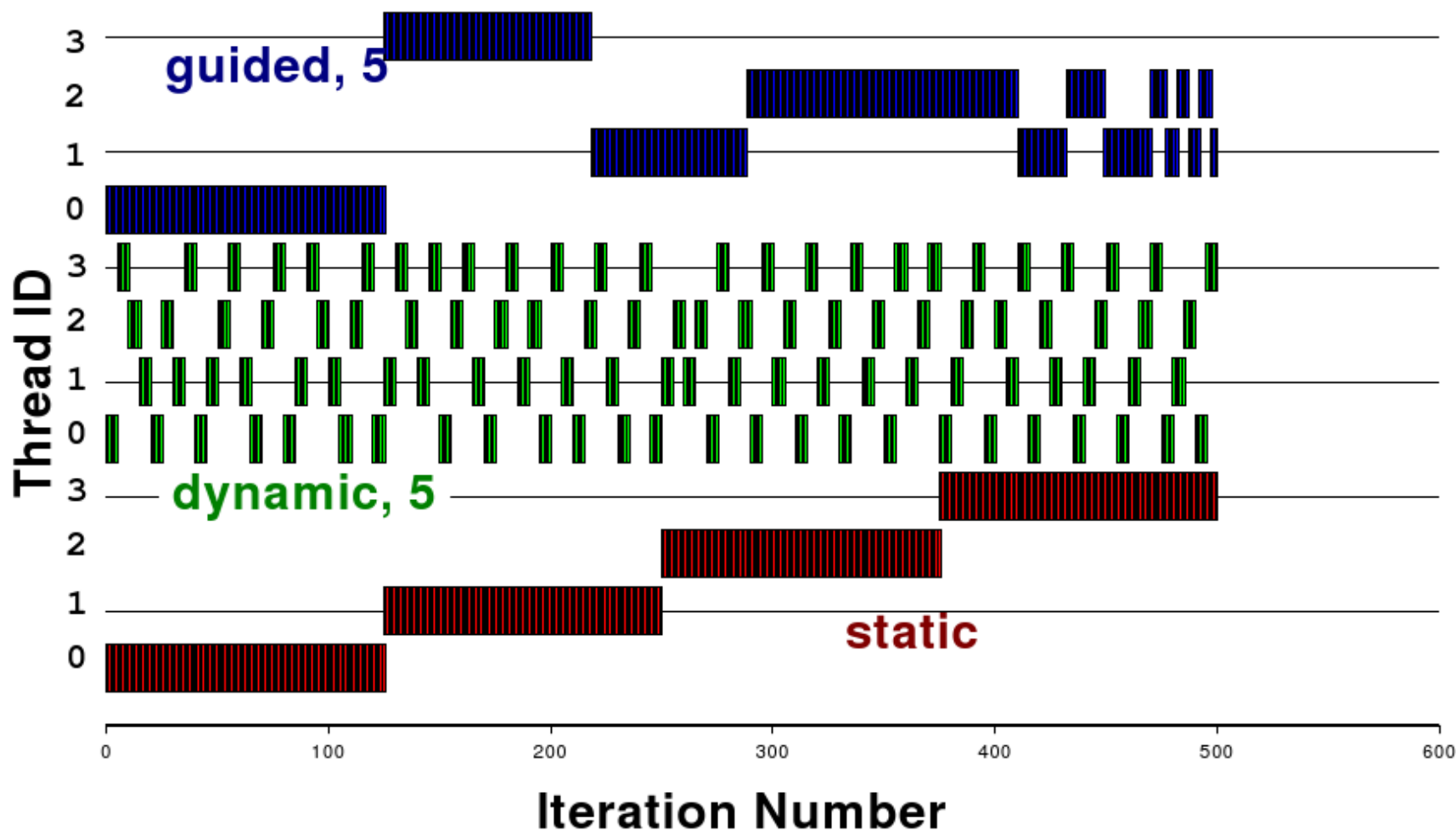
- *Guided*: the iterations are assigned to threads in the team in chunks as the executing threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned. The chunk decrease with time.
- *Runtime*: the decision regarding scheduling is deferred until run time.
- *Auto*: the decision regarding scheduling is delegated to the compiler and/or runtime system.

*\$omp parallel do schedule (type, [chunk])*



# Schedule clauses

*500 iterations on 4 threads*





## Environment Variables

- **OMP\_NUM\_THREADS**: sets the number of threads to use for parallel regions;
- **OMP\_SCHEDULE**: controls the schedule type and chunk size of all loop directives that have the schedule type runtime.

*csh :*

```
% setenv OMP_NUM_THREADS 8
```

```
% setenv OMP_SCHEDULE "guided,4"
```

*sh :*

```
$ export OMP_NUM_THREADS=8
```

```
$ export OMP_SCHEDULE="guided,4"
```



## Runtime Libraries Routines

OpenMP provides several user-callable function to control and query parallel environment.

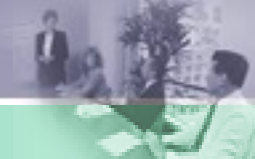
- The Runtime Libraries take precedence over the corresponding environment variables;
- Recommended to use under control of conditional compilation (`#ifdef _OPENMP`);
- C/C++ programs need to include `<omp.h>`;
- Fortran program may want to use `"USE OMP_LIB"` or *include "omp\_lib.h"*.



## Runtime Libraries Routines

- **omp\_set\_num\_threads(n):**  
Sets number of threads;
- **n=omp\_get\_num\_threads():**  
Gets number of threads in team;
- **omp\_set\_schedule(sched,chunk):**  
Sets schedule and chunk;
- **n=omp\_get\_thread\_num():**  
Gets thread ID;





# OpenMP Compiler

**GNU** (Version  $\geq 4.3.2$ ) Compile with ***-fopenmp***  
For Linux, Solaris, AIX, MacOSX, Windows:

**IBM** Compile with ***-qsmp=omp*** for Windows, AIX and Linux.

## **Sun Microsystems**

Compile with ***-xopenmp*** for Solaris and Linux.

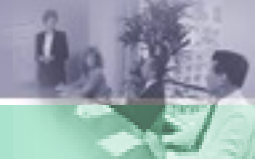
## **Intel**

Compile with ***-Qopenmp*** on Windows, or just ***-openmp*** on Linux or Mac

## **Portland Group Compilers**

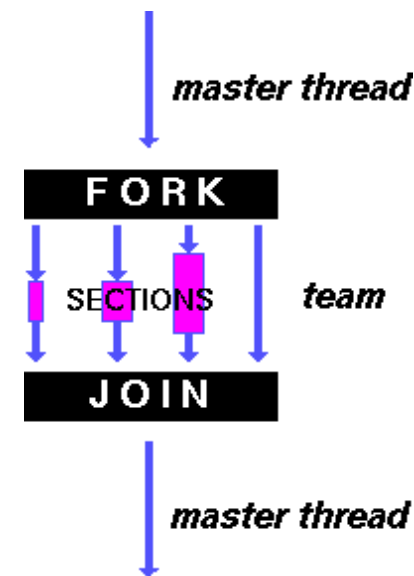
Compile with ***-mp***

Emit useful information to stderr. ***-Minfo=mp***



## Workshare: Sections construct

The sections construct is a noniterative worksharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team.





# Workshare: Sections construct

Fortran:

**!\$OMP PARALLEL**

...

**!\$OMP SECTIONS**

**!\$OMP SECTION**

*call subr\_A(c,d)*

**!\$OMP SECTION**

*call subr\_B(e,f)*

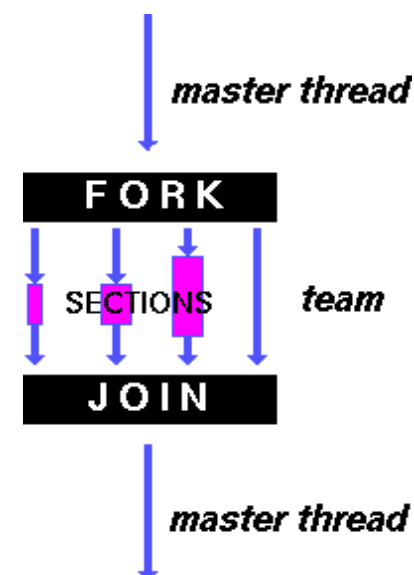
**!\$OMP SECTION**

*call subr\_c(g,h,i)*

**!\$OMP END SECTIONS**

...

**!\$OMP END PARALLEL**





# Workshare: Sections construct

C/C++:

```
#pragma omp parallel
```

```
{
```

```
...
```

```
#pragma omp sections
```

```
{
```

```
#pragma omp section
```

```
 $A = \text{subr\_}A(c, d)$ 
```

```
#pragma omp section
```

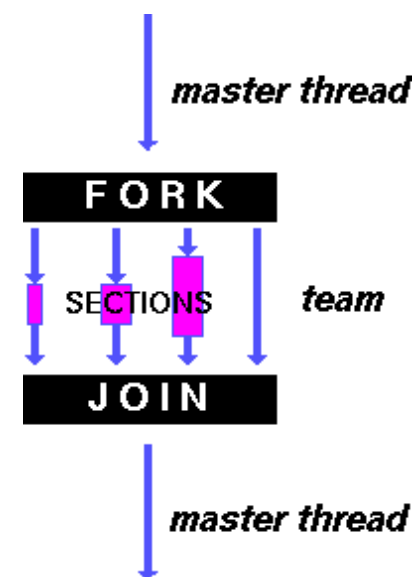
```
 $B = \text{subr\_}B(e, f)$ 
```

```
#pragma omp section
```

```
 $C = \text{subr\_}c(g, h, i)$ 
```

```
}
```

```
...
```

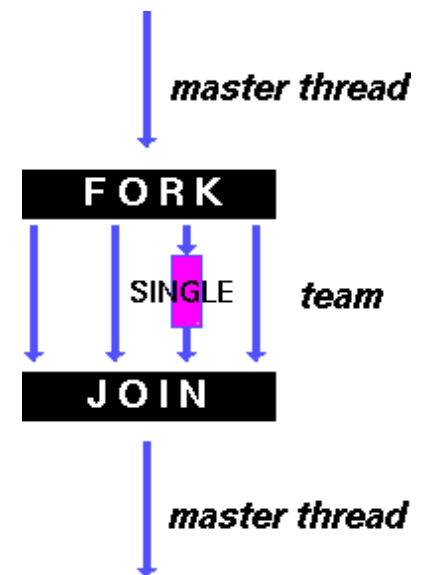




## Workshare: Single construct

The single construct specifies that the associated structured block is executed by only one of the threads in the team (not necessarily the master thread).

The other threads in the team, which do not execute the block, wait at an implicit barrier at the end of the single construct unless a nowait clause is specified.





# Workshare: Single construct

Fortran:

**!\$OMP PARALLEL**

...

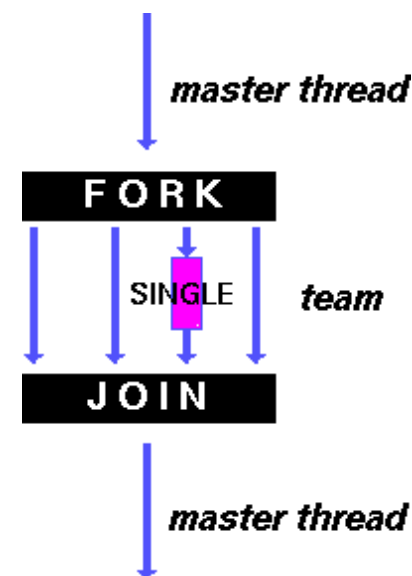
**!\$OMP SINGLE**

*read \*, a*

**!\$OMP END SINGLE**

...

**!\$OMP END PARALLEL**





# Workshare: Single construct

C/C++:

```
#pragma omp parallel
```

```
{
```

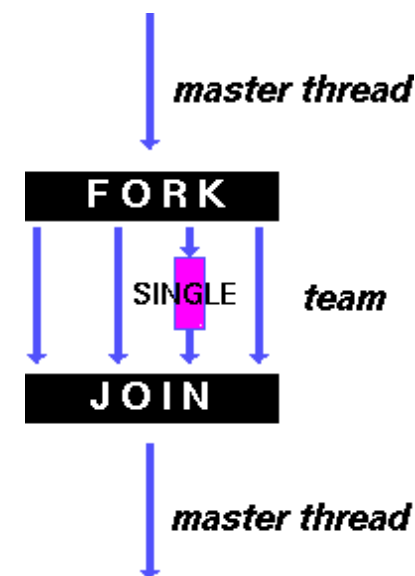
```
...
```

```
#pragma omp single
```

```
printf("Beginning work");
```

```
...
```

```
}
```





# Synchronization constructs

- Master;
- Critical;
- Atomic;
- Barrier;
- Ordered;





## Master construct

The master construct specifies a structured block that is executed by the master thread of the team.

There is no implied barrier either on entry to, or exit from, the master construct.

Fortran:

```
!$OMP PARALLEL
```

```
...
```

```
!$OMP MASTER
```

```
read *, a
```

```
!$OMP END MASTER
```

```
...
```

```
!$OMP END PARALLEL
```



# Master construct

C/C++:

```
#pragma omp parallel  
{  
...  
#pragma omp master  
printf("Beginning work");  
...  
}
```



## Critical construct

The critical construct restricts execution of the associated structured block to a single thread at a time. An optional name may be used to identify the critical construct.

Fortran:

```
!$OMP PARALLEL
```

```
...
```

```
!$OMP CRITICAL [NAME]
```

```
 $X = FUNC\_A(X)$ 
```

```
!$OMP END CRITICAL
```

```
...
```

```
!$OMP END PARALLEL
```



# Critical construct

C/C++:

```
#pragma omp parallel
```

```
{
```

```
...
```

```
#pragma omp critical [name]
```

```
x=subr_A(x)
```

```
...
```

```
}
```



## Barrier construct

The barrier construct specifies an explicit barrier at the point at which the construct appears.

Fortran:

```
!$OMP PARALLEL
```

```
...
```

```
X = FUNCTION_A(X)
```

```
!$OMP BARRIER
```

```
...
```

```
!$OMP END PARALLEL
```



## Barrier construct

The barrier construct specifies an explicit barrier at the point at which the construct appears.

C/C++:

```
#pragma omp parallel
{
...
x=subr_A(x)
#pragma omp barrier
...
}
```



## Atomic construct

The atomic construct ensures that a specific storage location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.

Fortran:

```
!$OMP PARALLEL
```

```
...
```

```
!$OMP ATOMIC
```

```
 $X = X + 1$ 
```

```
...
```

```
!$OMP END PARALLEL
```

## Atomic construct

The atomic construct ensures that a specific storage location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.

C/C++:

```
#pragma omp parallel  
{  
...  
#pragma omp atomic  
x++;  
...  
}
```





## Ordered construct

The ordered construct specifies a structured block in a loop region that will be executed in the order of the loop iterations. This sequentializes and orders the code within an ordered region while allowing code outside the region to run in parallel.



## Ordered construct

Fortran:

```
!$OMP PARALLEL
```

```
!$OMP DO ORDERED
```

```
DO i=1,N
```

```
  A(i)=...
```

```
!$OMP ORDERED
```

```
  PRINT *,a(i)
```

```
!$OMP END ORDERED
```

```
ENDDO
```

```
!$OMP END DO ORDERED
```

```
!$OMP END PARALLEL
```



## Ordered construct

C/C++:

```
#pragma omp parallel
{
    ...
    #pragma omp for ordered
    for (i=0;i<n;++i)
        a[i] = b[i] + 1.0
    #pragma omp ordered
    {
        printf("%f\n",a[i]);
    }
}
```



## Data-Sharing Attribute Clauses

- **Shared**: declares one or more list items to be shared by tasks generated by a parallel construct.
- **Private**: declares one or more list items to be private to a task.
- **Firstprivate**: declares one or more list items to be private to a task, and initializes each of them with the value that the corresponding original item has when the construct is encountered.



## Data-Sharing Attribute Clauses

**Lastprivate:** declares one or more list items to be private to an implicit task, and causes the corresponding original list item to be updated after the end of the region.

```
!$omp do lastprivate (i)
```

```
do i = 1,n-1
```

```
  a(i) = b(i+1)
```

```
enddo
```

```
!$omp end do
```

```
a(i) = b(0)
```



## Data-Sharing Attribute Clauses

**Reduction:** The reduction clause specifies an operator and one or more list items. For each list item, a private copy is created in each implicit task, and is initialized appropriately for the operator. After the end of the region, the original list item is updated with the values of the private copies using the specified operator.

```
!$omp do reduction (+:x)
```

```
do i = 1,n
```

```
  x = x + a(i)
```

```
enddo
```

```
!$omp end do
```

Support for most arithmetic and logical operators

+, \*, -, .MIN., .MAX., .AND., .OR., ...