Master in High Performance Computing

# Shared Memory Programming Paradigm

**Ivan Girotto – igirotto@ictp.it**

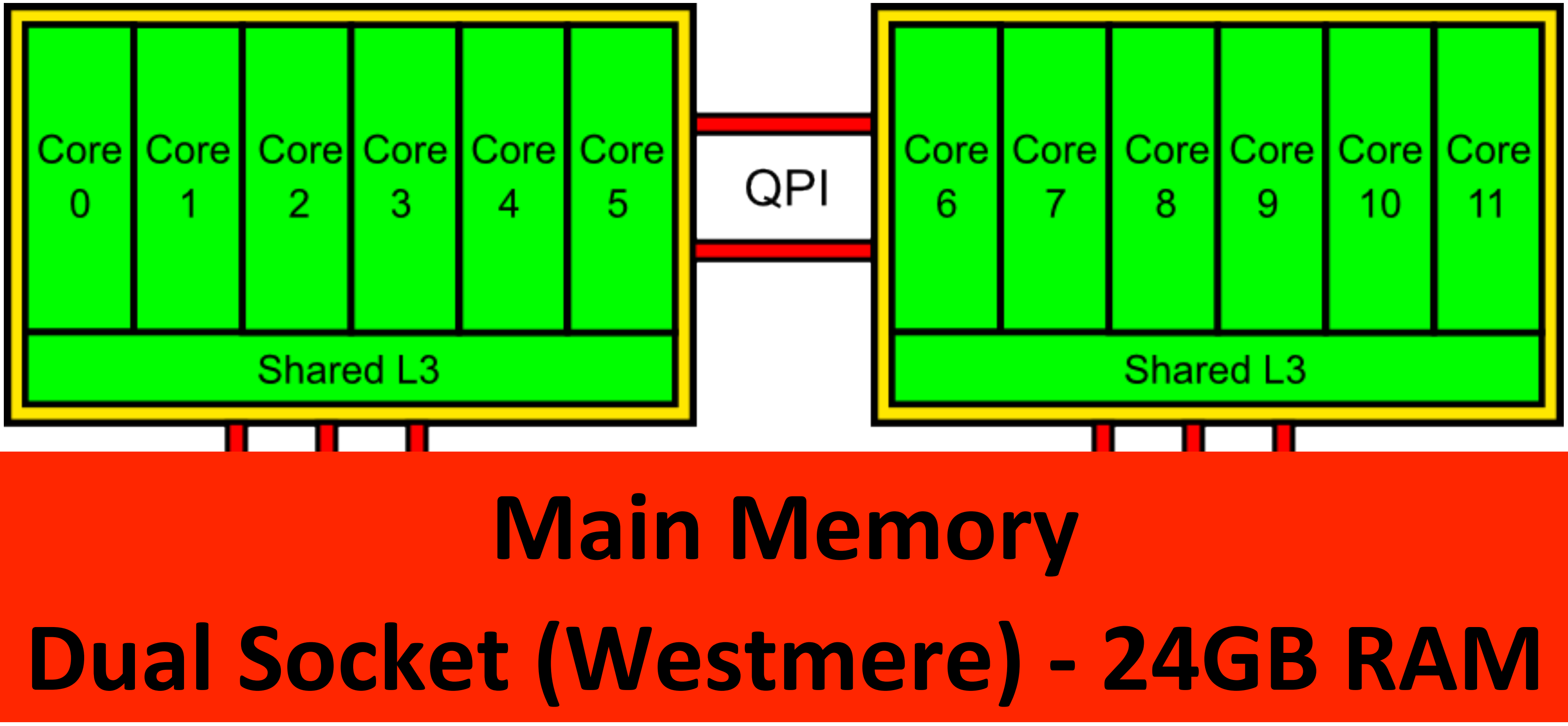
Information & Communication Technology Section (ICTS)

International Centre for Theoretical Physics (ICTP)

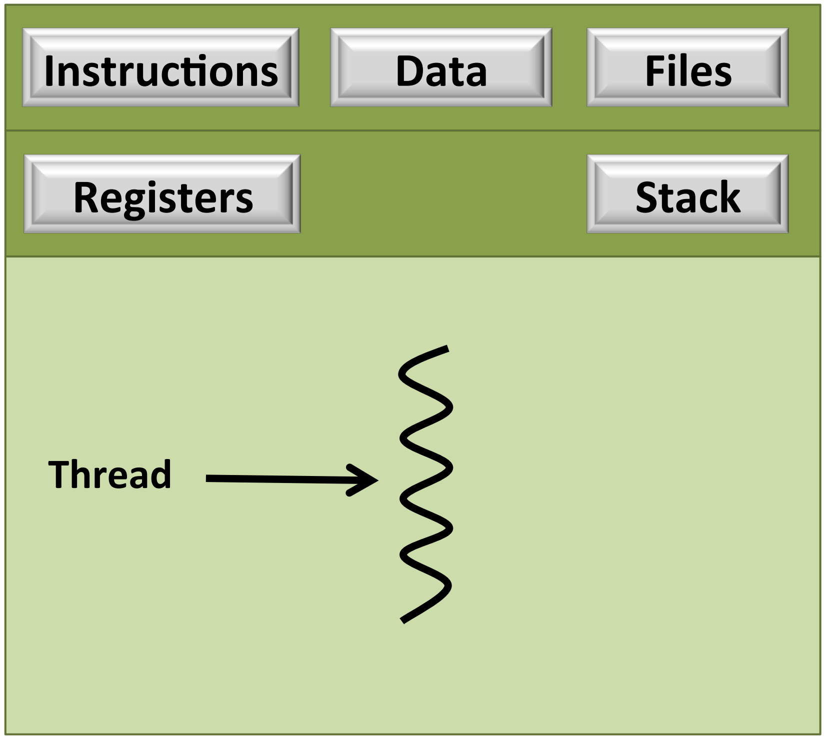
Scuola Internazionale Superiore di Studi Avanzati

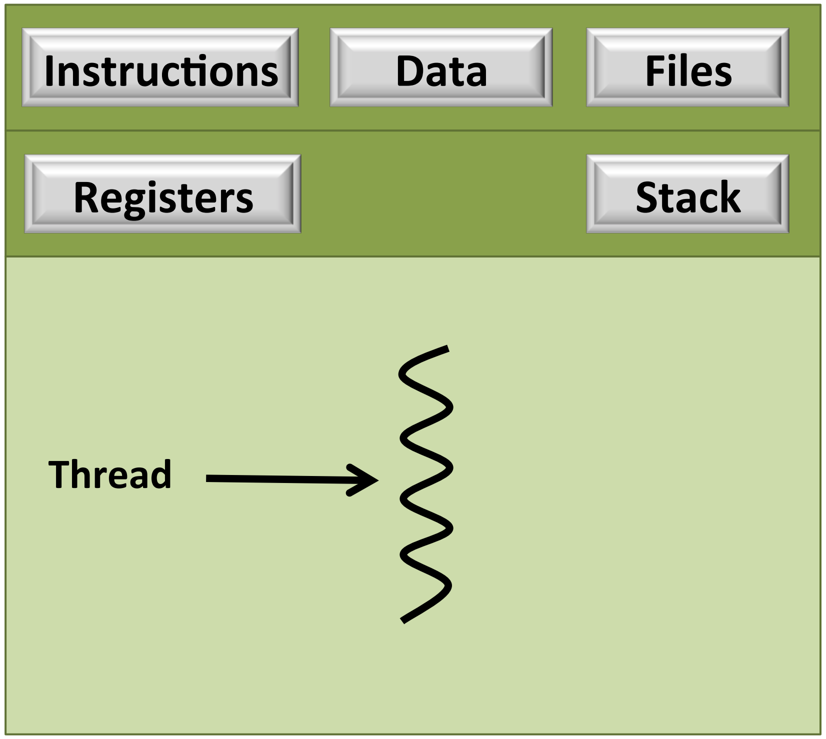
The Abdus Salam International Centre for Theoretical Physics

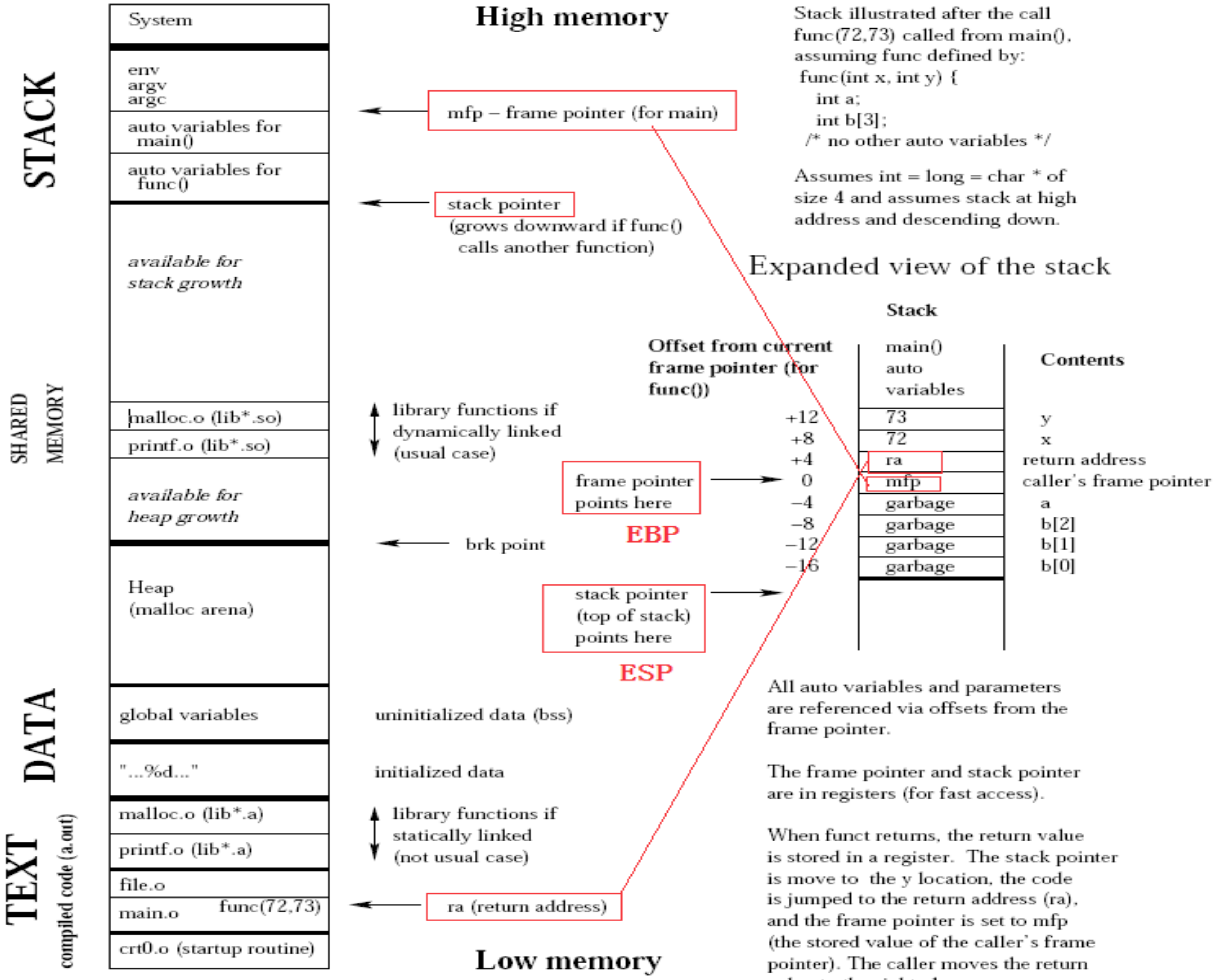
## Multi-CPUs & Multi-cores NUMA system



# Processes and Threads







Stack illustrated after the call func(72,73) called from main(), assuming func defined by:

```
 func(int x, int y) {
    int a;
    int b[3];
  /* no other auto variables */
```

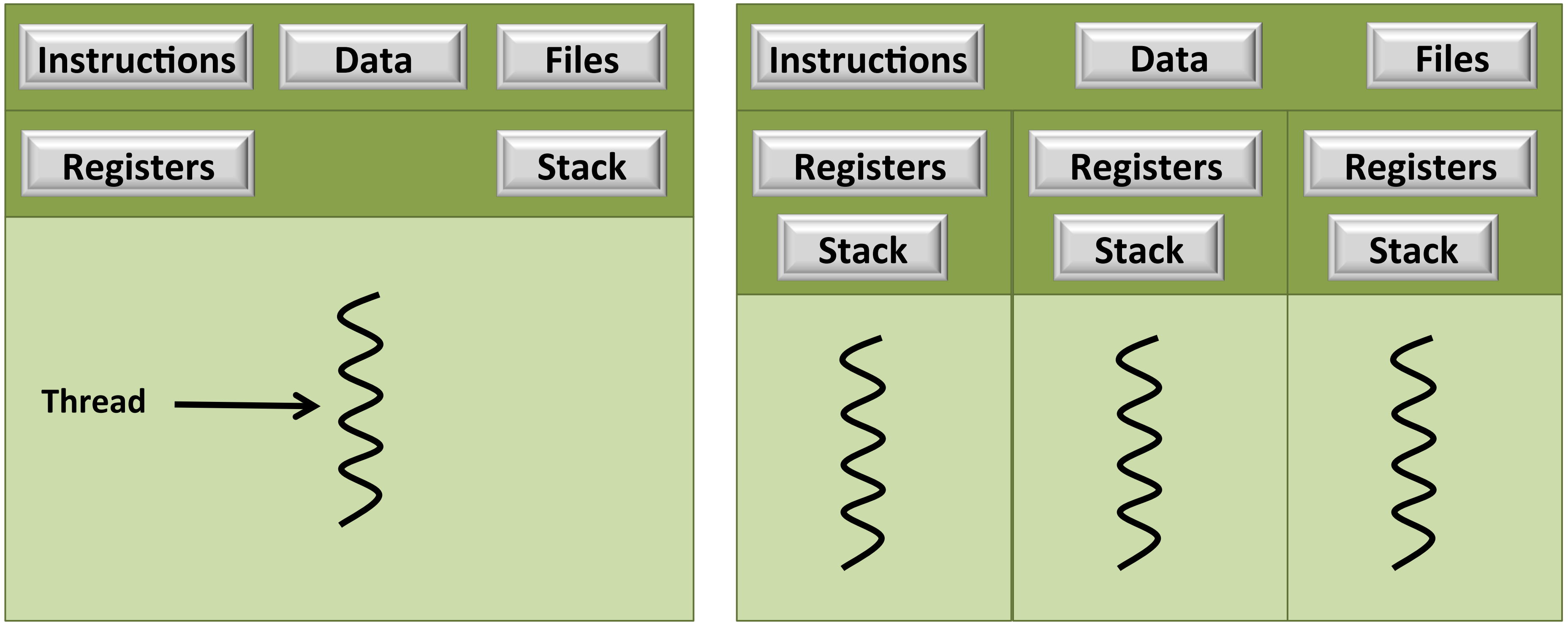
Assumes int = long = char * of size 4 and assumes stack at high address and descending down.

All auto variables and parameters are referenced via offsets from the frame pointer.

The frame pointer and stack pointer are in registers (for fast access).

When funct returns, the return value is stored in a register. The stack pointer is move to the y location, the code is jumped to the return address (ra), and the frame pointer is set to mfp (the stored value of the caller's frame pointer). The caller moves the return value to the right place.

# Processes and Threads



## Multi-threading - Recap

- A thread is a (lightweight) process - an instance of a program plus its own data (private memory)
- Each thread can follow its own flow of control through a program
- Threads can share data with other threads, but also have private data
- Threads communicate with each other via the shared data.
- A *master thread* is responsible for co-ordinating the threads group

MHPC
Master in High Performance Computing

OpenMP™

ma per seguir virtute e canoscenza
SISSA
Scuola Internazionale Superiore
di Studi Avanzati

ICTP
The Abdus Salam
International Centre
for Theoretical Physics

## OpenMP (*Open spec. for Multi Processing*)

OpenMP is not a computer language

- Rather it works in conjunction with existing languages such as standard Fortran or C/C++

Application Programming Interface (API)

- that provides a portable model for parallel applications
- Three main components:
  - Compiler directives
  - Runtime library routines
  - Environment variables

# OpenMP Parallelization

OpenMP is directive based

- code (can) work without them

OpenMP can be added incrementally

OpenMP only works in shared memory

- multi-socket nodes, multi-core processors

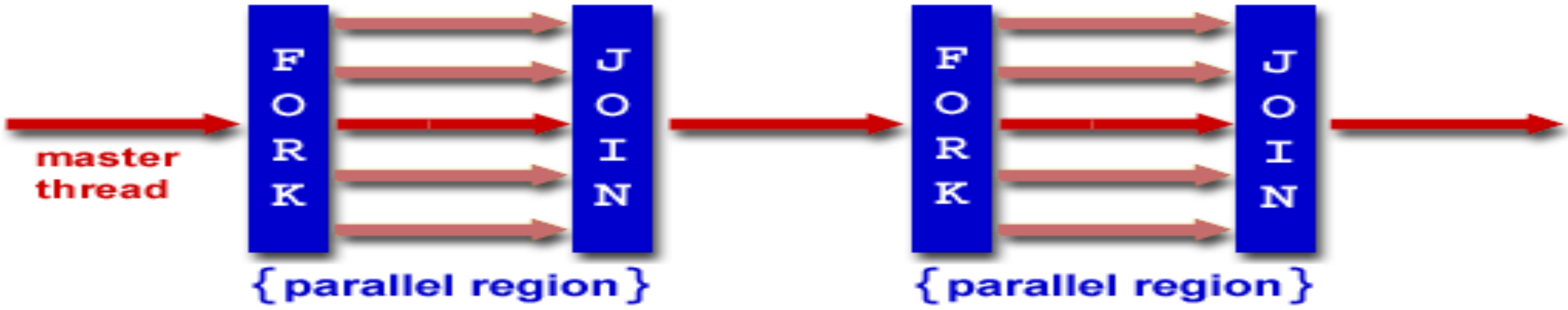
OpenMP hides the calls to a threads library

- less flexible, but much less programming

**Caution:** write access to shared data can easily lead to race conditions and incorrect data

Master in High Performance Computing

# OpenMP Parallelization



- Thread-based Parallelism
- Explicit Parallelism
- Fork-Join Model
- Compiler Directive Based
- Dynamic Threads

Scuola Internazionale Superiore di Studi Avanzati

The Abdus Salam International Centre for Theoretical Physics

Master in High Performance Computing

## Getting Started with OpenMP

OpenMP's constructs fall into 5 categories:

- Parallel Regions
- Work sharing
- Data Environment (scope)
- Synchronization
- Runtime functions/environment variables

OpenMP is essentially the same for both Fortran and C/C++

SISSA
Scuola Internazionale Superiore di Studi Avanzati

The Abdus Salam International Centre for Theoretical Physics

# Directives Format

A directive is a special line of source code with meaning only to certain compilers.

A directive is distinguished by a sentinel at the start of the line.

OpenMP sentinels are:

- Fortran: **!$OMP** (or **C$OMP** or ***$OMP**)
- C/C++: **#pragma omp**

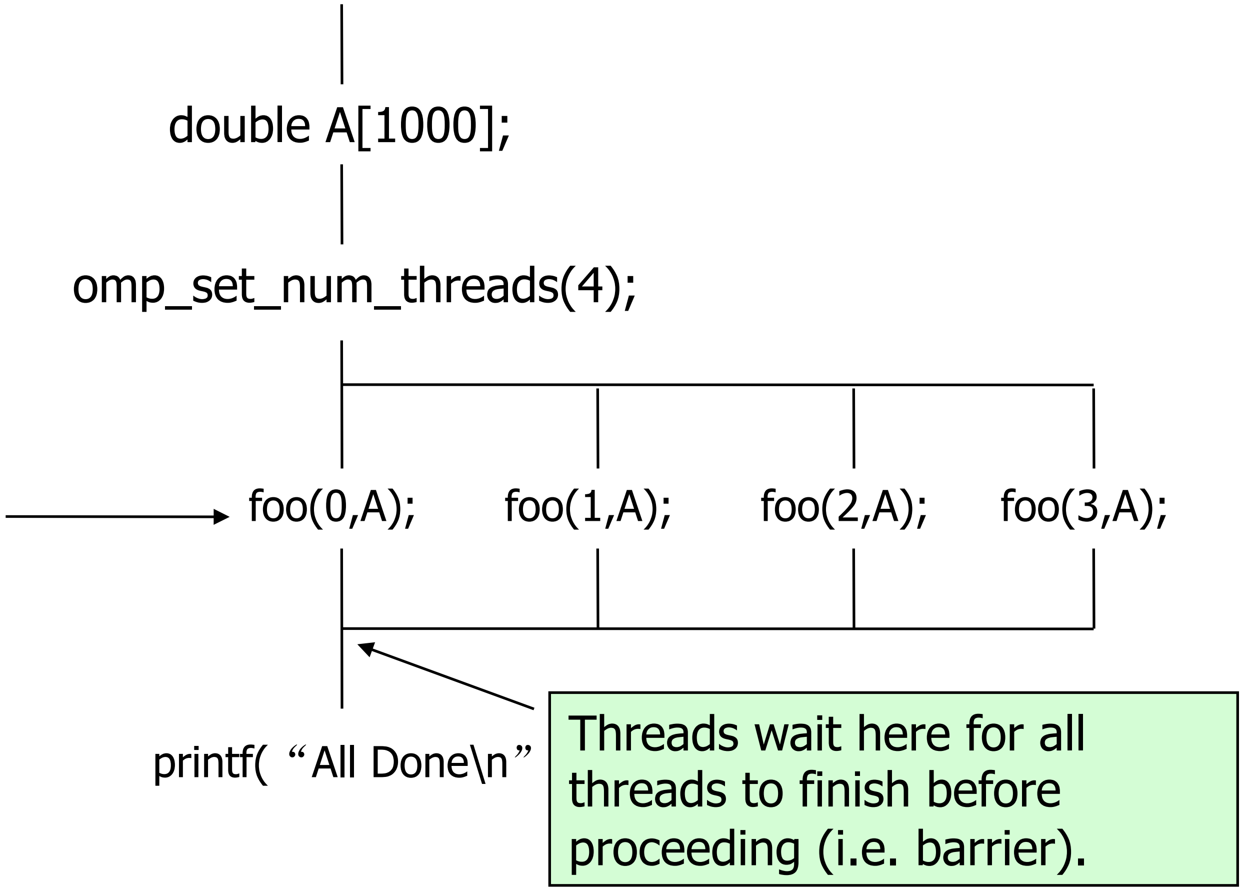
## OpenMP: Parallel Regions

For example, to create a 4-thread parallel region:
each thread calls foo(ID,A) for **ID** = **0** to **3**

Each thread redundantly executes the code within the structured block

thread-safe routine: A routine that performs the intended function even when executed concurrently (by more than one thread)

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
   int ID =omp_get_thread_num();
   foo(ID,A);
}
printf( “All Done\n” );
```


double A[1000];
omp_set_num_threads(4);
A single copy of A is shared between all threads.
foo(0,A);
foo(1,A);
foo(2,A);
foo(3,A);
printf( "All Done\n"
Threads wait here for all threads to finish before proceeding (i.e. barrier).

## How many threads?

- The number of threads in a parallel region is determined by the following factors:
  - Use of the omp_set_num_threads() library function
  - Setting of the OMP_NUM_THREADS environment variable
  - The implementation default
  - Threads are numbered from 0 (master thread) to N-1.

# Compiling OpenMP

```
gcc -fopenmp -c my_openmp.c
gcc -fopenmp -o my_openmp.x my_openmp.o

icc -openmp -c my_openmp.c
icc -openmp -o my_openmp.x my_openmp.o
```

## OpenMP runtime library

OMP_GET_NUM_THREADS() – returns the current # of threads.

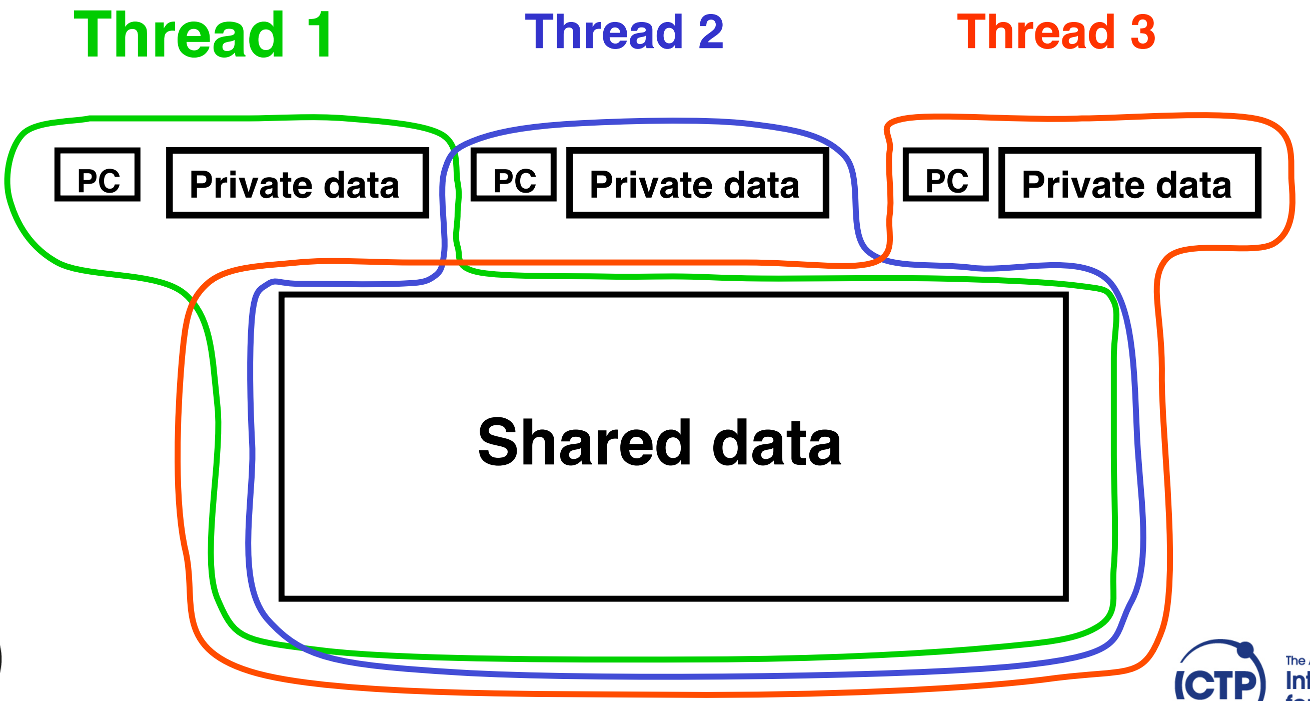
OMP_GET_THREAD_NUM() - returns the id of this thread.

OMP_SET_NUM_THREADS(n) – set the desired # of threads.

OMP_IN_PARALLEL() – returns .true. if inside parallel region.

OMP_GET_MAX_THREADS() - returns the # of possible threads.

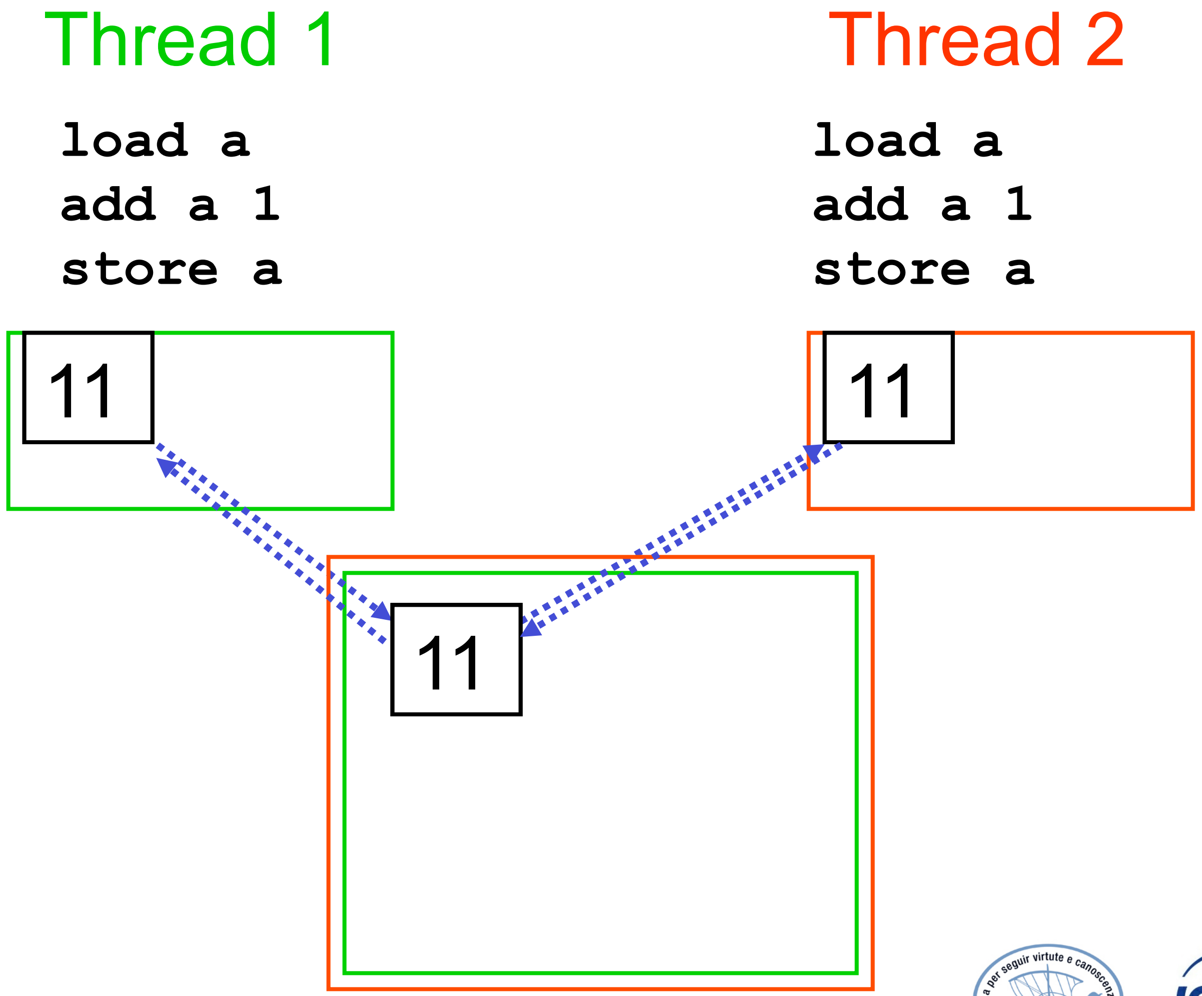
## Memory footprint



Program

Private data

Shared data



## Simple C OpenMP Program

```
#include <omp.h>
#include <stdio.h>

int main ( ) {

   printf("Starting off in the sequential world.\n");
   #pragma omp parallel
   {
      printf("Hello from thread number %d\n", omp_get_thread_num() );
   }
   printf("Back to the sequential world.\n");
   return 0;
}
```

```
PROGRAM HELLO
    INTEGER NTHREADS, TID, OMP_GET_NUM_THREADS
    INTEGER OMP_GET_THREAD_NUM
!!Fork a team of threads giving them their own copies of variables
!$OMP PARALLEL PRIVATE(NTHREADS, TID)
!!Obtain thread number
    TID = OMP_GET_THREAD_NUM()
    PRINT *, 'Hello World from thread = ', TID

!!Only master thread does this
    IF (TID .EQ. 0) THEN
        NTHREADS = OMP_GET_NUM_THREADS()
        PRINT *, 'Number of threads = ', NTHREADS
    END IF

!!All threads join master thread and disband
!$OMP END PARALLEL
END PROGRAM
```

Master in High Performance Computing

# Variable Scooping

All existing variable still exist inside a parallel region

- by default SHARED between all threads

But work sharing requires private variables

- PRIVATE clause to OMP PARALLEL directive
- Index variable of a worksharing loop
- All declared local variable within a parallel region
- The FIRSTPRIVATE clause would initialize the private instances with the contents of the shared instance

Be aware of the sharing nature of static variables

SISSA
Scuola Internazionale Superiore di Studi Avanzati

The Abdus Salam International Centre for Theoretical Physics

## Exploiting Loop Level Parallelism

Loop level Parallelism: parallelize only loops

Easy to implement

Highly readable code

Less than optimal performance (sometimes)

Most often used

## Parallel Loop Directives

Fortran do loop directive

- !$omp do

C\C++ for loop directive

- #pragma omp for

These directives do not create a team of threads but assume there has already been a team forked.

If not inside a parallel region shortcuts can be used.

- !$omp parallel do
- #pragma omp parallel for

Master in High Performance Computing

## Parallel Loop Directives /2

These are equivalent to a parallel construct followed immediately by a worksharing construct.

```
!$omp parallel do

        Same as

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

```
#pragma omp parallel for

        Same as

#pragma omp parallel
...
#pragma omp for
```

SISSA
Scuola Internazionale Superiore di Studi Avanzati

ICTP
The Abdus Salam International Centre for Theoretical Physics

```
integer :: N, start, len, numth, tid, i, end
double precision, dimension (N) :: a, b, c
!$OMP PARALLEL PRIVATE (start, end, len, numth, tid, i)
  numth  = omp_get_num_threads()
  tid = omp_get_thread_num()
  len = N / numth
  if( tid .lt. mod( N, numth ) ) then
     len = len + 1
     start = len * tid + 1
  else
     start = len * tid + mod( N, numth ) + 1
  endif
  end =  start +  len - 1
  do i = start, end
    a(i) = b(i) + c(i)
  end do
!OMP END PARALLEL
```

**Not the intended mode for OpenMP**

## How is OpenMP Typically Used?

OpenMP is usually used to parallelize loops:

**Split-up this loop between multiple threads**

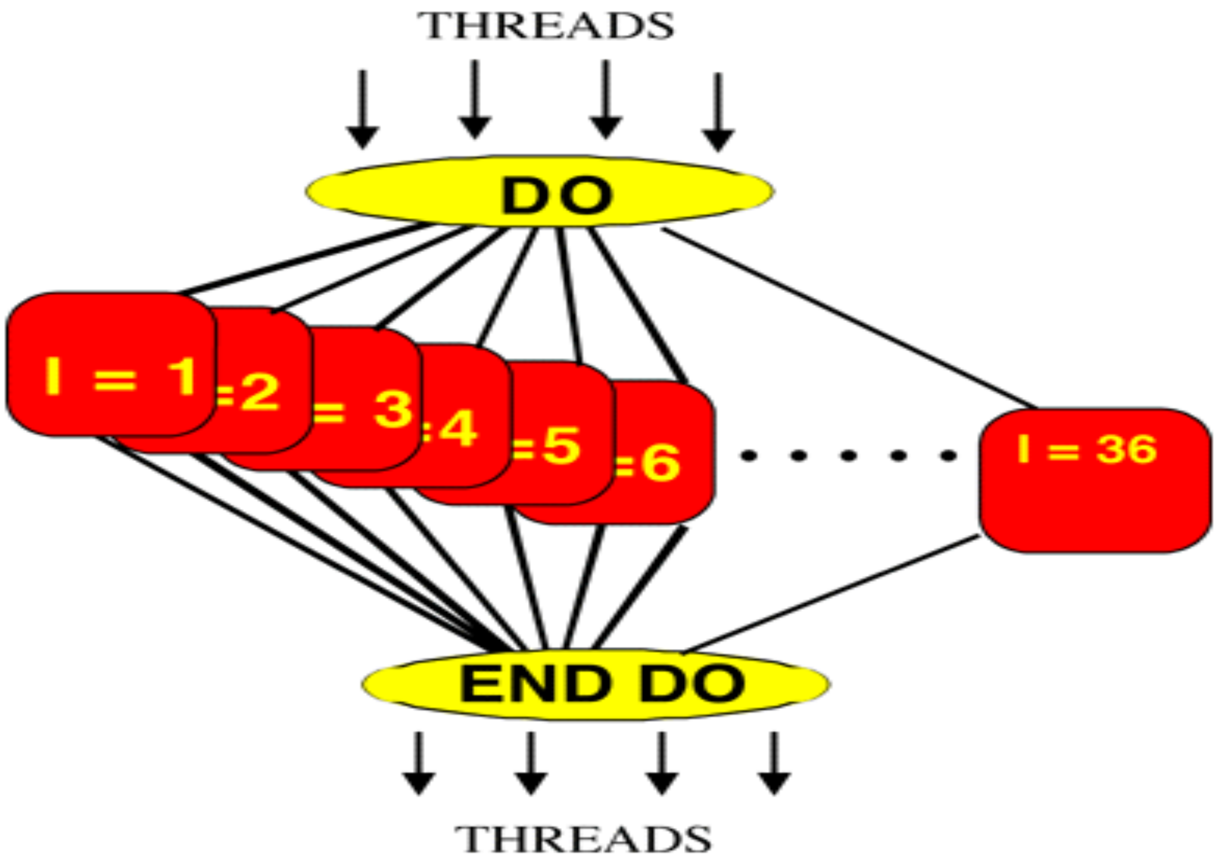
```
void main()
{
   double Res[1000];

   for(int i=0;i<1000;i++) {
      do_huge_comp(Res[i]);
   }
}
```

Sequential program

```
void main()
{
   double Res[1000];
   #pragma omp parallel for
   for(int i=0;i<1000;i++) {
      do_huge_comp(Res[i]);
   }
}
```

Parallel program

Master in High Performance Computing



Scuola Internazionale Superiore di Studi Avanzati

The Abdus Salam International Centre for Theoretical Physics

## Work-Sharing Constructs

Divides the execution of the enclosed code region among the members of the team that encounter it.

Work-sharing constructs **do not** launch new threads.

No implied barrier upon entry to a work sharing construct.

However, there is an implied barrier at the end of the work sharing construct (unless nowait is used).

## Work Sharing Constructs - example

**Sequential code**

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

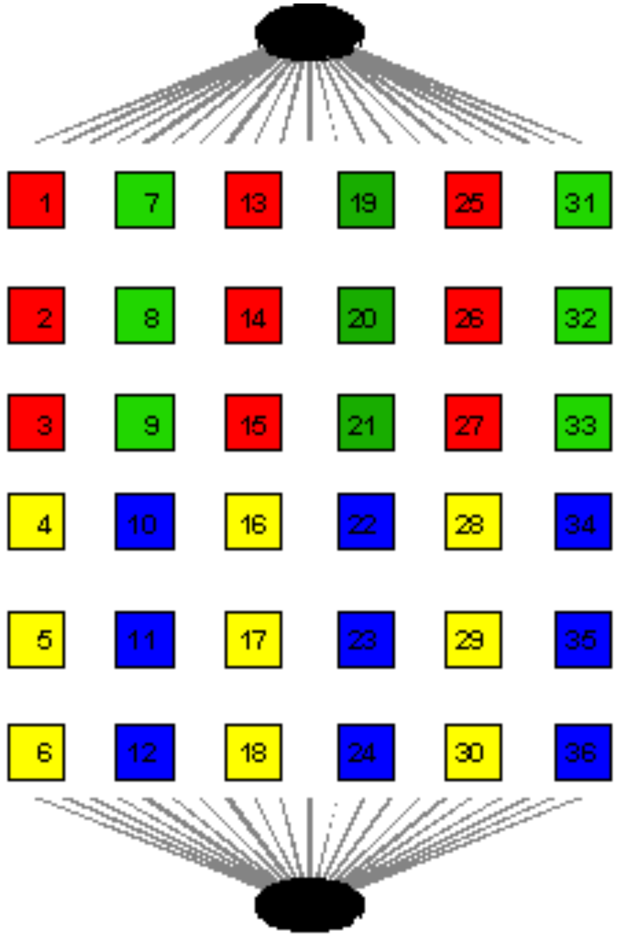
**OpenMP // Region**

```
#pragma omp parallel
{
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    for(i=istart;I<iend;i++) {a[i]=a[i]+b[i];}
}
```

**OpenMP Parallel Region and a work-sharing for construct**

```
#pragma omp parallel
#pragma omp for schedule(static)
for(i=0;I<N;i++) { a[i]=a[i]+b[i];}
```

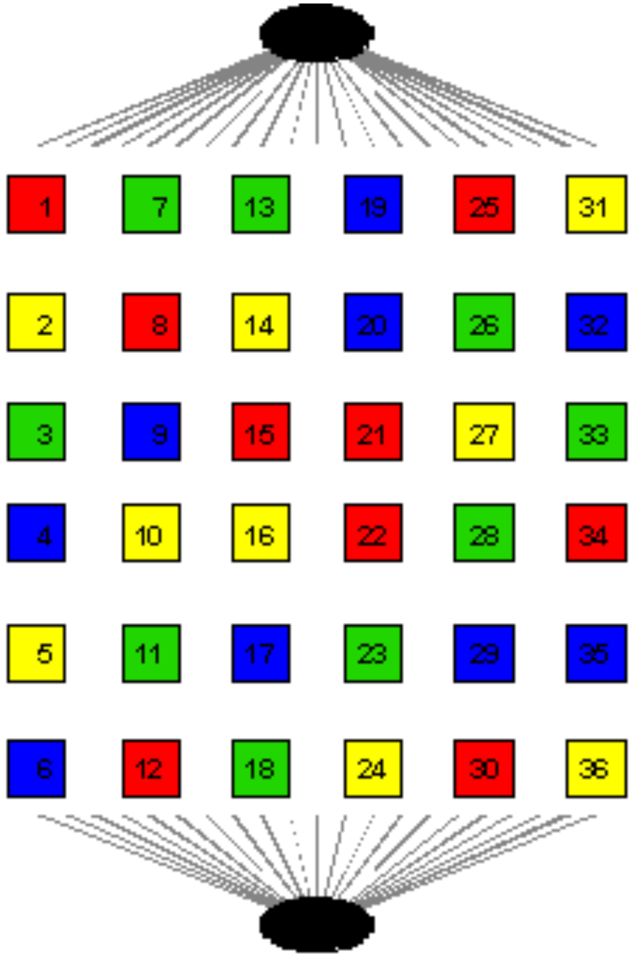
## schedule(static [,chunk])



- Iterations are divided evenly among threads
- If chunk is specified, divides the work into chunk sized parcels
- If there are N threads, each thread does every Nth chunk of work.

```
!$OMP PARALLEL DO &
!$OMP SCHEDULE(STATIC,3)

DO J = 1, 36
         Work (j)
END DO

!$OMP END DO
```

## schedule(dynamic [,chunk])



- Divides the workload into chunk sized parcels.
- As a thread finishes one chunk, it grabs the next available chunk.
- Default value for chunk is one.
- More overhead, but potentially better load balancing.

```
!$OMP PARALLEL DO & !
$OMPSCHEDULE(DYNAMIC,1)

DO J = 1, 36
            Work (j)
END DO

!$OMP END DO
```

## The Schedule Clause SCHEDULE (type [,chunk])

The schedule clause effects how loop iterations are mapped onto threads

schedule(static [,chunk])

- Deal-out blocks of iterations of size “chunk” to each thread

schedule(dynamic [,chunk])

- Each thread grabs “chunk” iterations off a queue until all iterations have been handled

schedule(guided [,chunk])

- Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds

schedule(runtime)

- Schedule and chunk size taken from the OMP_SCHEDULE environment variable

## No Wait Clauses

- No wait: if specified then threads do not synchronise at the end of the parallel loop.

For Fortran, the END DO directive is optional with NO WAIT being the default.

Note that the nowait clause is incompatible with a simple parallel region meaning that using the composite directives will not allow you to use the nowait clause.

## OpenMP: Reduction(op : list)

The variables in “list” must be shared in the enclosing parallel region.

Inside a parallel or a worksharing construct:

- A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”)
- pair wise “op” is updated on the local value
- Local copies are reduced into a single global copy at the end of the construct.

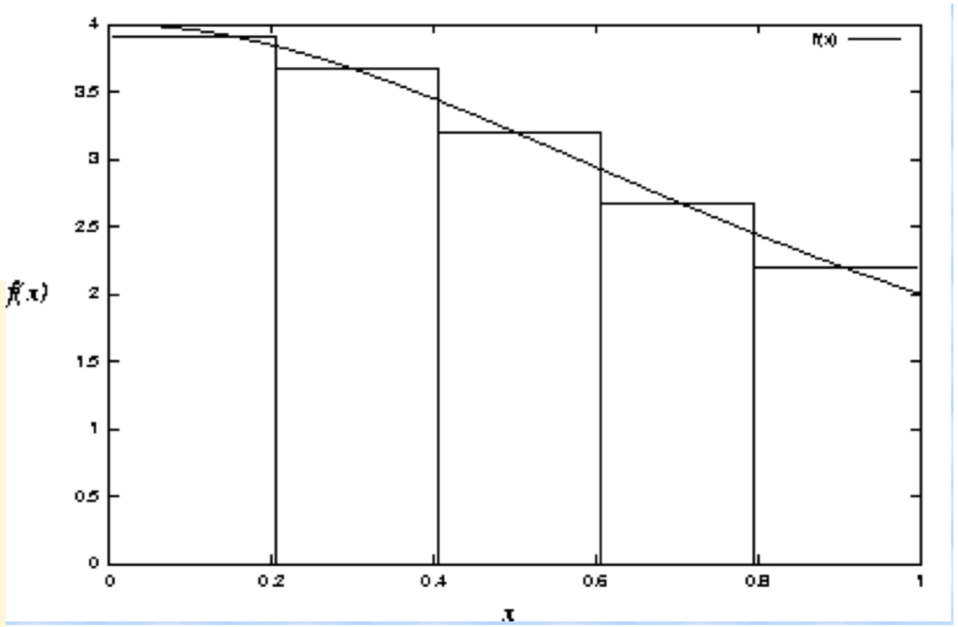
## OpenMP: A Reduction Example

```
#include <omp.h>
#define NUM_THREADS 2
void main ()
{
      int i;
      double ZZ, func(), sum=0.0;

      omp_set_num_threads(NUM_THREADS);
      #pragma omp parallel for reduction(+:sum) private(ZZ)
      for (i=0; i< 1000; i++){
            ZZ = func(i);
            sum = sum + ZZ;
      }
}
```

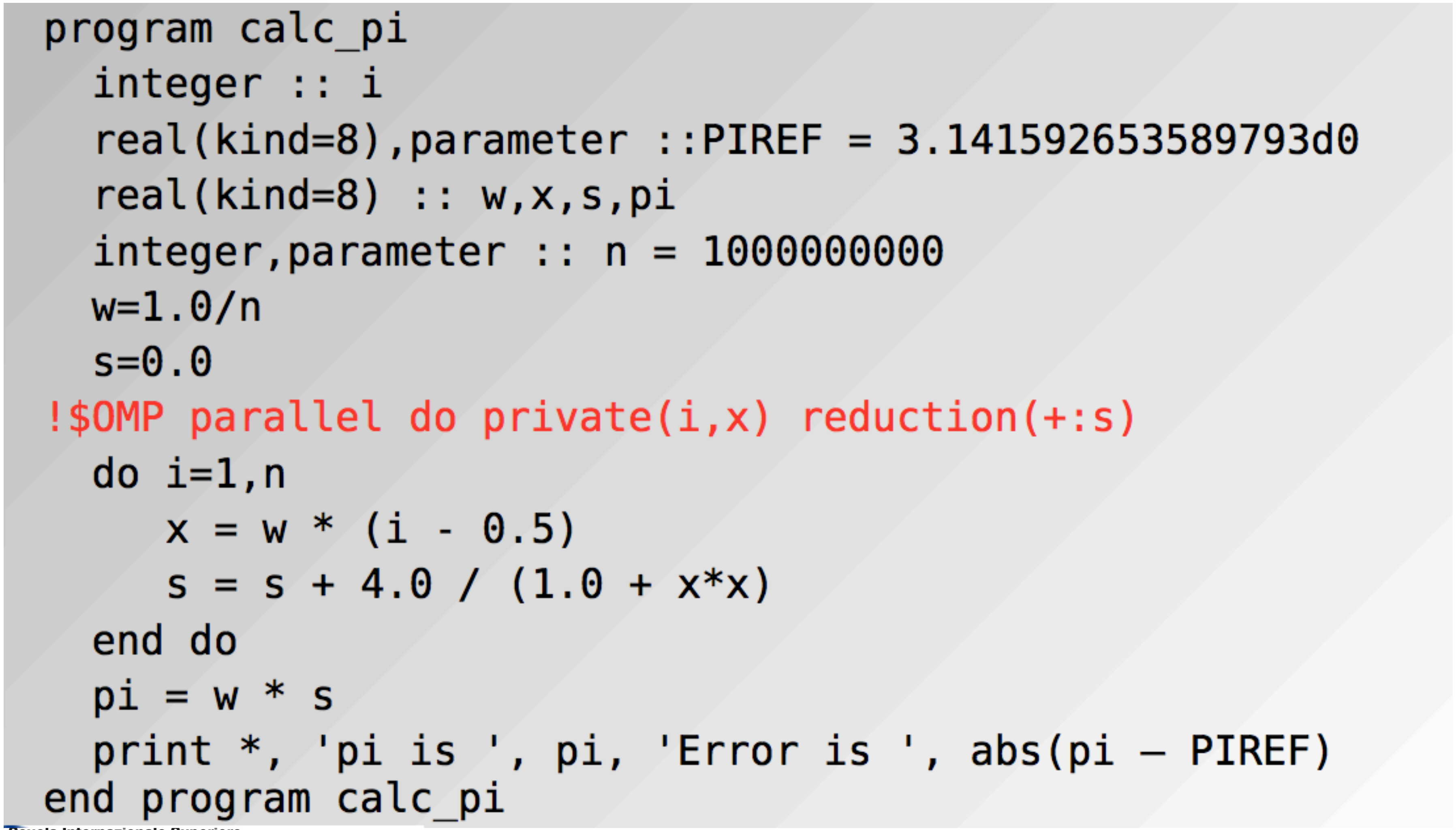
## Compute PI

$$\int_0^1 \frac{1}{1+x^2}dx = arctan(x)\Big|_0^1 = arctan(1) - arctan(0) = arctan(1) = \frac{\pi}{4}$$

$$\pi = 4\int_0^1 \frac{1}{1+x^2}dx$$



Integrate, i.e determine area under function numerically using slices of h * f(x) at midpoints



```
program calc_pi
  integer :: i
  real(kind=8),parameter ::PIREF = 3.1415926535897932d0
  real(kind=8) :: w,x,s,pi
  integer,parameter :: n = 1000000000
  w=1.0/n
  s=0.0
!$OMP parallel do private(i,x) reduction(+:s)
  do i=1,n
     x = w * (i - 0.5)
     s = s + 4.0 / (1.0 + x*x)
  end do
  pi = w * s
  print *, 'pi is ', pi, 'Error is ', abs(pi – PIREF)
end program calc_pi
```

## if CLAUSE

We can make the parallel region directive itself conditional.

Fortran: IF (*scalar logical expression*)

C/C++: if (*scalar expression*)

```
#pragma omp parallel if (tasks > 1000)
{
  while(tasks > 0) donexttask();
}
```

Master in High Performance Computing

## SYNCHRONIZATION

Scuola Internazionale Superiore di Studi Avanzati

The Abdus Salam International Centre for Theoretical Physics

MHPC
Master in High Performance Computing

## OpenMP: How do Threads Interact?

OpenMP is a shared memory model.

- Threads communicate by sharing variables.

Unintended sharing of data can lead to race conditions:

- race condition: when the program’s outcome changes as the threads are scheduled differently.

To control race conditions:

- Use synchronization to protect data conflicts.

Synchronization is expensive so:

- Change how data is stored to minimize the need for synchronization.

SISSA
Scuola Internazionale Superiore di Studi Avanzati

The Abdus Salam International Centre for Theoretical Physics

Note that updates to shared variables:

(e.g. a = a + 1)

are *not* atomic!

If two threads try to do this at the same time, one of the updates may get overwritten.

Thread 1

Thread 2

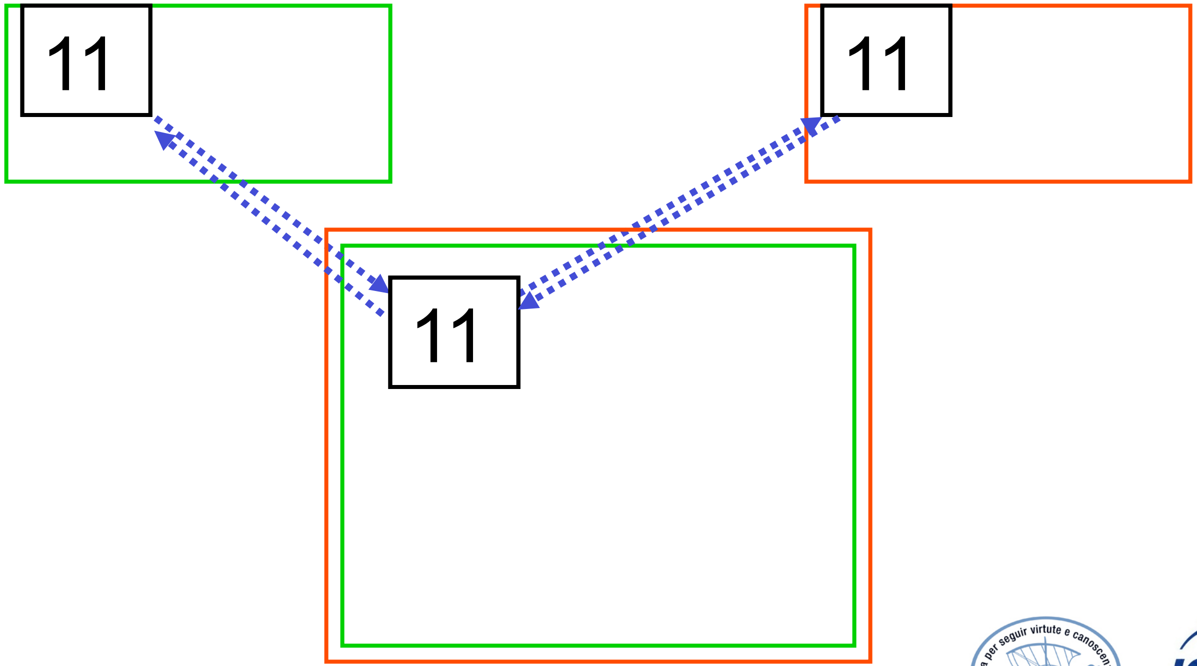
Program

```
load a
add a 1
store a
```

```
load a
add a 1
store a
```

Private data

Shared data



Master in High Performance Computing

## Barrier

| | |
|---|---|
| Fortran | - !$OMP BARRIER |
| C\C++ | - #pragma omp barrier |

This directive synchronises the threads in a team by causing them to wait until all of the other threads have reached this point in the code.

Implicit barriers exist after work sharing constructs. The nowait clause can be used to prevent this behaviour.

Add a note about single/master

SISSA
Scuola Internazionale Superiore di Studi Avanzati

ICTP The Abdus Salam International Centre for Theoretical Physics

## Critical

Only one thread at a time can enter a critical section.

Example: pushing and popping a task stack

```
!$OMP PARALLEL SHARED(STACK),PRIVATE(INEXT,INEW)
     ...
!$OMP CRITICAL (STACKPROT)
     inext = getnext(stack)
!$OMP END CRITICAL (STACKPROT)
     call work(inext,inew)
!$OMP CRITICAL (STACKPROT)
     if (inew .gt. 0) call putnew(inew,stack)
!$OMP END CRITICAL (STACKPROT)
     ...
!$OMP END PARALLEL
```

Master in High Performance Computing

## Atomic

Atomic is a special case of a critical section that can be used for certain simple statements

Fortran: `!$OMP ATOMIC`

*statement*

where *statement* must have one of these forms:

*x* = *x op expr*, *x* = *expr op x*, *x* = *intr* (*x*, *expr*) or

*x* = *intr* (*expr*, *x*)

*op* is one of `+`, `*`, `-`, `/`, `.and.`, `.or.`, `.eqv.`, or `.neqv.`

*intr* is one of `MAX`, `MIN`, `IAND`, `IOR` or `IEOR`

SISSA
Scuola Internazionale Superiore di Studi Avanzati

The Abdus Salam International Centre for Theoretical Physics

# Non Parallelizzabile

Show an example of Instruction dependency

Master in High Performance Computing

# OpenMP Tasking

- Useful to deal with unbalanced problem
- Linked lists is a good example
- Mostly applied for functional parallelism

```
#pragma omp task
```

```
!$omp task
```

Defines a task

Scuola Internazionale Superiore di Studi Avanzati

The Abdus Salam International Centre for Theoretical Physics

## Runtime Environment

- When a thread encounters a task construct, a new task is generated
- The moment of execution of the task is up to the runtime system
- Execution can be either immediate or delayed
- Completion of a task can be enforced  through task synchronization

Master in High Performance Computing

## Task Synchronization

**#pragma omp barrier**

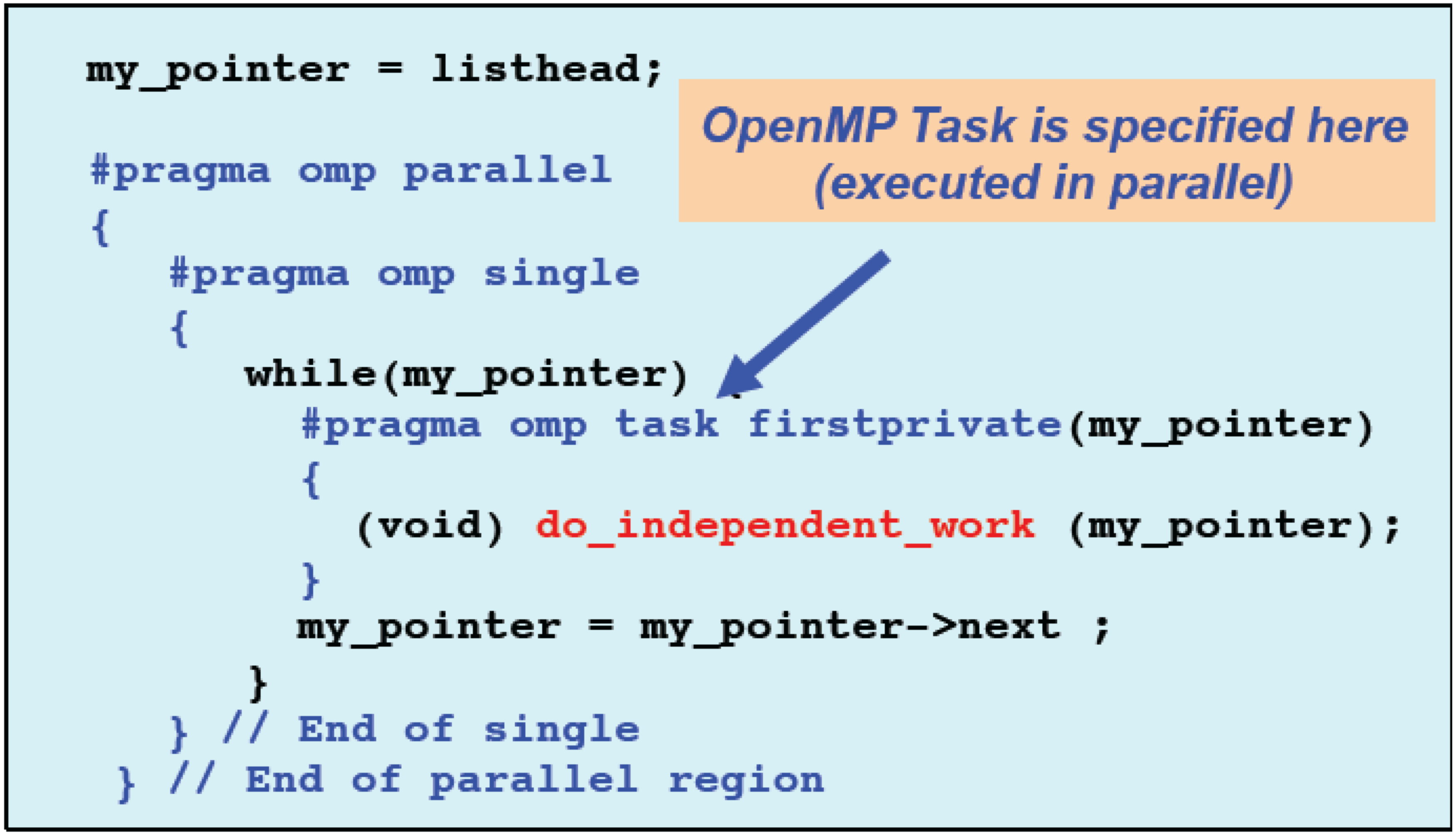
```
#pragma omp barrier
```

```
!$omp barrier
```

**#pragma omp taskwait**

```
#pragma omp taskwait
```

```
!$omp taskwait
```



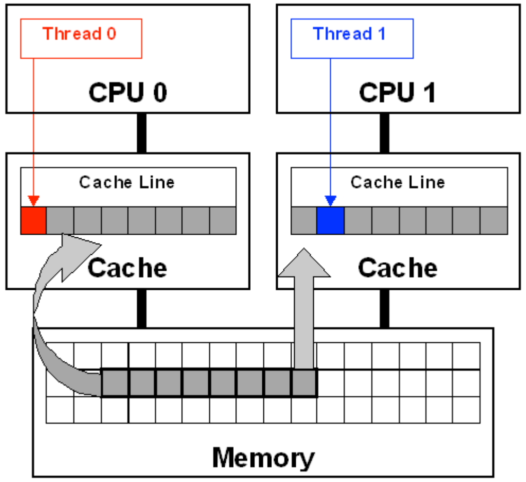
```
my_pointer = listhead;

#pragma omp parallel
{
   #pragma omp single
   {
      while(my_pointer) {
         #pragma omp task firstprivate(my_pointer)
         {
            (void) do_independent_work (my_pointer);
         }
         my_pointer = my_pointer->next ;
      }
   } // End of single
 } // End of parallel region
```

*OpenMP Task is specified here (executed in parallel)*

## Poor Performances

Often naive approaches multi-threaded programming results in poor performances

- Modern NUMA architecture requires specific attention, specially considering multithreaded-programming
- The overhead of thread-management is not always negligible
- Reduce to minimum the critical regions
- FALSE SHARING is behind the corner
- Anything shared is a possible source of race condition (if write access)



## Exercises

1. write an "hello-world" program that prints on std output how many threads are executed and the thread_ID for each thread
2. parallelize the heat equation code using OpenMP
3. parallelize the fast-transpose using OpenMP
4. parallelize the code provided in class using OpenMP

Note: perform performance analysis for the points 2-4. Write a Makefile that somehow allows to compile the serial and the OpenMP versions of the code. Instrument the code to print at runtime the number of threads, in case of a parallel version.