hpc.cineca.it

Introduction to OpenMP

Fabio Pitari - f.pitari@cineca.it
Massimiliano Guarrasi - m.guarrasi@cineca.it

Bologna - July 3, 2019

SuperComputing Application and Innovation

Introduction

Disadvantages of MPI

- Each MPI process can only access its local memory
  - ⇒ The data to be shared must be exchanged with explicit inter-process communications
  - ! It is in charge to the programmer to design and implement the data exchange between process (taking care of work balance)
- You can not adopt a strategy of incremental parallelization
  - ⇒ The communication structure of the entire program has to be implemented
- It is difficult to maintain a single version of the code for the serial and MPI program
  - ⇒ Additional variables are needed
  - ⇒ Need to add a lot of boilerplate code

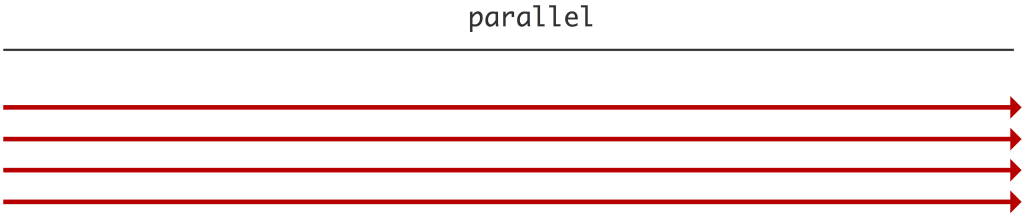
What is OpenMP?

- De-facto standard Application Program Interface (API) to write shared memory parallel applications in C, C++ and Fortran
- Consists of **compilers directives**, **run-time routines** and **environment variables**
- "Open specifications for Multi Processing" maintained by the OpenMP Architecture Review Board (http://www.openmp.org)

Founding concepts

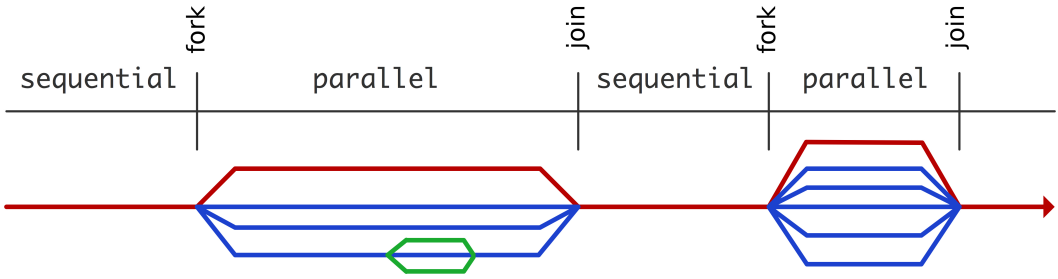
The "workers" who do the work in parallel (thread) "cooperate" through shared memory

- Memory accesses instead of explicit messages
- "local" model parallelization of the serial code
- It allows an incremental parallelization



MPI model

- Everything lies in a huge parallel region where the same code is executed by all the ranks
- Comunication among ranks has to be explicitely built when needed



Fork-Join model

Fork At the beginning of a parallel region the master thread creates a team of threads composed by itself and by a set of other threads

Join At the end of the parallel region the thread team ends the execution and only the master thread continues the execution of the (serial) program

A set of instructions can be executed on the whole set of threads using the *parallel* directive

C/C++

```
int main () {

    printf("Hello world\n");

  return 0;
}
```

A set of instructions can be executed on the whole set of threads using the *parallel* directive

C/C++

```
int main () {

  /* Serial part */

#pragma omp parallel
  {

    printf("Hello world\n");

  }

  /* Serial part */

  return 0;
}
```

A set of instructions can be executed on the whole set of threads using the *parallel* directive

Fortran

```
PROGRAM HELLO



  Print *, "Hello World!!!"



END PROGRAM HELLO
```

A set of instructions can be executed on the whole set of threads using the *parallel* directive

Fortran

```
PROGRAM HELLO

! Serial code

!$OMP PARALLEL

  Print *, "Hello World!!!"

!$OMP END PARALLEL

! Resume serial code


END PROGRAM HELLO
```

Compiler directives

```
// C++
#pragma omp <directive> [clause [clause] ...]
```

```
! Fortran
!$omp <directive> [clause [clause]...]
```

`directive` mark the block of code that should be made parallel

`clause` add information to the directives

⇒ Variables handling and scoping (shared, private, default)

⇒ Execution control (how many threads, work distribution...)

Note

GNU (gcc, g++, gfortran) `-fopenmp`

Intel (icc, icpc, ifort) `-qopenmp`

- **Compiler directives**
- **Environment variables**

`OMP_NUM_THREADS size`	set the number of threads
`OMP_DYNAMIC true|false`	set the number of threads automatically
`OMP_PLACES cores|num_places`	set the place where to allocate threads
`OMP_PROC_BIND true|false|close|spread`	bound threads to such place
`OMP_NESTED true|false`	allows nested parallelism

Other commonly used variables will be clearer later (listed here for reference):

`OMP_STACKSIZE size [B|K|M|G]`	size of the stack for threads
`OMP_SCHEDULE schedule[,chunk]`	iteration scheduling scheme

Note

```
$ export VARIABLE_NAME = value
```

- **Compiler directives**
- **Environment variables**
- **Run-time library**

`omp_get_thread_num()`	get thread ID
`omp_get_num_threads()`	get number of threads in the team for threads
`omp_set_num_threads(int n)`	set the number of threads
`omp_get_wtime()`	returns elapsed wallclock time
`omp_get_wtick()`	returns number of seconds between ticks

Note

```
#include <omp.h>  // C++

USE omp_lib       ! Fortran
```

Conditional compilation

C/C++

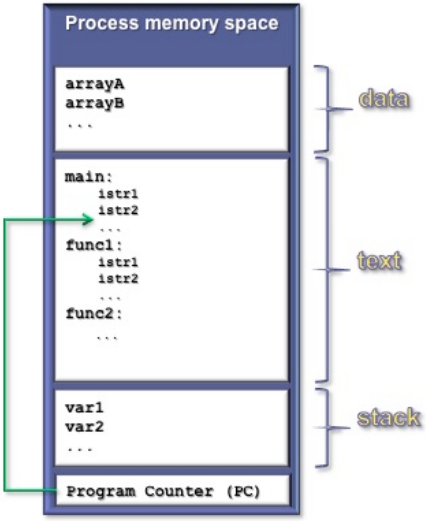
```
#ifdef _OPENMP
  printf("OpenMP support:%d",_OPENMP);
#else
  printf("Serial execution.");
#endif
```

Fortran

```
#ifdef _OPENMP
  print *,"OpenMP support"
#else
  print *,"Serial execution."
#endif
```

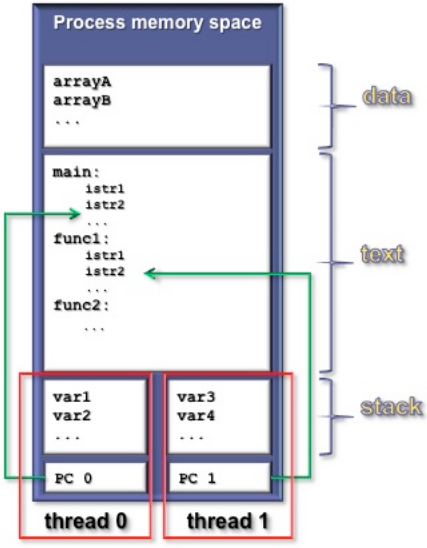
Note The macro `_OPENMP` has the value `yyyymm`, which contains the release date of the OpenMP version currently being used

- A process is an instance of a computer program
- Some information included in a process are:
  - Text
    - ⇒ Machine code
  - Data
    - ⇒ Global variables
  - Stack
    - ⇒ Local variables
  - Program counter (PC)
    - ⇒ A pointer to the instruction to be executed

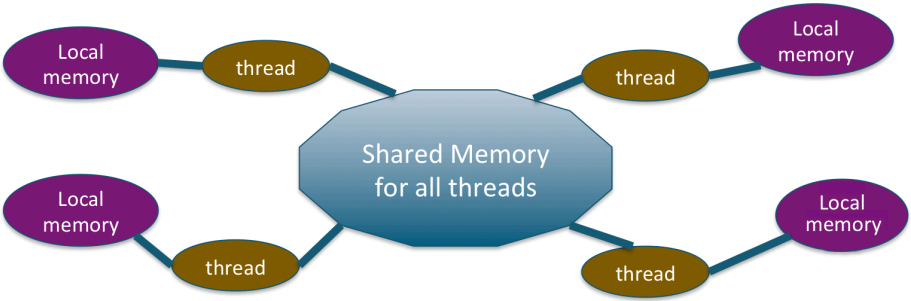


Multi-threaded processes

- The process contains several concurrent execution flows (threads)
  - Each thread has its own program counter (PC)
  - Each thread has its own private stack (variables local to the thread)
  - The instructions executed by a thread can access:
    - the process global memory (data)
    - the thread local stack



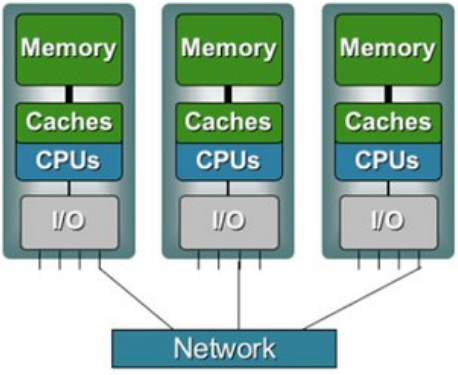
OpenMP memory model

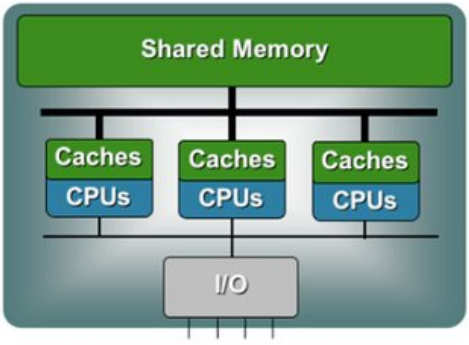


Shared-Local model

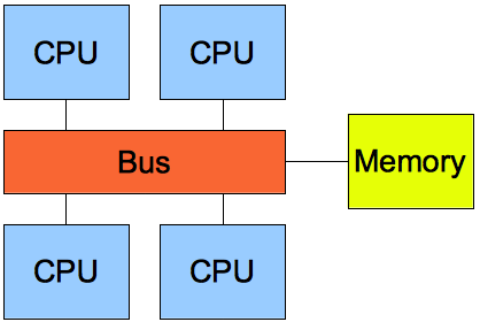
- each thread is allowed to have a temporary view of the shared memory
- each thread has access to a thread-private memory
- two kinds of data-sharing attributes: private and shared

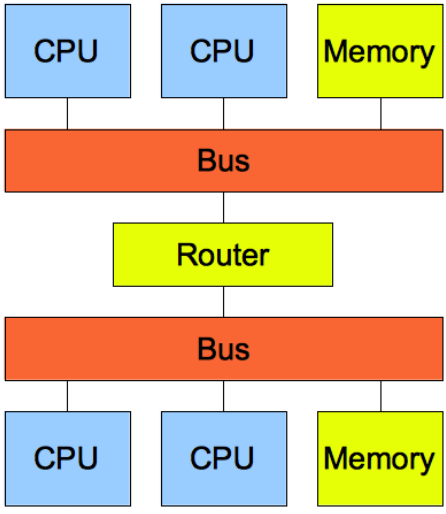
Distributed and shared memory





UMA and NUMA systems



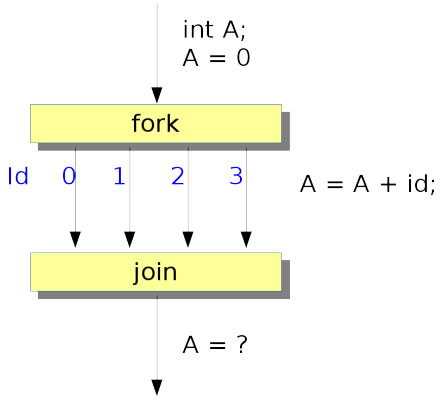


Defined or undefined?

What's the result?

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

void main(){
    int a;
    a = 0;
    #pragma omp parallel
    {
        // omp_get_thread_num
        // returns the id of the thread
        a = a + omp_get_thread_num();
    }
    printf("%d\n", a);
}
```



- A **race condition** (or data race) is when two or more threads access the same memory location:
  - asyncronously and,
  - without holding any common exclusive locks and,
  - at least one of the accesses is a **write/store**

- In this case the resulting values are **undefined**

The `critical` construct is a possible solution to **data races**:

- The block of code inside a critical construct is executed by only one thread at a time
- It locks the associated region

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

void main(){
    int a;
    a = 0;
    #pragma omp parallel
    {
        // omp_get_thread_num
        // returns the id of the thread
        #pragma omp critical
        a = a + omp_get_thread_num();
    }
    printf("%d\n", a);
}
```

Exercise 1.1

```
#include <stdio.h>
#ifdef _OPENMP
#include<omp.h>
#endif

int main(int argc, char* argv[])
{
#ifdef _OPENMP
int iam;
#pragma omp parallel
    { /* the parallel block starts here */
        iam=omp_get_thread_num();
#pragma omp critical
        printf("Hello from %d\n",iam);
    } /* the parallel block ends here */

#else
        printf("Hello, this is a serial program.\n");
#endif
    return 0;
}
```

- Compile
- Run
- Experiment with the `OMP_NUM_THREADS` variable.

Did you obtain the behaviour you expected?

Shared and private variables

Inside a parallel region the scope of a variable can be `shared` or `private`.

Shared

There is only one instance of the variable.

- Directive: `shared(a,b,c,…)`
  - The variable is accessible by all threads in the team
  - Threads can read and write the variable simultaneously

Note: Variables are `shared` by default

Private

Each thread has a copy of the variable. The variable is accessible only by the owner thread.

- Directive: `private(a,b,c,…)`
  - Values are undefined on entry and exit
- Directive: `firstprivate(a,b,c,…)`
  - variables are initialized with the value that the original object had before entering the parallel construct
- Directive: `lastprivate(a,b,c,…)`
  - the thread that executes the sequentially last iteration or section updates the value of the variable

Data-sharing attributes

```
int a=0;
float b=1.5;
int c=3;
#pragma omp parallel shared(a) private(b) firstprivate (c)
{
    // each thread can access to "a"
    // each thread has its own copy of "b", with an undefined value
    // each thread has its own copy of "c", with c=3
}
```

Best practice: Nullify the default with the clause `default (none)`

Solution of exercise 1.1

```c
#include <stdio.h>
#ifdef _OPENMP
#include<omp.h>
#endif

int main(int argc, char* argv[])
{
#ifdef _OPENMP
int iam;
#pragma omp parallel private(iam)
    { /* the parallel block starts here */
        iam=omp_get_thread_num();
#pragma omp critical
        printf("Hello from %d\n",iam);
    } /* the parallel block ends here */

#else
        printf("Hello, this is a serial program.\n");
#endif
    return 0;
}
```

- The *parallel* directive has an implicit barrier at its end, i.e. all the threads have to wait that every other thread complete its code block;
- *critical* doesn't have an implicit barrier at the end; it just executes the threads one by one;
- In order to build an OpenMP parallelization it is enough to use:
  - parallel directive;
  - critical directive;
  - omp_get_thread_num() function;
  - omp_get_num_threads() function;

However, this might imply to readapt the code (just like in MPI), while the aim of the OpenMP approach is to keep the parallelization as simple as possible by sharing the work among the threads in an automatic way.
In order to automatize such distribution you can use *worksharing directives*.

Worksharing constructs

Worksharing constructs: rules

Distribute the execution of the associated region

Rules

1. A worksharing region has **no barrier** on entry
2. An **implied barrier** exists at the end, unless `nowait` is specified
3. Each region **must** be encountered by all threads or none
   ⇒ Every thread must encounter the **same sequence** of worksharing regions and barrier regions

Constructs

- **for/do loop**
- **sections**
- **single**
- **workshare**

Loop construct: syntax

C/C++

```
#pragma omp for [clause[[,] clause] ... ]
  for-loops
```

Fortran

```
!$omp do [clause[[,] clause] ... ]
  do-loops
!$omp end do [nowait]
```

Useful Clauses

`schedule` can be used to specify how iterations are divided into chunks

`collapse` can be used to specify how many loops are parallelized

Rules

1. The iterations of the loop are **distributed** over the threads that already exist in the team (`scheduling` clause can manage their distribution)
2. The **iteration variable** in the `for` loop
   - if shared, is **implicitly** made private
   - must **not be modified** during the execution of the loop
   - has an **unspecified value** after the loop
3. Only loops with **canonical forms** are allowed, i.e.:
   - the **iteration count** must be known before executing the loops
   - the **incremental expression** must be addition or subtraction expression.
4. By default (i.e. without the `collapse` clause) only the external loop is parallelized, whereas the internal ones are executed sequentially for each thread. The indexes of the internal loops:
   - in Fortran are **private** by default
   - in C/C++ are **shared** by default

Loop construct: scheduling clause

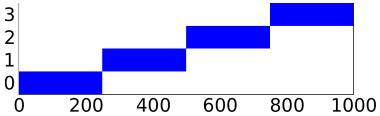
C/C++

```
#pragma omp for schedule(kind[, chunk_size])
  for-loops
```

Fortran

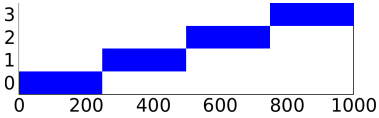
```
!$omp do schedule(kind[, chunk_size])
  do-loops
[!$omp end do [nowait] ]
```

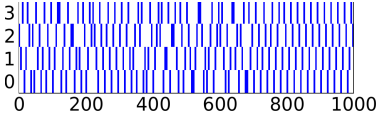
1. **static**
   - if no `chunk_size` is specified the iterations space is divided in chunks of equal size and one chunk per thread
   - if `chunk_size` is specified, chunks are assigned to the threads in a round-robin fashion



Loop construct: schedule kind

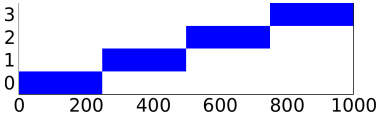
1. **static**
   - if no `chunk_size` is specified the iterations space is divided in chunks of equal size and one chunk per thread
   - if `chunk_size` is specified, chunks are assigned to the threads in a round-robin fashion
2. **dynamic**
   - iterations are divided into chunks of size `chunk_size` with default value `1`
   - the chunks are assigned to the threads as they request them



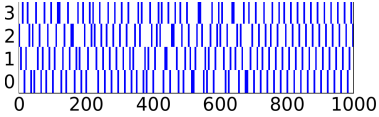


Loop construct: schedule kind

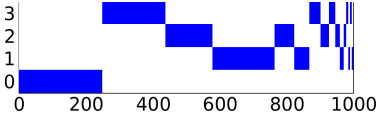
1. **static**
   - if no `chunk_size` is specified the iterations space is divided in chunks of equal size and one chunk per thread
   - if `chunk_size` is specified, chunks are assigned to the threads in a round-robin fashion



2. **dynamic**
   - iterations are divided into chunks of size `chunk_size` with default value `1`
   - the chunks are assigned to the threads as they request them



3. **guided**
   - iterations are divided into chunks of decreasing size, where `chunk_size` controls the minimum size
   - the chunks are assigned to the threads as they request them



Example

```
#pragma omp parallel
{
#pragma omp for collapse(2)
  for(int ii = 0; ii < n; ii++) {
    for(int jj = 0; jj < m; jj ++) {
      A[ii][jj] = ii*m + jj;
    }
  }
}
```

- The collapse clause indicates how many loops should be collapsed (including the external one)
- Allows parallelization of perfectly **nested rectangular loops**
- Compiler forms a **single loop** (e.g. of length NxM) and then parallelizes it
- Useful if N < no. of threads, so parallelizing the outer loop makes balancing the load difficult.

```
#pragma omp parallel
{
  #pragma omp for
  for (i=0; i<n; i++){
    // do something
  }
}
```

is equivalent to:

```
#pragma omp parallel for
  for (i=0; i<n; i++){
    // do something
  }
```

Sections construct

C/C++

```
#pragma omp sections [clause[[,] clause]...]
{
  #pragma omp section
    structured-block
  #pragma omp section
    structured-block
  ...
}
```

Fortran

```
!$omp sections [clause[[,] clause]...]
  !$omp section
    structured-block
  !$omp section
    structured-block
  ...
!$omp end sections [nowait]
```

1. `sections` is a non-iterative worksharing construct
   - it contains a set of `structured-blocks`
   - each `section` is executed **once** by **only one** of the threads
2. Scheduling of the sections is **implementation defined**
3. There is an implied barrier at the end of the construct

Single construct

C/C++

```
#pragma omp single [clause[[,] clause]...]
    structured-block
```

Fortran

```
!$omp single [clause[[,] clause] ... ]
    structured-block
!$omp end single [nowait]
```

1. The associated structured block is executed **by only one thread**
2. The other threads wait at an **implicit barrier** (unless a `nowait` clause is specified)
3. The method of choosing a thread is **implementation defined**

Note: The `master` construct is similar but specifies a structured block:

- that is **executed by the master** thread
- with **no implied barrier** on entry or exit

C/C++

```
#pragma omp master
  structured-block
```

Fortran

```
!$omp master
  structured-block
!$omp end master
```

Note: only available on Fortran

Fortran

```
!$omp workshare
  structured-block
!$omp end workshare [nowait]
```

Divides the lines of the code block into **units of work**, and each of them is then executed by a different thread. The allowed instructions are (mainly):

1. scalar assignments
2. array assignments (one element for thread)
3. `FORALL` statements
4. `WHERE` statements

When the code lines are not included in the cases above, they're treated as a single unit of work (and thus executed sequentially by one thread).

http://www.hpc.cineca.it/content/exercise-2-1

The code performs a serial matrix multiplication

- Try to parallelize it with OpenMP, acting only on the most important loop
- Try also to add the timing functions before and after the loop and print the elapsed time.

How to avoid data races

```
// C++
#pragma omp critical [clause [clause] ...]
```

```
! Fortran
!$omp critical [clause [clause]...]
```

- As previously shown, the block of code inside a critical construct is executed by only one thread at time
- This is clearly unefficient since it serializes the execution of the process
- It has to be considered an extreme solution, but less invasive possibilities has to be considered in the first instance, like the following ones

Reduction clause: some facts

```
double x[n];
double sum=0;
#pragma omp parallel for reduction (+:sum)
for (i=0; i<n; i++){
  sum+=x[i]
}
```

A reduction variable is used to accumulate a value from the different threads

Reduction clause: some facts

```
double x[n];
double sum=0;
#pragma omp parallel for reduction (+:sum)
for (i=0; i<n; i++){
  sum+=x[i]
}
```

A reduction variable is used to accumulate a value from the different threads

- is valid both on `parallel` and `work-sharing` constructs

Reduction clause: some facts

```
double x[n];
double sum=0;
#pragma omp parallel for reduction (+:sum)
for (i=0; i<n; i++){
  sum+=x[i]
}
```

A reduction variable is used to accumulate a value from the different threads

- is valid both on `parallel` and `work-sharing` constructs
- specifies an operator and one or more list items

C/C++ `+, *, -, &, |, &&, ||, max, min`

Fortran `+, *, -, .and., .or., .eqv., .neqv., max, min, iand, ior, ieor`

Reduction clause: some facts

```
double x[n];
double sum=0;
#pragma omp parallel for reduction (+:sum)
for (i=0; i<n; i++){
  sum+=x[i]
}
```

A reduction variable is used to accumulate a value from the different threads

- is valid both on `parallel` and `work-sharing` constructs
- specifies an operator and one or more list items

  C/C++ `+, *, -, &, |, &&, ||, max, min`

  Fortran `+, *, -, .and., .or., .eqv., .neqv., max, min, iand, ior, ieor`
- The items that appears in a `reduction` clause must be shared
  - a **local copy** is created and initialized based on the reduction operation
  - **updates** occur on the local copy.
  - local copies are **reduced** into a single value and combined with the original global value.

http://www.hpc.cineca.it/content/exercise-3-0

The code determines the value of π, by calculating an integral between 0 and 1. The integral is approximated as a sum of n intervals.

- Parallelize it with OpenMP
- Try also to solve the exercise without using the reduction clause

Barrier construct

C/C++

```
#pragma omp barrier
```

Fortran

```
!$omp barrier
```

Example: waiting for the master

```
int counter = 0;
#pragma omp parallel
{
#pragma omp master
  counter = 1;
#pragma omp barrier
  printf("%d\n", counter);
}
```

The `barrier` construct specifies an **explicit barrier** at the point at which the construct appears

Reminder: implicit barriers are assumed at the end of a worksharing construct, and can be removed via the `nowait` clause

Note: when not necessary, a barrier can cause slowdowns

C/C++

```
#pragma omp atomic [read | write | update | capture]
  expression-stmt
```

The `atomic` construct:

C/C++

```
#pragma omp atomic [read | write | update | capture]
  expression-stmt
```

The `atomic` construct:

- Ensures a specific storage location to be **updated atomically**, i.e. does not expose it to multiple, simultaneous writing threads

C/C++

```
#pragma omp atomic [read | write | update | capture]
  expression-stmt
```

The `atomic` construct:

- Ensures a specific storage location to be **updated atomically**, i.e. does not expose it to multiple, simultaneous writing threads
- Possible clauses are:

`read`	forces an atomic read regardless of the machine word size
`write`	forces an atomic write regardless of the machine word size
`update`	forces an atomic update (default)
`capture`	same as an update, but captures original or final value (e.g. allows a = b++ with both a and b atomically updated)

C/C++

```
#pragma omp atomic [read | write | update | capture]
  expression-stmt
```

The `atomic` construct:

- Ensures a specific storage location to be **updated atomically**, i.e. does not expose it to multiple, simultaneous writing threads
- Possible clauses are:

`read`	forces an atomic read regardless of the machine word size
`write`	forces an atomic write regardless of the machine word size
`update`	forces an atomic update (default)
`capture`	same as an update, but captures original or final value (e.g. allows a = b++ with both a and b atomically updated)

- Accesses to the same location must have **compatible** types

http://www.hpc.cineca.it/content/exercise-4-0

The code solves a 2-D Laplace equation by using a relaxation scheme.

- Parallelize the code by using OpenMP directives. Work on the most computationally intensive loop
- Try to include also the while loop in the parallel region

OpenMP tasking

The `task` construct

C/C++

```
#pragma omp task [clause]
  // structured block
```

Fortran

```
!$omp task [ clauses ]
   ! structured block
!$omp end task
```

The `task` construct timeline:

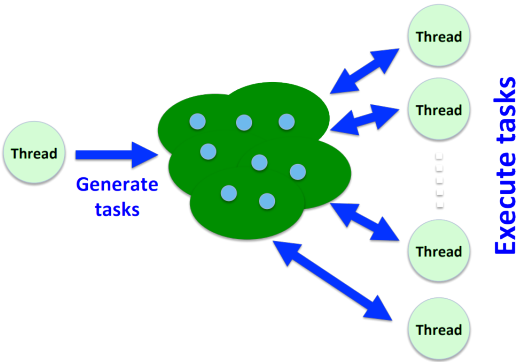
- 2008 Introduced in OpenMP 3.0
- 2013 Improved in OpenMP 4.0
- 2018 Further improvements in OpenMP 5.0 (notably reduction on tasks)

Useful for dealing with:

- large and complex applications
- load unbalancing
- irregular and dynamic structures

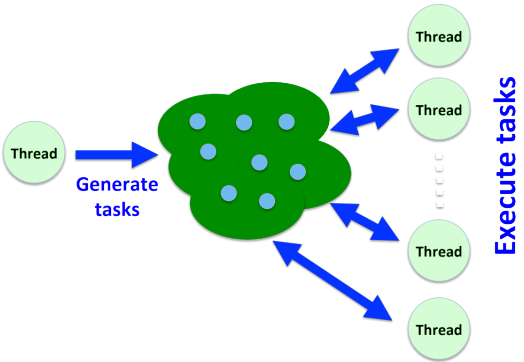
What is a task?

Is a block of instructions and data that is scheduled to be executed by a thread



What is a task?

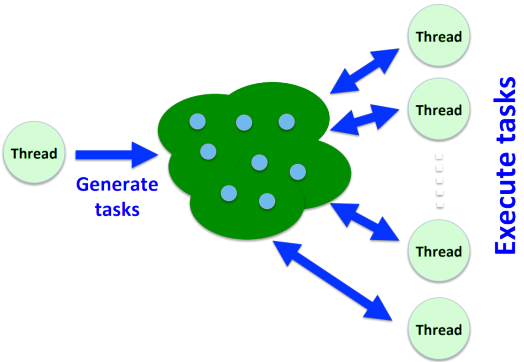
Is a block of instructions and data that is scheduled to be executed by a thread



1. A parallel region creates a team of threads

What is a task?

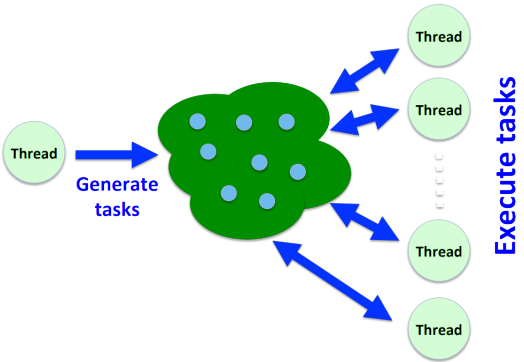
Is a block of instructions and data that is scheduled to be executed by a thread



1. A parallel region creates a team of threads
2. A single thread then creates the tasks, adding them to a queue that belongs to the team

What is a task?

Is a block of instructions and data that is scheduled to be executed by a thread



1. A parallel region creates a team of threads
2. A single thread then creates the tasks, adding them to a queue that belongs to the team
3. The *task scheduler* assigns the tasks in the queue to the threads in the team

to do things that are hard or impossible with the loop and section constructs

Linked list example

```
#pragma omp parallel            // create a team of threads
{
#pragma omp single              // where a single thread
{
  p = head_of_list();           // starts from the head of the list
  while (!end_of_list(p)){      // and, until the end of the list,
    #pragma omp task            // submits a task
      process( p );             // that will process the element
    p = next_element(p);        // and goes to the next element
  }
}
}
```

```
int a=1;
void foo(){
  int b=2, c=3;
  #pragma omp parallel shared(b) private(c)
  {
    int d=4;
    #pragma omp task
    {
    int e=5;
    // a is shared
    // b is shared
    // c is firstprivate
    // d is firstprivate
    // e is private
    }
  }
}
```

Rules (if `default` is not specified):

1. A variable that is determined to be `shared` in all enclosing constructs is `shared`
2. It is `firstprivate` otherwise

Tasks synchronization

```
void foo(){
  int a, b=2, c=3;
  #pragma omp parallel
  {
  #pragma omp single
  {
    #pragma omp task
    b=b+7;
    #pragma omp task
    c=c+4;
    //-- some kind of barrier
    #pragma omp task
    a=b+c;
  }
  }
}
```

1. **`taskwait`**: waits for tasks spawned by current task and itself

   ```
   #pragma omp taskwait
   ```

   (the equivalent of **`barrier`** for tasks)

2. **`depend`**: explicit declaration of tasks dependencies. Some more words to say ...

The depend clause

```
void foo(){
  int a, b=2, c=3;
  #pragma omp parallel
  {
  #pragma omp single
  {
    #pragma omp task depend(out:b)
    b=b+7;
    #pragma omp task depend(out:c)
    c=c+4;

    #pragma omp task depend(in:b,c)
    a=b+c;
  }
  }
}
```

- `in` it will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `out` or `inout` clause
- `out` it will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `in` clause
- `inout` same as `out`, added for completeness

The depend clause: an example

```
int sum(int x, int y){
    sleep(1);
    return x+y;
}
int square(int x){
    sleep(1);
    return x*x;
}

int main(){
  int v[6]={1,2,3,4,5,6};
  int tot=0;
  process(v, tot);
  printf("tot = %d \n", tot);
  return 0;
}
```

```
void process(int v[6], int& tot){




  v[0]=sum(v[0],v[1]);

  v[4]=sum(v[4],1);

  v[2]=sum(v[3],v[0]);

  v[5]=sum(v[2],v[4]);

  v[1]=sum(v[0],v[4]);

  v[3]=sum(v[3],-3);

  for(int i=0; i<6; i++)

    {

      tot+=square(v[i]);
    }


}
```

The depend clause: an example

```
int sum(int x, int y){
    sleep(1);
    return x+y;
}
int square(int x){
    sleep(1);
    return x*x;
}

int main(){
  int v[6]={1,2,3,4,5,6};
  int tot=0;
  process(v, tot);
  printf("tot = %d \n", tot);
  return 0;
}
```

Execution time:

12s

Output:

tot = 345

```
void process(int v[6], int& tot){




  v[0]=sum(v[0],v[1]);

  v[4]=sum(v[4],1);

  v[2]=sum(v[3],v[0]);

  v[5]=sum(v[2],v[4]);

  v[1]=sum(v[0],v[4]);

  v[3]=sum(v[3],-3);

  for(int i=0; i<6; i++)

    {

      tot+=square(v[i]);
    }


}
```

The depend clause: an example

```
int sum(int x, int y){
    sleep(1);
    return x+y;
}
int square(int x){
    sleep(1);
    return x*x;
}

int main(){
  int v[6]={1,2,3,4,5,6};
  int tot=0;
  process(v, tot);
  printf("tot = %d \n", tot);
  return 0;
}
```

```
void process(int v[6], int& tot){
#pragma omp parallel shared(v, tot)
{
#pragma omp single
{

  v[0]=sum(v[0],v[1]);

  v[4]=sum(v[4],1);

  v[2]=sum(v[3],v[0]);

  v[5]=sum(v[2],v[4]);

  v[1]=sum(v[0],v[4]);

  v[3]=sum(v[3],-3);

  for(int i=0; i<6; i++)

    {

      tot+=square(v[i]);
    }
}
}
}
```

The depend clause: an example

```
int sum(int x, int y){
    sleep(1);
    return x+y;
}
int square(int x){
    sleep(1);
    return x*x;
}

int main(){
  int v[6]={1,2,3,4,5,6};
  int tot=0;
  process(v, tot);
  printf("tot = %d \n", tot);
  return 0;
}
```

Execution time:

12s

Output:

tot = 345

```
void process(int v[6], int& tot){
#pragma omp parallel shared(v, tot)
{
#pragma omp single
{

  v[0]=sum(v[0],v[1]);

  v[4]=sum(v[4],1);

  v[2]=sum(v[3],v[0]);

  v[5]=sum(v[2],v[4]);

  v[1]=sum(v[0],v[4]);

  v[3]=sum(v[3],-3);

  for(int i=0; i<6; i++)

    {

      tot+=square(v[i]);
    }
}
}
}
```

The depend clause: an example

```
int sum(int x, int y){
    sleep(1);
    return x+y;
}
int square(int x){
    sleep(1);
    return x*x;
}

int main(){
  int v[6]={1,2,3,4,5,6};
  int tot=0;
  process(v, tot);
  printf("tot = %d \n", tot);
  return 0;
}
```

```
void process(int v[6], int& tot){
#pragma omp parallel shared(v, tot)
{
#pragma omp single
{
  #pragma omp task depend(inout:v[0]) depend(in:v[1])
  v[0]=sum(v[0],v[1]);

  v[4]=sum(v[4],1);

  v[2]=sum(v[3],v[0]);

  v[5]=sum(v[2],v[4]);

  v[1]=sum(v[0],v[4]);

  v[3]=sum(v[3],-3);

  for(int i=0; i<6; i++)

    {

      tot+=square(v[i]);
    }
}
}
}
```

The depend clause: an example

```
int sum(int x, int y){
    sleep(1);
    return x+y;
}
int square(int x){
    sleep(1);
    return x*x;
}

int main(){
  int v[6]={1,2,3,4,5,6};
  int tot=0;
  process(v, tot);
  printf("tot = %d \n", tot);
  return 0;
}
```

```
void process(int v[6], int& tot){
#pragma omp parallel shared(v, tot)
{
#pragma omp single
{
  #pragma omp task depend(inout:v[0]) depend(in:v[1])
  v[0]=sum(v[0],v[1]);
  #pragma omp task depend(inout:v[4])
  v[4]=sum(v[4],1);
  #pragma omp task depend(in:v[0],v[3]) depend(out:v[2])
  v[2]=sum(v[3],v[0]);
  #pragma omp task depend(in:v[2],v[4]) depend(out:v[5])
  v[5]=sum(v[2],v[4]);
  #pragma omp task depend(in:v[0],v[4]) depend(out:v[1])
  v[1]=sum(v[0],v[4]);
  #pragma omp task depend(inout:v[3])
  v[3]=sum(v[3],-3);
  #pragma omp taskwait
  for(int i=0; i<6; i++)

    {

      tot+=square(v[i]);
    }
}
}
}
```

The depend clause: an example

```
int sum(int x, int y){
    sleep(1);
    return x+y;
}
int square(int x){
    sleep(1);
    return x*x;
}

int main(){
  int v[6]={1,2,3,4,5,6};
  int tot=0;
  process(v, tot);
  printf("tot = %d \n", tot);
  return 0;
}
```

Execution time:

9s

Output:

tot = 345

```
void process(int v[6], int& tot){
#pragma omp parallel shared(v, tot)
{
#pragma omp single
{
  #pragma omp task depend(inout:v[0]) depend(in:v[1])
  v[0]=sum(v[0],v[1]);
  #pragma omp task depend(inout:v[4])
  v[4]=sum(v[4],1);
  #pragma omp task depend(in:v[0],v[3]) depend(out:v[2])
  v[2]=sum(v[3],v[0]);
  #pragma omp task depend(in:v[2],v[4]) depend(out:v[5])
  v[5]=sum(v[2],v[4]);
  #pragma omp task depend(in:v[0],v[4]) depend(out:v[1])
  v[1]=sum(v[0],v[4]);
  #pragma omp task depend(inout:v[3])
  v[3]=sum(v[3],-3);
  #pragma omp taskwait
  for(int i=0; i<6; i++)

    {

      tot+=square(v[i]);
    }
}
}
}
```

The depend clause: an example

```
int sum(int x, int y){
    sleep(1);
    return x+y;
}
int square(int x){
    sleep(1);
    return x*x;
}

int main(){
  int v[6]={1,2,3,4,5,6};
  int tot=0;
  process(v, tot);
  printf("tot = %d \n", tot);
  return 0;
}
```

```
void process(int v[6], int& tot){
#pragma omp parallel shared(v, tot)
{
#pragma omp single
{
  #pragma omp task depend(inout:v[0]) depend(in:v[1])
  v[0]=sum(v[0],v[1]);
  #pragma omp task depend(inout:v[4])
  v[4]=sum(v[4],1);
  #pragma omp task depend(in:v[0],v[3]) depend(out:v[2])
  v[2]=sum(v[3],v[0]);
  #pragma omp task depend(in:v[2],v[4]) depend(out:v[5])
  v[5]=sum(v[2],v[4]);
  #pragma omp task depend(in:v[0],v[4]) depend(out:v[1])
  v[1]=sum(v[0],v[4]);
  #pragma omp task depend(inout:v[3])
  v[3]=sum(v[3],-3);

  for(int i=0; i<6; i++)
    #pragma omp task depend(in:v[i])
    {
      #pragma omp atomic update
      tot+=square(v[i]);
    }
}
}
}
```

The depend clause: an example

```
int sum(int x, int y){
    sleep(1);
    return x+y;
}
int square(int x){
    sleep(1);
    return x*x;
}

int main(){
  int v[6]={1,2,3,4,5,6};
  int tot=0;
  process(v, tot);
  printf("tot = %d \n", tot);
  return 0;
}
```

Execution time:

4s

Output:

tot = 345

```
void process(int v[6], int& tot){
#pragma omp parallel shared(v, tot)
{
#pragma omp single
{
  #pragma omp task depend(inout:v[0]) depend(in:v[1])
  v[0]=sum(v[0],v[1]);
  #pragma omp task depend(inout:v[4])
  v[4]=sum(v[4],1);
  #pragma omp task depend(in:v[0],v[3]) depend(out:v[2])
  v[2]=sum(v[3],v[0]);
  #pragma omp task depend(in:v[2],v[4]) depend(out:v[5])
  v[5]=sum(v[2],v[4]);
  #pragma omp task depend(in:v[0],v[4]) depend(out:v[1])
  v[1]=sum(v[0],v[4]);
  #pragma omp task depend(inout:v[3])
  v[3]=sum(v[3],-3);

  for(int i=0; i<6; i++)
    #pragma omp task depend(in:v[i])
    {
      #pragma omp atomic update
      tot+=square(v[i]);
    }
}
}
}
```

Optional exercise

- Try to repeat the Exercise 4 using tasks
- Do you obtain the same results?
- Do you obtain the same performances?
- If you see any difference, can you explain why?

Conclusions

- Always check the OpenMP version installed
- This lecture was just an introduction, and many important aspects were not covered:
  - flush
  - locks
  - simd
  - offload
  - taskloops
  - etc etc
- Most of the topic of this lecture are compliant since OpenMP 4.0, but OpenMP 5.0 went deeper in tasking directives.

If interested, www.openmp.org is your bible!

Credits

A special thank to Alessandro Colombo and to all the people who contributed more or less synchronously and more or less consciently to these slides so far:

Mirko Cestari, Fabio Affinito, Cristiano Padrin, Neva Besker, Pietro Bonfá, Gian Franco Marras, Marco Comparato, Massimiliano Culpo, Giorgio Amati, Federico Massaioli, Marco Rorro, Vittorio Ruggiero, Francesco Salvadore, Claudia Truini, etc

http://www.hpc.cineca.it/content/exercise-5-0

The code computes the interaction forces of N point charges at a set potential V and periodic boundary conditions. Try to parallelize the code keeping in mind the following variations:

- while parallelizing the for loop you can choose between different scheduling options (static, guided, dynamic, ...). Make some tests and see the differences.
- pay attention to the update of the ``forces'' array. You may want to update it atomically or by reducing it. Test both options and see the differences.

Good luck!