



22nd Summer School on **PARALLEL** COMPUTING

Introduction to OpenMP

Massimiliano Culpo - m.culpo@cinca.it

Gian Franco Marras - g.marras@cinca.it

Marco Rorro - m.rorro@cinca.it

SuperComputing Applications and Innovation Department



Outline

- 1 Introduction
 - Shared Memory
 - The OpenMP Model
- 2 Main Elements
- 3 Synchronization And Other Functionalities
- 4 Conclusions



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 (messages)
 - It is the responsibility of the programmer to design and implement the exchange of data between processes
- You can not adopt a strategy of incremental parallelization
 - The communication structure of the entire program has to be implemented
- The communications have a cost
- It is difficult to have a single version of the code for the serial and MPI program
 - Additional variables are needed
 - You need to manage the correspondence between local variables and global data structure

What is OpenMP?

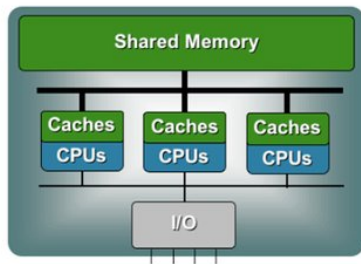
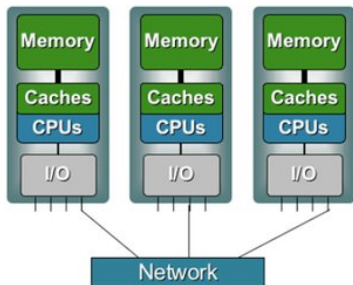
- De-facto standard Application Program Interface (API) to write **shared memory parallel applications** in C, C++ and Fortran
- Consists of **compilers directives**, **run-time routines** and **environment variables**
- “Open specifications for Multi Processing” maintained by the OpenMP Architecture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

A bit of history

- Born to satisfy the need of unification of proprietary solutions
- The past
 - October 1997 - Fortran version 1.0
 - October 1998 - C/C++ version 1.0
 - November 1999 - Fortran version 1.1 (interpretations)
 - November 2000 - Fortran version 2.0
 - March 2002 - C/C++ version 2.0
 - May 2005 - combined C/C++ and Fortran version 2.5
 - May 2008 - version 3.0 (*task!*)
- The present
 - July 2011 - version 3.1
- The future
 - version 4.0 will be released soon

Shared memory architectures

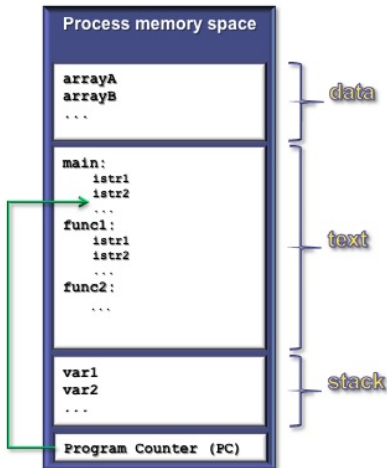
- All processors may access the whole main memory



- Non-Uniform Memory Access**
 - Memory access time is non-uniform
- Uniform Memory Access**
 - Memory access time is uniform

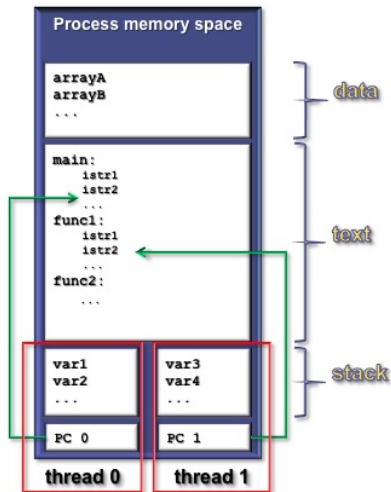
Process and thread

- 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

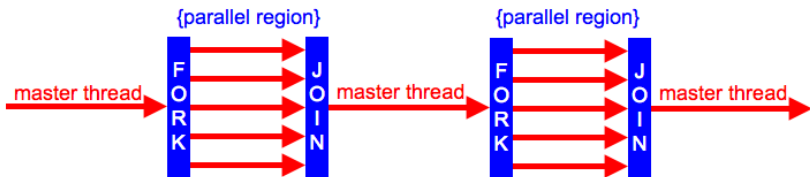


Multi-threading

- 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



The OpenMP execution model

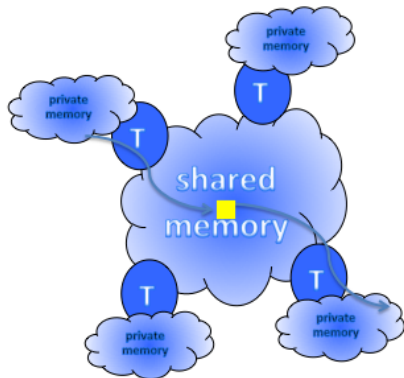


- The **Fork & Join** Model

- Each OpenMP program begins to execute with a single thread (Master thread) that runs the program in serial
- 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
- The thread team runs in parallel the code contained in the parallel region (Single Program Multiple Data model)
- 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

The OpenMP memory model

- All threads have access to the same globally **shared** memory
- Data in **private** memory is only accessible by the thread owning this memory
- No other thread sees the change(s)
- Data transfer is through shared memory and is completely transparent to the application



Outline

- 1 Introduction
- 2 **Main Elements**
 - Foremost Constructs And Data-Sharing Clauses
 - Worksharing Construct
 - Data-Sharing Clauses
- 3 Synchronization And Other Functionalities
- 4 Conclusions



Directives

- Syntax:
 - in C/C++:
`#pragma omp direttiva`
 - in Fortran:
`!$omp direttiva`
 - in Fortran (fixed format):
`c$omp direttiva`
- Mark a block of code
- Specify to the compiler how to run in parallel the code block
- The serial code "coexists" with the parallel code
 - A serial compilation ignores the directives
 - A compilation with OpenMP support takes them into account

Clauses

- Syntax: *directive* [*clause* [*clause*]...]
- Specify additional information to the directives
- Variables handling
 - What are shared among all threads (the default)
 - Which are private to each thread
 - How to initialize the private ones
 - What is the default
- Execution control
 - How many threads in the team
 - How to distribute the work
- ATTENTION: they may alter code semantic
 - The code can be corrected in serial but not in parallel or vice versa

Environment variables

- `OMP_NUM_THREADS`: sets number of threads
- `OMP_STACKSIZE "size [B|K|M|G]"`: size of the stack for threads
- `OMP_DYNAMIC {TRUE|FALSE}`: dynamic thread adjustment
- `OMP_SCHEDULE "schedule [, chunk]"`: iteration scheduling scheme
- `OMP_PROC_BIND {TRUE|FALSE}`: bound threads to processors
- `OMP_NESTED {TRUE|FALSE}`: nested parallelism
- ...
- To set them
 - In `bash`: `setenv OMP_NUM_THREADS 4`
 - In `sh`: `export OMP_NUM_THREADS=4`



Runtime functions

- Query/specify some specific feature or setting
 - `omp_get_thread_num()`: get thread ID (0 for master thread)
 - `omp_get_num_threads()`: get number of threads in the team
 - `omp_set_num_threads(int n)`: set number of threads
 - ...
- Allow you to manage fine-grained access (lock)
 - `omp_init_lock(lock_var)`: initializes the OpenMP lock variable `lock_var` of type `omp_lock_t`
 - ...
- Timing functions
 - `omp_get_wtime()`: returns elapsed wallclock time
 - `omp_get_wtick()`: returns timer precision
- Functions interface:
 - C/C++: `#include <omp.h>`
 - Fortran: use `omp_lib` (or `include 'omp_lib.h'`)



Conditional compilation

- To avoid dependency on OpenMP libraries you can use pre-processing directives
 - and the preprocessor macro `_OPENMP` predefined by the standard
 - C preprocessing directives can be used in Fortran too as well `!$` in free form and old style fixed form `*$` and `c$`

C/C++

```
#ifdef _OPENMP
printf("Compiled with OpenMP support:%d",_OPENMP);
#else
printf("Compiled for serial execution.");
#endif
```

Fortran

```
!$ print *, "Compiled with OpenMP support",_OPENMP
```




Compiling and linking

- The compilers that support OpenMP interpret the directives only if they are invoked with a compiler option (switch)
 - GNU: `-fopenmp` for Linux, Solaris, AIX, MacOSX, Windows.
 - IBM: `-qsmp=omp` for Windows, AIX and Linux.
 - Sun: `-xopenmp` for Solaris and Linux.
 - Intel: `-openmp` on Linux or Mac, or `-Qopenmp` on Windows
 - PGI: `-mp`
- Most compilers emit useful information enabling extra warning or report options



parallel construct

- It is the main construct of OpenMP that creates a parallel region
 - A construct is the lexical extent to which an executable directive applies
 - A region is the dynamic extent to which an executable directive applies
 - A parallel region is a block of code executed by all threads in the team

C/C++

```
#pragma omp parallel
{
// some code to execute in parallel
} // end of the parallel region (implied barrier)
```

Fortran

```
!$omp parallel
! some code to execute in parallel
!$omp end parallel
```



Hello world!

C

```
#include <stdio.h>
int main()
{
  #pragma omp parallel
  {
    printf("Hello world!\n");
  }
  return 0;
}
```

Fortran

```
Program Hello
!$omp parallel
  print *, "Hello world!"
!$omp end parallel
end program Hello
```



shared and private variables

- Inside a parallel region, the variables of the serial program can be essentially **shared** or **private**
 - **shared**: there is only one instance of the data
 - Data is accessible by all threads in the team
 - Threads can read and write the data simultaneously
 - All threads access the same address space
 - **private**: each thread has a copy of the data
 - No other thread can access this data
 - Changes are only visible to the thread owning the data
 - Values are undefined on entry and exit
- Variables are shared by default but with the clause **default (none)**
 - No implicit default, you have to scope all variables explicitly



Data races & critical construct

- A data race is when two or more threads access the same(=shared) memory location
 - Asynchronously 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 block of code inside a **critical** construct is executed by only one thread at time
- It is a synchronization to avoid simultaneous access to shared data

It could be enough ...

C

```
sum = 0;
#pragma omp parallel private(i, MyThreadID)
{
    ThreadID = omp_get_thread_num(); NumThreads = omp_get_num_threads();
    int psum = 0;
    for (i=ThreadID*N/NumThreads; i<(ThreadID+1)*N/NumThreads; i++)
        psum +=x[i];
#pragma omp critical
    sum +=psum;
}
```

Fortran

```
sum = 0
!$omp parallel private(i, MyThreadID, psum)
MyThreadID = omp_get_thread_num(); NumThreads = omp_get_num_threads()
psum =0
do i=MyThreadID*N/NumThreads+1, min((MyThreadID+1)*N/NumThreads,N)
    psum = psum + x(i)
end do
!$omp critical
    sum = sum + psum;
!$omp end critical
!$omp end parallel
```



but life is easier

- Essentially for a parallelization it could be enough:
 - the `parallel` construct
 - the `critical` construct
 - the `omp_get_thread_num()` function
 - and the `omp_get_num_threads()` function
- But we need to distribute the serial work among threads
- And doing it by hand is tiring
- The worksharing constructs automate the process



Worksharing construct

- A worksharing construct distributes the execution of the associated parallel region over the threads that must encounter 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
- The OpenMP API defines the following worksharing constructs:
 - `for/do` loop construct
 - `sections` construct
 - `single` construct
 - `workshare` construct (only Fortran)



Loop construct

- The iterations of the loop are distributed over the threads that already exist in the team
- The iteration variable of the loop is made private by default
- The inner loops are executed sequentially by each thread
- Beware the data-sharing attribute of the inner loop iteration variables
 - In Fortran they are private by default
 - In C/C++ they aren't
- Requirements for (loop) parallelization:
 - **no dependencies** (between loop indices)

Loop construct syntax

C/C++

```
#pragma omp for [clauses]
  for(i=0; i<n; i++)
  { ... }
```

Fortran

```
!$omp do [clauses]
  do i = 1, n
    ...
  end do
[!$omp end do [nowait] ]
```

- Random access iterators are supported too

C++

```
#pragma omp for [clauses]
  for(i=v.begin(); i < v.end(); i++)
  { ... }
```

Loop construct example

C

```
int main ()
{
    int i, n=10;
    int a[n], b[n], c[n];
    ...
#pragma omp parallel
    {
        #pragma omp for
        for (i=0; i<n; i++)
            {
                a[i] = b[i] = i;
                c[i] = 0;
            }
        #pragma omp for
        for (i=0; i<n; i++)
            c[i] = a[i] + b[i];
    }
    ...
}
```



Loop construct example

Fortran

```
Program doexample
integer, parameter:: n=10
integer:: i, a(n),b(n),c(n)
!$omp parallel
!$omp do
do i=1, n
    a(i) = i
    b(i) = i
    c(i) = 0
end do
!$omp end do
!$omp do
do i=1, n
    c(i) = a(i) + b(i);
end do
!$omp end do
!$omp end parallel
...
```

Loop collapse

- Allows parallelization of perfectly nested loops
- The **collapse** clause on **for/do** loop indicates how many loops should be collapsed
- Compiler forms a single loop and then parallelizes it

C/C++

```
#pragma omp for collapse(2) private(j)
for (i=0; i<nx; i++)
    for (j=0; j<ny; j++)
        ...
```

Fortran

```
!$omp do collapse(2)
do j=1, ny
do i=1, nx
    ...
```

The schedule clause

- `schedule (static | dynamic | guided | auto [, chunk])` 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.

C/C++

```
#pragma omp for \  
schedule(kind [,chunk])
```

Fortran

```
!$omp do &  
!$omp schedule(kind [,chunk])
```

- Note continuation line

static scheduling

- Iterations are divided into chunks of size **chunk**, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number
- It is the default schedule and the default **chunk** is approximately $N_{iter} / N_{threads}$
- For example:

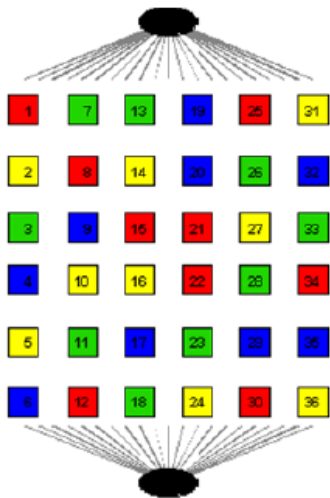
```
!$omp parallel do &  
!$omp schedule(static,3)
```



dynamic scheduling

- 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.
- The default **chunk** is 1
- For example:

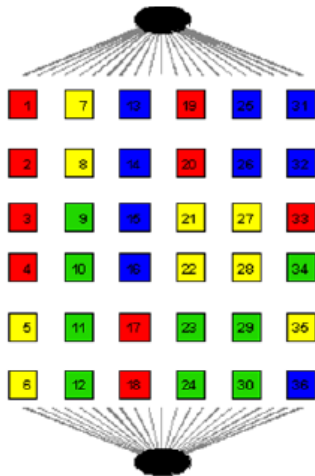
```
!$omp parallel do &  
!$omp schedule(dynamic,1)
```



guided scheduling

- 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 decreases to **chunk**
- The default value of **chunk** is 1
- For example:

```
!$omp parallel do &  
!$omp schedule(guided,1)
```





runtime and auto scheduling

- **runtime**: iteration scheduling scheme is set at runtime through the environment variable **OMP_SCHEDULE**
 - For example:

```
!$omp parallel do &  
!$omp schedule(runtime)
```
 - the scheduling scheme can be modified without recompiling the program changing the environment variable **OMP_SCHEDULE**, for example: `setenv OMP_SCHEDULE "dynamic,50"`
 - Only useful for experimental purposes during the parallelization
- **auto**: the decision regarding scheduling is delegated to the compiler and/or runtime system

Scheduling experiment

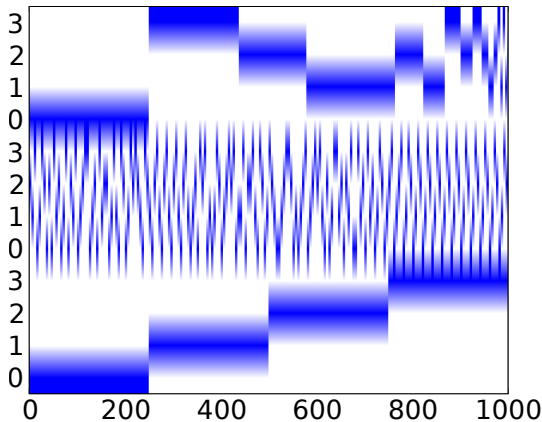


Figure : Different scheduling for a 1000 iterations loop with 4 threads:
guided (top), dynamic (middle), static (bottom)



sections construct

C/C++

```
#pragma omp sections [clauses]
{
  #pragma omp section
  { structured block }
  #pragma omp section
  { structured block }
  ...
}
```

Fortran

```
!$omp sections [clauses]
!$omp section
! structured block
!$omp end section
!$omp section
! structured block
!$omp end section
!...
!$omp end sections
```

- It is a worksharing construct to distribute structured blocks of code among threads in the team
 - Each thread receives a **section**
 - When a thread has finished to execute its section, it receives another **section**
 - If there are no other **sections** to execute, threads wait for others to end up



single construct

C/C++

```
#pragma omp single [private] [firstprivate] [copyprivate] [nowait]  
{ structured block }
```

Fortran

```
!$omp single [private] [firstprivate]  
! structured block  
!$omp end single [copyprivate] [nowait]
```

- It is a worksharing construct
- The first thread that reaches it executes the associated block
- The other threads in the team wait at the implicit barrier at the end of the construct unless a **nowait** clause is specified



The Fortran workshare construct

Fortran

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

- The structured block enclosed in the **workshare** construct is divided into units of work that are then assigned to the thread such that each unit is executed by one thread only once
- It is only supported in Fortran in order to parallelize the array syntax



Data-sharing attributes

- In a **parallel** construct the data-sharing attributes are *implicitly determined* by the **default** clause, if present
 - if no **default** clause is present they are **shared**
- Certain variables have a *predetermined* data-sharing attributes
 - Variables with automatic storage duration that are declared in a scope inside a construct are **private**
 - Objects with dynamic storage duration are **shared**
 - The loop iteration variable(s) in the associated for-loop(s) of a **for** construct is (are) **private**
 - A loop iteration variable for a sequential loop in a **parallel** construct is **private** in the innermost such construct that encloses the loop (only Fortran)
 - Variables with static storage duration that are declared in a scope inside the construct are shared
- ...



Data-sharing attributes clauses

- *Explicitly determined* data-sharing attributes are those that are referenced in a given construct and are listed in a data-sharing attribute clause
 - **shared(list)**: there is only one instance of the objects in the list accessible by all threads in the team
 - **private(list)**: each thread has a copy of the variables in the list
 - **firstprivate(list)**: same as **private** but all variables in the list are initialized with the value that the original object had before entering the parallel construct
 - **lastprivate(list)**: same as **private** but the thread that executes the sequentially last iteration or section updates the value of the objects in the list
- The **default** clause sets the implicit default
 - **default (none | shared)** in C/C++
 - **default (none | shared | private | firstprivate)** in Fortran

The reduction clause

- With the Data-Sharing attributes clause `reduction(op:list)`
- For each list item, a private copy is created in each implicit task
- The local copy is initialized appropriately according to the operator (for example, if `op` is `+` they are initialized to 0)
- After the end of the region, the original list item is updated with the values of the private copies using the specified operator
- Supported operators for a `reduction` clause are:
 - C: `+, *, -, &, |, ^, &&, || max e min` dalla 3.1)
 - Fortran: `+, *, -, .and., .or., .eqv., .neqv., max, min, iand, ior, ieor`
- Reduction variables must be shared variables
- The `reduction` clause is valid on `parallel, for/do` loop and `sections` constructs



reduction example

C/C++

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

Fortran

```
!$omp parallel for reduction(+:sum)
do i=1, n
    sum = sum + x(i)
end do
!$omp end parallel do
```

- Yes, worksharing constructs can be combined with `parallel`
- Beware that the value of a reduction is undefined from the moment the first thread reaches the clause till the operation is completed

Outline

- 1 Introduction
- 2 Main Elements
- 3 Synchronization And Other Functionalities
 - `barrier` Construct And `nowait` Clause
 - `atomic` Construct
 - Task Parallelism Overview
- 4 Conclusions



barrier construct and nowait

- In a parallel region threads proceed asynchronously
- Until they encounter a barrier
 - At the barrier all threads wait and continue only when all threads have reached the barrier
 - The barrier guarantees that ALL the code above has been executed
- Explicit barrier
 - `#pragma omp barrier` in C/C++
 - `!$omp barrier` in Fortran
- Implicit barrier
 - At the end of the worksharing construct
 - Sometimes it is not necessary, and would cause slowdowns
 - It can be removed with the clause `nowait`
 - In C/C++, it is one of the clauses on the pragma
 - In Fortran, it is appended at the closing part of the construct



atomic construct

- The **atomic** construct applies only to statements that update the value of a variable
 - Ensures that no other thread updates the variable between reading and writing
- The allowed instructions differ between Fortran and C/C++
 - Refer to the OpenMP specifications
- It is a special lightweight form of a **critical**
 - Only read/write are serialized, and only if two or more threads access the same memory address

C/C++

```
#pragma omp atomic [clause]  
<statement>
```

Fortran

```
!$omp atomic [clause]  
<statement>
```

atomic Examples

C/C++

```
#pragma omp atomic update
x += n*mass; // default update

#pragma omp atomic read
v = x; // read atomically

#pragma omp atomic write
x = n*mass; write atomically

#pragma omp atomic capture
v = x++; // capture x in v and
         // update x atomically
```

Fortran

```
!$omp atomic update
x = x + n*mass // default
update

!$omp atomic read
v = x // read atomically

!$omp atomic write
x = n*mass write atomically

!$omp atomic capture
v = x // capture x in v and
x = x+1 // update x atomical
!$omp end atomic
```



master construct

C/C++

```
#pragma omp master  
{<code-block>}
```

Fortran

```
!$omp master  
  <code-block>  
!$omp end master
```

- Only the master thread executes the associated code block
- There is no implied barrier on entry or exit!



The threadprivate directive

C/C++

```
#pragma omp threadprivate(list)
```

Fortran

```
!$omp threadprivate(list)
```

- Is a declarative directive
- Is used to create private copies of
 - *file-scope*, *namespace-scope* or **static** variables in C/C++
 - **common** block or module variables in Fortran
- Follows the variable declaration in the same program unit
- Initial data are undefined, unless the **copyin** clause is used

Orphaning

- The OpenMP specification does not restrict worksharing construct and synchronization directives to be within the lexical extent of a parallel region. These directives can be **orphaned**
- That is, they can appear outside the lexical extent of a parallel region
- They will be ignored if called from a serial region
- but data-sharing attributes will be applied



Task parallelism

- Main addition to OpenMP 3.0 enhanced in 3.1 and 4.0
- Allows to parallelize irregular problems
 - Unbounded loop
 - Recursive algorithms
 - Producer/consumer schemes
 - Multiblock grids, Adaptive Mesh Refinement
 - ...



Pointer chasing in OpenMP 2.5

C/C++

```
#pragma omp parallel private(p)
  p = head;
  while ( p ) {
    #pragma omp single nowait
      process(p);
    p = p->next;
  }
```

Fortran

```
!$omp parallel private(p)
  p = head
  do while ( associated( p ) )
    !$omp single nowait
      call process(p)
    p => p%next
  end do
```

- Transformation to a “canonical” loop can be very labour-intensive/expensive
- The main drawback of the **single nowait** solution is that it is not composable
- Remind that all worksharing construct can not be nested

Tree traversal in OpenMP 2.5

C/C++

```

void preorder (node *p) {
  process(p->data);
  #pragma omp parallel sections \
    num_threads(2)
  {
    #pragma omp section
    if (p->left)
      preorder(p->left);
    #pragma omp section
    if (p->right)
      preorder(p->right);
  }
}

```

Fortran

```

recursive subroutine preorder(p)
  type(node), pointer :: p
  call process(p%data)
  !$omp parallel sections
  !$omp num_threads(2)
  !$omp section
  if (associated(p%left))
    call preorder(p%left)
  end if
  !$omp section
  if (associated(p%right))
    call preorder(p%right)
  end if
  !$omp end sections
end subroutine preorder

```

- You need to set **OMP_NESTED** to true, but stressing nested parallelism so much is not a good idea ...



First & foremost tasking construct

C/C++

```
#pragma omp parallel [clauses]
{
    <structured block>
}
```

Fortran

```
!$omp parallel [clauses]
    <structured block>
!$omp end parallel
```

- Creates both threads and tasks
- These tasks are “implicit”
- Each one is immediately executed by one thread
- Each of them is tied to the assigned thread

New tasking construct

C/C++

```
#pragma omp task [clauses]
{
  <structured block>
}
```

Fortran

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

- Immediately creates a new task but not a new thread
- This task is “explicit”
- It will be executed by a thread in the current team
- It can be deferred until a thread is available to execute
- The data environment is built at creation time
 - Variables inherit their data-sharing attributes but
 - **private variables become firstprivate**



Pointer chasing using task

C/C++

```
#pragma omp parallel private(p)
#pragma omp single
{
    p = head;
    while ( p ) {
        #pragma omp task
            process(p);
        p = p->next;
    }
}
```

Fortran

```
!$omp parallel private(p)
!$omp single
    p = head
    do while (associated(p))
        !$omp task
            call process(p)
        !$omp end task
        p => p%next
    end do
!$omp end single
!$omp end parallel
```

- One thread creates task
 - It packages code and data environment
 - Then it reaches the implicit barrier and starts to execute the task
- The other threads reach straight the implicit barrier and start to execute task

Load balancing on lists with task

C/C++

```
#pragma omp parallel
{
    #pragma omp for private(p)
    for (i=0; i<num_lists; i++) {
        p = head[i];
        while ( p ) {
            #pragma omp task
            process(p);
            p = p->next;
        }
    }
}
```

Fortran

```
!$omp parallel
  !$omp do private(p)
  do i=1,num_lists
    p => head[i]
    do while (associated(p))
      !$omp task
      call process(p)
      !$omp end task
      p => p%next
    end do
  end do
!$omp end do
!$omp end parallel
```

- Assign one list per thread could be unbalanced
- Multiple threads create task
- The whole team cooperates to execute them

Tree traversal with task

C/C++

```
void preorder (node *p) {
    process(p->data);
    if (p->left)
        #pragma omp task
        preorder(p->left);
    if (p->right)
        #pragma omp task
        preorder(p->right);
}
```

Fortran

```
recursive subroutine preorder(p)
    type(node), pointer :: p
    call process(p%data)
    if (associated(p%left))
        !$omp task
        call preorder(p%left)
        !$omp end task
    end if
    if (associated(p%right))
        !$omp task
        call preorder(p%right)
        !$omp end task
    end if
end subroutine preorder
```

- Tasks are composable
- It isn't a worksharing construct

Postorder tree traversal with task

C/C++

```
void postorder (node *p) {
    process(p->data);
    if (p->left)
        #pragma omp task
            postorder(p->left);
    if (p->right)
        #pragma omp task
            postorder(p->right);
    #pragma omp taskwait
    process(p->data);
}
```

Fortran

```
recursive subroutine postorder(p)
    type(node), pointer :: p
    if (associated(p%left))
        !$omp omp task
            call postorder(p%left)
        !$omp end task
    end if
    if (associated(p%right))
        !$omp omp task
            call postorder(p%right)
        !$omp end task
    end if
    !$omp taskwait
    call process(p%data)
end subroutine postorder
```

- **taskwait** suspends parent task until children tasks are completed



Outline

- 1 Introduction
- 2 Main Elements
- 3 Synchronization And Other Functionalities
- 4 Conclusions**

Conclusions

- What we left out
 - **flush** directive and lock
 - **ordered** construct
 - Data copying clause **copyin** and **copyprivate**
 - and many other
- Where to find more
 - In the OpenMP specification that can be downloaded from www.openmp.org
 - The same web site make available further resources: forum, tutorial, news, etc.
- Credits
 - Several people of the SCAI staff: Marco Comparato, Federico Massaioli, Simone Meloni, Francesco Salvatore, ...
 - Many people involved on OpenMP: Ruud van der Pas, Alejandro Duran, Bronis de Supinski, Tim Mattson and Larry Meadows, ...