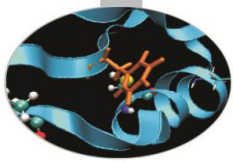
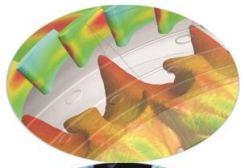
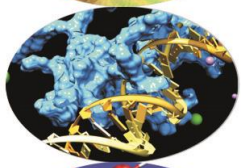
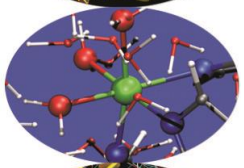
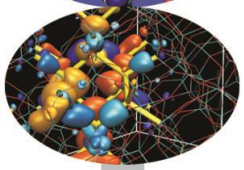
CINECA SCAI
SuperComputing Applications and Innovation



OpenMP

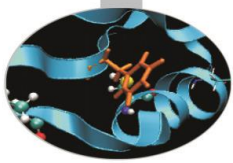
Introduction to Parallel Computing with MPI and OpenMP

P.Dagna

May 2015

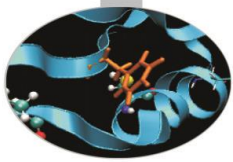
CINECA

A bit of history

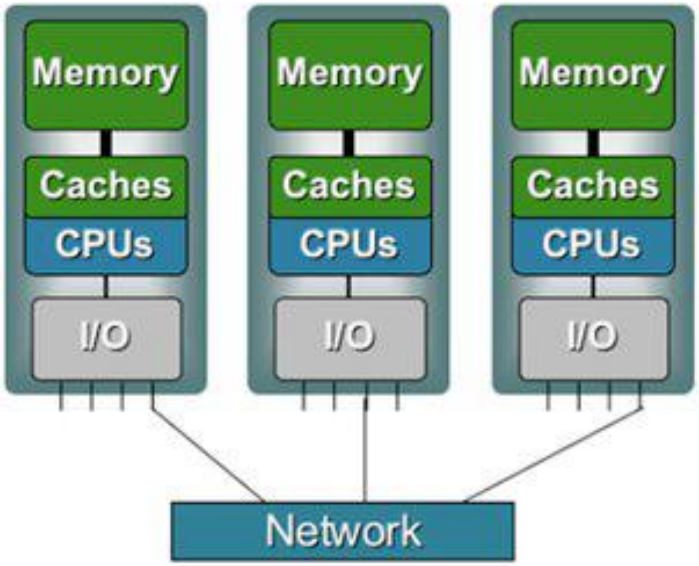


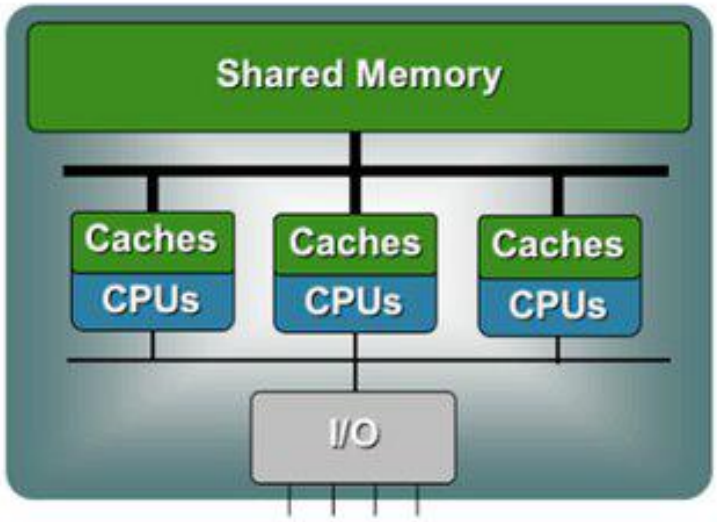
- Born to satisfy the need of unification of proprietary solutions
- The past
  - October 1997 - Fortran version 1.0
  - October 1998 - C/C++ version 1.0
  - November 1999 - Fortran version 1.1
  - November 2000 - Fortran version 2.0
  - March 2002 - C/C++ version 2.0
  - May 2005 - combined C/C++ and Fortran version 2.5
  - May 2008 - version 3.0 (task!)
- The present
  - July 2011 - version 3.1
  - July 2013 - version 4.0 (Accelerator, SIMD extensions, …)
- The future
  - Version 4.1/5.0

Shared memory architectures



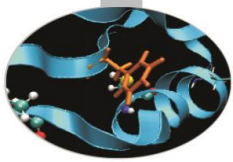
- All processors may access the whole main memory



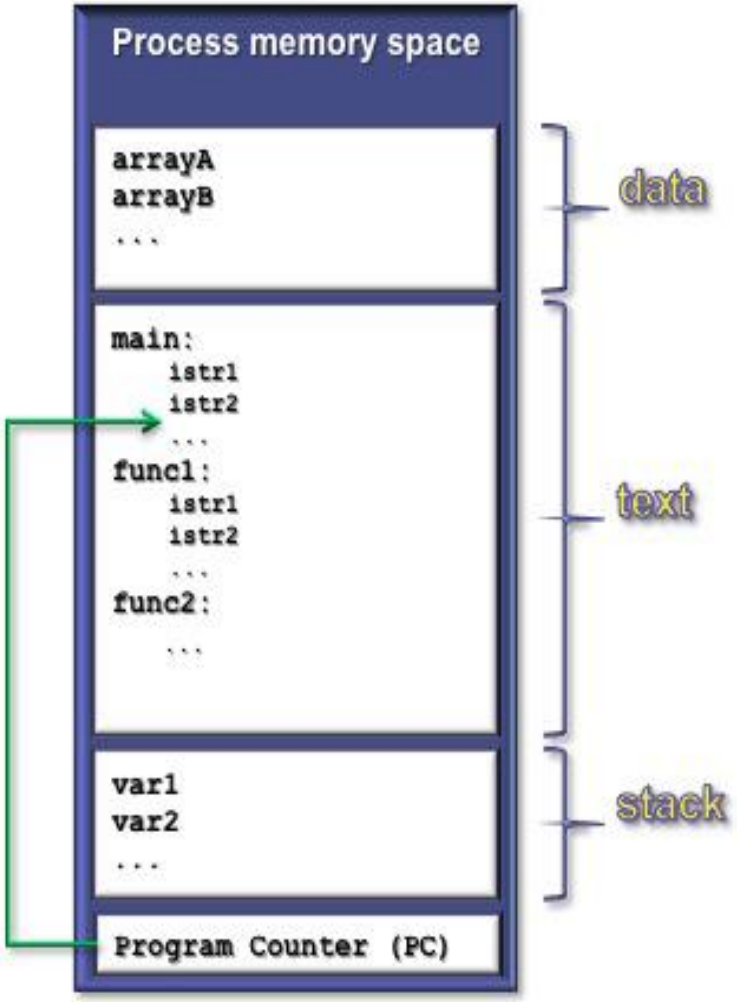


- **N**on-**U**niform **M**emory **A**ccess
  - Memory access time is non - uniform
- **U**niform **M**emory **A**ccess
  - Memory access time is uniform

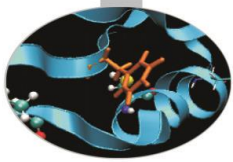
Shared memory architectures



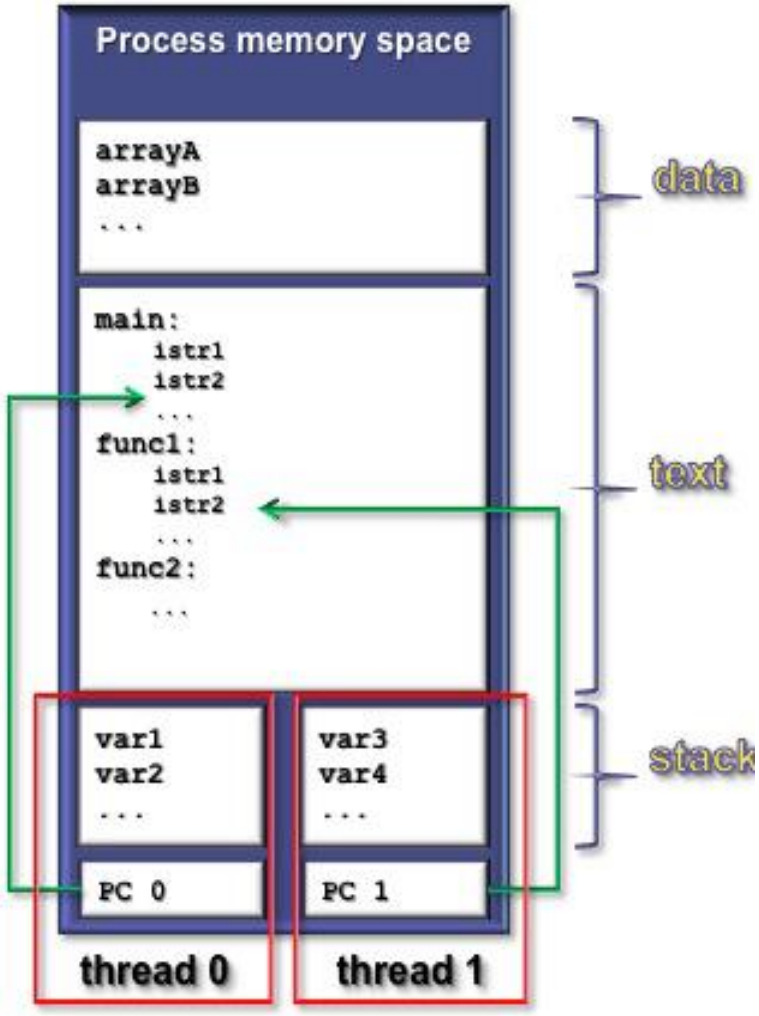
- A process is an instance of a computer program
- Some information included in a process are:
- Text
  - Machine code
- Data
  - Global variables
- Stack
  - Local variables
- Program counter (PC)
  - A pointer to the istruction to be executed



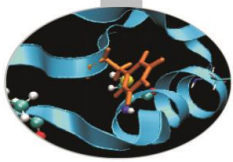
Shared memory architectures



- 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



Shared memory parallelism

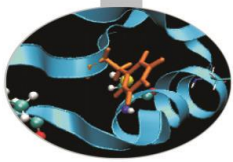


Shared memory parallel programs may be described as processes in which the execution flow is divided in different threads when needed. Threads, being generated inside a process, do share many resources, particularly all the threads have access to the process global memory.

In these programs there is of course no need of communications between threads. Parallelization may therefore be easily achieved by means of automatic tools or by placing directives in the source code.

At process activation only thread 0 (the *master* thread) is running. On entering a parallel region, the master awakes the other threads.

Writing shared memory parallel programs appears to be easier than writing message passing programs but issues often arise because of the accessibility of memory by all threads.



Shared memory parallelism

Loops are among the program constructs most often parallelized by means of directives. Anyhow in a parallelized loop the order of execution of instructions can not be pre-defined.

The following loop is not problematic:

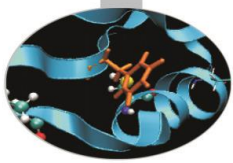
fortran
```
DO I = 1, N
            A(I) = A(I) + B(I) * C(I)
END DO
```

c/c++
```
for(i=1;i<n;i++){
        a[i] = a[i] + b[i] * c[i]
}
```

On the contrary the following loop can not be parallelized because of dependencies issues:

fortran
```
DO I = 1, N
           A(I) = A(I-1) + K * B(I)
END DO
```

c/c++
```
for(i=1;i<n;i++){
        a[i] = a[i-1] + k * b[i]
}
```



Data dependence

In the following example the i index loop can be parallelized:

fortran
```
DO I = 1, N
         DO J = 1, N
            A(J,I) = A(J-1,I) + B(J,I)
         END DO
END DO
```

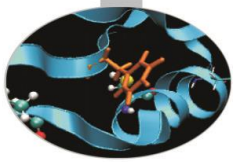
c/c++
```
For (i=1; i<n; i++){
    for (j=1 ;j<n; j++){
        a[j][i] = a[j-1][i] + b[j][i];
        }
    }
```

In this loop parallelization is dependent on the K value:

fortran
```
DO I = M, N
            A(I) = A(I-K) + B(I)/C(I)
END DO
```

c/c++
```
For (i=m; i<n; i++){
    a[i] = a[i-k] + b[i]/c[i];
    }
```

If `K > N-M` or `K < M-N` parallelization is straightforward.



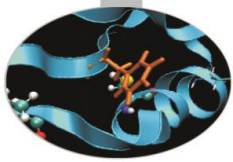
OpenMP directives

Shared memory parallelization is often realized by using directives. But directives may be compiler or platform dependent, thus contrasting portability of programs. On the contrary **OpenMP** is a well known and widely used standard consisting of directives, functions and environment variables.

OpenMP is supported and maintained by the OpenMP Architecture Review Board (Ref. http://www.openmp.org) and may be used to parallelize Fortran and C/C++ programs.

Directives are treated as comments by unaware compilers, thus a program parallelized with OpenMP directives can always be compiled and run also sequentially.

OpenMP directives



In Fortran codes all OpenMP directives are introduced by the sentinel `!$OMP` or `C$OMP`.

In C/C++ codes OpenMP directives are preceded by `#pragma omp`.

```
!$OMP                                   fortran
```

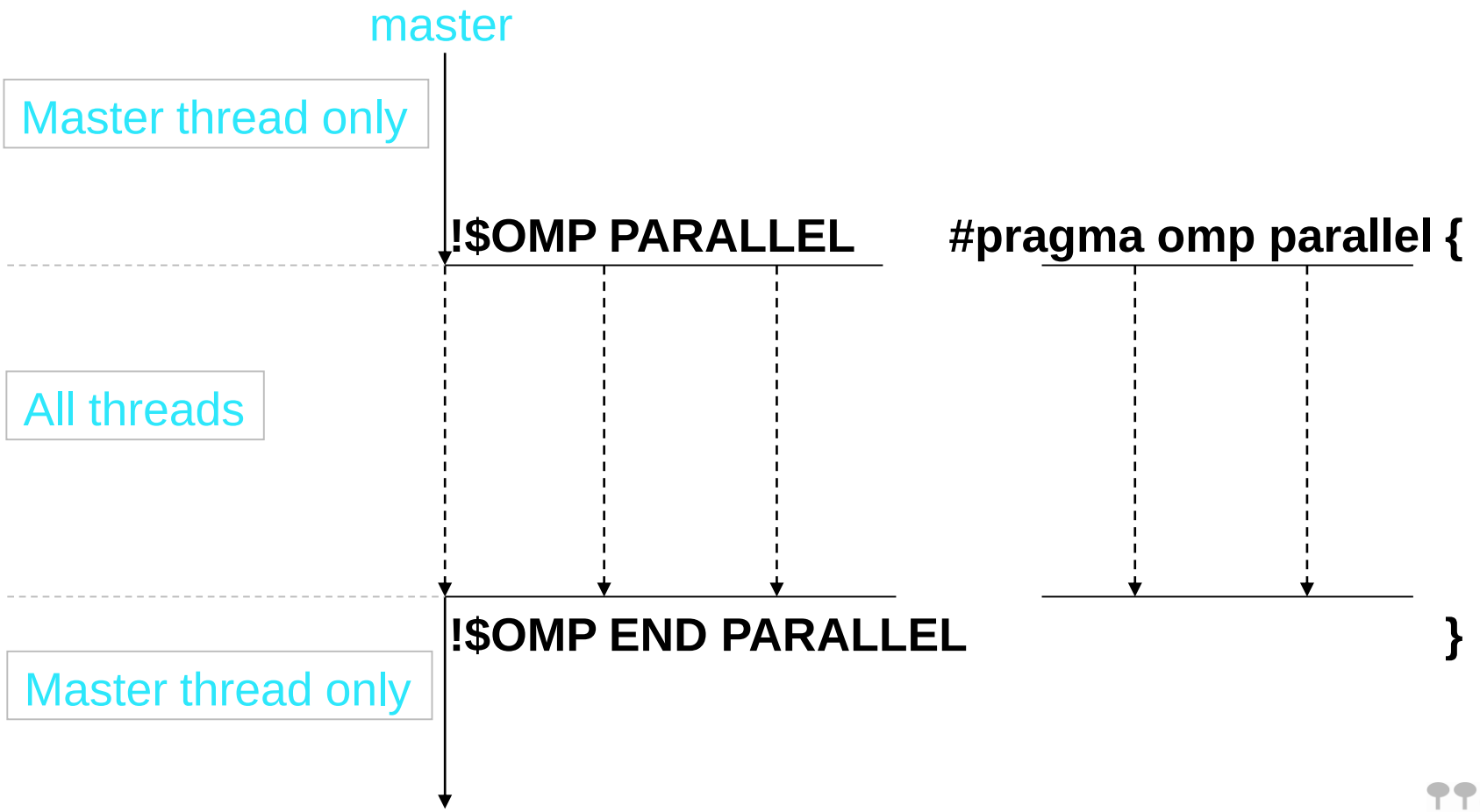
```
#pragma omp                             c/c++
```

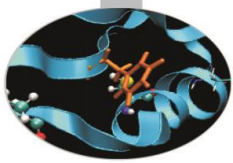
Furthermore Fortran directives are case insensitive, while C/C++ directives are case sensitive.

OpenMP programs begin execution with one thread only. Other threads are activated on entering parallel regions, delimited by specific directives. One of the most used is `!$OMP PARALLEL` / `!$OMP END PARALLEL` (Fortran) or `#pragma omp parallel { .... }` (C/C++).

Outside parallel regions execution is continued by the master thread only.

Execution model





Parallel

The `parallel` directive defines a region of code in which the instructions are executed by all threads:

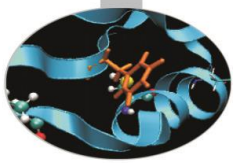
```
!$OMP PARALLEL                                    fortran
   …
!$OMP END PARALLEL
```

```
#pragma omp parallel {                            c/c++
        c/c++ instructions
      }
```

Several clauses may be used with this directive:

- **if(**scalar-expression**)**
- **num_threads(**integer-expression**)**
- **default(shared** | **none)**
- **private(**list**)**
- **firstprivate(**list**)**
- **shared(**list**)**
- **copyin(**list**)**
- **reduction(**operator**:** list**)**

Do - for



In a parallel region the directive `DO – for` may be used to distribute loops to the threads:

fortran

```
!$OMP DO
  DO I = 1, N
    fortran instructions
  END DO
!$OMP END DO
```

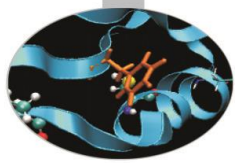
c/c++

```
#pragma omp for
  for (i=1; i<n; i++) {
      c/c++ instructions
    }
```

On exiting the loop, threads do halt, waiting for all other threads having ended their iterations, unless the **nowait** clause is used.

Care must be taken to use this directive in a *parallel* region, otherwise all the iterations of the loop will be executed by the master thread only.

Do - for



fortran

```
!$OMP PARALLEL
        .
        .
        .
 !$OMP DO [SCHEDULE(...,...)]
     DO I = 1, N
        A(I) = A(I) + B(I) * C(I)
     END DO
 !$OMP END DO [NOWAIT]
        .
        .
        .
 !$OMP END PARALLEL
```

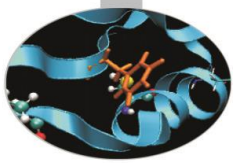
c/c++

```
#pragma omp parallel
{
    .
    .
    .
    #pragma omp for [schedule(…,…)]
    for (i=1; i<n; i++){
          a[i] = a[i] + b[i] * c[i]
          }
    .
    .
    .
}
```

In this example the loop iterations are equally distributed to the threads.

It is possible to change the distribution procedure by using the clause

schedule (type [, chunk]).



Schedule

```
schedule (type [, chunk])
```

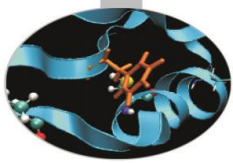
Where:

`chunk` shall be an **integer expression**.

`type` may be one of the following:

- **static** : iterations are divided in blocks with dimension *chunk*. The blocks are statically and orderly distributed to the threads in a round-robin fashion
- **dynamic** : iterations are divided in blocks with dimension *chunk*. The blocks are dinamically assigned to the free threads
- **guided** : iterations are divided in blocks with decreasing size until chunk is reached. Blocks are dinamically distributed to the threads
- **runtime** : scheduling procedure is decided before launching the execution by means of the environment variable OMP_SCHEDULE:

```
setenv OMP_SCHEDULE "type,chunk"
export OMP_SCHEDULE="type,chunk"
```



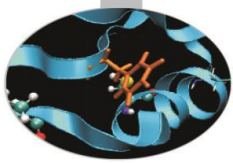
Loop Collapse

- 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

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

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

Sections



Wherever there are portions of code that can be spread among the threads the directive `sections` may be used:

fortran

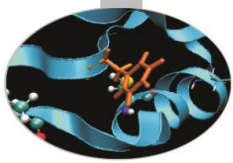
```
!$OMP SECTIONS
 !$OMP SECTION
   fortran instructions
 !$OMP SECTION
   fortran instructions
!$OMP END SECTIONS
```

c/c++

```
#pragma omp sections
{
#pragma omp section
  c/c++ instructions
#pragma omp section
  c/c++ instructions
...
}
```

Each parallel section must be preceded by the `section` directive as in the above example.

Again, care must be taken to use these directives inside a parallel region, otherwise all the sections will be executed by the master thread only.



Sections

Example - the portions PART A and PART B are executed by different threads:

fortran

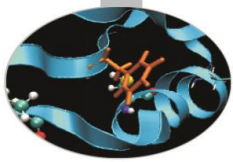
```
!$OMP PARALLEL
        .
        .
        .

 !$OMP SECTIONS

  !$OMP SECTION
    PART A
  !$OMP SECTION
    PART B
 !$OMP END SECTIONS
        .
        .
  .
 !$OMP END PARALLEL
```

c/c++

```
#pragma omp parallel
{
        .
        .
        #pragma omp sections
        {
            #pragma omp section
                PART A
            #pragma omp section
                PART B
        }
}
```

These directives are used to realize a *functional* parallelism, in which different threads execute different instructions, opposite to *data* parallelism in which threads execute the same instructions on different data sets.

Single



The `single` directive defines a portion of code that shall be executed by one thread only:

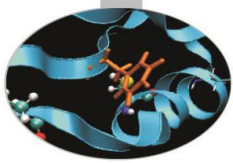
```
!$OMP SINGLE
…                                                    fortran
!$OMP END SINGLE
```

```
#pragma omp single {
      c/c++ instructions                             c/c++
     }
```

The code inside a *single* portion is executed by the first free thread. On reaching the same portion of code the other threads skip it and stay blocked until the single region has been completely executed, unless the `NOWAIT` clause is specified.

This directive must be used in parallel regions to access disk devices (i.e. open, read, write files) or may be used to update shared variables.

Single



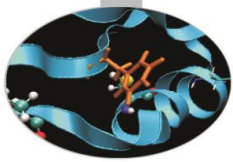
fortran

```
!$OMP PARALLEL
     …
!$OMP SINGLE
    PRINT *,'**********************'
    PRINT *,'STEP ', ij
    PRINT *,'**********************'
    T1=T/TFF
    PRINT *,'t=',T1
    WRITE(18,110)T1,RL1TES,RL25TE,RL5TES
  &,RL75TE,RL9TES,RR(IR(1)),RR(IR(NPAR))
!$OMP END SINGLE
       …
!$OMP END PARALLEL
```

c/c++

```
#pragma omp parallel
{
    ...
#pragma omp single
    {
          cout << "********************";
          cout << "Step " << ij
          cout << "********************";
          t1=t/tff
          cout << "t=" << t1
          outfile << t1 << rl1tesrl25te <<
           rl5tes << rl75te << rl9tes <<
           rr(ir[1]) << rr(ir[npar]);
    }
     ...
}
```

Master

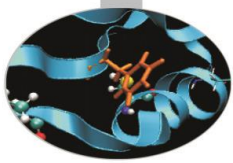


The portion of code delimited by the `master` directive is only executed by the *master* thread. The other threads simply skip it and go on without waiting.

```fortran
!$OMP MASTER

    Instructions

!$OMP END MASTER
```

```c/c++
#pragma omp master
        {
                instructions
        }
```

Parallel do – parallel for



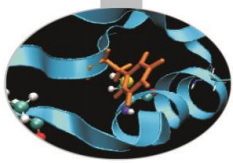
```fortran
!$OMP PARALLEL DO
    DO
        instructions
    END DO
!$OMP END PARALLEL DO
```

```c/c++
#pragma omp parallel for
     for {
         instructions
      }
```

The `parallel do/for` directive enables distribution of the iterations of a loop to the threads.

This directive does not require to be used inside a parallel region.

At the end of the code portion delimited by this directive the execution continues in a sequential mode



Parallel Sections

- The **parallel sections** construct is a shortcut for specifying a **parallel** construct containing one **sections** construct and no other statements

fortran

```
!$OMP PARALLEL SECTIONS

 !$OMP SECTION
   PART A
 !$OMP SECTION
   PART B
!$OMP END PARALLEL SECTIONS
```

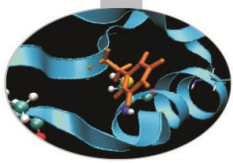
c/c++

```
#pragma omp parallel sections
{
            #pragma omp section
                 PART A
            #pragma omp section
                 PART B
}
```

This directive does not require to be used inside a parallel region.

At the end of the code portion delimited by this directive the execution continues in a sequential mode

Workshare

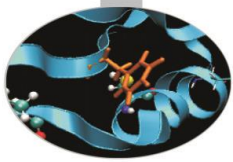


```fortran
!$OMP WORKSHARE
      VA(1:n) = VA(1:n) + VB(1:n) * K
      VC(1:m) = ( VL(1:m) + VM(1:m) ) / ( K * VL(1:m) )
!$OMP END WORKSHARE NOWAIT
```

The `WORKSHARE` directive enables distribution of iterations implied by array syntax, FORALL and WHERE constructs in Fortran only programs.

This directive must be used inside a parallel region.

If



The `IF` clause may be used with the directives `PARALLEL`, `PARALLEL SECTIONS`, `PARALLEL DO`. It depends on the value of the *condition* if the workload is distributed or not.

fortran
```
!$OMP PARALLEL DO &
!$OMP IF (((N-K)<M).OR.((M-K)>N))
   DO I = M, N
     A(I) = A(I-K) + B(I)/C(I)
   END DO
!$OMP PARALLEL END DO
```

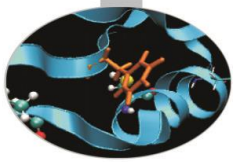
c/c++
```
#pragma omp parallel for \
         if (((n-k)<m) || ((m-k)>n))
{
    for(i>=m;i<n;i++)
        a[i] = a[i-k] + b[i]/c[i]
}
```

In the above example the loop can be parallelized only if K is in a range of values. Some time it can be useful to check if the number of iterations is high enough to have any benefit from parallelization:

fortran
```
!$OMP PARALLEL DO IF (N>1000)
DO I=1,N
  A(i)=...
END DO
!$OMP END PARALLEL DO
```

c/c++
```
#pragma omp parallel for if(n>10000)
{
    for(i=1;i<n;i++)
        a[i]=...
}
```

SHARED and PRIVATE variables

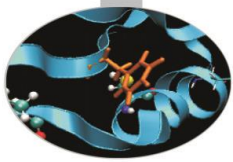


In an OpenMP program all threads have visibility of the allocated variables, unless otherwise specified. Anyhow there are variables that should not be shared by the threads. An example is a loop counter: every thread executes a different set of iterations, therefore the value of the loop counter shall not be shared. In such a case if in a parallel region a number T of threads have been activated, there will be T+1 distinct copies of the counter: one per each thread and one visible by all the threads. Therefore a loop counter shall be declared of a **private** type.

Otherwise, if a variable (scalar, matrix or other) is read only or shall be updated by all threads, it will be declared of **shared** type.

The programmer needs to pay attention to properly manage shared variables because of synchronization issues.

SHARED and PRIVATE variables



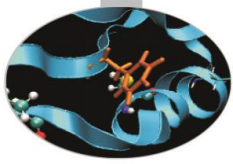
When needed shared variables must be declared in the directives that define parallel regions (`PARALLEL - parallel`, `PARALLEL DO - parallel for`, `PARALLEL SECTIONS - parallel sections`).

Private variables instead may be declared also in the directives `DO - for` and `SECTIONS - sections`.

If none is declared, all the variables are shared, unless the *default* clause is used. This clause may be used to state the type of the variables not otherwise declared. If `default(none)` is specified (which may be useful in many cases) all the variables must be explicitly declared either shared or private. If `default(private)` (Fortran only) is specified all variables are private (i.e. will be duplicated per each thread) unless explicitly declared. The clause `default(shared)` may be used too, either in Fortran or in C.

The programmer is adviced to use the clause `default(none)` in order to be sure to have analyzed all the envolved variables

SHARED and PRIVATE variables



fortran

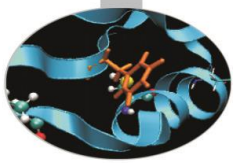
```
....
REAL, DIMENSION :: a(N)
INTEGER :: i,k
.....
!$OMP PARALLEL &
!$OMP DEFAULT(NONE) &
!$OMP SHARED (a,...) &
!$OMP PRIVATE (i,k,...)
...
!$OMP DO
  DO i=1,N
   a(i)=a(i)+k
  END DO
!$OMP END DO
.....
!$OMP END PARALLEL
```

c/c++

```
...
float a[n];
int i,k;
...
#pragma omp parallel
default(none) shared (a,...)
private(i,k,...)
{
    #pragma omp for
    for(i=1;i<n;i++)
        a[i]=a[i]+k
}
```

In the above example the loop counter (i) *must* be declared *private*. The variable k has been declared *private* too, therefore each thread will be assigned a distinct memory location to keep the value of k. Then each thread will use a *different* value, but care must be taken to properly define the value of k.

The a(:) vector instead is shared, but in this case no problem comes up because each thread will execute a different set of iterations, therefore will update distinct elements of the vector.



Subroutines and functions

Functions and subroutines may be called in a parallel region. In such a case:

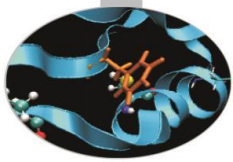
- All the activated threads will call the function
- Each variable declared in the function is private to the thread
- Dummy arguments keep their original state, i.e. are shared if they were shared

fortran

```
!$omp parallel num_threads(2) &
!$omp shared(i)
   call sub1(i)
!$omp end parallel

subroutine sub1(a)
integer :: a,b
b = a**2

...   ...

end subroutine
```

c/c++

```
#pragma omp parallel shared(i) \
 num_threads(2)
{
  sub1(i);
}

void sub1( int a) {
  int b;
  b = pow(a,2);
    ...  ...
}
```



Firstprivate

It has already pointed out that care must be paid to define the value of variables that have been stated to be private. In such a case there will be a copy of the variable shared by all the threads (that may have been defined previously) and a copy per each thread (that has never been initialized). The clause `FIRSTPRIVATE – firstprivate` may be used to initialize the value of the private copies of a variable with the value of the shared instance.

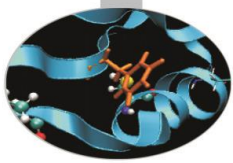
This clause is used in the «parallel» directives:

fortran

```
...
 k=a+b
!$OMP PARALLEL FIRSTPRIVATE(k)
!$OMP DO PRIVATE(i)
 DO i=1,N
   v(i)=k
   k=i+1
 END DO
!$OMP END DO
   ...
!$OMP END PARALLEL
   ...
```

c/c++

```
...
k = a+b
#pragma omp parallel firstprivate(k)
{
    #pragma omp for private(i)
    for (i=1; i<n; i++){
        v[i]=k
        k=i+1
    }
}
```



Lastprivate

The clause `LASTPRIVATE – lastprivate` can be used in parallel loops only. It may be used to copy the value relevant to the last iteration (according to a sequential execution) into the shared instance of a variable.

As an example the shared instance of variable b will have value v(N):

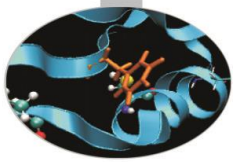
fortran

```
...
REAL, DIMENSION ::  v(N)
REAL :: b
...
!$OMP PARALLEL DO &
!$OMP PRIVATE (i) &
!$OMP LASTPRIVATE (b)
DO I = 1, N
    b=v(i)
    ...
END DO
!$OMP END PARALLEL DO
WRITE (*,*) b ! Resulting b=v(N)
```

c/c++

```
...
float v[n];
float b;
...
#pragma parallel for \
         private(i) lastprivate(b)
{
    for (i=1; i<n; i++)
         b=v[i]
}
cout << b; /* Resulting b=v[n] */
```

Threadprivate



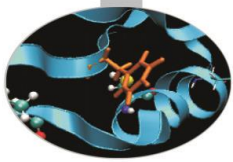
In Fortran programs the directive THREADPRIVATE may be used to create private copies of a named variable or named COMMON block. It must be placed immediately after the variable or common block declaration.

```fortran
subroutine sub(c,n)
integer :: n
real :: x,y
real, dimension(n) :: a,b
real, dimension(n,n):: c
common /dati/ a,b
!$omp THREADPRIVATE(/dati/)
do i=1,n
 a(i)=10+i
 b(i)=5-i
end do
x=5
y=6
```

```fortran
!$omp parallel do &
!$omp default(none)&
!$omp shared(c,n)&
!$omp private (i,j,x,y) &
!$omp copyin(a,b)
 do i=1,n
   do j=1,i
     a(j)=a(i)*sin(real(i))
     b(j)=b(i)*cos(real(i))
   end do
 end do
!$omp end parallel do
end
```

In the above example problems would arise in **PARALLEL DO** if A and B were not private.

The **COPYIN** clause should be used to copy the values of the shared instance of common block in the private copies.



Threadprivate

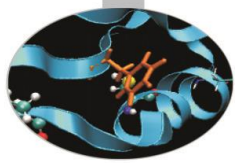
In C/C++ programs the `threadprivate` directive may be used to create private copies of *file scope* variables and *static* variables. The directive should be placed immediately after the variable declarations.

```
int counter = 0;                                c/c++
#pragma omp threadprivate(counter)

int sub()
{
counter++;
return(counter);
}
```

In the above example the `counter` variable has file scope and must be privatized.

«Threadprivate» variables differ from «private» variables because the first ones do not vanish between parallel regions.

Critical



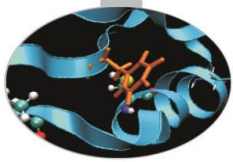
```fortran
!$OMP CRITICAL
…
!$OMP END CRITICAL
```

```c/c++
#pragma omp critical {
      istruzioni c/c++
      }
```

This directive delimits a portion of code that is executed by all threads but only one at a time:

```fortran
...
NMAX=k
!$OMP PARALLEL DO
 DO i=1,N
  if (a(i).gt.NMAX) then
     !$OMP CRITICAL
      if (a(i).gt.NMAX) then NMAX=a(i)
   !$OMP END CRITICAL
  end if
 END DO
!$OMP END PARALLEL DO
```

```c/c++
...
nmax=k
#pragma omp parallel for
for(1=1;i<n;i++){
    if(a[i]>nmax){
        #pragma omp critical
            if(a[i]>nmax)
                nmax=a[i];
    }
}
```



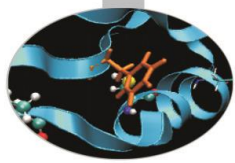
Barrier

The `BARRIER - barrier` directive defines a synchronization point in the code where threads must wait until all threads have reached the directive place.

This directive must not be positioned inside parallel loops or regions or critical sections.

Synchronization points should be used only if they are unavoidable.

As an example a barrier might be properly used after a parallel loop with *nowait* clause, before accessing variables that are updated inside the loop



Atomic

- The **`ATOMIC - 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

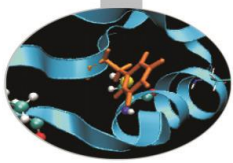
fortran

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

c/c++

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

Atomic - Examples

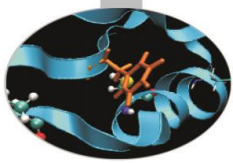


fortran

```
!$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
```

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
```



Reduction

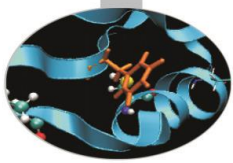
Whenever in a parallel loop a reduction operation is implemented, the `reduction` clause should be used.

Supported operations :

- **C/C++ :** +, *, -, &, |, ^, &&, || max and min operators (3.1)
- **Fortran :** +, *, -, .and., .or., .eqv., .neqv., max, min, iand, ior, ieor

The variables associated to this clause must be of **shared** type: all threads execute reduction operations on automatic local copies, which at last are used to compute the global result.

If T=threads and N=operands, in the case that T << N the reduction operation may achieve a good parallel efficiency because, in spite of the unavoidable last operation that requires a synchronization effort, only T operations have to be done sequentially while each thread executes roughly N/T operations only.



Reduction

The `reduction` clause has the following syntax:

fortran
```
reduction (operator|intrinsic: &
        reduction shared variable)
```

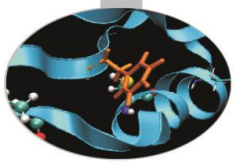
c/c++
```
reduction(operator: \
        reduction shared variable)
```

Example: using reduction to compute maximum in a vector:

fortran
```
 maxa=a(1)
!$omp parallel do &
!$omp shared (a,N)&
!$omp private (i) &
!$omp reduction (max:maxa)
 do i=1,N
   maxa=max(a(i),maxa)
 end do
!$omp end parallel do
```

c/c++
```
 max_val=arr[0];
#pragma omp parallel for \
        reduction(max : max_val)
    for( i=0;i<10; i++)
    {
        if(arr[i] > max_val)
            max_val = arr[i];
    }
```

Reduction



The previous example may be re-written as follow using critical regions with a bit of improved functionality:

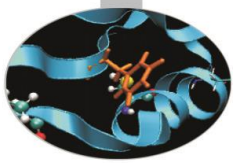
fortran

```
max_local=a(1); maxloc_local = 1
!$omp parallel private(i) &
!$omp firstprivate(max_local, &
!$omp    maxloc_local)

!$omp DO
   DO i = 1, N
      IF (a(i)> max_local) THEN
         max_local=a(i)
         maxloc_local = i
      END IF
   END DO
!$omp END DO nowait
!$omp critical
   if (max_local > maxv) THEN
       maxv = max_local
       maxloc = maxloc_local
   END IF
!$omp END critical
!$omp END parallel
```

c/c++

```
max_local=a[0]; maxloc_local = 0;
#pragma omp parallel \
shared(a,n,max, maxloc) private(i) \
firstprivate(max_local, maxloc_local)
{
   #pragma omp for nowait
   for(i=ifi; i<ila; i++)
         if (a[i]> max_local) {
                max_local=a[i];
                maxloc_local = i;
         }
   #pragma omp critical
   {
       if (max_local > max) {
            max = max_local;
            maxloc = maxloc_local;
       }
    }
}
```

Reduction



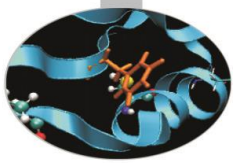
An example of sum reduction:

fortran

```
!$OMP DO
!$OMP REDUCTION(+:EKIN)&
!$OMP REDUCTION(+:ETERM)
        DO I=1,NPAR
          X=DBLE(P1(I))
          Y=DBLE(P2(I))
          Z=DBLE(P3(I))
          VX=DBLE(VF1(I))
          VY=DBLE(VF2(I))
          VZ=DBLE(VF3(I))
          M=DBLE(MS(I))
          .......


EKIN=.5*M*(VX*VX+VY*VY+VZ*VZ)+EKIN
          IF (I.LE.NSPH) THEN

ETERM=ETERM+MEN*DBLE(UF(I))
          END IF
        END DO
!$OMP END DO
```

c/c++

```
#pragma omp for reduction(+:ekin)
reduction(+:eterm)
    for(i=1;i<npar;i++){
          x=dble(p1[i]);
          y=dble(p2[i]);
          z=dble(p3[i]);
          vx=dble(vf1[i]);
          vy=dble(vf2[i]);
          vz=dble(vf3[i]);
          m=dble(ms[i]);
          ...
   ekin=.5*m*(vx*vx+vy*vy+vz*vz)+ekin;
   if(i<nsph)
       eterm=eterm+men*dble(uf(i));
    }
```



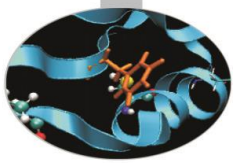
Orphaned directives

Directives that would distribute work among threads but are not placed in parallel regions are called orphaned directives.

Orphaned directives are often written in functions, which could be called from within paralell regions or not.

In the case the directive does not occur in parallel regions, execution is carried on sequentially.

Orphaned directives



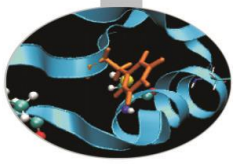
fortran

```
integer ,parameter :: N=100,M=N*100
real, dimension :: a(N)
real, dimension :: b(M)
real :: x,y
.....
do i=1,N
  a(i)=real(i)
end do
call somma (x,a,N)
!$omp parallel &
!$omp shared (b,N)&
!$omp do private(i)
do i=1,M
 b(i)=1/real(i+1)
end do
!$omp end do
```

c/c++

```
int n,m;
n=100;
m=n*100;
float a[n],b[m];
float x,y;
...
for(i=1;i<n;i++)
   a[i]=(float)i;
somma(x,a,n)
#pragma omp parallel shared(b,n)
#pragma omp for private(i)
{
for(i=1;i<n;i++)
   b[i]=1/(float)(i+1);
}
```

Orphaned directives



fortran

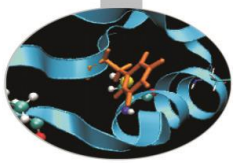
```
y=0.
call somma (y,b,M)
!$omp end parallel
....
subroutine somma(z,c,L)
integer :: i,L
real, dimension :: c(L)
real:: z
!$omp do reduction (+:z)
 do i=1,L
   z=z+c(i)
 end do
!$omp end do
end
```

c/c++

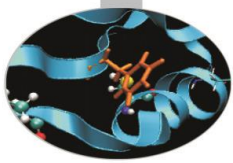
```
    y=0;
    somma(y,b,m)
}

function somma(z,c,l){
    int i,l;
    float c[l];
    float z;
    #pragma omp for reduction(+:z)
    {
          for(i=1;i<l;i++)
              z=z+c[i];
    }
}
```

At the first invocation of the function somma (`call somma(x,a)`) execution is carried on sequentially, while the latter call (`call somma(y,b)`) is executed in parallel because it is placed inside a *parallel* region.

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
  - …



New tasking construct

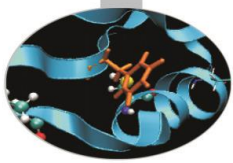
fortran

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

c/c++

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

- 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

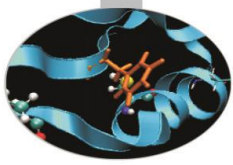
```fortran
!$omp parallel private(p)
 !$omp single
   p = head
   do while (associated(p))
     !$omp task
        call process(p)
     !$omp end task
     p => p%next
   end do
 !$omp end single
!$omp end parallel
```

fortran

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

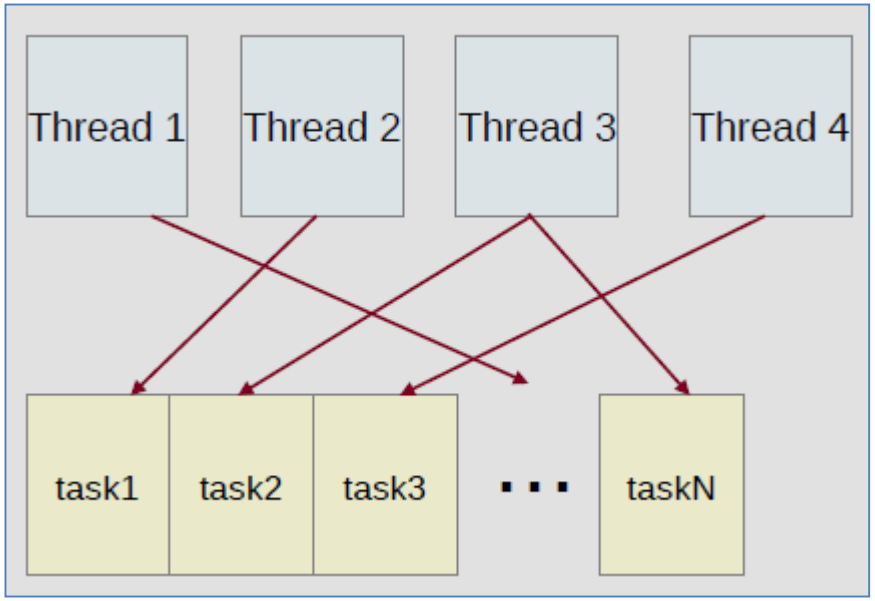
c/c++

- **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**



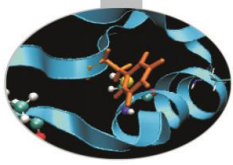
Pointer chasing using task

```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
```



TASK QUEUE

Load balancing on lists with task



fortran

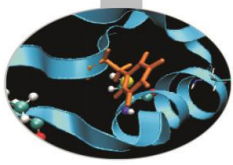
```
!$omp parallel
 !$omp do private(p)
 do i=1,num_lists
    p => head[i]
    do while (associated(p))
       !$omp task
         call process(p)
       !$omp end task
       p => p%next
    end do
 end do
 !$omp end do
!$omp end parallel
```

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;
     }
  }
}
```

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

Tree traversal with task



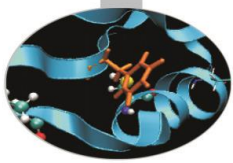
```fortran
recursive subroutine preorder(p)
 type(node), pointer :: p
 call process(p%data)
 if (associated(p%left))
   !$omp task
      call preorder(p%left)
   !$omp end task
 end if
 if (associated(p%right))
   !$omp task
      call preorder(p%right)
   !$omp end task
 end if
end subroutine preorder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

fortran

```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);
}
```

c/c++

- Tasks are composable
- It isn't a worksharing construct
- **`Taskwait`** directive suspends parent task until children tasks are completed



OpenMP functions

```
INTEGER OMP_GET_THREAD_NUM()
int omp_get_thread_num()
```

returns identity of the calling thread, i.e. a number between 0 and T-1 if T=number of threads.

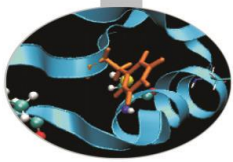
```
INTEGER OMP_GET_NUM_THREADS()
int omp_get_num_threads()
```

returns the number T of the activated threads.

```
REAL(8) OMP_GET_WTIME()
double omp_get_wtime()
```

returns the elapsed wall clock time in seconds.

OpenMP functions



```
SUBROUTINE OMP_SET_DYNAMIC ( logical dynamic_threads)
void omp_set_dynamic(int dynamic_threads)
```

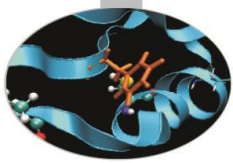
sets or disables dynamic number of threads. In order to use THREADPRIVATE directive dynamic threads should be disabled.

```
SUBROUTINE OMP_SET_NUM_THREADS(num_threads)
void omp_set_num_threads( int num_threads)
```

sets number of threads to be used in the following parallel region.

On entering a parallel region the number of threads that are activated may vary according to:

- IF clause
- NUM_THREADS clause (to be used in *parallel* directive)
- omp_set_num_threads() function
- OMP_NUM_THREADS environment variable
- Default: most often the number of processor units on a node



OpenMP functions

The sentinel `!$` must be used where calling the OpenMP functions in Fortran programs.

On the contrary in C/C++ programs these functions are called following the usual syntax, but:

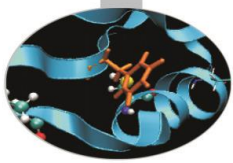
- **omp.h** file must be included in the source code
- `#ifdef _OPENMP ..... #endif` construct must be used to be able of compiling the program using OpenMP unaware compilers.

fortran

```
!$ thread_id = OMP_GET_THREAD_NUM()

!$ threads = OMP_GET_NUM_THREADS()
```

c/c++

```
#ifdef _OPENMP

threadid = omp_get_thread_num()

threads = omp_get_num_threads()

#endif
```



Program compilation

Compiling OpenMP programs in Linux

Intel compilers:

```
ifort   -openmp -O3 -o nomefile.exe nomefile.f90
icpc    -openmp -O3 -o nomefile.exe nomefile.cpp
icc    -openmp -O3 -o nomefile.exe nomefile.c
```

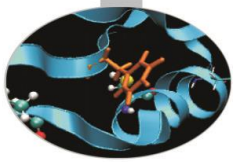
PGI compilers:

```
pgf90    -mp -O3 -o nomefile.exe nomefile.f90
pgCC    -mp -O3 -o nomefile.exe nomefile.cpp
pgcc    -mp -O3 -o nomefile.exe nomefile.c
```

GNU compilers:

```
gfortran   -fopenmp -O3 -o nomefile.exe nomefile.f90
c++    -fopenmp -O3 -o nomefile.exe nomefile.cpp
gcc    -fopenmp -O3 -o nomefile.exe nomefile.c
```

Program execution



There are no peculiar manners for launching execution of OpenMP programs. The only thing that is worth while considering is the opportunity of defining the value of a few environmental variables.

Two of these might often be taken into consideration:

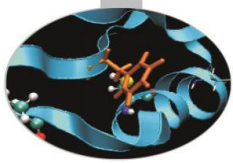
Defining the number fo threads (most useful when running on nodes with many cores):

```
setenv OMP_NUM_THREADS number_of_threads (tcsh  shell)
export OMP_NUM_THREADS=number_of_threads (bash  shell)
```

Defining workload distribution method:

```
setenv OMP_SCHEDULE “type,chunk” (tcsh shell)
export OMP_SCHEDULE=“type,chunk” (bash shell)
```

Bibliography



OpenMP official site:

- http://openmp.org/wp/
- http://openmp.org/wp/openmp-specifications/
- http://openmp.org/wp/2015/05/openmp-40-examples-published/

OpenMP, *Blaise Barney, Lawrence Livermore National Laboratory*

- https://computing.llnl.gov/tutorials/openMP/

"Using OpenMP" - Portable Shared Memory Parallel Programming, Chapman, Jost, van der Pas - MIT Press, 2008 - ISBN-10: 0-262-53302-2, ISBN-13: 978-0-262-53302-7

Parallel computing with Fortran modules and OpenMP directives: *Panoramica sulle tecnologie e sugli strumenti per la programmazione parallela (I parte)*, G. Bottoni, M. Cremonesi, Bollettino del CILEA, N. 73