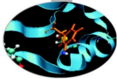
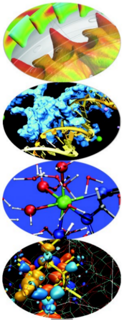


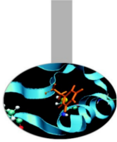
Introduction to OpenMP

Mirko Cestari - `m.cestari@cineca.it`

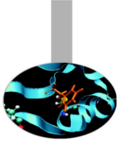
CINECA - SuperComputing Applications and Innovation Department



Outline

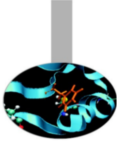


Disadvantages of MPI



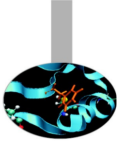
- Each MPI process can only access its local memory
  - The data to be shared must be exchanged with explicit inter-process communications (messages)
  - It is the responsibility of the programmer to design and implement the exchange of data between processes
- You can not adopt a strategy of incremental 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?



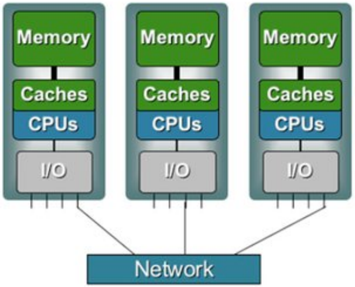
- De-facto **standard** Application Program Interface (API) to write shared memory parallel applications in C, C++ and Fortran
- Consists of **compilers directives**, **run-time routines** and **environment variables**
- "Open specifications for Multi Processing" maintained by the OpenMP Architecture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

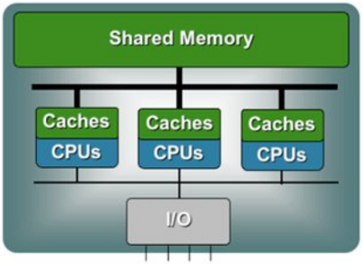
History



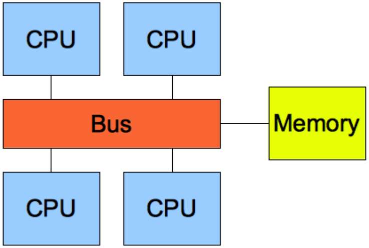
- Born to satisfy the need of unification of proprietary solutions
- **The past**
  - October 1997 - Fortran version 1
  - October 1998 - C/C++ version 1
  - November 1999 - Fortran version 1.1 (interpretations)
  - November 2000 - Fortran version 2
  - March 2002 - C/C++ version 2
  - May 2005 - combined C/C++ and Fortran version 2
  - May 2008 - version 3.0
- **The present**
  - July 2011 - version 3.1
  - July 2013 - version 4.0
  - November 2015 - version 4.5
- **The future**
  - version 5.0

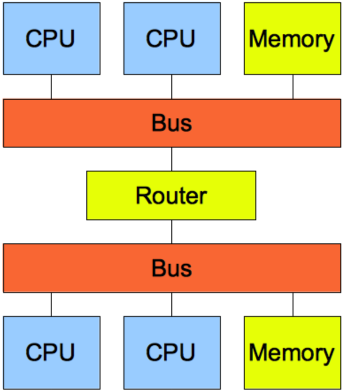
Distributed and shared memory





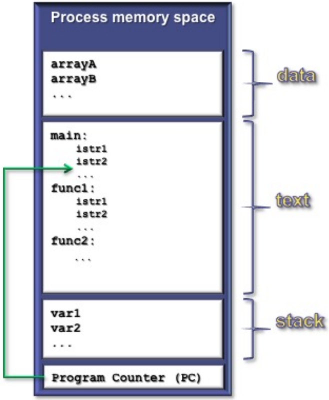
UMA and NUMA systems





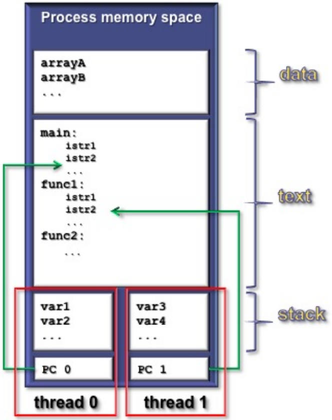
Process and thread

- 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 istruction to be executed

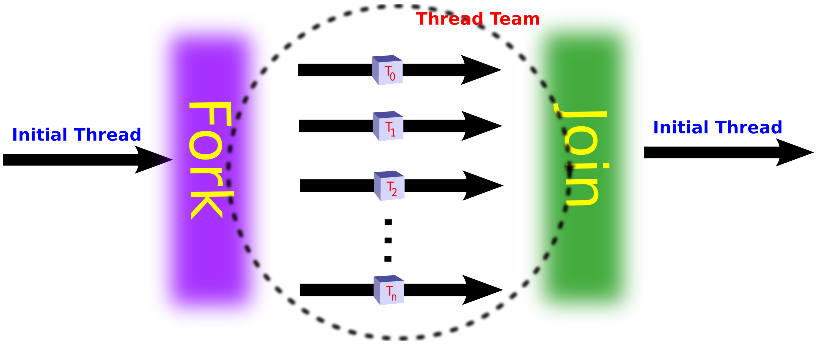


Multi-threaded processes

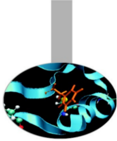
- The process contains several concurrent execution flows (threads)
  - Each thread has its own program counter (PC)
  - Each thread has its own private stack (variables local to the thread)
  - The instructions executed by a thread can access:
    - the process global memory (data)
    - the thread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



Execution model

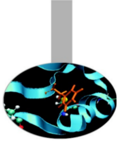


Why should I use OpenMP?

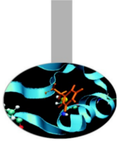


1. **Standardized**
   - enhance portability
2. **Ease of use**
   - limited set of directives
   - fast code parallelization
   - parallelization is incremental
   - coarse/fine parallelism
3. **Portability**
   - `C`, `C++` and `Fortran` API
   - part of many compilers

OpenMP (possible) issues



1. **Performance**
   - may be non-portable
   - increase memory traffic
2. **Limitations**
   - shared memory systems
   - mainly used for loops

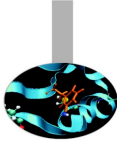


Structure of an OpenMP program

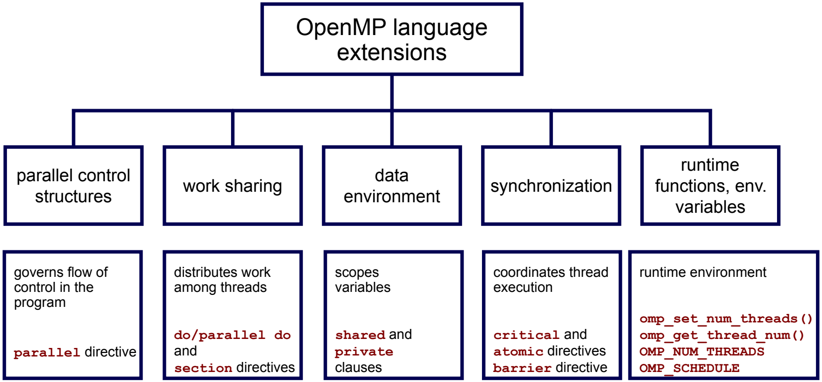
1. **Execution model**
   - the program starts with an initial thread
   - when a `parallel` construct is encountered a team is created
   - `parallel` regions may be nested arbitrarily
   - worksharing constructs permit to divide work among threads

2. **Shared-memory model**
   - all threads have access to the memory
   - each thread is allowed to have a temporary view of the memory
   - each thread has access to a thread-private memory
   - two kinds of data-sharing attributes: private and shared
   - data-races trigger undefined behavior

3. **Programming model**
   - compiler directives + environment variables + run-time library

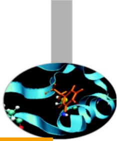


OpenMP core elements



OpenMP language extensions
parallel control structures
work sharing
data environment
synchronization
runtime functions, env. variables
governs flow of control in the program
parallel directive
distributes work among threads
do/parallel do and section directives
scopes variables
shared and private clauses
coordinates thread execution
critical and atomic directives
barrier directive
runtime environment
omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

Conditional compilation



C/C++

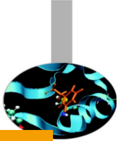
```
#ifdef _OPENMP
printf("OpenMP support:%d",_OPENMP);
#else
printf("Serial execution.");
#endif
```

Fortran

```
!$ print *,"OpenMP support"
```

1. The macro `_OPENMP` has the value `yyyymm`
2. `Fortran 77` supports `!$`, `*$` and `c$` as sentinels
3. `Fortran 90` supports `!$` only

Directive format



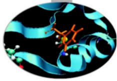
C/C++

```
#pragma omp directive-name [clause...]
```

Fortran

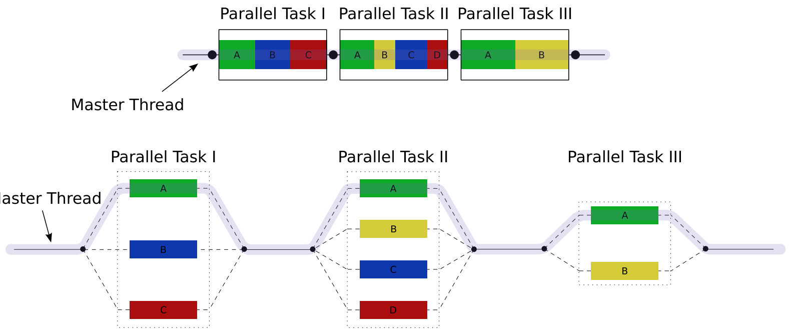
```
sentinel directive-name [clause...]
```

1. Follows conventions of `C` and `C++` compiler directives
2. From here on free-form directives will be considered



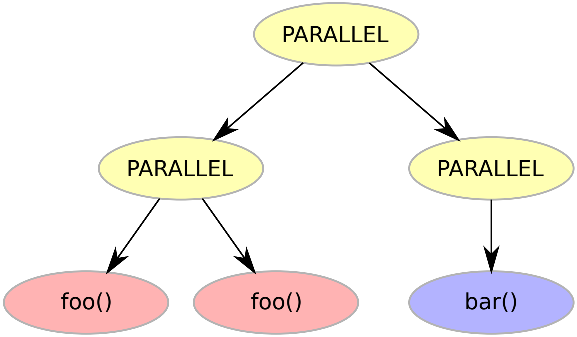
Outline

`parallel` construct



1. The encountering thread becomes the master of the new team
2. All threads execute the parallel region
3. There is an implied barrier at the end of the parallel region

Nested parallelism



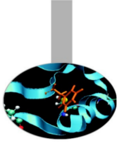
1. Nested parallelism is allowed in OpenMP 3.1
2. Most constructs bind to the innermost parallel region

OpenMP: Hello world

C/C++

```
int main () {



    printf("Hello world\n");



  return 0;
}
```

OpenMP: Hello world



C/C++

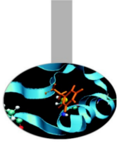
```
int main () {
  /* Serial part */

#pragma omp parallel
  {
    printf("Hello world\n");
  }


  /* Serial part */
  return 0;
}
```

OpenMP: Hello world

Fortran

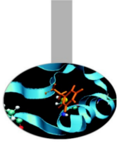
```
PROGRAM HELLO


   Print *, "Hello World!!!"



END PROGRAM HELLO
```

OpenMP: Hello world



Fortran

```
PROGRAM HELLO
! Serial code


!$OMP PARALLEL
   Print *, "Hello World!!!"
!$OMP END PARALLEL


! Resume serial code

END PROGRAM HELLO
```

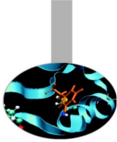


OpenMP: Hello world

What's wrong?

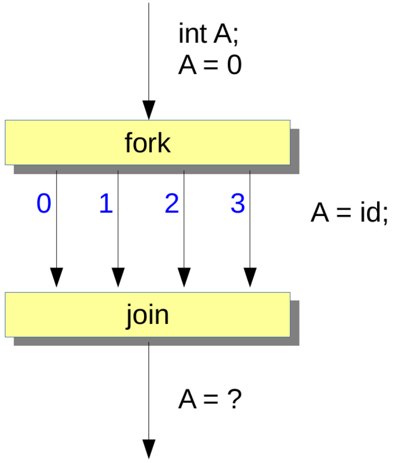
```
int main() {
  int i;
#pragma omp parallel
  {
    for(i = 0; i < 10; ++i)
      printf("iteration %d\n", i);
  }
  return 0;
}
```

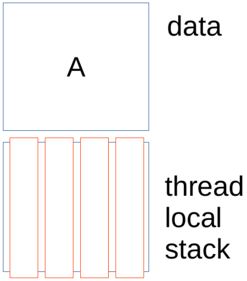
Race condition



- 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

Race condition

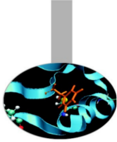




Worksharing constructs: rules

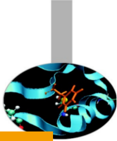
1. Distribute the execution of the associated region
2. A worksharing region has no barrier on entry
3. An implied barrier exists at the end, unless `nowait` is specified
4. Each region must be encountered by all threads or none
   - Every thread must encounter the same sequence of worksharing regions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

Worksharing constructs: types



- The OpenMP API defines four worksharing constructs:
  - **loop**
  - **sections**
  - **single**
  - **workshare**

Loop construct: syntax



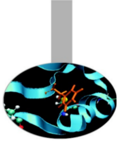
C/C++

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

Fortran

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

Loop construct: restrictions



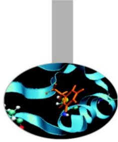
1. Only loops with canonical forms are allowed
2. The iteration count needs to be computed before executing the loops
3. incr-expr: addition or subtraction expression.

Wrong loop construct

wrong incremental expression

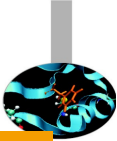
```
#include <stdio.h>
#include <omp.h>

void incr(int *var){
    *var = *var + 1;
}
void main(){
    int a;
#ifdef _OPENMP
    #pragma omp parallel for
#endif
    for (a=0;a<10;incr(&a))
        printf("%d\n", a);
}
```

Loop construct: the rules



1. The iterations of the loop are **distributed** over the threads that already exist in the team
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. The `schedule` clause:
   - may be used to specify how iterations are divided into chunks
4. The `collapse` clause:
   - may be used to specify how many loops are parallelized
   - valid values are constant positive integer expressions

Loop construct: scheduling



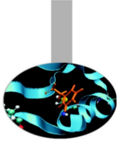
C/C++

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

Fortran

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

Loop construct: schedule kind



1. **static**
   - iterations are divided into chunks of size `chunk_size`
   - the chunks are assigned to the threads in a round-robin fashion
   - must be reproducible within the same parallel region
2. **dynamic**
   - iterations are divided into chunks of size `chunk_size`
   - the chunks are assigned to the threads as they request them
   - the default `chunk_size` is 1
3. **guided**
   - iterations are divided into chunks of decreasing size
   - the chunks are assigned to the threads as they request them
   - `chunk_size` controls the minimum size of the chunks
4. **auto**
   - When the runtime can "learn" from previous executions of the same loop

Loop construct: schedule kind

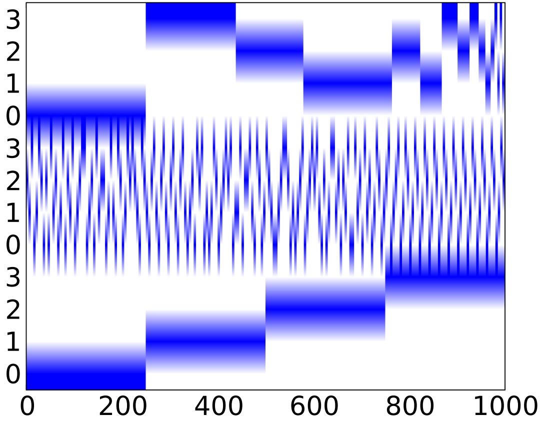
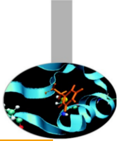


Figure : Different scheduling for a 1000 iterations loop with 4 threads: guided (top), dynamic (middle), static (bottom)

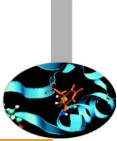
Loop construct: nowait clause

Where are the implied barriers?

```
void nowait_example(int n, int m, float *a,
  float *b, float *y, float *z) {
#pragma omp parallel
  {
#pragma omp for
    for (int i=1; i<n; i++)
      b[i] = (a[i] + a[i-1]) / 2.0;
#pragma omp for
    for (int i=0; i<m; i++)
      y[i] = sqrt(z[i]);
  }
}
```



Loop construct: nowait clause



Where are the implied barriers?

```
void nowait_example(int n, int m, float *a,
  float *b, float *y, float *z) {
#pragma omp parallel
  {
#pragma omp for nowait
    for (int i=1; i<n; i++)
      b[i] = (a[i] + a[i-1]) / 2.0;
#pragma omp for nowait
    for (int i=0; i<m; i++)
      y[i] = sqrt(z[i]);
  }
}
```

Loop construct: nowait clause

Is the following snippet semantically correct?

```
...
int i;
#pragma omp parallel
  {
#pragma omp for
    for (i=0; i<n; i++)
      c[i] = (a[i] + b[i]) / 2.0f;
#pragma omp for
    for (i=0; i<n; i++)
      z[i] = sqrtf(c[i]);
#pragma omp for
    for (i=1; i<=n; i++)
      y[i] = z[i-1] + a[i];
  }
...
```

Loop construct: nowait clause

```
c[i] = (a[i] + b[i]) / 2.0f;
// ...
z[i] = sqrtf(c[i]);
// ...
y[i] = z[i-1] + a[i];
```

	th 0	knows	for
c[i]	i=0,...,9		i=0,...,n-1
z[i]	i=?	c[0,...,9]	i=0,...,n-1
y[i]	i=?	z[0,...,9]	i=1,...,n

Loop construct: nowait clause

Is the following snippet semantically correct?

```
...
int i;
#pragma omp parallel
  {
#pragma omp for schedule(static) nowait
    for (i=0; i<n; i++)
      c[i] = (a[i] + b[i]) / 2.0f;
#pragma omp for schedule(static) nowait
    for (i=0; i<n; i++)
      z[i] = sqrtf(c[i]);
#pragma omp for schedule(static) nowait
    for (i=1; i<=n; i++)
      y[i] = z[i-1] + a[i];
  }
...
```

Loop construct: nested loops

Am I allowed to do the following?

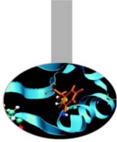
```
#pragma omp parallel
{
#pragma omp for
  for(int ii = 0; ii < n; ii++) {
#pragma omp for
    for(int jj = 0; jj < m; jj ++) {
      A[ii][jj] = ii*m + jj;
    }
  }
}
```

Loop construct: collapse clause

The right way to collapse nested loops

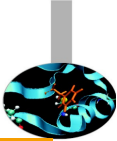
```
#pragma omp parallel
{
#pragma omp for collapse(2)
  for(int ii = 0; ii < n; ii++) {
    for(int jj = 0; jj < m; jj ++) {
      A[ii][jj] = ii*m + jj;
    }
  }
}
```

Loop collapse



- Allows parallelization of perfectly nested rectangular loops
- The collapse clause indicates how many loops should be collapsed
- 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.

Loop dependencies

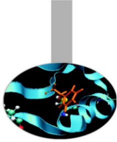


loop carried dependencies

```
int i, j, ARR[N];
j = 3;
for (i=0; i<N; i++) {
    j+=2;
    ARR[i] = func(j);
}
```

i	j
0	3+2
1	3+4
2	3+6
...	...
n	3+(2*n+2)

Removing loop dependencies



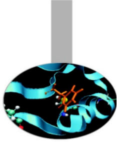
removig loop carried dependencies

```
int i, j, ARR[N];
for (i=0; i<N; i++) {
    j = 3+(2*i+2);
    ARR[i] = func(j);
}
```

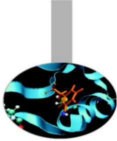
Sections construct: syntax

C/C++

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



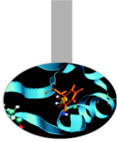
Sections construct: syntax



Fortran

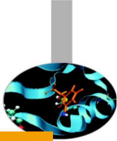
```
!$omp sections [clause[[,] clause]...]
!$omp section
  structured-block
!$omp section
  structured-block
...
!$omp end sections [nowait]
```

Sections construct: some facts



1. `sections` is a non-iterative worksharing construct
   - it contains a set of `structured-blocks`
   - each one is executed once by 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: syntax



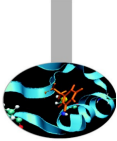
C/C++

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

Fortran

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

Single construct: some facts



1. The associated structured block is executed by only one thread
2. The other threads wait at an implicit barrier
3. The method of choosing a thread is implementation defined

Workshare construct: syntax

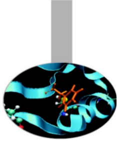
Fortran

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

Divides the following into shared units of work:

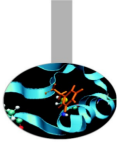
1. array assignments
2. `FORALL` statements or constructs
3. `WHERE` statements or constructs

Reduction clause: some facts



1. The `reduction` clause:
   - is valid on `parallel`, `loop` and `work-sharing` constructs
   - specifies an operator and one or more list items
2. A list item that appears in a `reduction` clause must be shared
3. For each item in the list:
   - a local copy is created and initialized appropriately based on the reduction operation (e.g * -> 1)
   - updates occur on the local copy.
   - local copies are reduced into a single value and combined with the original global value.
4. Items must not be const-qualified

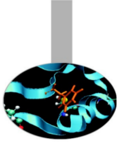
Reduction clause: example



Sum over many iterations

```
int a = 5;
#pragma omp parallel
{
#pragma omp for reduction(+:a)
  for(int i = 0; i < 10; ++i)
    ++a;
}
printf("%d\n", a);
```

Outline



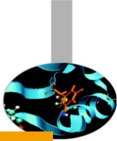
Master construct: syntax

C/C++

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

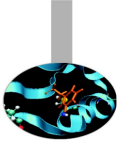
Fortran

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



Master construct: some facts

1. The `master` construct specifies a structured block:
   - that is executed by the master thread
   - with no implied barrier on entry or exit
2. Used mainly in:
   - hybrid `MPI-OpenMP` programs
   - progress/debug logging



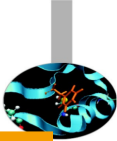
Critical construct: syntax

C/C++

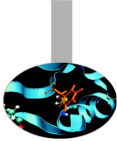
```
#pragma omp critical [name]
  structured-block
```

Fortran

```
!$omp critical [name]
  structured-block
!$omp end critical [name]
```

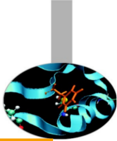


Critical contruct: some facts



1. The `critical` construct restricts the execution:
   - to a single thread at a time (**wait on entry**)
2. An optional name may be used to identify a region
3. All `critical` without a name share the same unspecified tag
4. In `Fortran` the names of `critical` constructs:
   - are global entities of the program
   - may conflict with other names (and trigger undefined behavior)

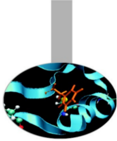
Critical construct: example



Named critical regions

```
#pragma omp parallel
{
#pragma omp critical(long_critical_name)
  doSomeCriticalWork_1();
#pragma omp critical
  doSomeCriticalWork_2();
#pragma omp critical
  doSomeCriticalWork_3();
}
```

Barrier construct: syntax



C/C++

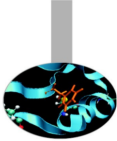
```
#pragma omp barrier
```

Fortran

```
!$omp barrier
```

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

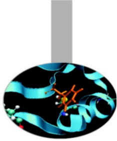
Barrier construct: example



Waiting for the master

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

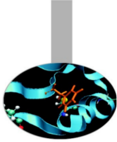
Atomic construct: syntax



C/C++

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

#pragma omp atomic capture
  structured-block
```

Atomic construct: syntax



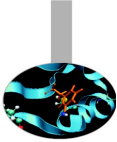
Fortran

```
!$omp atomic read
  capture-statement
[!$omp end atomic]

!$omp atomic write
  write-statement
[!$omp end atomic]
```

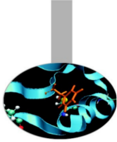
Atomic construct: syntax

Fortran

```
!$omp atomic [update]
  update-statement
[!$omp end atomic]

!$omp atomic capture
  update-statement
  capture-statement
!$omp end atomic
```



Atomic construct: some facts



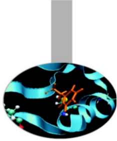
1. The `atomic` construct:
   - ensures a specific storage location to be updated atomically
   - does not expose it to multiple, simultaneous writing threads
2. The binding thread set for an atomic region is all threads
3. The `atomic` construct with the clause:

`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

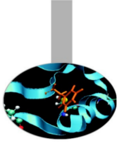
4. Accesses to the same location must have compatible types

SuperComputing Applications and Innovation

Outline

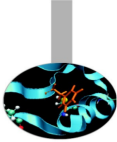


Data-sharing attributes: C/C++



1. The following are always shared:
   - objects with dynamic storage duration
   - variables with static storage duration
   - file scope variables
2. The following are always private:
   - loop iteration variable in the loop construct
   - variables with automatic storage duration
3. Arguments passed by reference inherit the attributes

Data-sharing attributes: Fortran



1. The following are always private:
   - variables with automatic storage duration
   - loop iteration variable in the loop construct
2. The following are always shared:
   - assumed size arrays
   - variables with save attribute
   - variables belonging to common blocks or in modules
3. Arguments passed by reference inherit the attributes

Data-sharing clauses: syntax

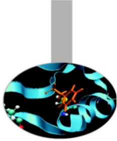
C/C++

```
#pragma omp ... shared(...) private(...)
```

Fortran

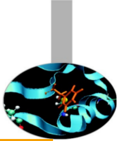
```
!$omp ...  shared(...) private(...)
...
!$omp end  ...
```

Default/shared/private clauses



1. The clause `default`:
   - is valid on `parallel`
   - accepts only `shared` or `none` in `C/C++` and `Fortran`
   - accepts also `private` and `firstprivate` in `Fortran`
   - `default(none)` **requires** each variable to be listed in a clause
2. The clause `shared(list)`:
   - is valid only on `parallel` contruct
   - declares one or more list items to be shared
3. The clause `private(list)`:
   - is valid on `parallel`, and worksharing contructs
   - declares one or more list items to be private
   - allocates a new item of the same type with undefined value

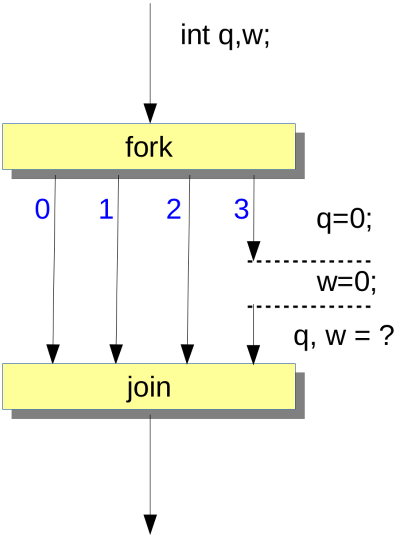
Default/shared/private clauses

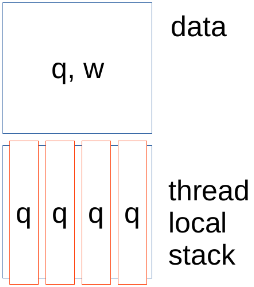


Example

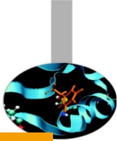
```
int q,w;
#pragma omp parallel private(q) shared(w)
{
  q = 0;
#pragma omp single
  w = 0;
#pragma omp critical(stdout_critical)
  printf("%d %d\n", q, w);
}
```

Firstprivate clause





Firstprivate clause

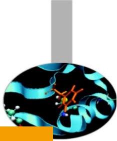


Example

```
int q = 3, w;
#pragma omp parallel firstprivate(q) shared(w)
{
#pragma omp single
  w = 0;
#pragma omp critical(stdout_critical)
  printf("%d %d\n", q, w);
}
```

Same as `private`, but initializes items

Lastprivate clause

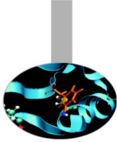


Example

```
#pragma omp parallel
{
#pragma omp for lastprivate(i)
  for(i = 0; i < (n1); ++i)
    a[i] = b[i] + b[i + 1];
}
a[i] = b[i];
```

1. valid on `for`, `sections`
2. the value of each new list item is the sequentially last value

Copyprivate clause

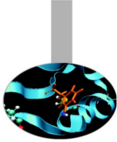


C/C++

```
#pragma omp single copyprivate(tmp)
{
  tmp = (float *) malloc(sizeof(float));
} /* copies the pointer only */
```

1. Valid only on `single`
2. Broadcasts the value of a private variable

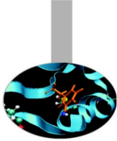
False sharing



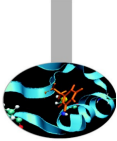
- say we have array elements contiguous in memory
- if independent data elements are on the same cache line threads might share the same cache line
- each update on one element will cause the cache lines of the remaining threads to be trashed
- this is called **false sharing**
- poor scalability

- Solution:
  - When updates to an item are frequent, work with local copies of data instead of an array indexed by the thread ID.
  - Pad arrays so elements you use are on distinct cache lines.

Outline



Runtime library routines

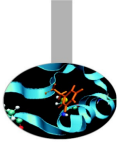


Most used functions

```
int omp_get_num_threads(void);// # of threads
int omp_get_thread_num(void);// thread id
double omp_get_wtime(void);// get wall-time
```

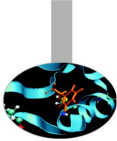
1. Prototypes for `C/C++` runtime are provided in `omp.h`
2. Interface declarations for `Fortran` are provided as:
   - a `Fortran` include file named `omp_lib.h`
   - a `Fortran 90` module named `omp_lib`

Environment variables

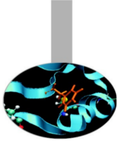


OMP_NUM_THREADS	sets the number of threads for parallel regions
OMP_STACKSIZE	specifies the size of the stack for threads
OMP_SCHEDULE	controls schedule type and chunk size of `runtime`
OMP_PROC_BIND	controls whether threads are bound to processors
OMP_NESTED	enables or disables nested parallelism

Outline



OpenMP Compilers



GNU:

(Version $>=$ 4.3.2) Compile with **-fopenmp** For Linux, Solaris, AIX, MacOSX, Windows.

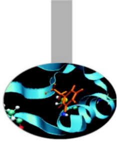
IBM:

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

Intel:

Compile with **-Qopenmp** on Windows, or **-qopenmp** on Linux or Mac

OpenMP Compilers



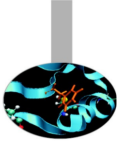
Sun Microsystems:

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

Portland Group Compilers:

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

OpenMP: THE END!!!



Good luck and enjoy OpenMP!!!