Introduction to OpenMP

Gian Franco Marras[1] Massimiliano Culpo[1]

[1]CINECA - SuperComputing Applications and Innovation Department - SCAI, Via Magnanelli 6/3, 40033 Casalecchio di Reno, Bologna (Bo),
g.marras@cineca.it,m.culpo@cineca.it

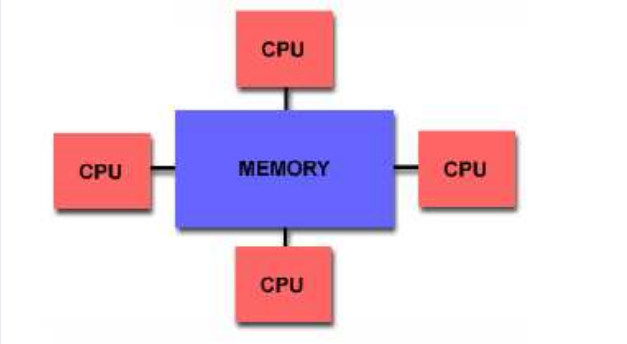
October 25, 2012

Introduction

Shared Memory System

Shared memory:

- Refers to a large block of RAM that can be accessed by several different CPUs in a multiple-processor computer system.
- Usually the system is a Simmetric MultiProcessor (SMP). SMP involves a multiprocessor computer hardware architecture where two or more identical processors are connected to a single shared main memory.



OpenMP

- The OpenMP API provides a relaxed-consistency, **shared-memory model**.
- All OpenMP threads have access to a place to store and to retrieve variables, called the **memory**.
- Each thread also has access to another type of memory that must not be accessed by other threads, called **threadprivate memory**.
- The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms.
- OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

Pros & Cons

Pros

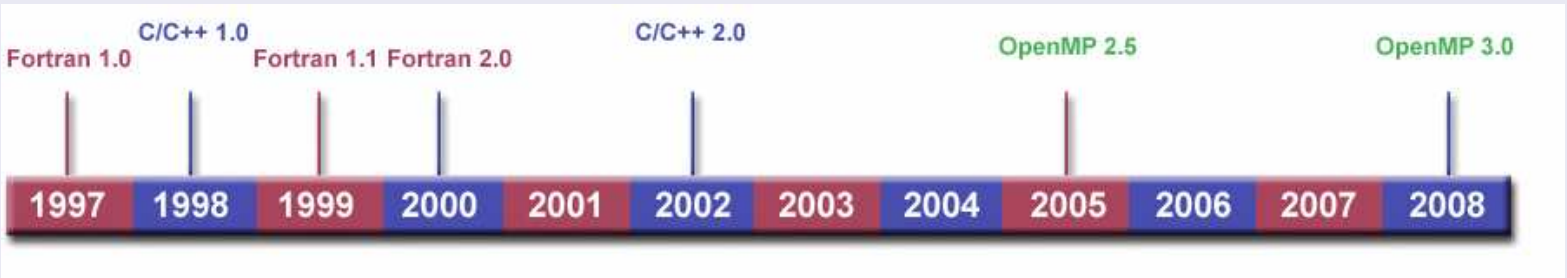
- easier to program and debug;
- directives can be added incrementally - gradual parallelization;
- can still run the program as a serial code;
- serial code statements usually don't need modification.

Cons

- can only be run in shared memory computers;
- mostly used for loop parallelization;
- traffic between CPU and memory increases with the number of CPUs;

Releases

- October 1997: Fortran 1.0;
- 1998: C/C++ 1.0;
- June 2000: Fortran 2.0;
- April 2002: C/C++ 2.0;
- 2008-2009: 3.0 Fortran and C/C++.



Execution model

OpenMP consists of a set of:

- Compiler directives;
- Runtime library routines;
- Environment variables.

- The OpenMP API uses the **fork-join** model of parallel execution.
- An OpenMP program begins as a single thread of execution, called the **initial thread**. The initial thread executes sequentially until encounters a parallel construct.
- The initial thread creates a team of threads and becomes the **master** of the new team. Beyond the end of the parallel construct, only the master thread resume execution.

Conditional Compilation

In implementations that support a preprocessor, the _OPENMP macro name is defined to have the decimal value yyyymm where yyyy and mm are the year and month designations of the version of the OpenMP API that the implementation supports.

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
```

GNU:

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

IBM:

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

Sun Microsystems:

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

Intel:

Compile with **-Qopenmp** on Windows, or just **-openmp** on Linux or Mac Emit useful information to stderr. **-openmp-report2**

Portland Group Compilers:

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

Directives

OpenMP directives for C/C++ are specified with the pragma preprocessing directive.
The syntax of an OpenMP directive is formally specified as follows:

C/C++:

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

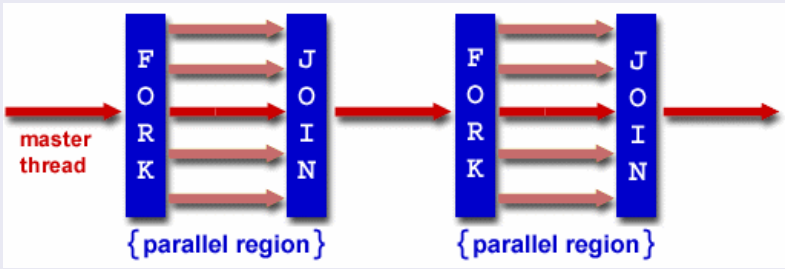
OpenMP directives for Fortran are specified as follows:

Fortran:

```
$omp directive-name [clause[[,]clause]...]
```

Parallel Construct

- Start **parallel execution**;
- A **team of threads** is created to execute the parallel region;
- The thread that encountered the parallel construct becomes the **master thread** of the new team with a thread number zero.
- There is an **implicit barrier** at the end of the construct;
- Within a parallel region, **thread numbers** uniquely identify each thread. Thread numbers are consecutive whole numbers ranging from zero for the master thread up to one less than the number of threads in the team.



A first program in Fortran:

```
PROGRAM HELLO
INTEGER VAR1, VAR2, VAR3
!Serial code


  Print *, "Hello World!!!"


!Resume serial code
END
```

A first program in Fortran:

```
PROGRAM HELLO
INTEGER VAR1, VAR2, VAR3
!Serial code

!$OMP PARALLEL

  Print *, "Hello World!!!"

!$OMP END PARALLEL

!Resume serial code
END
```

A first program in C:

```
int main ()
{
  int var1, var2, var3;
  Serial code
!  Beginning of parallel region. Fork a team of threads.
! Specify variable scoping


 {
 printf(``Hello world\n'');
 }
}
```

A first program in C:

```
int main ()
{
  int var1, var2, var3;
  Serial code
!  Beginning of parallel region. Fork a team of threads.
! Specify variable scoping

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

Worksharing Construct

- A worksharing construct distributes the execution of the associated region among the members of the team that encounters 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:
  - loop construct;
  - **sections** construct;
  - **single** construct;
  - **workshare** contruct.

Loop construct

The **loop construct** specifies that the iterations of one or more associated loops will be executed in parallel by threads in the team. **The iterations are distributed across threads** that already exist in the team executing the parallel region to which the loop region binds.

C/C++:

```
#pragma omp for [clause[[,] clause] ... ]
  for(i=0;...)
```

Fortran:

```
!$omp do [clause[[,] clause] ... ]
do i=0,n
 ...
enddo
[!$omp end do [nowait] ]
```

Loop construct

Loop in Fortran:

```
integer :: i,n=200
real :: a(n),b(n),c(n)


do i=1, n
  a(i) = b(i) + c(i)
enddo
```

Loop construct

Loop in Fortran:

```
integer :: i,n=200
real :: a(n),b(n),c(n)
!$OMP PARALLEL
!$OMP DO
do i=1, n
  a(i) = b(i) + c(i)
enddo
!$OMP END DO
!$OMP END PARALLEL
```

Loop construct

Loop in C:

```
int main ()
{
  int i, n, var2, var3;
  ...
  Serial code
  ...

 {

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

Loop construct

Loop in C:

```
int main ()
{
  int i, n, var2, var3;
  ...
  Serial code
  ...
#pragma omp parallel
 {
 #pragma omp for
   for(i=1; i<=n; i++)
      a[i] = b[i] + c[i]
 }
}
```

Loop construct

Requirements for Loop Parallelization:

- **no dependencies** between loop indicies;
- an element of an array is assigned to by at most one iteration;
- no loop iteration reads array elements modified by any other dependency;
- due to overhead of parallelization - use only on loops where individual iterations take a long time.

Loop construct

Example of code with **NO** data dependencies

- Fortran:

```
!$omp parallel do
   do i = 1, n
     a(i) = b(i) + c(i)
   enddo
```

- C/C++:

```
#pragma omp parallel for
  for(i=1; i<=n; i++)
    a[i] = b[i] + c[i]
```

Loop construct

Example of code **with** data dependencies

- Fortran:

```
do i = 2, 5
   a(i) = a(i) + a(i-1)
enddo
```

- C/C++:

```
for(i=2; i<=5; i++)
   a[i] = a[i] + a[i-1];
```

Scheduling

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.

Static:

iterations are divided into chunks of size chunk_size, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number.

Dynamic:

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.

Scheduling

Guided:

the 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 decrease with time.

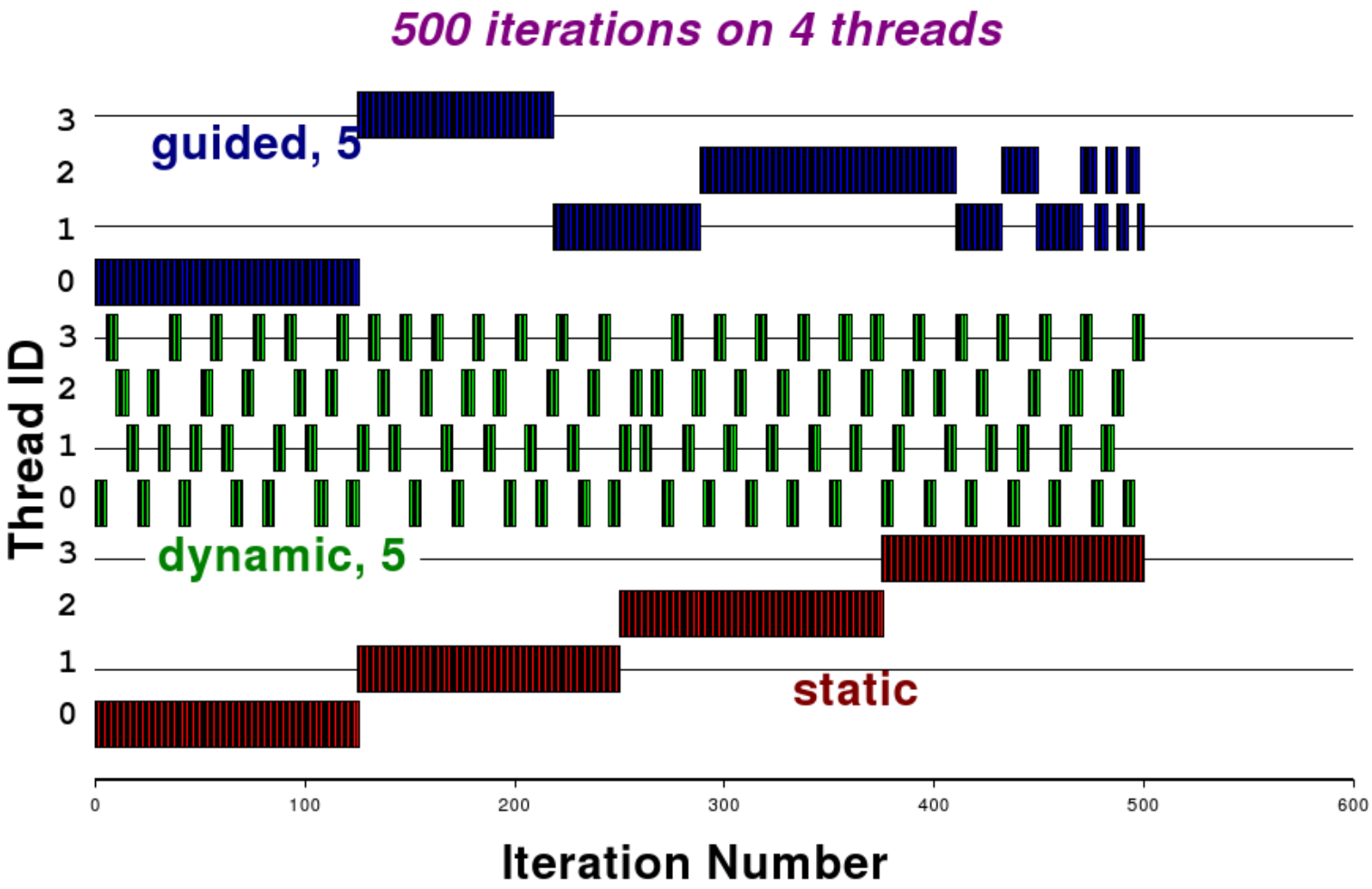
Runtime:

the decision regarding scheduling is deferred until run time.

Auto:

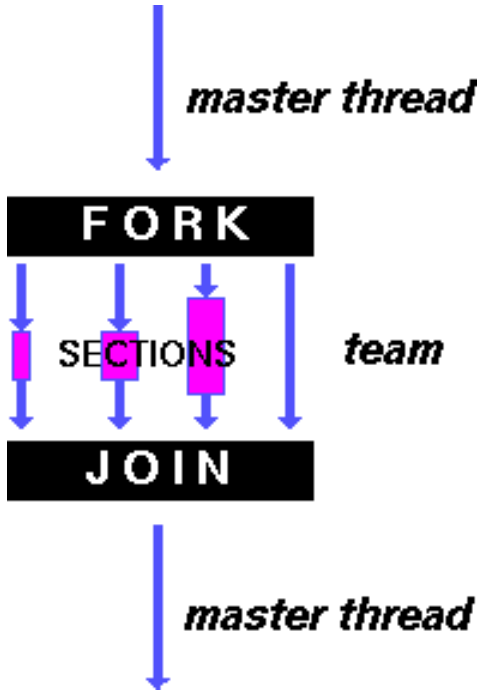
the decision regarding scheduling is delegated to the compiler and/or runtime system.

Scheduling


500 iterations on 4 threads
guided, 5
dynamic, 5
static
Thread ID
Iteration Number
0
100
200
300
400
500
600

Sections

The **sections construct** is a noniterative worksharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team.



Sections

Fortran:

```
!$OMP PARALLEL
...

  call subrA(c,d)

  call subrB(e,f)

  call subrC(g,h,i)

...
!$OMP END PARALLEL
```

Sections

Fortran:

```
!$OMP PARALLEL
...
!$OMP SECTIONS
!$OMP SECTION
  call subrA(c,d)
!$OMP SECTION
  call subrB(e,f)
!$OMP SECTION
  call subrC(g,h,i)
!$OMP END SECTIONS
...
!$OMP END PARALLEL
```

Sections

C/C++:

```
#pragma omp parallel
{
...

 {

A=subr_A(c,d)

B=subr_B(e,f)

C=subr_c(g,h,i)
 }
...
}
```

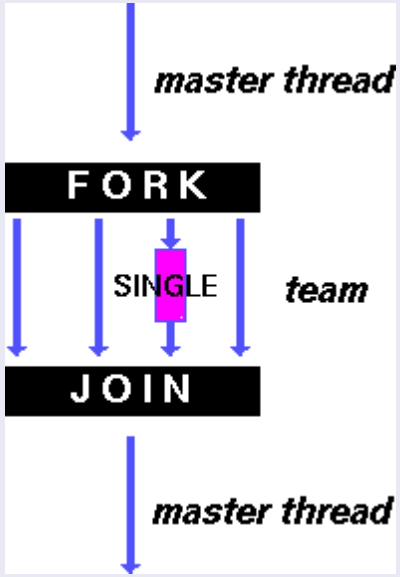
Sections

C/C++:

```
#pragma omp parallel
{
...
#pragma omp sections
 {
#pragma omp section
A=subr_A(c,d)
#pragma omp section
B=subr_B(e,f)
#pragma omp section
C=subr_c(g,h,i)
 }
...
}
```

Single

The **single construct** specifies that the associated structured block is executed by only one of the threads in the team (not necessarily the master thread). The other threads in the team, which do not execute the block, wait at an implicit barrier at the end of the single construct unless a nowait clause is specified.



Single

Fortran:

```
!$OMP PARALLEL
...

read *, a

...
!$OMP END PARALLEL
```

Single

Fortran:

```
!$OMP PARALLEL
...
!$OMP SINGLE
read *, a
!$OMP END SINGLE
...
!$OMP END PARALLEL
```

Single

C/C++:

```
#pragma omp parallel
{
...

 {
printf(‘‘Beginning work’’);
 }
...
}
```

Single

C/C++:

```
#pragma omp parallel
{
...
#pragma omp single
 {
printf(‘‘Beginning work’’);
 }
...
}
```

Master & Sinchronization constructs

Master:

- the **master** construct.

Sinchronization constructs:

- the **critical** construct.
- the **barrier** construct.
- the **atomic** construct.
- the **ordered** construct.

Master

The **master construct** specifies a structured block that is executed by the master thread of the team. There is no implied barrier either on entry to, or exit from, the master construct.

Fortran:

```
!$OMP PARALLEL
...

read , a

...
!$OMP END PARALLEL
```

Master

The **master construct** specifies a structured block that is executed by the master thread of the team. There is no implied barrier either on entry to, or exit from, the master construct.

Fortran:

```
!$OMP PARALLEL
...
!$OMP MASTER
read , a
!$OMP END MASTER
...
!$OMP END PARALLEL
```

Master

C/C++:

```
#pragma omp parallel
{
...

 {
printf(‘‘Beginning work’’);
 }
...
}
```

Master

C/C++:

```
#pragma omp parallel
{
...
#pragma omp master
 {
printf(‘‘Beginning work’’);
 }
...
}
```

Barrier

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

Fortran:

```
!$OMP PARALLEL
...
X=FUNC_A(X)

...
!$OMP END PARALLEL
```

Barrier

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

Fortran:

```
!$OMP PARALLEL
...
X=FUNC_A(X)
!$OMP BARRIER
...
!$OMP END PARALLEL
```

Critical

The **critical construct** restricts execution of the associated structured block to a single thread at a time. An optional name may be used to identify the critical construct.

Fortran:

```
!$OMP PARALLEL
...

X=FUNC_A(X)


...
!$OMP END PARALLEL
```

Critical

The **critical construct** restricts execution of the associated structured block to a single thread at a time. An optional name may be used to identify the critical construct.

Fortran:

```
!$OMP PARALLEL
...
!$OMP CRITICAL FOO
X=FUNC_A(X)
!$OMP END CRITICAL FOO
...
!$OMP END PARALLEL
```

Critical

All **critical constructs** without a name are considered to have the same unspecified name.

C/C++:

```
#pragma omp parallel
{
...

 {
x=subr_A(x)
 }
...
}
```

Critical

All **critical constructs** without a name are considered to have the same unspecified name.

C/C++:

```
#pragma omp parallel
{
...
#pragma omp critical foo
 {
x=subr_A(x)
 }
...
}
```

Atomic

The **atomic construct** ensures that a specific storage location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.

Fortran:

```
!$OMP PARALLEL
...

X=X+1
...
!$OMP END PARALLEL
```

Atomic

The **atomic construct** ensures that a specific storage location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.

Fortran:

```
!$OMP PARALLEL
...
!$OMP ATOMIC
X=X+1
...
!$OMP END PARALLEL
```

Atomic

C/C++:

```
#pragma omp parallel
{
...

 X++;
...
}
```

Atomic

C/C++:

```
#pragma omp parallel
{
...
#pragma omp atomic
 X++;
...
}
```

Ordered

The **ordered construct** specifies a structured block in a loop region that will be executed in the order of the loop iterations. This sequentializes and orders the code within an ordered region while allowing code outside the region to run in parallel.

Fortran:

```
!$OMP PARALLEL
!$OMP DO
DO i=1,N
  A(i)=...

  PRINT *,a(i)

ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

Ordered

The **ordered construct** specifies a structured block in a loop region that will be executed in the order of the loop iterations. This sequentializes and orders the code within an ordered region while allowing code outside the region to run in parallel.

Fortran:

```
!$OMP PARALLEL
!$OMP DO ORDERED
DO i=1,N
  A(i)=...
!$OMP ORDERED
  PRINT *,a(i)
!$OMP END ORDERED
ENDDO
!$OMP END DO ORDERED
!$OMP END PARALLEL
```

Ordered

C/C++:

```
#pragma omp parallel
{
...
#pragma omp for
  for (i=0;i<n;++i)
  {
    a[i] = b[i] + 1.0;

     printf(``%f\n``,a[i]);
  }
...
}
```

Ordered

C/C++:

```
#pragma omp parallel
{
...
#pragma omp for ordered
  for (i=0;i<n;++i)
  {
    a[i] = b[i] + 1.0;
#pragma omp ordered
     printf(``%f\n``,a[i]);
  }
...
}
```

OpenMP Memory Model

- OpenMP provides a consistent shared-memory model. All threads have access to the **main memory** to retrieve shared variables.
- Each thread also has access to another type of memory that cannot be accessed by another threads, called **thread private memory**.
- A directive that accepts data-sharing attribute clauses determines two kinds of access to variables used in the directive's associated structured block: **shared** and **private**.

Data-Sharing Attribute Clauses

C/C++:

- Variables with **automatic** storage duration that are declared in a scope inside the construct are **private**.
- Objects with **dynamic** storage duration are **shared**.
- Variables with **static** storage duration that are declared in a scope inside the construct are **shared**.
- Formal arguments of called routines in the region that are passed by reference inherit the data-sharing attributes of the associated actual argument.
- Other variables declared in called routines in the region are private.
- The **loop iteration variable** in the associated for-loop of a for or parallel for construct is **private**.

Fortran

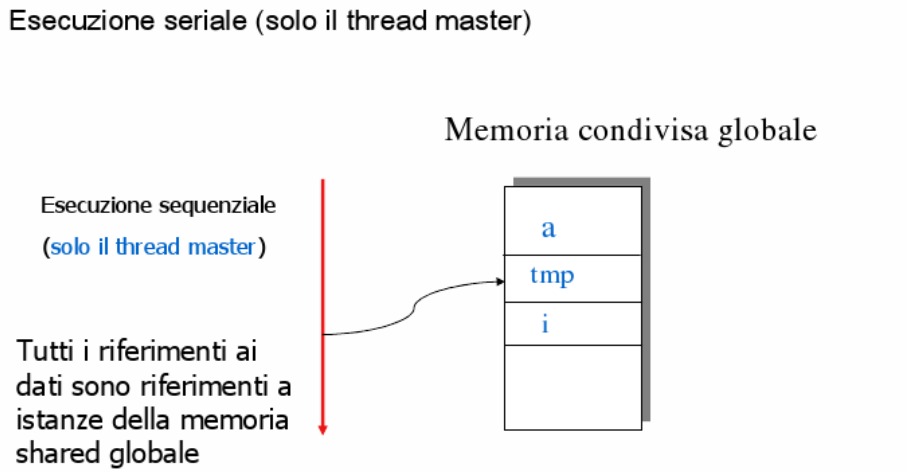
- Variables and common blocks appearing in threadprivate directives are threadprivate.
- **The loop iteration variable(s)** in the associated do-loop(s) of a do or parallel do construct is(are) **private**.
- A loop iteration variable for a sequential loop in a parallel construct is private in the innermost such construct that encloses the loop.
- Assumed-size arrays are shared.
- Local variables declared in called routines in the region and that have the save attribute, or that are data initialized, are shared unless they appear in a threadprivate directive.

Fortran

- Variables belonging to common blocks, or declared in modules, and referenced in called routines in the region are shared unless they appear in a threadprivate directive.
- Dummy arguments of called routines in the region that are passed by reference inherit the data-sharing attributes of the associated actual argument.
- Implied-do indices and other local variables declared in called routines in the region are private.

Data-Sharing Attribute Clauses

```
integer :: i=5,n=200
real :: tmp=7
```

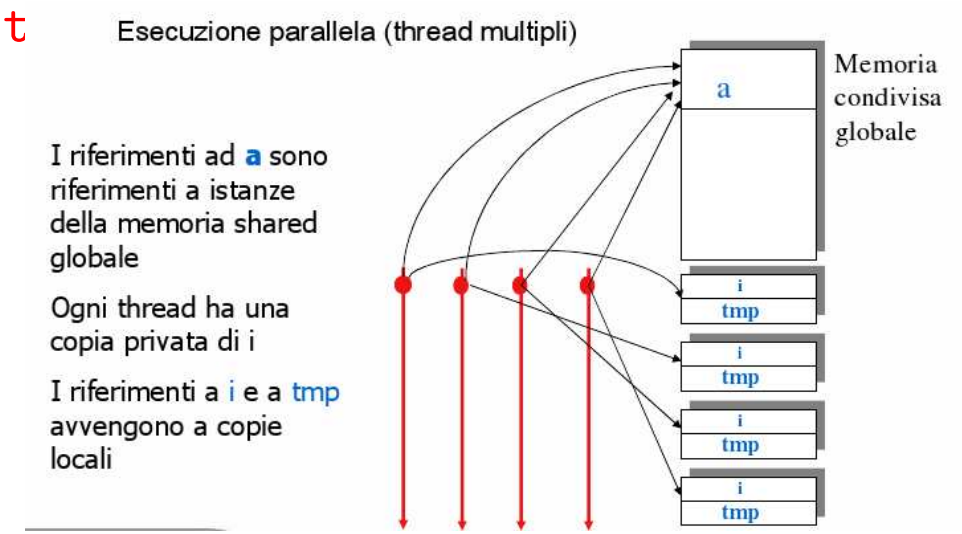



```
!$OMP PARALLEL
!$OMP DO
do i=1, n
  tmp = func(b(i))
  a(i) = b(i) + tmp
enddo
!$OMP END DO
!$OMP END PARALLEL
```

Data-Sharing Attribute Clauses

```
integer :: i=5,n=200
real :: tmp=7
```

```
!$OMP PARALLEL PRIVATE(t
!$OMP DO
do i=1, n
  tmp = func(b(i))
  a(i) = b(i) + tmp
enddo
!$OMP END DO
!$OMP END PARALLEL
```



Data-Sharing Attribute Clauses

Shared:

declares a list of one or more items to be shared by threads generated by a parallel construct.

Private:

declares one or more list items to be private to a task. No other thread can access this data. Changes can only visible to the thread owning the data.

Firstprivate:

declares one or more list items to be private to a task, and initializes each of them with the value that the corresponding original item has when the construct is encountered.

Data-Sharing Attribute Clauses

Lastprivate:

declares one or more list items to be private to an implicit task, and causes the corresponding original list item to be updated after the end of the region.

```
!$omp do
do i = 1,n-1
  a(i) = b(i+1)
enddo
!$omp end do
print *, i
```

Data-Sharing Attribute Clauses

Lastprivate:

declares one or more list items to be private to an implicit task, and causes the corresponding original list item to be updated after the end of the region.

```
!$omp do lastprivate(i)
do i = 1,n-1
  a(i) = b(i+1)
enddo
!$omp end do
print *, i
```

Data-Sharing Attribute Clauses

Default:

The **default** clause explicitly determines the data-sharing attributes of variables that are referenced in a parallel or task construct and would otherwise be implicitly determined. Only a single default clause may be specified on a parallel directive.

C/C++:

```
default(shared | none)
```

Fortran:

```
default(private | firstprivate | shared | none)
```

Data-Sharing Attribute Clauses

```
!$omp do
do i = 1,n
  x = x + a(i)
enddo
!$omp end do
```

Data-Sharing Attribute Clauses

```
!$omp do reduction(+:x)
do i = 1,n
  x = x + a(i)
enddo
!$omp end do
```

Reduction:

The **reduction** clause specifies an operator and one or more list items. For each list item, a private copy is created in each implicit task, and is initialized appropriately for the operator. After the end of the region, the original list item is updated with the values of the private copies using the specified operator.

Support for most arithmetic and logical operators
$+, *, -, .MIN., .MAX., .AND., .OR., \ldots$

Runtime Library

OpenMP provides several user-callable functions to control and query parallel environment.

- The Runtime Libraries take precedence over the corrisponding environment variables;
- Recommended to use under control of conditional compilation (#ifdef _OPENMP);
- C/C++ programs need to include <omp.h>;
- Fortran program may want to use "USE OMP_LIB" or include "omp_lib.h".

Runtime Library

omp_get_num_threads

`num_threads=omp_get_num_threads():`

Gets number of threads in team;

omp_get_thread_num

`thread_id=omp_get_thread_num():`

Gets thread ID;

omp_get_wtime

`time=omp_get_wtime():`

Return elapsed wall clock time in seconds.

Environment Variables

OMP_NUM_THREADS:

sets the number of threads to use for parallel regions;

OMP_SCHEDULE:

controls the schedule type and chunk size of all loop directives that have the schedule type runtime.

OMP_STACKSIZE:

specifies the size of the stack for threads created by the OpenMP implementation.

- **sh**:

```
$ export OMP_NUM_THREADS=8
$ export OMP_SCHEDULE="guided,4"
```