Introduction to Programming with OpenMP

OpenMP Team
Kent Milfeld, Lars Koesterke, and Antonio Gomez
milfeld/lars/agomez@tacc.utexas.edu
2016/03/31

Sides at: tinyurl.com/tacc-openmp or portal.tacc.utexas.edu/training click View Details

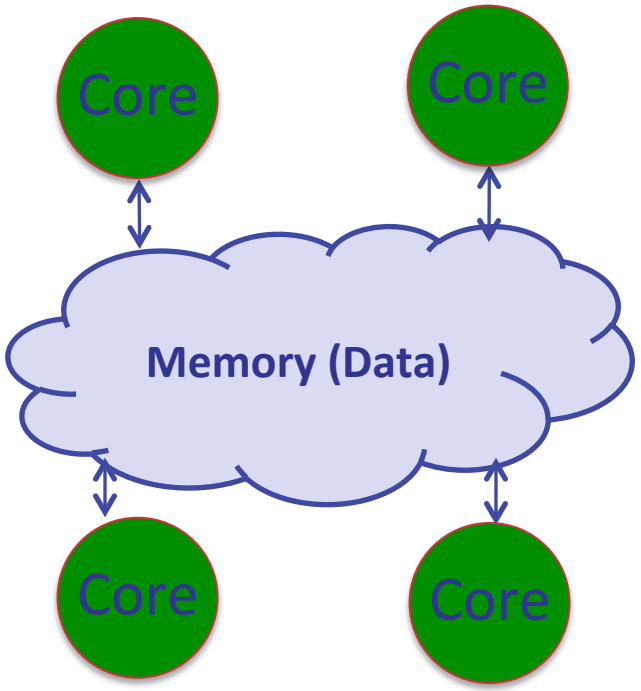
Outline

- What is OpenMP?
- How OpenMP Parallelism Works
  - Architecture
  - Parallelism
    - Execution Model (Fork-Join)
    - Memory Model & Synchronization
- OpenMP Syntax and API
  - Compiler Directives
  - Runtime Library Routines
  - Environment variables
- OpenMP 3.0 Changes

What is OpenMP?

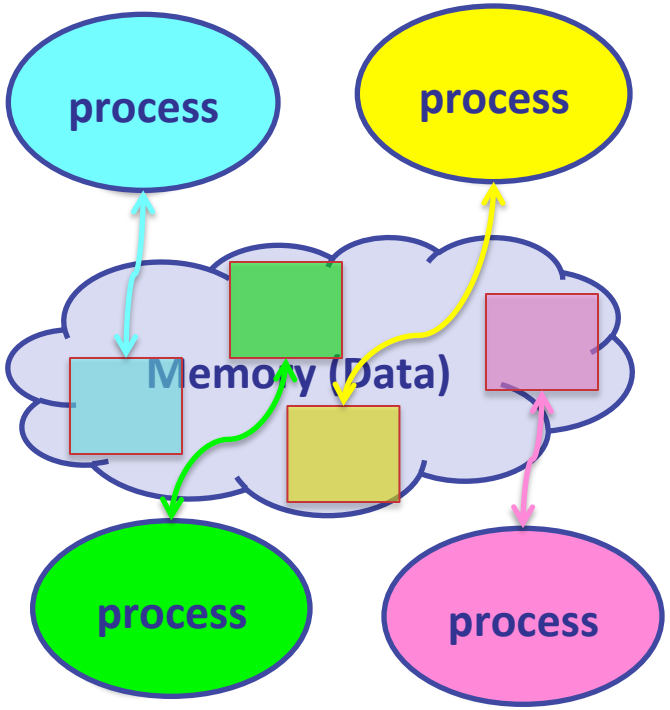
- OpenMP stands for **Open Multi-Processing**
- A directive-based programming language for executing program components in parallel within a shared memory architecture.
- Three primary components are:
  - Directives to the Compiler within code (pragmas in C/C++, comments in Fortran)
  - Runtime Library Routines
  - Environment Variables
- Standard specifies C, C++, and FORTRAN Directives & API
- http://www.openmp.org/ has the specification, examples, tutorials and documentation

Hardware Architecture



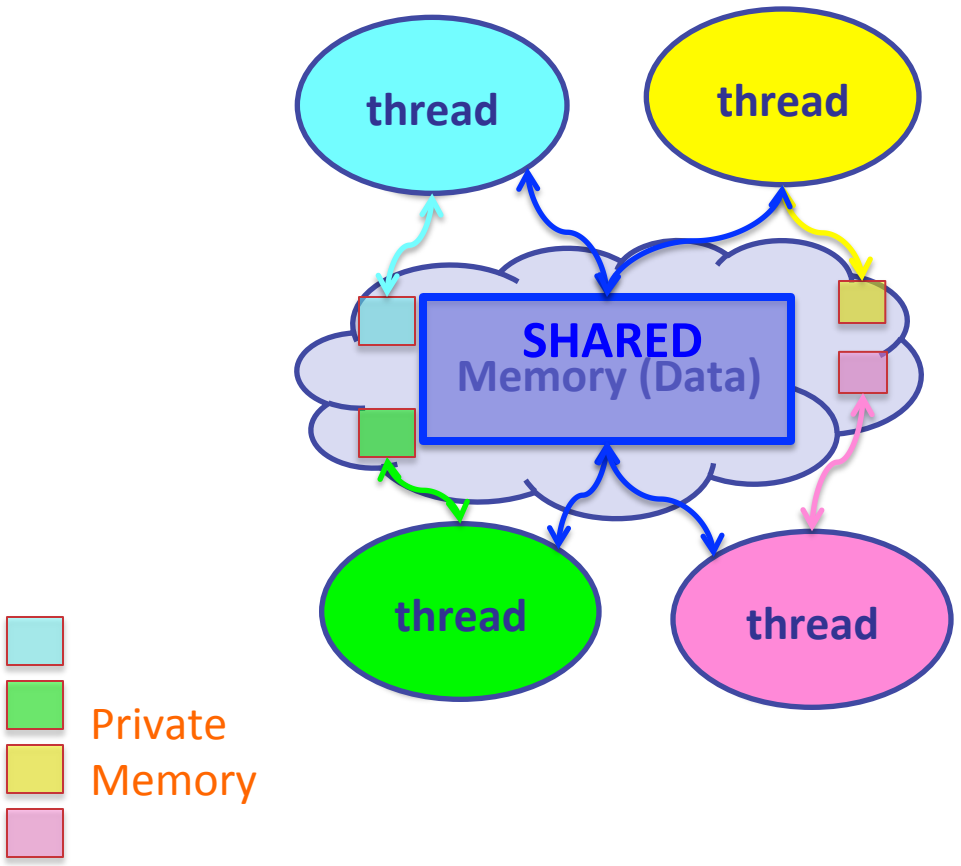
- Cores: where processes or lightweight processes (threads) can run.
- Memory
  - Multiple cores have access to the Memory
  - Processes have their own section of memory to use

Software Architecture



- Cores: where processes or lightweight processes (threads) can run.
- Memory
  - Multiple cores have access to the Memory
  - Processes have their own section of memory to use

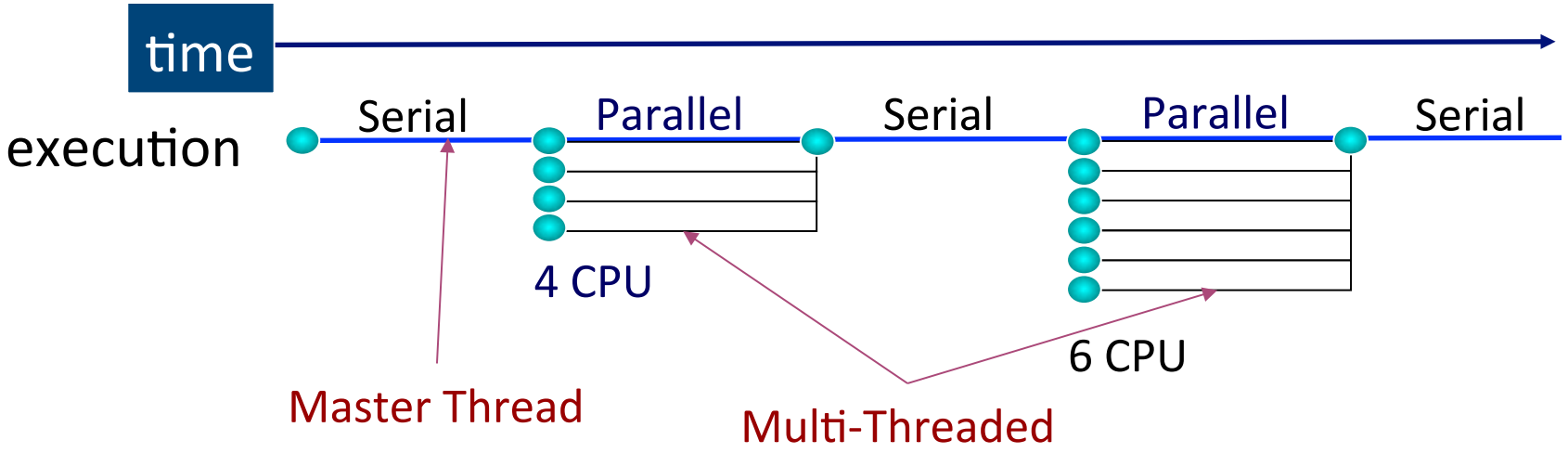
Software Architecture



- Threads Execute on the Cores
- Threads within an application can share memory.
- Data: shared or private
  - Shared data: all threads can access data in shared memory
  - Private data: can only be accessed by threads that own it

OpenMP Fork-Join Parallelism

- Programs begin as a single process: master thread
- Master thread executes in serial mode until the parallel region construct is encountered
- After executing the statements in the parallel region, team threads synchronize and terminate (join) but master continues



Memory Model and Synchronization

- Every thread has access to shared memory.
  When multiple threads are created all threads share the same address space.

- Simultaneous updates to shared memory can create a *race condition*. Results change with different thread scheduling.

- Use mutual exclusion directives to avoid race conditions - but don't use too many because this will serialize performance.

- Barriers are used to synchronize threads.

- A flush exists at barriers, to make make local changes to data (writes) seen by all threads.

Race conditions

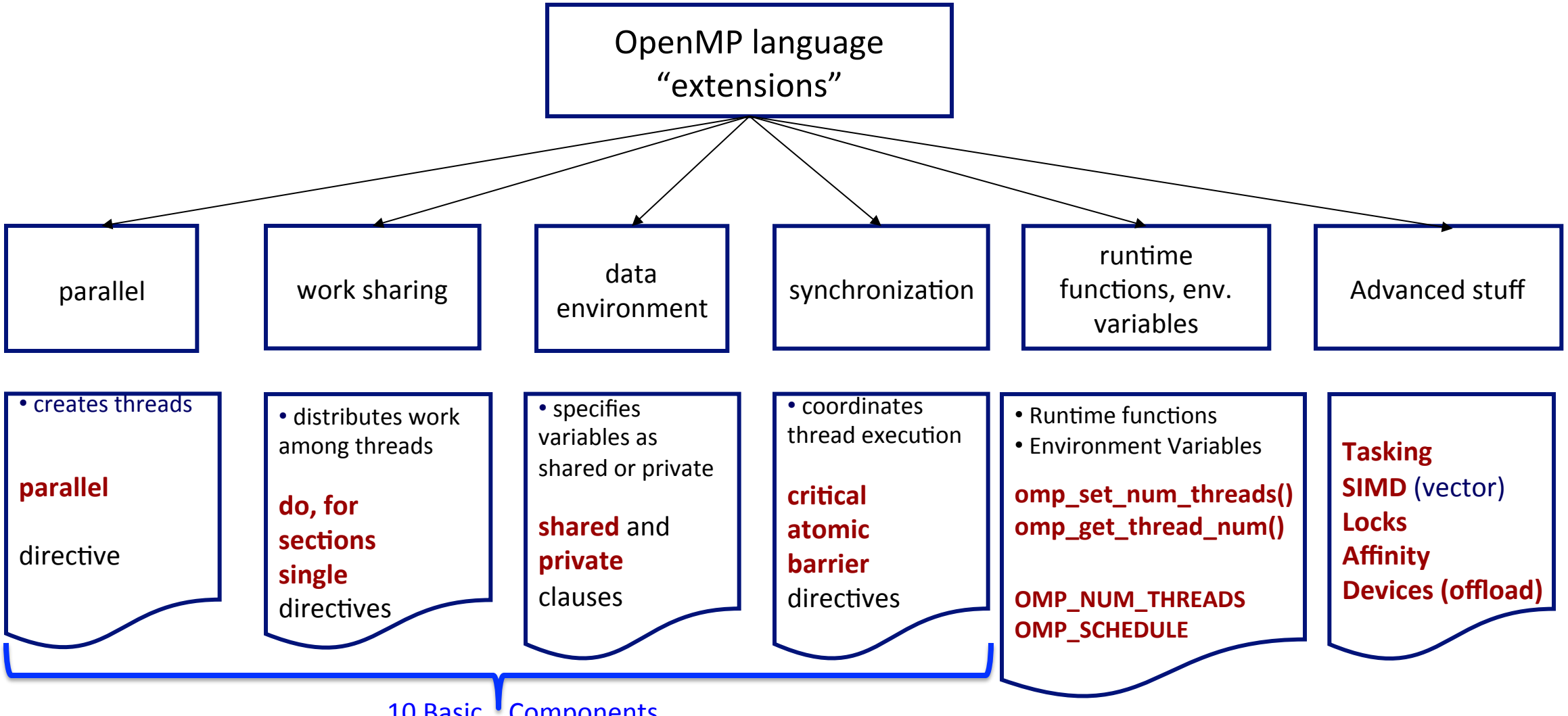
Thread 0: **i = i+2**
Thread 1: **i = i+3**

Initial value of **i** in memory = 0.

scenario 1.		scenario 2.		scenario 3.	
Thread 0	Thread 1	Thread 0	Thread 1	Thread 0	Thread 1
read I = 0	read I = 0	read I = 0	read I = 0	read I = 0	
compute I = 2	compute I = 3	compute I = 2	compute I = 3	compute I = 2	
write I = 2			write I = 3	write I = 2	
	write I = 3	write I = 2			read I = 2
					compute I = 5
					write I = 5
I = 3		I = 2		I = 5	

I = 0

OpenMP Constructs



OpenMP Syntax

- Function prototypes and types are in the files:

```
#include <omp.h>                                   C
use omp_lib                                        F90
```

- Compiler directive syntax:

```
#pragma omp construct [clause [[,]clause]…]        C
        !$omp construct [clause [[,]clause]…]      F90
```

- Example

```
#pragma omp parallel num_threads(4)                C
        !$omp parallel num_threads(4)              F90
```

- Most OpenMP constructs apply to a “structured block”; that is, a block of one or more statements with one point of entry at the top and one point of exit at the bottom

OpenMP Directive Scope

- OpenMP directives are comments/pragmas in source:

  F90 : **!$OMP** free-format*

  C/C++ : directives begin with the **# pragma omp** sentinel

Parallel regions are marked by enclosing parallel directives

  F90 : parallel … **end** parallel

  C/C++ : Enclosed in block **{ …}**, or single statement

- Work-sharing loops are marked by parallel do/for

* Fortran Fixed Format:
!$OMP, **C$OMP** or ***$OMP**

Fortran

```
!$omp parallel
  ...
!$omp end parallel

!$omp parallel
  call foo(...)
!$omp end parallel
```

C/C++

```
# pragma omp parallel
  {...
  }

# pragma omp parallel for
  foo(...);
```

Parallel Region & Work-Sharing

Use OpenMP directives to specify Parallel Region & Work-Sharing constructs

```
parallel


end parallel
```

Code block	**Each Thread Executes**
do / for	Work Sharing
sections	Work Sharing
single	Work Sharing (one thread)

Sentinels "!$omp" and "#pragma omp") not shown here.

C/C++

Parallel Regions

```
1   #pragma omp parallel
2   {
3       code block
4       work(…);
5   }
```

Line 1 Team of threads formed at parallel region

Lines 3-4 Each thread executes code block and subroutine calls.

No branching (in or out) in a parallel region

Line 5 All threads synchronize at end of parallel region (implied barrier)

Use the thread number to divide work among threads

F90

Parallel Regions

```
1   !$OMP PARALLEL
2       code block
3       call work(…)
4   !$OMP END PARALLEL
```

Line 1 Team of threads formed at parallel region.

Lines 2-3 Each thread executes code block and subroutine calls. No branching (in or out) in a parallel region.

Line 4 All threads synchronize at end of parallel region (implied barrier).

Use the thread number to divide work among.

C/C++

Parallel Region & Number of Threads

- For example, to create a 4-thread Parallel region.
- export OMP_NUM_THREADS=4; a.out

```
double A[1000];
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    foo(ID, A);
}
```

In C++ a local variables are private to the thread…

- Each thread executes the code within the structured block
- Thread numbers range from 0 to Nthreads-1
- Each thread calls foo(ID,A) with a different ID {= 0 to 3}

F90

Parallel Region & Number of Threads

- For example, to create a 4-thread Parallel region.
- export OMP_NUM_THREADS=4; a.out

```
real ::  A(1000; integer :: ID
!$omp parallel


   ID = omp_get_thread_num()
   call foo(ID, A)
!$omp end parallel
```

But we need to make id Private to the thread– more later

- Each thread executes the code within the structured block
- Thread numbers range from 0 to Nthreads-1
- Each thread calls foo(ID,A) with a different `ID {= 0 to 3}`

C/C++

Work sharing: Loop

```
1   #pragma omp parallel
2   {
3     #pragma omp for
4     for (i=0; i<N; i++)
5     {
6       a[i] = b[i] + c[i];
7     }
8   }
```

Or

```
#pragma omp parallel
#pragma omp for
for (i=0; i<N; i++)
      a[i] = b[i] +
c[i];
```

Line 1 Team of threads formed (parallel region).

Line 3-7 Loop iterations are split among threads.
implied barrier at }

Each loop iteration must be independent of other iterations.

F90

Work sharing: Loop

```
1   !$OMP PARALLEL
2      !$OMP DO
3      do i=1,N
4         a(i) = b(i) + c(i)
5      enddo
6   !$OMP END PARALLEL
```

Line 1 Team of threads formed (parallel region).

Line 3-4 Loop iterations are split among threads by DO construct.

Line 5 !$OMP END DO is optional after the enddo.

Line 5 Implied barrier at enddo.

Each loop iteration must be independent of other iterations.

OpenMP Combined Constructs

Fortran

```
!$omp parallel
  !$omp do
    do ...
    end do
  !$omp end do
!$omp end parallel


!$omp parallel do
  do ...
  end do
                    !*
```

C/C++

```
# pragma omp parallel
{
  #pragma omp for
  for (...){...}
}



# pragma omp parallel for
  for(){...}
```

* Since Fortran do has a block structure "!$omp end do" Is not required

C/C++

Work-Sharing: Sections

```
1  #pragma omp parallel sections
2  {
3    #pragma omp section
4      {
5        work_1();
6      }
7    #pragma omp section
8      { work_2(); }
9  }
```

Line 1 Team of threads formed (parallel region).

Line 3-8 One thread is working on each section.

Line 9 End of parallel sections with an implied barrier.

Scales only to the number of sections.

F90

Work-Sharing: Sections

```
 1   !$omp parallel sections
 2
 3     !$omp section
 4
 5       call work_1()
 6
 7     !$omp section
 8       call work_2()
 9
10   !$omp end parallel sections
```

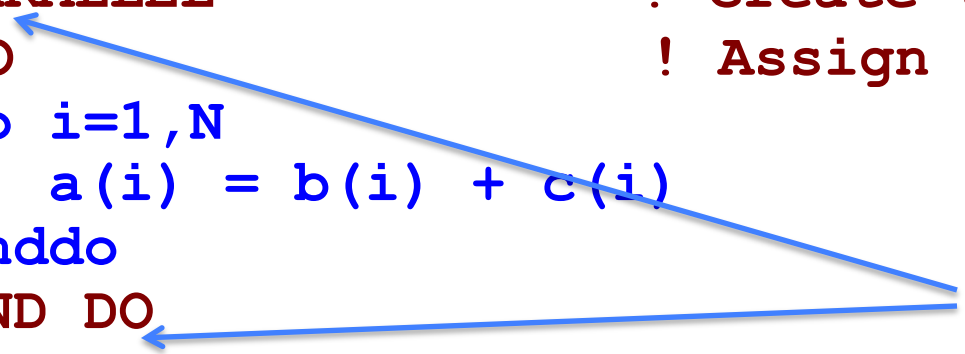
Line 1 Team of threads formed (parallel region).
Line 3-9 One thread is working on each section.
Line 10 End of parallel sections with an implied barrier.

Scales only to the number of sections.

Shorthand (again)

```
1   !$OMP PARALLEL DO
2         do i=1,N
3            a(i) = b(i) + c(i)
4         enddo
5   !$OMP END PARALLEL DO
```

```
1   !$OMP PARALLEL                ! Create team of threads
2   !$OMP DO                      ! Assign iterations to threads
3         do i=1,N
4            a(i) = b(i) + c(i)
5         enddo
6   !$OMP END DO
7   !$OMP END PARALLEL
```



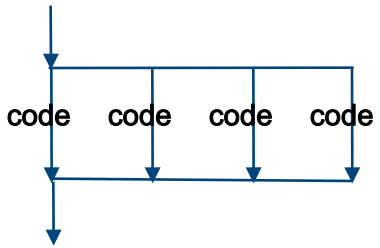
But, how does all this work
if we need other code/work here?

OpenMP Parallel Constructs

Replicated : Work blocks are executed by all threads.
Work-Sharing : Work is divided among threads.

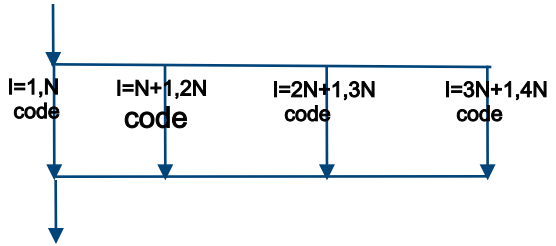
4 Threads

```
PARALLEL
  {code}
END PARALLEL
```



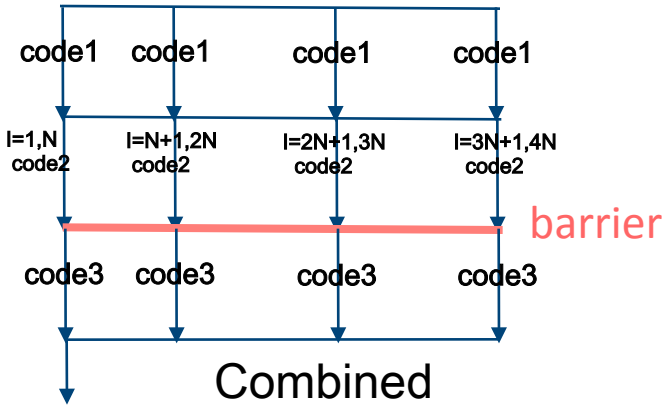
Replicated

```
PARALLEL DO
  do I = 1,N*4
    {code}
  end do
END PARALLEL DO
```



Work-Sharing

```
PARALLEL
  {code1}
DO
  do I = 1,N*4
    {code2}
  end do
END DO
  {code3}
END PARALLEL
```



Combined

OpenMP Clauses

Clauses control the behavior of an OpenMP directive:

1. Schedule for for/do worksharing (Static, Dynamic, Guided, etc.)
2. Data scoping (Private, Shared, Default)
3. Initialization (e.g. COPYIN, FIRSTPRIVATE)
4. Parallelize a region or not (IF(logic_expression))
5. Number of threads used (NUM_THREADS)

Schedule Clause for loop worksharing

schedule(static)

Each CPU receives one set of contiguous iterations

```
!$omp do schedule(...)
   do i=1,128
     A(i)=B(i)+C(i)
   enddo
```

schedule(static, C)

Iterations are divided round-robin fashion in chunks of size C

schedule(dynamic, C)

Iterations handed out in chunks of size C as CPUs become available

schedule(guided, C)

Each of the iterations are handed out in pieces of exponentially decreasing size, with C minimum number of iterations to dispatch each time

schedule (runtime)

Schedule and chunk size taken from the OMP_SCHEDULE environment variable

Example - schedule(static,16), threads = 4

```
!$omp parallel do schedule(static,16)
      do i=1,128
       A(i)=B(i)+C(i)
      enddo
```

thread0:
`do i=1,16`
`  A(i)=B(i)+C(i)`
`enddo`
`do i=65,80`
`  A(i)=B(i)+C(i)`
`enddo`
	thread2:
	`do i=33,48`
	`  A(i)=B(i)+C(i)`
	`enddo`
	`do i = 97,112`
	`  A(i)=B(i)+C(i)`
	`enddo`

thread1:
`do i=17,32`
`  A(i)=B(i)+C(i)`
`enddo`
`do i = 81,96`
`  A(i)=B(i)+C(i)`
`enddo`
	thread3:
	`do i=49,64`
	`  A(i)=B(i)+C(i)`
	`enddo`
	`do i = 113,128`
	`  A(i)=B(i)+C(i)`
	`enddo`

Comparison of Scheduling Options

name	type	chunk	chunk size	chunk #	static or dynamic	compute overhead
partitioned	static	no	N/P	P	static	lowest
interleaved	static	yes	C	N/C	static	low
simple dynamic	dynamic	optional	C	N/C	dynamic	medium
guided	guided	optional	decreasing from N/P*	no fewer than C	dynamic	high
runtime	runtime	no	varies	varies	varies	varies

*Decreases as (N-unassigned)/P

OpenMP Data Environment

- Clauses control the data-sharing attributes of variables within a parallel region:

  shared, private, reduction (firstprivate, lastprivate)

Default variable scope (in parallel region):

1. Variables declared in main/program (C/F90) are shared by default
2. Global variables are shared by default
3. Automatic variables within subroutines called from within a parallel region are private (reside on a stack private to each thread)
4. Loop index of worksharing loops are private.
5. Default scoping rule can be changed with **`default`** clause

C/C++

Private & Shared Data

shared - Variable is shared (seen) by all threads

private - Each thread has a private instance (copy) of the variable

Defaults: The for-loop index is private, all other variables are shared

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

OK to be explicit;
but not necessary.

All threads have access to the same storage areas for a, b, c, and n, but each loop has its own private copy of the loop index, i

F90

Private & Shared Data

shared - Variable is shared (seen) by all threads

private - Each thread has a private instance (copy) of the variable

Defaults: The for-loop index is private, all other variables are shared

```
#pragma omp parallel for shared(a,b,c,n)  private(i)
      for (i=0; i<n; i++){
            a[i] = b[i] + c[i];
         }
```

OK to be explicit;
but not necessary.

All threads have access to the same storage areas for a, b, c, and n, but each loop has its own private copy of the loop index, i

Private Data Example

- In the following loop, each thread needs its own private copy of temp
- If temp were shared, the result would be unpredictable since each thread would be writing/reading to/from the same memory location

```
#pragma omp parallel for shared(a,b,c,n) private(temp,i)
for (i=0; i<n; i++){
    temp = a[i] / b[i];
    c[i] = temp + cos(temp);
}
```

- A **lastprivate(temp)** clause will copy the last loop(stack) value of temp to the (global) temp storage when the parallel for is complete.
- A **firstprivate(temp)** would copy the global temp value to each stack's temp.

F90

Private Data Example

- In the following loop, each thread needs its own private copy of temp
- If temp were shared, the result would be unpredictable since each thread would be writing/reading to/from the same memory location

```
!$omp parallel for shared(a,b,c,n) private(temp,i)
do i = 1,n
   temp = a(i) / b(i)
   c(i) = temp + cos(temp)
endo
```

- A **lastprivate(temp)** clause will copy the last loop(stack) value of temp to the (global) temp storage when the parallel DO is complete.
- A **firstprivate(temp)** would copy the global temp value to each stack's temp.

C/C++

Reduction

- Operation that combines multiple elements to form a single result
- A variable that accumulates the result is called a reduction variable
- In parallel loops reduction operators and variables must be declared

```
float asum=0.0, aprod=1.0;

#pragma omp parallel for reduction(+:asum) reduction(*:aprod)
  for (i=0; i<n; i++){
    asum  = asum  + a[i];
    aprod = aprod * a[i];
  }
```

Each thread has a private **asum** and **aprod**, initialized to the operator's identity

- **After the loop execution, the master thread collects the private values of each thread and finishes the (global) reduction**

F90

Reduction

- Operation that combines multiple elements to form a single result
- A variable that accumulates the result is called a reduction variable
- In parallel loops reduction operators and variables must be declared

```
 real asum=0.0, aprod=1.0

!$omp parallel do reduction(+:asum) reduction(*:aprod)
   do i = 1,n
      asum  = asum  + a(i)
      aprod = aprod * a(i)
   enddo
 print*, asum, aprod
```

Each thread has a private **asum** and **aprod**, initialized to the operator's identity

- **After the loop execution, the master thread collects the private values of each thread and finishes the (global) reduction**

Synchronization

- Synchronization is used to impose order constraints and to protect access to shared data
- High-Level Synchronization
  - critical
  - atomic
  - barrier
  - ordered
- Low-Level Synchronization
  - locks

Synchronization: Critical/Atomic Directives

- When each thread must execute a section of code serially the region must be marked with critical/end critical directives
- Use the #pragma omp atomic directive for simple cases: can use hardware support

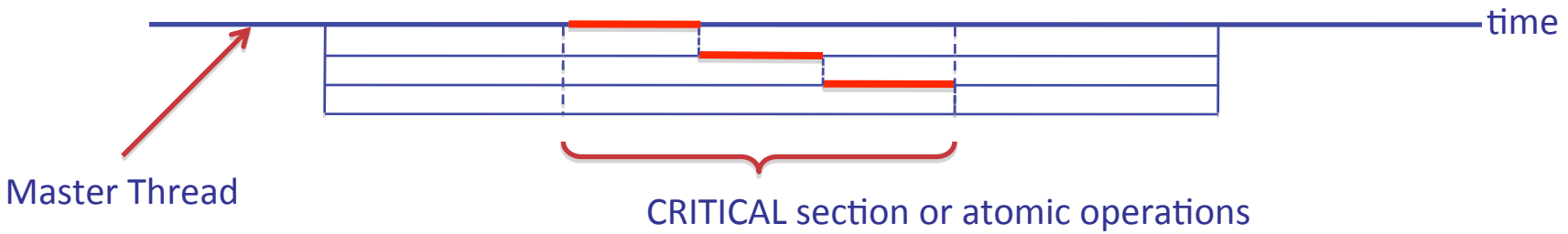
```
#pragma omp parallel shared(sum,x,y)
{...
   #pragma omp critical

      update(x);
      update(y);
      sum=sum+1;
   }
....
}
```

```
#pragma omp parallel shared(sum)
{...

   #pragma omp atomic
   sum=sum+1
...
}
```

Atomic has read, write, update, capture clauses.



Synchronization: Critical/Atomic Directives

- When each thread must execute a section of code serially the region must be marked with critical/end critical directives
- Use the !$ omp atomic directive for simple cases: can use hardware support

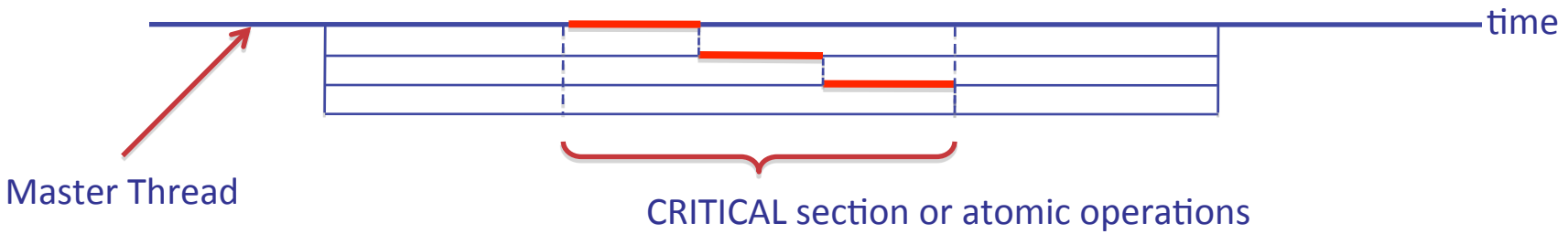
```
!$omp parallel shared(sum,x,y)
 ...

   !$omp critical
     update(x);
     update(y);
     sum=sum+1;
   !$omp end critical
....
!$omp end parallel
```

```
!$omp parallel shared(sum)
...

      !$omp atomic
      sum=sum+1;
...
!$omp end parallel
```

Atomic has read, write, update, capture clauses.



C/C++

Synchronization: Barrier

- Barrier: Each thread waits until all threads arrive and a flush occurs

```
#pragma omp parallel shared (A, B, C) private(id)
{
   id=omp_get_thread_num();
   A[id] = big_calc1(id);
   #pragma omp barrier
   #pragma omp for
   for(i=0;i<N;i++){
          C[i]=big_calc3(i,A);
   }<-------------------------------- Implicit barrier
   #pragma omp for nowait
   for(i=0;i<N;i++){
           B[i]=big_calc2(C, i);
   }<-------------------------------- No implicit barrier due to nowait
   A[id] = big_calc4(id);
}<---------------------------------- Implicit barrier
```

F90

Synchronization: Barrier

- Barrier: Each thread waits until all threads arrive and flush occurs

```
!$omp parallel shared (A, B, C) private(id)

   id=omp_get_thread_num()
   A(id) = big_calc1(id)
   !$omp barrier

   !$omp do
     do i = 1,N; C(i)=big_calc3(i,A); enddo
   !$omp end do <-------------------- Implicit barrier

   !$omp do
     do i = 1,N; B(i)=big_calc2(C, i); enddo
   !$omp end do nowait <------------- No implicit barrier due to nowait
   A(id) = big_calc4(id);
!$omp end parallel <-------------- Implicit barrier
```

Barriers

- The previous example could have been done with multiple parallel regions:
  1. There is an implicit barrier at the end of a parallel region.
  2. However, creating and recreating thread teams is expensive.

C/C++

Synchronization: Ordered

- The ordered region executes in the sequential order

```
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:countVal)
for (i=0;i<N;i++){
  tmp = foo(i);
  #pragma omp ordered
  countVal+= consume(tmp);
}
```

F90

Synchronization: Ordered

- The ordered region executes in the sequential order

```
!$omp parallel private (tmp)
!$omp do ordered reduction(+:countVal)
do i =1,N{
  tmp = foo(i)
  !$omp ordered
  countVal = countVal + consume(tmp)
end do
```

NOWAIT

- When a work-sharing region is exited, a barrier is implied - all threads must reach the barrier before any can proceed.
- By using the NOWAIT clause at the end of each loop inside the parallel region, an unnecessary synchronization of threads can be avoided.

```
#pragma omp parallel
{
  #pragma omp for nowait
       {
          for (i=0; i<n; i++)
            {work(i);}
       }
  #pragma omp for schedule(guided,k)
       {
          for (i=0; i<m; i++)
            {x[i]=y[i]+z[i];}
       }
}
```

F90

NOWAIT

- When a work-sharing region is exited, a barrier is implied - all threads must reach the barrier before any can proceed.
- By using the NOWAIT clause at the end of each loop inside the parallel region, an unnecessary synchronization of threads can be avoided.

```
!$OMP PARALLEL
   !$OMP DO
      do i=1,n
         work(i)
      enddo
   !$OMP END DO NOWAIT
   !$OMP DO schedule(guided,k)
      do i=1,m
         x(i)=y(i)+z(i)
      enddo
   !$OMP END DO
!$OMP END PARALLEL
```

Runtime Library Routines

function	description
omp_get_num_threads()	Number of threads in team, N
omp_get_thread_num()	Thread ID {0 -> N-1}
omp_get_num_procs()	Number of machine CPUs
omp_in_parallel()	True if in parallel region & multiple thread executing
omp_set_num_threads(#)	Set the number of threads in the team

Environment Variables

variable	description
OMP_NUM_THREADS=*integer*	Set to default no. of threads to use
OMP_SCHEDULE="*schedule-type[, chunk_size]*"	Sets "runtime" in loop schedule clause: "…omp for/do schedule(runtime)"
OMP_DISPLAY_ENV=*anyvalue*	Prints runtime environment at beginning of code execution.

[…] = optional

OpenMP Wallclock Timers

```
real*8 :: omp_get_wtime,   omp_get_wtick()        (Fortran)
double    omp_get_wtime(), omp_get_wtick();       (C)
```

```
double t0, t1, dt, res;
...
t0 = omp_get_wtime();
<work>
t1 = omp_get_wtime();
dt = t1 - t0;

res = 1.0/omp_get_wtick();
printf("Elapsed time = %lf\n",dt);
printf("clock resolution = %lf\n",res);
```

C/C++

NUM_THREADS clause

- Use the NUM_THREADS clause to specify the number of threads to execute a parallel region

```
#pragma omp parallel num_threads(scalar int expression)
{
    <code block>
}
```

where scalar integer expression must evaluate to a positive integer

- num_threads() supersedes the number of threads specified by the OMP_NUM_THREADS environment variable or that set by the OMP_SET_NUM_THREADS function

F90

NUM_THREADS clause

- Use the NUM_THREADS clause to specify the number of threads to execute a parallel region

```
!$omp parallel num_threads(scalar integer expression)
    <code block>
!$omp end parallel
```

where scalar integer expression must evaluate to a positive integer

- num_threads() supersedes the number of threads specified by the OMP_NUM_THREADS environment variable or that set by the OMP_SET_NUM_THREADS function

F90

Mutual Exclusion: Lock Routines

When each thread must execute a section of code serially, locks provide a more flexible way of ensuring serial access than **CRITICAL** and **ATOMIC** directives

```
call OMP_INIT_LOCK(maxlock)
!$OMP PARALLEL SHARED(X,Y)
...
call OMP_set_lock(maxlock)
call update(x)
call OMP_unset_lock(maxlock)
...
!$OMP END PARALLEL
call OMP_DESTROY_LOCK(maxlock)
```

OpenMP 3.0

- First update to the spec since 2005
- Tasking: move beyond loops with generalized tasks and support complex and dynamic control flows
- Loop collapse: combine nested loops automatically to expose more concurrency
- Enhanced loop schedules: Support aggressive compiler optimizations of loop schedules and give programmers better runtime control over the kind of schedule used
- Nested parallelism support: better definition of and control over nested parallel regions, and new API routines to determine nesting structure

Tasks Parallelism

- Allows to parallelize irregular problems
- OpenMP had tasks internally: chunks of execution; with OpenMP
- Tasks depend on other tasks: dataflow
- You can declare many tasks, OpenMP executes them, satisfying dependencies

Task dependencies

```
1   ! In parallel single region here.  Single thread generates tasks
    ! Other threads of team execute tasks.
2   for (i=0; i<children; i++)
3   #pragma omp task
4     f(i)
5   #pragma omp taskwait
6   parent_function()
```

Children tasks are put in a queue
can be executed in any order
but before the parent

Tasks: Usage

Task Construct:

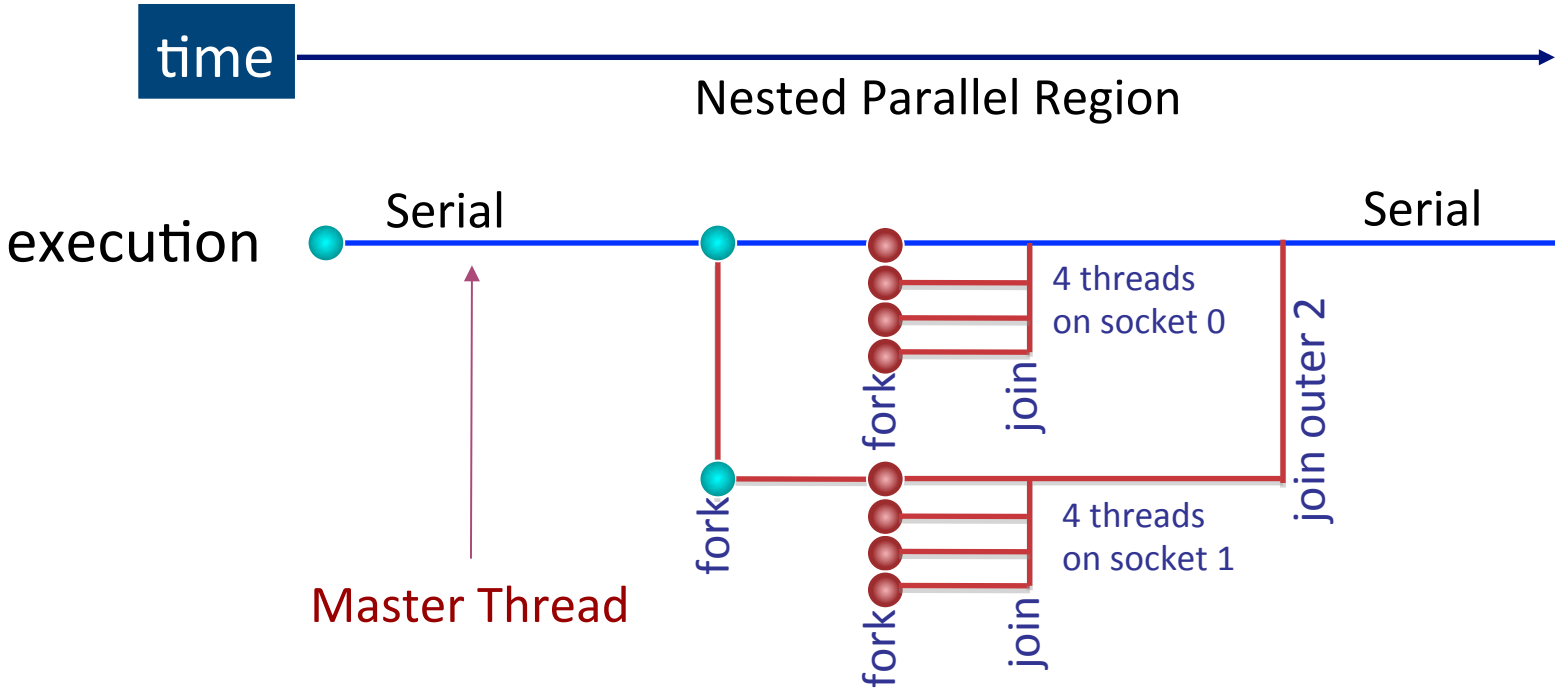
#pragma omp task [*clause*[[,]*clause*] ...]

structured-block

where clause can be

- Data scoping clauses
  - **shared (list), private (list), firstprivate (list), default(shared | none)**
- Scheduling clauses
  - **untied**
- Other clauses
  - **if (expression)**

Loop Nesting



While OpenMP 3.0 supports nested parallelism, many implementations may ignore the nesting by serializing the inner parallel regions

C/C++

Loop Collapse

- Allow collapsing of perfectly nested loops

- Will form a single loop and then parallelize it:

```
#pragma omp parallel do collapse(2)
for(i=0;i<n;i++){
   for(j=0;j<n;j++){
      .....
   }
}
```

F90

Loop Collapse

- Allow collapsing of perfectly nested loops
- Will form a single loop and then parallelize it:

```
!$omp parallel do collapse(2)
do i=1,n
   do j=1,n
      .....
   end do
end do
```

References

- http://www.openmp.org/

- *Parallel Programming in OpenMP*, by Chandra,Dagum, Kohr, Maydan, McDonald, Menon

- *Using OpenMP*, by Chapman, Jost, Van der Pas (OpenMP2.5)

- http://www.nic.uoregon.edu/iwomp2005/iwomp2005_tutorial_openmp_rvdp.pdf

- http://webct.ncsa.uiuc.edu:8900/public/OPENMP/

FOR FORTRAN USERS
and backups

Parallel Regions & Modes

There are two OpenMP "modes"

- ***static*** mode
  - Fixed number of threads -- set in the **OMP_NUM_THREADS** env.

  Or the threads may be set by a function call (or clause) inside the code:

  - **omp_set_num_threads** **runtime function**
    num_threads(#) **clause**
- ***dynamic*** mode:
  - Number of threads can change under OS control from one parallel region to another using:

***Note**: the user can only define the maximum number of threads, compiler can use a smaller number*

Default variable scoping (Fortran example)

```
Program Main
Integer, Parameter :: nmax=100
Integer :: n, j
Real*8  :: x(n,n)
Common /vars/ y(nmax)
...
n=nmax; y=0.0
!$OMP Parallel do
   do j=1,n
   call Adder(x,n,j)
   end do
...
End Program Main
```

```
Subroutine Adder(a,m,col)
Common /vars/ y(nmax)
SAVE array_sum
Integer :: i, m
Real*8  :: a(m,m)

do i=1,m
   y(col)=y(col)+a(i,col)
end do
array_sum=array_sum+y(col)

End Subroutine Adder
```

Default data scoping in Fortran (cont.)

Variable	Scope	Is use safe?	Reason for scope
n	shared	yes	declared outside parallel construct
j	private	yes	parallel loop index variable
x	shared	yes	declared outside parallel construct
y	shared	yes	common block
i	private	yes	parallel loop index variable
m	shared	yes	actual variable *n* is shared
a	shared	yes	actual variable *x* is shared
col	private	yes	actual variable *j* is private
array_sum	shared	no	declared with SAVE attribute

Kent Milfeld

Lars Koesterke

Antonio Gomez

milfeld/lars/agomez@tacc.utexas.edu

For more information:

www.tacc.utexas.edu