TACC
TEXAS ADVANCED COMPUTING CENTER
WWW.TACC.UTEXAS.EDU

PRACE

XSEDE
Extreme Science and Engineering Discovery Environment

Introduction to OpenMP Programming

IHPCSS
July 7-12, 2019

PRESENTED BY:

Kent Milfeld / Lars Koesterke
milfeld@tacc.utexas.edu
lars@tacc.utexas.edu

RIKEN

R-CCS

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Lecture and Lab slides available at: tinyurl.com/tacc-2019-ihpcss
tinyurl.com/tacc-2019-ihpcss

Outline

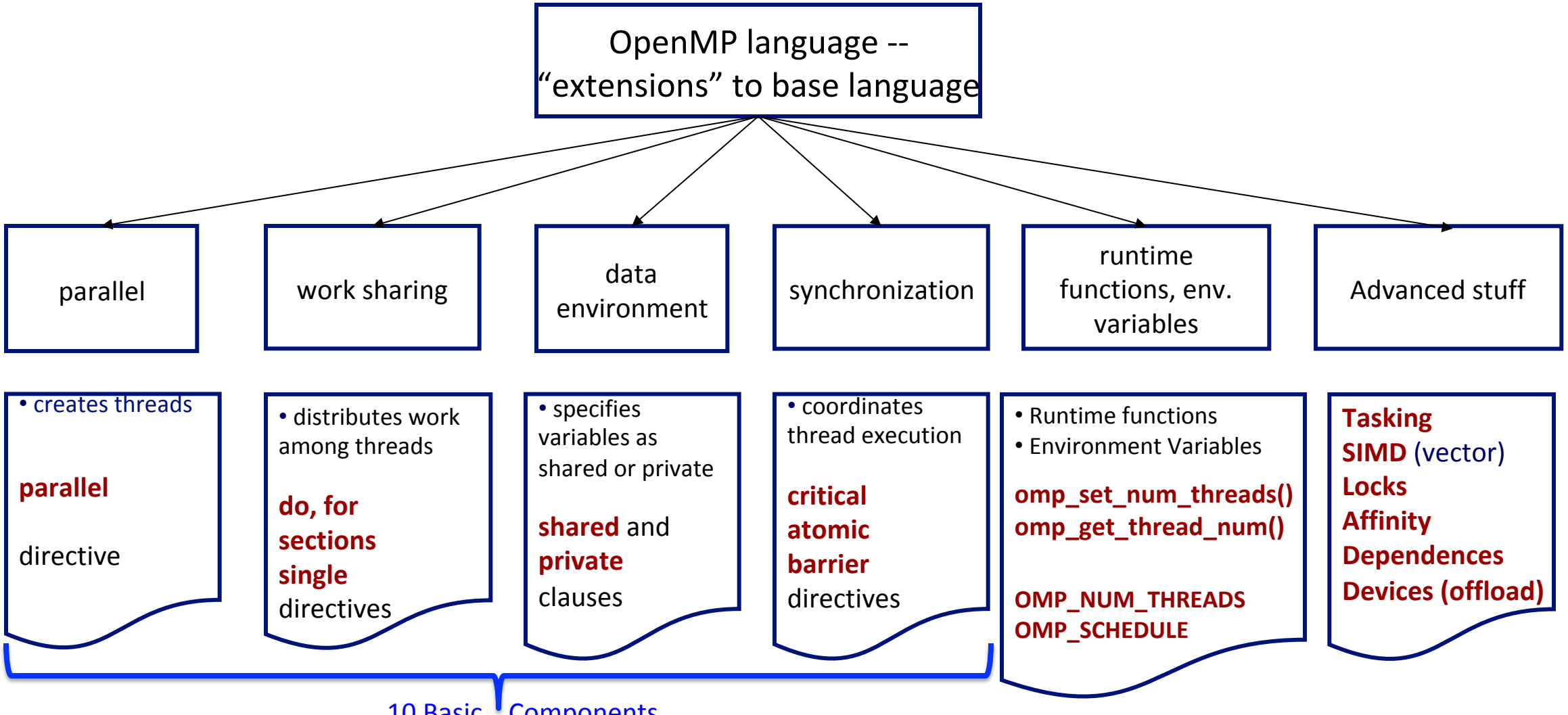
OpenMP Directive Review – The Concepts

- Execution Model: Parallel Fork-Join
- Directive Syntax
- Memory Model
- Using OpenMP Directives
- Worksharing
- Directive Scope
- Clauses
- Thread Scheduling
- Data Model / Data Environment
- Synchronization
- API functions / Environment Variables

OpenMP?

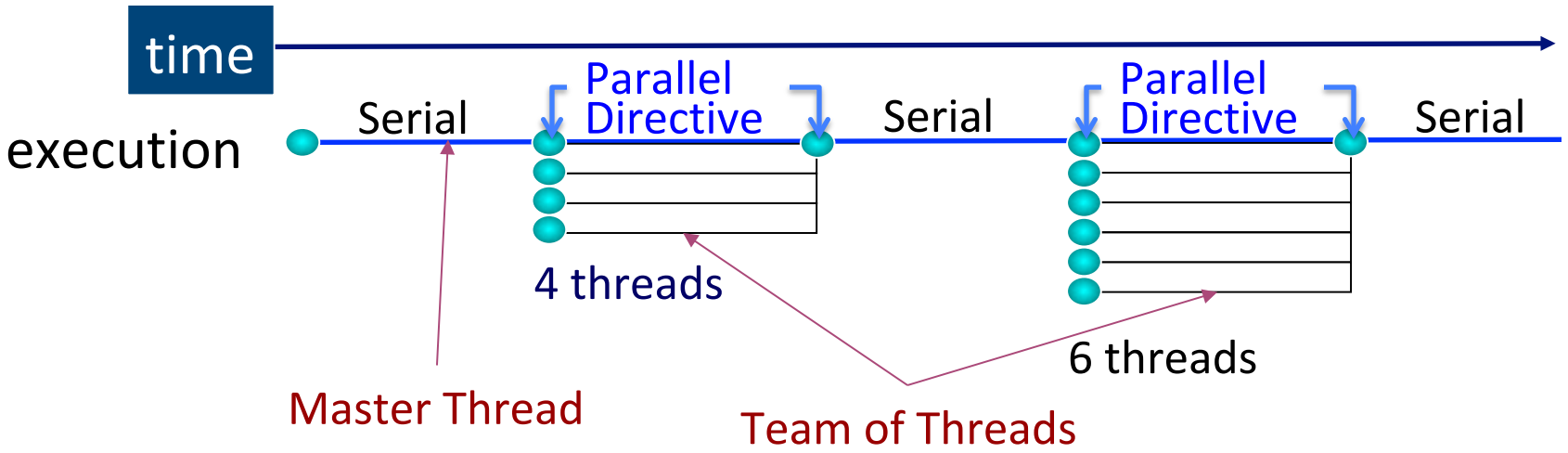
- OpenMP stands for **Open Multi-Processing**
- Three primary components:
  - Directives to the Compiler within code (pragmas in C/C++, comments in Fortran)
  - Runtime Library Routines
  - Environment Variables
- http://www.openmp.org/
  - *OpenMP 5.0 Complete Specifications* (Nov. 2018)
  - *OpenMP 4.5 Complete Specifications* (Nov. 2015)
  - *OpenMP Examples* 4.5 (Nov. 2015)
  - OpenMP Summary Card, *Reference Guide* (C/C++/Fortran)

OpenMP Constructs



Execution Model

- Program begins as a single process: master thread executes serially.
- At a parallel directive (construct) a team of threads is forked.
- At end of parallel region, (team of) threads synchronize and terminate (join) but master continues.



OpenMP Directive Scope

- OpenMP directives are are formed with comments/pragmas in the source:

F90 : **!$omp ...**
C/C++ : **#pragma omp ...**

Parallel Region

```
program main                                F
...
!$omp parallel
  call foo(...)
!$omp end parallel

!$omp parallel

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

end program
```

Applies to next statement

Parallel Region

```
int main(){                                 C/C++
  ...
  #pragma omp parallel
  foo(...);


  #pragma omp parallel
  {
    foo(...)
    bar(...)
  }

}
```

Applies to next block {...}

OpenMP Syntax

- Compiler directive syntax:

#pragma omp *construct* [*clause* [[,]*clause*]…] **C**

!$omp *construct* [*clause* [[,]*clause*]…] **F90**

- Example

Fortran

```
print*,"serial"

!$omp  parallel num_threads(4)
   …

!$omp end parallel

print*,"serial"
```

C/C++

```
printf("serial\n");

#pragma omp parallel num_threads(4)
{
    …
}

printf("serial\n");
```

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*-- Causing results change with different thread scheduling.

- Use mutual exclusion type 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 local changes to data (writes) seen by all threads.

C/C++

Parallel Regions

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

Line 1 Team of threads formed at beginning of parallel region

Lines 3-4 Each thread executes code block

No branching into or out of a parallel region

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

How to make individualized work: 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 beginning of parallel region.

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

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

How to make individualized work: use the thread number to divide work among threads.

C/C++

Parallel Region & Number of Threads

- For example, to create a 4-thread Parallel region.

```
gcc -fopenmp code.c
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 parallel region
- Thread numbers range from 0 to Nthreads-1
- Each thread calls foo(ID,A) with a different `ID {= 0,1,2,or 3}`

F90

Parallel Region & Number of Threads

- For example, to create a 4-thread Parallel region.

```
gcc -fopenmp code.c
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 parallel region
- Thread numbers range from 0 to Nthreads-1
- Each thread calls foo(ID, A) with a different **ID {= 0, 1, 2 or 3}**

Parallel Region & Worksharing

Use OpenMP directives to specify Worksharing inside a Parallel Region

Parallel Region

```
parallel


end parallel
```

Code block	Each Thread Executes	Unit of Work
do / for	Worksharing	loop
sections	Worksharing	several blocks
single	Worksharing (one thread)	single block

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

Work-sharing Directive assigns threads to unit(s) of work.
There is an implied barrier at the end of a worksharing construct!

C/C++

Worksharing: 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];
```

Or

```
#pragma omp parallel 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

Worksharing: 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
```

Or

```
!$OMP PARALLEL DO
do i=1,N
   a(i) = b(i) + c(i)
enddo
```

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.

C/C++

Worksharing: 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 Only **One** thread works on a section.

Line 9 End of parallel sections with an implied barrier.

Scales only to the number of sections.

F90

Worksharing: 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
```

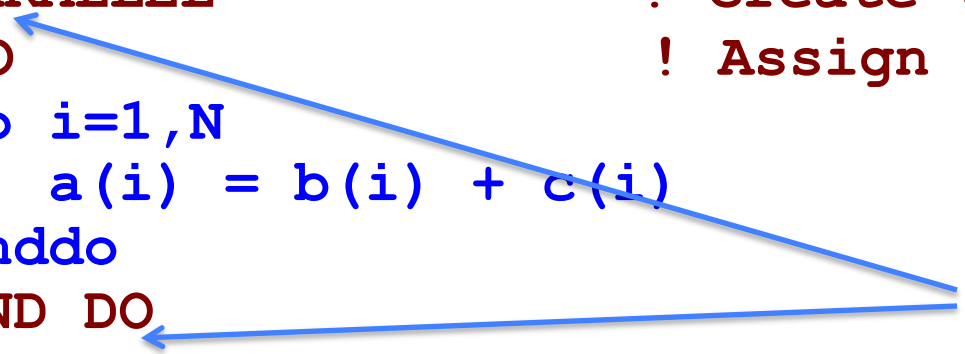
Line 1 Team of threads formed (parallel region).
Line 3-9 Only **One** thread works on a section.
Line 10 End of parallel sections with an implied barrier.

Scales only to the number of sections.

Question

```
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
```



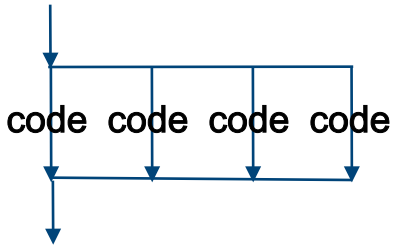
But, how does all this work
if we need other task parallel work here?

OpenMP Parallel + Worksharing

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

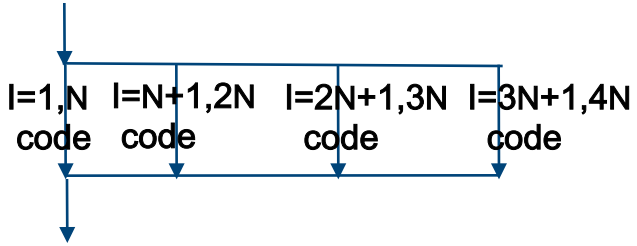
4 Threads

```
!$OMP PARALLEL
  {code}
!$OMP END PARALLEL
```



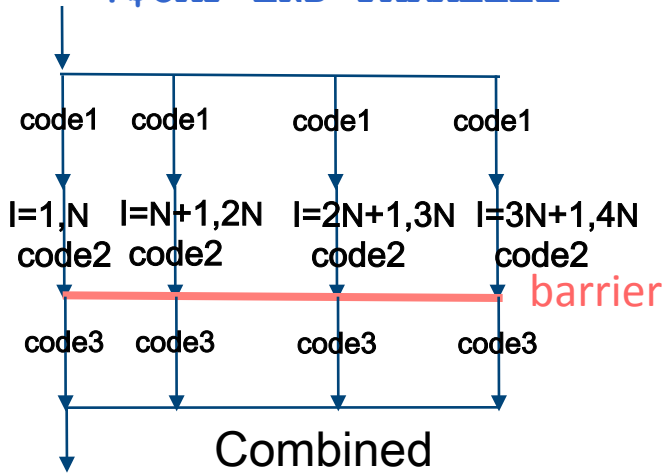
Replicated

```
!$OMP PARALLEL
!$OMP DO
  do i = 1,N*4
    {code}
  end do
!$OMP END PARALLEL DO
```



Work-Sharing

```
!$OMP PARALLEL
  {code1}
!$OMP DO
  do i = 1,N*4
    {code2}
  end do
!$OMP END DO
  {code3}
!$OMP END PARALLEL
```



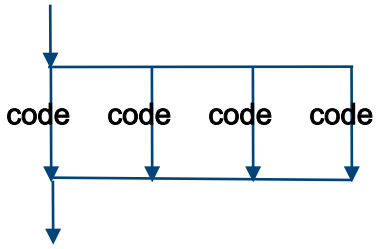
Combined

OpenMP Parallel + Worksharing (sections)

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

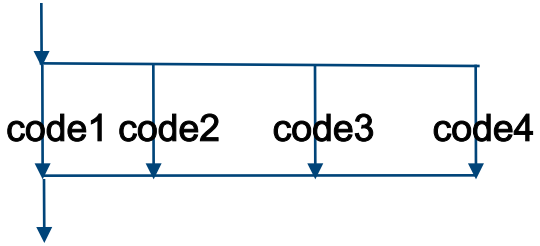
4 Threads

```
!OMP PARALLEL
  {code}
!OMP END PARALLEL
```



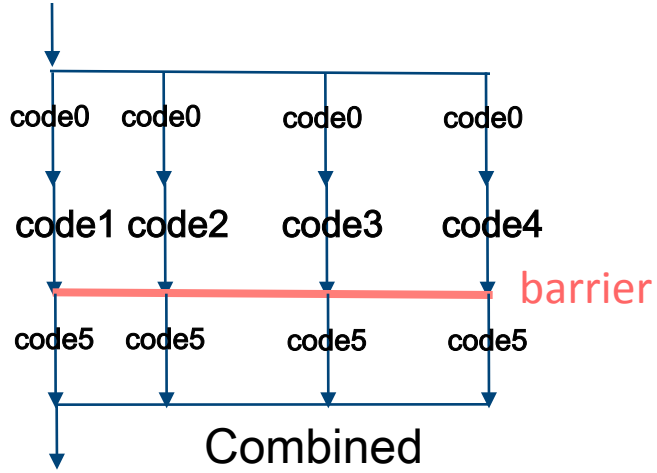
Replicated

```
!OMP PARALLEL sections
!OMP section
      {code1}
!OMP section
      {code2}
!OMP section
      {code3}
!OMP section
      {code4}
!OMP END PARALLEL sections
```



Work-Sharing

```
!OMP PARALLEL
  {code0}
!OMP sections
!OMP section
   {code1}
  ...! +4 more
!OMP END sections
  {code5}
!OMP END PARALLEL
```

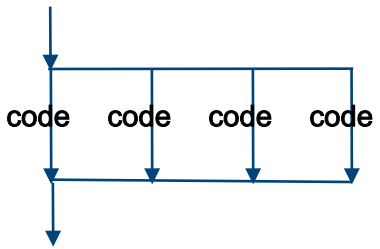


Combined

OpenMP Parallel + Worksharing

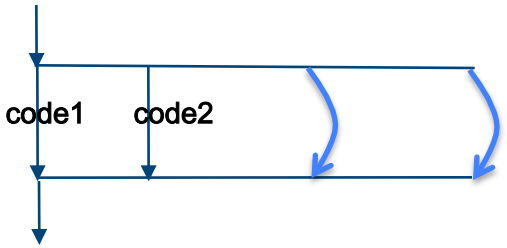
4 Threads

```
#pragma PARALLEL
  {code}
```



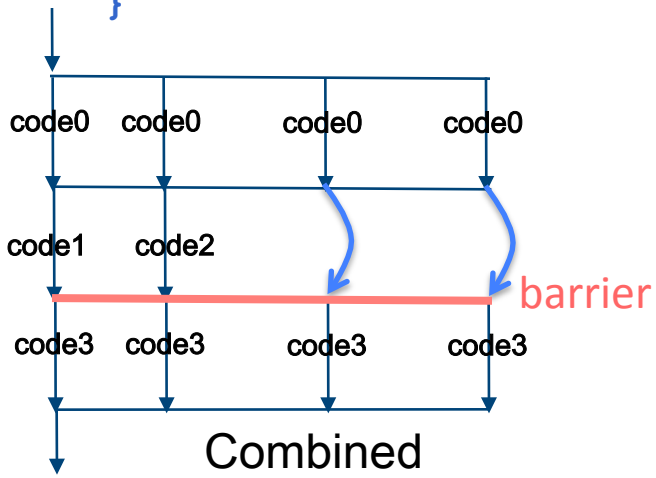
Replicated

```
#pragma PARALLEL sections
{
    #pragma section
        {code1}
    #pragma section
        {code2}
}
```



Work-Sharing

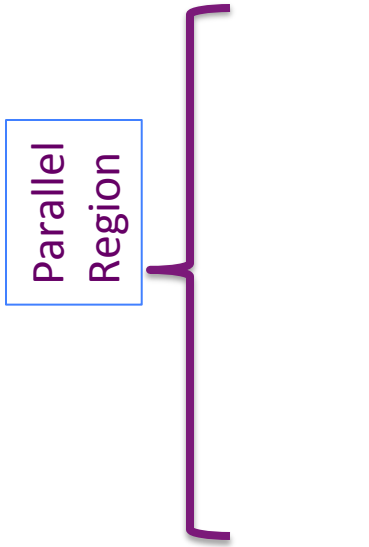
```
#pragma PARALLEL
{
    {code0}
    #pragma sections
    {
        #pragma section
          {code1}
        #pragma section
          {code2}
    }
    {code3}
}
```



Combined

OpenMP Directive Scope

- construct – the lexical extent of executable directive
- region – all code encountered in a construct (construct + routines)



Fortran

```
program ex1
use  omp_lib

!$omp parallel
   call foo(omp_get_thread_num())
!$omp end parallel

end program ex1

subroutine foo(id)
  write( *, '(" thrd_id ",i0.3)' ),id
end subroutine
```

C/C++

```
#include "omp.h"
void foo(int);
int main(){
  #pragma omp parallel
  {
     foo(omp_get_thread_num());
  }
}

void foo(int id){
  printf("thrd_id %0.3d\n",id);
}
```

OpenMP Clauses

Clauses control the behavior of an OpenMP directive:

Control	Clause
Schedule for *for/do* worksharing	*schedule( )*
Data Scoping	*private( ), shared( ), default( )*
Initialization	*firstprivate( ), copyin( )*
Parallelize a region or not	*if( )*
Number of threads to use	*num_threads( )*

Schedule Clause for loop worksharing

E.G. Fort.

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

Type	Description
schedule (static)	Each CPU receives one set of contiguous iterations
schedule (static, C)	Iterations are divided round-robin fashion in chunks of size C iterations
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
schedule (auto)	Let the runtime decide

Before Execution:

```
export OMP_SCHEDULE="static,100"
```

E.G. C

```
#pragma omp for schedule(runtime)
  for(i=0;i<N;i++) A[i]=B[i]+C[i];
```

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`

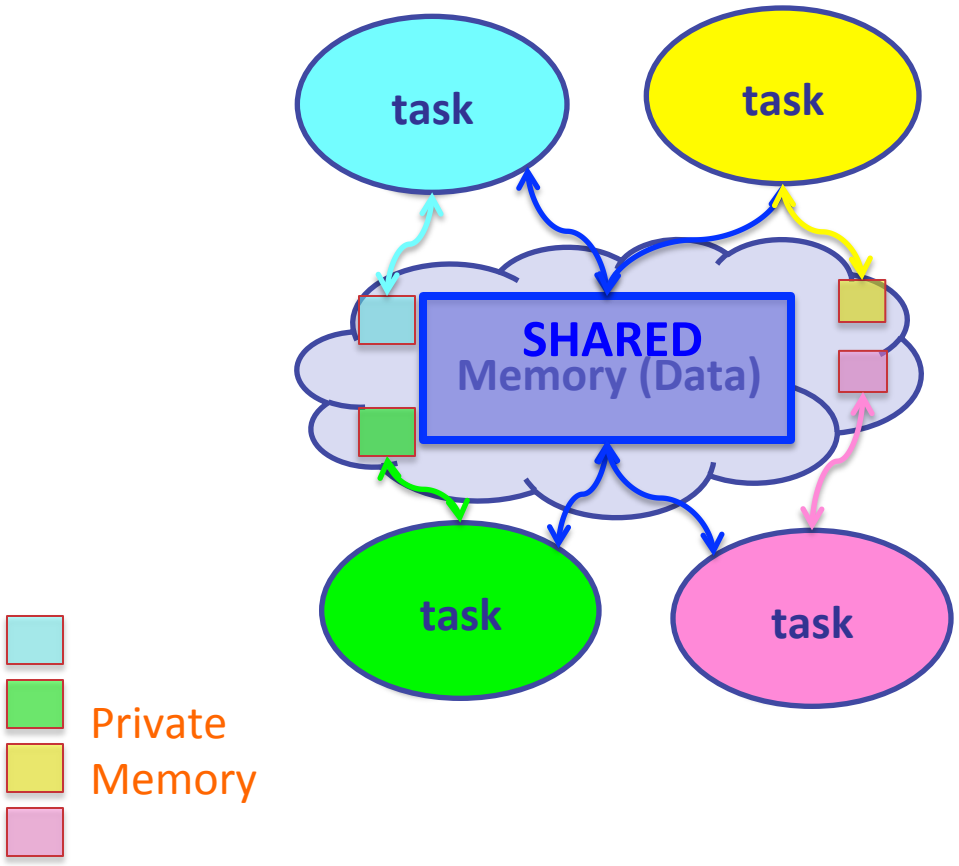
Round Robin"
Green= Round 1
Blue= Round 2

Comparison of Scheduling Options

type	compute overhead	distribution	chunk size	static or dynamic
static	lowest	partitioned	N/P	static
static, c	low	round-robin	C	static
dynamic [, c]	medium	simple dynamic	C	dynamic
guided [, c]	Less than medium	guided	decreasing unassigned/P	dynamic
runtime	varies	runtime	varies	varies

[, C] = optional
default = 1

Data Model – for parallel region



- Threads Execute on Cores/HW-threads
- In a parallel region, team threads are assigned (tied) to implicit tasks to do work. Think of tasks and threads as being synonymous.
- Tasks by "default" share memory declared in scope before a parallel region.
- Data: shared or private
  - Shared data: accessible by all tasks
  - Private data: only accessible by the owner task

OpenMP Data Environment

- The following clauses control the data-sharing attributes of variables:

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 function|subroutine, 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

```
#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

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

OK to be explicit;
but not necessary.

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

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

C/C++

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.

Reduction

- 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

- 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 (not explored here)
- Low-Level Synchronization
  - locks (no explored here)

C/C++

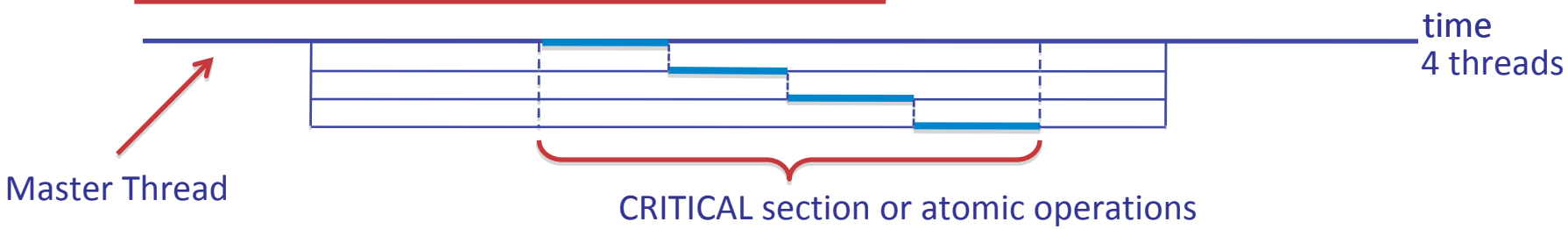
Synchronization: Critical/Atomic Directives

- 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.



F90

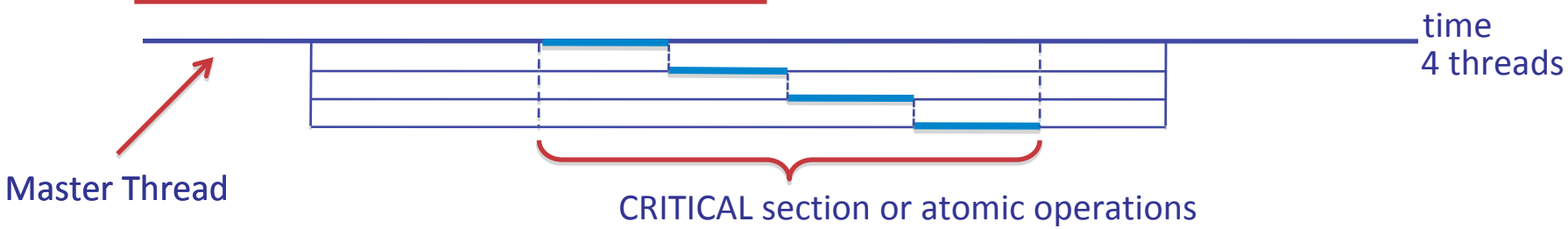
Synchronization: Critical/Atomic Directives

- 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
```

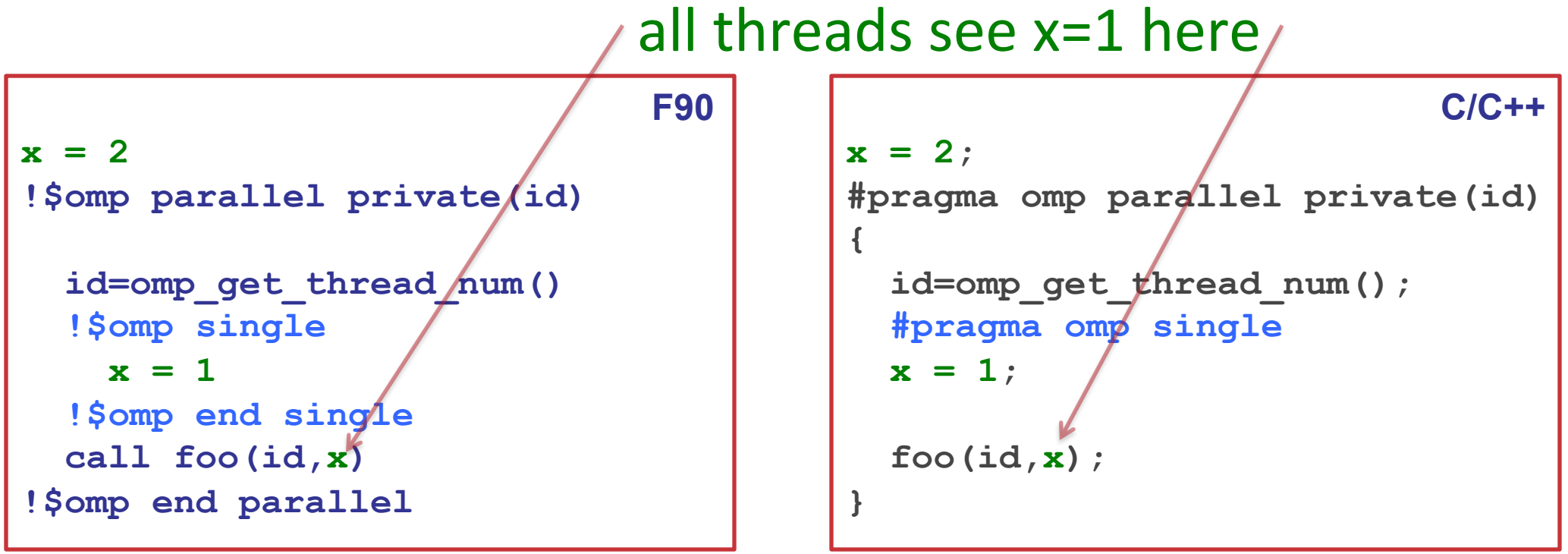
```
!$omp parallel shared(sum)
...

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

Atomic has read, write, update, capture clauses.



Single Construct

- Single: Any single thread executes the construct. Since Single is worksharing there is an implied barrier.



Master Construct

- Master: Only the master executes the construct. There is no implied barrier for this construct.

race condition

```
x=2                                                    F90
!$omp parallel private(id)
  id=omp_get_thread_num()
  !$omp master
    x = 1
  !$omp end master
  call foo(id,x)
!$omp end parallel
```

race condition

```
x=2;                                                   C/C++
#pragma omp parallel private(id)
{
  id=omp_get_thread_num();
  #pragma omp master
  x = 1;

  foo(id,x);
}
```

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
```

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 for subsequent parallel regions

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);
```

Progression

Concept	What to learn	Level	Optimize
Setup	How to compile OMP_NUM_THREADS	Basic	1 thread per 'core'
Parallel region	Forking/joining threads	Easy	Minimize number of fork/join
Work-sharing/replicated work	What do the threads do? 'omp do/for'	Work-sharing: easy Replicated: medium	Optimize scheduling Remove implicit barriers
Avoiding race conditions	----------	Will take effort!	----------
- Private variables	Why/how to shelter data	medium	
- reduction	Condensing a result from pieces	medium	
- Critical/atomic	All threads, but one thread at a time	harder	Do not serialize everything
- Single/master	One thread, and only one thread	harder	See 'min. fork/join'
Advanced	----------		
- Hybrid	MPI + OpenMP	medium	Interplay MPI/OpenMP
- Thread/memory pinning	Affinity	medium	Utilize all cores
- SIMD	Vectorization with OpenMP	hard	Utilize vector lanes
- Tasking	Irregular problems	hard	

OpenMP 3.0 and above

- First update to the spec since 2005
- Tasking: move beyond loops with generalized tasks and support complex and dynamic control flows (dependences), priority and other features.
- Loop collapse: combine nested loops automatically to expose more concurrency
- Nested parallelism support: better definition of and control over nested parallel regions, and new API routines to determine nesting structure
- OpenMP Affinity
- User-defined reductions
- Specify SIMD instructions for loop
- Offloading to devices (GPUs)
- Provide features implementation must provide (requires) and conditional directives (metadirective)

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)
- ***Using OpenMP – The Next Step Affinity, Accelerators, Tasking and SIMD***, Rudd van der Pas, Eric Stotzer, and Christian Terboven

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

Kent Milfeld

Lars Koesterke

milfeld|lars@tacc.utexas.edu

For more information:

www.tacc.utexas.edu

Miscellaneous and backups

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

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++){
      .....
   }
}
```

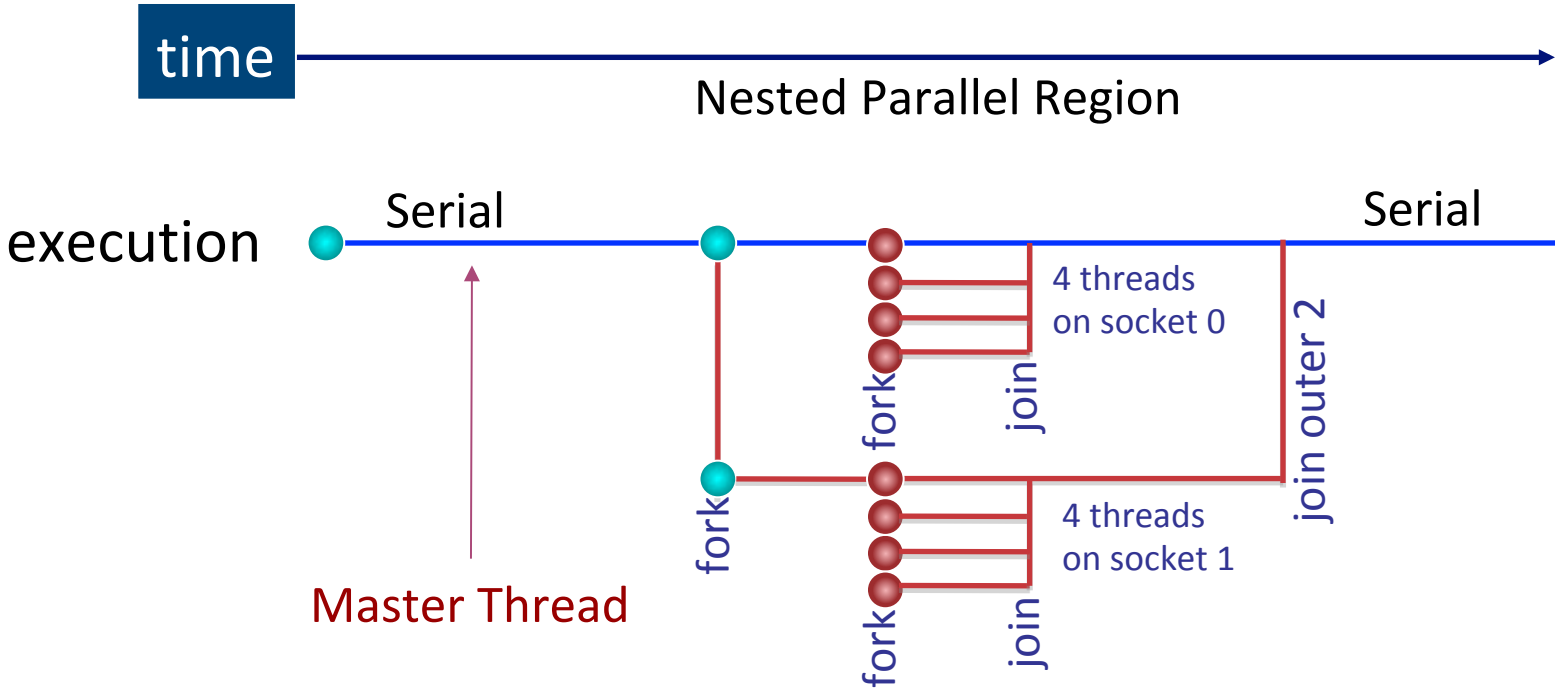
F90

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
```

Loop Nesting



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

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*