Introduction to Programming with OpenMP

Lars Koesterke

ASU: April 1, 2014

CSU: April 4, 2014

THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

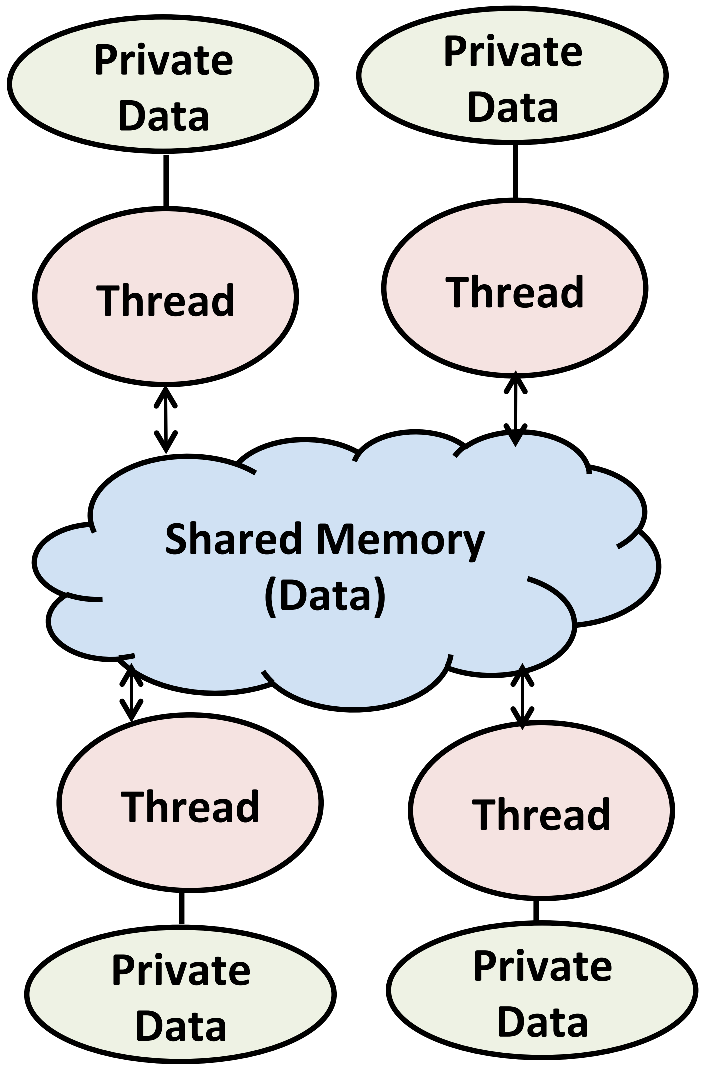
Outline

- What is OpenMP?
- How does OpenMP work?
  - Architecture
  - Fork-Join model of parallelism
  - Communication
- OpenMP Syntax
  - Compiler Directives
  - Runtime Library Routines
  - Environment variables
- What's new? OpenMP 3.1

What is OpenMP?

- OpenMP stands for **Open Multi-Processing**
- An Application Programming Interface (API) for developing parallel programs for shared memory architectures
- Three primary components of the API are:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- Standard specifies C, C++, and Fortran Directives & API
- http://www.openmp.org/ has the specification, examples, tutorials and documentation

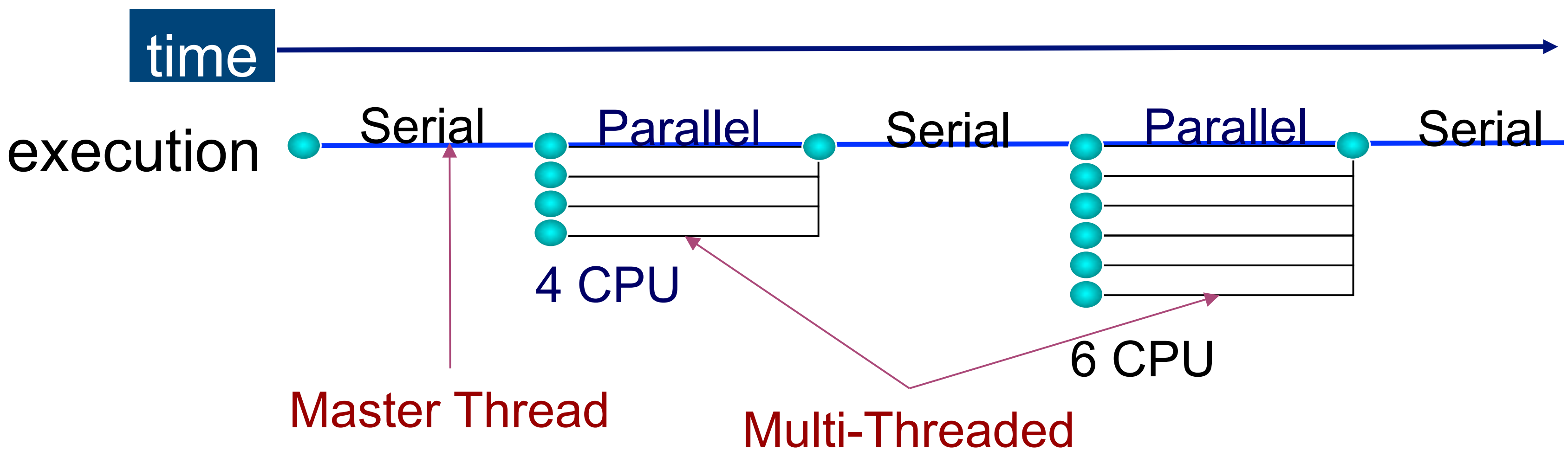
Architecture



- 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
- Data transfer is transparent to the programmer

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
- Master thread creates a team of parallel threads (fork) that simultaneously execute statements in the parallel region
- After executing the statements in the parallel region, team threads synchronize and terminate (join) but master continues



How do threads communicate?

or better:

How do threads synchronize their work

- Every thread has access to "global" memory (shared)
- All threads share the same address space
- Threads communicate by reading/writing to the global memory
- Simultaneous updates to shared memory can create a *race condition*. Results change with different thread scheduling
- Use mutual exclusion to avoid data sharing - but don't use too many because this will serialize performance

OpenMP Syntax

- Most of the constructs in OpenMP are compiler directives

  #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**

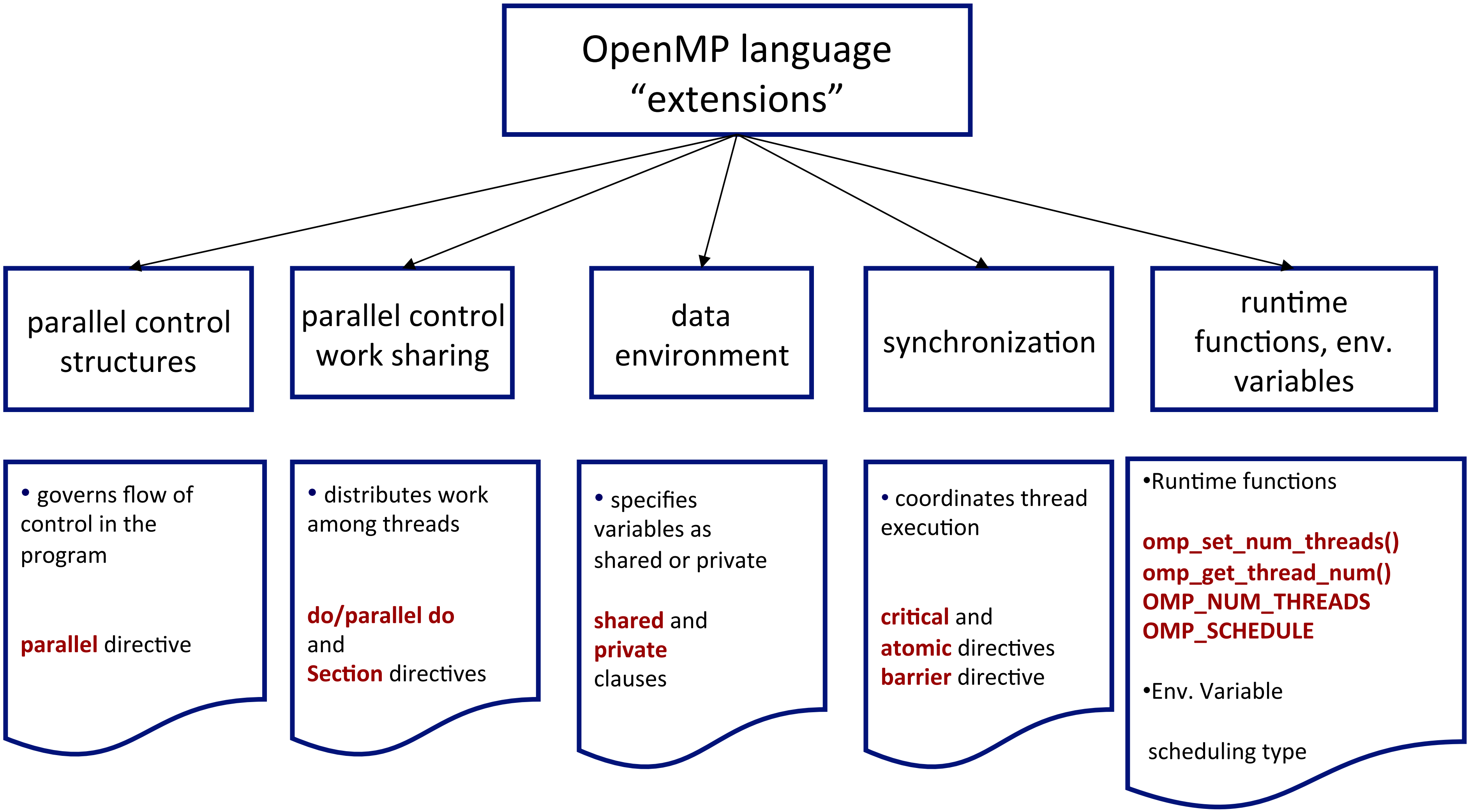
- Function prototypes and types are in the file:

  #include <omp.h> **C**

  use omp_lib **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 Constructs



OpenMP Directives

- OpenMP directives are comments in source code that specify parallelism for shared memory machines

  FORTRAN : directives begin with the **!$OMP**, **C$OMP** or ***$OMP** sentinel.

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

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

- Parallel regions are marked by enclosing parallel directives
- Work-sharing loops are marked by parallel do/for

Fortran

```
!$OMP parallel
  ...
!$OMP end parallel

!$OMP parallel do
  do ...; enddo
!$OMP end parallel do
```

C/C++

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


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

Parallel Region & Work-Sharing

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

```
Parallel

End Parallel
```

Code block	**Each Thread Executes**
DO	Work Sharing
SECTIONS	Work Sharing
SINGLE	One Thread (Work sharing)
CRITICAL	One Thread at a time

Parallel DO/for
Parallel SECTIONS

Work-Sharing
Parallel Region

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

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.

Parallel Region & Number of Threads

- For example, to create a 10-thread Parallel region:

```
double A[1000];
omp_set_num_threads(10);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    foo(ID, A);
}
```

But we need to make ID private to the thread– later…

- Each thread redundantly executes the code within the structured block
- Each thread calls foo(ID,A) for **ID** = 0 to **9**

Parallel Region & Number of Threads

- For example, to create a 10-thread Parallel region:

```
real :: A(1000); integer :: ID
call omp_set_num_threads(10)
!$omp parallel
ID = omp_get_thread_num()
call foo(ID, A);
!$omp end parallel
```

But we need to make ID private to the thread– later…

- Each thread redundantly executes the code within the structured block
- Each thread calls foo(ID,A) for **ID** = 0 to 9

Parallel Regions & Modes

There are two OpenMP "modes"

- ***static*** mode (This is what you will be using!)
  - 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 (This is something for later, if needed at all)
  - 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*

Work-Sharing: Loop

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

Line 1 Team of threads formed (parallel region).

Line 2-4 Loop iterations are split among threads.

Line 5 (Optional) end of parallel loop (implied barrier at enddo).

Each loop iteration must be independent of other iterations.

Work-Sharing: Loop

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

Line 1 Team of threads formed (parallel region).

Line 2-5 Loop iterations are split among threads.
implied barrier at enddo

Each loop iteration must be independent of other iterations.

Work-Sharing: Sections

```
1   !$OMP PARALLEL SECTIONS
2   !$OMP SECTION
3         call work_1()
4   !$OMP SECTION
5         call work_2()
6   !$OMP END SECTIONS
```

Line 1   Team of threads formed (parallel region).

Line 2-5 One thread is working on each section.

Line 6   End of parallel sections with an implied barrier.

Scales only to the number of sections.

Work-Sharing: Sections

```
1   #pragma omp 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.

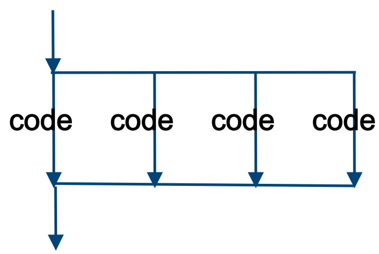
Line 9   End of parallel sections with an implied barrier.

Scales only to the number of sections.

OpenMP Parallel Constructs

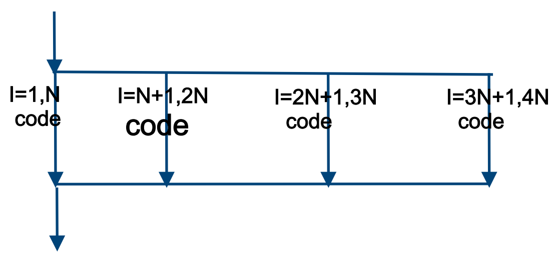
Replicated : Work blocks are executed by all threads.
Work-Sharing : Work is divided among threads.

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



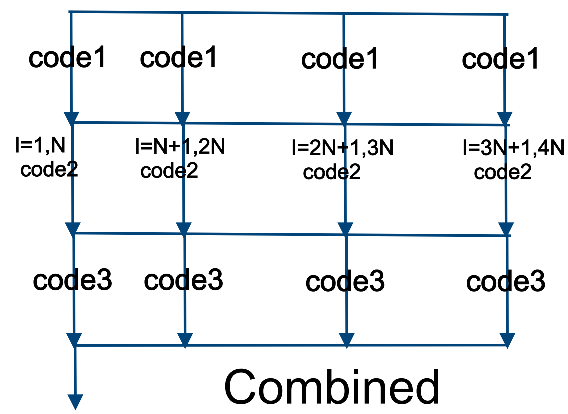
Replicated

```
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
  {code3}
END PARALLEL
```



Combined

OpenMP Clauses

Clauses control the behavior of an OpenMP directive:

1. Data scoping (Private, Shared, Default)
2. Schedule (Guided, Static, Dynamic, etc.)
3. Initialization (e.g. COPYIN, FIRSTPRIVATE)
4. Whether to parallelize a region or not (if-clause)
5. Number of threads used (NUM_THREADS)

Schedule Clause

schedule(static)

Each CPU receives one set of contiguous iterations

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

Comparison of Scheduling Options

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

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

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

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

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

OpenMP Data Environment

- Data scoping clauses control the sharing behavior of variables within a parallel construct.
- These include **`shared, private, firstprivate, lastprivate, reduction`** clauses

Default variable scope:

1. Variables 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), unless scoped otherwise
4. Default scoping rule can be changed with **`default`** clause

Private & Shared Data

SHARED - Variable is shared (seen) by all processors.

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

Defaults: All DO LOOP indices are private, all other variables are shared.

```
!$OMP PARALLEL DO SHARED(A,B,C,N) PRIVATE(i)
      do i=1,N
         A(i) = B(i) + C(i)
      enddo
!$OMP END PARALLEL DO
```

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 & Shared Data

shared - Variable is shared (seen) by all processors

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];
         }
```

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 processor would be writing and reading to/from the same memory location.

```
!$OMP PARALLEL DO SHARED(A,B,C,N) PRIVATE(temp,i)
      do i=1,N
         temp = A(i)/B(i)
         C(i) = temp + cos(temp)
      enddo
!$OMP END PARALLEL DO
```

- 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.

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 and 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 DO is complete.
- A **firstprivate(temp)** would copy the global temp value to each stack's temp.

Reduction

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

```
      real*8 asum, aprod
      asum  = 0.
      aprod = 1.
!$OMP PARALLEL DO REDUCTION(+:asum) REDUCTION(*:aprod)
      do i=1,N
         asum  = asum  + a(i)
         aprod = aprod * a(i)
      enddo
!$OMP END PARALLEL DO
      print*, asum, aprod
```

- Each thread has a private **ASUM** and **APROD**, initialized to the operator's identity, 0 & 1, respectively.
- **After the loop execution, the master thread collects the private values of each thread and finishes the (global) 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, aprod;
asum  = 0.;
aprod = 1.;
#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**

Synchronization

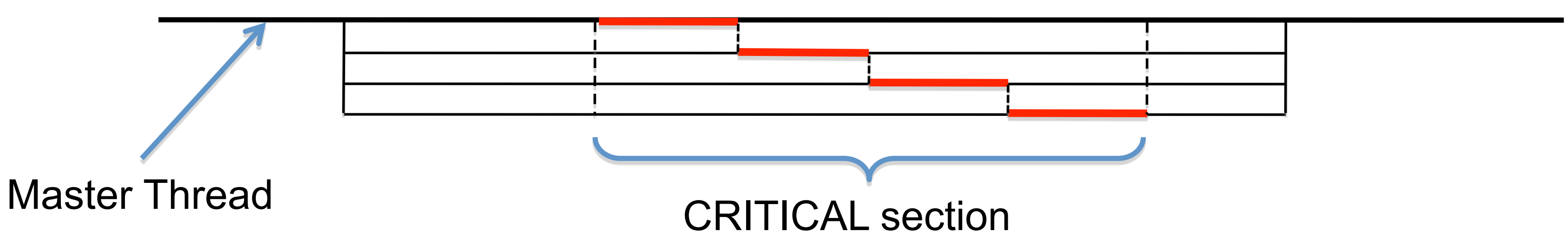
- Synchronization is used to impose order constraints and to protect access to shared data
- High-Level Synchronization
  - critical
  - atomic
  - barrier
  - ordered
- Low-Level Synchronization
  - locks

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 !$OMP ATOMIC directive if executing only one operation serially.

```
!$OMP PARALLEL SHARED(sum,X,Y)
 ...
!$OMP CRITICAL
   call update(x)
   call update(y)
   sum=sum+1
!$OMP END CRITICAL
...
!$OMP END PARALLEL
```

```
!$OMP PARALLEL SHARED(X,Y)
 ...
!$OMP ATOMIC
      sum=sum+1
...
!$OMP END PARALLEL
```

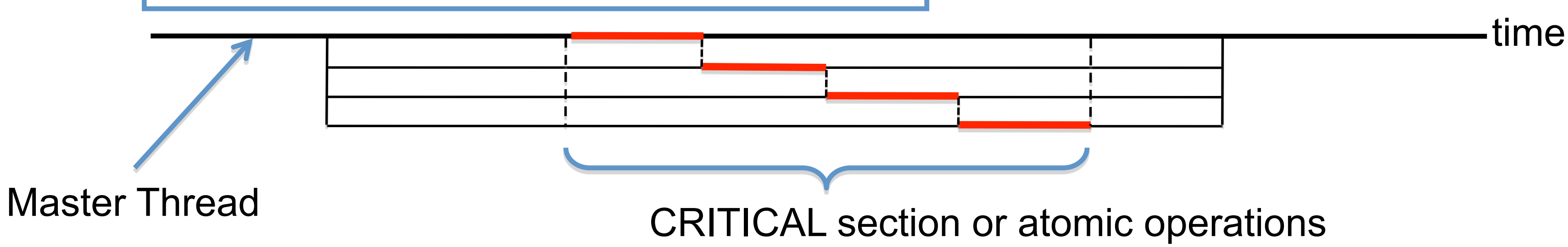


Synchronization: Critical/Atomic Directives

- 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 if executing only one operation serially

```
#pragma omp parallel shared(sum,x,y)
 ...
#pragma omp critical
   {
     update(x);
     update(y);
     sum=sum+1;
   }
...
!$OMP END PARALLEL
```

```
#pragma omp parallel shared(sum)
...
    {
#pragma omp atomic
        sum=sum+1;
        ...
    }
```

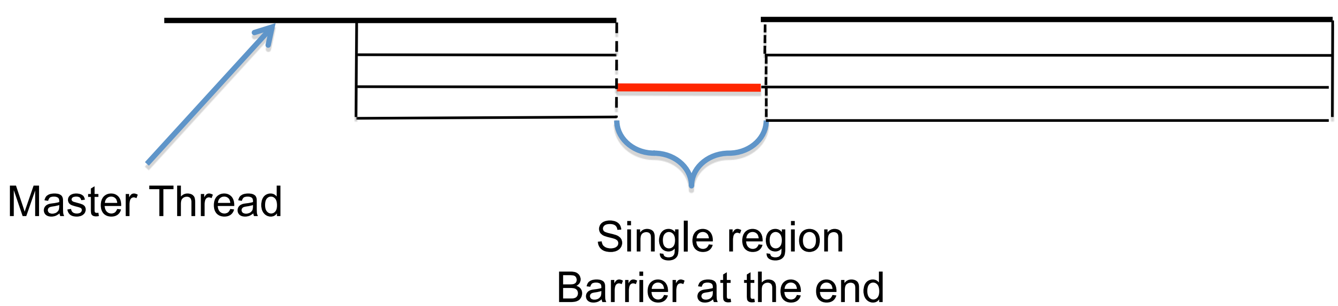


Synchronization: Single/Master Directives

- Only one thread executes the statements in the single/master region
- Single: An arbitrary thread is chosen and the is an implied barrier at the end of the single construct

```
!$OMP PARALLEL SHARED(sum,x,y)
 ...
!$OMP SINGLE
   icount = icount + 1
!$OMP END SINGLE
   call work1(x)
   call work2(y)
...
!$OMP END PARALLEL
```

```
#pragma omp parallel shared(sum,x,y)
 ...
#pragma omp single
   {
    icount = icount + 1
   }
     work1(x);
     work2(y);
...
!$OMP END PARALLEL
```

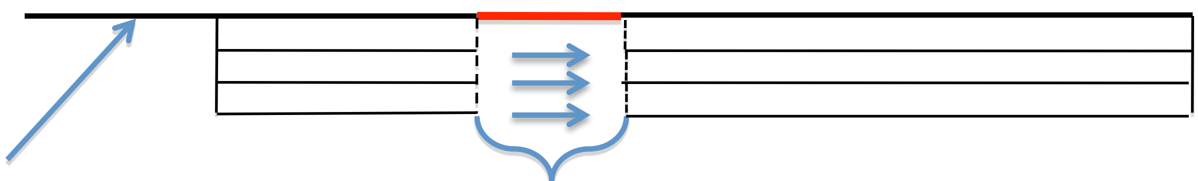


Synchronization: Single/Master Directives

- Only one thread executes the statements in the single/master region
- Single: An arbitrary thread is chosen and the is an implied barrier at the end of the single construct

```
!$OMP PARALLEL SHARED(sum,x,y)
 ...
!$OMP MASTER
   icount = icount + 1
!$OMP END MASTER
   call work1(x)
   call work2(y)
...
!$OMP END PARALLEL
```

```
#pragma omp parallel shared(sum,x,y)
 ...
#pragma omp master
   {
    icount = icount + 1
   }
     work1(x);
     work2(y);
...
!$OMP END PARALLEL
```



Master Thread

Master region
No barrier at the end, other threads jump ahead

Synchronization: Barrier

- Barrier: Each thread waits until all threads arrive

```
#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
```

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)
```

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
  print tmp;
}
```

```
!$omp parallel private (tmp)
!$omp do ordered reduction(+:countVal)
do i=1, n
  tmp = foo(i)
  !$omp ordered
  write (0,*) tmp
}
```

Mutual Exclusion Overhead

OMP exclusion directive	cycles
OMP_SET_LOCK	330
OMP_UNSET_LOCK	330
OMP_ATOMIC	480
OMP_CRITICAL	510

All measurements made in dedicated mode

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(dynamic,k)
      do i=1,m
         x(i)=y(i)+z(i)
      enddo
!$OMP END DO
!$OMP END PARALLEL
```

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(dynamic,k)
      {
         for (i=0; i<m; i++)
           {x[i]=y[i]+z[i];}
      }
     }
```

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
omp_get_dynamic()	True if dynamic threading is on
omp_set_dynamic()	Set state of dynamic threading (true/false)

Environment Variables

variable	description
OMP_NUM_THREADS int_literal	Set to default no. of threads to use
OMP_SCHEDULE “schedule[, chunk_size]”	Control how “omp for schedule(RUNTIME)” loop iterations are scheduled
OMP_DYNAMIC	TRUE/FALSE for enable/disable dynamic threading

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);
```

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

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

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

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
```

Tasks Parallelism

- Allows to parallelize irregular problems
  - Recursive loops
  - Unbounded algorithms
  - Threads can jump between tasks

What is a Task?

- A specific instance of executable code and its data environment, generated when a thread encounters a **task** construct or a **parallel** construct
- Tasks consist of
  - Code to execute
  - Data environment
  - Internal control variables (new from 2.5)
- Each encountering thread creates a new task which packages its own code and data
- Execution of the new task could be immediate, or deferred until later
- Can be nested into
  - Another task or a work sharing construct

What is a Task?

- Tasks have been fully integrated into OpenMP
- Note: OpenMP has always had tasks but they were never called that way before the 3.0 release!
  - Thread encountering **parallel** construct packages up a set of implicit tasks, one per thread
  - Team of threads is created
  - Each thread in team is assigned to one of the tasks (and tied to it)
  - Barrier holds original master thread until all implicit tasks are finished
- Now we have a way to create a task explicitly for the team to execute

Tasks: Usage

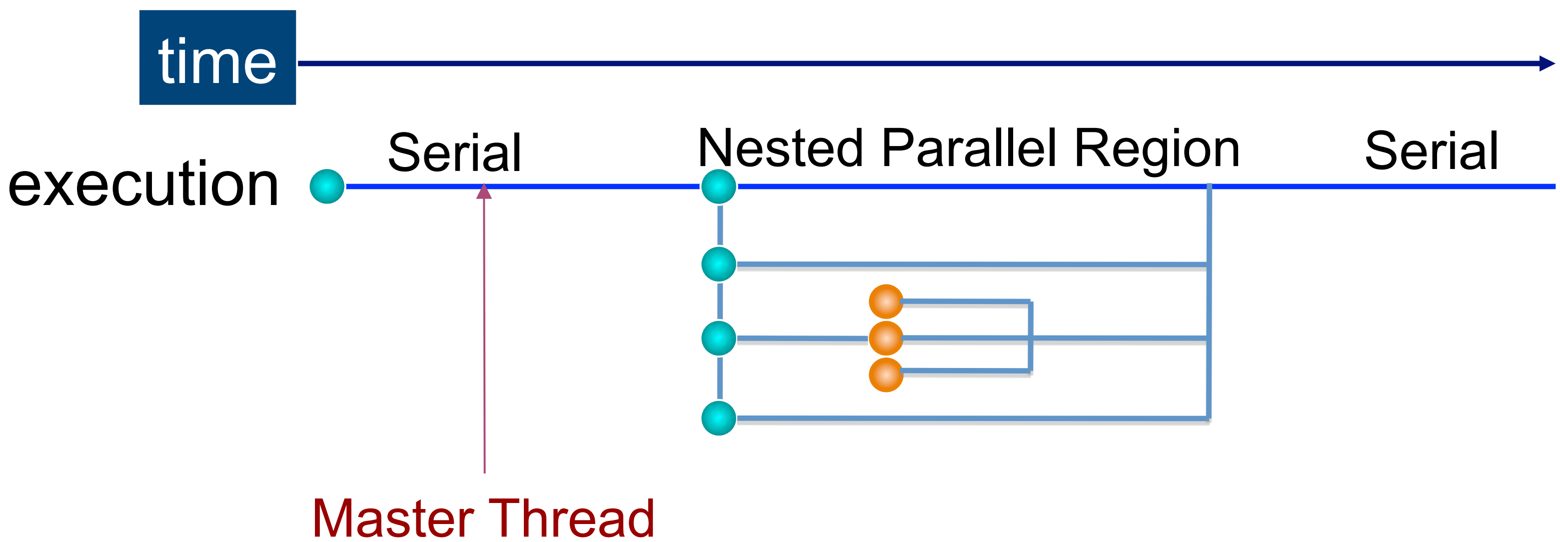
Task Construct:

```
#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

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/

Thank you very much

lars@tacc.utexas.edu

Please participate in our survey

http://bit.ly/CSUXSEDE

Additional material for Fortran Users

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

Workshare directive

- **WORKSHARE** directive enables parallelization of Fortran 90 array expressions and **FORALL** constructs

```
Integer, Parameter :: N=1000
Real*8             :: A(N,N), B(N,N), C(N,N)
!$OMP WORKSHARE
   A=B+C
!$OMP End WORKSHARE
```

- Enclosed code is separated into units of work
- All threads in a team share the work
- Each work unit is executed only once
- A work unit may be assigned to any thread

Reduction on array variables

- Supported in Fortran only!
- Array variables may now appear in the **REDUCTION** clause

```
Real*8 :: A(N), B(M,N)
Integer :: i, j
A(1:m) = 3.
!$OMP Parallel Do Reduction(+:A)
     do i=1,n
       A(1:m)=A(1:m)+B(1:m,i)
     end do
!$OMP End Parallel Do
```

- Assumed size and allocatable arrays are not supported
- Variable must be shared in the enclosing context