



OpenMP on Ranger and Stampede (with Labs)

Steve Lantz
Senior Research Associate
Cornell CAC

Parallel Computing at TACC: Ranger to Stampede Transition

November 6, 2012

Based on materials developed by Kent Milfeld at TACC



What is OpenMP?

- OpenMP is an acronym for **Open Multi-Processing**
- An Application Programming Interface (API) for **developing parallel programs in shared-memory architectures**
- Three primary components of the API are:
 - **Compiler Directives**
 - Runtime Library Routines
 - Environment Variables
- De facto standard -- specified for C, C++, and FORTRAN
- <http://www.openmp.org/> has the specification, examples, tutorials and documentation



Parallel Region: C/C++ and Fortran

```
1 #pragma omp parallel
2   { code block
3     a = work(...);
4   }
```

```
!$omp parallel
   code block
   call work(...)
!$omp end parallel
```

- Line 1 Team of threads is formed at parallel region
- Lines 2–3 Each thread executes code block and subroutine call, no branching into or out of a parallel region
- Line 4 All threads synchronize at end of parallel region (implied barrier)



OpenMP = Multithreading

- All about executing concurrent work (tasks)
 - Tasks execute independently
 - Tasks access the same shared memory
 - Shared variable updates must be mutually exclusive
 - Synchronization through barriers
- Simple way to do **multithreading** – run tasks on multiple cores/units
- Insert parallel directives to run tasks on concurrent threads

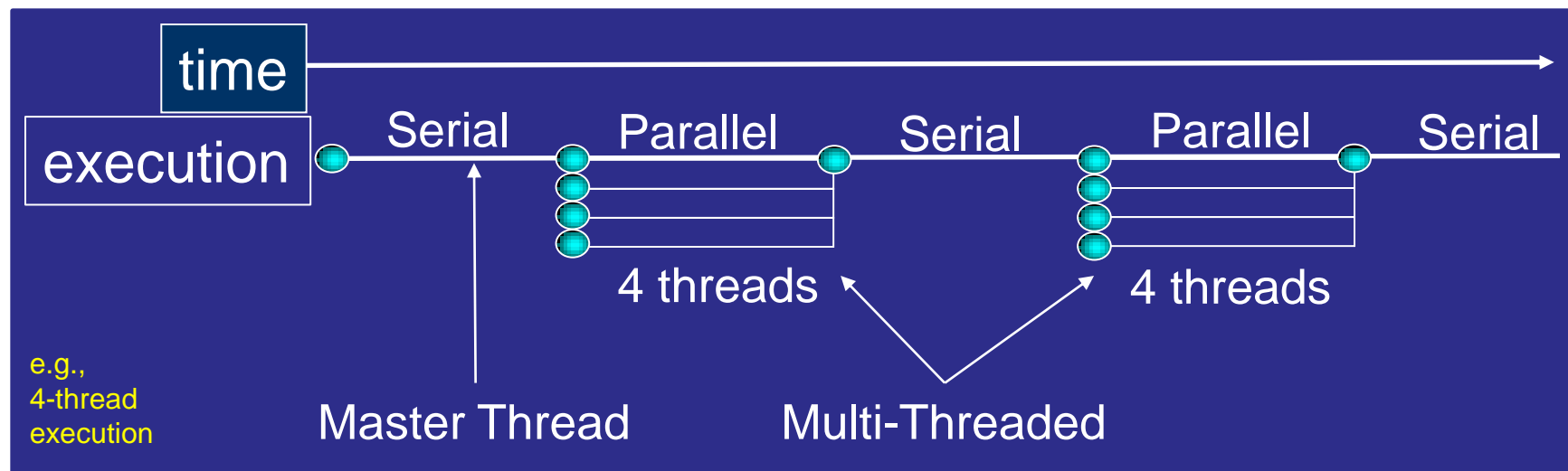
```
// repetitive work
#pragma omp parallel for
for (i=0; i<N; i++)
    a[i] = b[i] + c[i];
```

```
// repetitive updates
#pragma omp parallel for
for (i=0; i<N; i++)
    sum = sum + b[i]*c[i];
```



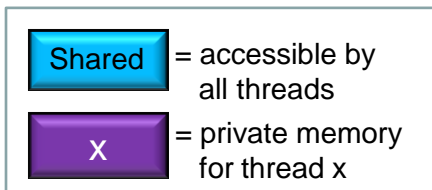
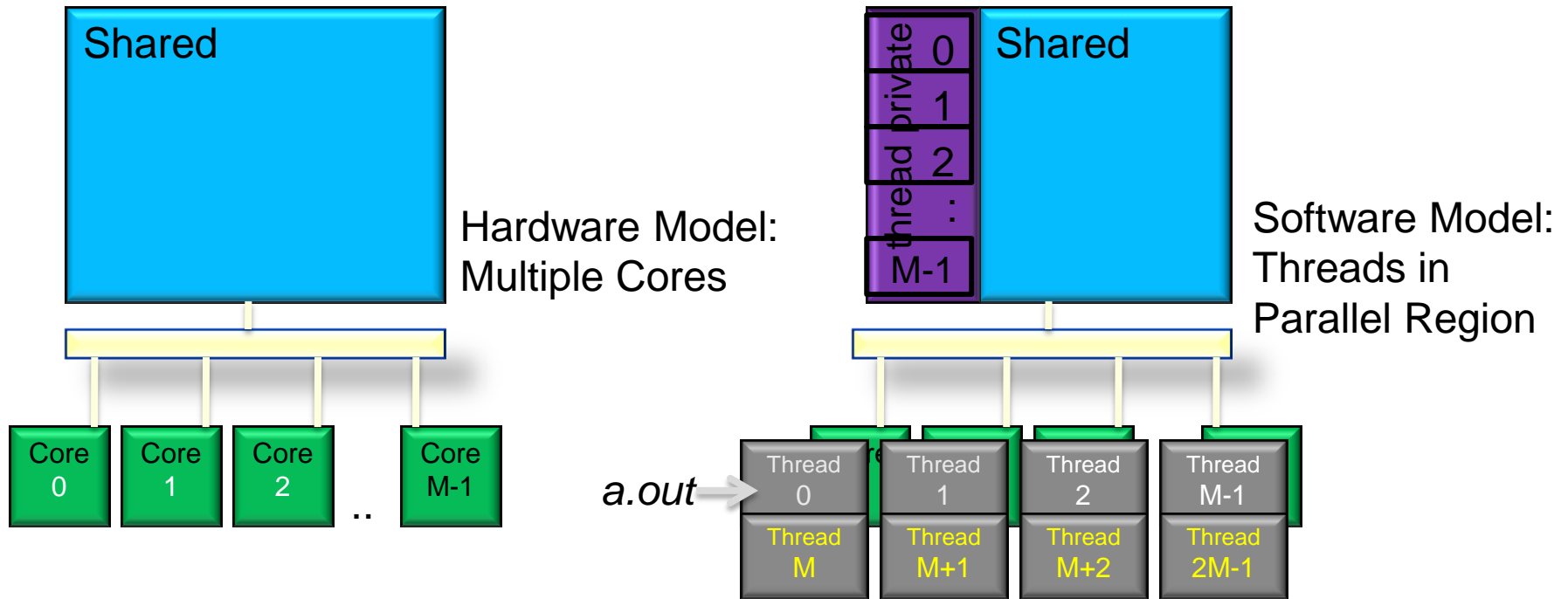
OpenMP Fork-Join Parallelism

- Programs begin as a single process: **master thread**
- Master thread executes until a **parallel region** is encountered
 - Master thread creates (**forks**) a team of parallel threads
 - Threads in **team** simultaneously execute tasks in the parallel region
 - Team threads synchronize and terminate (**join**); master continues





OpenMP on Shared Memory Systems



M threads are usually mapped to M cores.

For HyperThreading, 2 SW threads are mapped to 2 HW threads on each core.

On the Intel Xeon Phi Coprocessors, there are 4 HW threads/core.



Thread Memory Access

- Every thread has access to “global” (shared) memory
 - All threads share the same address space
 - Threads don’t communicate like MPI processes
- But need to avoid **race conditions** with shared memory. Examples:
 1. If multiple writers are going in no particular order, last writer “wins”
 2. A reader may either precede or follow a writer – lack of synchronization
 3. Threads may overlap in a code block, causing conditions 1 *and* 2
- What do you with a race condition?
 - Don’t introduce one in the first place: it’s a bug, hard to debug
 - Impose order with barriers (explicit/implicit synchronization)
- Use **mutual exclusion** (mutex) directives to protect **critical sections**, where one thread must run at a time (at a performance penalty)



Example of a Critical Section

Intended

Thread 0		Thread 1		Value
read	←			0
increment				0
write	→			1
		read	←	1
		increment		1
		write	→	2

Possible...

Thread 0		Thread 1		Value
				0
read	←			0
increment		read	←	0
write	→	increment		1
		write	→	1
				1

- In a critical section, need *mutual exclusion* to get intended result
- The following OpenMP directives prevent this race condition:

`#pragma omp critical` – for a code block (C/C++)

`#pragma omp atomic` – for single statements



OpenMP Directives

- OpenMP directives are comments in source code that specify parallelism for shared-memory parallel (SMP) machines
- FORTRAN compiler directives begin with one of the sentinels **!\$OMP**, **C\$OMP**, or ***\$OMP** – use **!\$OMP** for free-format F90
- C/C++ compiler directives begin with the sentinel **#pragma omp**

Fortran 90

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

!$OMP parallel do
DO ...
!$OMP end parallel do
```

C/C++

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

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



Role of the Compiler

- OpenMP relies on the compiler to do the multithreading
 - Compiler recognizes OpenMP directives, builds in appropriate code
- A special flag is generally required to enable OpenMP
 - GNU: `gcc -fopenmp`
 - Intel: `icc -openmp`
- Additional flags are required to enable MIC instructions
 - Offload marked sections to MIC: `icc -offload-build -openmp`
 - Build whole code native to MIC: `icc -mmic [-openmp]`
 - These options are valid for Intel compilers only
 - Details could change when “Xeon Phi” (the MIC product) is released



OpenMP Syntax

- OpenMP Directives: Sentinel, construct, and clauses

<code>#pragma omp construct [clause [,]clause]...</code>	C
<code>!\$omp construct [clause [,]clause]...</code>	F90

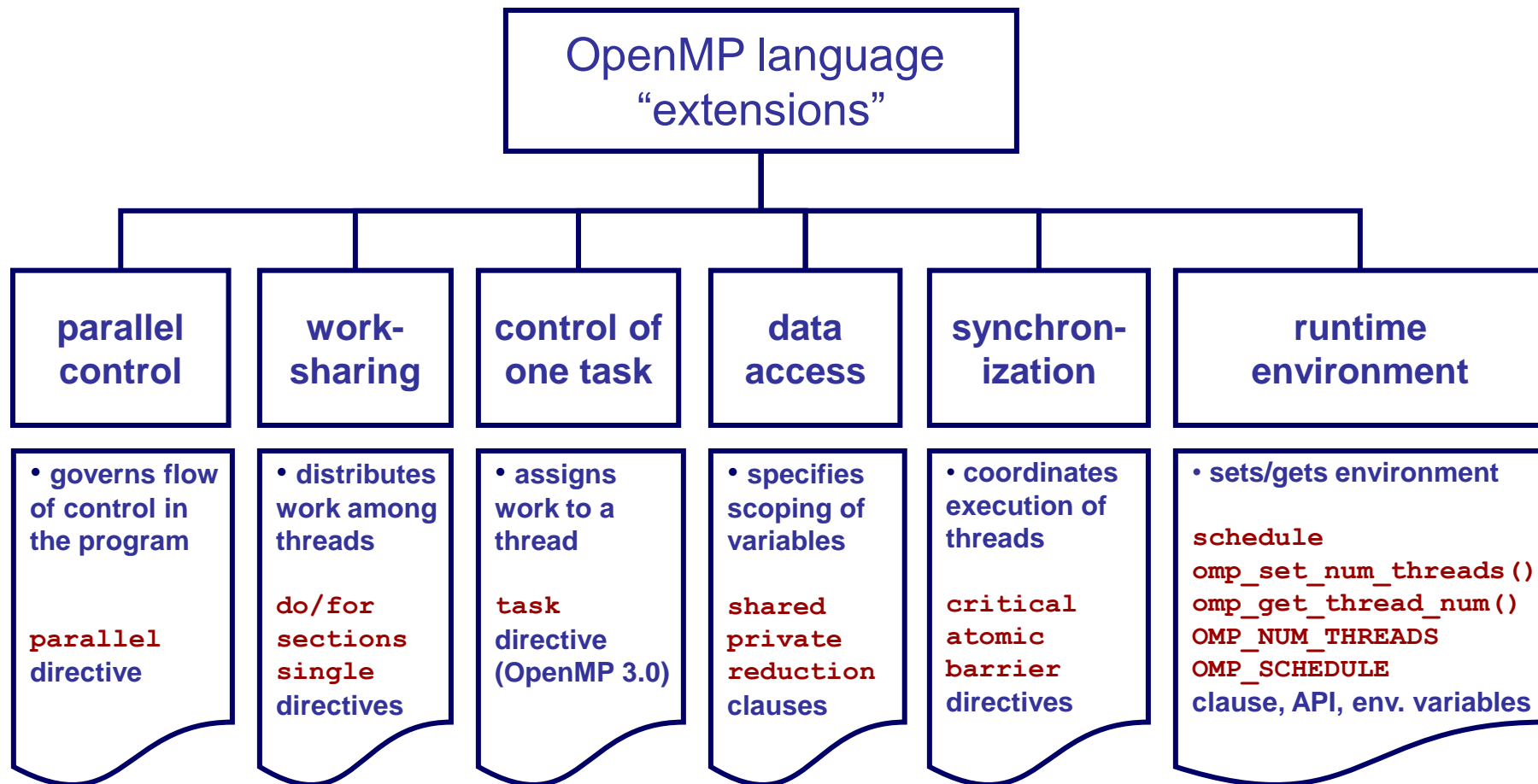
- Example

<code>#pragma omp parallel private(i) reduction(+:sum)</code>	C
<code>!\$omp parallel private(i) reduction(+:sum)</code>	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





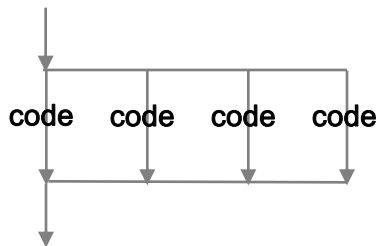
OpenMP Parallel Directives

- Replicated – executed by all threads
- Worksharing – divided among threads

```

PARALLEL
  {code}
END PARALLEL

```

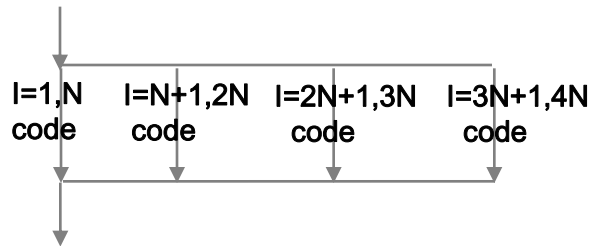


Replicated

```

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

```

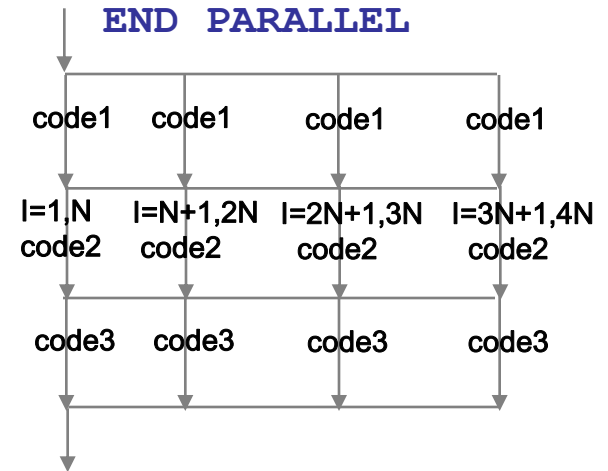


Worksharing

```

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

```



Combined



OpenMP Worksharing, Mutual Exclusion

Use OpenMP directives to specify worksharing in a parallel region, as well as mutual exclusion

```
#pragma omp parallel  
{  
    #pragma omp  
}  
// end parallel
```

<i>Code block</i>	<i>Thread action</i>
for	Worksharing
sections	Worksharing
single	One thread
critical	One thread at a time

parallel do/for
parallel sections

Directives can be combined,
if a parallel region has just
one worksharing construct.



Worksharing Loop: C/C++

General form:

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

```
#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).

Lines 2–6 Loop iterations are split among threads.
 Implied barrier at end of block(s) {}.

Each loop iteration must be independent of other iterations.



Worksharing Loop: Fortran

General form:

```
1  !$omp parallel do
2  do i=1,N
3      a(i) = b(i) + c(i)
4  enddo
5  !$omp end parallel do
6
```

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

Line 1 Team of threads formed (parallel region).

Lines 2–5 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.



OpenMP Clauses

- *Directives* dictate what the OpenMP thread team will do
- Examples:
 - *Parallel regions* are marked by the `parallel` directive
 - *Worksharing loops* are marked by `do`, `for` directives (Fortran, C/C++)
- *Clauses* control the behavior of any particular OpenMP directive
- Examples:
 1. Scoping of variables: `private`, `shared`, `default`
 2. Initialization of variables: `copyin`, `firstprivate`
 3. Scheduling: `static`, `dynamic`, `guided`
 4. Conditional application: `if`
 5. Number of threads in team: `num_threads`



Private, Shared Clauses

- In the following loop, each thread needs a private copy of temp
 - The result would be unpredictable if temp were shared, because each processor would be writing and reading to/from the same location

```
!$omp parallel do private(temp,i) shared(A,B,C)
  do i=1,N
    temp = A(i)/B(i)
    C(i) = temp + cos(temp)
  enddo
!$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)” initializes each thread’s temp to the global value

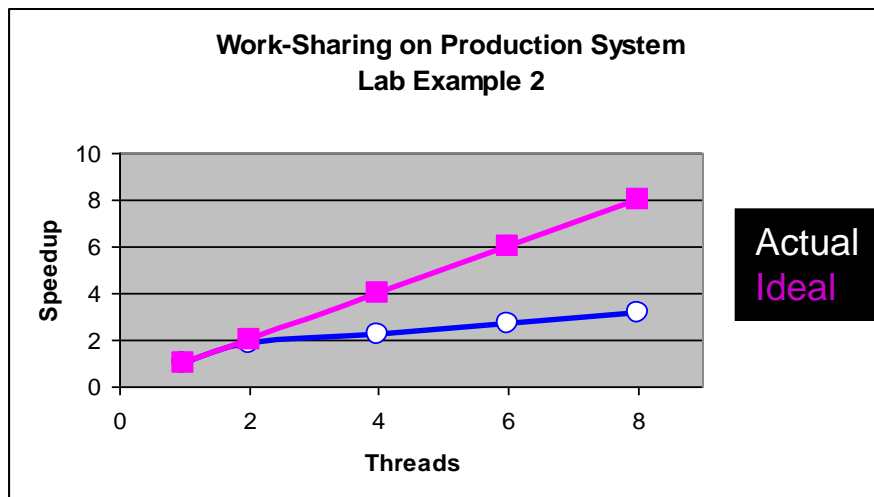
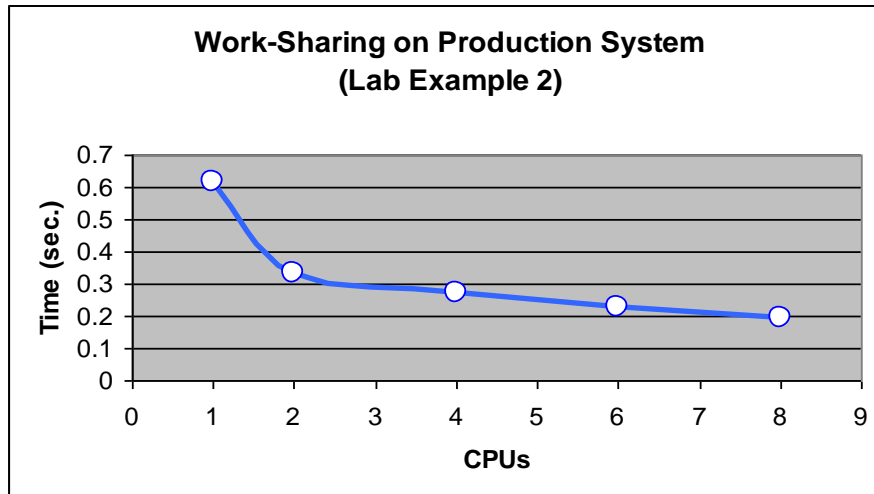


Worksharing Results

$$\text{Speedup} = \text{cputime}(1) / \text{cputime}(N)$$

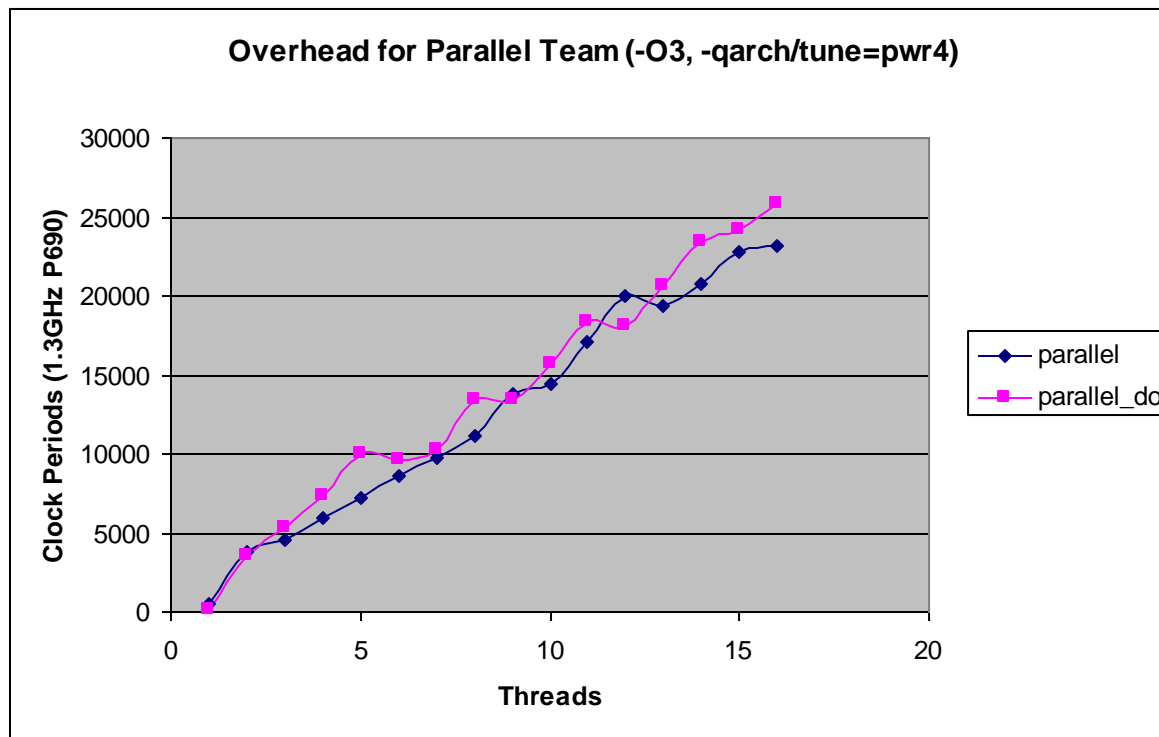
If work is completely parallel, scaling is linear.

Scheduling, memory contention and overhead can impact speedup and Mflop/s rate.





Overhead to Fork a Thread Team



- Increases roughly linearly with number of threads



Merging Parallel Regions

The !\$OMP PARALLEL directive declares an entire region as parallel; therefore, merging work-sharing constructs into a single parallel region eliminates the overhead of separate team formations

```
!$OMP PARALLEL DO
  do i=1,n
    a(i)=b(i)+c(i)
  enddo
!$OMP END PARALLEL DO
!$OMP PARALLEL DO
  do i=1,m
    x(i)=y(i)+z(i)
  enddo
!$OMP END PARALLEL DO
```



```
!$OMP PARALLEL
  !$OMP DO
    do i=1,n
      a(i)=b(i)+c(i)
    enddo
  !$OMP END DO
  !$OMP DO
    do i=1,m
      x(i)=y(i)+z(i)
    enddo
  !$OMP END DO
!$OMP END PARALLEL
```



Runtime Library Functions

<code>omp_get_num_threads()</code>	Number of threads in current team
<code>omp_get_thread_num()</code>	Thread ID, {0: N-1}
<code>omp_get_max_threads()</code>	Number of threads in environment, <code>OMP_NUM_THREADS</code>
<code>omp_get_num_procs()</code>	Number of machine CPUs
<code>omp_in_parallel()</code>	True if in parallel region & multiple threads executing
<code>omp_set_num_threads(#)</code>	Changes number of threads for parallel region, if dynamic threading is enabled



Environment Variables, More Functions

- To control the OpenMP runtime environment

<code>OMP_NUM_THREADS</code>	Set to permitted number of threads: this is the value returned by <code>omp_get_max_threads()</code>
<code>OMP_DYNAMIC</code>	TRUE/FALSE for enable/disable dynamic threading (can also use the function below)

- To enable dynamic thread count (*not* dynamic scheduling!)

<code>omp_set_dynamic()</code>	Set state of dynamic threading: if equal to "true", <code>omp_set_num_threads()</code> controls thread count
<code>omp_get_dynamic()</code>	True if dynamic threading is on



Additional Topics to Explore...

- Schedule clause: specify how to divide work among threads
`schedule (static)` `schedule (dynamic ,M)`
- Reduction clause: perform collective operations on shared variables
`reduction (+:asum)` `reduction (*:aproduct)`
- Nowait clause: remove the barrier at the end of a parallel section
`for ... nowait` `end do nowait`
- Lock routines: make mutual exclusion more lightweight and flexible
`omp_init_lock (var)` `omp_set_lock (var)`



Some Programming Models for Intel MIC

- Intel Threading Building Blocks (TBB)
 - For C++ programmers
- Intel Cilk Plus
 - Task-oriented add-ons for OpenMP
 - Currently for C++ programmers, may become available for Fortran
- Intel Math Kernel Library (MKL)
 - Automatic offloading by compiler for some MKL features
 - MKL is inherently parallelized with OpenMP
- **OpenMP**
 - On Stampede, TACC expects that this will be the most interesting programming model for HPC users



MIC Programming with OpenMP

- In OpenMP 4.0, accelerator syntax may ultimately be standardized, but for now, we use special MIC directives for the Intel compilers
- OpenMP pragma is preceded by MIC-specific **pragma**
 - Fortran: `!dir$ omp offload target(mic) <...>`
 - C: `#pragma offload target(mic) <...>`
- All data transfer is handled by the compiler
 - User control provided through **optional keywords**
- I/O can be done from within offloaded region
 - Data can “stream” through the MIC; no need to leave MIC to fetch new data
 - Also very helpful when debugging (print statements)
- Specific subroutines can be offloaded, including MKL subroutines



Example 1

2-D array (**a**) is filled with data on the coprocessor

Data management done automatically by compiler

- Memory is allocated on coprocessor for (**a**)
- Private variables (**i**, **j**, **x**) are created
- Result is copied back

```
use omp_lib                ! OpenMP
integer                    :: n = 1024        ! Size
real, dimension(:, :), allocatable :: a     ! Array
integer                    :: i, j          ! Index
real                      :: x             ! Scalar
allocate(a(n,n))           ! Allocation
!dir$ omp offload target(mic)                ! Offloading
!$omp parallel do shared(a,n), &           ! Par. region
  private(x, i, j), schedule(dynamic)
do j=1, n
  do i=j, n
    x = real(i + j); a(i,j) = x
```

```
#include <omp.h>           /* C example */
const int n = 1024;      /* Size of the array */
int      i, j;           /* Index variables */
float a[n][n], x
#pragma offload target(mic)
#pragma omp parallel for shared(a), \
  private(x), schedule(dynamic)
for(i=0; i<n; i++) {
  for(j=i; j<n; j++) {
    x = (float)(i + j); a[i][j] = x; }}
```



Example 2

I/O from offloaded region:

- File is opened and closed by one thread (**omp single**)
- All threads take turns reading from the file (**omp critical**)

Threads may also read in parallel (not shown)

- Parallel file system
- Threads read parts from different targets

```
#pragma offload target(mic) //Offload region
#pragma omp parallel
{
    #pragma omp single /* Open File */
    {
        printf("Opening file in offload region\n");
        f1 = fopen("/var/tmp/mydata/list.dat","r");
    }

    #pragma omp for
    for(i=1;i<n;i++) {
        #pragma omp critical
        { fscanf(f1,"%f",&a[i]); }
        a[i] = sqrt(a[i]);
    }

    #pragma omp single
    {
        printf("Closing file in offload region\n");
        fclose (f1);
    }
}
```



Example 3

Two routines, MKL's `sgemm` and `my_sgemm`

- Both are called with `offload` directive
- `my_sgemm` specifies explicit `in` and `out` data movement

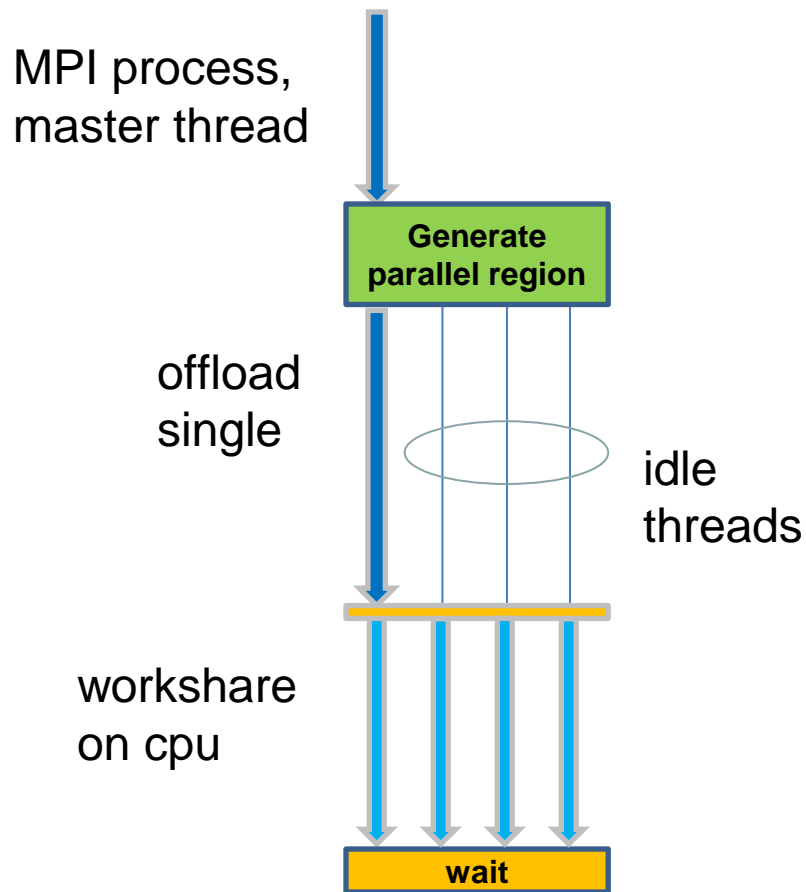
Use `attributes` to have routine compiled for the coprocessor, or link coprocessor-based MKL

```
! snippet from the caller...
! offload MKL routine to accelerator
!dir$ attributes offload:mic :: sgemm
!dir$ offload target(mic)
call & sgemm('N','N',n,n,n,alpha,a,n,b,n,beta,c,n)
! offload hand-coded routine with data clauses
!dir$ offload target(mic) in(a,b) out(d)
call my_sgemm(d,a,b)
```

```
! snippet from the hand-coded subprogram...
!dir$ attributes offload:mic :: my_sgemm
subroutine my_sgemm(d,a,b)
real, dimension(:,:) :: a, b, d
!$omp parallel do
do j=1, n
  do i=1, n
    d(i,j) = 0.0
    do k=1, n
      d(i,j) = d(i,j)+a(i,k)*b(k,j)
    enddo; enddo; endo
end subroutine
```



Heterogeneous Threading, Sequential



```
#pragma omp parallel           C/C++
{
  #pragma omp single
  { offload(); }

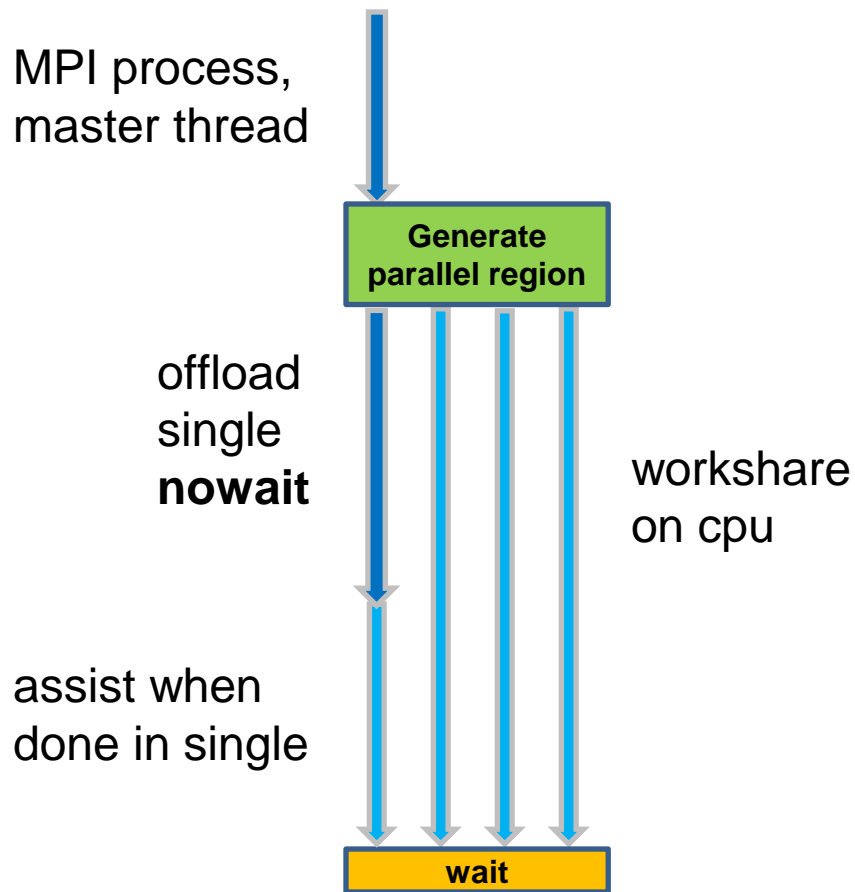
  #pragma omp for
  for(i=0; i<N; i++){...}
}
```

```
!$omp parallel                 F90
  !$omp single
  call offload();
  !$omp end single

  !$omp do
  do i=1,N; ...
  end do
!$omp end parallel
```



Heterogeneous Threading, Concurrent



```
#pragma omp parallel           C/C++
{
  #pragma omp single nowait
  { offload(); }

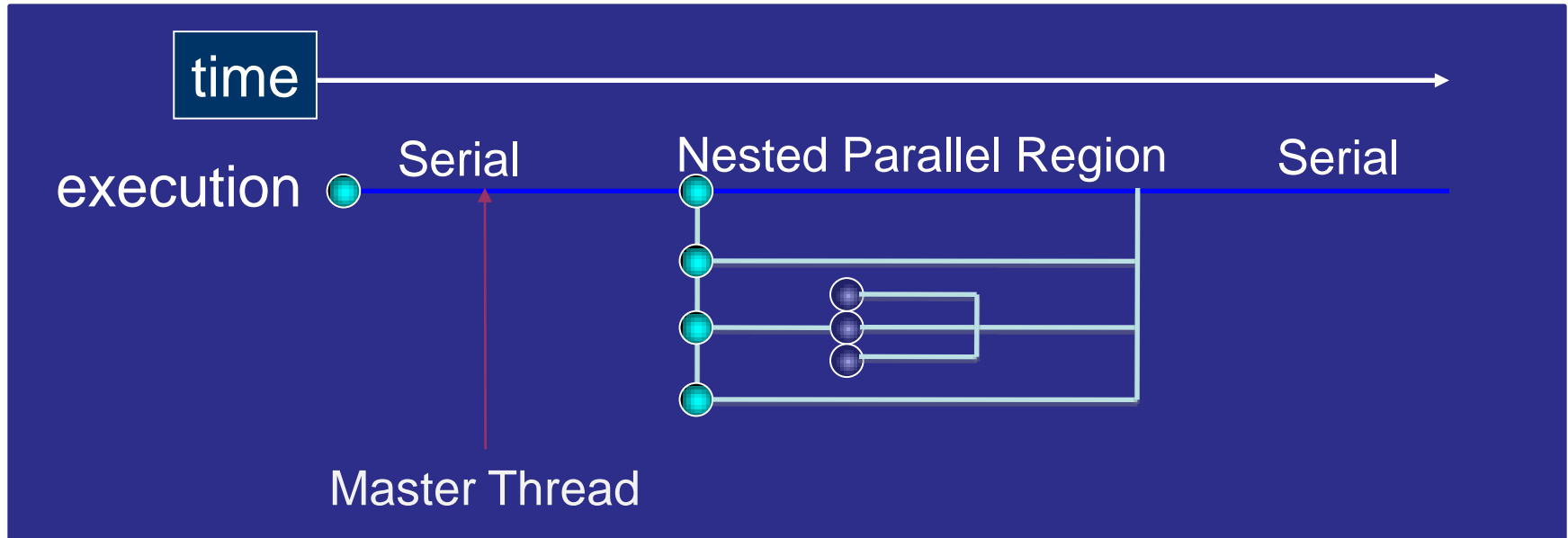
  #pragma omp for schedule(dynamic)
  for(i=0; i<N; i++){...}
}
```

```
!$omp parallel                 F90
  !$omp single
  call offload();
  !$omp end single nowait

  !$omp do schedule(dynamic)
  do i=1,N; ...
  end do
!$omp end parallel
```



Loop Nesting



- OpenMP 3.0 supports nested parallelism, older implementations may ignore the nesting and serialize inner parallel regions.
- A nested parallel region can specify any number of threads to be used for the thread team, new id's are assigned.