



Cornell University  
Center for Advanced Computing

# Programming OpenMP

Susan Mehringer  
Cornell Center for Advanced Computing

Based on materials developed CAC and TACC



## Overview

---

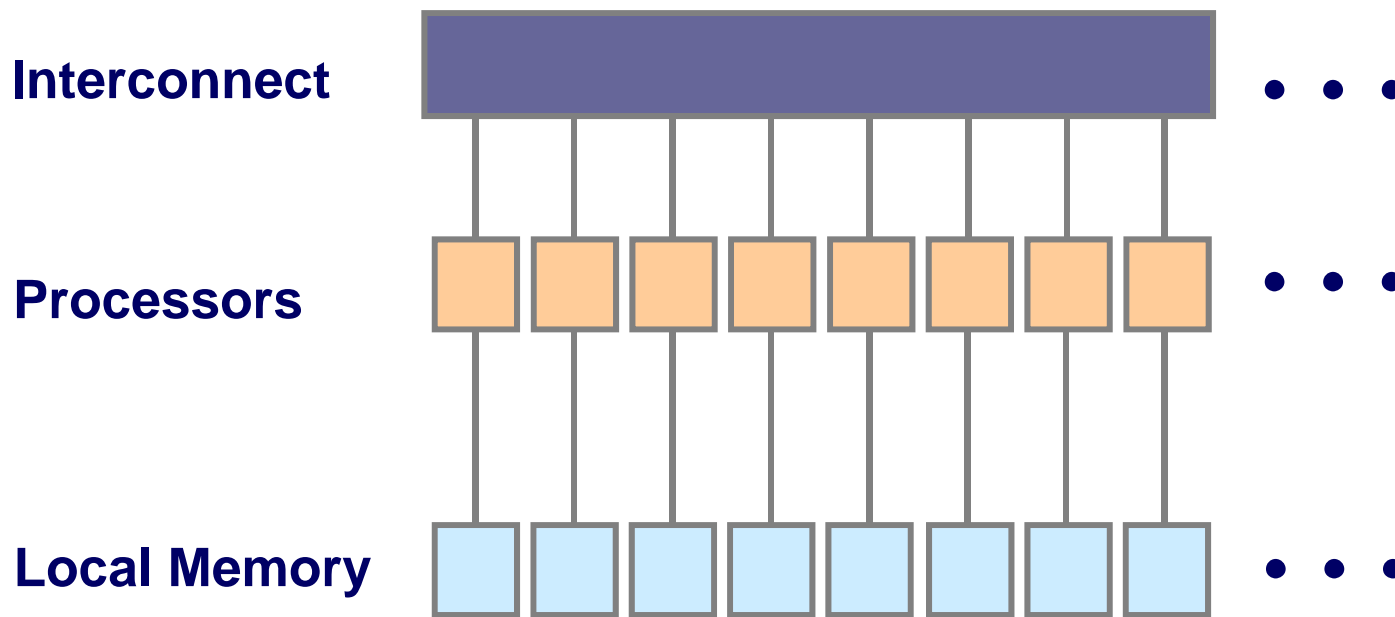
- Parallel processing
  - MPP vs. SMP platforms
  - Motivations for parallelization
- What is OpenMP?
- How does OpenMP work?
  - Architecture
  - Fork-join model of parallelism
  - Communication
- OpenMP constructs
  - Directives
  - Runtime Library API
  - Environment variables

MPP = Massively Parallel Processing  
SMP = Symmetric MultiProcessing



## MPP platforms

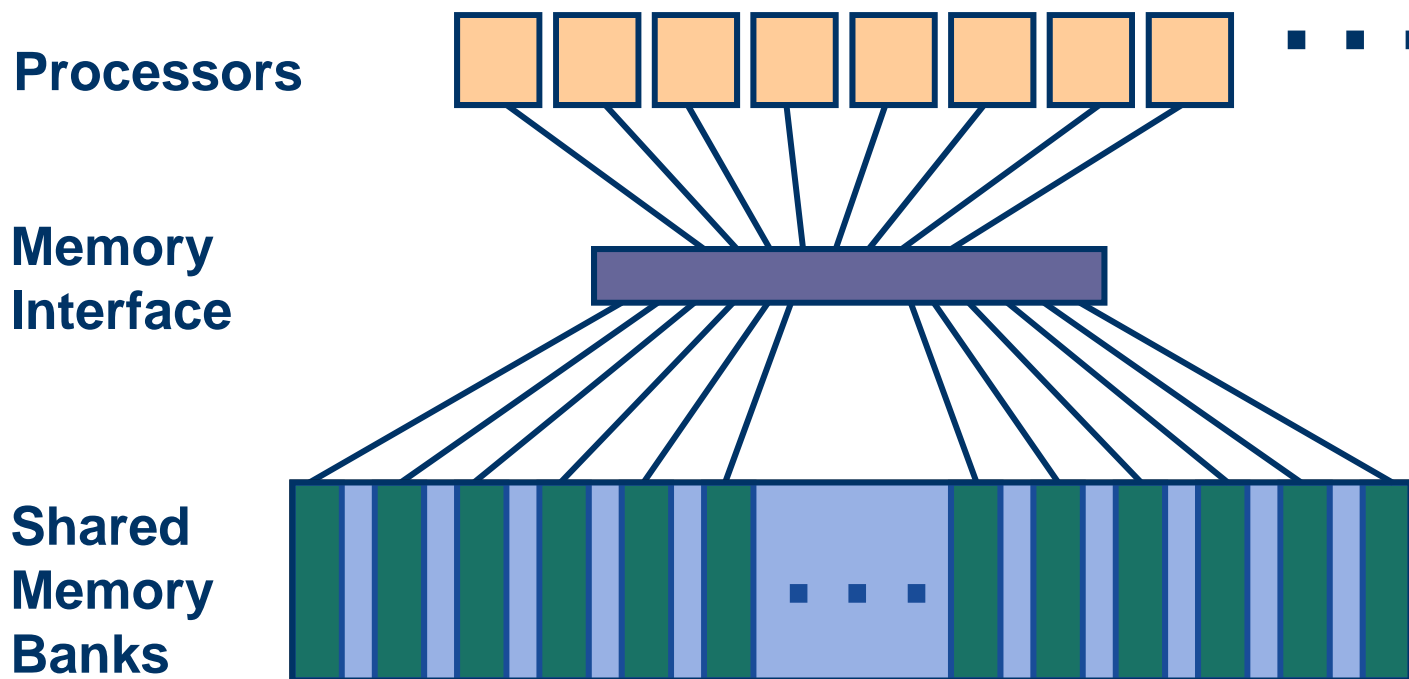
- Clusters are distributed memory platforms in which each processor has its own local memory; **use MPI on these systems.**





## SMP platforms

- In each Ranger node, the 16 cores share access to a common pool of memory; likewise for the 8 cores in each node of CAC's v4 cluster





## What is OpenMP?

---

- De facto open standard for scientific parallel programming on Symmetric MultiProcessor (SMP) systems
  - Allows fine-grained (e.g., loop-level) and coarse-grained parallelization
  - Can express both data and task parallelism
- Implemented by:
  - **Compiler directives**
  - Runtime library (an API, Application Program Interface)
  - Environment variables
- Standard specifies Fortran and C/C++ directives and API
- Runs on many different SMP platforms
- Find tutorials and description at <http://www.openmp.org/>



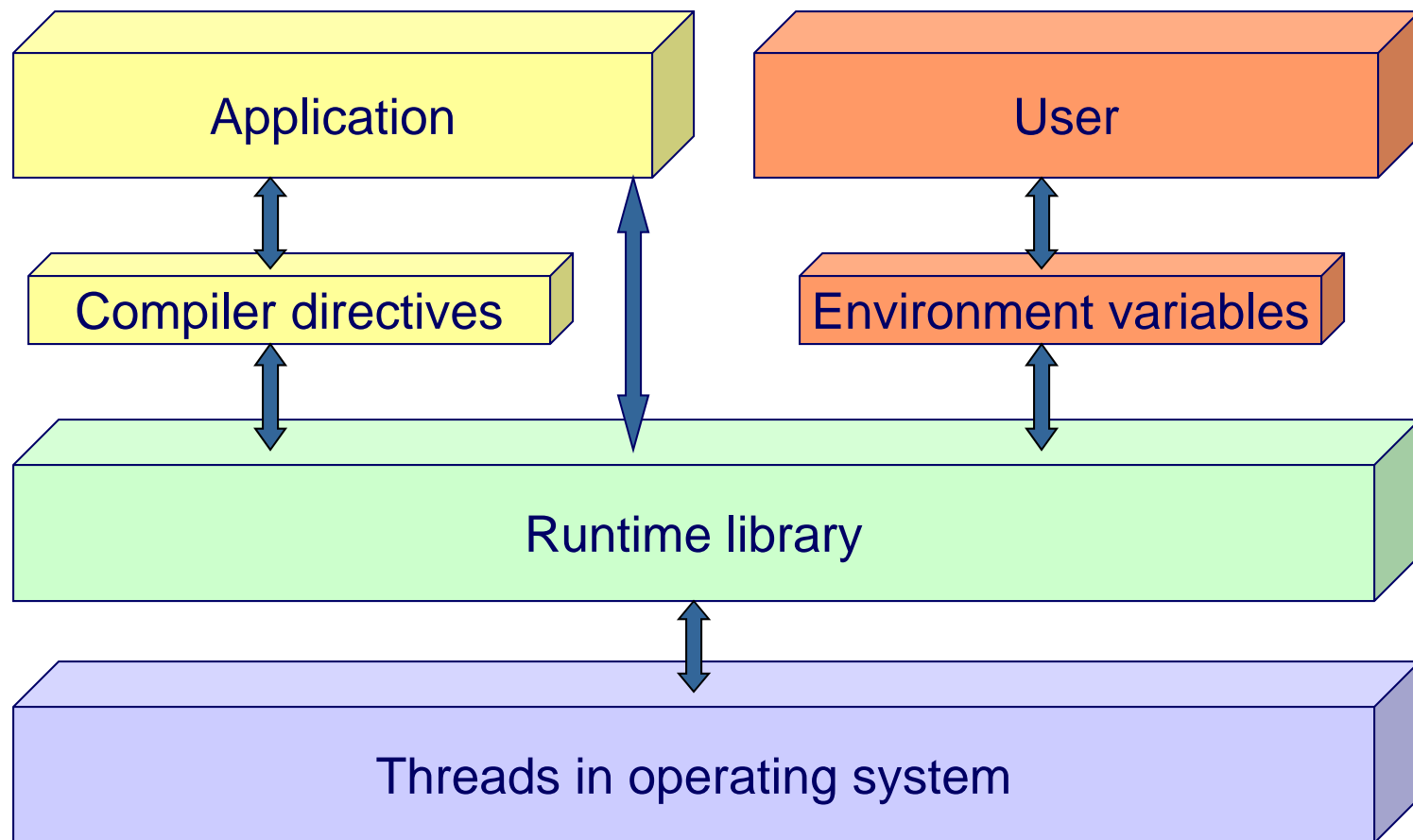
## Advantages/disadvantages of OpenMP

---

- Pros
  - Shared Memory Parallelism is easier to learn
  - Parallelization can be incremental
  - Coarse-grained or fine-grained parallelism
  - Widely available, portable
- Cons
  - Scalability limited by memory architecture
  - Available on SMP systems only
- Benefits
  - *Helps prevent CPUs from going idle on multi-core machines*
  - *Enables faster processing of large-memory jobs*



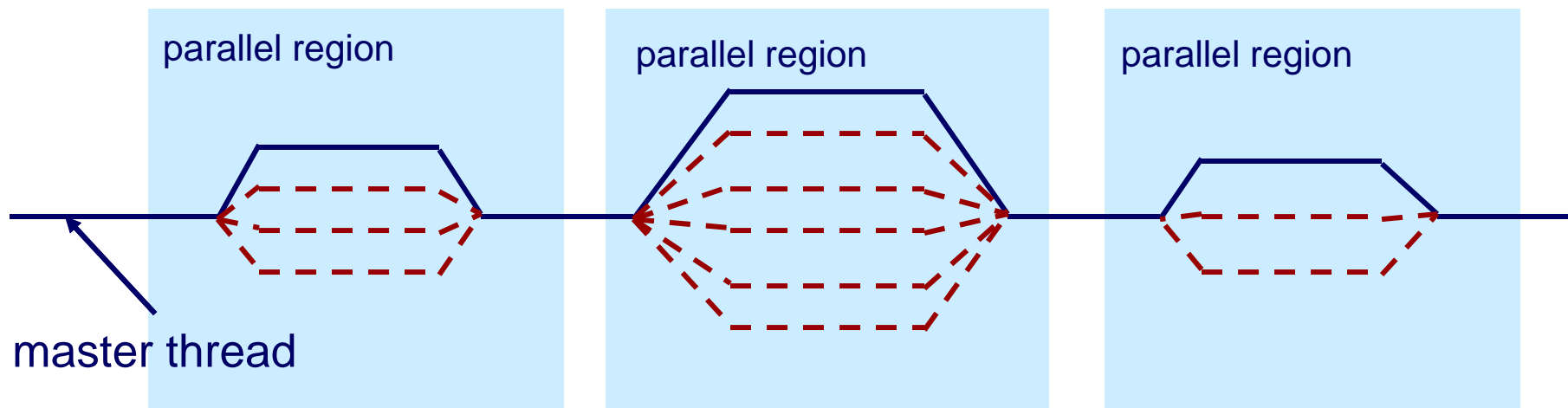
## OpenMP architecture





## OpenMP fork-join parallelism

- Parallel regions are basic “blocks” within code
- A master thread is instantiated at run time and persists throughout execution
- The master thread assembles teams of threads at parallel regions







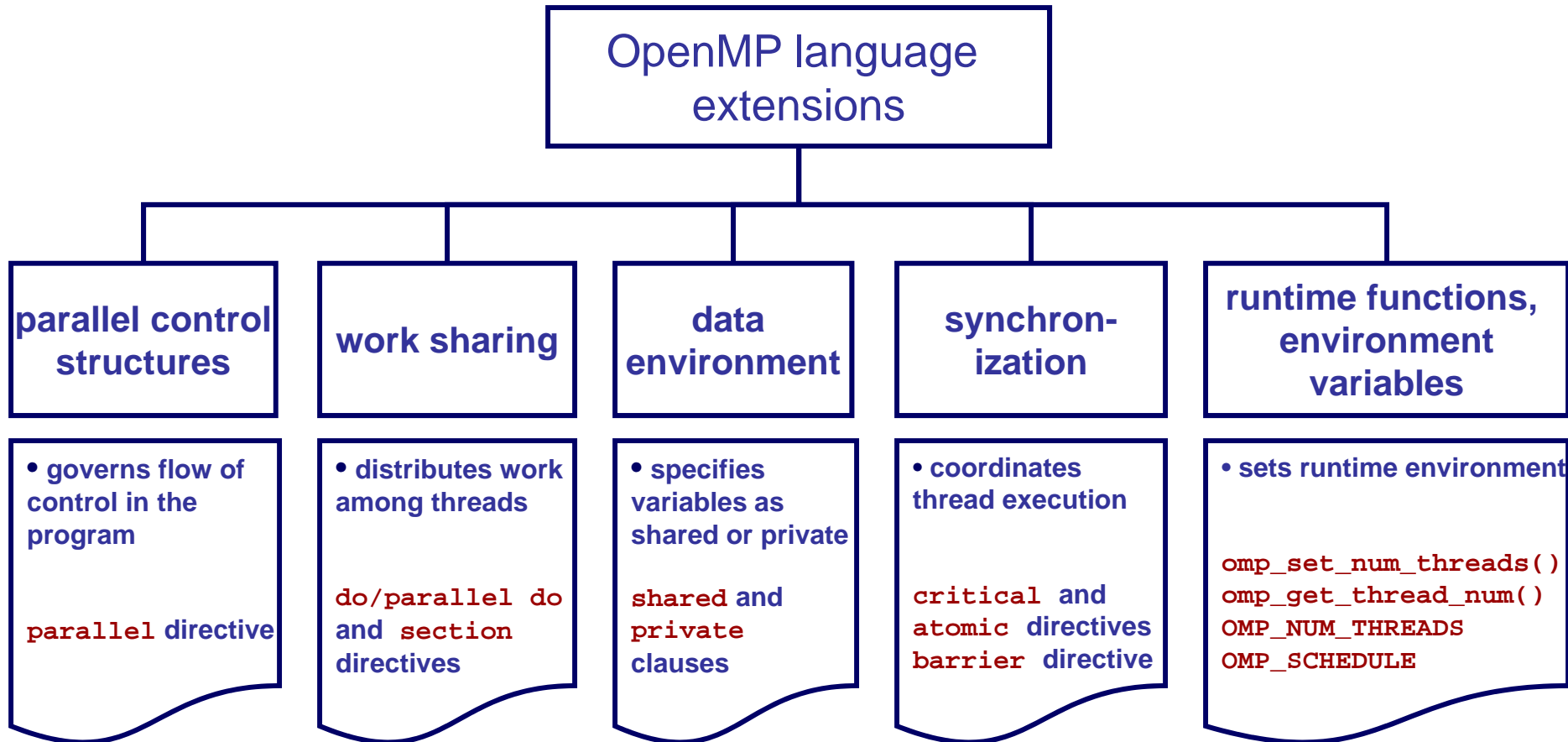
## How do threads communicate?

---

- Every thread has access to “global” memory (shared) and its own stack memory (private)
- Use shared memory to communicate between threads
- Simultaneous updates to shared memory can create a *race condition*: the results change with different thread scheduling
- Use mutual exclusion to avoid race conditions
  - But understand that “mutex” serializes performance wherever it is used
  - By definition only one thread at a time can execute that section of code



# OpenMP constructs





## OpenMP directives

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

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

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

### C/C++

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

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



## Directives and clauses

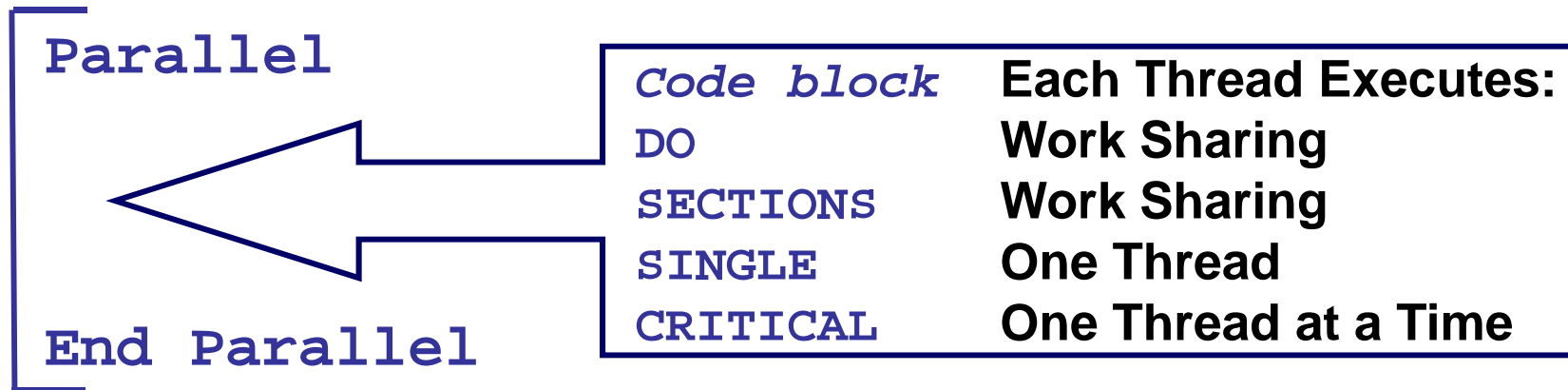
---

- *Parallel regions* are marked by the `parallel` directive
- *Work-sharing loops* are marked by
  - `parallel do` directive in Fortran
  - `parallel for` directive in C
- Clauses control the behavior of a particular 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)



## Parallel region and work sharing

Use OpenMP directives to specify Parallel Region and Work Sharing constructs



**Parallel DO/for**  
**Parallel SECTIONS**

**Stand-alone  
parallel constructs**



## Parallel regions

---

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

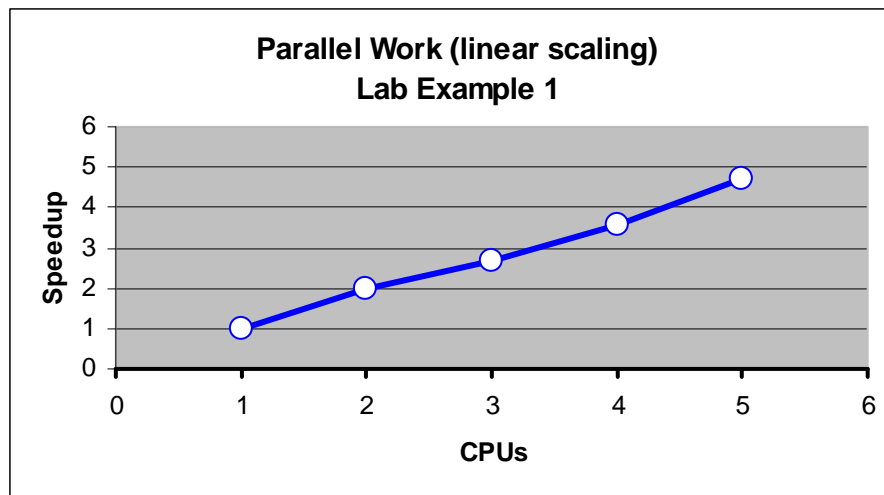
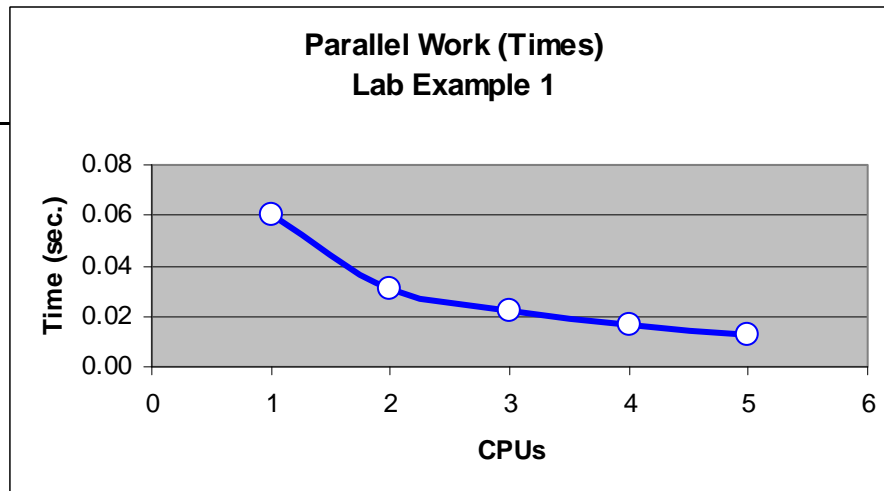
- Line 1** Team of threads is formed at parallel region
- Lines 2-3** Each thread executes code block and subroutine call, no branching into or out of a parallel region
- Line 4** All threads synchronize at end of parallel region (implied barrier)



## Parallel work example

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

If work is completely parallel, scaling is linear





## Work sharing

---

```
1 !$OMP PARALLEL DO
2     do i=1,N
3         a(i) = b(i) + c(i)    !not much work
4     enddo
5 !$OMP END PARALLEL DO
```

- Line 1** Team of threads is formed at parallel region
- Lines 2-4** Loop iterations are split among threads, each loop iteration must be independent of other iterations
- Line 5** (Optional) end of parallel loop (implied barrier at enddo)

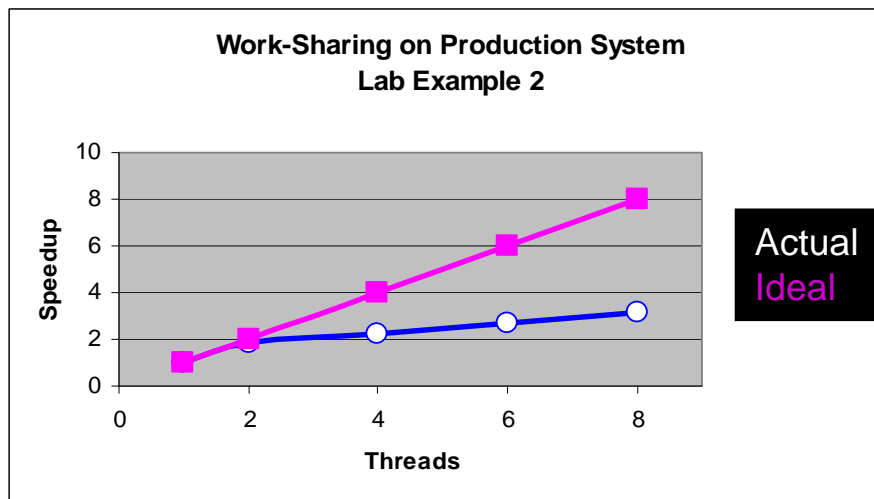
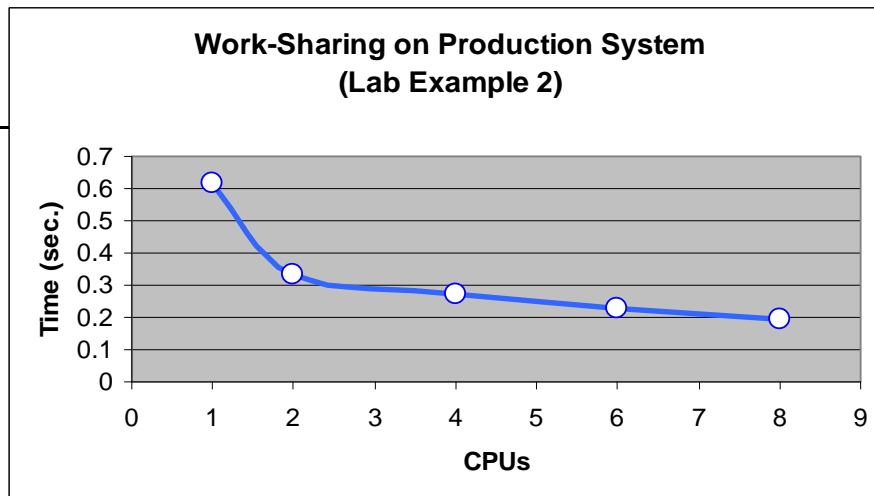




## Work-sharing example

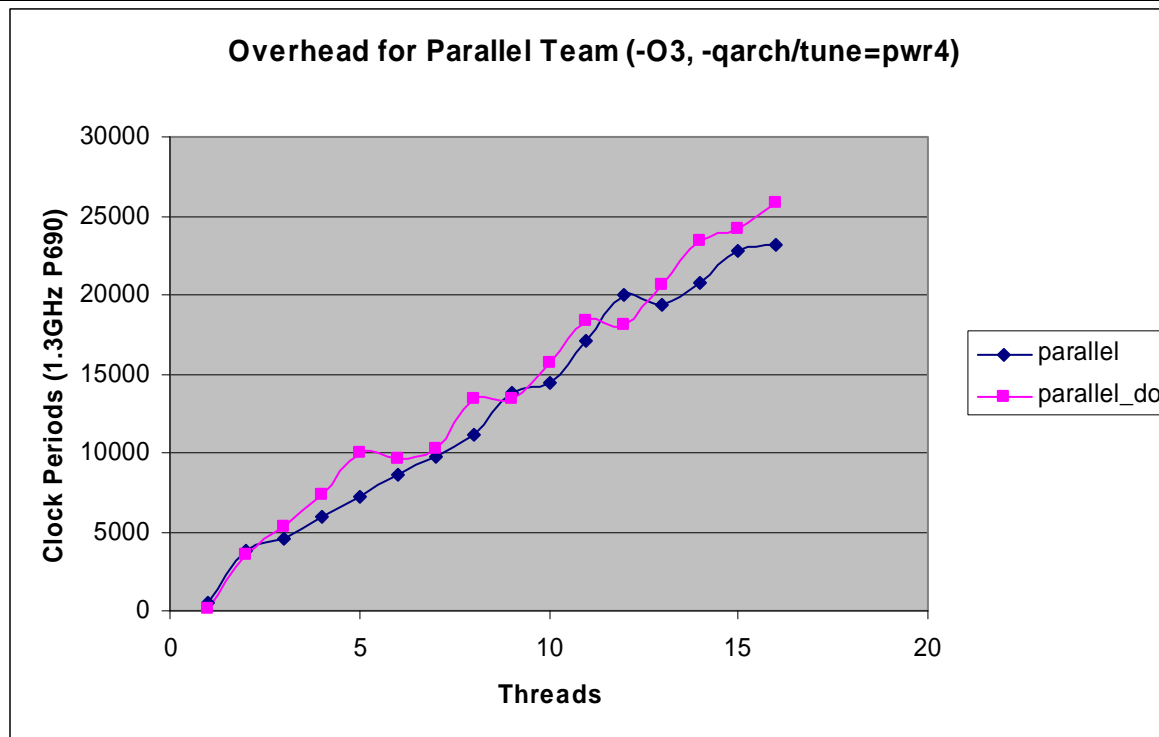
$$\text{Speedup} = \text{cputime}(1) / \text{cputime}(N)$$

Scheduling, memory contention and overhead can impact speedup





## Team overhead



- Increases roughly linearly with number of threads

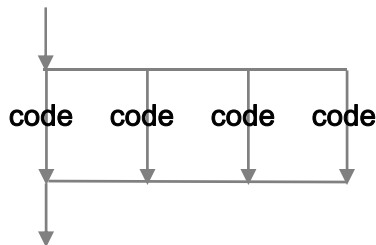


# OpenMP parallel constructs

- **Replicated** : executed by all threads
- **Work sharing** : divided among threads

```

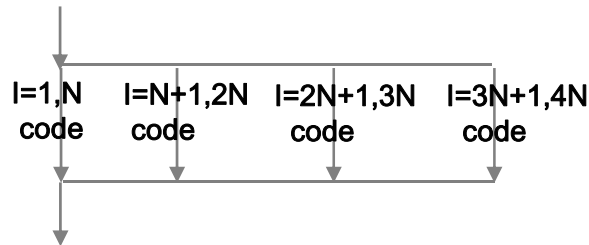
PARALLEL
  {code}
END PARALLEL
  
```



Replicated

```

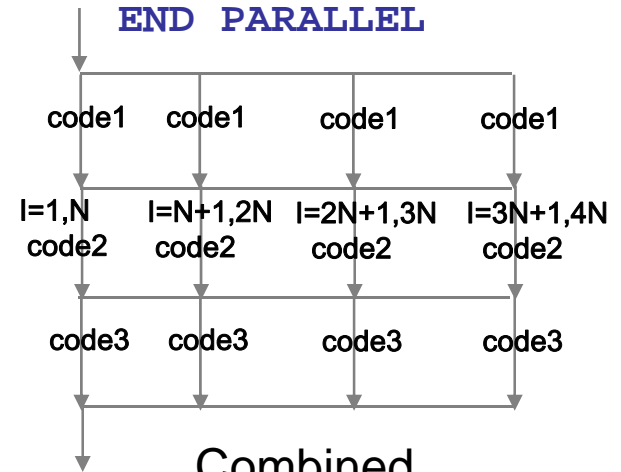
PARALLEL DO
  do I = 1, N*4
    {code}
  end do
END PARALLEL DO
  
```



Work sharing

```

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



Combined




## 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

```
!$OMP PARALLEL
  !$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
```

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





## Distribution of work: SCHEDULE clause

---

- `!$OMP PARALLEL DO SCHEDULE(STATIC)`
  - Default schedule: each CPU receives one set of contiguous iterations
  - Size of set is  $\sim$  (total\_no\_iterations / no\_of\_cpus)
- `!$OMP PARALLEL DO SCHEDULE(STATIC,N)`
  - Iterations are divided round-robin fashion in chunks of size N
- `!$OMP PARALLEL DO SCHEDULE(DYNAMIC,N)`
  - Iterations handed out in chunks of size N as threads become available
- `!$OMP PARALLEL DO SCHEDULE(GUIDED,N)`
  - Iterations handed out in pieces of exponentially decreasing size
  - N = minimum number of iterations to dispatch each time (default is 1)
  - Can be useful for load balancing (“fill in the cracks”)



## OpenMP data scoping

---

- Data-scoping clauses control how variables are shared within a parallel construct
- These include the **shared**, **private**, **firstprivate**, **lastprivate**, **reduction** clauses
- Default variable scope:
  - Variables are shared by default
  - Global variables are shared by default
  - Automatic variables within a subroutine that is called from inside a parallel region are private (reside on a stack private to each thread), unless scoped otherwise
  - Default scoping rule can be changed with **default** clause



## PRIVATE and SHARED data

---

- **SHARED** - Variable is shared (seen) by all processors
- **PRIVATE** - Each thread has a private instance (copy) of the variable
- Defaults: loop indices are private, other variables are shared

```
!$OMP PARALLEL DO
  do i=1,N
    A(i) = B(i) + C(i)
  enddo
!$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 data example

---

- 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

```
!$OMP PARALLEL DO SHARED(A,B,C,N) PRIVATE(temp,i)
  do i=1,N
    temp = A(i)/B(i)
    C(i) = temp + cos(temp)
  enddo
!$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)” initializes each thread’s temp to the global value





## REDUCTION

---

- An operation that “combines” multiple elements to form a single result, such as a summation, is called a reduction operation

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

- Each thread has a private ASUM and APROD (declared as real\*8, e.g.), 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



## 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,M)
    do i=1,m
        x(i)=y(i)+z(i)
    enddo
!$OMP END
!$OMP END PARALLEL
```



## Mutual exclusion: atomic and critical directives

---

- When threads must execute a section of code serially (only one thread at a time can execute it), the region must be marked with **CRITICAL / END CRITICAL** directives
- Use the “**!\$OMP ATOMIC**” directive if executing only one operation

```
!$OMP PARALLEL SHARED(sum,X,Y)
...
!$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
```



## Mutual exclusion: lock routines

---

- When each thread must execute a section of code serially (only one thread at a time can execute it), 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)
```



## Overhead associated with mutual exclusion

---

All measurements were made in dedicated mode

<b>Open MP exclusion routine/directive</b>	<b>cycles</b>
OMP_SET_LOCK/OMP_UNSET_LOCK	330
OMP_ATOMIC	480
OMP_CRITICAL	510



## 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_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



## More functions and variables

---

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

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

- To control the OpenMP runtime environment

<code>OMP_NUM_THREADS</code>	Set to permitted number of threads
<code>OMP_DYNAMIC</code>	TRUE/FALSE for enable/disable dynamic threading



## OpenMP 2.0/2.5: what's new?

---

- Wallclock timers
- Workshare directive (Fortran 90/95)
- Reduction on array variables
- NUM\_THREAD clause





## 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);
```



## References

---

- Current standard
  - <http://www.openmp.org/>
- Books
  - *Parallel Programming in OpenMP*, by Chandra, Dagum, Kohr, Maydan, McDonald, Menon
  - *Using OpenMP*, by Chapman, Jost, Van der Pas (OpenMP 2.5)
- Virtual Workshop Module
  - <https://www.cac.cornell.edu/Ranger/OpenMP/>