Introduction to Programming with OpenMP

Jay Boisseau (Lars Koesterke)

July 15, 2008

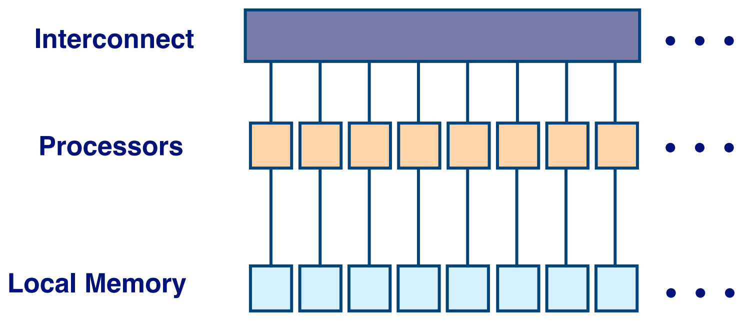
THE UNIVERSITY OF TEXAS AT AUSTIN
Texas Advanced Computing Center

Overview

- Parallel processing
  - Review: distributed vs. shared memory 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
- What's new? OpenMP 2.0/2.5

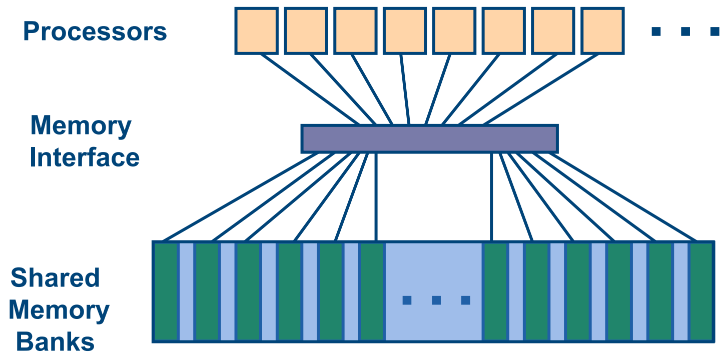
Distributed Memory Platforms

Clusters are Distributed Memory platforms.
Each processor/node has its own memory. Use MPI across these systems.



Shared Memory Platforms

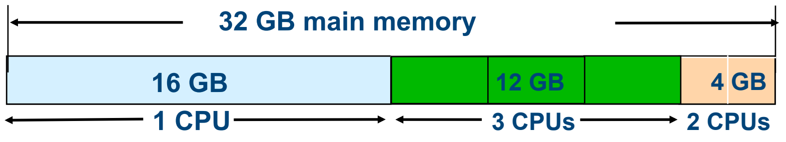
The Lonestar/Ranger nodes are shared-memory platforms.
Each processor has equal access to a common pool of shared memory.
Lonestar and Ranger have 4 and 16 cores per node, respectively.



Motivation for Parallel Processing

- Shorten Execution Wall-Clock Time.
- Access Larger Share of Memory, with minimal impact on other users.

A single-processor, large-memory job will crowd out smaller jobs. Example:
On a 16 processor system, the following memory map indicates 10 CPUs are idle!



Run large-memory jobs on multiple CPUs to maximize CPU usage and reduce everyone's turnaround time.
Fair Share of Memory   Total Size of Memory/#CPUs_you_use

What is OpenMP?

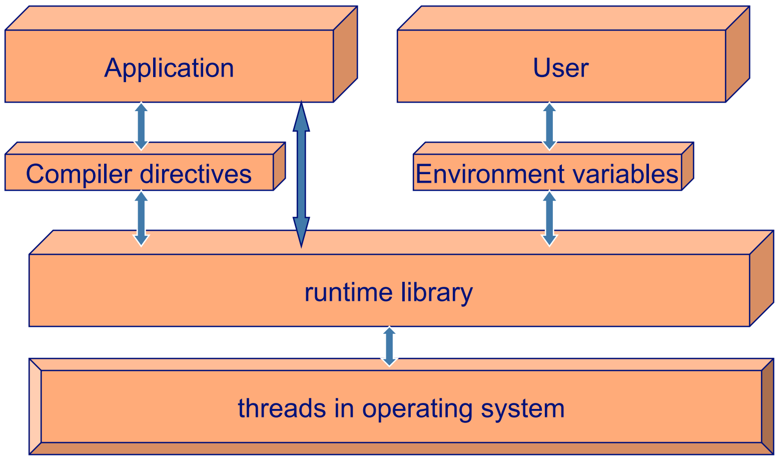
- **De facto open standard for Scientific Parallel Programming on Symmetric MultiProcessor (SMP) Systems.**
- **Implemented by:**
  - **Compiler Directives**
  - **Runtime Library (an API, Application Program Interface)**
  - **Environment Variables**
- **http://www.openmp.org/   has tutorials and description.**
- **Runs on many different SMP platforms.**
- **Standard specifies Fortran and C/C++ Directives & API.**
  Not all vendors have developed C/C++ OpenMP yet.
- **Allows both fine-grained (e.g. loop-level) and coarse-grained parallelization.**

Advantages/Disadvantages of OpenMP

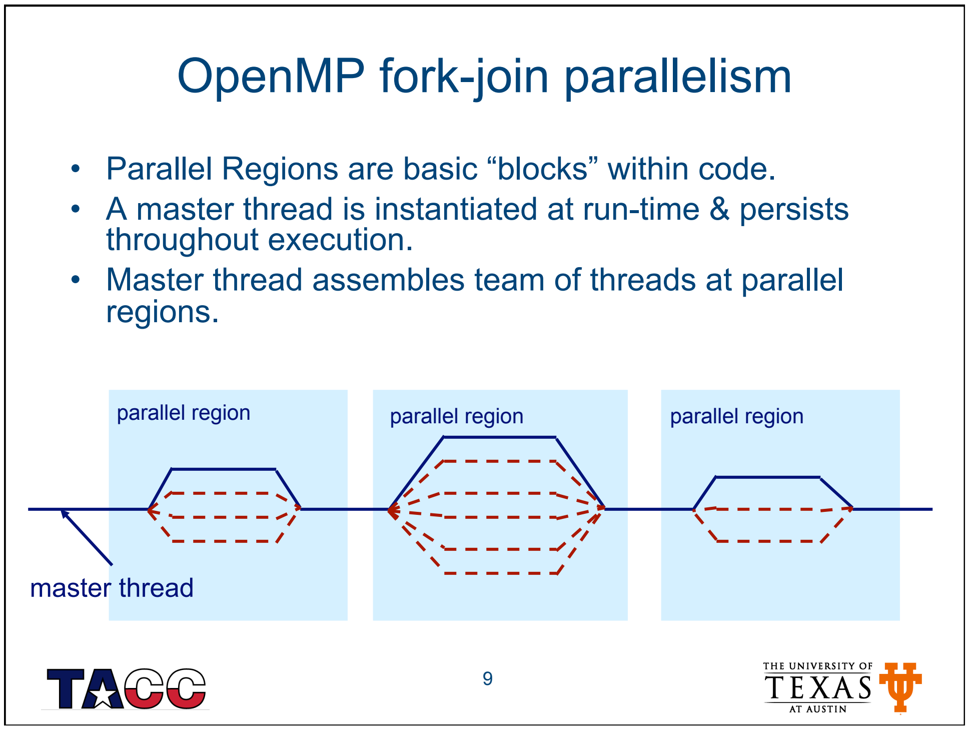
- Pros
  - Shared Memory Parallelism is easier to learn.
  - Parallelization can be incremental
  - Coarse-grained or fine-grained parallelism
  - Widely available, portable
- Cons
  - Scalability limited by memory architecture
  - Available on SMP systems only

TACC 7 THE UNIVERSITY OF TEXAS AT AUSTIN

OpenMP Architecture

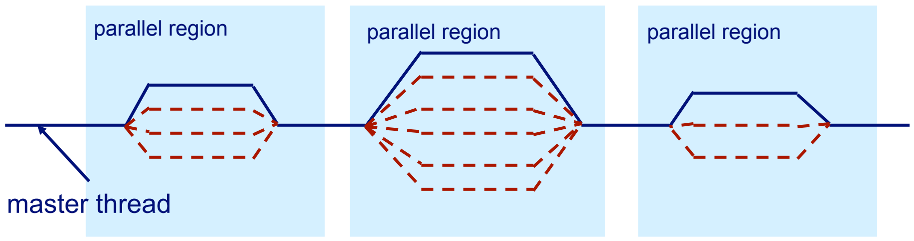


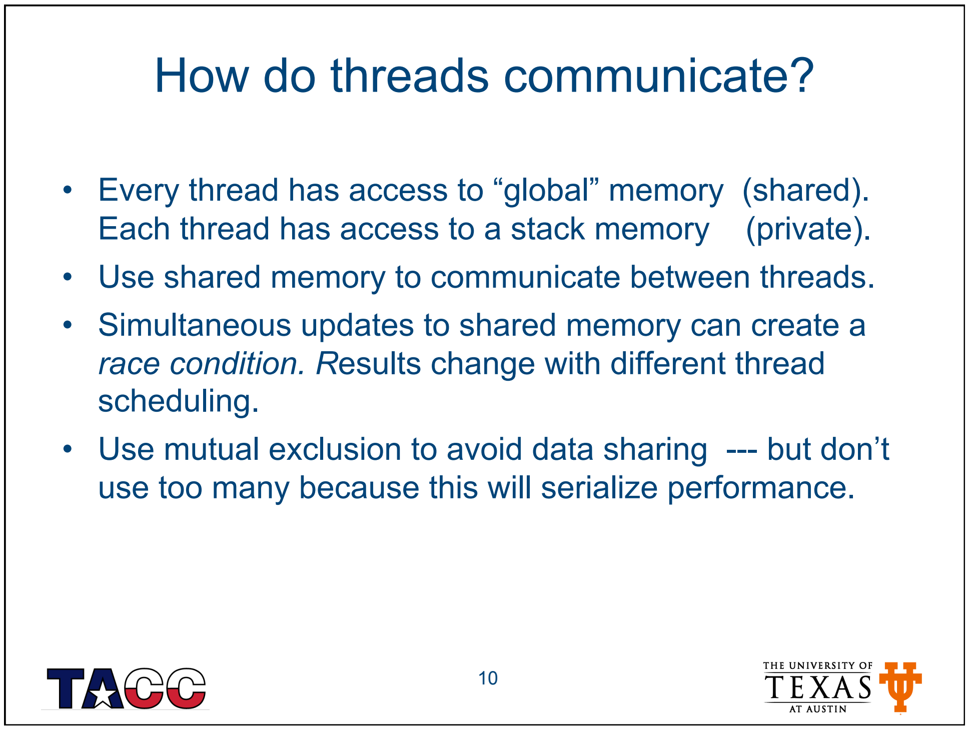
TACC 8 THE UNIVERSITY OF TEXAS AT AUSTIN



OpenMP fork-join parallelism

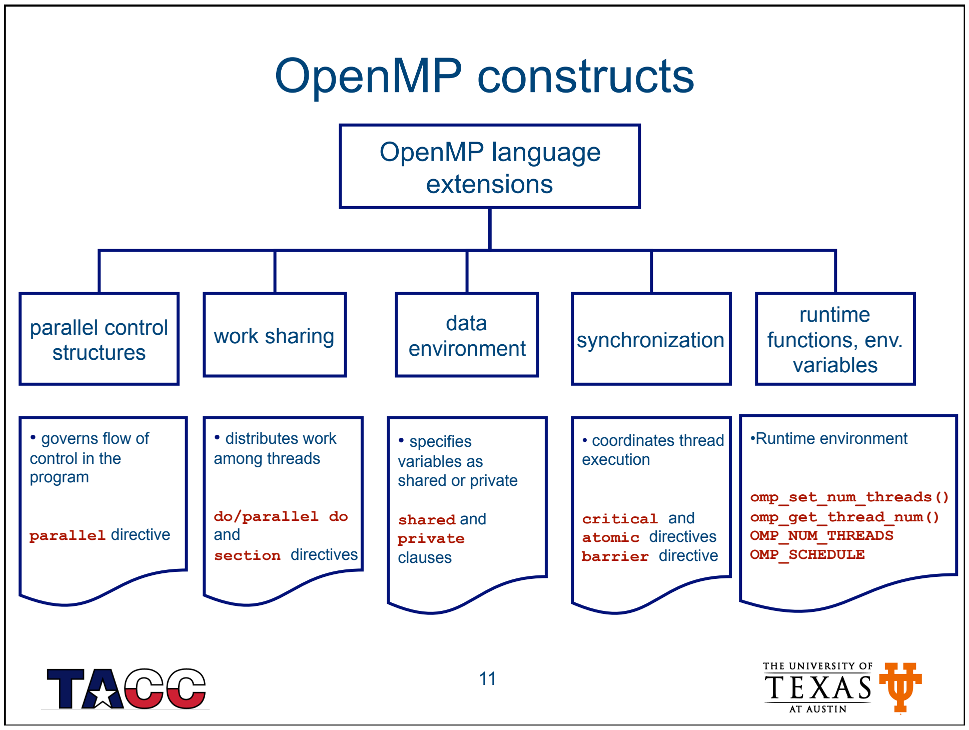
- Parallel Regions are basic “blocks” within code.
- A master thread is instantiated at run-time & persists throughout execution.
- Master thread assembles team of threads at parallel regions.





How do threads communicate?

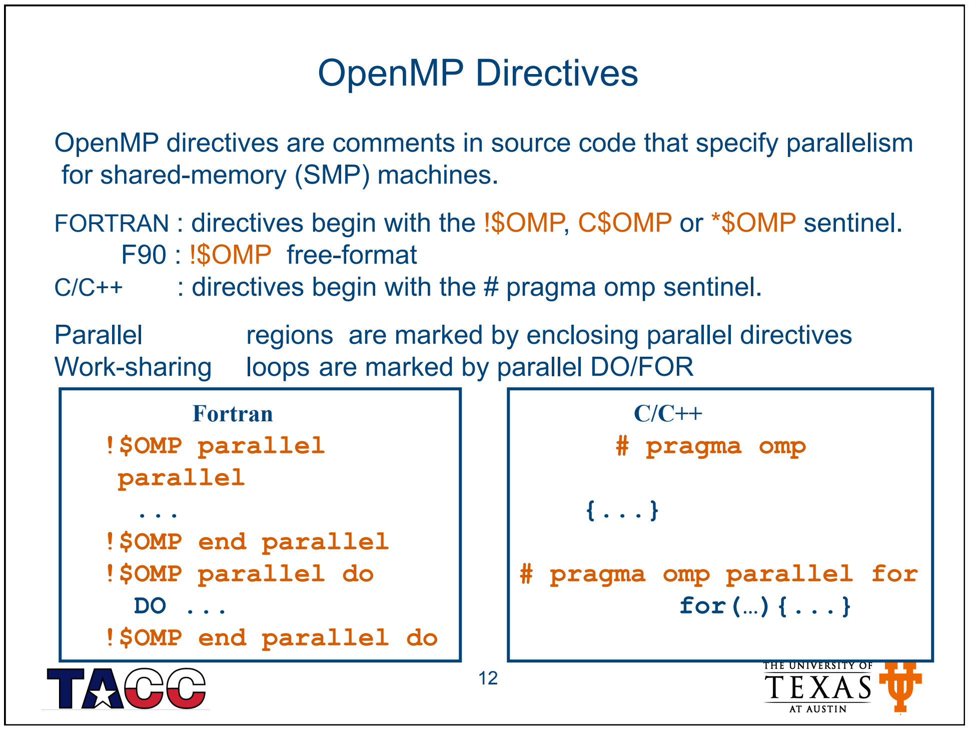
- Every thread has access to “global” memory (shared). Each thread has access to a stack memory (private).
- Use shared memory to communicate between threads.
- Simultaneous updates to shared memory can create a *race condition.* Results change with different thread scheduling.
- Use mutual exclusion to avoid data sharing --- but don’t use too many because this will serialize performance.



OpenMP constructs

OpenMP language extensions

parallel control structures	work sharing	data environment	synchronization	runtime functions, env. variables
• governs flow of control in the program parallel directive	• distributes work among threads do/parallel do and section directives	• specifies variables as shared or private shared and private clauses	• coordinates thread execution critical and atomic directives barrier directive	•Runtime environment omp_set_num_threads() omp_get_thread_num() OMP_NUM_THREADS OMP_SCHEDULE



OpenMP Directives

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

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

C/C++

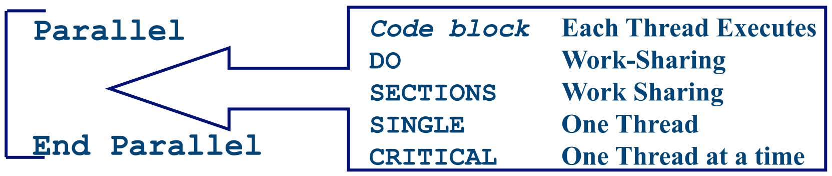
```
      # pragma omp

    {...}

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

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)

Parallel Region/Worksharing

- **Use OpenMP directives to specify Parallel Region and Work-Sharing constructs.**

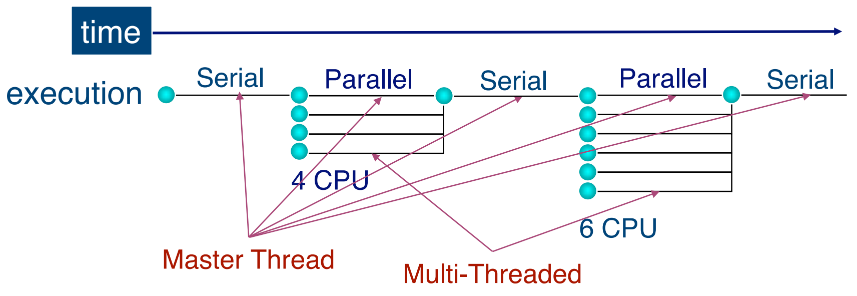


Code block	Each Thread Executes
DO	Work-Sharing
SECTIONS	Work Sharing
SINGLE	One Thread
CRITICAL	One Thread at a time

```
Parallel DO/for          Stand-alone
Parallel SECTIONS        Parallel Constructs
```

Code Execution: What happens during OpenMP?

Execution begins with a single "Master Thread".
- **A team of threads is created at each parallel region.**
  - **Number of threads equals OMP_NUM_THREADS.**
  - **Thread executions are distributed among available processors.**
- **Execution is continued after parallel region by the Master Thread.**



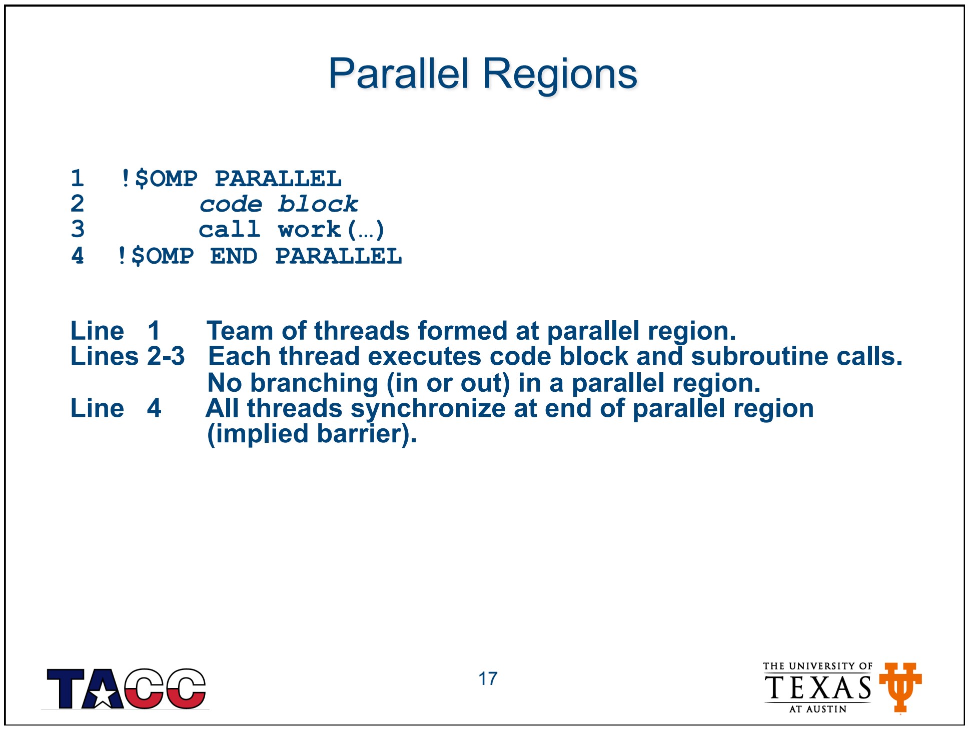
More about OpenMP parallel regions…

There are two OpenMP "modes"

- In *static* mode
  - Programmer makes use of a fixed number of threads
- In *dynamic* mode:
  - the number of threads can change under user control from one parallel region to another (use function `OMP_set_num_threads`)
  - specified by setting an environment variable

```
setenv OMP_DYNAMIC true
```

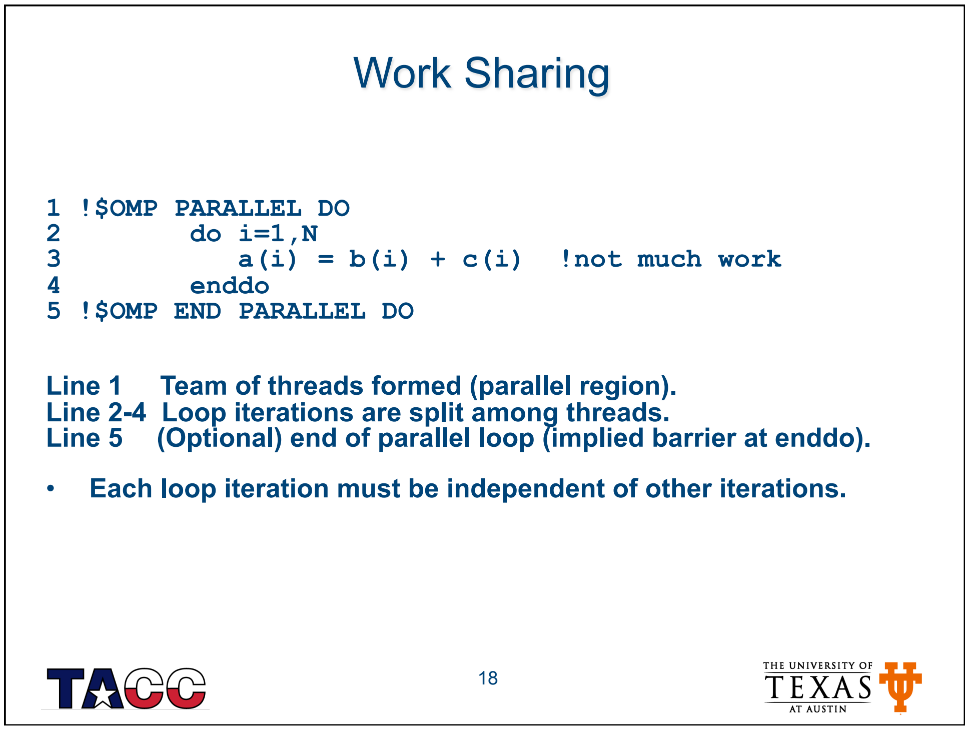
Note: the user can only define the maximum number of threads, compiler can use a smaller number



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).



Work Sharing

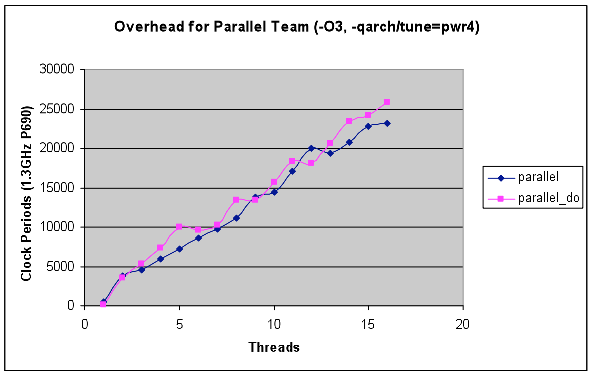
```
1 !$OMP PARALLEL DO
2        do i=1,N
3           a(i) = b(i) + c(i)   !not much work
4        enddo
5 !$OMP END PARALLEL DO
```

Line 1   Team of threads 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.

Team Overhead

Example from Champion (IBM system)



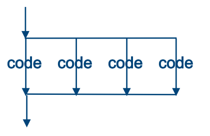
OpenMP (parallel constructs)

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

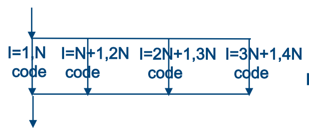
```
PARALLEL
   {code}
END PARALLEL
```

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

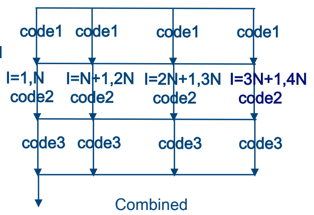
```
PARALLEL
      {code1}
DO
   do I = 1,N*4
      {code2}
   end do
      {code3}
END PARALLEL
```



Replicated



Work Sharing



Combined

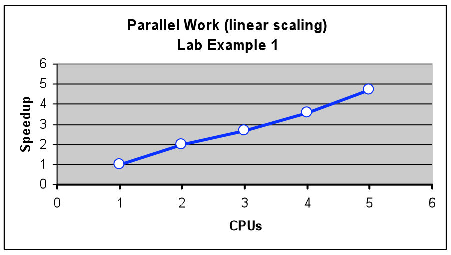
Merging Parallel Regions

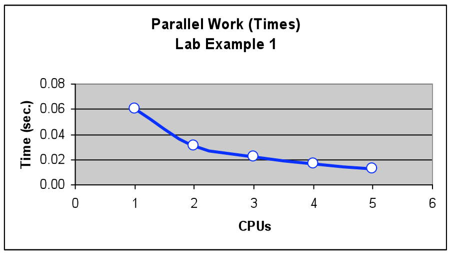
The !$OMP PARALLEL directive declares an entire region as parallel.
Merging work-sharing constructs into a single parallel region eliminates the overhead of separate team formations.

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

Parallel Work

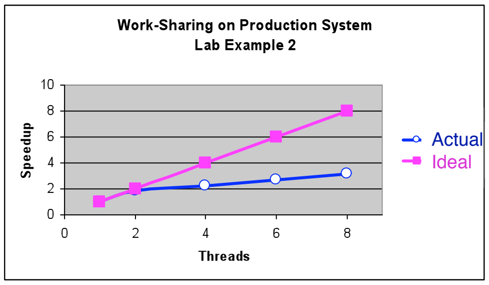


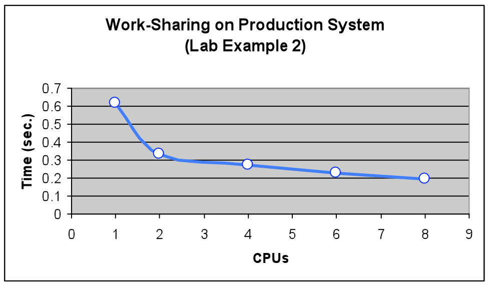


Speedup =
cpu-time(1) / cpu-time(N)

If work is completely parallel, scaling is linear.

Work-Sharing





Scheduling, memory contention and overhead can impact speedup.

Distribution of work - SCHEDULE Clause

```
!OMP$ PARALLEL DO SCHEDULE(STATIC)
```

Each CPU receives one set of contiguous iterations (~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 CPUs become available.

```
!OMP$ PARALLEL DO SCHEDULE(GUIDED,N)
```

Each of the iterations are handed out in pieces of exponentially decreasing size with N minimum number of iterations to dispatch each time (Important for load balancing.)

Comparison of scheduling options

name	type	chunk	chunk size	number of chunks	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	high
runtime	runtime	no	varies	varies	varies	varies

Example - SCHEDULE(STATIC,16)

```
!$OMP parallel do schedule(static,16)
      do i=1,128                !OMP_NUM_THREADS=4
       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

Dynamic

Pros: **potential for better load balancing, especially if chunk is low**

Cons: **higher compute overhead**
synchronization cost associated per chunk of work

Static

Pros: **low compute overhead**
no synchronization overhead per chunk
takes better advantage of data locality

Cons: **cannot compensate for load imbalance**

Comparison of scheduling options

- **When shared array data is reused multiple times, prefer static scheduling to dynamic**
- **Every invocation of the scaling would divide the iterations among CPUs the same way for static but not so for dynamic scheduling**

```
!$OMP parallel private (i,j,iter)
   do iter=1,niter
      ...
!$OMP do
      do j=1,n
         do i=1,n
            A(i,j)=A(i,j)*scale
         end do
      end do
      ...
   end do
!$OMP end parallel
```

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.

TACC 29 THE UNIVERSITY OF TEXAS AT AUSTIN

PRIVATE and 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.

TACC 30 THE UNIVERSITY OF TEXAS AT AUSTIN

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.

TACC 31 THE UNIVERSITY OF TEXAS AT AUSTIN

Default variable scoping in Fortran

```
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
```

TACC 32 THE UNIVERSITY OF TEXAS AT AUSTIN

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

REDUCTIONS

An operation that "combines" multiple elements to form a single result, such as a summation, is called a reduction operation. 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
      ...
!$OMP PARALLEL DO REDUCTION(+:asum) REDUCTION(*:aprod)
      do i=1,N
         asum  = asum  + a(i)
         aprod = aprod * a(i)
      enddo
!$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.

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 each thread 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)
```

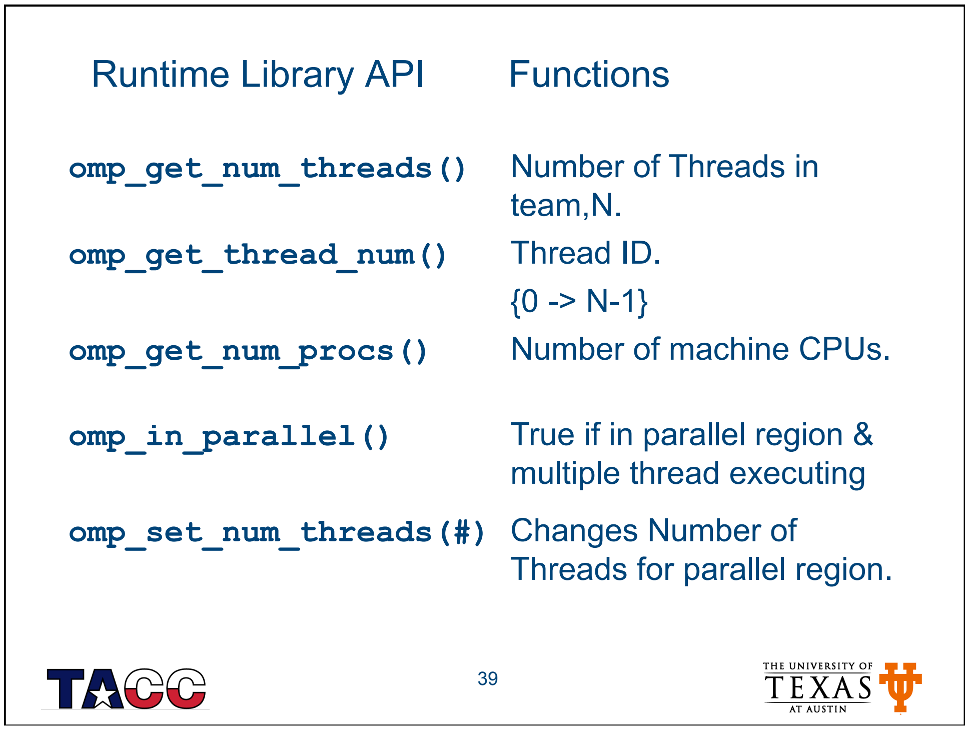
TACC 37 THE UNIVERSITY OF TEXAS AT AUSTIN

Overhead associated with mutual exclusion

All measurements were made in dedicated mode

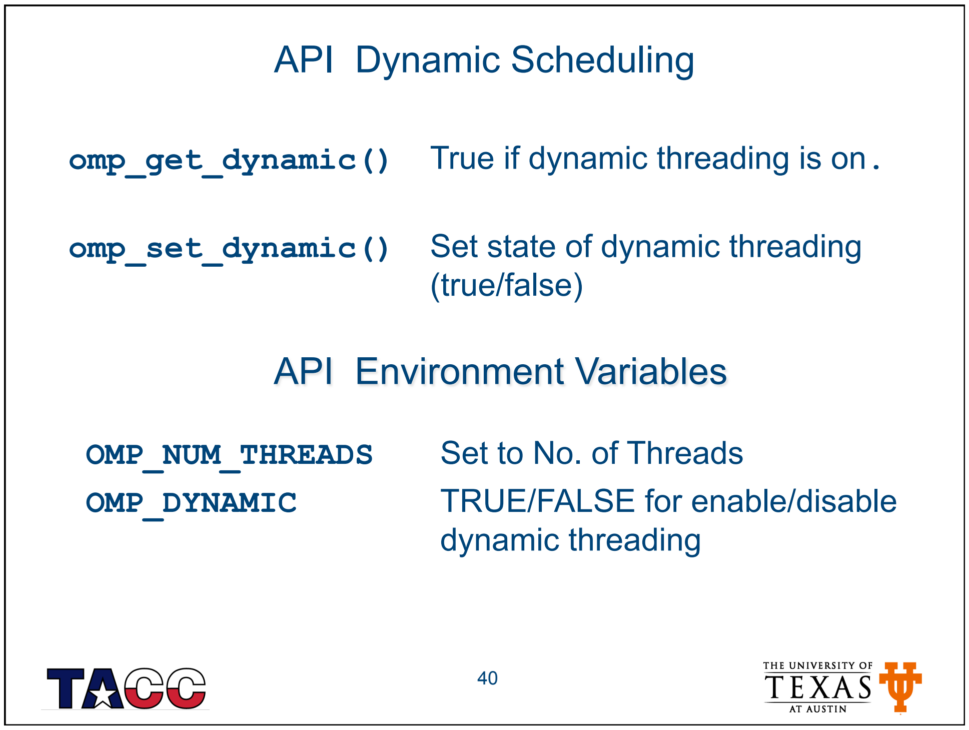
Open MP exclusion routine/directive	**cycles**
OMP_SET_LOCK/OMP_UNSET_LOCK	330
OMP_ATOMIC	480
OMP_CRITICAL	510

TACC 38 THE UNIVERSITY OF TEXAS AT AUSTIN



Runtime Library API	Functions
`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(#)`	Changes Number of Threads for parallel region.

39



API Dynamic Scheduling

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

API Environment Variables

`OMP_NUM_THREADS`	Set to No. of Threads
`OMP_DYNAMIC`	TRUE/FALSE for enable/disable dynamic threading

40

What’s new? -- OpenMP 2.0/2.5

- Wallclock timers
- Workshare directive (Fortran)
- Reduction on array variables
- NUM_THREAD clause

TACC

THE UNIVERSITY OF TEXAS AT AUSTIN

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);
```

TACC

THE UNIVERSITY OF TEXAS AT AUSTIN

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

- Array variables may now appear in the REDUCTION clause

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

- Exceptions are assumed size and deferred shape arrays
- Variable must be shared in the enclosing context

NUM_THREADS clause

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

  Usage:

```
!$OMP PARALLEL NUM_THREADS(scalar integer expression)
   <code block>
!$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

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