
Programming Shared-memory Platforms with OpenMP

Introduction to OpenMP

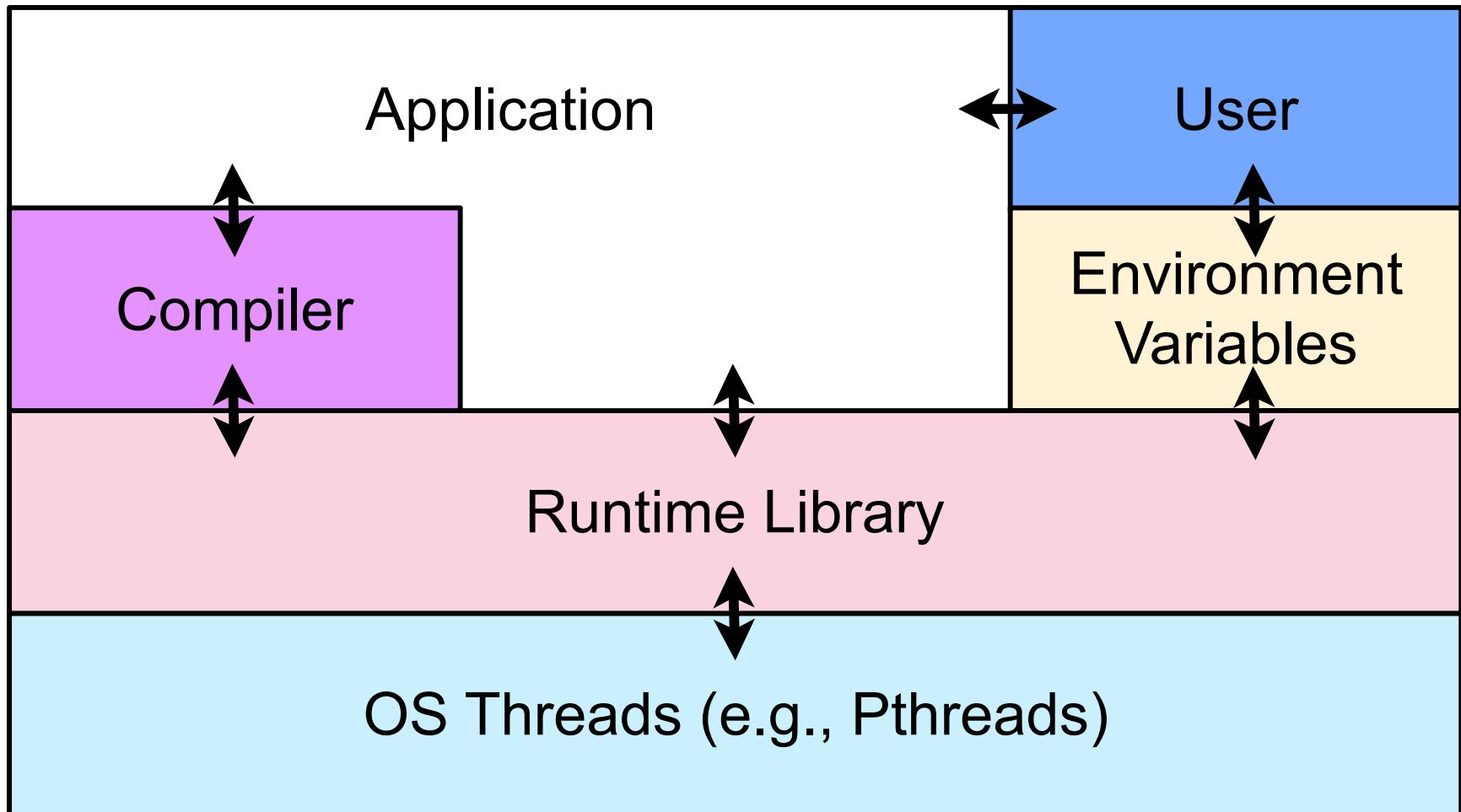
- **Introduction to OpenMP**
- **OpenMP directives**
 - concurrency directives**
 - parallel regions
 - loops, sections, tasks
 - synchronization directives**
 - reductions, barrier, critical, ordered
 - data handling clauses**
 - shared, private, firstprivate, lastprivate
 - tasks**
- **Performance tuning hints**
- **Library primitives**
- **Environment variables**

What is OpenMP?

Open specifications for Multi Processing

- An API for explicit multi-threaded, shared memory parallelism
- Three components
 - compiler directives
 - runtime library routines
 - environment variables
- Higher-level than library-based programming models
 - implicit mapping and load balancing of work
- Portable
 - API is specified for C/C++ and Fortran
 - implementations on almost all platforms
- Standard

OpenMP at a Glance



OpenMP Is Not

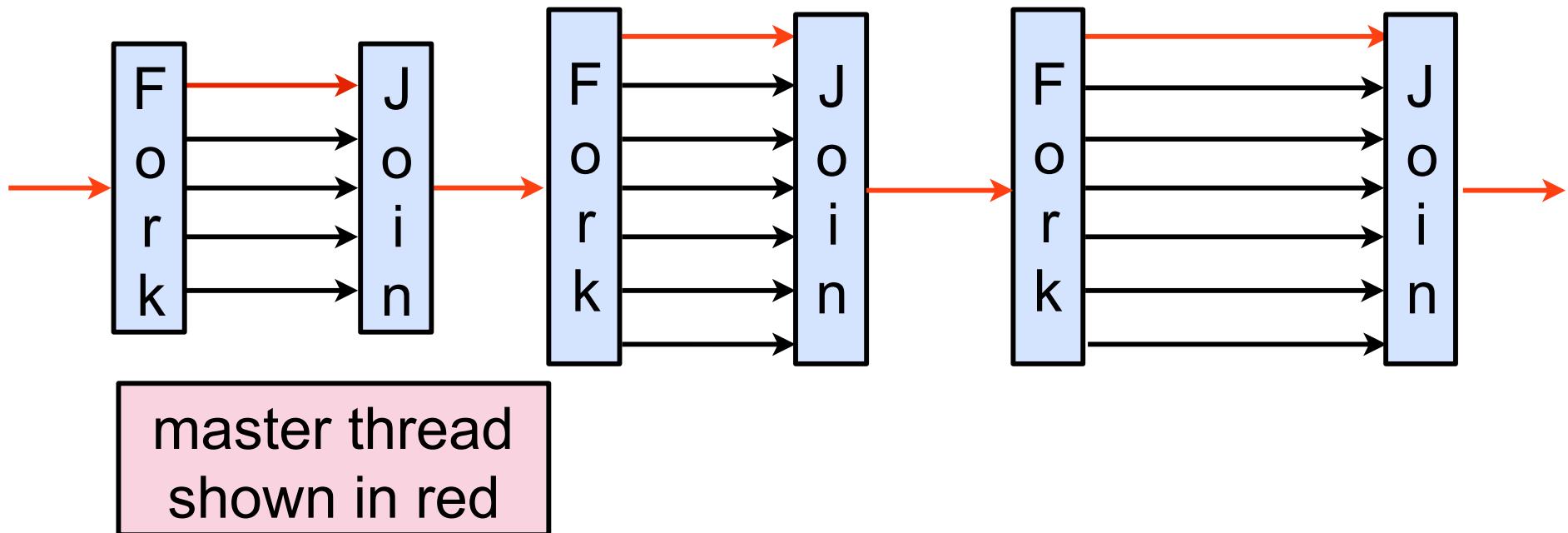
- An automatic parallel programming model
 - **parallelism is explicit**
 - **programmer has full control (and responsibility) over parallelization**
- Meant for distributed-memory parallel systems (by itself)
 - **designed for shared address space machines**
- Necessarily implemented identically by all vendors
- Guaranteed to make the most efficient use of the memory hierarchy

OpenMP Targets Ease of Use

- OpenMP does not require that single-threaded code be changed for threading
 - enables incremental parallelization of a serial program
- OpenMP relies primarily on compiler directives
 - pragmas (C/C++); significant comments in Fortran
 - if a compiler does not recognize a directive, it ignores it
 - significant parallelism possible using just 3 or 4 directives
 - both coarse-grain and fine-grain parallelism
- If the compiler is not instructed to process OpenMP directives, the program will execute sequentially
- Runtime routines have default implementations if a compiler is not instructed to process OpenMP directives

OpenMP: Fork-Join Parallelism

- An OpenMP program begins execution as a single master thread
- Master thread executes sequentially until 1st parallel region
- When a parallel region is encountered, master thread
 - creates a group of threads
 - becomes the master of this group of threads
 - is assigned the thread id 0 within the group



OpenMP Directive Format

- OpenMP directive forms
 - C and C++ use compiler directives
 - prefix: `#pragma ...`
 - Fortran uses significant comments
 - prefixes: `!$omp`, `c$omp`, `*$omp`
- Components: prefix, directive, optional clauses
 - C: `#pragma omp parallel num_threads(4)...`
 - Fortran: `!$omp parallel num_threads(4)...`

A Simple Example Using `parallel`

Program

```
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel num_threads(4)
    {
        int i = omp_get_thread_num();
        printf("Hello from thread %d\n", i);
    }
}
```

Output

```
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
```

order of output may vary!

OpenMP **parallel** Region Directive

```
#pragma omp parallel [clause list]
```

Typical clauses in [clause list]

- Conditional parallelization
 - **if (scalar expression)**
 - determines whether the **parallel** construct creates threads
- Degree of concurrency
 - **num_threads (integer expression) :** max # threads to create
- Data Scoping
 - **private (variable list)**
 - specifies variables local to each thread
 - **firstprivate (variable list)**
 - similar to **private**
 - private variables are initialized to variable value before the parallel directive
 - **shared (variable list)**
 - specifies that variables are shared among all threads in the region
 - **default (data scoping specifier)**
 - default data scoping specifier may be **shared or none**

A few more clauses
on slide 38

Interpreting an OpenMP Parallel Directive

```
#pragma omp parallel if (n > LARGE) num_threads(k + 1) \
    shared(b) private(a) firstprivate(c) default(none)
{
    /* structured block */
}
```

Meaning

- **if (n > LARGE)**
 - if **n > LARGE**, the parallel region should create threads
- **num_threads(k + 1)**
 - the value **k + 1** is the number of threads requested
- **shared(b)**
 - each thread shares a single copy of variable **b**
- **private(a) firstprivate(c)**
 - each thread gets private copies of variables **a** and **c**
 - each private copy of **c** is initialized with the value of **c** in the “initial thread” (the thread that encounters the parallel directive)
- **default(none)**
 - default state of a variable is specified as **none** (rather than **shared**)
 - signals error if not all variables are specified as **shared** or **private**

Specifying Worksharing

Within the scope of a parallel directive, a worksharing directive indicates that threads should cooperatively execute statements, blocks, iterations or tasks

OpenMP provides four directives

- **do / for**: threads cooperatively execute loop iterations
- **sections**: threads cooperatively execute tasks
- **single**: one arbitrary thread executes the code; others wait
- **workshare**: partitions execution of statements in a block

Worksharing **DO/for** Directive

for directive partitions parallel iterations across threads

do is the analogous directive for Fortran

- Usage:

```
#pragma omp for [clause list]  
/* for loop */
```

- Possible clauses in [clause list]

- **private, firstprivate, lastprivate**
- **reduction**
- **schedule, nowait, and ordered**

- Implicit barrier at end of **for** loop

A Simple Example Using **parallel** and **for**

Program

```
void main() {  
#pragma omp parallel num_threads(3)  
{  
    int i;  
    printf("Hello world\n");  
#pragma omp for  
    for (i = 1; i <= 4; i++) {  
        printf("Iteration %d\n",i);  
    }  
    printf("Goodbye world\n");  
}  
}
```

Output

```
Hello world  
Hello world  
Hello world  
Iteration 1  
Iteration 2  
Iteration 3  
Iteration 4  
Goodbye world  
Goodbye world  
Goodbye world
```

Reduction Clause for Parallel Directive

Specifies how to combine local copies of a variable in different threads into a single copy at the master when threads exit

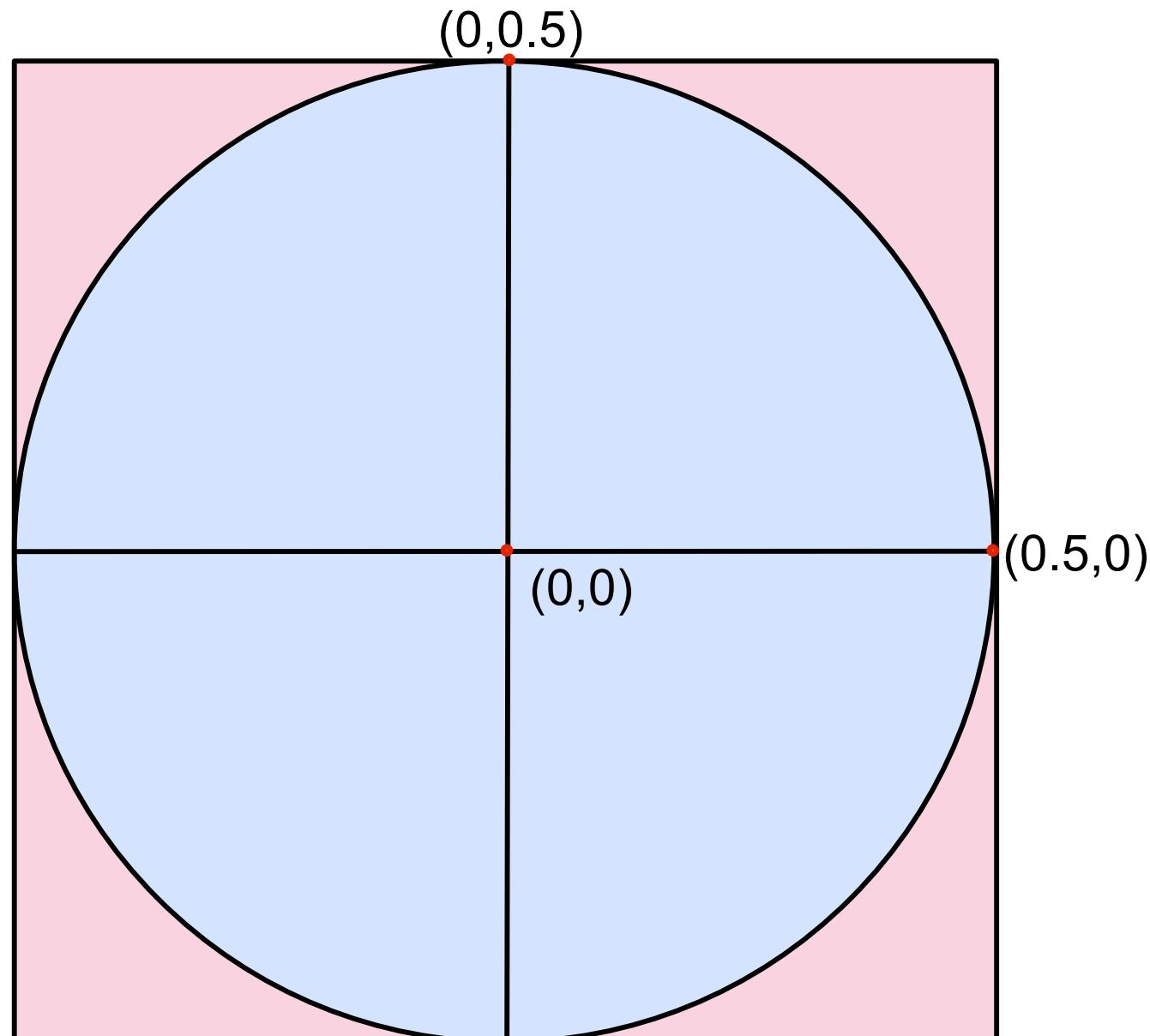
- Usage: **reduction (operator: variable list)**
 - variables in list are implicitly private to threads
- Reduction operators: +, *, -, &, |, ^, &&, and ||
- Usage sketch

```
#pragma omp parallel reduction(+: sum) num_threads(8)
{
/* compute local sum in each thread here */
}
/* sum here contains sum of all local instances of sum */
```

Running Example: Monte Carlo Estimation of Pi

Approximate Pi

- generate random points with $x, y \in [-0.5, 0.5]$
- test if point inside the circle, i.e.,
 $x^2 + y^2 < (0.5)^2$
- ratio of circle to square =
 $\pi r^2 / 4r^2 = \pi / 4$
- $\pi \approx 4 * (\text{number of points inside the circle}) / (\text{number of points total})$



OpenMP Reduction Clause Example

OpenMP threaded program to estimate Pi

```
#pragma omp parallel default(private) shared (npoints) \
reduction(+: sum) num_threads(8)
```

```
{
```

```
    num_threads = omp_get_num_threads();
    sample_points_per_thread = npoints / num_threads;
    sum = 0; seed = 17 * omp_get_thread_num();
    for (i = 0; i < sample_points_per_thread; i++) {
        coord_x =(double)(rand_r(&seed))/(double)(RAND_MAX) - 0.5;
        coord_y =(double)(rand_r(&seed))/(double)(RAND_MAX) - 0.5;
        if ((coord_x * coord_x + coord_y * coord_y) < 0.25)
            sum++;
}
```

here, user
manually
divides work

```
}
```

- a local copy of sum for each thread
- all local copies of sum added together and stored in master

Using Worksharing for Directive

```
#pragma omp parallel default(private) shared (npoints) \
reduction(+: sum) num_threads(8)

{
    sum = 0;
    seed = 17 * omp_get_thread_num();
    #pragma omp for
    for (i = 0; i < npoints; i++) {
        rand_no_x =(double)(rand_r(&seed))/(double)(RAND_MAX);
        rand_no_y =(double)(rand_r(&seed))/(double)(RAND_MAX);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            sum++;
    }
}
```

worksharing `for` divides work

Implicit barrier at end of loop

Mapping Iterations to Threads

schedule clause of the `for` directive

- Recipe for mapping iterations to threads
- Usage: `schedule(scheduling_class[,chunk])`.
- Four scheduling classes
 - **static**: work partitioned at compile time
 - iterations statically divided into pieces of size *chunk*
 - statically assigned to threads
 - **dynamic**: work evenly partitioned at run time
 - iterations are divided into pieces of size *chunk*
 - chunks dynamically scheduled among the threads
 - when a thread finishes one chunk, it is dynamically assigned another
 - default chunk size is 1
 - **guided**: guided self-scheduling
 - chunk size is exponentially reduced with each dispatched piece of work
 - the default minimum chunk size is 1
 - **runtime**:
 - scheduling decision from environment variable `OMP_SCHEDULE`
 - illegal to specify a chunk size for this clause.

Statically Mapping Iterations to Threads

```
/* static scheduling of matrix multiplication loops */  
#pragma omp parallel default(private) \  
    shared (a, b, c, dim) num_threads(4)  
#pragma omp for schedule(static)  
for (i = 0; i < dim; i++) {  
    for (j = 0; j < dim; j++) {  
        c(i,j) = 0;  
        for (k = 0; k < dim; k++) {  
            c(i,j) += a(i, k) * b(k, j);  
        }  
    }  
}
```

static schedule maps iterations
to threads at compile time

Avoiding Unwanted Synchronization

- Default: worksharing **for** loops end with an implicit barrier
- Often, less synchronization is appropriate
 - series of independent **for**-directives within a parallel construct
- **nowait** clause
 - modifies a **for** directive
 - avoids implicit barrier at end of for

Avoiding Synchronization with `nowait`

```
#pragma omp parallel  
{  
    #pragma omp for nowait  
    for (i = 0; i < nmax; i++)  
        a[i] = ...;
```

```
#pragma omp for ←  
for (i = 0; i < mmax; i++)  
    b[i] = ... anything but a ...;  
}
```

any thread can begin second loop immediately without
waiting for other threads to finish first loop

Worksharing **sections** Directive

sections directive enables specification of task parallelism

- Usage

```
#pragma omp sections [clause list]
```

```
{
```

```
[ #pragma omp section
```

```
/* structured block */
```

```
]
```

```
[ #pragma omp section
```

```
/* structured block */
```

```
]
```

```
...
```

```
}
```

brackets here represent that
section is optional,
not the syntax for using them

Using the **sections** Directive

```
#pragma omp parallel  
{
```

parallel section encloses all parallel work

```
#pragma omp sections  
{
```

sections: task parallelism

```
#pragma omp section  
{  
    taskA();
```

```
} #pragma omp section  
{  
    taskB();
```

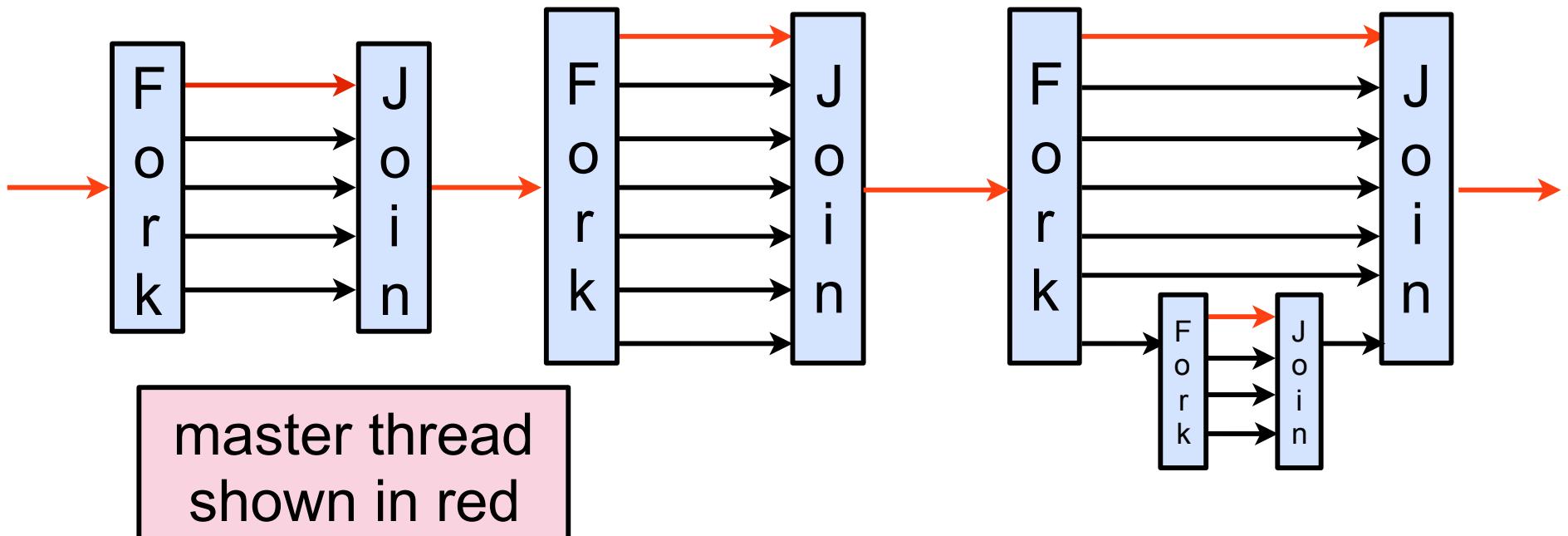
```
#pragma omp section  
{  
    taskC();
```

three concurrent tasks;
tasks need not
be procedure calls

```
}
```

Nesting parallel Directives

- Nested parallelism enabled using the **OMP_NESTED** environment variable
 - **OMP_NESTED = TRUE** → nested parallelism is enabled
- Each parallel directive creates a new team of threads



Synchronization Constructs in OpenMP

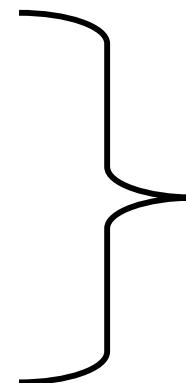
#pragma omp barrier wait until all threads arrive here

#pragma omp single [clause list]

structured block

#pragma omp master

structured block



single-threaded
execution

Use **MASTER** instead of **SINGLE** wherever possible

- **MASTER** = IF-statement with no implicit **BARRIER**
 - equivalent to

```
IF(omp_get_thread_num() == 0) {...}
```
- **SINGLE**: implemented like other worksharing constructs
 - keeping track of which thread reached **SINGLE** first adds overhead

Synchronization Constructs in OpenMP

#pragma omp critical [(name)] critical section: like a named lock
structured block

#pragma omp ordered for loops with carried dependences
structured block

Example Using **critical**

```
#pragma omp parallel
{
#pragma omp for nowait shared(best_cost)
    for (i = 0; i < nmax; i++) {
        my_cost = ...;

        ...
#pragma omp critical
{
        if (best_cost < my_cost)
            best_cost = my_cost;
    }
    ...
}
}
```

critical ensures mutual exclusion
when accessing shared state

Example Using **ordered**

```
#pragma omp parallel
{
#pragma omp for nowait shared(a)
    for (k = 0; k < nmax; k++) {
        ...
#pragma omp ordered
{
    a[k] = a[k-1] + ...;
}
...
}
}
```

ordered ensures carried dependence
does not cause a data race

Orphaned Directives

- Directives may not be lexically nested in a parallel region
 - may occur in a separate program unit

```
...  
!$omp parallel  
call phase1  
call phase2  
!$omp end parallel  
...
```

```
subroutine phase1  
!$omp do private(i) shared(n)  
do i = 1, n  
call some_work(i)  
end do  
!$omp end do  
end
```

```
subroutine phase2  
!$omp do private(j) shared(n)  
do j = 1, n  
call more_work(j)  
end do  
!$omp end do  
end
```

- Dynamically bind to enclosing parallel region at run time
- Benefits
 - enables parallelism to be added with a minimum of restructuring
 - improves performance: enables single parallel region to bind with worksharing constructs in multiple called routines
- Execution rules
 - an orphaned worksharing construct is executed serially when not called from within a parallel region

OpenMP 3.0 Tasks

- Motivation: support parallelization of irregular problems
 - unbounded loops
 - recursive algorithms
 - producer consumer
- What is a task?
 - work unit
 - execution can begin immediately, or be deferred
 - components of a task
 - code to execute, data environment, internal control variables
- Task execution
 - data environment is constructed at creation
 - tasks are executed by threads of a team
 - a task can be tied to a thread (i.e. migration/stealing not allowed)
 - by default: a task is tied to the first thread that executes it

OpenMP 3.0 Tasks

`#pragma omp task [clause list]`

Possible clauses in [clause list]

- Conditional parallelization
 - `if (scalar expression)`
 - determines whether the construct creates a task
- Binding to threads
 - `untied`
- Data scoping
 - `private (variable list)`
 - specifies variables local to the child task
 - `firstprivate (variable list)`
 - similar to the private
 - private variables are initialized to value in parent task before the directive
 - `shared (variable list)`
 - specifies that variables are shared with the parent task
 - `default (data handling specifier)`
 - default data handling specifier may be `shared` or `none`

Composing Tasks and Regions

```
#pragma omp parallel  
{  
#pragma omp task  
    x();  
#pragma omp barrier  
#pragma omp single  
{  
#pragma omp task  
    y();  
}  
}  
•
```

one x task created for each thread in the parallel region

all x tasks complete at barrier

one y task created

region end: y task completes

Data Scoping for Tasks is Tricky

If no default clause specified

- Static and global variables are **shared**
- Automatic (local) variables are **private**
- Variables for orphaned tasks are **firstprivate** by default
- Variables for non-orphaned tasks inherit the shared attribute
 - task variables are **firstprivate** unless **shared** in the enclosing context

Fibonacci using OpenMP 3.0 Tasks

```
int fib ( int n )
{
    int x,y;
    if ( n < 2 ) return n;
#pragma omp task shared(x)
    x = fib(n - 1);
#pragma omp task shared(y)
    y = fib(n - 2);
# pragma omp taskwait
    return x + y;
}
```

need **shared** for x and y;
default would be
firstprivate

suspend parent task until
children finish

```
int main (int argc, char **argv)
{
    int n, result;
    n = atoi(argv[1]);
# pragma omp parallel
{
# pragma omp single
{
    result = fib(n);
}
printf("fib(%d) = %d\n",
n, result);
}
```

create team
of threads to
execute tasks

only one thread
performs the
outermost call

List Traversal

```
Element first, e;  
#pragma omp parallel  
#pragma omp single  
{  
    for (e = first; e; e = e->next)  
#pragma omp task firstprivate(e)  
    process(e);  
}
```

Is the use of variables safe as written?

Task Scheduling

- Tied tasks
 - only the thread that the task is tied to may execute it
 - task can only be suspended at a suspend point
 - task creation
 - task finish
 - taskwait
 - barrier
 - if a task is not suspended at a barrier, it can only switch to a descendant of any task tied to the thread
- Untied tasks
 - no scheduling restrictions
 - can suspend at any point
 - can switch to any task
 - implementation may schedule for locality and/or load balance

Summary of Clause Applicability

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	•				•	•
PRIVATE	•	•	•	•	•	•
SHARED	•	•			•	•
DEFAULT	•				•	•
FIRSTPRIVATE	•	•	•	•	•	•
LASTPRIVATE		•	•		•	•
REDUCTION	•	•	•		•	•
COPYIN	•				•	•
SCHEDULE		•			•	
ORDERED		•			•	
NOWAIT		•	•	•		

Performance Tuning Hints

Parallelize at the highest level, e.g. outermost **DO / for** loops

```
!$OMP PARALLEL  
....  
do j = 1, 20000  
 !$OMP DO  
   do k = 1, 10000  
   ...  
   enddo !k  
 !$OMP END DO  
 enddo !j  
 ...  
 !$OMP END PARALLEL
```

```
!$OMP PARALLEL  
....  
 !$OMP DO  
   do k = 1, 10000  
     do j = 1, 20000  
     ...  
     enddo !j  
   enddo !k  
 !$OMP END DO  
 ...  
 !$OMP END PARALLEL
```

Slower

Faster

Performance Tuning Hints

Merge independent parallel loops when possible

```
!$OMP PARALLEL  
....  
!$OMP DO  
    statement 1  
!$OMP END DO  
!$OMP DO  
    statement 2  
!$OMP END DO  
....  
!$OMP END PARALLEL
```

Slower

```
!$OMP PARALLEL  
....  
!$OMP DO  
    statement 1  
    statement 2  
!$OMP END DO  
....  
!$OMP END PARALLEL
```

Faster

Performance Tuning Hints

Minimize use of synchronization

- **BARRIER**
- **CRITICAL sections**
 - if necessary, use named CRITICAL for fine-grained locking
- **ORDERED regions**
- Use **NOWAIT clause to avoid unnecessary barriers**
 - adding NOWAIT to a region's final DO eliminates a redundant barrier
- Use explicit **FLUSH** with care
 - flushes can evict cached values
 - subsequent data accesses may require reloads from memory

```
data = ...
#pragma omp flush (data)
data_available = true;
```

OpenMP Library Functions

- Processor count

```
int omp_get_num_procs(); /* # processors currently available */  
int omp_in_parallel(); /* determine whether running in parallel */
```

- Thread count and identity

```
/* max # threads for next parallel region. only call in serial region */  
void omp_set_num_threads(int num_threads);
```

```
int omp_get_num_threads(); /*# threads currently active */  
int omp_get_max_threads(); /* max # concurrent threads */  
  
int omp_get_thread_num(); /* thread id */
```

OpenMP Library Functions

- Controlling and monitoring thread creation

```
void omp_set_dynamic (int dynamic_threads);  
int omp_get_dynamic ();  
void omp_set_nested (int nested);  
int omp_get_nested ();
```

- Mutual exclusion

```
void omp_init_lock(omp_lock_t *lock);  
void omp_destroy_lock(omp_lock_t *lock);  
  
void omp_set_lock(omp_lock_t *lock);  
void omp_unset_lock(omp_lock_t *lock);  
int omp_test_lock(omp_lock_t *lock);
```

- Lock routines have a nested lock counterpart for recursive mutexes

OpenMP Environment Variables

- **OMP_NUM_THREADS**
 - specifies the default number of threads for a parallel region
- **OMP_DYNAMIC**
 - specifies if the number of threads can be dynamically changed
- **OMP_NESTED**
 - enables nested parallelism (may be nominal: one thread)
- **OMP_SCHEDULE**
 - specifies scheduling of **for-loops** if the clause specifies runtime
- **OMP_STACKSIZE** (for non-master threads)
- **OMP_WAIT_POLICY** (active or passive)
- **OMP_MAX_ACTIVE_LEVELS**
 - integer value for maximum # nested parallel regions
- **OMP_THREAD_LIMIT** (# threads for entire program)

OpenMP Directives vs. Library-based Models

- Directive advantages
 - directives facilitate a variety of thread-related tasks
 - frees programmer from
 - initializing thread attributes
 - setting up thread arguments
 - partitioning iteration spaces, ...
- Directive disadvantages
 - data exchange is less apparent
 - leads to mysterious overheads
 - data movement, false sharing, and contention
 - API is less expressive than Pthreads
 - lacks condition waits, locks of different types, and flexibility for building composite synchronization operations

OpenMP is Continuing to Evolve

- OpenMP 5.0 is the most recent standard (November 2018)
- Features new to OpenMP 4
 - SIMD support
 - e.g., `a[0:n-1] = 0`
 - locality and affinity
 - control mapping of threads to processor cores
 - `proc_bind (master, spread, close)`
 - additional synchronization mechanisms
 - e.g., `taskgroup, taskwait`
 - offload computation to accelerators, e.g. GPUs
- OpenMP 5.1 will be released in November 2020
 - changes include full support for C11 and C++11/14/17, extensions to support C++ attribute specifiers; tile, error and assume directives; iterator support for data motion clauses, the interop directive, extensions to task dependences, extensions to SIMD constructs and the declare variant directive as well as clarifications and improvements to existing features

References

- Blaise Barney. LLNL OpenMP tutorial. <http://www.llnl.gov/computing/tutorials/openMP>
- Adapted from slides “Programming Shared Address Space Platforms” by Ananth Grama
- Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Introduction to Parallel Computing. Chapter 7. Addison Wesley, 2003.
- Sun Microsystems. OpenMP OpenMP API User's Guide. Chapter 7 “Performance Considerations” http://docs.sun.com/source/819-3694/7_tuning.html
- Alberto Duran. OpenMP 3.0: What’s New?. IWOMP 2008. <http://cobweb.ecn.purdue.edu/ParaMount/iwomp2008/documents/omp30>
- Stephen Blair-Chappell. “Expressing Parallelism Using the Intel Compiler.” http://www.polyhedron.com/web_images/documents/Expressing%20Parallelism%20Using%20Intel%20Compiler.pdf
- Rusty Lusk et al. Programming Models and Runtime Systems, Exascale Software Center Meeting, ANL, Jan. 2011.
- OpenMP 4.5 Standard, <https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf>
- OpenMP Application Program Interface Examples. <https://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf>

References

- OpenMP 5.0 Standard, <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf>
- OpenMP TR 8. <https://www.openmp.org/wp-content/uploads/openmp-TR8.pdf>