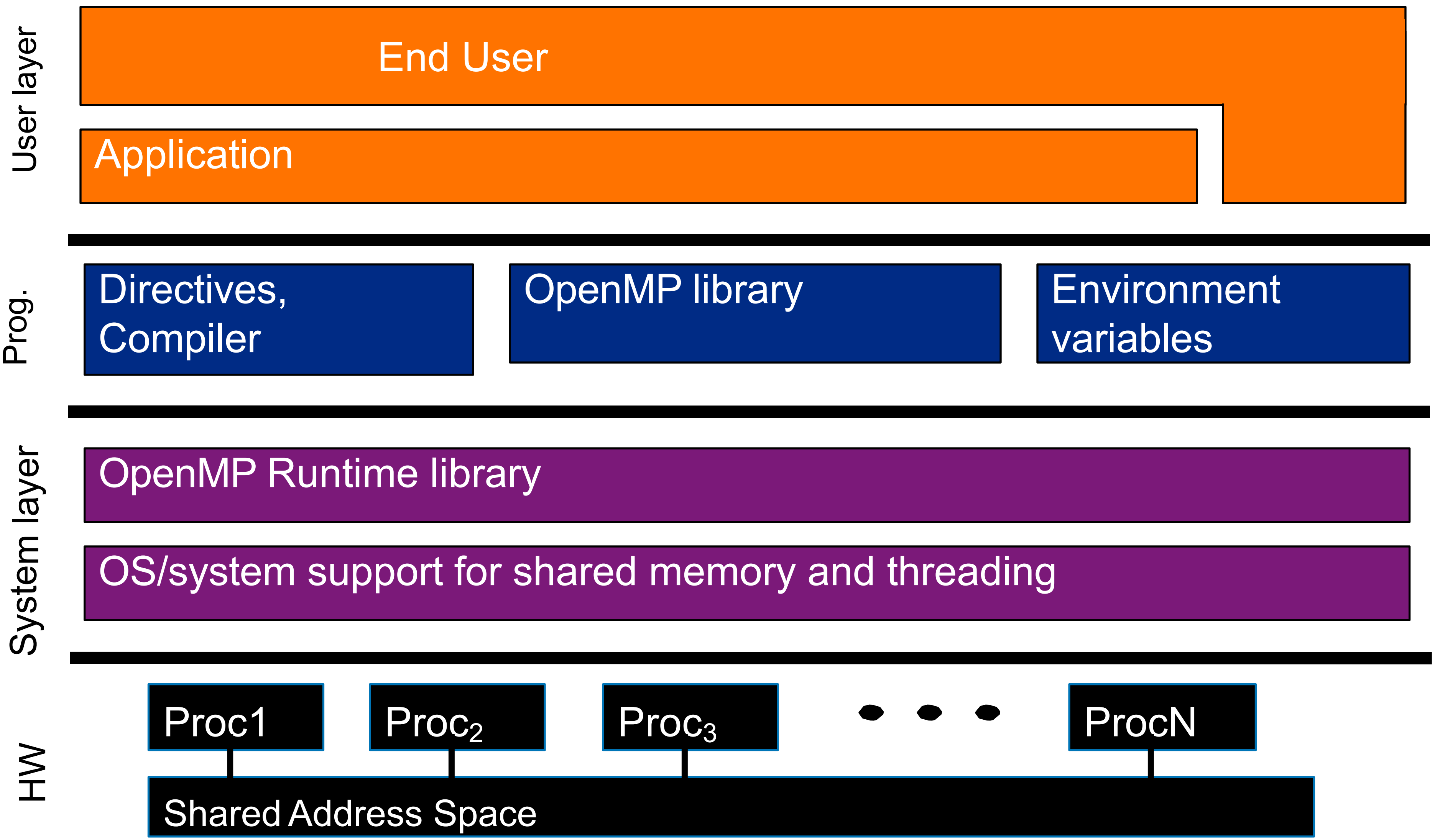
Parallel Computing Using OpenMP

CPSC 424/524
Lecture #06
September 26, 2018

OpenMP

- API for writing multithreaded applications
  - Developed in 1990s
  - Now at version 4.5, but we'll use version 4.0 supported by the compiler we're using
- Intended to be portable while delivering high performance
- Not a new parallel language, but extensions to a number of base languages already in use (C, C++, FORTRAN)
  - Coupled to compilers: small (but growing!) set of compiler directives
  - Runtime support via library routines
  - Runtime control via environment variables
  - Applicable to SMPs, vectorization, and accelerators (e.g., GPUs)

OpenMP basic definitions: Basic Solution stack

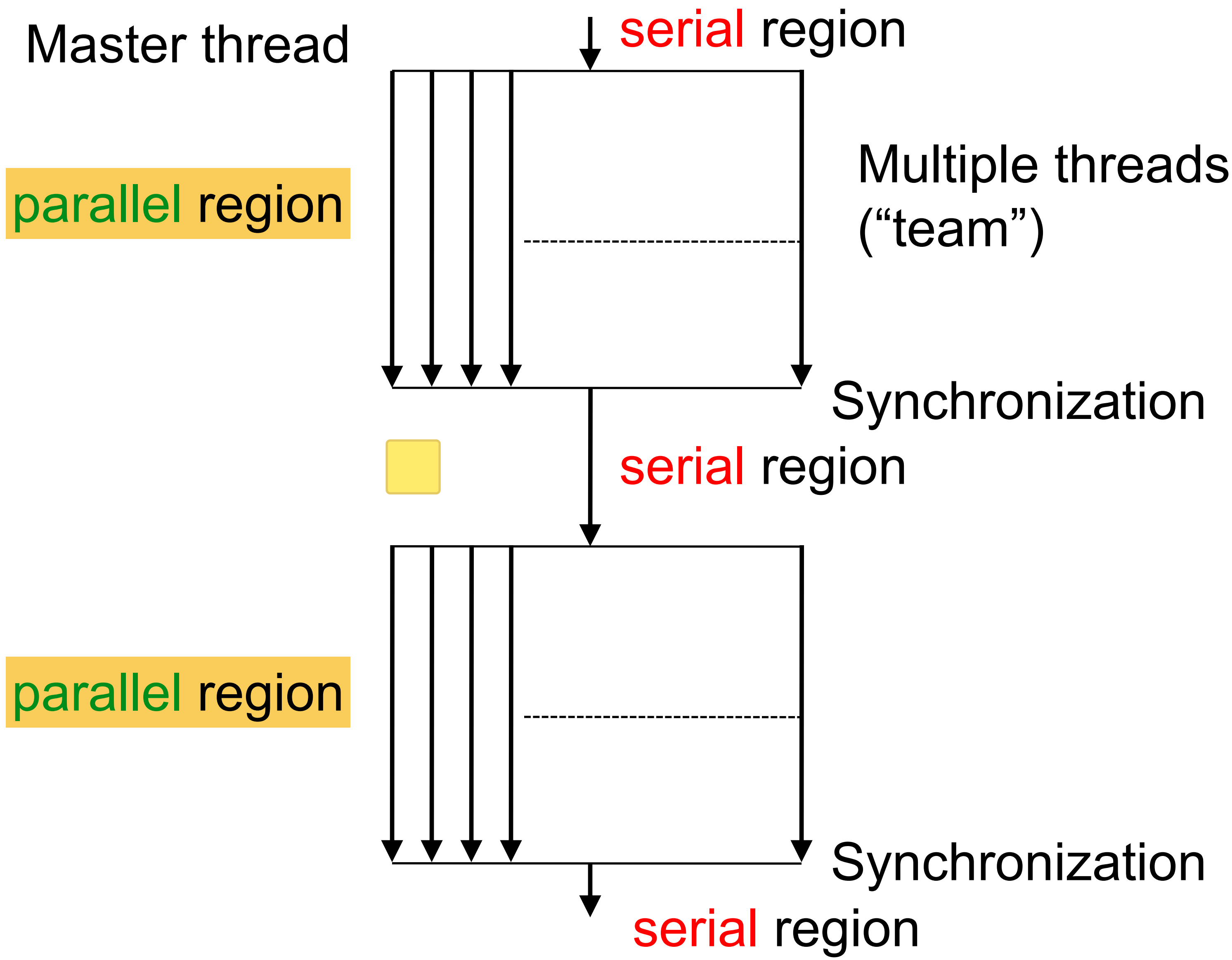


Slide from OpenMP Tutorial at SC16

OpenMP Features

- Shared memory programming using thread-based fork-join model
- OpenMP programs start with a single master thread which spawns multiple child threads (a "team") implicitly in "parallel regions"
- Multiple approaches to parallelism: Loops, Sections, Tasks
- All threads can access global memory (e.g., variables declared outside of any parallel region; file scope or static variables), and data in global memory may be either shared among all of threads or private to a single thread.
- Weakly consistent memory model
- Data transfer and cache management is generally hidden
- Synchronization & management of critical sections is often implicit

OpenMP Fork-Join Model



OpenMP Directives

- OpenMP directives are instructions to compilers or pre-compilers
- Syntax is designed so that directives don't interfere with non-OpenMP compilations or compilers that don't support OpenMP
- C/C++ Syntax:

```
#pragma omp directive_name …
```

- Fortran 77 Syntax

```
c$omp directive_name …
```

- Fortran 9x Syntax

```
!$omp directive_name …
```

- Directives may have parameters ("clauses") after the name
- Directives generally control execution of code that is specified in a structured block that follows the directive. Then the directive and structured block form a "construct."

Parallel Directive

```
#include <omp.h>

#pragma omp parallel
   <structured_block>
```

The parallel directive creates multiple threads, each one executing the specified `structured_block`, which is either a single statement or a compound statement (enclosed in “`{...}`”) having a single entry point and a single exit point.

There is an implicit barrier at the end of the construct.

Details may be controlled by including additional clauses.

“Hello World!” in OpenMP

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

int main (int argc, char **argv)
{
omp_set_num_threads(4)
#pragma omp parallel
    {
      printf("Hello World! from thread %d of %d\n",
             omp_get_thread_num(), omp_get_num_threads());
    }
}
```

OpenMP directive for a parallel region

Barrier

OpenMP library routines

Sample output :

```
Hello World! from thread 3 of 4
Hello World! from thread 1 of 4
Hello World! from thread 2 of 4
Hello World! from thread 0 of 4
```

Setting the Number of Threads in a Team

- Number of threads in the team for a parallel region can be set by:
  - `num_threads()` clause in the parallel directive
  - `omp_set_num_threads()` library routine called before parallel region
  - `OMP_NUM_THREADS` environment variable
  - Implementation default (often number of cores/processors)
- In some cases, the number of threads may be system dependent
  - May limit the number available
- Dynamic thread count
  - OMP system controls the number of threads dynamically at run time
  - Can control maximum number of threads
  - May not be supported on all systems

Variables: Shared and Private

Existing variables may be either globally shared (the default) or private to individual threads. This may be specified using `shared()` or `private()` clauses in the parallel directive, or using the `threadprivate()` directive. Newly created variables in a thread are private to the thread.

```
int tid; static x;
#pragma omp threadprivate(x)
#pragma omp parallel private(tid){
   tid = omp_get_thread_num();
   printf("Hello World! from thread %d\n", tid);
}
```

Other possibilities (several may be used):

`firstprivate:` Private; initialized on entry to master's variable value

`lastprivate:` Private; at end of block, master's variable set to "final" value, where "final" varies with construct; often used to match serial behavior for loops (value from last iteration)

Variables: Private vs. Threadprivate

"Private" variables:

- Allocated for threads in a specific parallel region
- "Mask" original variables in the master thread (if one exists)
- Each thread, master thread included, gets a (new) private copy
- No automatic initialization; initialize explicitly or use firstprivate

"Threadprivate" variables:

- Provides threads with private copies of global variables declared in the master thread that are persistent across parallel regions
- Master thread continues to use its original variable (not masked)
- Initialized once automatically (at an unspecified time); best to initialize explicitly (e.g., using copyin() clause)

OpenMP's Weak Memory Consistency

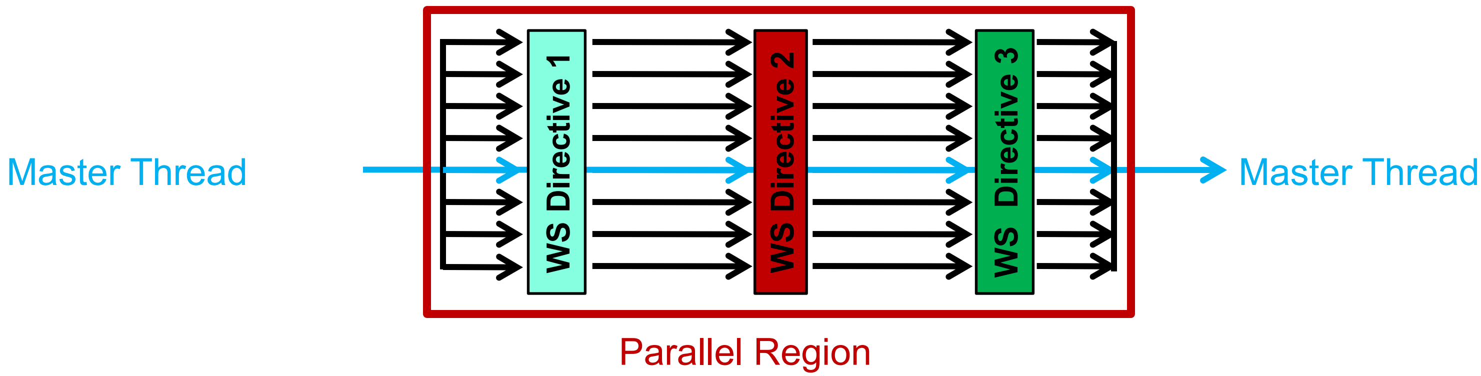
- Relaxed (or weak) consistency: At any time, a thread's local, temporary view of shared memory may differ from other threads' views and from the actual contents of the memory.
- In most cases, variable reads/writes use the local temporary view until it is forced back into shared memory. Generally, the compiler and runtime system will force consistency when needed. Moreover, synchronizations like barriers implicitly force consistency.
- You can also use the flush() directive to explicitly force consistency of some or all of the local view, but this is rarely needed. Note that if two threads perform concurrent flushes, the result is as if the flushes were serialized (one after the other). This can lead to race conditions that programmers must manage.

Useful Library Routines

- `int omp_get_num_threads(void)`
  - Returns number of threads currently being used in parallel directive
- `int omp_get_thread_num(void)`
  - Returns my thread number `t`. (Master is always thread 0.)
  - `0 <= t < omp_get_num_threads()`
- `void omp_set_num_threads(int num_threads)`
  - Sets number of threads to use (overrides `OMP_NUM_THREADS`)
- `int omp_get_thread_limit (void)`
  - Returns maximum number of threads available to the program
- `int omp_get_num_procs(void)`
  - Returns the number of processors (cpus/cores) available
- `int omp_in_parallel(void)`
  - C/C++: Returns nonzero value if in parallel region; 0 otherwise
  - Fortran: Returns `.TRUE.` if in parallel region; `.FALSE.` otherwise

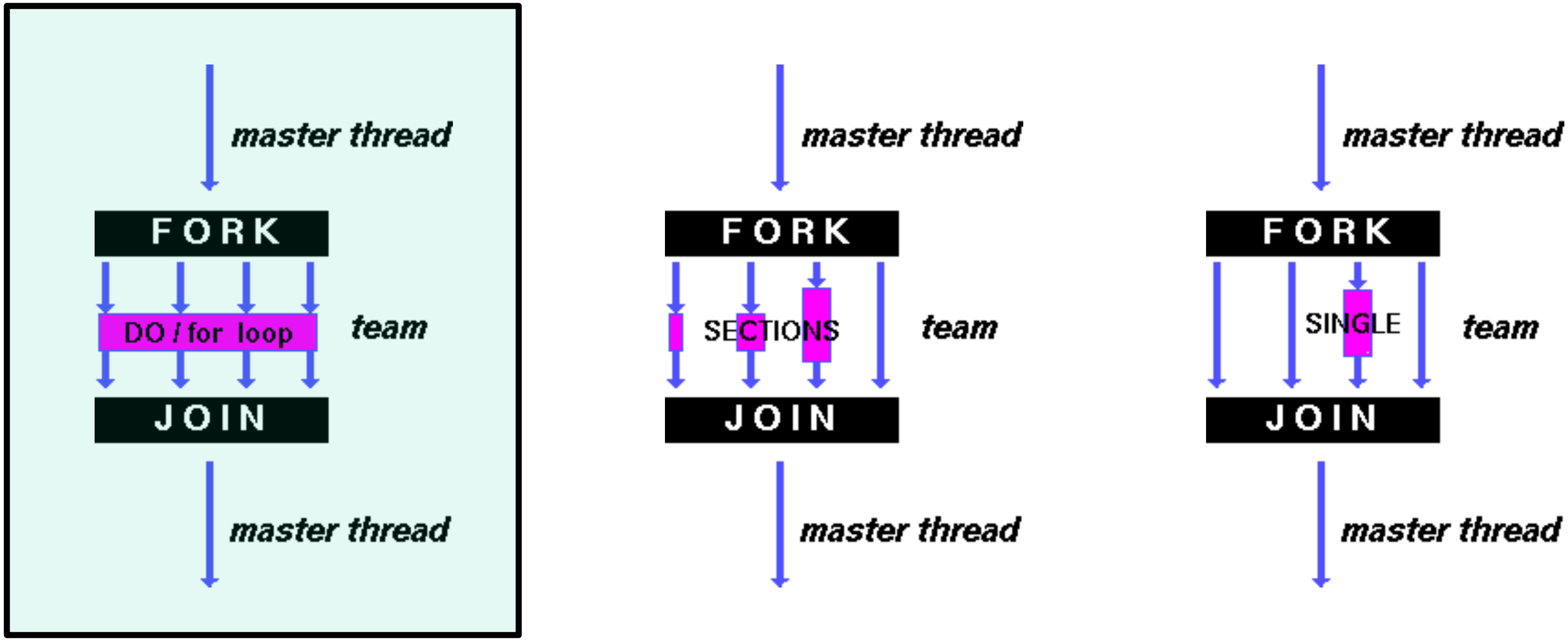
Basic Work Sharing in OpenMP

- The parallel directive simply sets up a team of threads all executing the same code in a parallel region. Additional directives are required to distribute work among the threads.



- Common Work Sharing Directives: These operate within a parallel region, with an implicit barrier at the end of each one unless you use a nowait clause:
  - Loops: Iterative computation is shared among the threads
  - Sections: Each thread runs a different block of code
  - Single: Block of code executed by exactly one of the threads
  - Master: Block of code executed only by the master (no sync!)

Work-Sharing Constructs



For Loops

The directive

```
#pragma omp for
        for_loops
```

causes the loop iterations to be divided into parts, with each part executed by one of the threads in the team, which must already exist.

The for loops must satisfy:

1. The iterations must be independent—It's up to you to check this!
2. Total number of iterations must be computable in advance at runtime
3. Loop termination must be simple, depending only on <, <=, > or >=
4. Loop increment must be simple addition or subtraction involving a fixed increment. (E.g.: `i++`, `i+=incr`, `i-=incr`)
5. Loop must not terminate with a `break` statement
6. Only 1 level of nested loops is parallelized unless the loops are "perfectly nested" and the collapse() clause is used.

Iteration Independence

Assume that **a** and **b** are non-overlapping and that arrays are large enough

```
for (i=1; i<n; i++) b[i] = a[i-1];
```

```
for (i=0; i<n; i++) b[i] = a[j][i]*b[i+n];
```

```
a[0] = f(0);
for (i=1; i<n; i++) {
      b[i] = a[i-1]*b[i];
      a[i] = f(i);
}
```

Fix: Loop splitting

```
for (i=1; i<n; i++) a[i] = f(i);
for (i=1; i<n; i++) b[i] = a[i-1]*b[i];
```

Iteration Independence (Cont.)

```
for (i=1; i<n; i++) b[i] = b[i-1] + 1;
```

Fix: Eliminate inter-iteration dependencies

```
for (i=1; i<n; i++) b[i] = b[0] + i;
```

```
i1 = 0; i2 = 0;
for (i=0; i<n; i++) {
      i1++; a[i1] = f(i1);
      i2 += i; b[i2] = g(i2);
}
```

Fix: Eliminate relative iteration count dependencies

```
for (i=0; i<n; i++) {
      a[i+1] = f(i+1);
      b[(i*i+i)/2] = g((i*i+i)/2);
}
```

Loop Scheduling/Partitioning Clauses

- **`schedule(static[, chunk_size])`**
  - Round-robin distribution of chunks of size **`chunk_size`** to threads.
  - Omitting **`chunk_size`** leads to near-equal-size chunks (1 per thread)
- **`schedule(dynamic[, chunk_size])`**
  - Similar to **`static`**, (with **`chunk_size`** specified), but assignment to threads is dynamic, one-thread-at-a-time, as threads request work
  - Default **`chunk_size`** is 1 (differs from **`static`**)
- **`schedule(guided[, parm])`**
  - Like **`dynamic`**, except that **`chunk_size`** starts big and shrinks, to reduce cost of task assignment and improve load balance
  - For **`parm = 1`**:

    `chunk_size ∝ ⌈(# iterations left)/ numthreads⌉`
  - For **`parm`** > 1: Like first case, except that **`chunk_size >= parm`**, except for last block

Loop Scheduling/Partitioning Clauses (cont.)

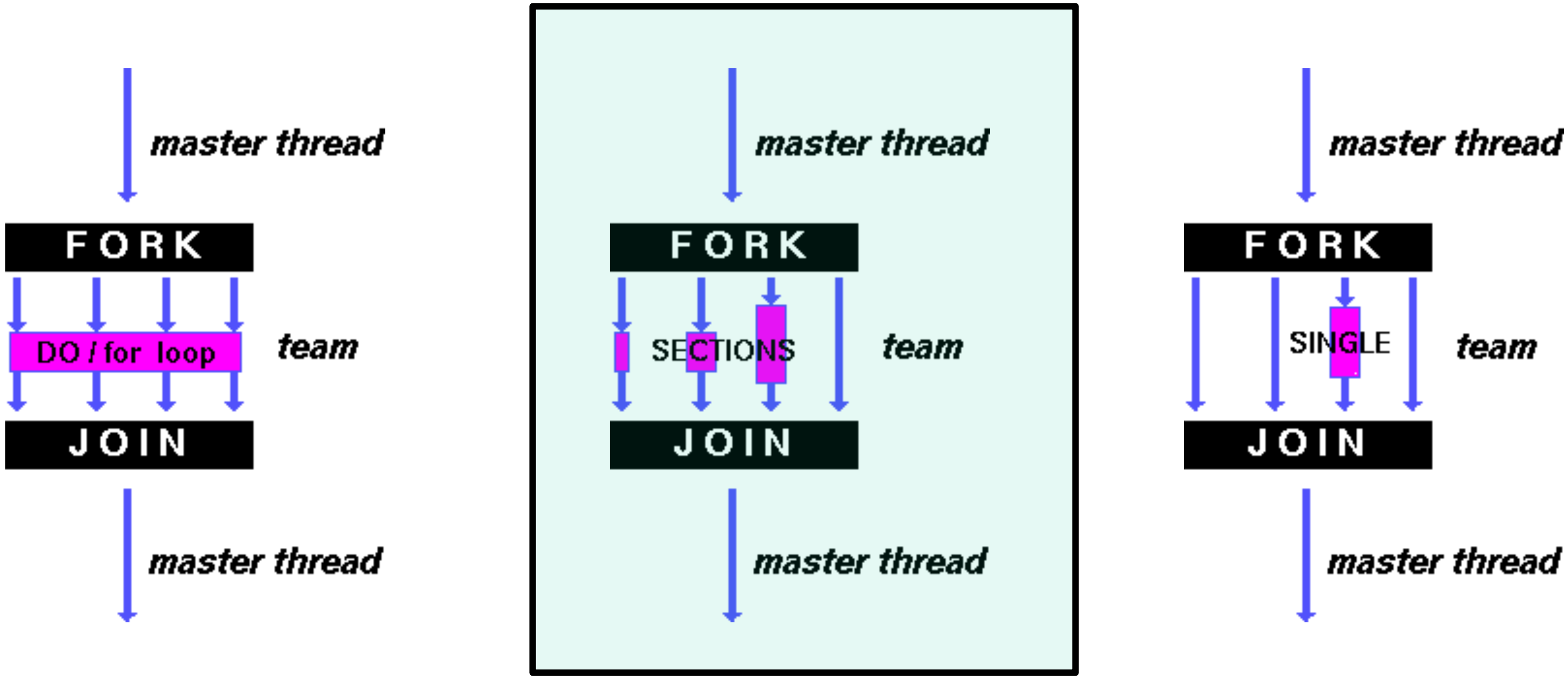
- `schedule(auto)`
  - Leaves scheduling up to compiler or runtime system
- `schedule(runtime)`
  - Uses `OMP_SCHEDULE` environment variable to specify one of the above methods. (Default is `auto` if `OMP_SCHEDULE` is not set.)
  - Settings might be:
    - `export OMP_SCHEDULE="guided"`
    - `export OMP_SCHEDULE="dynamic, 4"`

Example

```
#pragma omp parallel shared(a,b,c,nthreads,chunk)
                     default(none) private(i,tid) {
   tid = omp_get_thread_num();
   if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
   printf("Thread %d starting...\n",tid);

   #pragma omp for schedule(dynamic,chunk)
      for (i=0; i<N; i++) {
          c[i] = a[i] + b[i];
          printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
      }
}  // end of parallel region
```

Work-Sharing Constructs



Sections

The construct

```
#pragma omp sections
{
      #pragma omp section
            structured_block
      #pragma omp section
            structured_block
            .
            .
            .
}
```

causes the structured blocks to be assigned to threads in the team.

`#pragma omp sections` precedes the set of structured blocks.

`#pragma omp section` prefixes each structured block.

(The first `section` directive is optional. Note that the assignment of sections to threads is not predictable.)

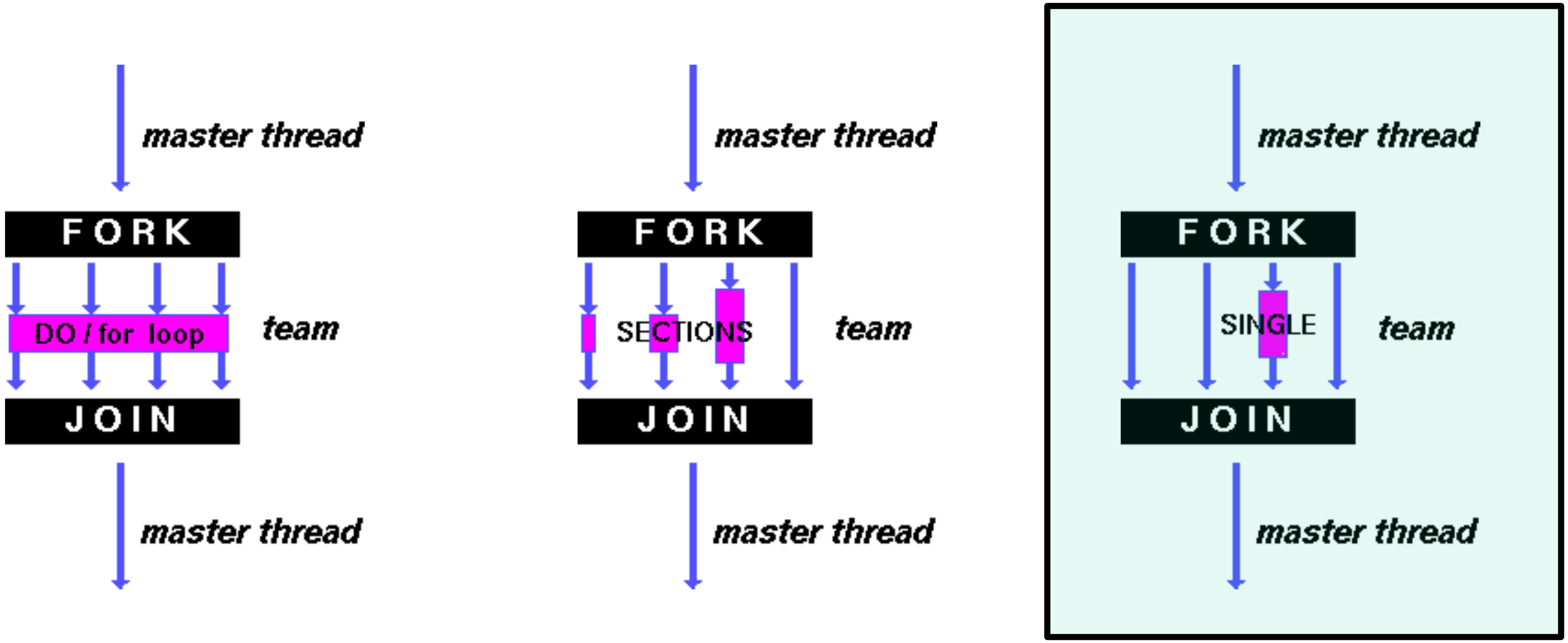
Example

```
#pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid) {
    tid = omp_get_thread_num();
    #pragma omp sections nowait  {
        #pragma omp section  {
            printf("Thread %d doing section 1\n",tid);
            for (i=0; i<N; i++) {
                c[i] = a[i] + b[i];
                printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
            }
        }
        #pragma omp section {
            printf("Thread %d doing section 2\n",tid);
            for (i=0; i<N; i++) {
              d[i] = a[i] * b[i];
              printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
            }
        }
    }  // end of sections
}  // end of parallel section
```

Some thread here

Some thread here

Work-Sharing Constructs



Single & Master Directives

The construct

```
#pragma omp single
   structured_block
```

causes the structured block to be executed by exactly one thread. Other threads wait at the implicit barrier following the structured block. Which thread executes the block is unpredictable.

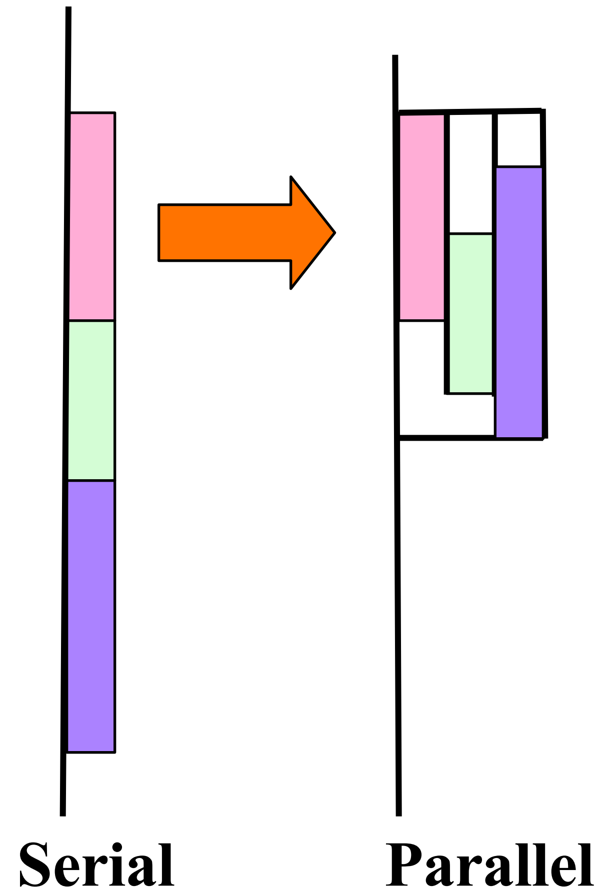
The directive

```
#pragma omp master
   structured_block
```

causes the structured block to be executed by the master thread. It is similar to the **`single`** directive, except there is no implicit barrier (either before or after). All the other threads ignore the **`master`** directive and go on.

Tasks

- Tasks are independent units of work
- Tasks are composed of:
  - Code to execute
  - A data environment
  - Internal control variables (ICV)
- Threads are assigned to perform the work of each task
- The runtime system will either:
  - Defer tasks for later execution
  - Execute the tasks immediately



Slide from OpenMP Tutorial at SC16

How Tasks Work

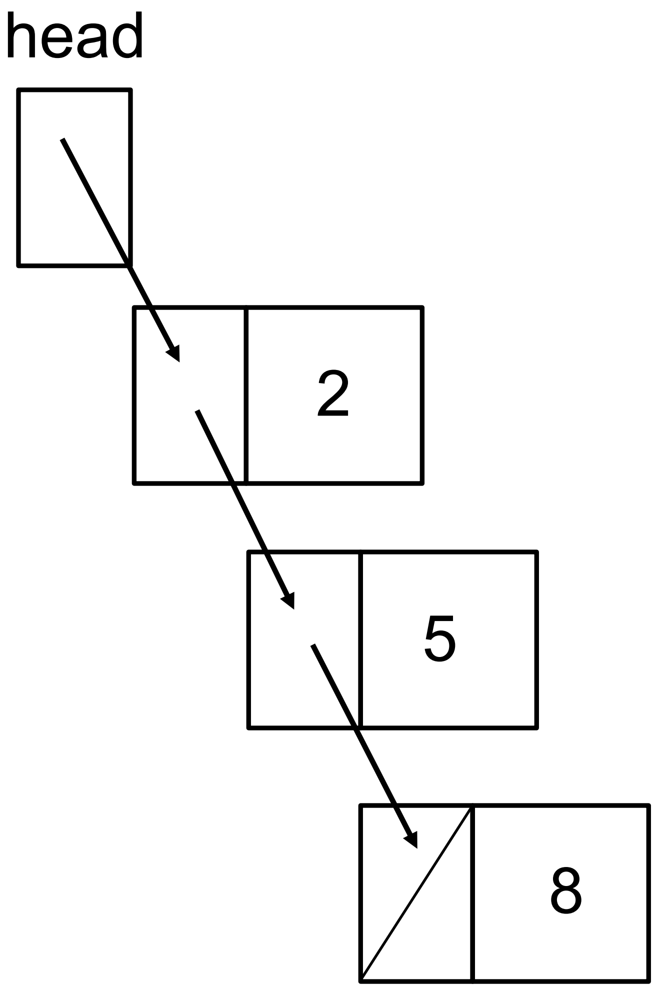
- The task construct defines a section of code

```
#pragma omp task
 {
    ...some code
 }
```

- Inside a parallel region, a thread encountering a task construct will package up the task for execution
- Some thread in the parallel region will execute the task at some point in the future
- Tasks may be nested: i.e., a task may itself generate tasks

Slide from OpenMP Tutorial at SC16

Example: Simple Linked List Traversal




```
struct node {
    struct node* next;
    int payload;
}
```

Keep in mind that there are 3 nodes in use.

Task Construct: Explicit Task View

- A team of threads is created at the omp parallel construct
- A single thread is chosen to execute the while loop – lets call this thread "L"
- Thread L operates the while loop, creates tasks, and fetches next pointers
- Each time L encounters the task construct it generates a new task
- Each task is eventually assigned to a thread that executes it
- All tasks will be complete at the barrier at the end of the single construct

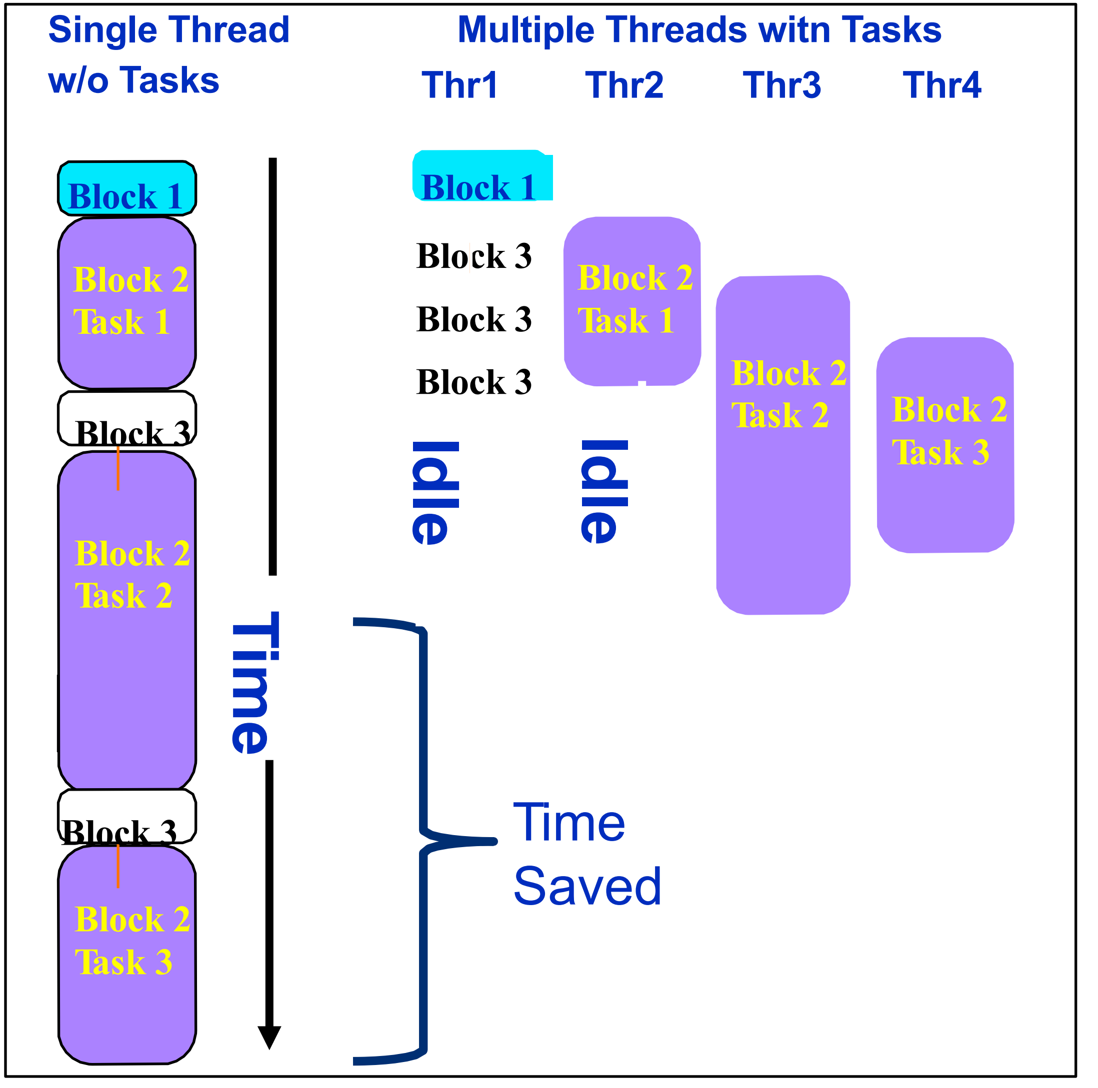
```
#pragma omp parallel
{
  #pragma omp single
  {  // block 1
     node * p = head;
     while (p) {  //block 2
     #pragma omp task firstprivate(p)
        myfunc(p);
     p = p->next;  //block 3
     }
  }
}
```

Slide from OpenMP Tutorial at SC16

Why Are Tasks Useful?

Have potential to parallelize irregular patterns and recursive function calls

```
#pragma omp parallel
{
  #pragma omp single
  {  // block 1
    node * p = head;
    while (p) {  //block 2
    #pragma omp task
      myfunc(p);
    p = p->next;  //block 3
    }
  }
}
```



Slide from OpenMP Tutorial at SC16

When Are Tasks Guaranteed to Complete?

- Tasks are guaranteed to be complete at thread barriers:

```
#pragma omp barrier
```

  – barrier applies to all threads; task completion applies to all tasks generated in the current parallel region by time of the barrier

- … or task barriers (inside of a task region)

```
#pragma omp taskwait
```

  – wait until all tasks generated in the current task have completed. Applies only to "child" tasks in the enclosing "task region," not "descendants". (Note: a "parallel region" contains an implicit task region.)

- … or by an implied barrier at the end of the structured block that created the tasks

Slide from OpenMP Tutorial at SC16

Task completion example

```
#pragma omp parallel
{
    for(int  i=0;i<N;i++){
        #pragma omp task
             foo();

    }
    #pragma omp single
    {for(int i=0;i<N;i++)
          #pragma omp task
             bar();

    }
}
```

Implicit task region here

N foo tasks created here by each thread

All foo tasks guaranteed to be completed at the end of the generating structured block.

N bar tasks created here

All bar tasks guaranteed to be completed here

Slide from OpenMP Tutorial at SC16

Data scoping with tasks

- The notions of shared and private variables can be confusing with respect to tasks
  - If a variable is shared on a task construct, the references to it inside the construct are to the original storage with that name at the point where the task was encountered
  - If a variable is private on a task construct, the references to it inside the construct are to new uninitialized storage that is created when the task is executed
  - If a variable is firstprivate on a task construct, the references to it inside the construct are to new storage that is created and initialized with the value of the existing storage of that name when the task is encountered

Slide from OpenMP Tutorial at SC16

Data scoping with tasks

- The behavior you want for tasks is usually **firstprivate**, because the task may not be executed until later (and variables may have gone out of scope)
  - Variables that are private when the task construct is encountered become **firstprivate** by default
- Variables that are shared in all constructs starting from the innermost enclosing parallel region are **shared** by default
- Use **default(none)** to help detect & avoid races!!!

Slide from OpenMP Tutorial at SC16

Data scoping with tasks: Fibonacci example

```
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;
  }
  int main()
  {int NN = 5000;
  #pragma omp parallel
  {
     #pragma omp single
       fib(NN);
  }
}
```

n is private (C is "call by value" so n is on the stack and therefore private)
n-1 and n-2 are firstprivate in the 2 tasks.

x is a shared variable
y is a shared variable

So this works!

Here's the implicit task region

Slide from OpenMP Tutorial at SC16

Reduction Clause

Used with `parallel`, `for`, and `sections` directives to combine parallel results (similar to `MPI_Reduce`).

Operation

Variable

```
sum = 0.
#pragma omp parallel for reduction(+:sum)
   for (k=0; k<100; k++) sum = sum + funct(k);
```

Private copy of `sum` will be created for each thread. All the private `sum` variables will be added to the master's `sum` at the end.

Avoids need for a critical section to do the final summation.

Built-in operations: +, -, *, &, |, ^, &&, or || and min/max. Custom operations may be defined, as well.

May use multiple `reduction` clauses in a single directive

Built-In OpenMP Reduction Operators in C/C++

Think of reduction operations as the following computation:

$$\textbf{\textit{Result}} = [\textbf{\textit{Initializer}}] \oplus a \oplus b \oplus c \ \ldots \oplus x \oplus y \oplus z$$

where $\oplus$ is a "combiner" corresponding to one of the permitted operations.

Valid Operators and Initialization Values		
Operation	**C/C++**	**Initializer**
Addition	`+`	0
Subtraction	`-`	0
Multiplication	`*`	1
Logical AND	`&&`	1 (true)
Logical OR	`||`	0 (false)
Bitwise AND	`&`	All bits on (1)
Bitwise OR	`|`	All bits off (0)
Bitwise XOR	`^`	All bits off (0)
Maximum	`max`	Most negative number
Minimum	`min`	Largest positive number

Array Sections in C/C++

- An array section designates a subset of the elements in an array.
- An array section is only allowed in clauses that explicitly allow it.
- To specify an array section in an OpenMP construct, array subscript expressions are extended with the following syntax:

  [lower-bound : length] or [lower-bound :] or [: length] or [:]

- The array section must be a subset of the original array.
- Array sections are allowed on multidimensional arrays.
- The lower-bound and length represent a set of integer values:

  { LB, LB+1, LB+2,... , LB+length-1 }

- The LB and length must evaluate to non-negative integers.
- If an array dimension is unknown, then length must be explicit.
- When length is absent, it defaults to the array dimension - LB
- When the lower-bound is absent it defaults to 0.

Array Section Examples

The following are examples of array sections:

```
a[0:6]
a[:6]
a[1:10]
a[1:]
b[10][:][:0]
c[1:10][42][0:6]
```

The first two examples are equivalent.

- If a is declared to be an eleven element array, the third and fourth examples are equivalent.
- The fifth example is a zero-length array section.
- The last example is not contiguous.

Synchronization Constructs

1. Barrier
2. Ordered
3. Critical
4. Atomic
5. Flush

Synchronization Constructs: Barrier

When a thread reaches the construct

```
#pragma omp barrier
```

it waits until all threads in the parallel region have reached the barrier and then they all proceed together. This implies that all explicit tasks created prior to the barrier are complete.

There are restrictions on the placement of barrier directive in a program. For example, either all or none of the threads must be able to reach the barrier.

Synchronization Constructs: Ordered

An `ordered` region executes in sequential order. It must occur within a parallel loop region. For example, you might use the following loop to force a particular summation order in a reduction loop:

```
#pragma omp parallel private(tmp)
#pragma omp for ordered reduction(+:countVal)
for (i=0;i<N;i++){
    tmp = foo(i);
    #pragma omp ordered
    countVal+= consume(tmp);
}
```

Note that the `ordered` keyword occurs twice:

1. As an `ordered` clause in the loop pragma;
2. As the `ordered` construct itself.

Synchronization Constructs: Critical

The **`critical`** construct will only allow one thread at a time to execute the associated **`structured_block`**.

```
#pragma omp critical [(name)]
   structured_block
```

When threads reach the **`critical`** construct they will wait until no other thread is executing the same critical section (one with the same **`name`**), and then one thread will proceed to execute the structured block. (Which one is unspecified: there is no fairness guarantee.) Eventually, all threads in the team will execute the structured block.

More About Critical Sections

`(name)` is optional. All unnamed critical sections map to one unspecified name (that is, they're all one big critical section---probably NOT what you want!)

Be very careful about nested critical sections. What happens here?

```
double f(double x) {
#pragma omp critical  (one)
  z = g(x); // z is a shared variable
    . . .
}
. . .
#pragma omp critical  (two)
    y = f(x);
```

Synchronization Constructs: Atomic

The atomic construct provides for atomic uses of variables (specifically, <u>individual memory locations)</u>:

```
#pragma omp atomic [read | write | update | capture]
     expression_statement
```

where **expression_statement** is one of the forms

```
v = x;                     // read only
x = expression;            // write only
x <binop>= expression;     // update or missing only
x++, ++x, x--, --x         // update or missing only
```

expression may not reference **x**, and **<binop>** may be one of:

```
+ - * / & ~ | << >>
```

This ensures that the storage location **x** is updated atomically. (Evaluation of **expression** **is not** atomic.)

Synchronization Constructs: Atomic (cont.)

The atomic construct using the capture clause provides for both atomic update and "capture" of the original or final variable value (depending on operation). The update of **`v`** in the following is not atomic.

```
#pragma omp atomic capture
     expression_statement
```

where **`expression_statement`** is one of the forms

```
v = x++;                         // or x-- (original)
v = ++x;                         // or --x (final)
v = x <binop>= expression;       // final
```

Same general rules as for other forms of atomic. May also take the form:

```
#pragma omp atomic capture
   {structured_block} // E.g.: {v = x; x = expression;}
```

in order to clarify which value of **`x`** is captured.

Atomic vs. Critical

From the OpenMP Standard:

"Atomic regions **do not** guarantee exclusive access with respect to any accesses outside of atomic regions to the same storage location **`x`**, even if those accesses occur during a critical or ordered region, or while an OpenMP lock is owned by the executing task, or during the execution of a reduction clause."

Atomic: Only guarantees atomic update of a single location in memory.

Consider: What if the location is a pointer?

Critical: Guarantees exclusive execution of a block of code. Variables updated in the block may also be modified elsewhere!

Example: Atomic vs. Critical

Atomic allows atomic updates of individual array entries, whereas a critical section protects the entire array. Assume there is only one update of a[ ] in the program:

```
#pragma omp atomic
   a[index[i]] += b;
```

This ensures that the referenced location is updated atomically, but it does not constrain other uses of the variables by other threads. In particular, multiple threads may be atomically and simultaneously updating different entries of the array.

```
#pragma omp critical
   a[index[i]] += b;
```

This causes all updates to be serialized, even if the different threads are updating different locations in the array.

Synchronization Constructs: Flush

The construct

```
#pragma omp flush [(variable_list)]
```

creates a synchronization point that allows a thread to have a "consistent" view of listed variables (or all the variables in its view of memory, if there is no list). This makes the caches consistent with real memory.

All current read/write operations on variables are completed, but no new memory operations in code after the `flush` are started until the `flush` completes. Caches are invalidated, so later loads come from memory.

Only applies to the thread executing `flush`, not to all threads in team.

`flush` occurs automatically at entry and exit of `parallel` and `critical` constructs, and at the exit of `for`, `sections`, and `single` constructs (except when a `nowait` clause is used). So flush is rarely needed, and it can obviously hurt performance.

Synchronization Constructs: Locks

Simple Lock: May be locked once

`omp_lock_t mylock`

Nestable Locks: May be locked more than once by same thread (Why??)

`omp_nest_lock_t mynestlock`

Corresponding OMP functions:

`omp_init_lock()`	`omp_init_nest_lock()`
`omp_destroy_lock()`	`omp_destroy_nest_lock()`
`omp_set_lock()`	`omp_set_nest_lock()`
`omp_unset_lock()`	`omp_unset_nest_lock()`
`omp_test_lock()`	`omp_test_nest_lock()`

Notes:
1. Locks are initialized to "unset" state. Must initialize before use.
2. Erroneous to try to relock a simple lock. Behavior is unspecified.
3. Tests of simple locks return 1 (if successful) or 0 otherwise.
4. Tests of nestable locks return new nest count (if successful) or 0.

Additional OpenMP Information

Best reference site is http://www.openmp.org

Specification Document & Reference Cards:

http://openmp.org/wp/openmp-specifications/