Disclaimer

- I am not an OpenMP expert

- But I've learned most of OpenMP
  - And have borrowed some slides from the experts
- We'll cover the basics
  - More information available on-line
- Anything I don't yet know the answer to…
  - … we can look it up and find it out

- Hopefully today's lecture is where "bottom-up" pays off
  - Hopefully the OpenMP constructs won't seem like magic

Acknowledgments

- Includes slides from:
  - "Shared Memory Control Parallelism: OpenMP"
    - Clay Breshears (Intel)
    - Presented at UIUC's UPCRC 2009 Summer School
    - With permission, **includes some modifications by me**
  - "A Hands-on Introduction to OpenMP" **Blue background slides**
    - Tim Mattson (Intel) & Larry Meadows (Intel)
    - http://openmp.org/mp-documents/omp-hands-on-SC08.pdf
- Other sources and references:
  - "An Overview of OpenMP"
    - Ruud van der Pas (Sun Microsystem)
    - http://openmp.org/mp-documents/ntu-vanderpas.pdf
  - LLNL OpenMP: https://computing.llnl.gov/tutorials/openMP/

Teaser: Easy Loop-Level Parallelism

```
#include <omp.h>
void compute_one(int num_particles, int* location,
                 int *weight, int *radius, int *answer) {
  #pragma omp parallel for
  for (int i = 0; i < num_particles; i++) {
    for (int j = 0; j < num_particles; j++) {
      if (distance(location[i], location[j]) < radius[i]) {
        answer[i] += weight[j];
      }
    }
  }
}
```

- Compiler-based parallelism with OpenMP (`gcc -fopenmp`)
  - Runtime system detects number of cores, runs loop in parallel!
  - Variables declared **inside** of loop: “**private**”; **outside** of loop: “**shared**”
  - Limitation: loops with known iteration count
  - Defaults to static partitioning, want dynamic?

    `#pragma omp parallel for schedule (dynamic, 10)`

OpenMP Intro

What is OpenMP

- Set of “compiler directives” and runtime library
  - Bindings for C/C++ and Fortran
  - Standard/portable, implemented by many compilers
- Developed by scientific computing compiler developers
  - Observation: if only the programmer could tell us what is parallel
  - Rather than doing “automatic parallelization”
  - Targets known-iteration parallel loops (“for” loops in C/C++)
- Originated in the mid-1990s
  - Motivated by development of “scalable shared memory” machines
  - Uses “shared memory” rather than “message passing”
- Uses a lightweight fork/join model of computation

OpenMP Overview:
How do threads interact?

- OpenMP is a multi-threading, shared address model.
  - Threads communicate by sharing variables.
- Unintended sharing of data causes race conditions:
  - race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization to protect data conflicts.
- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization.

OpenMP “Hello World”

```
#include <omp.h>
int main()
{
  #pragma omp parallel
  {
    printf("Hello world, thread %d of %d\n",
           omp_get_thread_num(),
           omp_get_num_threads());
  }
}
```

OpenMP include file

Parallel region with default number of threads

Runtime library functions

End of parallel region

- Example output on a four-core machine:

```
 Hello world, thread 0 of 4
Hello world, thread 2 of 4
Hello world, thread 1 of 4
Hello world, thread 3 of 4
```

Parallel Region & Structured Blocks (C/C++)

OpenMP constructs apply to "statements" or "structured blocks"

Structured block: a block with one point of entry at the top and one point of exit at the bottom

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
more: res[id] = do_big_job (id);
  if (conv (res[id]) goto more;
}
printf ("All done\n");
```

```
if (go_now()) goto more;
#pragma omp parallel
{
  int id = omp_get_thread_num();
more:  res[id] = do_big_job(id);
  if (conv (res[id]) goto done;
  goto more;
}
done: if (!really_done()) goto more;
```

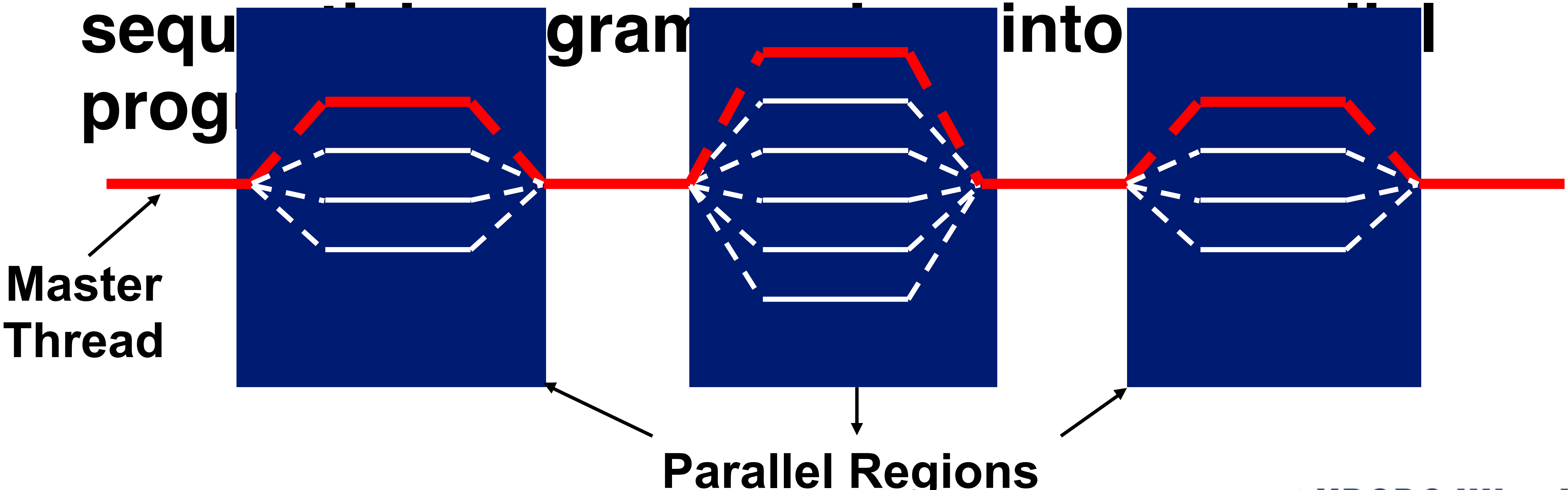
A structured block | **Not a structured block**

UPCRC Illinois
2009 Summer School on Multicore Programming

OpenMP Programming Model

Fork-Join Parallelism:

- **Master** spawns a **team of “threads”** as needed
- **Lightweight (keeps threads alive, avoids thread creation)**
- **Parallelism is added incrementally: that is, the sequ gram into l progr**



UPCRC Illinois
2009 Summer School on Multicore Programming

OpenMP Fork/Join Parallelism

```
int main()
{
  #pragma omp parallel
  {
    printf("Before ");
  }
  printf("\nSequential\n");
  #pragma omp parallel
  {
    printf("After ");
  }
  printf("\nSequential\n");
}
```

End of parallel region (implicit barrier)

Not in parallel region

Next parallel region

- Example output on a four-core machine:

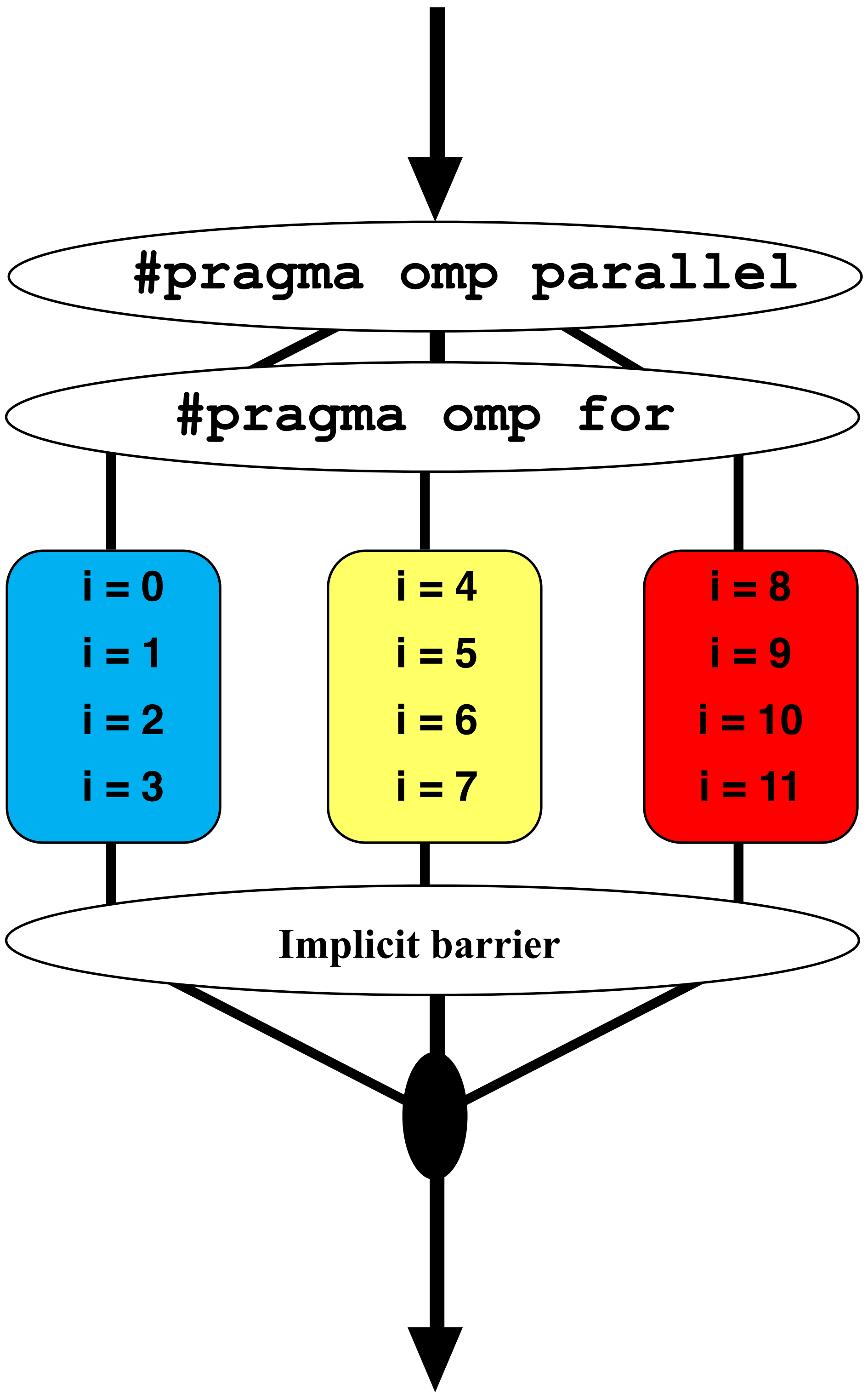
```
Before Before Before Before
Sequential
After After After After
Sequential
```

OpenMP “For” Construct

OpenMP "for" Construct

```
// assume N = 12
#pragma omp parallel
#pragma omp for
   for(i = 0; i < N; i++)
      c[i] = a[i] + b[i];
```

- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct (implicit barrier)



Combining constructs

- These two code segments are equivalent

```
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
     res[i] = huge();
    }
}
```

```
#pragma omp parallel for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
```

UPCRC Illinois
2009 Summer School on Multicore Programming

The schedule clause

The schedule clause affects how loop iterations are mapped onto threads

`schedule(static[,chunk])`

- Blocks of iterations of size "chunk" to threads
- Round robin distribution
- Low overhead, may cause load imbalance

`schedule(dynamic[,chunk])`

- Threads grab "chunk" iterations
- When done with iterations, thread requests next set
- Higher threading overhead, can reduce load imbalance

`schedule(guided[,chunk])`

- Dynamic schedule starting with large block
- Size of the blocks shrink; no smaller than "chunk"

Schedule Clause Example

```
#pragma omp parallel for schedule (static, 8)
    for(int i = start; i <= end; i += 2)
    {
       if (TestForPrime(i)) gPrimesFound++;
    }
```

Iterations are divided into chunks of 8

- If start = 3, then first chunk is **i**={3,5,7,9,11,13,15,17}

OpenMP Data Scoping

Data Scoping – What's shared

- OpenMP uses a shared-memory programming model
- **Shared variable** - a *variable* whose name provides access to a the <u>same</u> block of storage for each task region
  - Shared clause can be used to make items explicitly shared
  - Global variables are shared among tasks
    - C/C++: File scope variables, namespace scope variables, static variables, variables with const-qualified type having no mutable member are shared, static variables which are declared in a scope inside the construct are shared.

Data Scoping – What's private

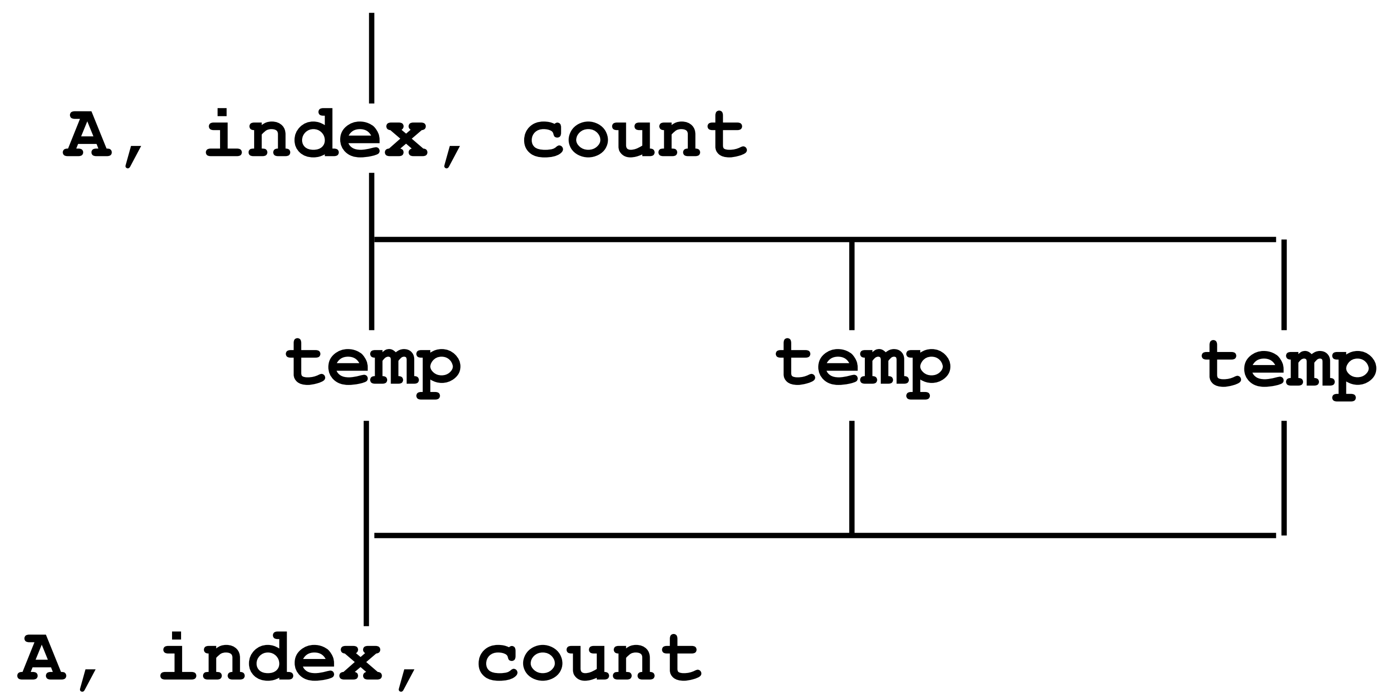
- But, not everything is shared...
- Examples of implicitly determined private variables:
  - Stack (local) variables in functions called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE
  - Loop iteration variables are private

A Data Environment Example

```
float A[10];
main ()
{
   int index[10];
   #pragma omp parallel
   {
      work(index);
   }
    printf (“%d\n”, index[1]);
}
```

```
extern float A[10];
void work (int *index)
{
   float temp[10];
   static int count;
   <...>
}
```

A, ***index***, **and** ***count*** **are shared by all threads, but** ***temp*** **is local to each thread**



The Private Clause

- Reproduces the variable for each task
  - Variables are un-initialized; C++ object is default constructed
  - Any value external to the parallel region is undefined

```
void* work(float* c, int N) {
  float x, y; int i;
 #pragma omp parallel for private(x,y)
     for(i=0; i<N; i++) {
        x = a[i]; y = b[i];
        c[i] = x + y;
     }
}
```

  - Alternative

```
v



        float x          float y
           x    y


}
```

UPCRC Illinois
2009 Summer School on Multicore Programming

Data Sharing: Private Clause When is the original variable valid?

- **The original variable's value is unspecified in OpenMP 2.5.**
- **In OpenMP 3.0, if it is referenced outside of the construct**
  - **Implementations may reference the original variable or a copy ….. A dangerous programming practice!**

```
int tmp;
void danger() {
      tmp = 0;
#pragma omp parallel private(tmp)
      work();
     printf("%d\n", tmp);
}
```

tmp has unspecified value

```
extern int tmp;
void work() {
      tmp = 5;
}
```

unspecified which copy of tmp

Data Sharing: Firstprivate Clause

- **Firstprivate is a special case of private.**
  - **Initializes each private copy with the corresponding value from the master thread.**

```
void useless() {
    int tmp = 0;
#pragma omp for firstprivate(tmp)
    for (int j = 0; j < 1000; ++j)
      tmp += j;
    printf("%d\n", tmp);
 }
```

Each thread gets its own tmp with an initial value of 0

tmp: 0 in 3.0, unspecified in 2.5

Data sharing: Lastprivate Clause

- **Lastprivate passes the value of a private from the last iteration to a global variable.**

```
void closer() {
    int tmp = 0;
#pragma omp parallel for firstprivate(tmp) \
   lastprivate(tmp)
   for (int j = 0; j < 1000; ++j)
      tmp += j;
   printf(“%d\n”, tmp);
}
```

Each thread gets its own tmp with an initial value of 0

tmp is defined as its value at the “last sequential” iteration (i.e., for j=999)

Data Sharing: Default Clause

- Note that the default storage attribute is **DEFAULT(SHARED)** (so no need to use it)
  - Exception: **#pragma omp task**
- To change default: **DEFAULT(PRIVATE)**
  - *each* variable in the construct is made private as if specified in a private clause
  - mostly saves typing
- **DEFAULT(NONE)**: *no* default for variables in static extent. Must list storage attribute for each variable in static extent. Good programming practice!

Only the Fortran API supports default(private).

C/C++ only has default(shared) or default(none).

Data sharing: Threadprivate

- Makes global data private to a thread
  - Fortran: COMMON blocks
  - C: File scope and static variables, static class members
- Different from making them PRIVATE
  - with PRIVATE global variables are masked.
  - THREADPRIVATE preserves global scope within each thread
- Threadprivate variables can be initialized using COPYIN or at time of definition (using language-defined initialization capabilities).

OpenMP Synchronization

Example: Dot Product

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for
   for(int i=0; i<N; i++) {
      sum += a[i] * b[i];
   }
  return sum;
}
```

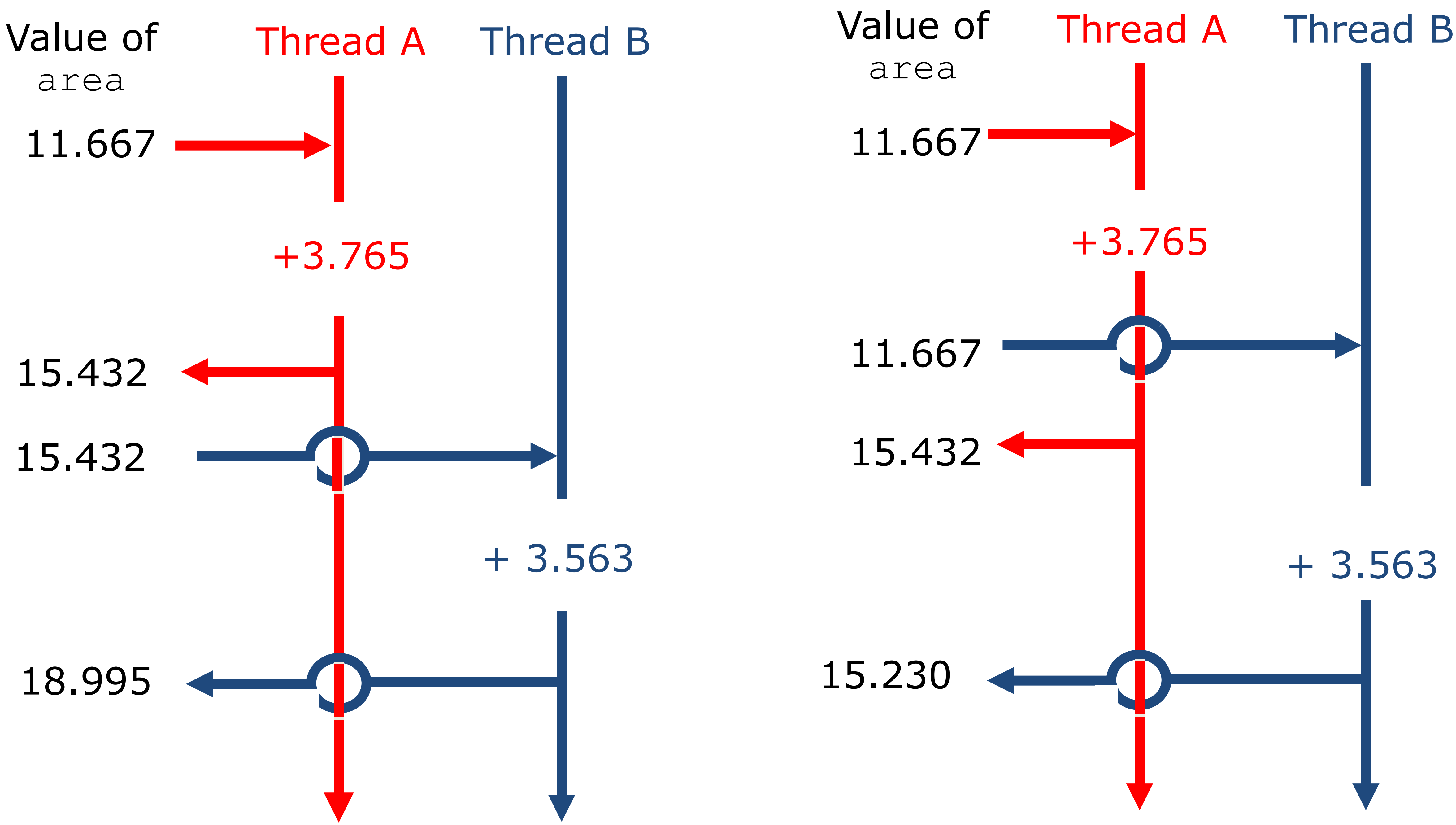
What is Wrong?

Race Condition

- A *race condition* is nondeterministic behavior caused by the times at which two or more threads access a shared variable
- For example, suppose both Thread A and Thread B are executing the statement

```
area += 4.0 / (1.0 + x*x);
```

Two Timings



Order of thread execution causes nondeterminant behavior in a data race

Protect Shared Data

- Must protect access to shared, modifiable data

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
  #pragma omp parallel for
  for(int i=0; i<N; i++) {
  #pragma omp critical
    sum += a[i] * b[i];
  }
  return sum;
}
```

- Note: fixes problem, but provides no parallelism in this example

OpenMP Critical Construct

```
#pragma omp critical [(lock_name)]
```

- Defines a critical region on a structured block (code locking)
- All critical sections with **same name** (or **"null" name**)

Threads wait their turn – only one at a time calls `consum()` thereby protecting "res" from race conditions

Naming the critical construct `"res_lock"` is optional

```
float res;
#pragma omp parallel
{ float B;
#pragma omp for
  for(int i=0; i<niters; i++){
    B = big_job(i);
#pragma omp critical (res_lock)
    consum (B, res);
  }
}
```

Good Practice – Name all critical sections

Atomic Construct

- Special case of a critical section
- Applies only to simple update of memory location

```
#pragma omp parallel for
   for (i = 0; i < n; i++) {
      #pragma omp atomic
        x[index[i]] += work1(i);
      y[i] += work2(i);
   }
```

OpenMP Reductions

Reduction

- How do we handle this case?

```
double  ave=0.0, A[MAX];    int i;
for (i=0;i< MAX; i++) {
    ave + = A[i];
}
ave = ave/MAX;
```

- We are combining values into a single accumulation variable (ave) … there is a true dependence between loop iterations that can't be trivially removed
- This is a very common situation … it is called a "reduction".
- Support for reduction operations is included in most parallel programming environments.

OpenMP Reduction Clause

```
reduction (op : list)
```

- The variables in "*list*" must be shared in the enclosing parallel region
- Inside parallel or work-sharing construct:
  - A PRIVATE copy of each list variable is created and initialized depending on the "op"
  - These copies are updated locally by threads
  - At end of construct, local copies are combined through "op" into a single value and combined with the value in the original SHARED variable

Reduction Example

```
#pragma omp parallel for reduction(+:sum)
   for(i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
```

- Local copy of *sum* for each thread
- All local copies of *sum* added together and stored in “global” variable

C/C++ Reduction Operations

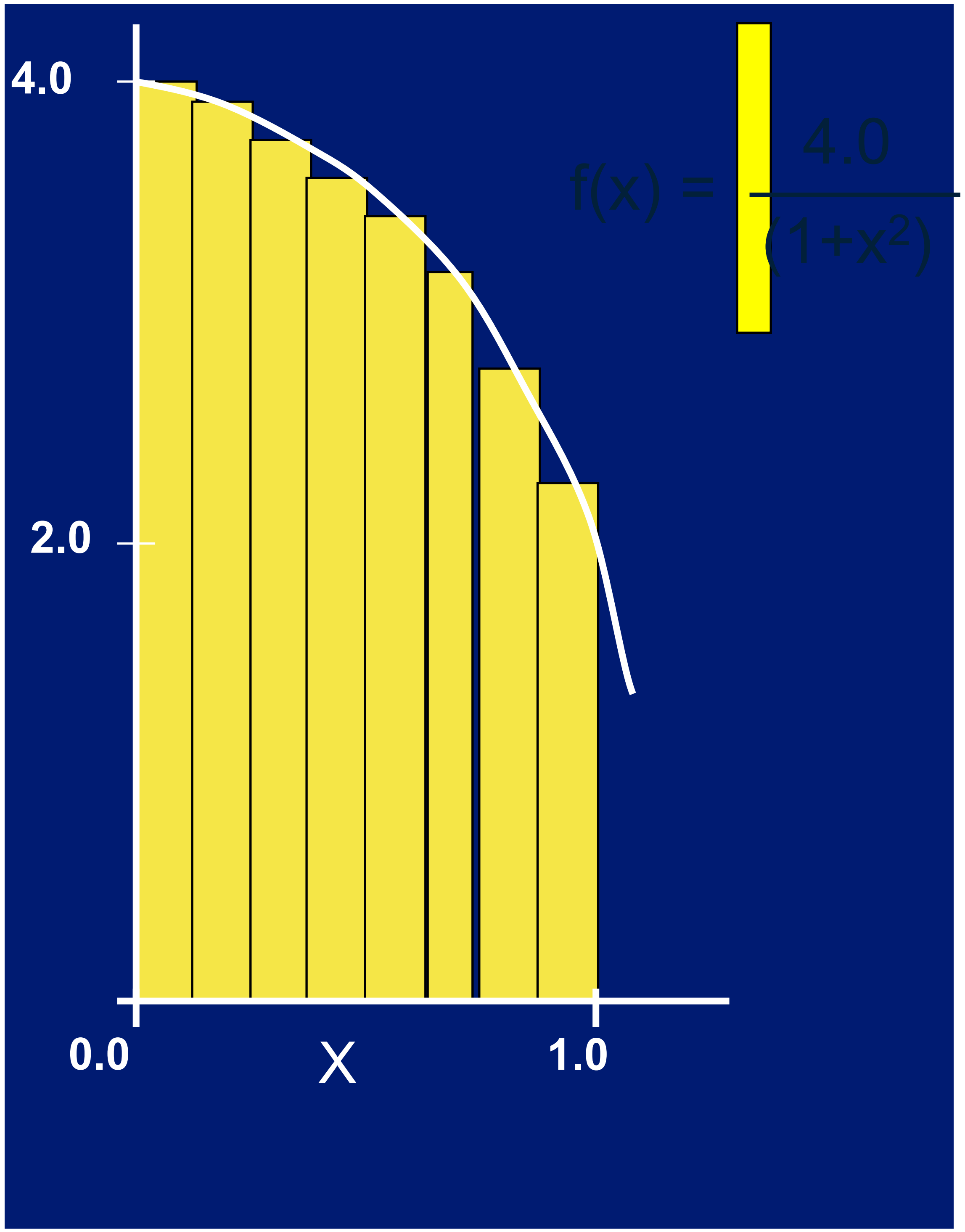
- A range of associative operands can be used with reduction
- Initial values are the ones that make sense mathematically

Operand	Initial Value
+	0
*	1
-	0
^	0

Operand	Initial Value
&	~0
|	0
&&	1
||	0

UPCRC Illinois
2009 Summer School on Multicore Programming

Numerical Integration Example



$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

```
static long num_steps=100000;
double step, pi;

void main()
{   int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

Numerical Integration Example

```
static long num_steps=100000;
double step, pi;

void main()
{   int i;
    double x;
    double sum = 0.0;
    step = 1.0/(double) num_steps;

    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

- What variables can be shared?
- What variables need to be private?
- What variables should be set up for reductions?

Numerical Integration with OpenMP Reduction

```
static long num_steps=100000;
double step, pi;

void main()
{   int i;
    double x;
    double sum = 0.0;
    step = 1.0/(double) num_steps;
#pragma omp parallel for private(x) reduction(+:sum)
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf(“Pi = %f\n”,pi);
}
```

UPCRC Illinois
2009 Summer School on Multicore Programming

Numerical Integration with OpenMP Reduction

```
static long num_steps=100000;
double step, pi;

void main()
{  int i;

   double sum = 0.0;
   step = 1.0/(double) num_steps;
#pragma omp parallel for reduction(+:sum)
   for (i=0; i< num_steps; i++){
      double x = (i+0.5)*step;
      sum = sum + 4.0/(1.0 + x*x);
   }
   pi = step * sum;
   printf("Pi = %f\n",pi);
}
```

Synchronization: ordered

- The ordered region executes in the sequential order.

```
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:res)

      for (I=0;I<N;I++){
            tmp = NEAT_STUFF(I);
#pragma ordered
            res += consum(tmp);
      }
```

OpenMP Control Constructs

Recall: OpenMP Fork/Join Parallelism

```
int main()
{
  #pragma omp parallel
  {
    printf("Before ");
  }
  printf("\nSequential\n");
  #pragma omp parallel
  {
    printf("After ");
  }
  printf("\nSequential\n");
}
```

End of parallel region (implicit barrier)

Not in parallel region

Next parallel region

- Example output on a four-core machine:

```
Before Before Before Before
Sequential
After After After After
Sequential
```

OpenMP Explicit "Barrier" Directive

```
int main()
{
  #pragma omp parallel
  {
    printf("Before ");
  }

  printf("\nSequential\n");
  #pragma omp parallel
  {
    printf("After ");
  }
  printf("\nSequential\n");
}
```

```
int main()
{
  #pragma omp parallel
  {
    printf("Before ");
    #pragma omp barrier
    if (omp_get_thread_num() == 0)
      printf("\nSequential\n");
    #pragma omp barrier

    printf("After ");
  }
  printf("\nSequential\n");
}
```

- Barrier directive
  - Waits until all threads arrive before any thread continues
  - Implicit at end of any "#pragma omp parallel" region

OpenMP “Master” Directive

```
int main()
{
  #pragma omp parallel
  {
    printf("Before ");
    #pragma omp barrier
    if (omp_get_thread_num() == 0)
    {
      printf("\nSequential\n");
    }
    #pragma omp barrier
    printf("After ");
  }
  printf("\nSequential\n");
}
```

```
int main()
{
  #pragma omp parallel
  {
    printf("Before ");
    #pragma omp barrier
    #pragma omp master
    {
      printf("\nSequential\n");
    }
    #pragma omp barrier
    printf("After ");
  }
  printf("\nSequential\n");
}
```

- Master directive
  - No implicit barriers (at either start or end)

OpenMP "Single" Directive

```
int main()
{
  #pragma omp parallel
  {
    printf("Before ");
    #pragma omp barrier
    #pragma omp master
    {
      printf("\nSequential\n");
    }
    #pragma omp barrier
    printf("After ");
  }
  printf("\nSequential\n");
}
```

```
int main()
{
  #pragma omp parallel
  {
    printf("Before ");
    #pragma omp barrier
    #pragma omp single
    {
      printf("\nSequential\n");
    }
    /* Implicit barrier */
    printf("After ");
  }
  printf("\nSequential\n");
}
```

- Single directive
  - Executed by first thread to reach (perhaps not the master)
  - Implicit barrier at end, but **not** at start

Implicit Barriers

- Several OpenMP constructs have implicit barriers
  - Parallel – necessary barrier – cannot be removed
  - for
  - single
- Unnecessary barriers hurt performance and can be removed with the nowait clause
  - The nowait clause is applicable to:
    - For clause
    - Single clause

Nowait Clause

```
#pragma omp for nowait
   for(...)
     {...};
```

```
#pragma single nowait
{ [...] }
```

- Use when threads unnecessarily wait between independent computations

```
#pragma omp for schedule(dynamic,1) nowait
 for(int i=0; i<n; i++)
   a[i] = bigFunc1(i);

#pragma omp for schedule(dynamic,1)
 for(int j=0; j<m; j++)
   b[j] = bigFunc2(j);
```

OpenMP Runtime Library

Runtime Library routines

- **Runtime environment routines:**
  - **Modify/Check the number of threads**
    - omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()
  - **Are we in an active parallel region?**
    - omp_in_parallel()
  - **Do you want the system to dynamically vary the number of threads from one parallel construct to another?**
    - omp_set_dynamic, omp_get_dynamic();
  - **How many processors in the system?**
    - omp_num_procs()

…plus a few less commonly used routines.

Synchronization: Lock routines

- **Simple Lock routines:**
  - **A simple lock is available if it is unset.**
    - omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock(), omp_destroy_lock()

A lock implies a memory fence (a "flush") of all thread visible variables

- **Nested Locks**
  - **A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function**
    - omp_init_nest_lock(), omp_set_nest_lock(), omp_unset_nest_lock(), omp_test_nest_lock(), omp_destroy_nest_lock()

Note: a thread always accesses the most recent copy of the lock, so you don't need to use a flush on the lock variable.

Synchronization: Simple Locks

- **Protect resources with locks.**

```
omp_lock_t lck;
omp_init_lock(&lck);
#pragma omp parallel private (tmp, id)
{
    id = omp_get_thread_num();
    tmp = do_lots_of_work(id);
    omp_set_lock(&lck);
      printf("%d %d", id, tmp);
    omp_unset_lock(&lck);
}
omp_destroy_lock(&lck);
```

Wait here for your turn. (omp_set_lock)

Release the lock so the next thread gets a turn. (omp_unset_lock)

Free-up storage when done. (omp_destroy_lock)

Environment Variables

- **Set the default number of threads to use.**
  - OMP_NUM_THREADS *int_literal*
- **Control how "omp for schedule(RUNTIME)" loop iterations are scheduled.**
  - OMP_SCHEDULE "schedule[, chunk_size]"

… Plus several less commonly used environment variables.