Shared Memory Control Parallelism: OpenMP

Clay Breshears

Intel

UPCRC Illinois
2009 Summer School on Multicore Programming

Agenda

What is OpenMP?

Parallel regions

Worksharing Constructs

Data environment

Synchronization

What Is OpenMP?

- Portable, shared-memory threading API
  - Fortran, C, and C++
  - Multi-vendor support for both Linux and Windows
- Standardizes task & loop-level parallelism
- Suppo
- Combi ource
- Standa threading experi

http://www.openmp.org

Current spec is OpenMP 3.0

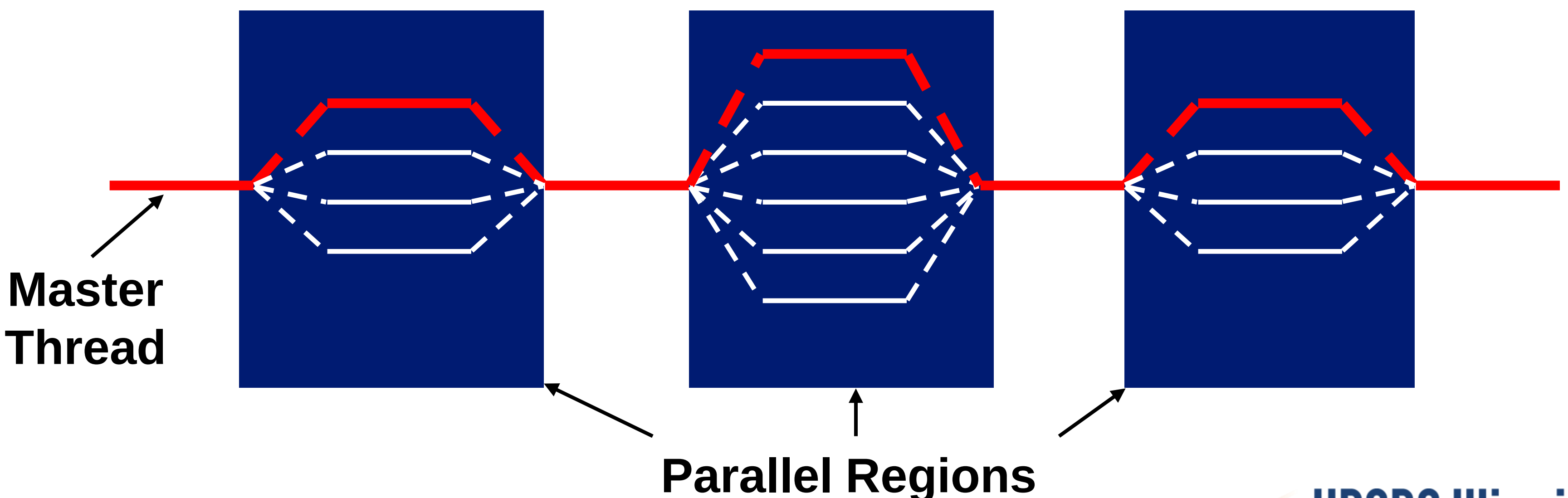
318 Pages

(combined C/C++ and Fortran)

Programming Model

Fork-Join Parallelism:

- **Master thread spawns a team of threads as needed**
- **Parallelism is added incrementally: that is, the sequential program evolves into a parallel program**



UPCRC Illinois
2009 Summer School on Multicore Programming

A Few Syntax Details to Get Started

- Most of the constructs in OpenMP are compiler directives or pragmas
  – For C and C++, the pragmas take the form:

```
#pragma omp construct [clause [clause]…]
```

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

OpenMP constructs apply to 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");
```

A structured block

```
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;
```

Not a structured block

Worksharing

- **Worksharing** is the general term used in OpenMP to describe distribution of work across threads.

- Three examples of worksharing in OpenMP are:
  - **`omp single`** construct
  - **`omp for`** construct
  - **`omp task`** construct

Automatically divides work among threads

Single Construct

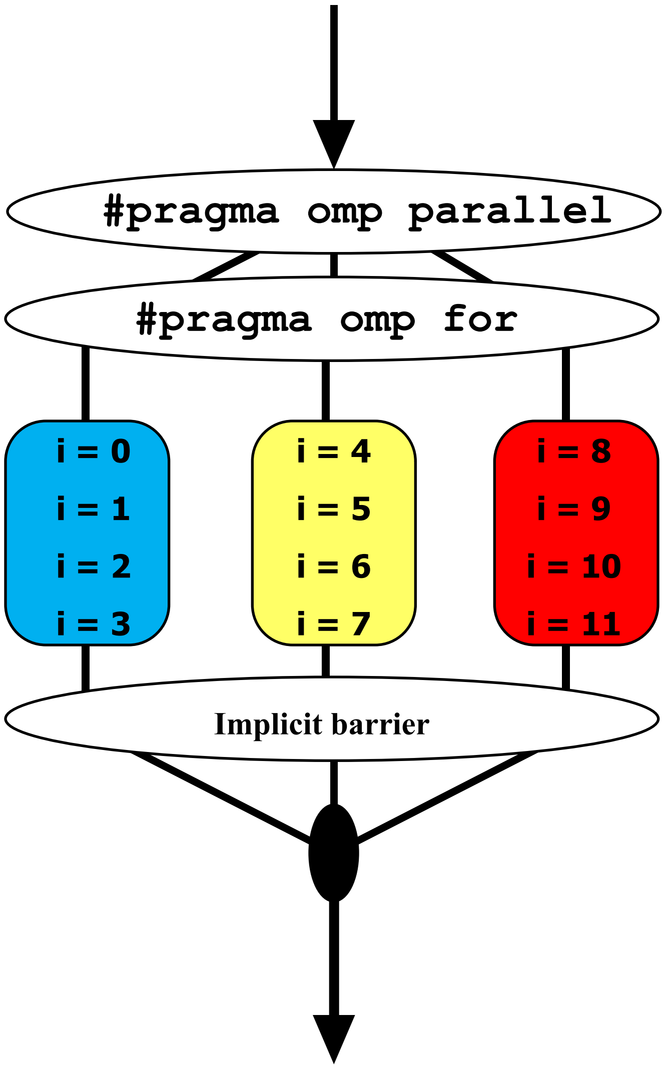
- Denotes block of code to be executed by only one thread
  - First thread to arrive is chosen
- Implicit barrier at end

```
#pragma omp parallel
{
   DoManyThings();
#pragma omp single
   {
     ExchangeBoundaries();
   }  // threads wait here for single
   DoManyMoreThings();
}
```

omp 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



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}

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.

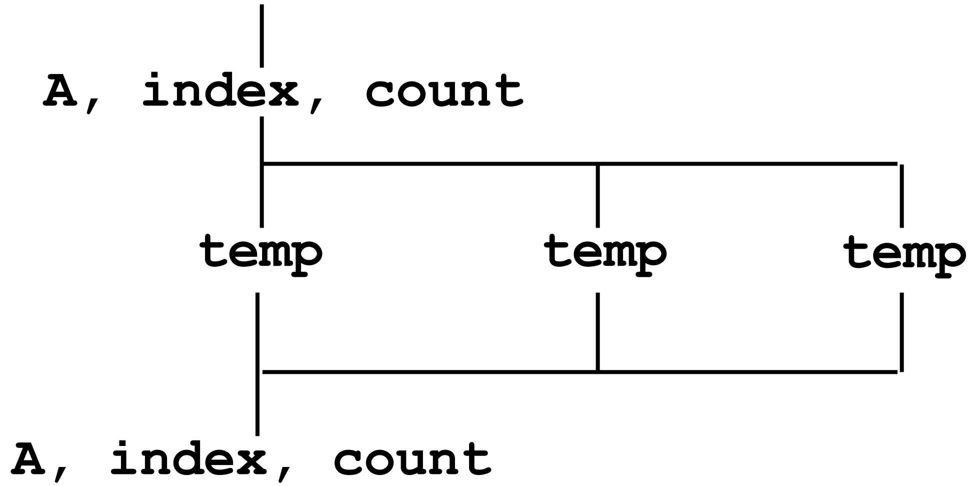
Data Scoping – What's private

- 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
    - Implicitly declared private variables within tasks will be treated as firstprivate

A Data Environment Example

```
float A[10];
main ()
{
  integer index[10];
  #pragma omp parallel
  {
      Work (index);
  }
    printf ("%d\n", index[1]);
}
```

```
extern float A[10];
void Work (int *index)
{
  float temp[10];
  static integer 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;
     }
}
```

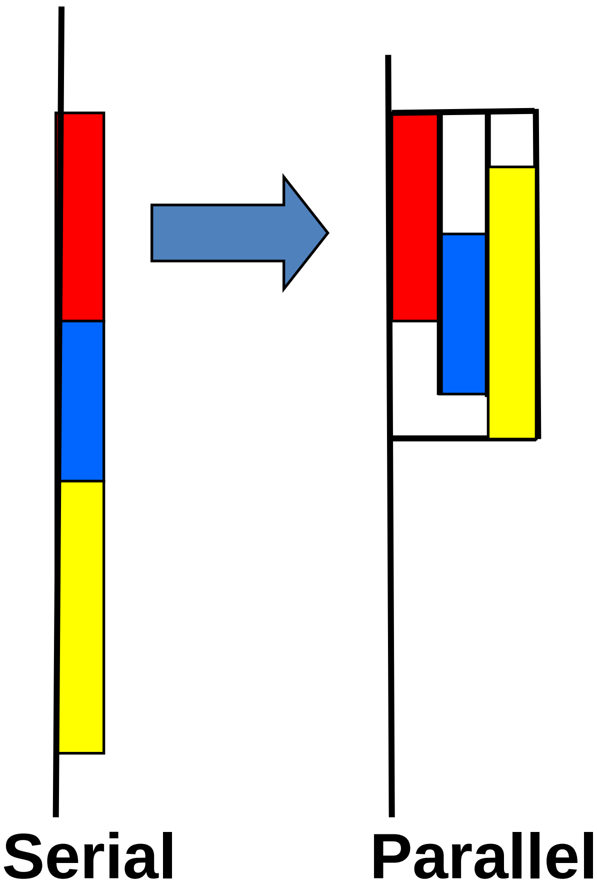
New Addition to OpenMP

- Tasks – Main change for OpenMP 3.0
- Allows parallelization of irregular problems
  - unbounded loops
  - recursive algorithms
  - producer/consumer

What are tasks?

- Tasks are independent units of work
- Threads are assigned to perform the work of each task
  - Tasks may be deferred
- Tasks may be executed immediately
- The runtime system decides which of the above

  - Tasks are composed of:
    - **code** to execute
    - **data** environment
    - **internal control variables** (ICV)



UPCRC Illinois
2009 Summer School on Multicore Programming

Task Construct – Explicit Task View

- A team of threads is created at the **`omp parallel`** construct
- A single thread, T0, is chosen to execute the **`while`** loop
- T0 operates the **`while`** loop, creates tasks, and fetches next pointers
- Each time T0 crosses the **`omp task`** construct it generates a new task
- Each task runs in its own thread
- All tasks complete at the barrier at the end of the parallel region's single construct

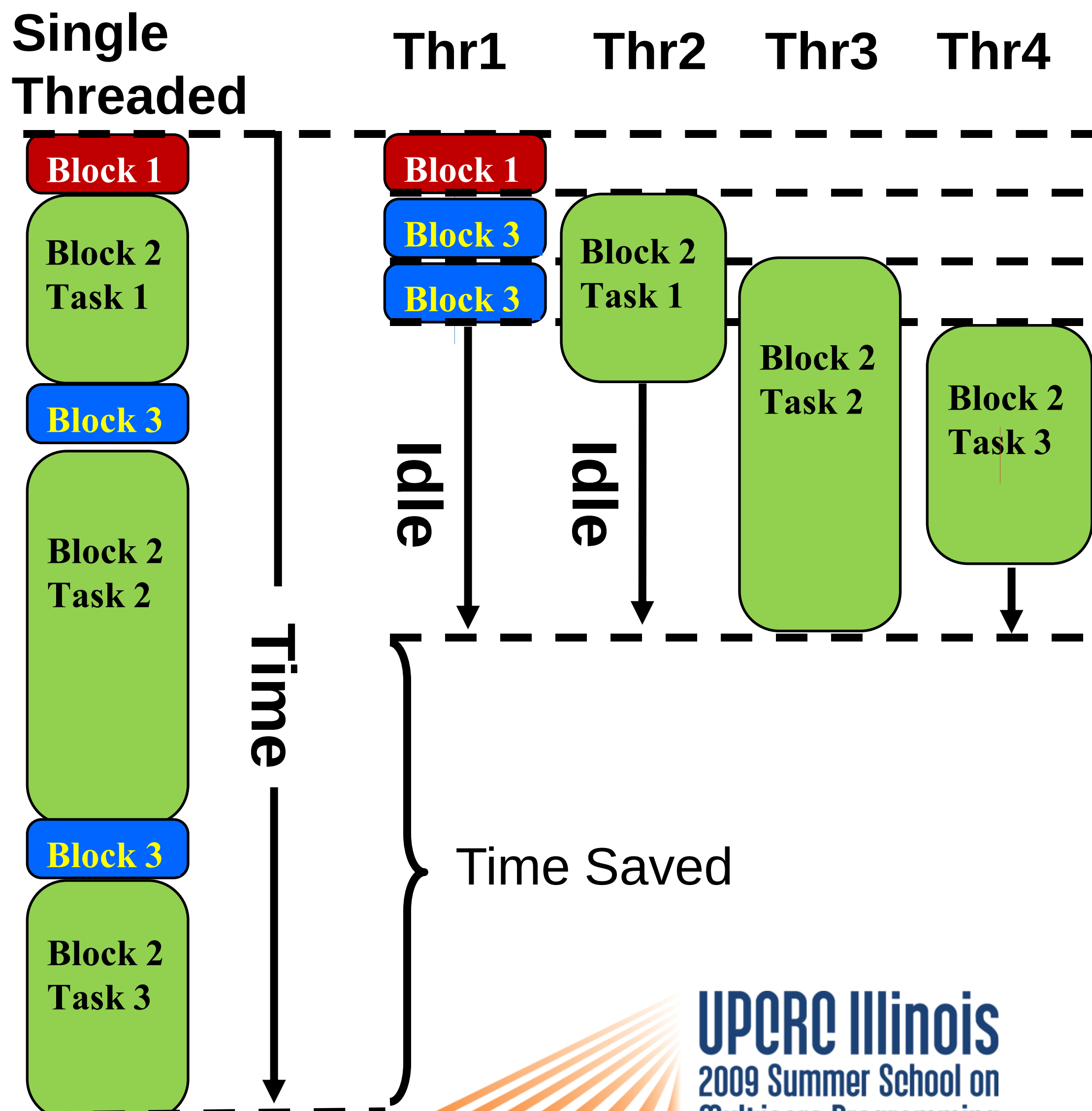
```
#pragma omp parallel
{
  #pragma omp single
  {  // block 1
    node * p = head;
    while (p) {  //block 2
    #pragma omp task
      process(p);
    p = p->next;  //block 3
    }
  }  // tasks done
}
```

UPCRC Illinois
2009 Summer School on Multicore Programming

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
      process(p);
    p = p->next;  //block 3
    }
  }
}
```



When are tasks gauranteed to be complete?

Tasks are gauranteed to be complete:

- At thread or task barriers
- At the directive: `#pragma omp barrier`
- At the directive: `#pragma omp taskwait`

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?

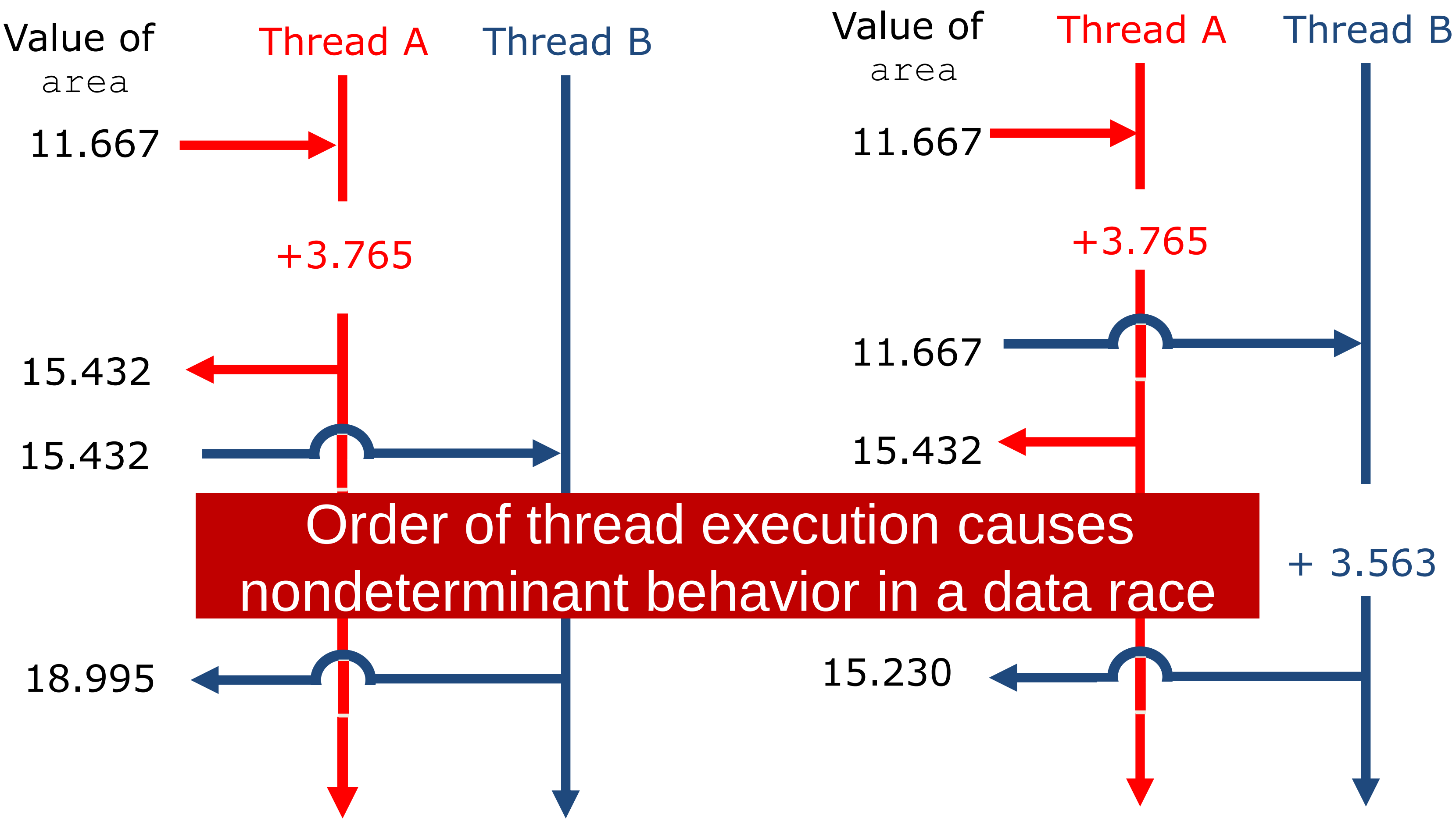
Race Condition

- 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);
```

UPCRC Illinois
2009 Summer School on Multicore Programming

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;
}
```

OpenMP Critical Construct

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

- Defines a critical region on a structured block

Threads wait their turn –at a time, only one 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

UPCRC Illinois
2009 Summer School on Multicore Programming

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

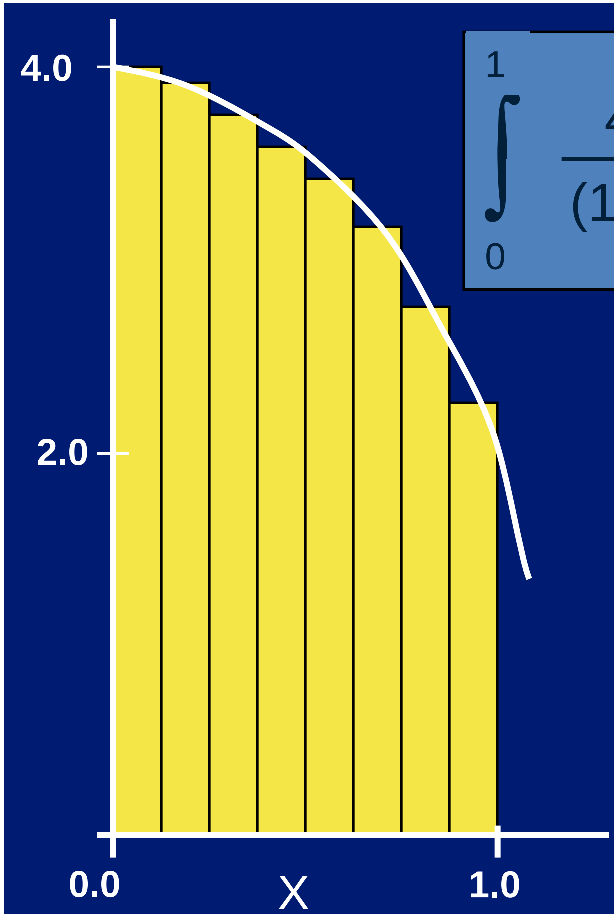
C/C++ Reduction Operations

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

<table>
<tr><th>Operand</th><th>Initial Value</th></tr>
<tr><td>+</td><td>0</td></tr>
<tr><td>*</td><td>1</td></tr>
<tr><td>-</td><td>0</td></tr>
<tr><td>^</td><td>0</td></tr>
</table>

<table>
<tr><th>Operand</th><th>Initial Value</th></tr>
<tr><td>&</td><td>~0</td></tr>
<tr><td>|</td><td>0</td></tr>
<tr><td>&&</td><td>1</td></tr>
<tr><td>||</td><td>0</td></tr>
</table>

Numerical Integration Example



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

```
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);
}
```

Computing Pi

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

void main()
{  int i;
   double x, 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);
}
```

can be

need

should

Atomic Construct

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

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

Master Construct

- Denotes block of code to be executed only by the master thread
- No implicit barrier at end

```
#pragma omp parallel
{
   DoManyThings();
#pragma omp master
   {                 // if not master skip to next stmt
     ExchangeBoundaries();
   }
   DoManyMoreThings();
}
```

UPCRC Illinois
2009 Summer School on Multicore Programming

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);
```

Barrier Construct

- Explicit barrier synchronization
- Each thread waits until all threads arrive

```
#pragma omp parallel shared (A, B, C)
{
        DoSomeWork(A,B);
        printf("Processed A into B\n");
#pragma omp barrier
        DoSomeWork(B,C);
        printf("Processed B into C\n");
}
```