

OpenMP

A parallel language standard that
support both data and functional
Parallelism on a shared memory
system

Threads Programming

- Use by system programmers more than application programmers
- Considered a low-level primitives package
- Application programmers need a higher level constructs/directives which hide the mechanics of manipulating threads
- Directive-based languages have recently been standardized in the form of OpenMP
- OpenMP provides support for concurrency, synchronization, and data handling while hiding the need for explicit thread management

OpenMP

- OpenMP: An application programming interface (API) for parallel programming on multiprocessors
 - Compiler directives
 - Library of support functions
- OpenMP works in conjunction with Fortran, C, or C++

What's OpenMP Good For?

- C + OpenMP sufficient to program multiprocessors
- C + MPI + OpenMP a good way to program multicomputers built out of multiprocessors
 - IBM RS/6000 SP
 - Fujitsu AP3000
 - Dell High Performance Computing Cluster

Parallelism

- Data Parallelism
 - Emphasize loop parallelism: elements in the loop will be processed in parallel
- Function Parallelism
 - OpenMP allows different threads to be assigned with different portions of code

Data Parallelism

- C programs often express data-parallel operations as `for` loops

```
for (i = first; i < size; i += prime)
    marked[i] = 1;
```

- OpenMP makes it easy to indicate when the iterations of a loop may execute in parallel
- Compiler takes care of generating code that forks/joins threads and allocates the iterations to threads



Pragma and Clause

- Pragma: a compiler directive in C or C++
- Stands for “pragmatic information”
- A way for the programmer to communicate with the compiler
- Syntax:

```
#pragma omp <rest of pragma> <Clause>
```

parallel for Pragma

- Format:

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

- Compiler must be able to verify the run-time system will have information it needs to schedule loop iterations

Execution Context

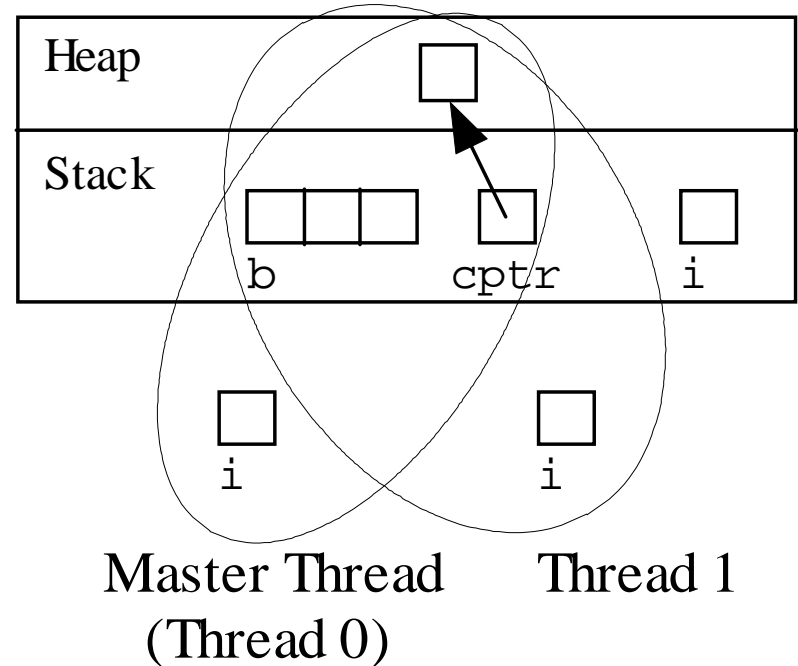
- Every thread has its own execution context
- Execution context: address space containing all of the variables a thread may access
- Contents of execution context:
 - static variables
 - dynamically allocated data structures in the heap
 - variables on the run-time stack
 - additional run-time stack for functions invoked by the thread

Shared and Private Variables

- Shared variable: has same address in execution context of every thread
- Private variable: has different address in execution context of every thread
- A thread cannot access the private variables of another thread

Shared and Private Variables

```
int main (int argc, char *argv[])  
{  
    int b[3];  
    char *cptr;  
    int i;  
  
    cptr = malloc(1);  
    #pragma omp parallel for  
    for (i = 0; i < 3; i++)  
        b[i] = i;
```



private Clause

- Clause: an optional, additional component to a pragma
- Private clause: directs compiler to make one or more variables private

```
private ( <variable list> )
```

Race Condition

- Consider this C program segment to compute π using the rectangle rule:

```
double area, pi, x;
int i, n;
...
area = 0.0;
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

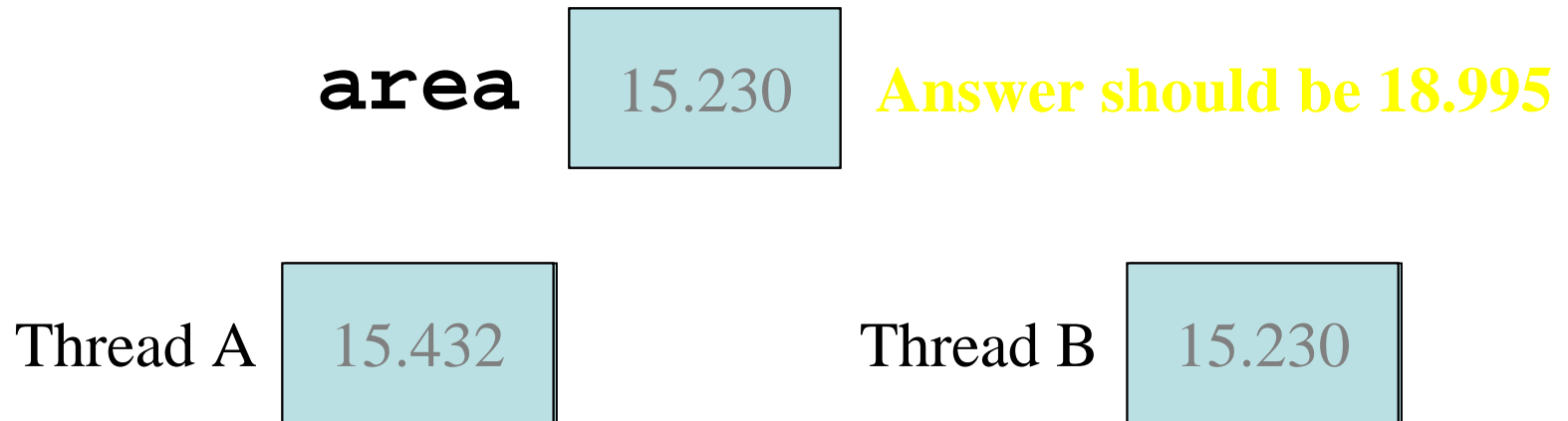
Race Condition (cont.)

- If we simply parallelize the loop...

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

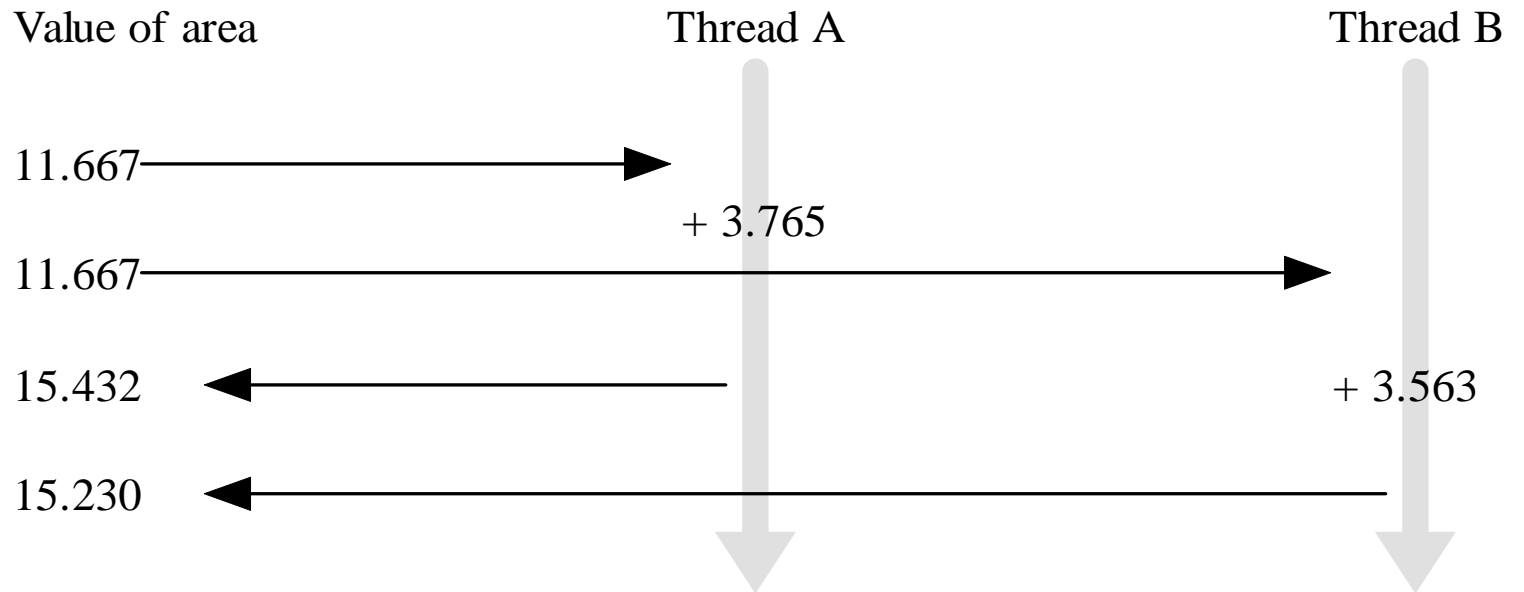
Race Condition (cont.)

- ... we set up a race condition in which one process may “race ahead” of another and not see its change to shared variable **area**



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

Race Condition Time Line



critical Pragma

- Critical section: a portion of code that only one thread at a time may execute
- We denote a critical section by putting the pragma

```
#pragma omp critical
```

in front of a block of C code

Correct, But Inefficient, Code

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
#pragma omp critical
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Source of Inefficiency

- Update to `area` inside a critical section
- Only one thread at a time may execute the statement; i.e., it is sequential code
- Time to execute statement significant part of loop
- By Amdahl's Law we know speedup will be severely constrained

Reductions

- Reductions are so common that OpenMP provides support for them
- May add reduction clause to `parallel for` pragma
- Specify reduction operation and reduction variable
- OpenMP takes care of storing partial results in private variables and combining partial results after the loop

reduction Clause

- The reduction clause has this syntax:
reduction (<op> :<variable>)
- Operators
 - +Sum
 - * Product
 - &Bitwise and
 - | Bitwise or
 - ^ Bitwise exclusive or
 - && Logical and
 - || Logical or

π -finding Code with Reduction Clause

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for \
    private(x) reduction(+:area)
for (i = 0; i < n; i++) {
    x = (i + 0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

parallel Pragma

- The `parallel` pragma precedes a block of code that should be executed by *all* of the threads
- Note: execution is replicated among all threads

```
#pragma omp parallel <Clause>  
    <statement_block>
```

Example Use of parallel Pragma

```
#pragma omp parallel private(i,j)
for (i = 0; i < m; i++) {
    low = a[i];
    high = b[i];
    if (low > high) {
        printf ("Exiting (%d)\n", i);
        break;
    }
    for (j = low; j < high; j++)
        c[j] = (c[j] - a[i])/b[i];
}
```

single Pragma

- Suppose we only want to see the output once
- The **single** pragma directs compiler that only a single thread should execute the block of code the pragma precedes
- Syntax:

```
#pragma omp single <Clause>  
    <statement_block>
```

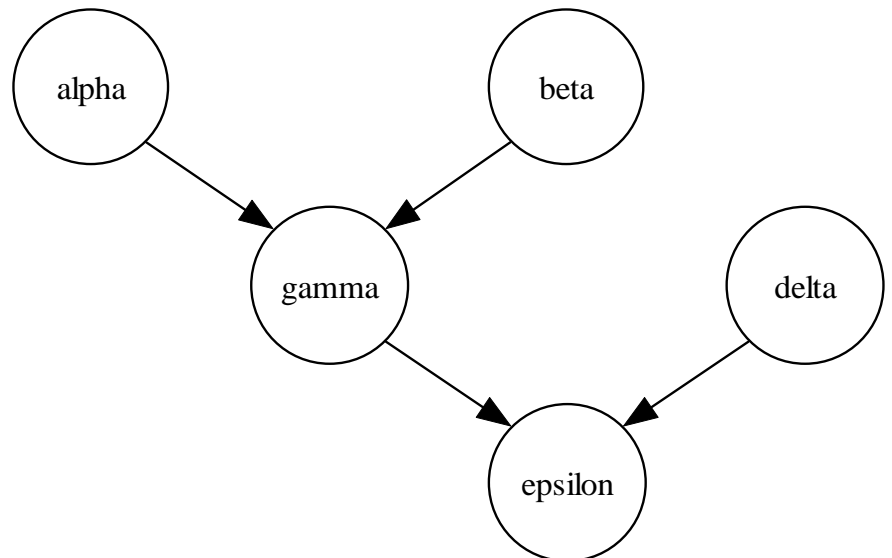
Example Use of single Pragma

```
#pragma omp parallel private(i,j)
for (i = 0; i < m; i++) {
    low = a[i];
    high = b[i];
    if (low > high) {
#pragma omp single
        printf ("Exiting (%d)\n", i);
        break;
    }
#pragma omp for
    for (j = low; j < high; j++)
        c[j] = (c[j] - a[i])/b[i];
}
```

Functional Parallelism Example

```
v = alpha();  
w = beta();  
x = gamma(v, w);  
y = delta();  
printf ("%6.2f\n", epsilon(x,y));
```

May execute alpha,
beta, and delta in
parallel



parallel sections Pragma

- Precedes a block of k blocks of code that may be executed concurrently by k threads
- Syntax:

```
#pragma omp parallel sections
```

section Pragma

- Precedes each block of code within the encompassing block preceded by the parallel sections pragma
- May be omitted for first parallel section after the parallel sections pragma
- Syntax:

```
#pragma omp section
```

Example of parallel sections

```
#pragma omp parallel sections
{
#pragma omp section /* Optional */
    v = alpha();
#pragma omp section
    w = beta();
#pragma omp section
    y = delta();
}
x = gamma(v, w);
printf ("%6.2f\n", epsilon(x,y));
```

Basic Parallel Functions

- `omp_get_thread_num` = `MPI_Comm_rank`
- `omp_get_num_threads` = `MPI_Comm_size`
- `omp_get_num_procs` = Return number of processors available in the system.
- `omp_set_num_threads` = Specify a number of threads needed in the computation.

Hello World

```
#include <omp.h>

int main (int argc, char *argv[]) {
    int nt, rank;

    #pragma omp parallel private(nt, rank)
    {
        rank = omp_get_thread_num();
        printf("Hello World from thread = %d\n", rank);

        if (rank == 0) {
            nt = omp_get_num_threads();
            printf("Number of threads = %d\n", nt);
        }
    }
}
```