PARALLEL AND DISTRIBUTED COMPUTING

An intoduction

Shared Memory Programming

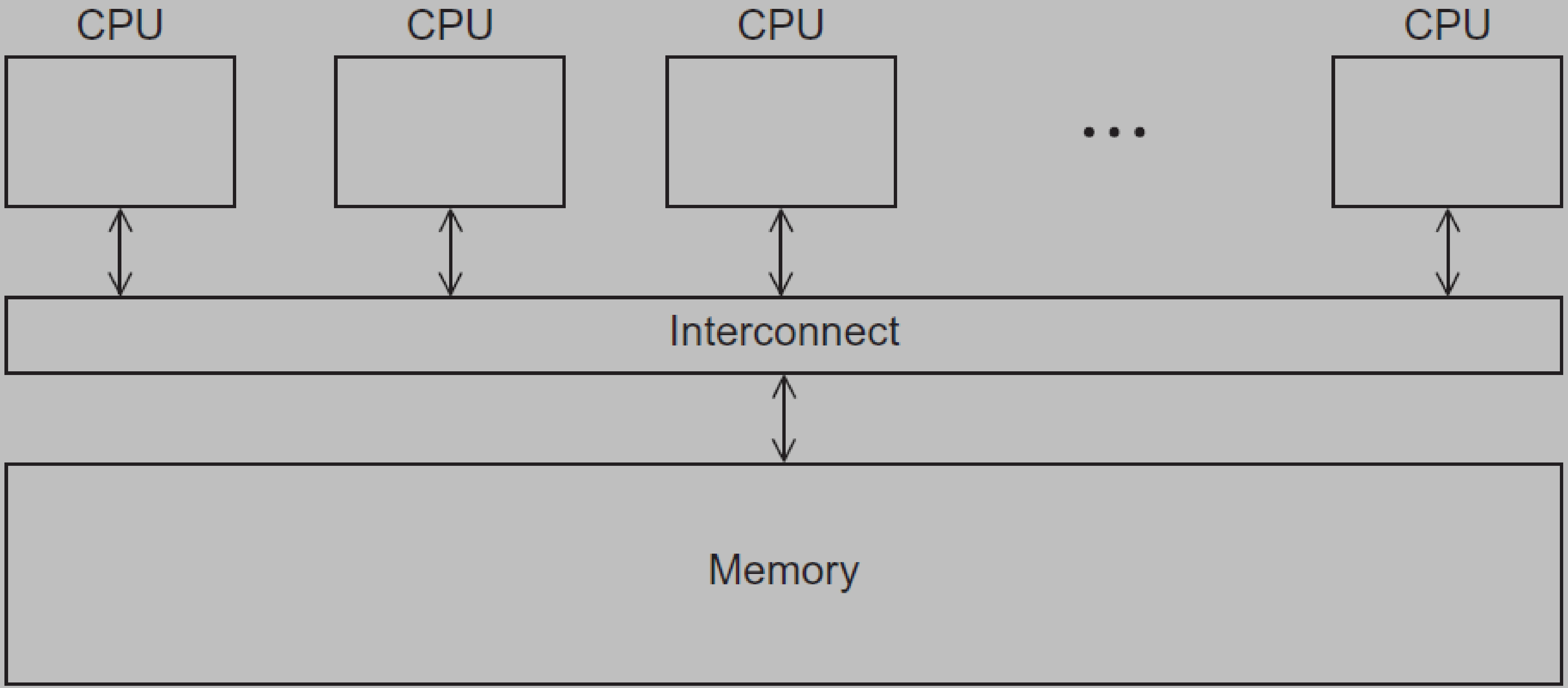
ROADMAP

- Writing programs that use OpenMP.
- Using OpenMP to parallelize many serial for loops with only small changes to the source code.
- Task parallelism.
- Explicit thread synchronization.
- Standard problems in shared-memory programming.

OpenMP

- An API for shared-memory parallel programming.
- MP = multiprocessing
- Designed for systems in which each thread or process can potentially have access to all available memory.
- System is viewed as a collection of cores or CPU's, all of which have access to main memory.

A Shared Memory System



PRAGMAS

- Special preprocessor instructions.

#pragma

- Typically added to a system to allow behaviors that aren't part of the basic C specification.

- Compilers that don't support the pragmas ignore them.

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

void Hello(void);  /* Thread function */

int main(int argc, char* argv[]) {
   /* Get number of threads from command line */
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n", my_rank, thread_count);

}  /* Hello */
```

```
gcc -g -Wall -fopenmp -o omp_hello omp_hello.c

./omp_hello 4
```

running with 4 threads

compiling

possible outcomes

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

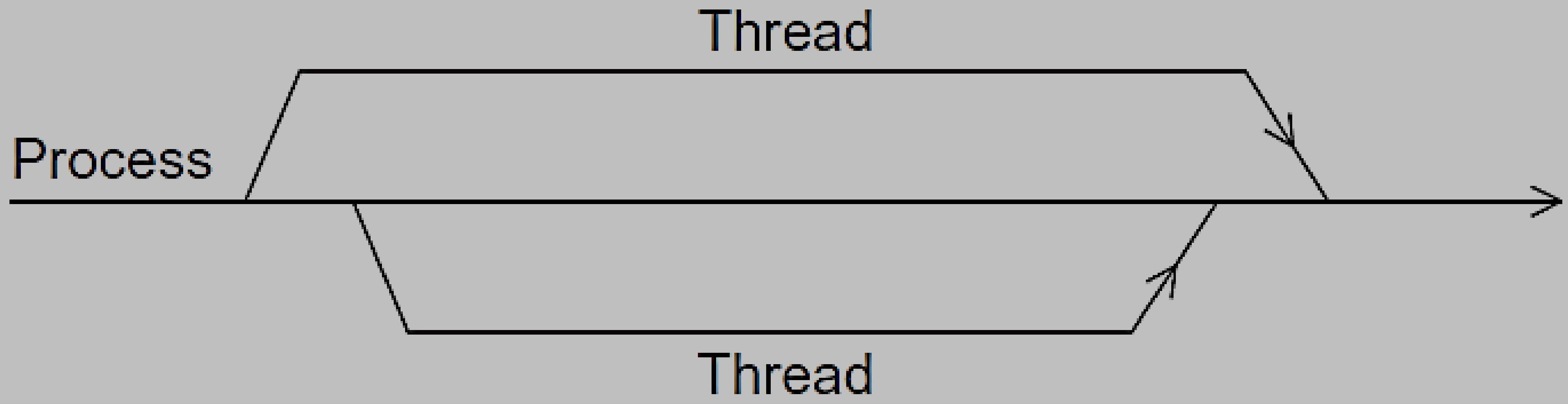
```
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

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

OpenMP Pragmas

- # pragma omp parallel
  - Most basic parallel directive.
  - The number of threads that run the following structured block of code is determined by the run-time system.

A Process Forking and Joining Two Threads



CLAUSE

- Text that modifies a directive.
- The num_threads clause can be added to a parallel directive.
- It allows the programmer to specify the number of threads that should execute the following block.

```
# pragma omp parallel num_threads ( thread_count )
```

Of note...

- There may be system-defined limitations on the number of threads that a program can start.
- The OpenMP standard doesn't guarantee that this will actually start thread_count threads.
- Most current systems can start hundreds or even thousands of threads.
- Unless we're trying to start a lot of threads, we will almost always get the desired number of threads.

SOME TERMINOLOGY

- In OpenMP parlance the collection of threads executing the ***parallel block***

- the original thread and the new threads:
  - is called a team,
  - the original thread is called the master,
  - and the additional threads are called slaves.

In case the compiler doesn't support OpenMP

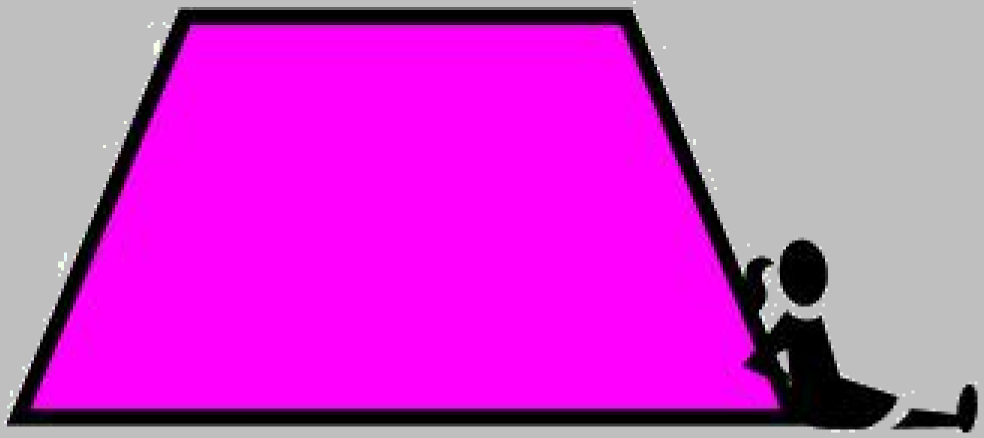
```
# include <omp.h>
```

$\searrow$

```
#ifdef _OPENMP
# include <omp.h>
#endif
```

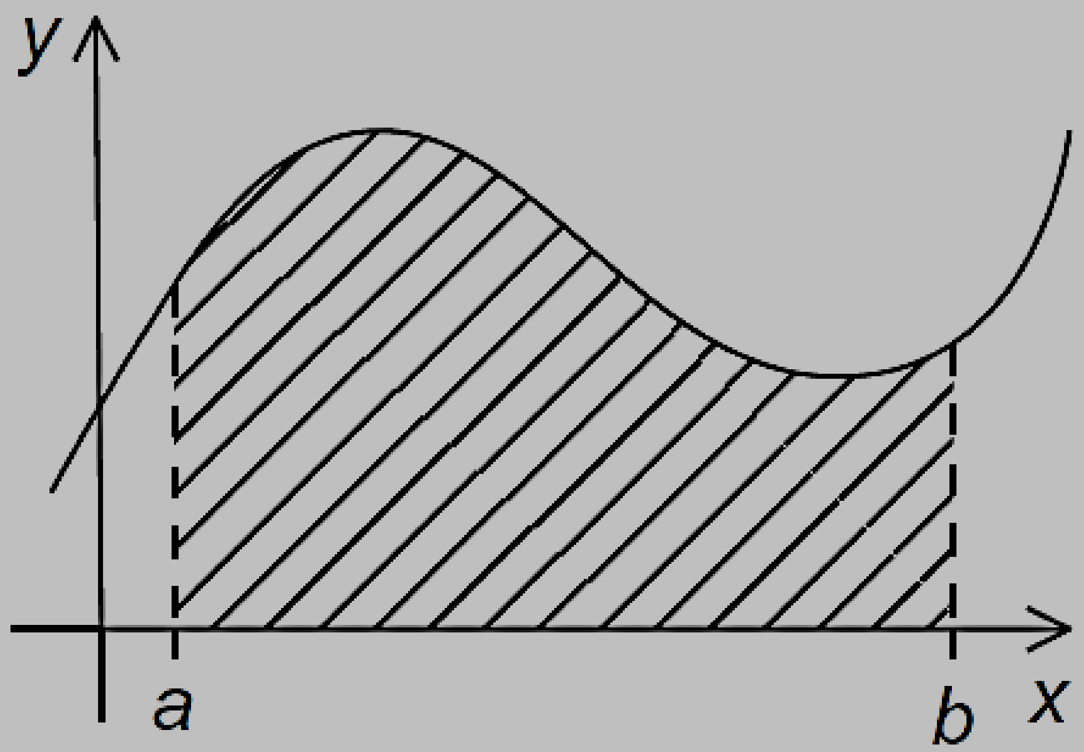
In case the compiler doesn't support OpenMP

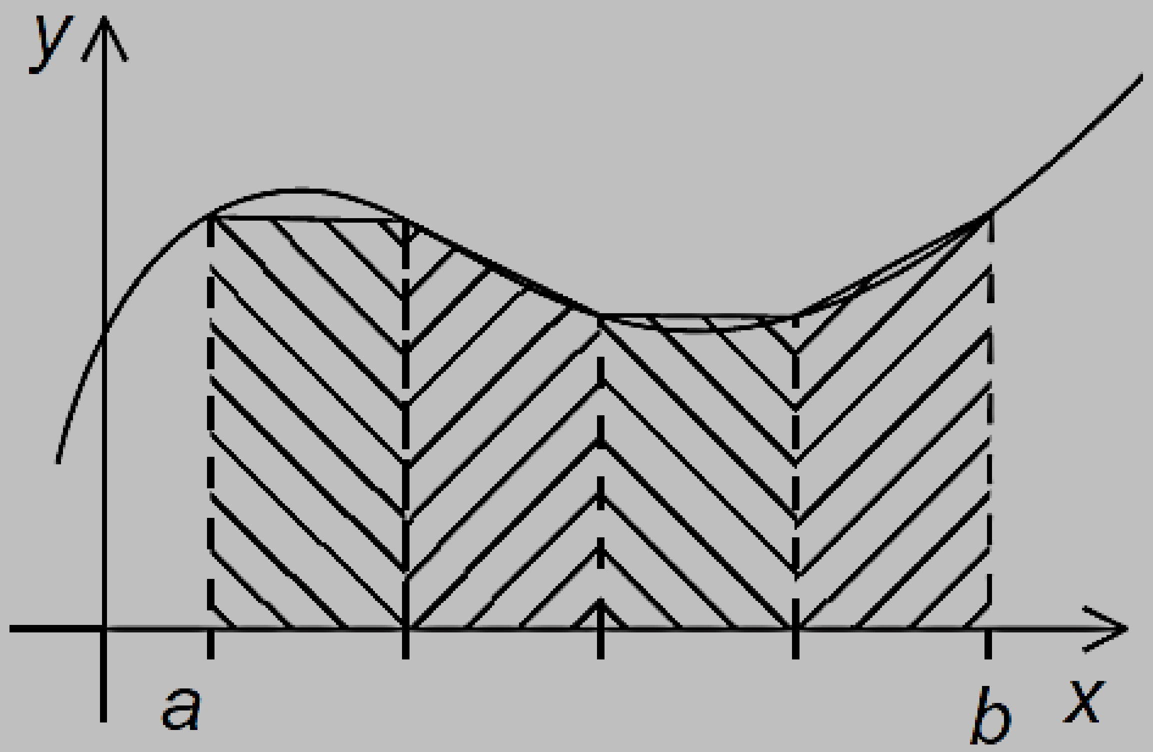
```
#ifdef _OPENMP
   int my_rank =
   omp_get_thread_num ( );
   int thread_count =
   omp_get_num_threads ( );
#else
   int my_rank = 0;
   int thread_count = 1;
#endif
```



THE TRAPEZOIDAL RULE

The Trapezoidal Rule





Serial Algorithm

```
/* Input: a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
    x_i = a + i*h;
    approx += f(x_i);
}
approx = h*approx;
```

A First OpenMP Version

We identified two types of tasks:

1. Area Calculation

- Computation of the areas of individual trapezoids, and

2. Summation Of Areas

- Adding the areas of trapezoids

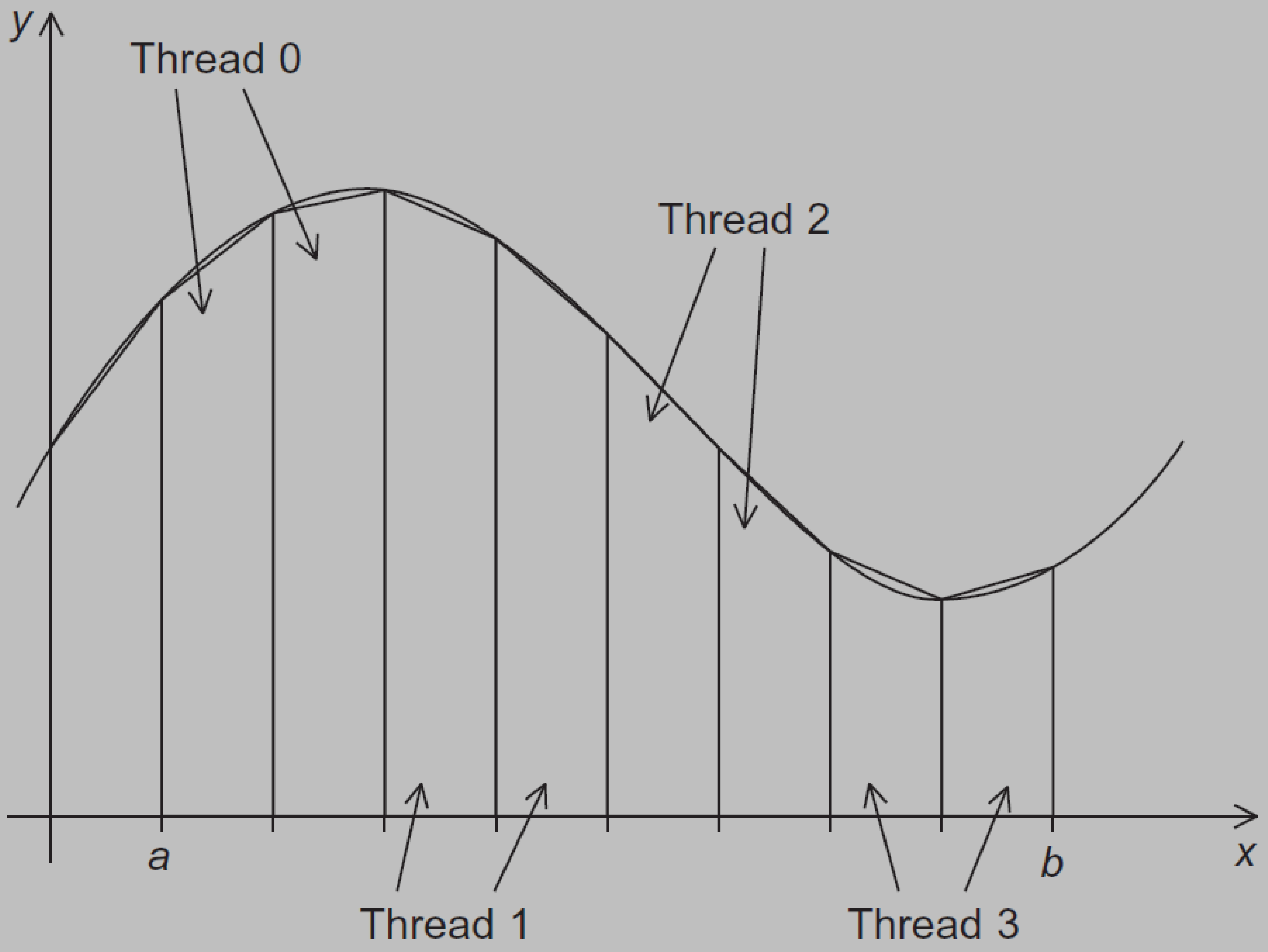
There is no communication among the tasks in the first collection, but each task in the first collection communicates with task 2.

A FIRST OPENMP VERSION

We assumed that there would be many more trapezoids than cores.

So, we aggregated tasks by assigning a contiguous block of trapezoids to each thread (and a single thread to each core).

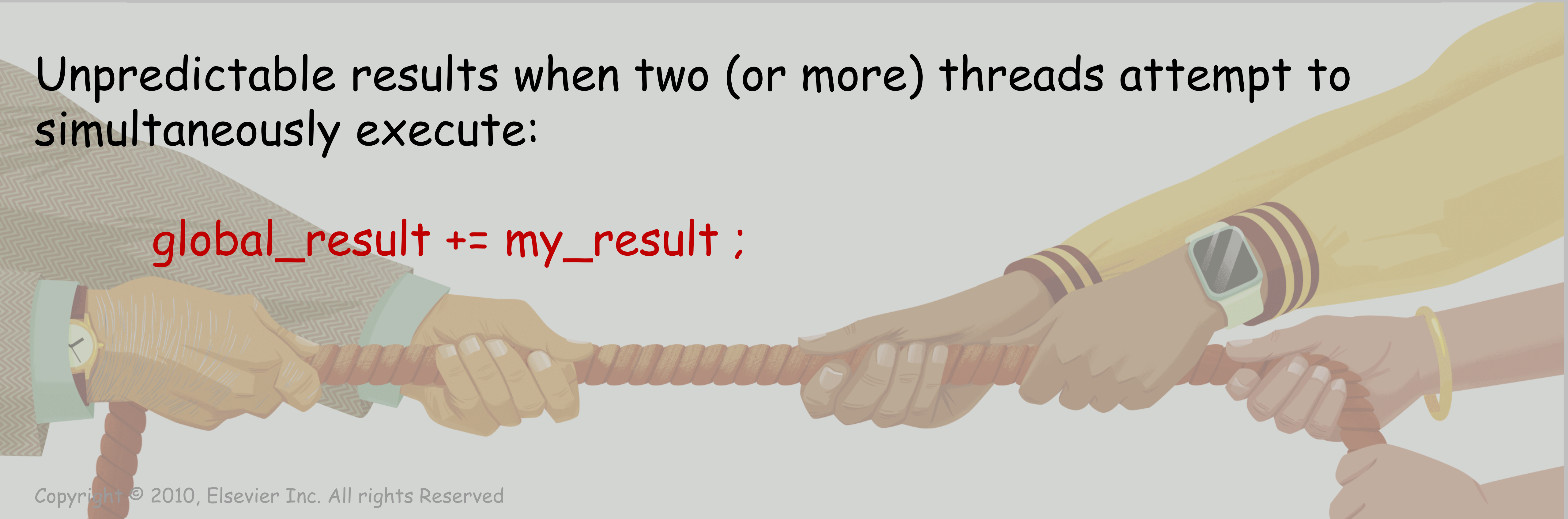
Assignment of trapezoids to threads



Time	Thread 0	Thread 1
0	global_result = 0 to register	finish my_result
1	my_result = 1 to register	global_result = 0 to register
2	add my_result to global_result	my_result = 2 to register
3	store global_result = 1	add my_result to global_result
4		store global_result = 2

Unpredictable results when two (or more) threads attempt to simultaneously execute:

```
global_result += my_result ;
```



Mutual Exclusion

```
# pragma omp critical
   global_result += my_result ;
```

only one thread can execute the following structured block at a time

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

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
   double  global_result = 0.0;  /* Store result in global_result */
   double  a, b;                 /* Left and right endpoints      */
   int     n;                    /* Total number of trapezoids    */
   int     thread_count;

   thread_count = strtol(argv[1], NULL, 10);
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", &a, &b, &n);
#  pragma omp parallel num_threads(thread_count)
   Trap(a, b, n, &global_result);

   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e\n",
      a, b, global_result);
   return 0;
}  /* main */
```

```
void Trap(double a, double b, int n, double* global_result_p) {
   double  h, x, my_result;
   double  local_a, local_b;
   int  i, local_n;
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   h = (b-a)/n;
   local_n = n/thread_count;
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   my_result = (f(local_a) + f(local_b))/2.0;
   for (i = 1; i <= local_n-1; i++) {
     x = local_a + i*h;
     my_result += f(x);
   }
   my_result = my_result*h;

#  pragma omp critical
   *global_result_p += my_result;
}  /* Trap */
```

Scope of Variables

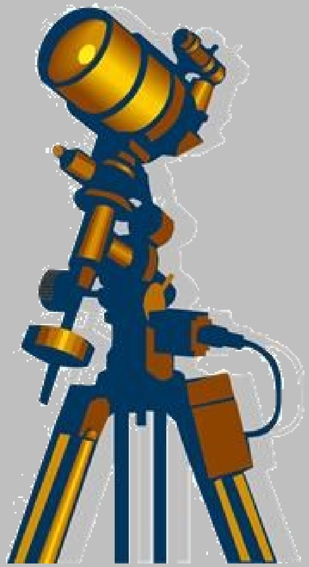
Scope

- In serial programming, the scope of a variable consists of those parts of a program in which the variable can be used.

- In OpenMP, the scope of a variable refers to the set of threads that can access the variable in a parallel block.

Scope in OpenMP

- A variable that can be accessed by all the threads in the team has shared scope.
- A variable that can only be accessed by a single thread has private scope.
- The default scope for variables declared before a parallel block is shared.



THE REDUCTION CLAUSE

We need this more complex version to add each thread's local calculation to get *global_result*.

```
void Trap(double a, double b, int n, double* global_result_p);
```

Although we'd prefer this.

```
double Trap(double a, double b, int n);
```

$$\downarrow$$

```
global_result = Trap(a, b, n);
```

If we use this, there's no critical section!

```
double Local_trap(double a, double b, int n);
```

If we fix it like this…

```
    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count)
    {
#      pragma omp critical
       global_result += Local_trap(double a, double b, int n);
    }
```

… we force the threads to execute sequentially.

We can avoid this problem by declaring a private variable inside the parallel block and moving the critical section after the function call.

```
    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count)
    {
        double my_result = 0.0;  /* private */

        my_result += Local_trap(double a, double b, int n);
#       pragma omp critical
        global_result += my_result;
    }
```

"Do the best you can until you know better. Then when you know better, do better."
– Maya Angelou

Reduction operators

- A reduction operator is a binary operation (such as addition or multiplication).
- A reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result.
- All of the intermediate results of the operation should be stored in the same variable: the reduction variable.

A reduction clause can be added to a parallel directive.

```
reduction(<operator>: <variable list>)
```

+, *, -, &, |, ^, &&, ||

```
    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count) \
       reduction(+: global_result)
    global_result += Local_trap(double a, double b, int n);
```

The "Parallel For" Directive

Parallel for

- Forks a team of threads to execute the following structured block.

- However, the structured block following the parallel for directive must be a for loop.

- Furthermore, with the parallel for directive the system parallelizes the for loop by dividing the iterations of the loop among the threads.

```
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++)
   approx += f(a + i*h);
approx = h*approx;
```

```
    h = (b-a)/n;
    approx = (f(a) + f(b))/2.0;
#   pragma omp parallel for num_threads(thread_count) \
       reduction(+: approx)
    for (i = 1; i <= n-1; i++)
       approx += f(a + i*h);
    approx = h*approx;
```

Legal forms for parallelizable for statements

$$
\textbf{for}\left(\begin{array}{lllll}
 & & & & \texttt{index++} \\
 & & & & \texttt{++index} \\
 & & \texttt{index < end} & & \texttt{index--} \\
 & & \texttt{index <= end} & & \texttt{--index} \\
\texttt{index = start} & ; & \texttt{index >= end} & ; & \texttt{index += incr} \\
 & & \texttt{index > end} & & \texttt{index -= incr} \\
 & & & & \texttt{index = index + incr} \\
 & & & & \texttt{index = incr + index} \\
 & & & & \texttt{index = index - incr}
\end{array}\right)
$$

CAVEATS

- The variable index must have integer or pointer type (e.g., it can't be a float).

- The expressions start, end, and incr must have a compatible type. For example, if index is a pointer, then incr must have integer type.

CAVEATS

- The expressions start, end, and incr must not change during execution of the loop.
- During execution of the loop, the variable index can only be modified by the "increment expression" in the for statement.

Data Dependencies

```
fibo[ 0 ]  =  fibo[ 1 ]  =  1;
for  (i  =  2;  i  <  n;  i++)
     fibo[ i ]  =  fibo[ i - 1 ] + fibo[ i - 2 ];
```

note 2 threads

```
     fibo[ 0 ]  =  fibo[ 1 ]  =  1;
#   pragma  omp  parallel  for  num_threads(2)
     for  (i  =  2;  i  <  n;  i++)
          fibo[ i ]  =  fibo[ i - 1 ] + fibo[ i - 2 ];
```

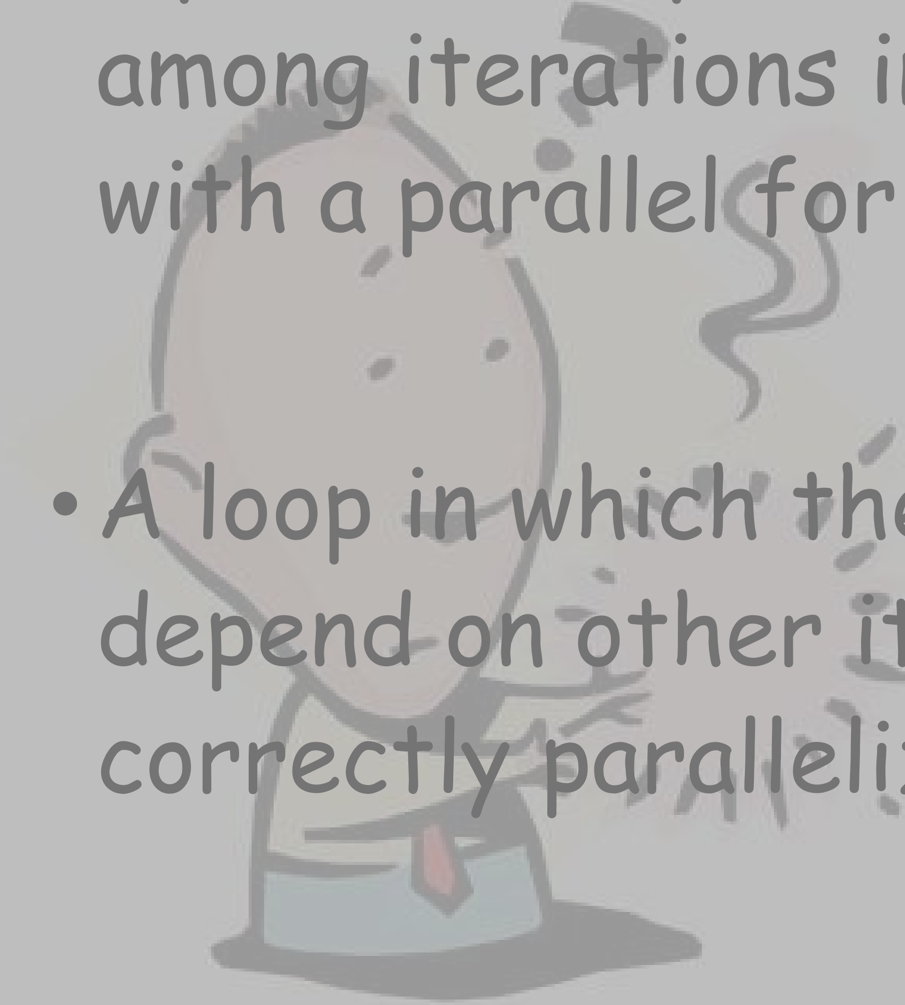
but sometimes we get this

```
1 1 2 3 5 8 13 21 34 55
```

this is correct

```
1 1 2 3 5 8 0 0 0 0
```

What happened?



- OpenMP compilers don't check for dependences among iterations in a loop that's being parallelized with a parallel for directive.

- A loop in which the results of one or more iterations depend on other iterations cannot, in general, be correctly parallelized by OpenMP.