Computação de Alto Desempenho (High Performance Computing)

Slides adapted from “An Introduction to Parallel Programming”, Peter Pacheco

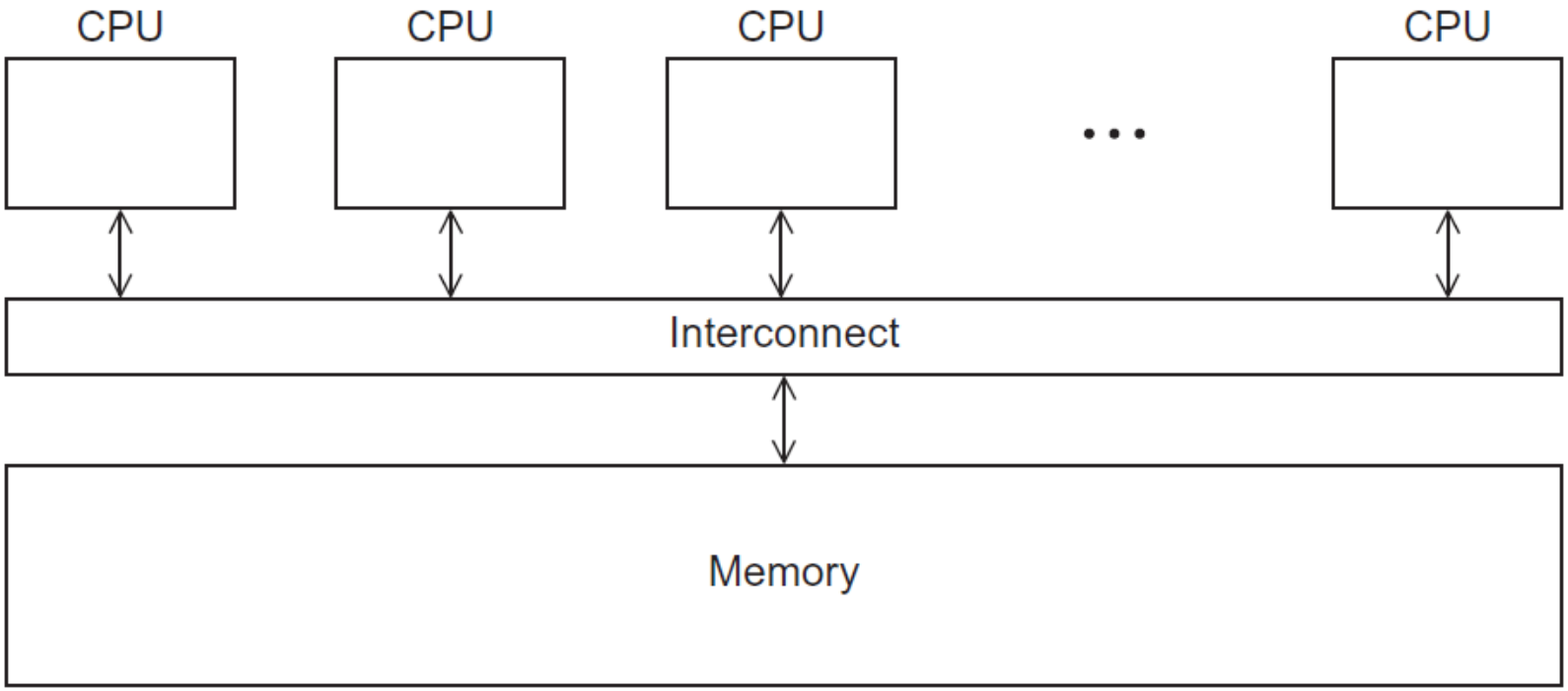
Shared Memory Programming with OpenMP (Chapter 5) -- Roadmap

- Writing programs that use OpenMP.
- Using OpenMP to parallelize many serial for loops with only small changes to the source code.

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 CPUs, all of which have access to main memory.

A shared memory system



OpenMP

- Allows incremental modification of serial programs (contrary to MPI programs)

Pragmas

- Special preprocessor instructions.
- Typically added to a system to allow behaviors that are not part of the basic C specification.
- Compilers that do not support the pragmas ignore them.

```
#pragma
```

```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 */
```

```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
```

compiling

running with 4 threads

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

possible outcomes

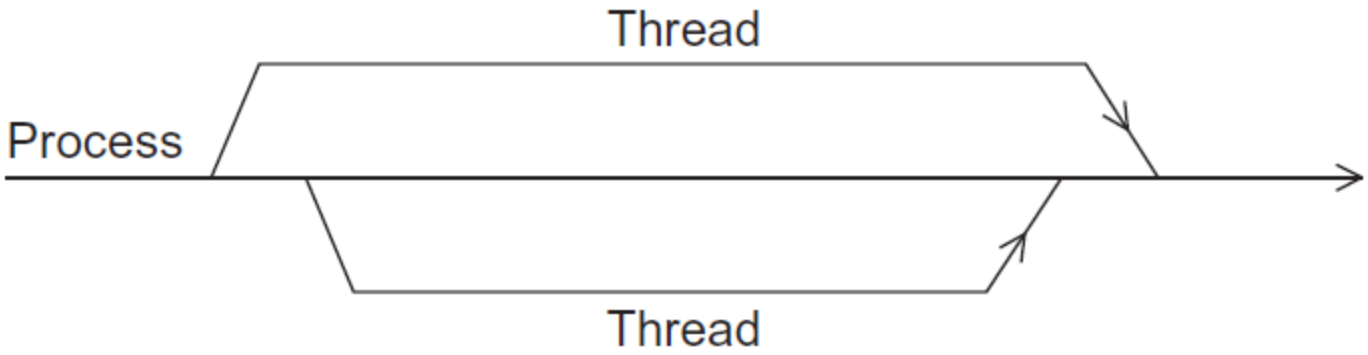
```
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

```
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…

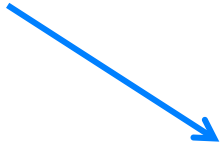
- There may be system-defined limitations on the number of threads that a program can start.
- The OpenMP standard does not guarantee that this will actually start thread_count threads.
- Most current systems can start hundreds or even thousands of threads.
- Unless we are trying to start a lot of threads, 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 does not support OpenMP

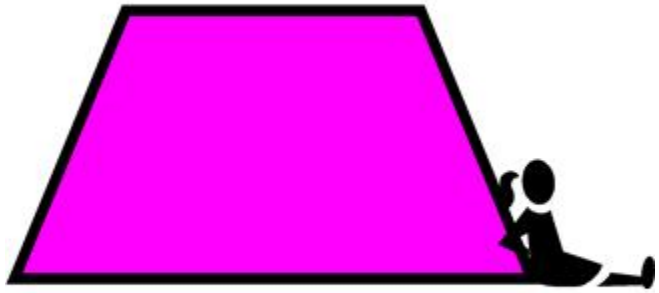
```
# include <omp.h>
```



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

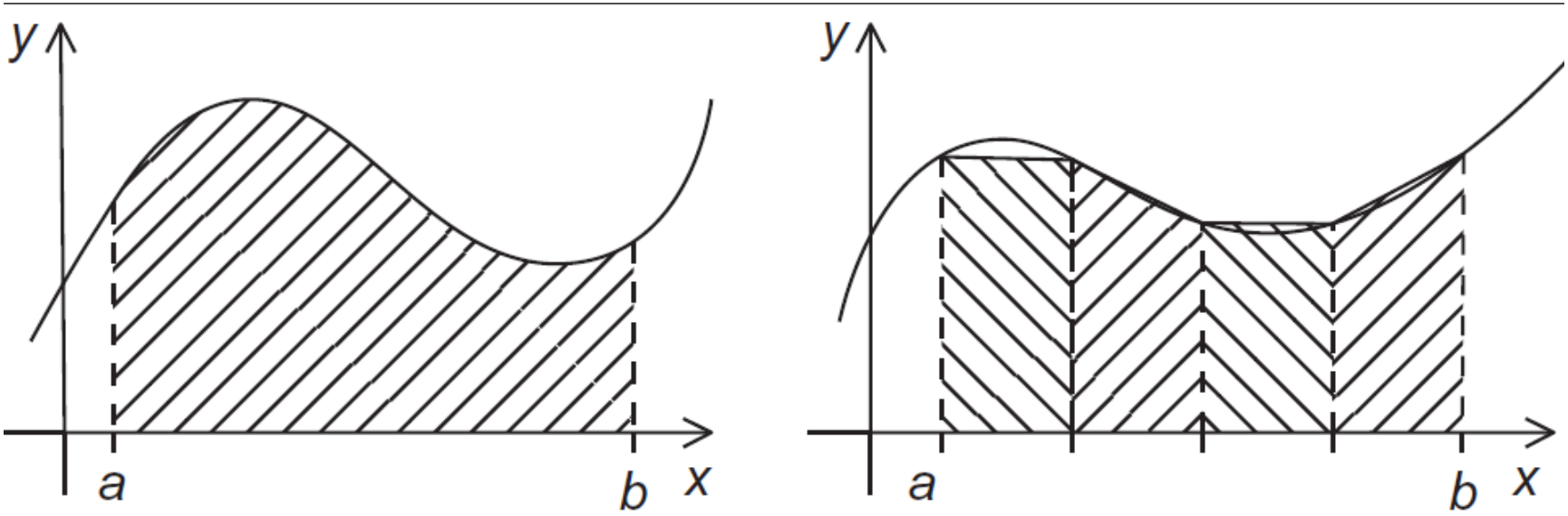
In case the compiler does not support OpenMP

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



THE TRAPEZOIDAL RULE

The trapezoidal rule

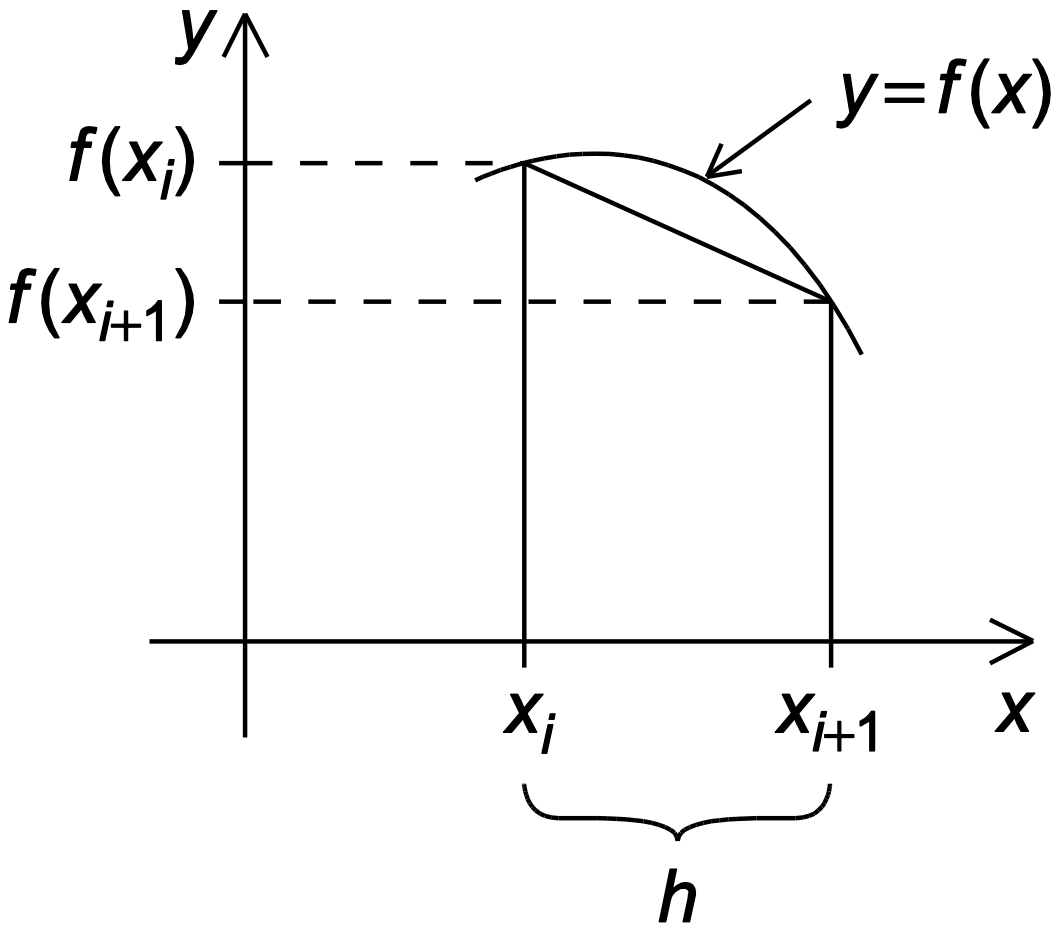


The Trapezoidal Rule

$$\text{Area of one trapezoid} = \frac{h}{2}[f(x_i) + f(x_{i+1})]$$

$$h = \frac{b-a}{n}$$

One trapezoid



The Trapezoidal Rule

$$\text{Area of one trapezoid} = \frac{h}{2}[f(x_i) + f(x_{i+1})]$$

$$h = \frac{b-a}{n}$$

$$x_0 = a,\ x_1 = a+h,\ x_2 = a+2h,\ \ldots,\ x_{n-1} = a+(n-1)h,\ x_n = b$$

$$\text{Sum of trapezoid areas} = h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2]$$

Pseudo-code for a serial program

```
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 0; i <= n-1; i++) {
    x_i = a + i*h;
    approx += f(x_i);
}
approx = h*approx;
```

Parallelizing the Trapezoidal Rule -- Foster's methodology

1. *Partitioning*

   Divide the computation to be performed and the data operated on by the computation into small tasks. The focus here should be on identifying tasks that can be executed in parallel.

2. *Communication*

   Determine what communication needs to be carried out among the tasks identified in the previous step.

3. *Agglomeration or aggregation*

   Combine tasks and communications identified in the first step into larger tasks.

4. *Mapping*

   Assign the composite tasks identified in the previous step to processes/ threads.

A First OpenMP Version

1) Two types of tasks:
   a) computation of the areas of individual trapezoids, and
   b) adding the areas of trapezoids.

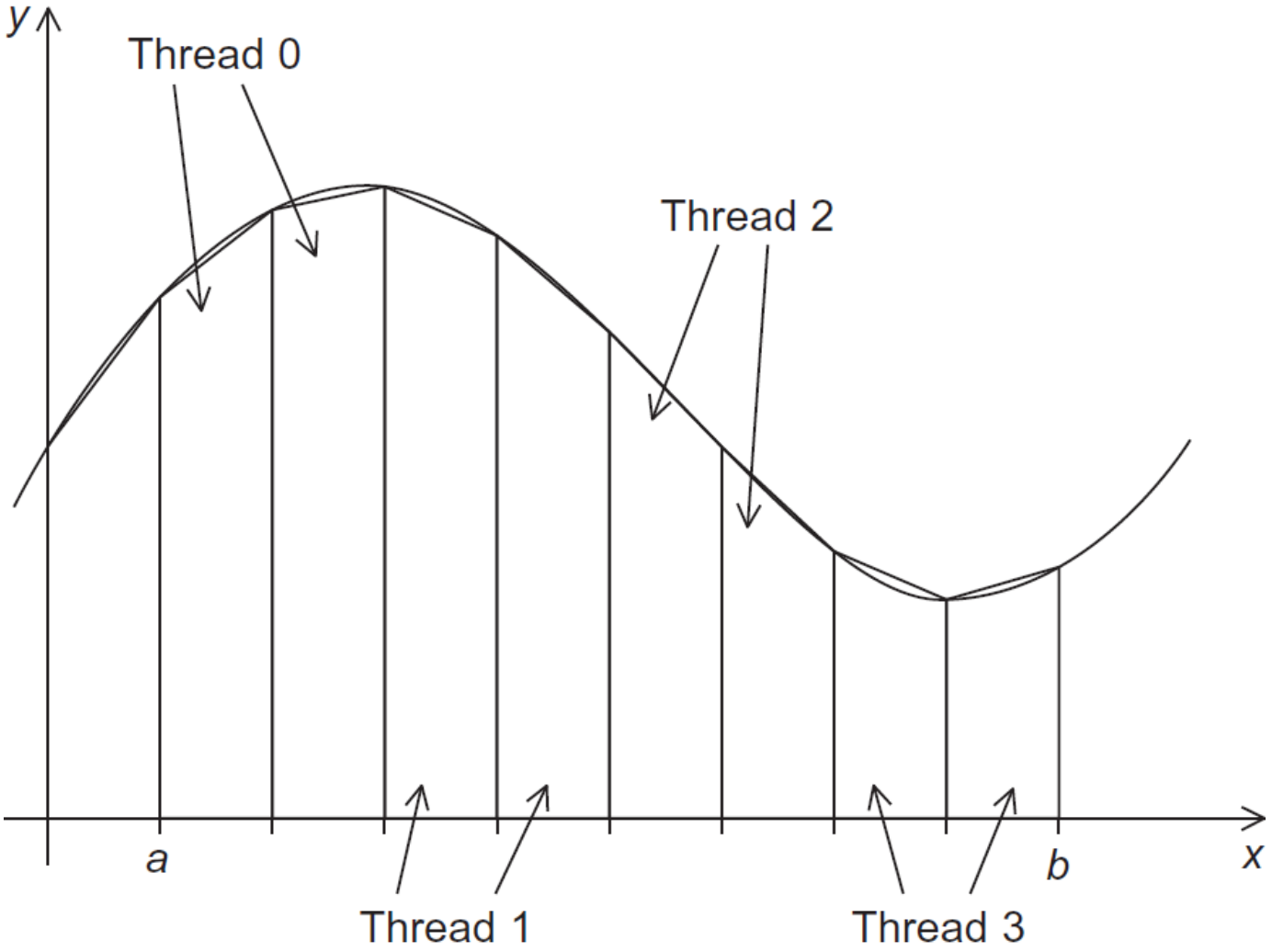
2) There is no communication among the tasks in the first collection, but each task in the first collection communicates with task 1b.

A First OpenMP Version

3) We assume that there are many more trapezoids than cores.

- Tasks are aggregated by assigning a contiguous block of trapezoids to each thread (and a single thread to each core).

Assignment of trapezoids to threads



Serial algorithm →Parallel

```
/* 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;
```

Division of the following tasks among two (or more threads) :
a) computation of the areas of individual trapezoids, and
b) adding the areas of trapezoids.

```c
#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 */
```

Parallel algorithm

```
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;
   if (argc != 2) Usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", &a, &b, &n);
   if (n % thread_count != 0) Usage(argv[0]);
   #  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 Usage(char* prog_name) {

   fprintf(stderr, "usage: %s <number of threads>\n", prog_name);
   fprintf(stderr, "   number of trapezoids must be evenly divisible by\n");
   fprintf(stderr, "   number of threads\n");
   exit(0);
}  /* Usage */
```

Assuming two threads

```
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;
   *global_result_p += my_result; }
```

Problems?

Assuming two 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

```
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 */
```

```
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 */
```

The directive *critical* defines a critical section

-- tells the compiler that the system needs to arrange for the threads to have mutually exclusive access to the following structured block of code.

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.

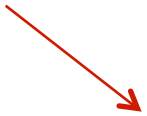
THE REDUCTION CLAUSE

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);
```



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

If we use this, there is no critical section!

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

If we use this, there is 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;
    }
```

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;
    }
```

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 *clause* modifies a *directive*)

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);
```

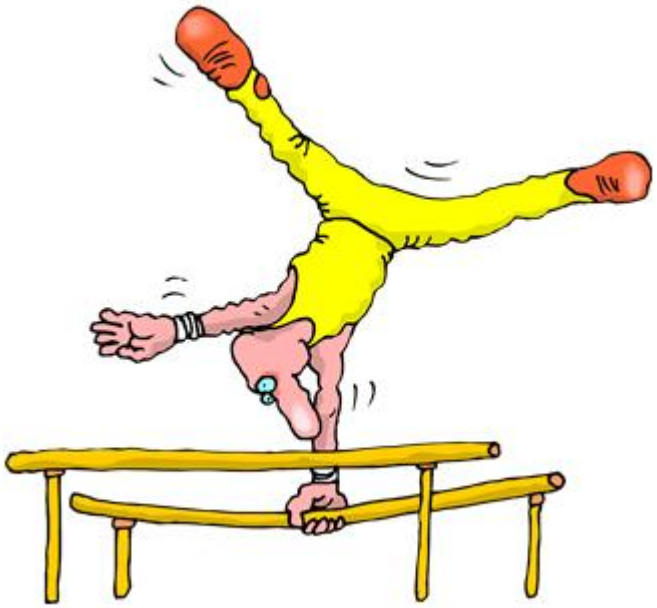
(a *clause* modifies a *directive*)

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);
```

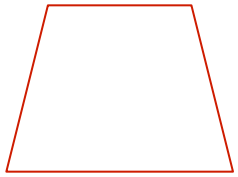


THE "PARALLEL FOR" DIRECTIVE

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
  - Usually block partitioning: *m iterations* and the first *m/thread count* are assigned to *thread 0*, the *next m/thread count* are assigned to *thread 1*, etc

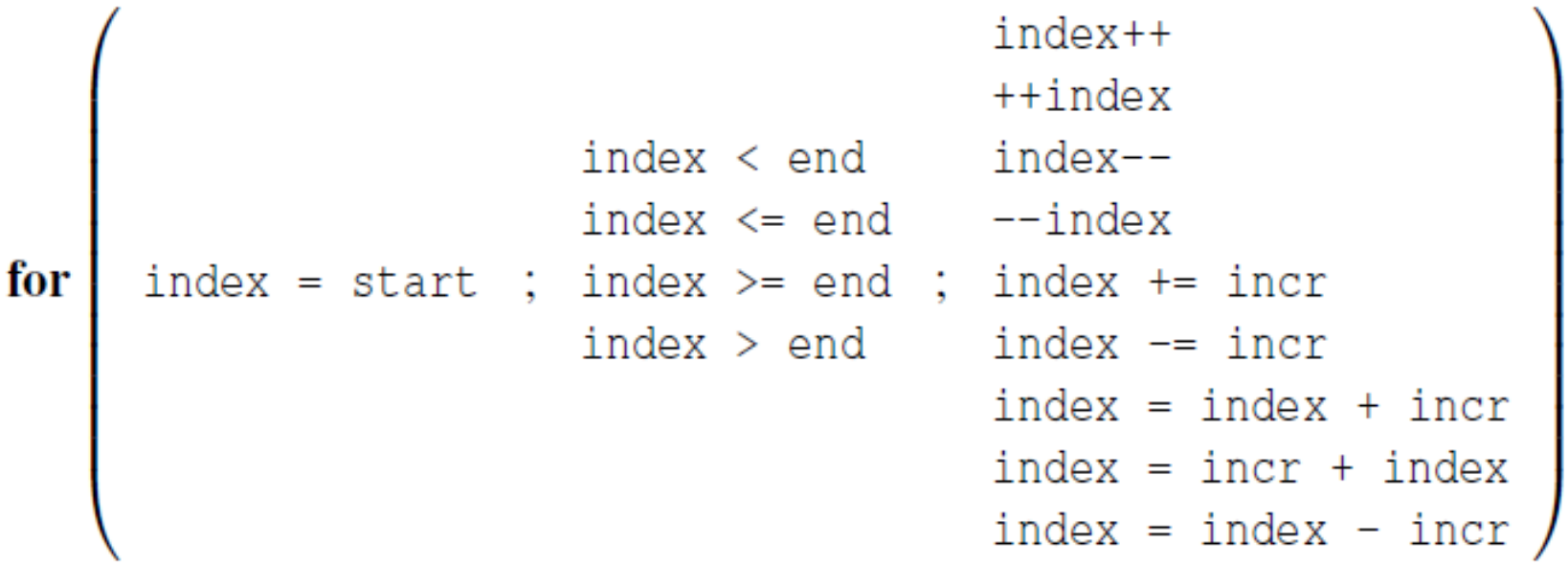
```
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++)
   approx += f(a + i*h);
approx = h*approx;
```



```
   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;
```

The varibles' **default scope** is **private**; *i* is private; there are local variables being used to add to the **reduction variable** *approx*

Legal forms for *parallelizable for statements*



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.

- However, the exit() call is valid inside a *parallel for*

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 ];
```

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 ];
```

1 1 2 3 5 8 13 21 34 55

this is correct

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 ];
```

1 1 2 3 5 8 13 21 34 55

this is correct

but sometimes we get this

1 1 2 3 5 8 0 0 0 0

What happened?

1. OpenMP compilers don't check for dependences among iterations in a loop that's being parallelized with a parallel for directive.
2. A loop in which the results of one or more iterations depend on other iterations cannot, in general, be correctly parallelized by OpenMP.

Estimating π

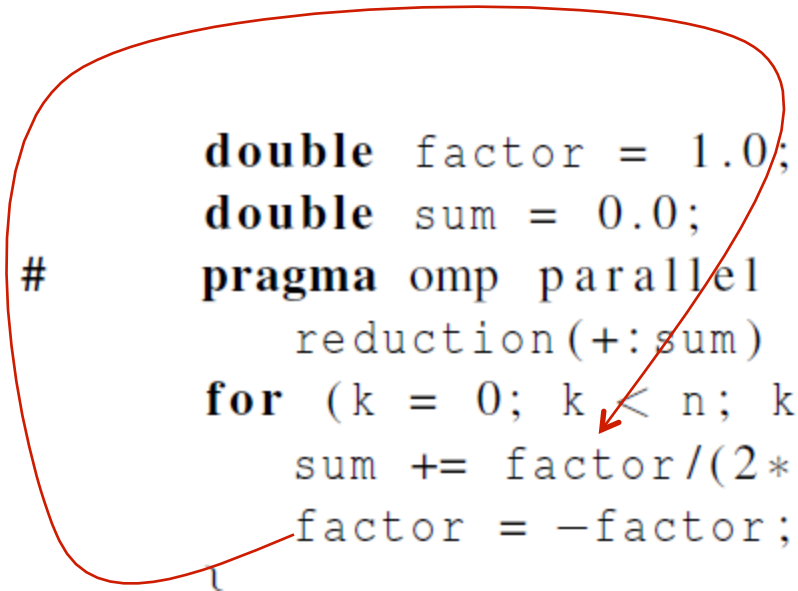
$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

```
double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++) {
    sum += factor/(2*k+1);
    factor = -factor;
}
pi_approx = 4.0*sum;
```

OpenMP solution #1

```
        double factor = 1.0;
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
            reduction(+:sum)
        for (k = 0; k < n; k++) {
            sum += factor/(2*k+1);
            factor = -factor;
        }
        pi_approx = 4.0*sum;
```

OpenMP solution #1




```
        double factor = 1.0;
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
            reduction(+:sum)
        for (k = 0; k < n; k++) {
            sum += factor/(2*k+1);
            factor = -factor;
        }
        pi_approx = 4.0*sum;
```

OpenMP solution #2

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum)
        for (k = 0; k < n; k++) {
           if (k % 2 == 0)
              factor = 1.0;
           else
              factor = -1.0;
           sum += factor/(2*k+1);
        }
```

OpenMP solution #2

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum)
        for (k = 0; k < n; k++) {
           if (k % 2 == 0)
              factor = 1.0;
           else
              factor = -1.0;
           sum += factor/(2*k+1);
        }
```

What is the problem here?

OpenMP solution #3

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
            reduction(+:sum) private(factor)
        for (k = 0; k < n; k++) {
            if (k % 2 == 0)
               factor = 1.0;
            else
               factor = -1.0;
            sum += factor/(2*k+1);
        }
```

Insures factor has private scope.

The default clause

- Lets the programmer specify the scope of each variable in a block.

```
default(none)
```

- With this clause the compiler will require that we specify the scope of each variable we use in the block and that has been declared outside the block.

The default clause

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           default(none) reduction(+:sum) private(k, factor) \
           shared(n)
        for (k = 0; k < n; k++) {
           if (k % 2 == 0)
              factor = 1.0;
           else
              factor = -1.0;
           sum += factor/(2*k+1);
        }
```

Concluding Remarks (1)

- OpenMP is a standard for programming shared-memory systems.
- OpenMP uses both special functions and preprocessor directives called pragmas.
- OpenMP programs start multiple threads rather than multiple processes.
- Many OpenMP directives can be modified by clauses.

Concluding Remarks (2)

- A major problem in the development of shared memory programs is the possibility of race conditions.
- OpenMP provides several mechanisms for insuring mutual exclusion in critical sections.
  - Critical directives
  - Named critical directives
  - Atomic directives
  - Simple locks