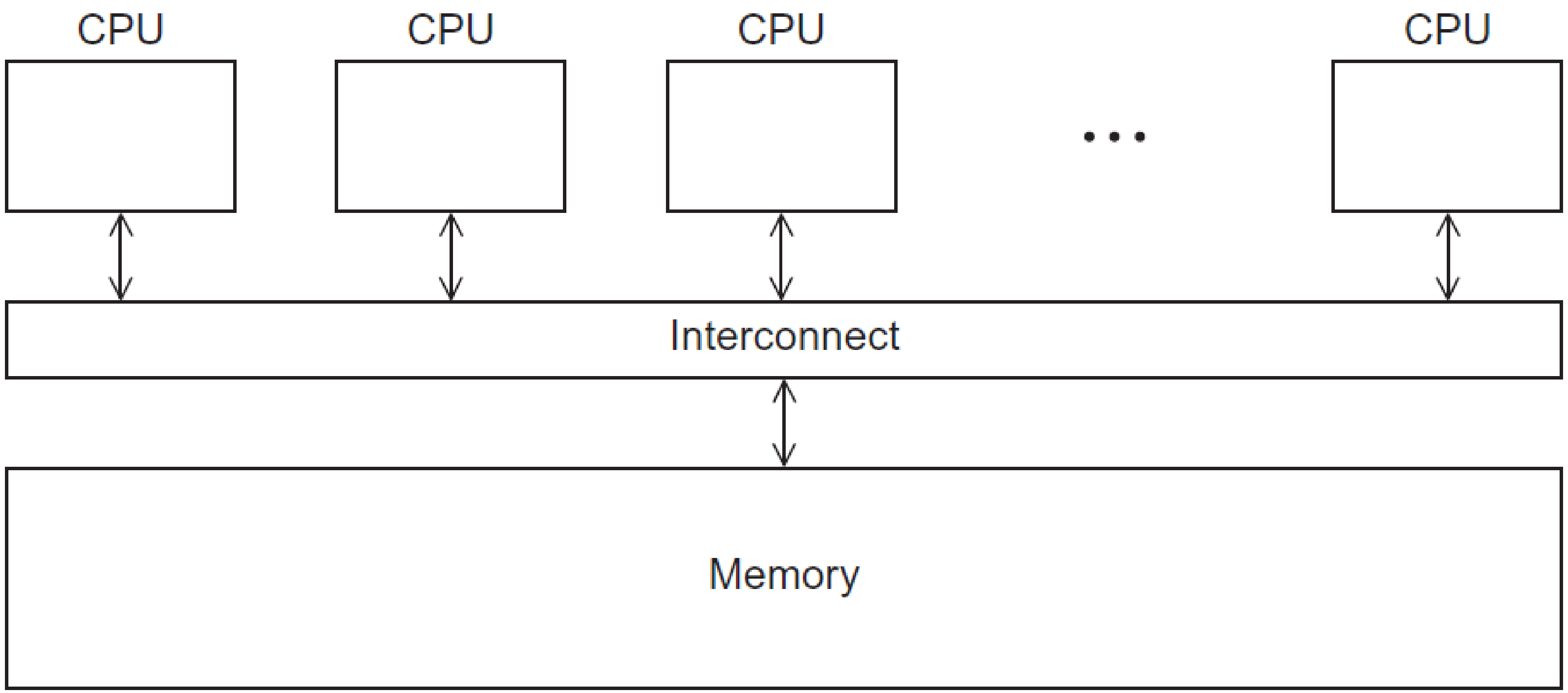
ECE 432/532
Programming for Parallel Processors

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



OpenMP vs Pthreads

- Pthreads requires that the programmer explicitly specify the behavior of each thread
  - OpenMP allows the programmer to simply state that a block of code should be executed in parallel.
- Pthreads (like MPI) is a library of functions that can be linked to a C program
  - any Pthreads program can be used with any C compiler, provided the system has a Pthreads library
- OpenMP, on the other hand, requires compiler support for some operations, and hence it's entirely possible that you may run across a C compiler that can't compile OpenMP programs into parallel programs

OpenMP vs Pthreads

- Pthreads is lower level and provides the power to program virtually any conceivable thread behavior.
    - But it’s up to us to specify every detail of the behavior of each thread.

- OpenMP allows the compiler and run-time system to determine some of the details of thread behavior
    - simpler to code some parallel behaviors using OpenMP
    - The cost is that some low-level thread interactions can be more difficult to program.

OpenMP vs Pthreads

- OpenMP was developed by a group of programmers and computer scientists ofr large-scale high-performance programs
    - Usign Pthreads was too difficult

- In OpenMP shared-memory programs can be developed at a higher level

- OpenMP was explicitly designed to allow programmers to *incrementally* parallelize existing serial programs
    - Virtually impossible with MPI and fairly difficult with Pthreads.

Pragmas

- Special preprocessor instructions.
- 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.

#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 */
```

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

. / omp_hello 4
```

compiling

running with 4 threads

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

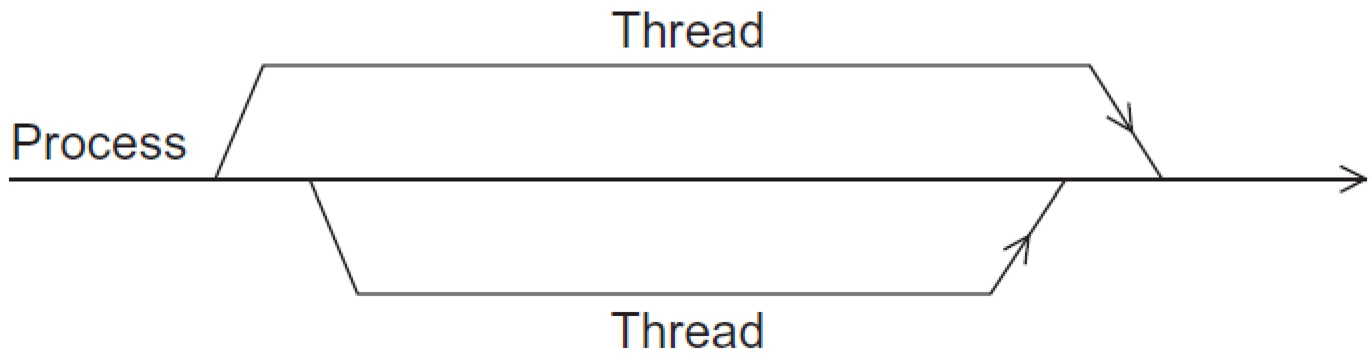
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4

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.
    - The system will typically run one thread on each available core

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.

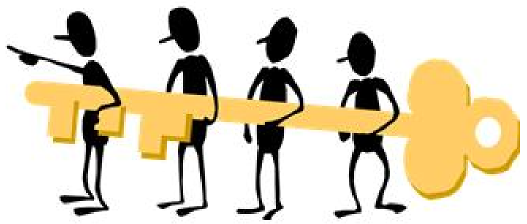
```
# pragma omp parallel num_threads ( thread_count )
```

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

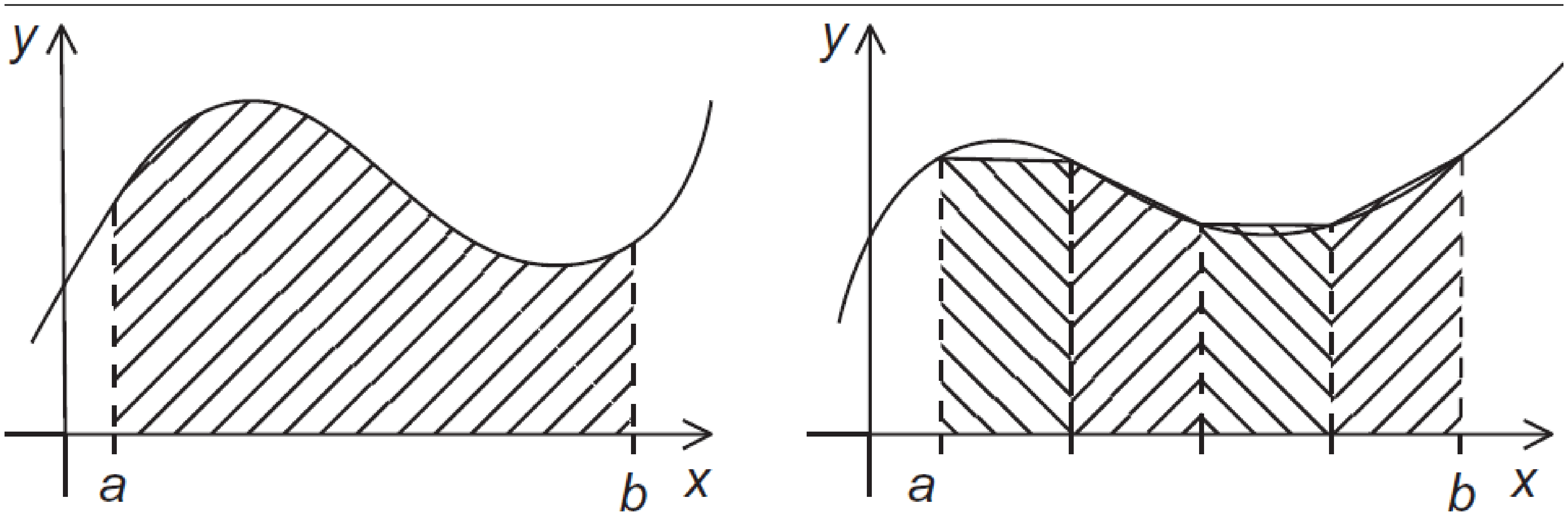
```
# include <omp.h>
```

```
#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 ( );
# e l s e
  int my_rank = 0;
  int thread_count = 1;
# endif
```

The trapezoidal rule



Serial algorithm

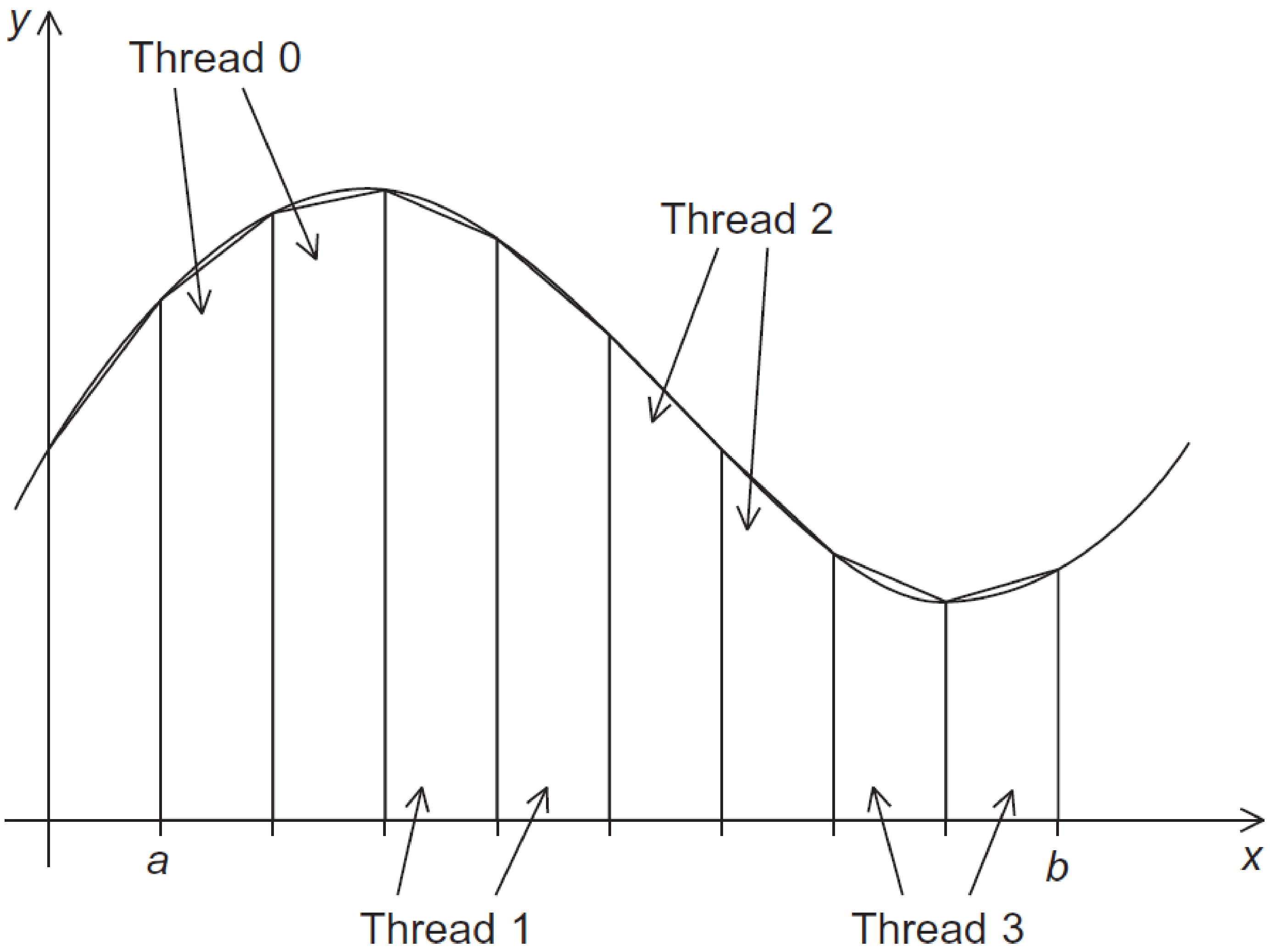
```
/* Input: a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
   x_i = a + i*h;
   approx += f(x_i);
}
approx = h*approx;
```

A First OpenMP Version

1) We identified 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.
3) 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).

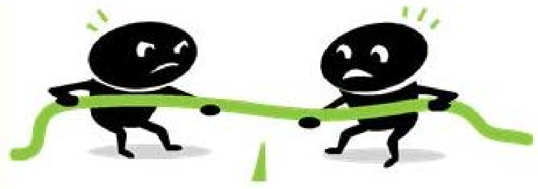
Assignment of trapezoids to threads



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

```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 */
```

```
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.

Comments from previous lecture

- The parallel directive (# pragma omp) specifies that the **structured block** of code that follows should be executed by multiple threads.

- A structured block is a C statement or a compound C statement with one point of entry and one point of exit
  - although calls to the function exit are allowed

- This definition simply prohibits code that branches into or out of the middle of the structured block

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'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;
}
```

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);
```

Reduction clause

- In C, operator can be any one of the operators +, *,-, &, | , ^, &&, ||

- However, the use of subtraction is problematic
  - subtraction isn't associative or commutative.
- Example, the code returns -10

```
result = 0;
for (i = 1; i <= 4; i++)
   result -= i;
```

Reduction clause

- If we split the iterations among two threads
  - thread 0 subtracting 1 and 2
  - thread 1 subtracting 3 and 4

- Thread 0 will compute -3 and thread 1 will compute -7
- result = -3 – (-7) = 4

```
result = 0;
for (i = 1; i <= 4; i++)
   result -= i;
```

Reduction clause

- In principle, the compiler should determine that the threads' individual results should actually be added (-3 + (-7)=-10)

- Even though this seems to be the case

- OpenMP Standard doesn't seem to guarantee this.

Reduction clause

- When a variable is included in a reduction clause, the variable itself is shared.
- However, a private variable is created for each thread in the team.
- In the parallel block each time a thread executes a statement involving the variable, it uses the private variable.
- When the parallel block ends, the values in the private variables are combined into the shared variable.

Reduction clause

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

=

```
   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;
   }
```

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;
```

Omp parallel for

- Like the parallel directive, the parallel **for** directive 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

- With the parallel **for** directive the system parallelizes the **for** loop by dividing the iterations of the loop among the threads.
  - in parallel directive, the work must be divided among the threads by the threads themselves.

```c
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;
```

```c
   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;
```

Omp parallel for

- Speaking of scope, the default scope for all variables in a parallel directive is shared
- But in parallel **for** if the loop variable i were shared, the variable update, i++, would also be an unprotected critical section.
- In a loop that is parallelized with a parallel **for** directive, the default scope of the loop variable is *private*
  - each thread in the team has its own copy of i.

Caveats – parallel for

- OpenMP will only parallelize **for** loops
  - Not while loops
  - Not do-while loops

- OpenMP will only parallelize **for** loops for which the number of iterations can be determined

```
for ( ; ; ) {
    . . .
}
```

```
for (i = 0; i < n; i++) {
    if ( . . . ) break;
    . . .
}
```

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.

Caveats – Is this valid?

```
int Linear_search(int key, int A[], int n) {
   int i;
   /* thread_count is global */
#  pragma omp parallel for num_threads(thread_count)
   for (i = 0; i < n; i++)
      if (A[i] == key) return i;
   return -1;  /* key not in list */
}
```

Caveats

```
int Linear_search(int key, int A[], int n) {
   int i;
   /* thread_count is global */
#  pragma omp parallel for num_threads(thread_count)
   for (i = 0; i < n; i++)
      if (A[i] == key) return i;
   return -1;  /* key not in list */
}
```

The `gcc` compiler reports:

```
Line 6: error: invalid exit from OpenMP structured block
```

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?

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

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

Data dependencies

- Is there any problem here?

```
#  pragma omp parallel for num_threads(thread_count)
   for (i = 0; i < n; i++) {
      x[i] = a + i*h;
      y[i] = exp(x[i]);
   }
```

Data dependencies

- Is there any problem here?

```
#  pragma omp parallel for num_threads(thread_count)
   for (i = 0; i < n; i++) {
      x[i] = a + i*h;
      y[i] = exp(x[i]);
   }
```

- there is a data dependence between Lines 3 and 4. However, there is no problem with the parallelization
  - the computation of x[i] and its subsequent use will always be assigned to the same thread

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

loop dependency

```
        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) 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);
}
```

```
int main(int argc, char* argv[]) {
   long long n, i;
   int thread_count;
   double factor;
   double sum = 0.0;

   if (argc != 3) exit(0);
   thread_count = strtol(argv[1], NULL, 10);
   n = strtoll(argv[2], NULL, 10);
   if (thread_count < 1 || n < 1) exit(0);

#  pragma omp parallel for num_threads(thread_count) reduction(+: sum) private(factor)
   for (i = 0; i < n; i++) {
      factor = (i % 2 == 0) ? 1.0 : -1.0;
      sum += factor/(2*i+1);}
   sum = 4.0*sum;
   printf("With n = %lld terms and %d threads,\n", n, thread_count);
   printf("   Our estimate of pi = %.14f\n", sum);
}  /* main */
```

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

Bubble Sort

```
for (list_length = n; list_length >= 2; list_length--)
    for (i = 0; i < list_length-1; i++)
        if (a[i] > a[i+1]) {
            tmp = a[i];
            a[i] = a[i+1];
            a[i+1] = tmp;
        }
```

Serial Odd-Even Transposition Sort

```
for (phase = 0; phase < n; phase++)
   if (phase % 2 == 0)
      for (i = 1; i < n; i += 2)
         if (a[i−1] > a[i]) Swap(&a[i−1],&a[i]);
   else
      for (i = 1; i < n−1; i += 2)
         if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

Serial Odd-Even Transposition Sort

Phase	Subscript in Array 0		1		2		3
0	9	↔	7		8	↔	6
	7		9		6		8
1	7		9	↔	6		8
	7		6		9		8
2	7	↔	6		9	↔	8
	6		7		8		9
3	6		7	↔	8		9
	6		7		8		9

First OpenMP Odd-Even Sort

```
      for (phase = 0; phase < n; phase++) {
         if (phase % 2 == 0)
#           pragma omp parallel for num_threads(thread_count) \
               default(none) shared(a, n) private(i, tmp)
            for (i = 1; i < n; i += 2) {
               if (a[i-1] > a[i]) {
                  tmp = a[i-1];
                  a[i-1] = a[i];
                  a[i] = tmp;
               }
            }
         else
#           pragma omp parallel for num_threads(thread_count) \
               default(none) shared(a, n) private(i, tmp)
            for (i = 1; i < n-1; i += 2) {
               if (a[i] > a[i+1]) {
                  tmp = a[i+1];
                  a[i+1] = a[i];
                  a[i] = tmp;
               }
            }
      }
```

Second OpenMP Odd-Even Sort

```
#   pragma omp parallel num_threads(thread_count) \
       default(none) shared(a, n) private(i, tmp, phase)
    for (phase = 0; phase < n; phase++) {
       if (phase % 2 == 0)
#         pragma omp for
          for (i = 1; i < n; i += 2) {
             if (a[i-1] > a[i]) {
                tmp = a[i-1];
                a[i-1] = a[i];
                a[i] = tmp;
             }
          }
       else
#         pragma omp for
          for (i = 1; i < n-1; i += 2) {
             if (a[i] > a[i+1]) {
                tmp = a[i+1];
                a[i+1] = a[i];
                a[i] = tmp;
             }
          }
    }
```

Odd-even sort with two parallel for directives and two for directives.

(Times are in seconds.)

thread_count	1	2	3	4
Two `parallel` **for** directives	0.770	0.453	0.358	0.305
Two **for** directives	0.732	0.376	0.294	0.239