# Introduction to OpenMP

- Introduction
- OpenMP basics
- OpenMP directives, clauses, and library routines

# Programming with Threads

## Several thread libraries, more being created

- PThreads is the POSIX Standard
  - Relatively low level
    - Programmer expresses thread management and coordination
    - Programmer decomposes parallelism and manages schedule
  - Portable but possibly slow
  - Most widely used for systems-oriented code, and also used for some kinds of application code
- OpenMP is newer standard
  - Higher-level support for scientific programming on shared memory architectures

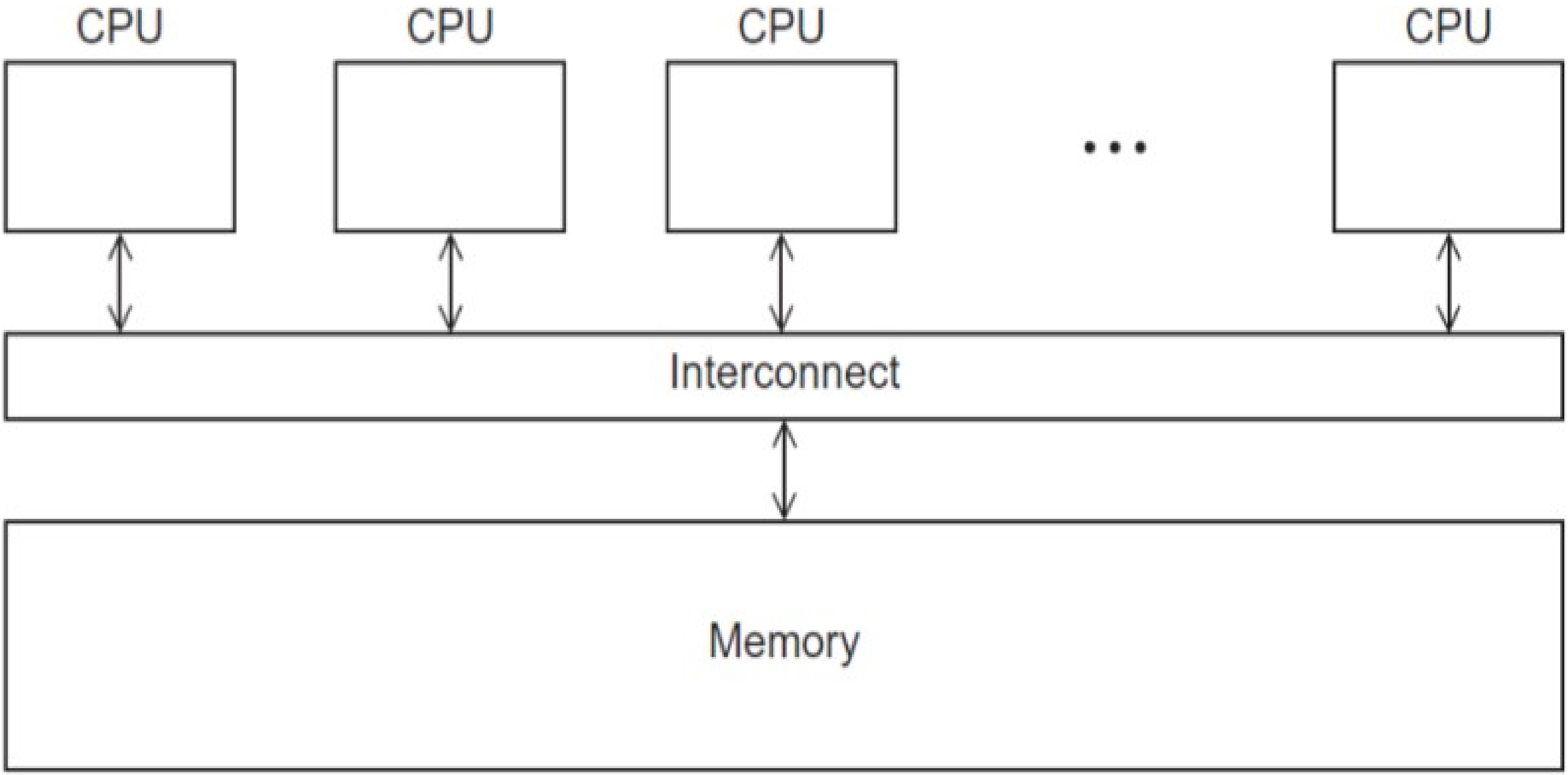
# Introduction to OpenMP

- Introduction
- OpenMP basics
- OpenMP directives, clauses, and library routines

# 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



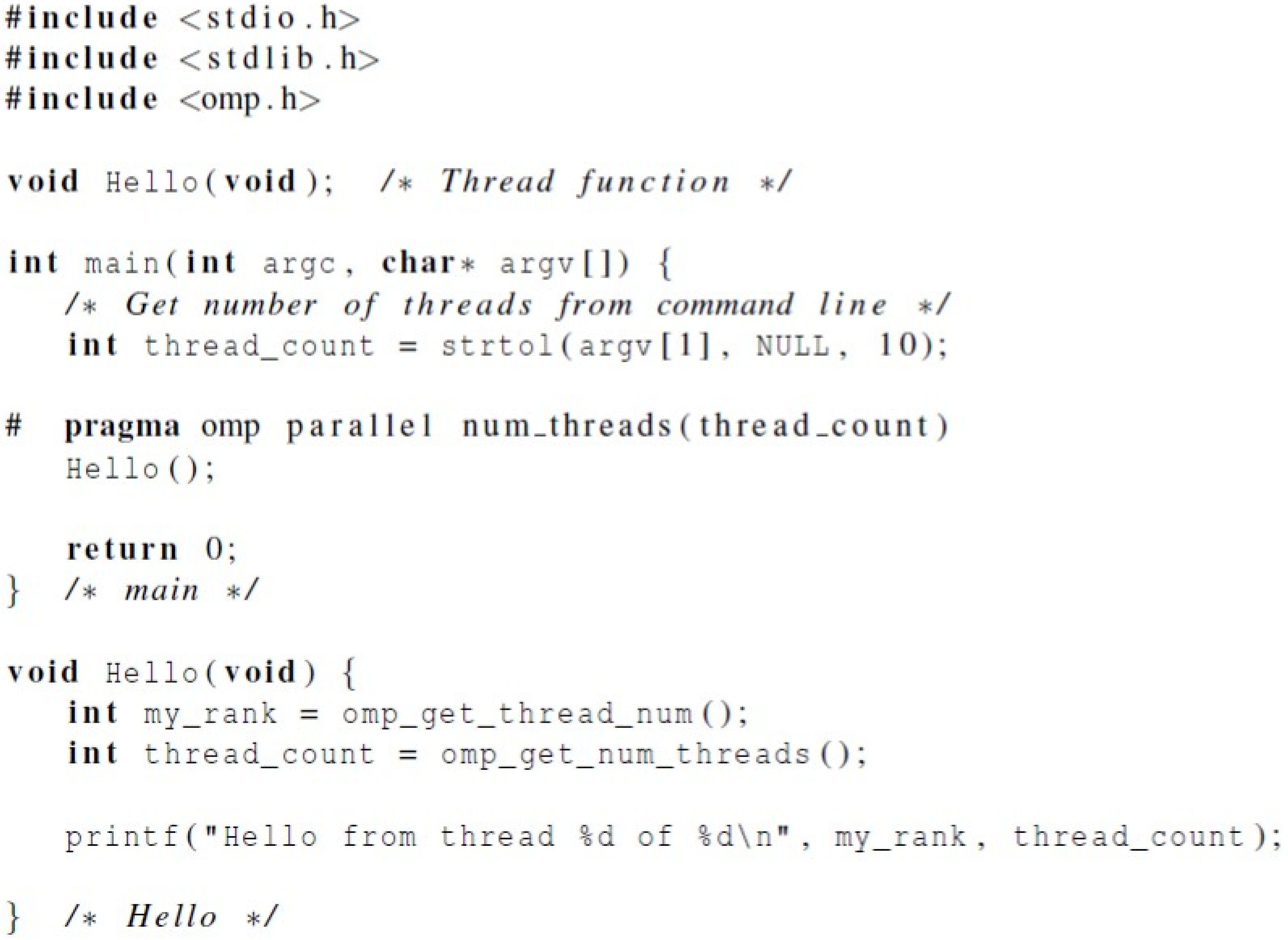
# Definition of OpenMP

- Application Program Interface (API) for Shared Memory Parallel Programming
- Directive based approach with library support
- Targets existing applications and widely used languages:
  - Fortran API released October `97
  - C, C++ API released October `98
- Multi-vendor/platform support

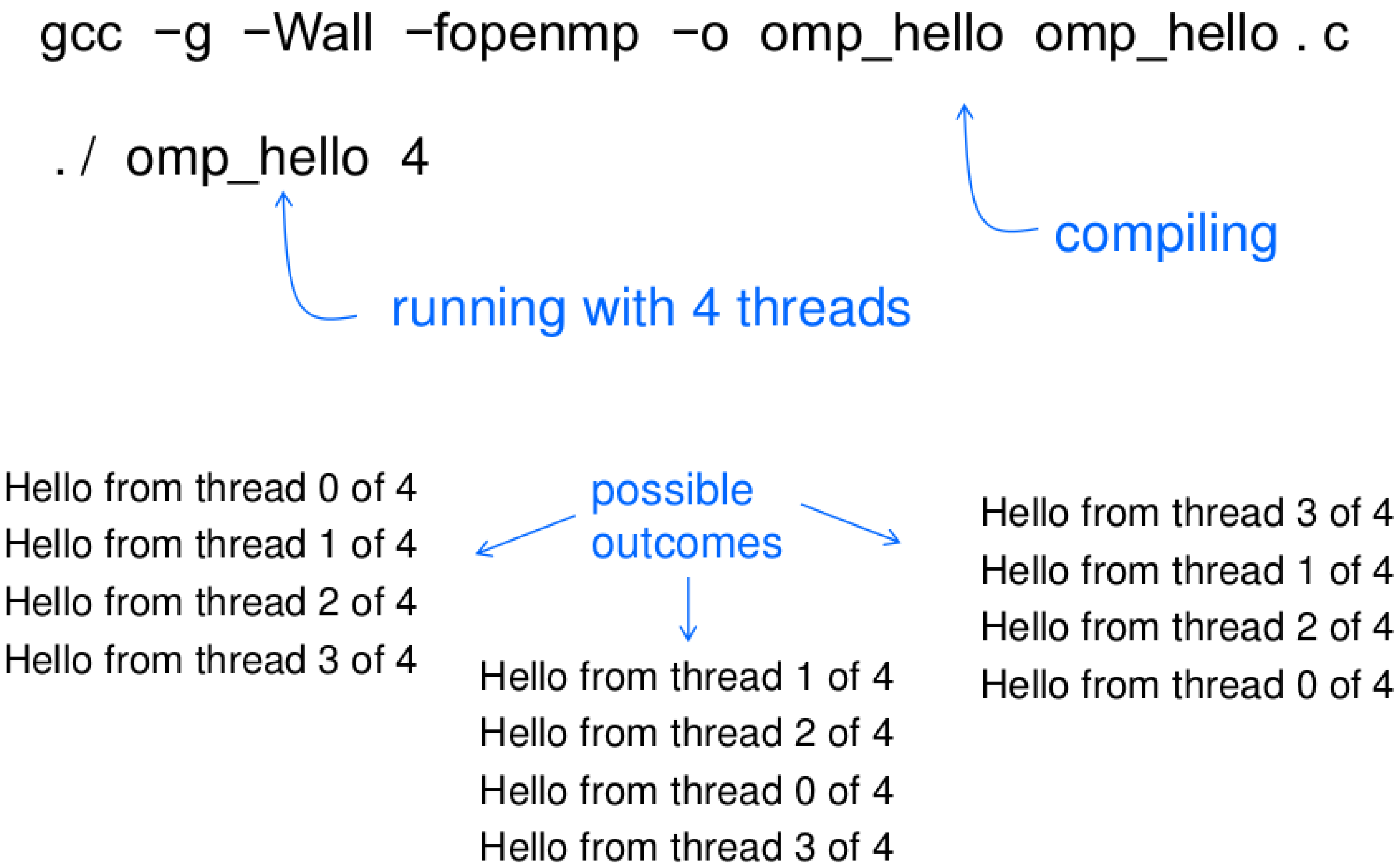
# Pragmas

- 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
```

compiling

```
./omp_hello 4
```

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
```

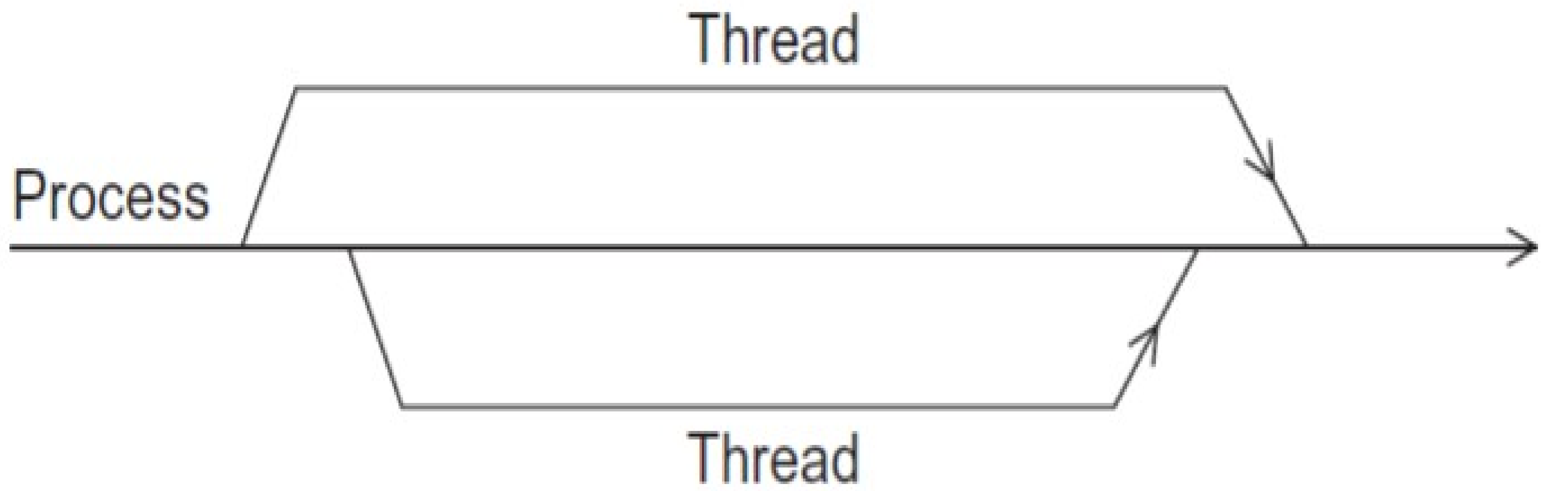
```
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
```

# 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.

# 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.

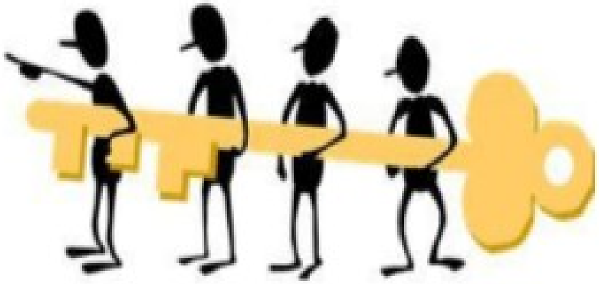
```
# pragma omp parallel num_threads ( thread_count )
```

# 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

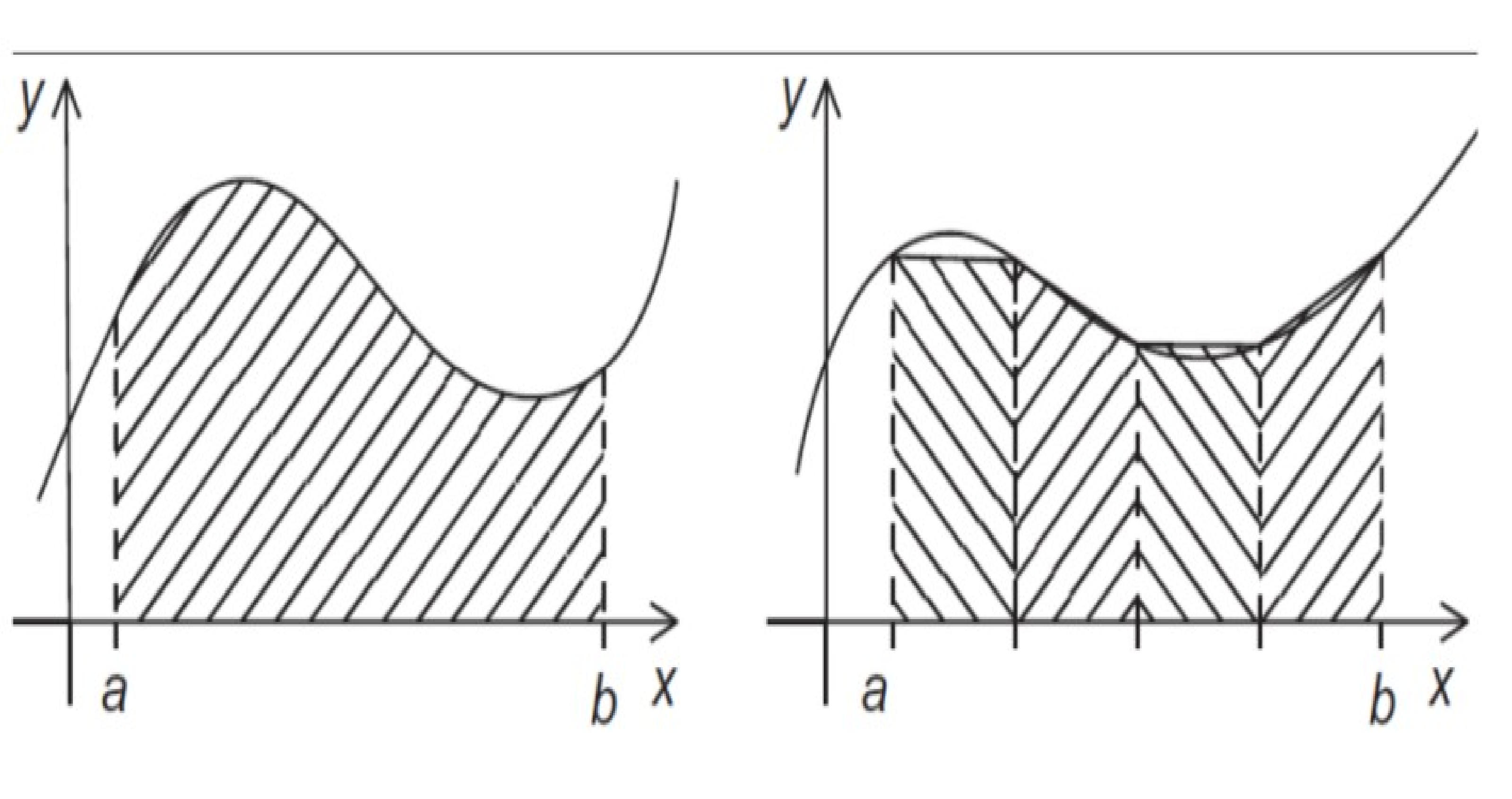
```
# include <omp.h>
```

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

# In case the compiler doesn’t support OpenMP

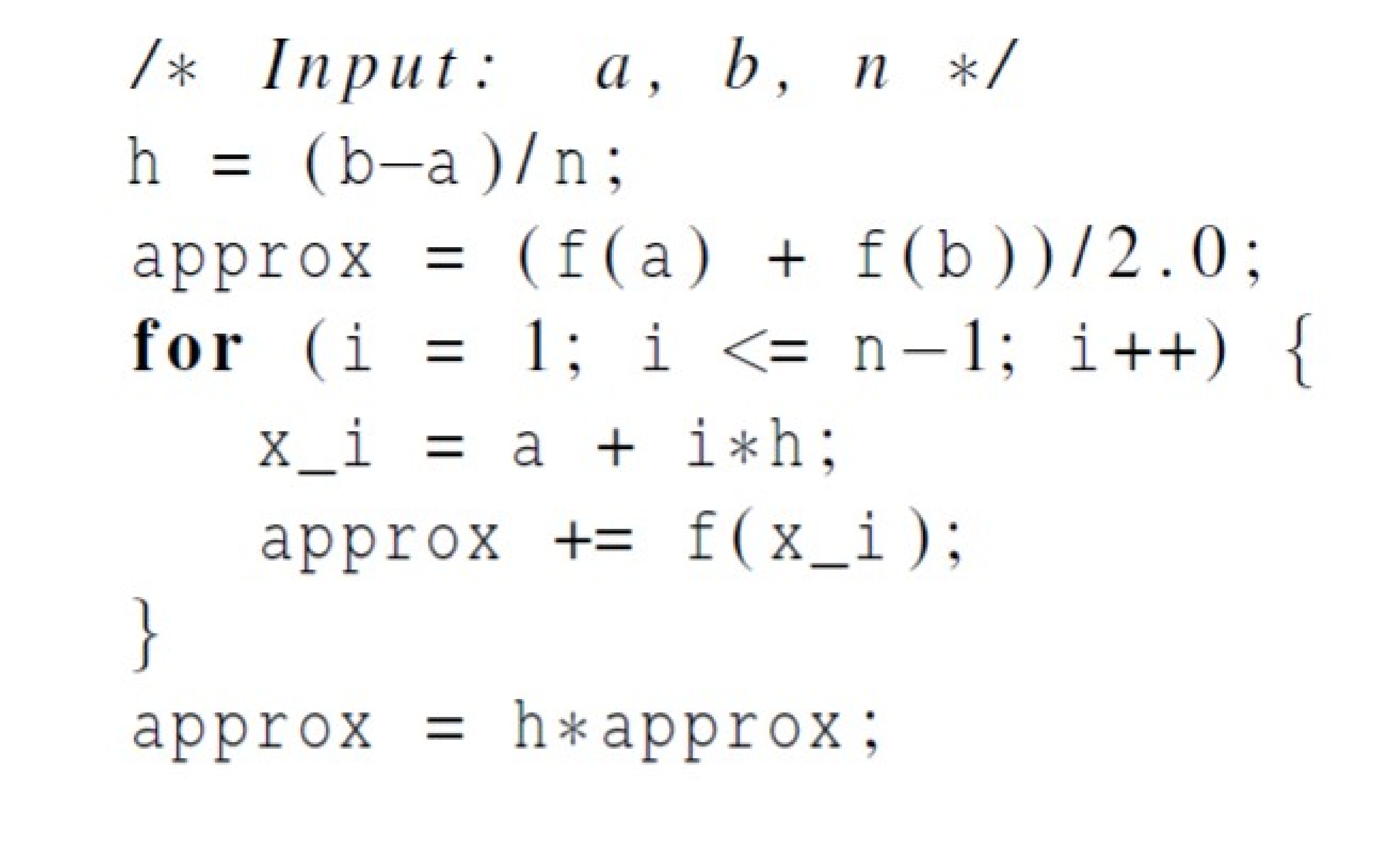
```
# ifdef _OPENMP
    int my_rank = omp_get_thread_num ( );
    int thread_count = omp_get_num_threads ( );
# else
    int my_rank = 0;
    int thread_count = 1;
# endif
```

# The trapezoidal rule



The approximate area under the curve is found by adding the area of the trapezoids.

# 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 **[different operations in a single iteration]**, 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.

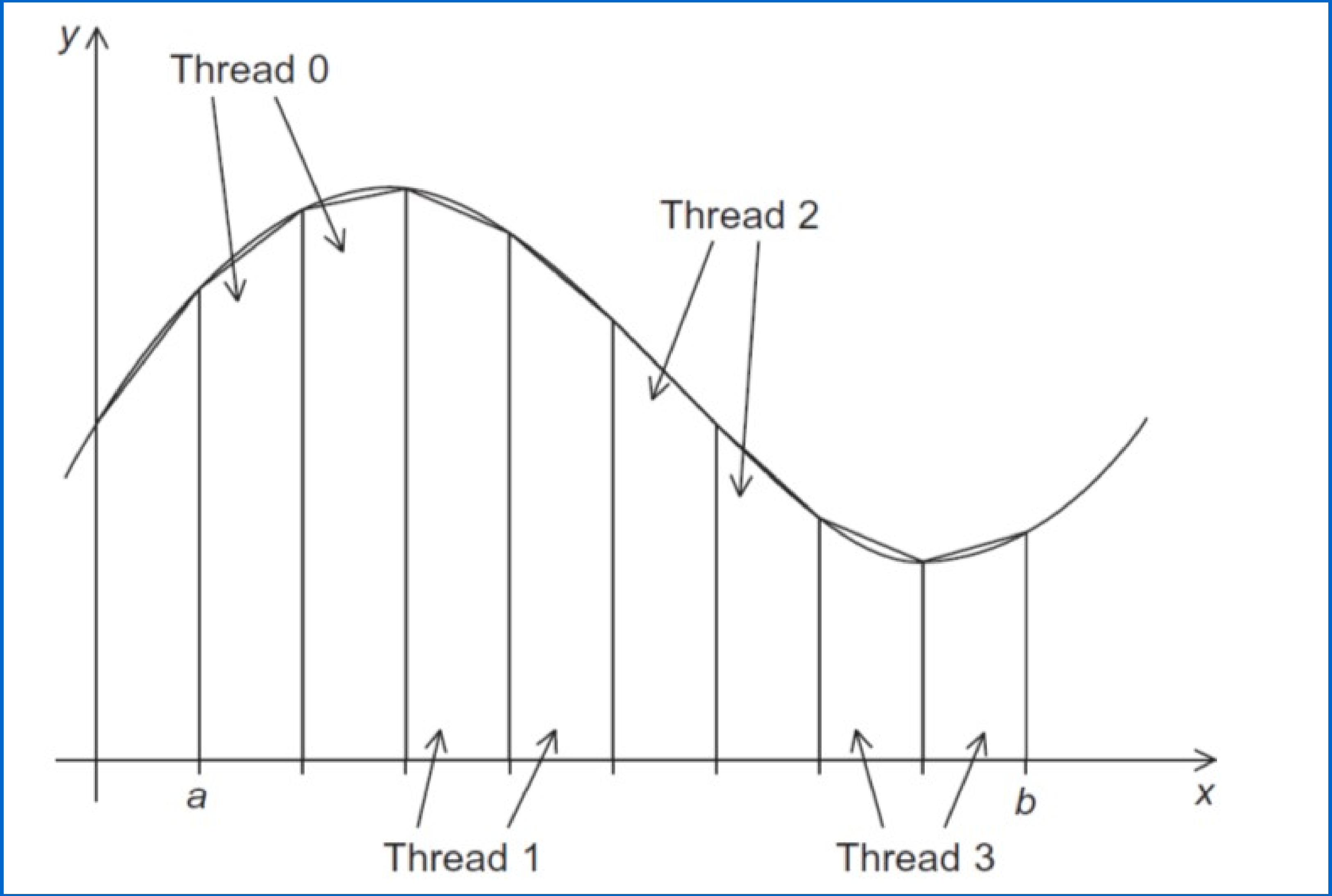
# A First OpenMP Version

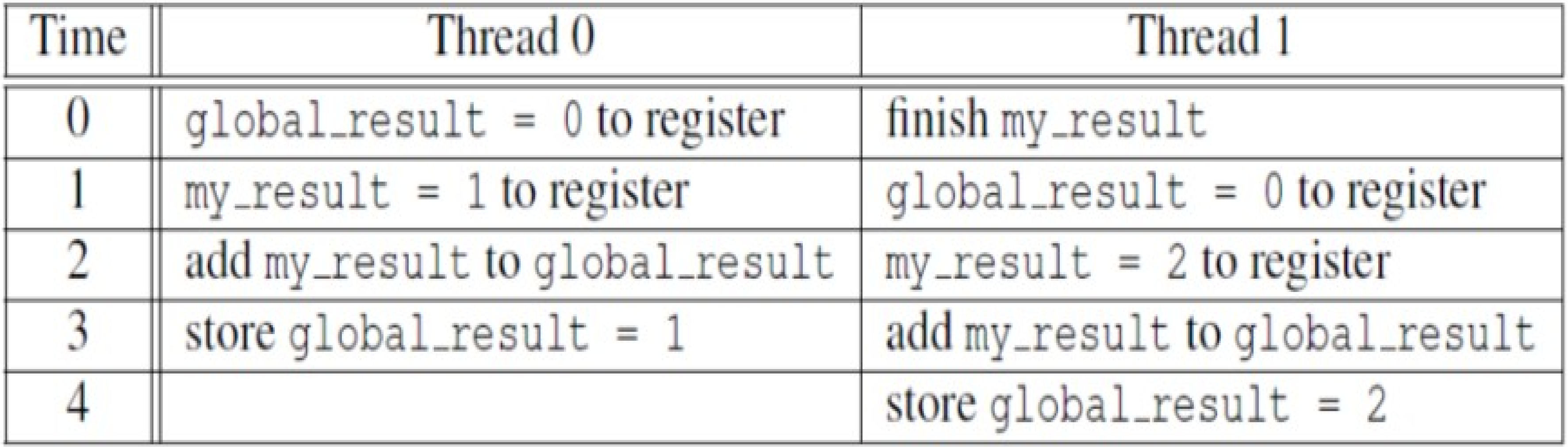
3) We assumed that there would be many

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

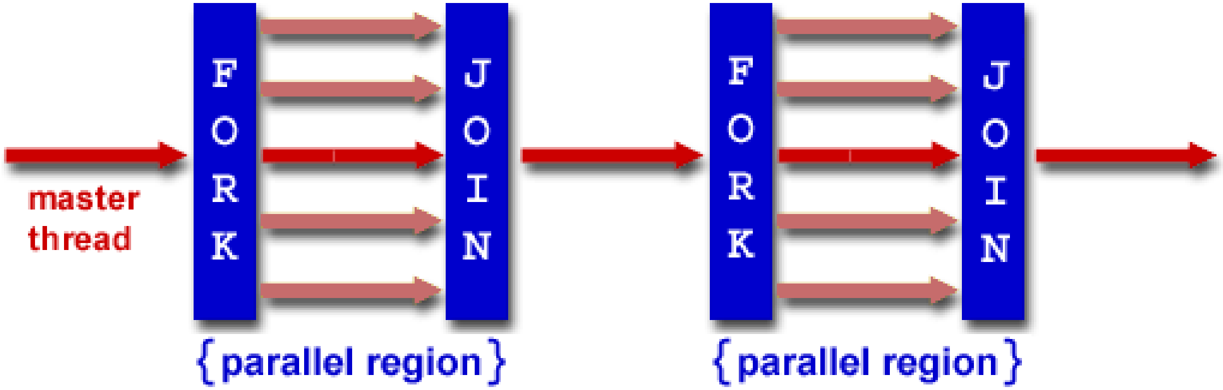
```
# pragma omp critical
global_result += my_result ;
```

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

# How to compile and run OpenMP programs?

- Gcc 4.2 and above supports OpenMP 3.0
  - gcc –fopenmp a.c
  - Try example1.c
- To run: ‘a.out’
  - To change the number of threads:
    - setenv OMP_NUM_THREADS 4 (tcsh) or export OMP_NUM_THREADS=4(bash)

# OpenMP execution model



- OpenMP uses the **fork-join model** of parallel execution.
  - All OpenMP programs begin with a single master thread.
  - The master thread executes sequentially until a parallel region is encountered, when it creates a team of parallel threads (FORK).
  - When the team threads complete the parallel region, they synchronize and terminate, leaving only the master thread that executes sequentially (JOIN).

# OpenMP general code structure

```
#include <omp.h>
main () {
   int var1, var2, var3;
   Serial code
   . . .
  /* Beginning of parallel section. Fork a team of threads. Specify
     variable scoping*/
   #pragma omp parallel private(var1, var2) shared(var3)
   {
      /* Parallel section executed by all threads */
     . . .
     /* All threads join master thread and disband*/
    }
    Resume serial 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.

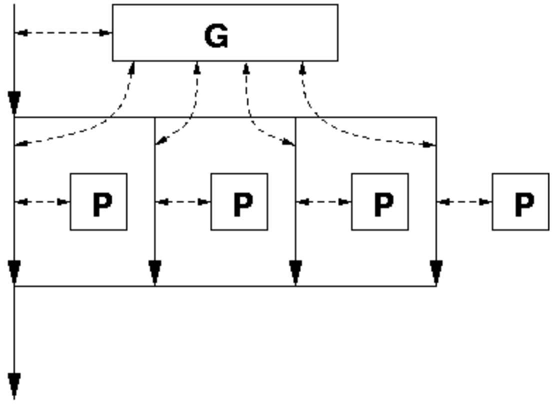
# Scope in OpenMP

- 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**.

# Data model

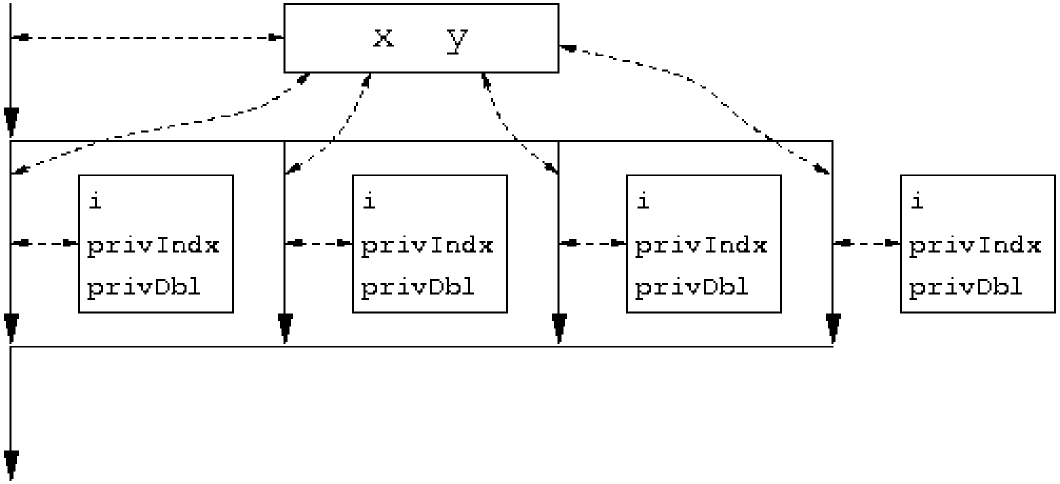


P = private data space
G = global data space

- Private and shared variables
  - Variables in the global data space are accessed by all parallel threads (shared variables).
  - Variables in a thread's private space can only be accessed by the thread (private variables)
    - several variations, depending on the initial values and whether the results are copied outside the region.

```
#pragma omp parallel for private( privIndx,
    privDbl )
 for ( i = 0; i < arraySize; i++ ) {
    for ( privIndx = 0; privIndx < 16; privIndx
    ++ ) { privDbl = ( (double) privIndx ) / 16;
     y[i] = sin( exp( cos( - exp( sin(x[i]) ) ) ) )
    + cos( privDbl );
   }
}
```

Parallel for loop index is Private by default.

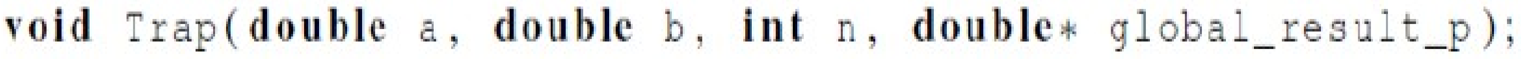


execution context for "arrayUpdate_II"

# Reduction Clause

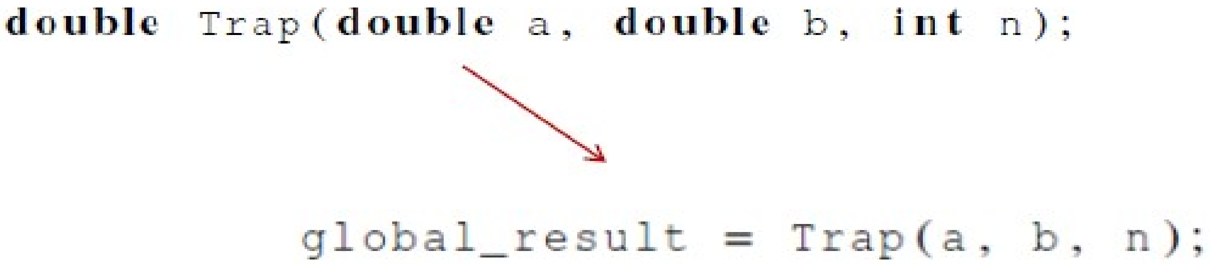
# Reduction Clause

**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);
```

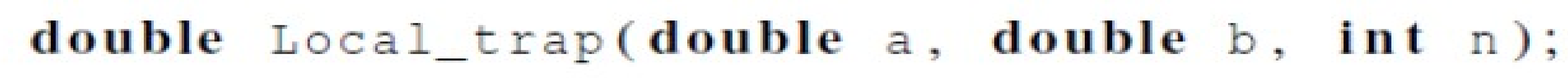
**Although we’d prefer this.**



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

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

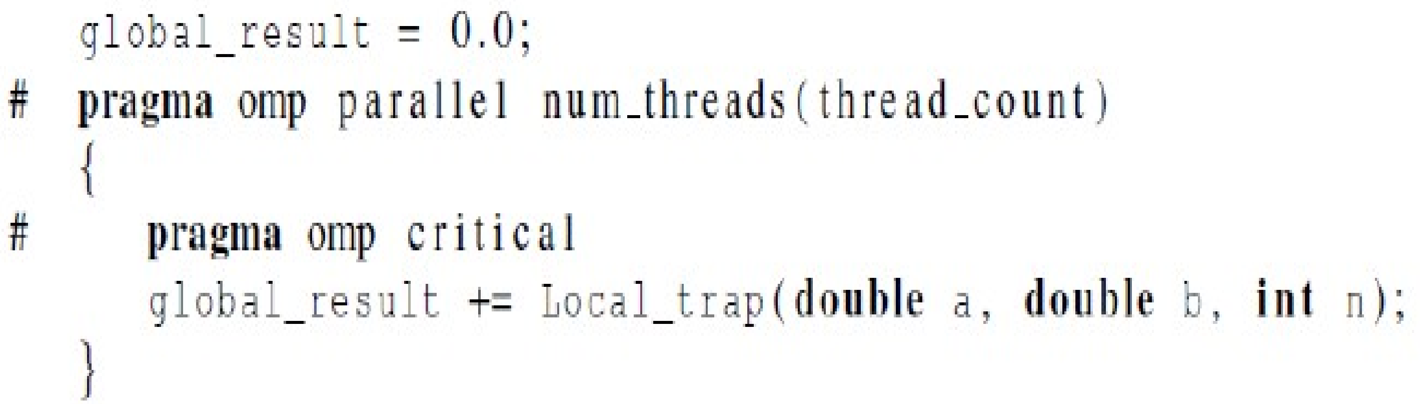
# Reduction Clause

**If we use this, there’s no critical section!**



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

**If we fix it like this…**

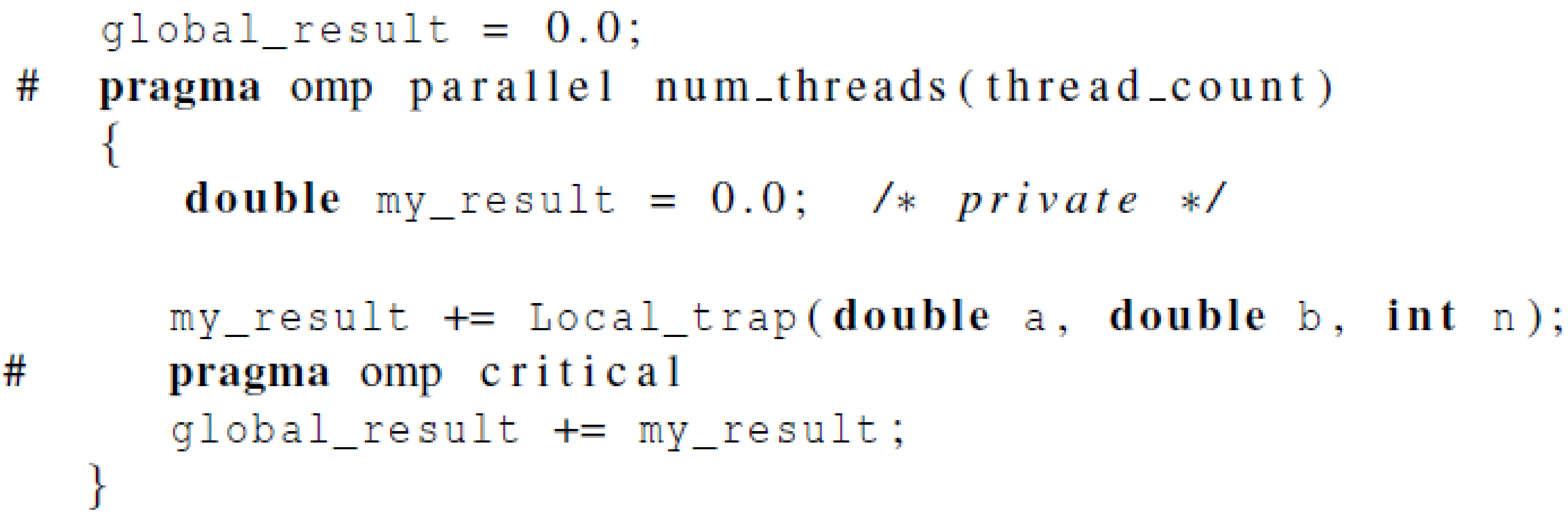


```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count)
   {
#     pragma omp critical
      global_result += Local_trap(double a, double b, int n);
   }
```

**… we force the threads to execute sequentially.**

# Reduction Clause

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



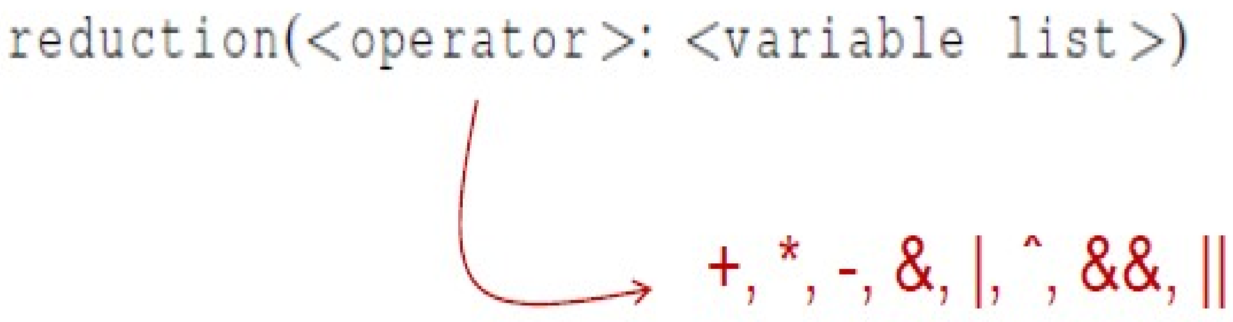
```
    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count)
    {
        double my_result = 0.0;  /* private */

        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.

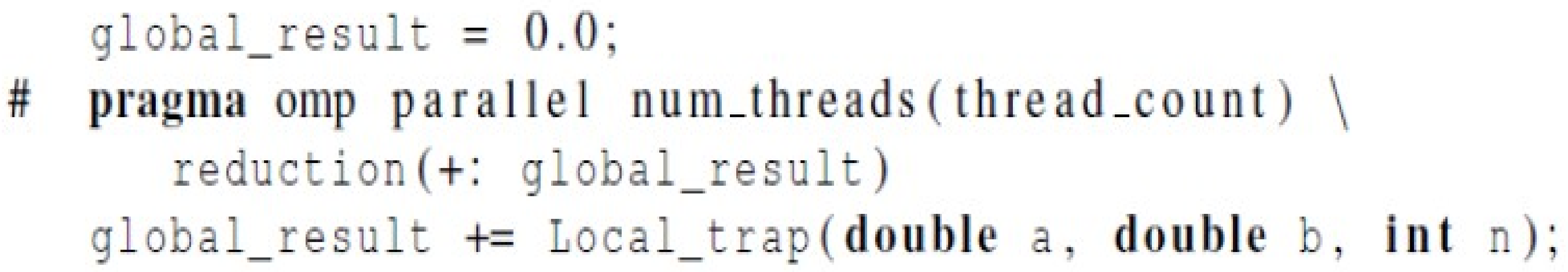
# Reduction Operators

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);
```

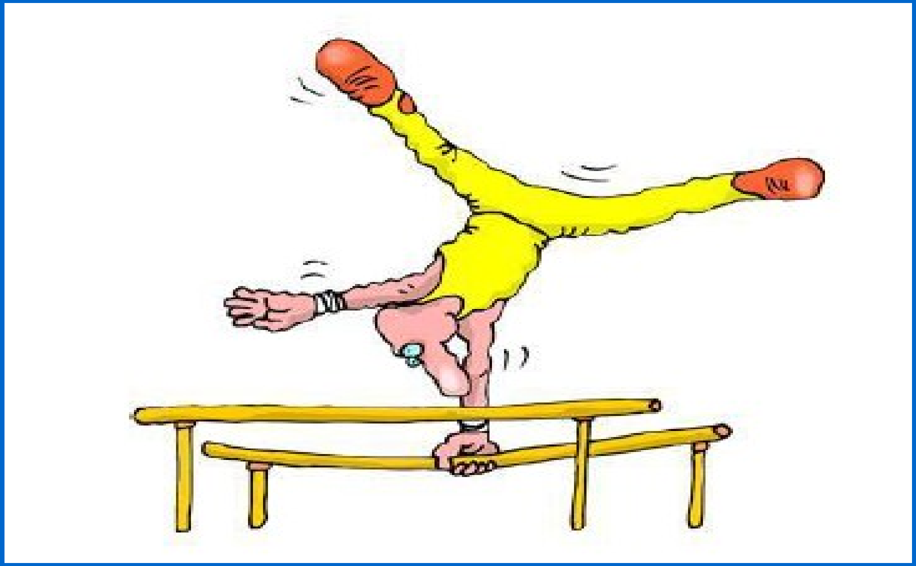
# OpenMP directives

- Format:

  #pragma omp directive-name [clause,..] newline

  (use ‘\’ for multiple lines)
- Example:

  ```
  #pragma omp parallel default(shared) private(beta,pi)
  ```
- Scope of a directive is one block of statements { …}

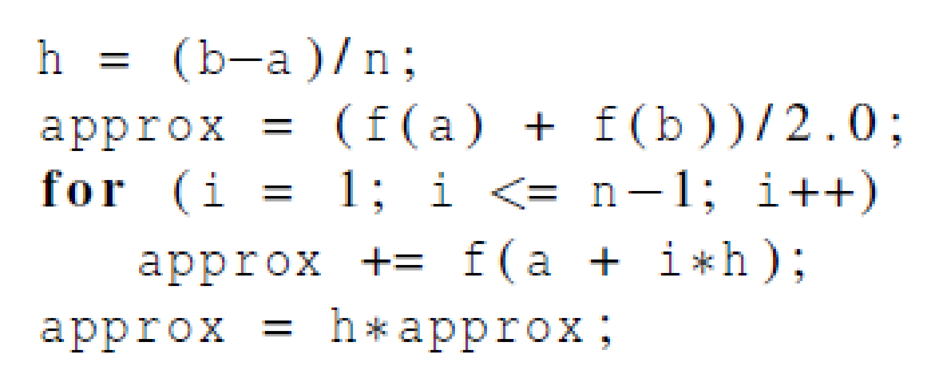


# THE “PARALLEL FOR” DIRECTIVE

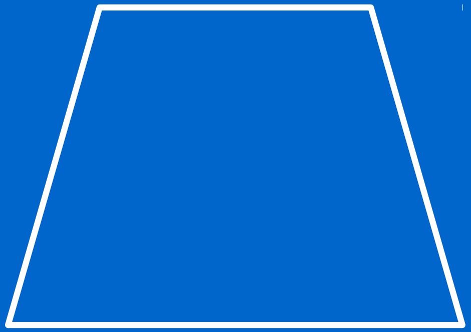
# Parallel for

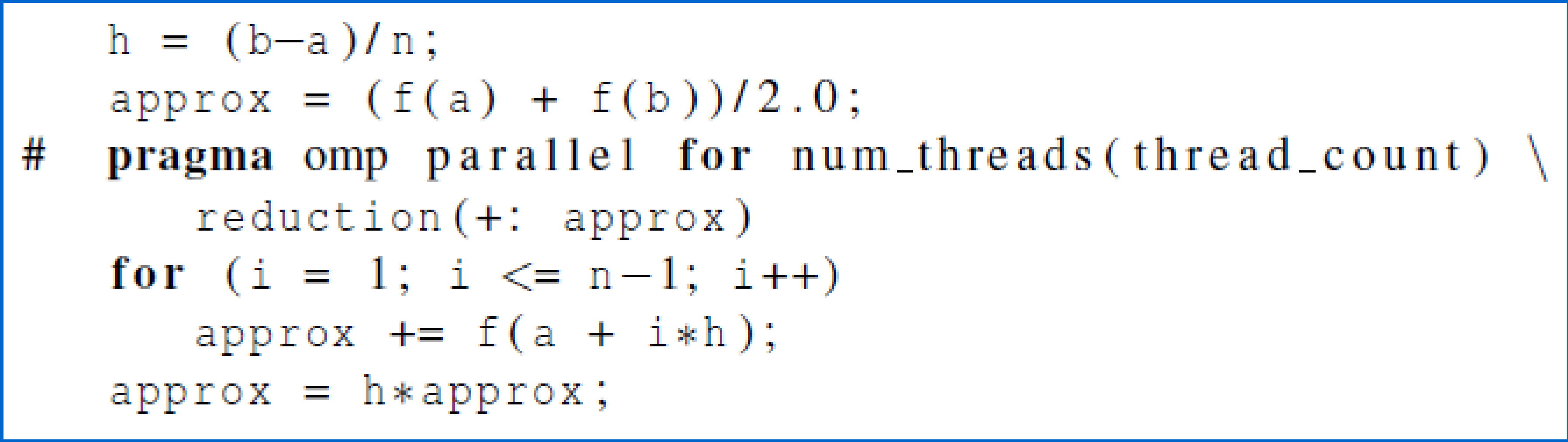
- Forks a team of threads to execute the following structured block.
- However, the structured block following the parallel for directive must be a for loop.
- Furthermore, with the parallel for directive the system parallelizes the for loop by dividing the iterations of the loop among the threads.

# Parallel for



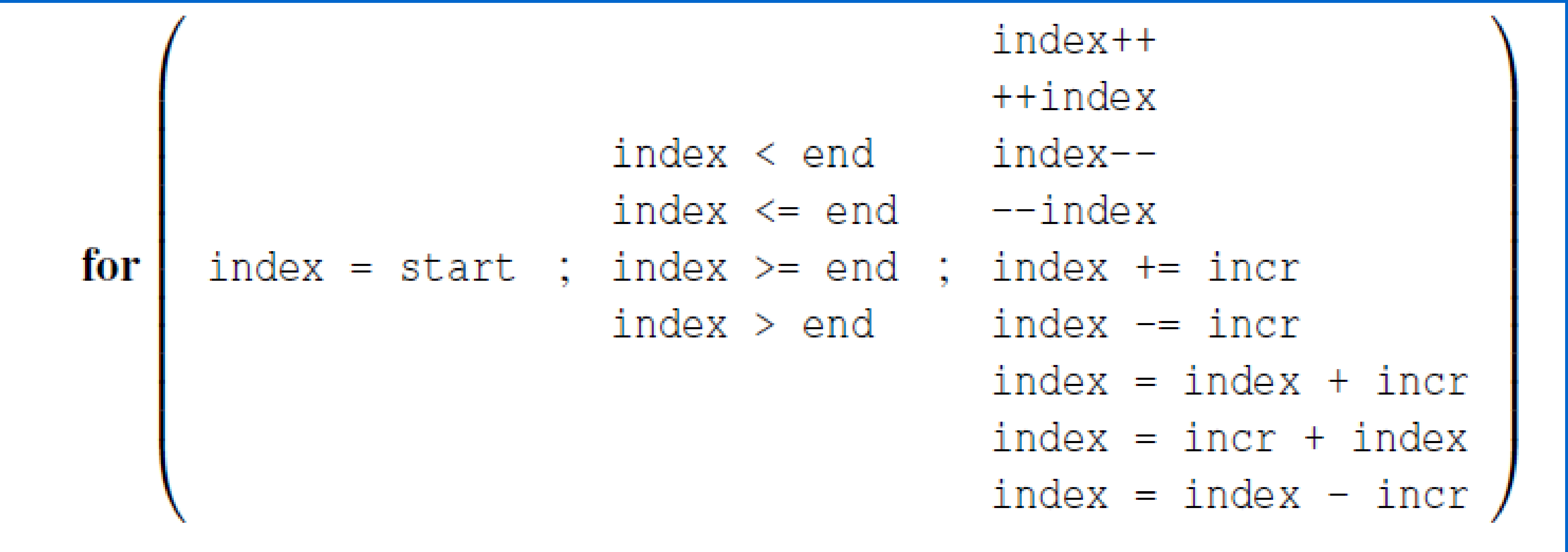
```
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++)
   approx += f(a + i*h);
approx = h*approx;
```





```
    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



```
                                         index++
                                         ++index
                    index < end          index--
                    index <= end         --index
for ( index = start ; index >= end ;     index += incr         )
                    index > end          index -= incr
                                         index = index + incr
                                         index = incr + index
                                         index = index - incr
```

# Caveats

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

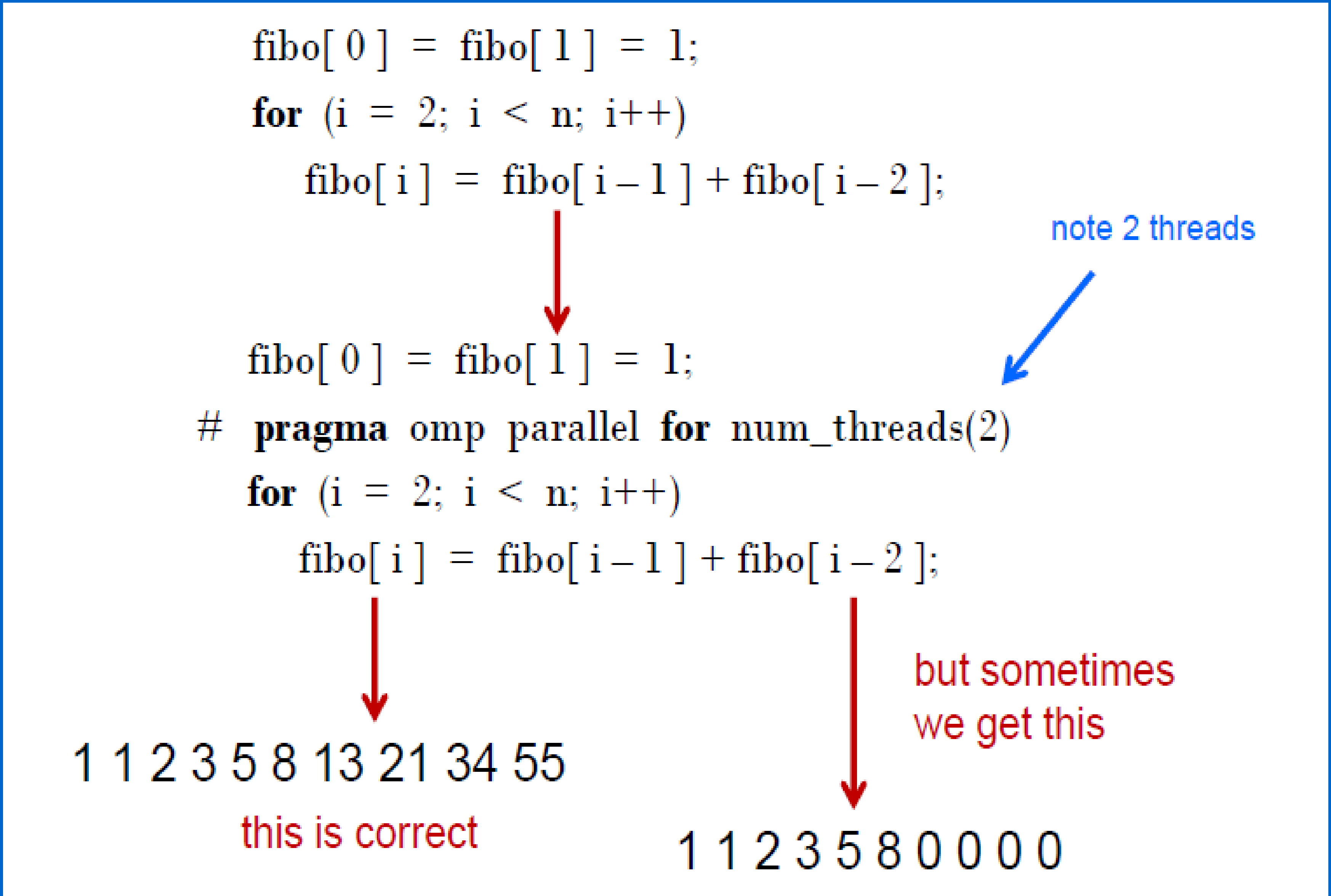
- The expressions **start**, **end**, and **incr** must have a compatible type. For example, if index is a pointer, then **incr** must have integer type.

# 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 ];
```

1 1 2 3 5 8 13 21 34 55

this is correct

but sometimes we get this

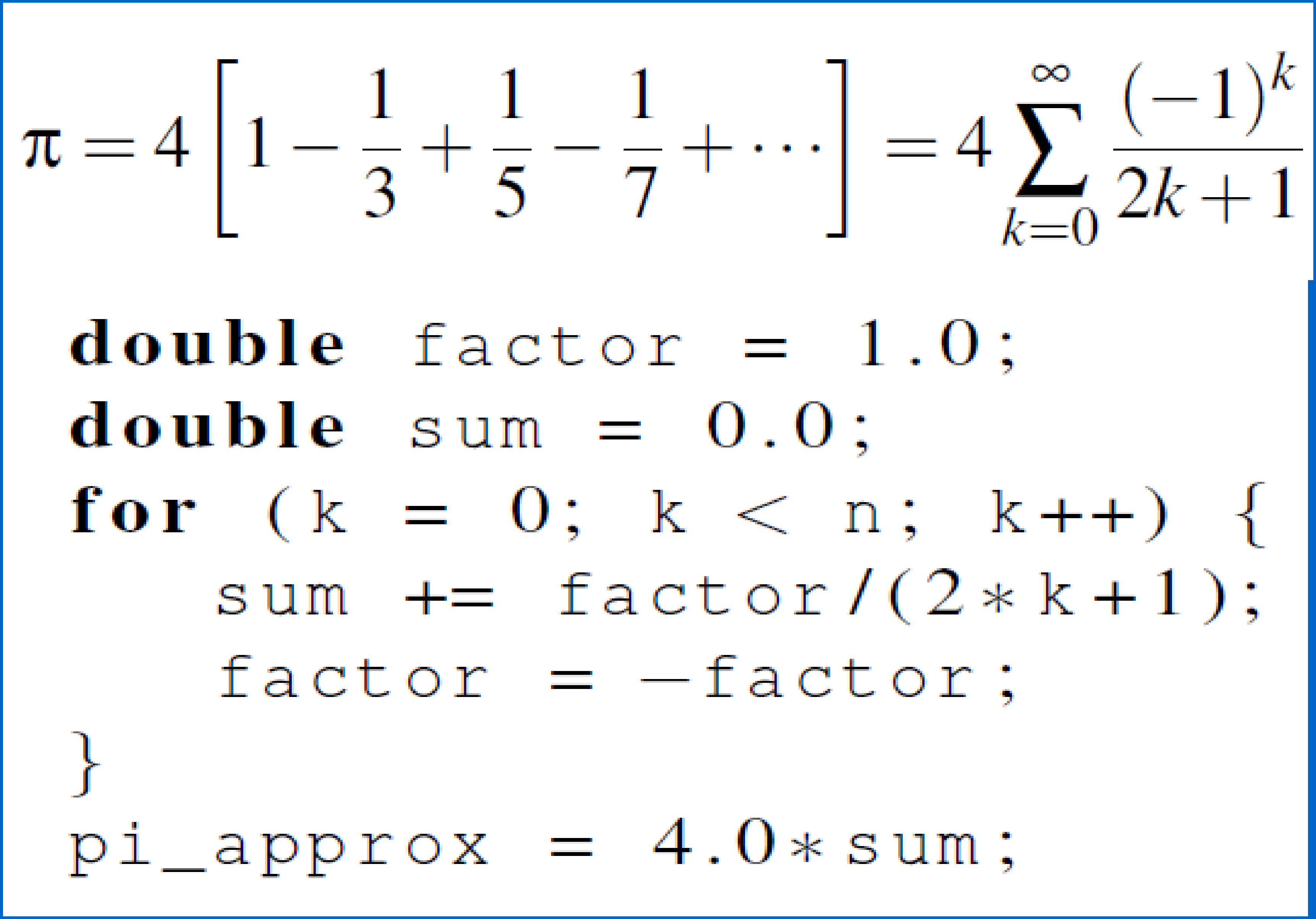
1 1 2 3 5 8 0 0 0 0

# 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.

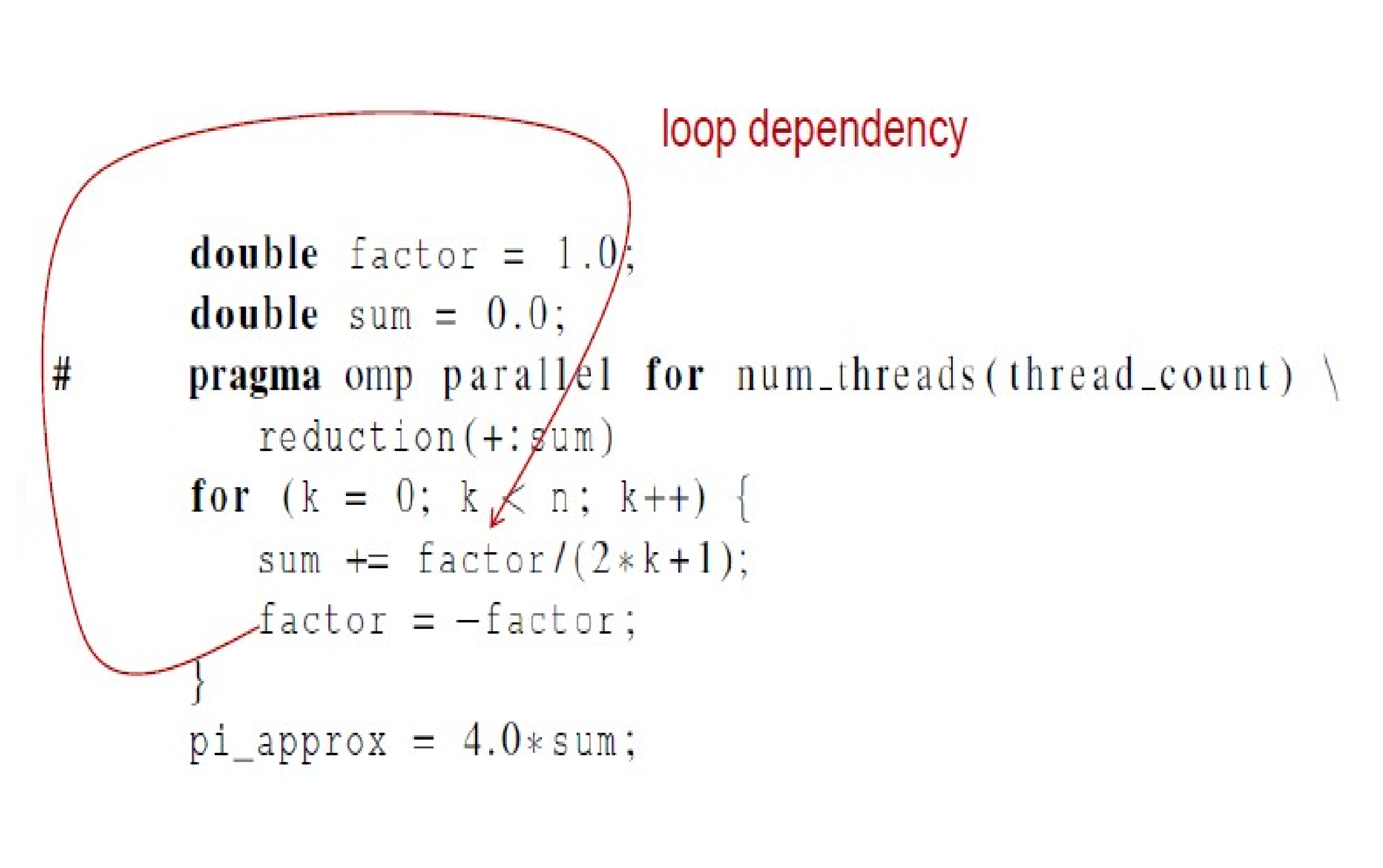
# Estimating π (For Eg:)



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

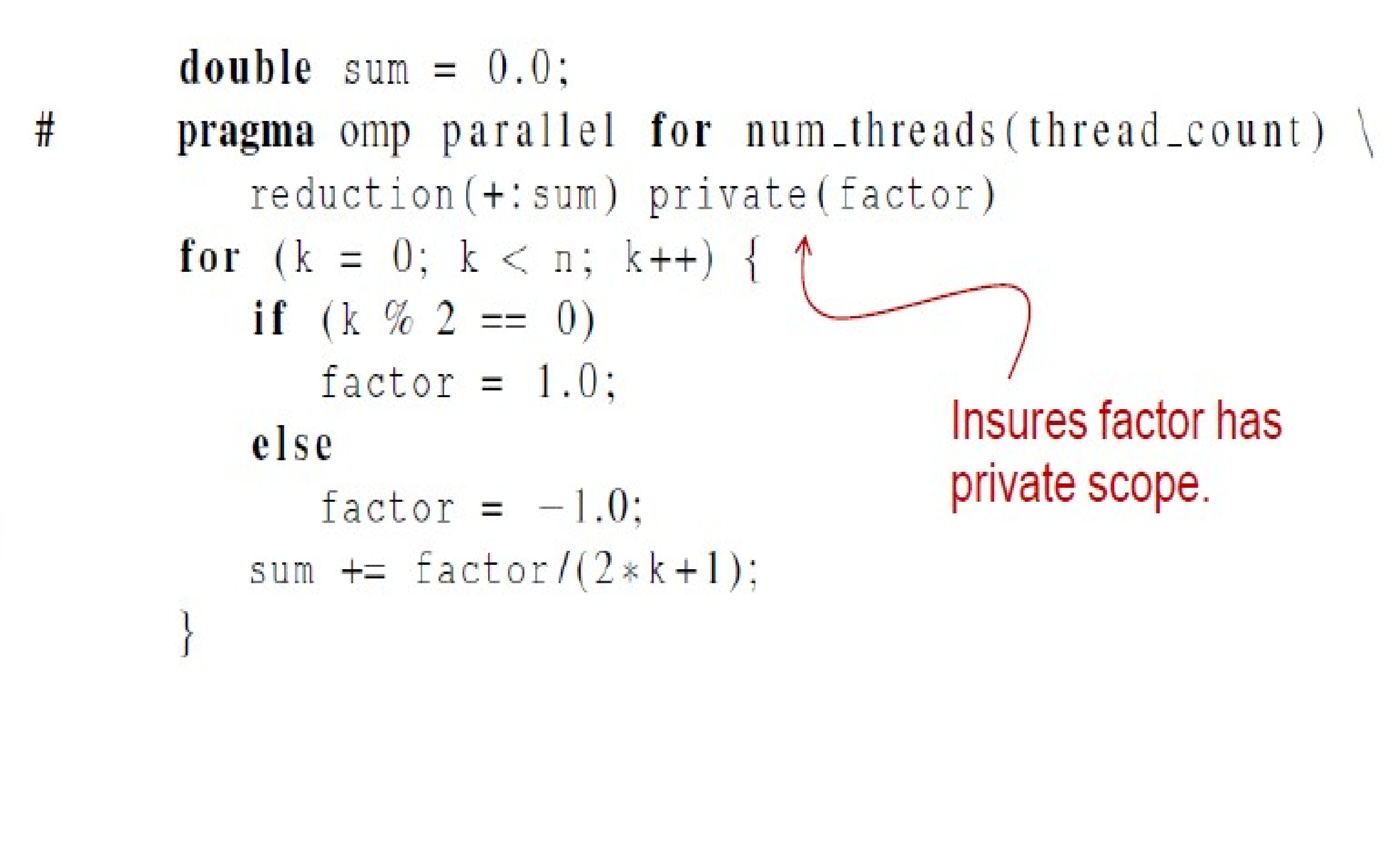
```
double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++) {
    sum += factor/(2*k+1);
    factor = −factor;
}
pi_approx = 4.0*sum;
```

# 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

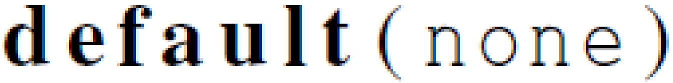


```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum) private(factor)
        for (k = 0; k < n; k++) {
           if (k % 2 == 0)
              factor = 1.0;
           else
              factor = −1.0;
           sum += factor/(2*k+1);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

Insures factor has private scope.

# The default clause

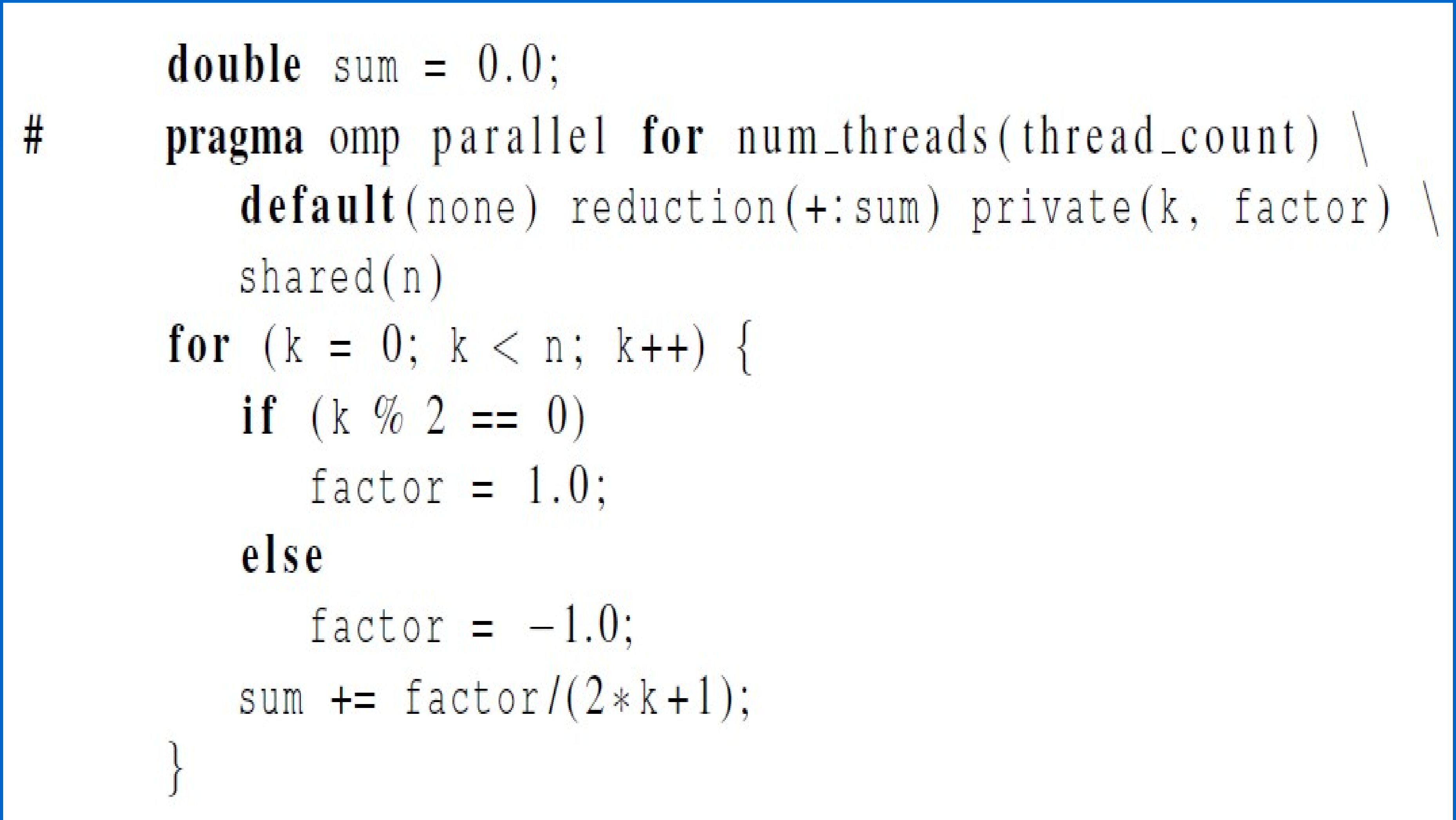
Lets the programmer specify the scope of each variable in a block.



```
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);
   }
```

# Forking Pthreads

**Signature:**

```
    int pthread_create(pthread_t *,
                                const pthread_attr_t *,
                                void * (*)(void *),
                                void *);
```

**Example call:**

```
  errcode = pthread_create(&thread_id,
   &thread_attribute,
                                  &thread_fun, &fun_arg);
```

- thread_id  is the thread id or handle (used to halt, etc.)
- thread_attribute various attributes
  - standard default values obtained by passing a NULL pointer
- thread_fun the function to be run (takes and returns void*)
- fun_arg an argument can be passed to thread_fun when it starts
- errorcode will be set to nonzero if the create operation fails

# Forking Pthreads, cont.

The effect of pthread_create

- Master thread actually causes the operating system to create a new thread
  - Each thread executes a specific function, thread_fun
    - The same thread function is ex ecuted by all threads that are created, representing the thread's *computation decomposition*
- For the program to perform different work `in different threads, the arguments passed at thread creation distinguish the thread's "id" and any other unique features of the thread.

# Simple Threading Example

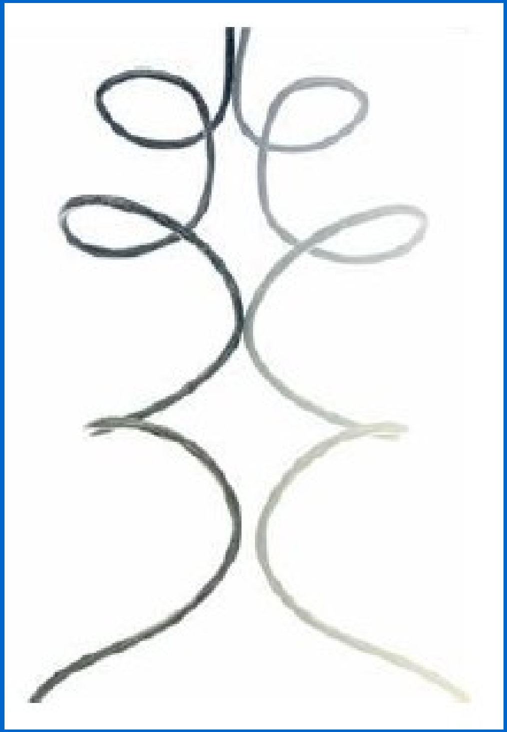
```
int main() {
  pthread_t threads[16];
  int tn;
  for(tn=0; tn<16; tn++) {
    pthread_create(&threads[tn], NULL, ParFun, NULL);
  }
  for(tn=0; tn<16 ; tn++) {
    pthread_join(threads[tn], NULL);
  }
  return 0;
}
```

Compile using gcc … –lpthread

This code creates 16 threads that execute the function “ParFun”.

Note that thread creation is costly, so it is important that ParFun do a lot of work in parallel to amortize this cost.

Slide source: Jim Demmel and Kathy Yelick

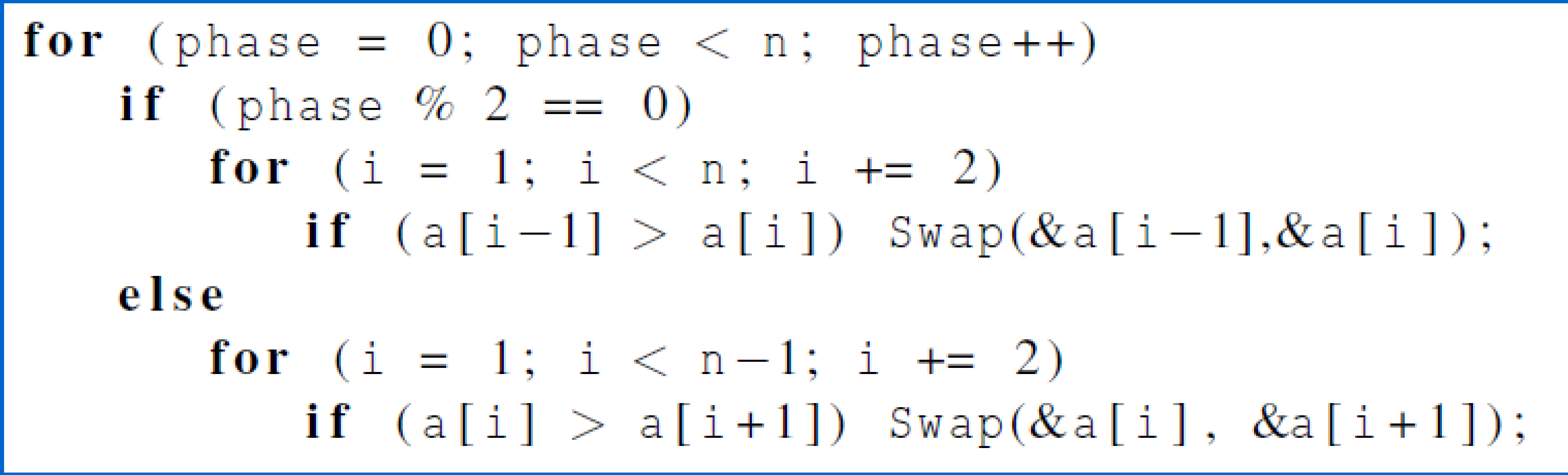


# MORE ABOUT LOOPS IN OPENMP: SORTING

# Bubble Sort

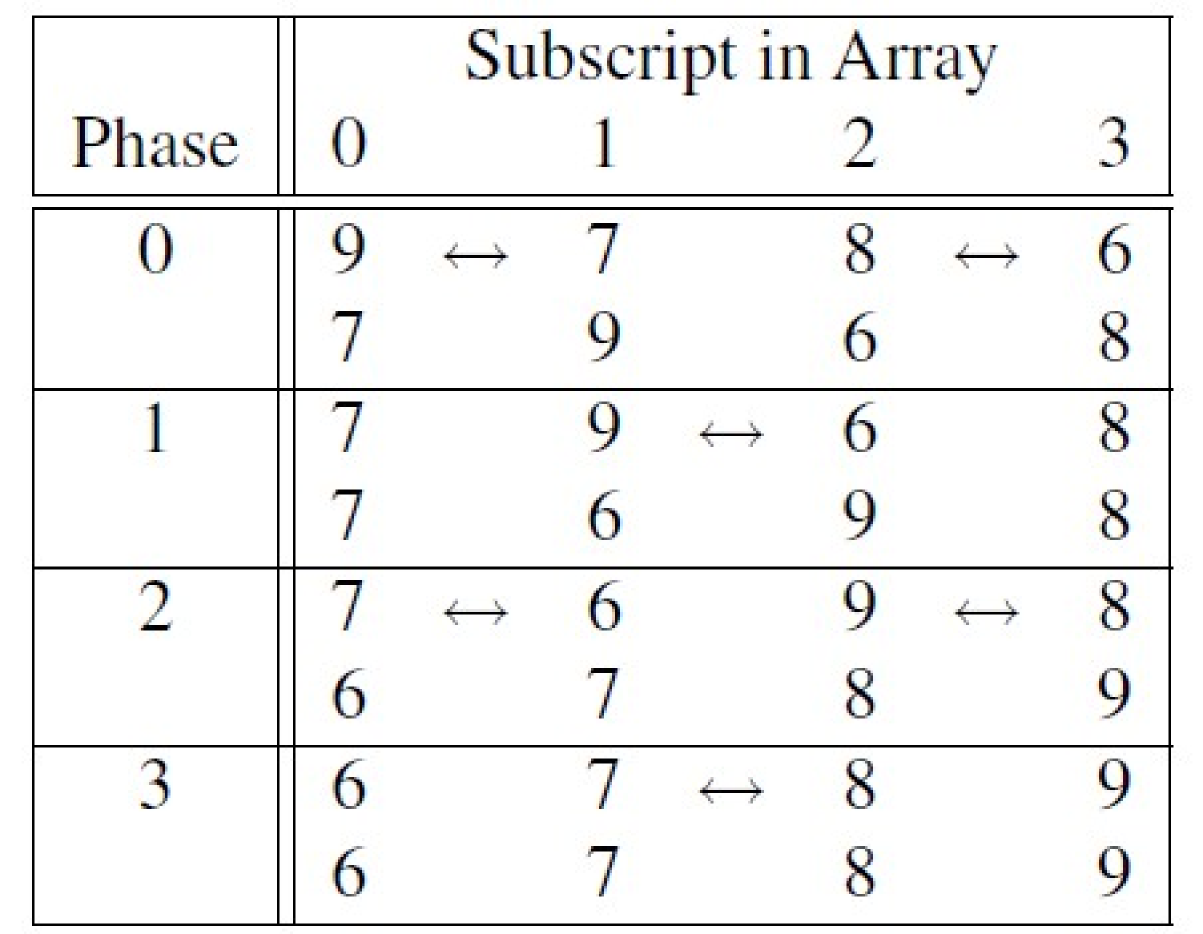
```
for (list_length = n; list_length >= 2; list_length−−)
   for (i = 0; i < list_length−1; i++)
      if (a[i] > a[i+1]) {
         tmp = a[i];
         a[i] = a[i+1];
         a[i+1] = tmp;
      }
```

# Serial Odd-Even Transposition Sort



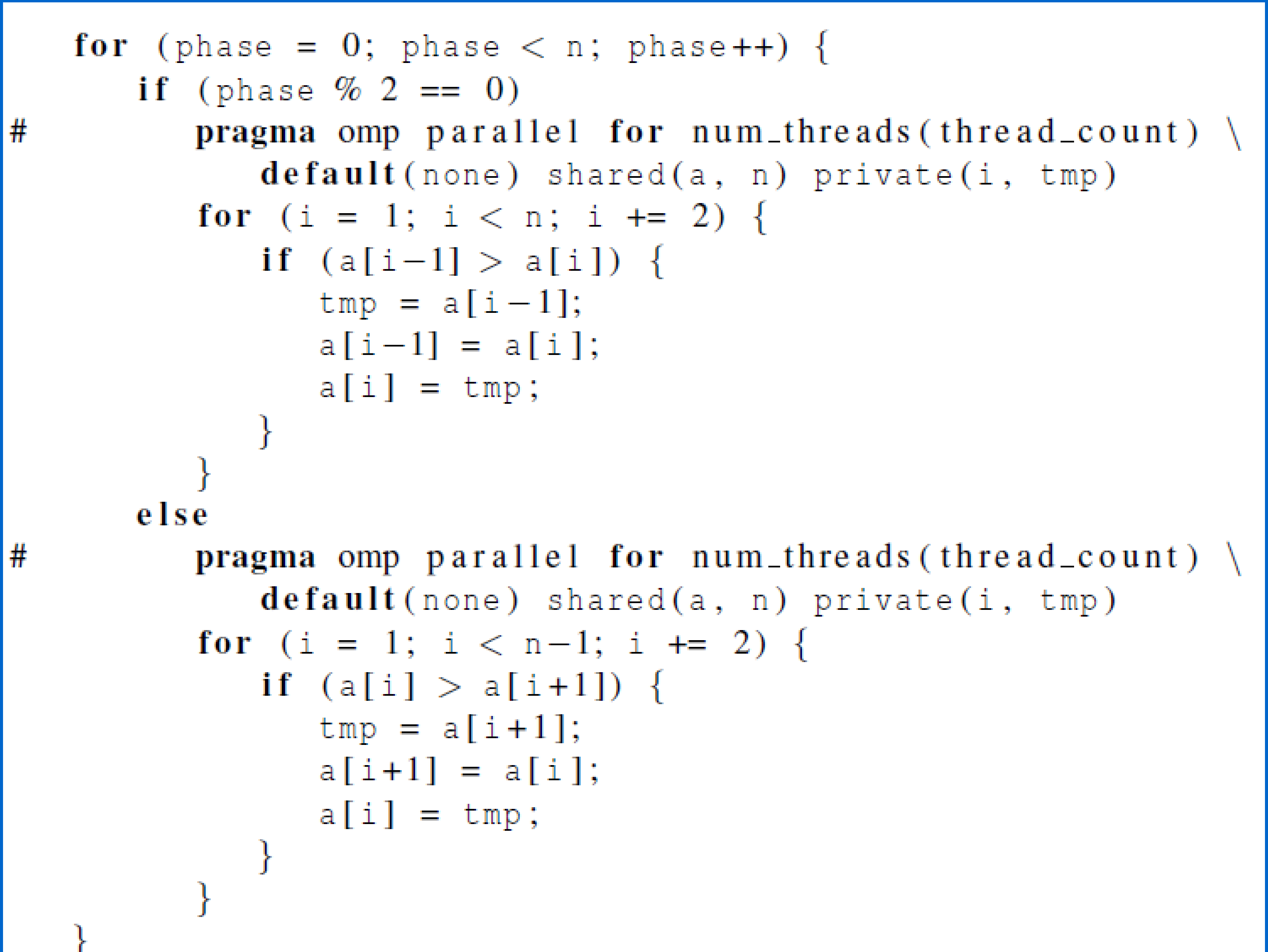
```
for (phase = 0; phase < n; phase++)
   if (phase % 2 == 0)
      for (i = 1; i < n; i += 2)
         if (a[i-1] > a[i]) Swap(&a[i-1],&a[i]);
   else
      for (i = 1; i < n-1; i += 2)
         if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

# Serial Odd-Even Transposition Sort



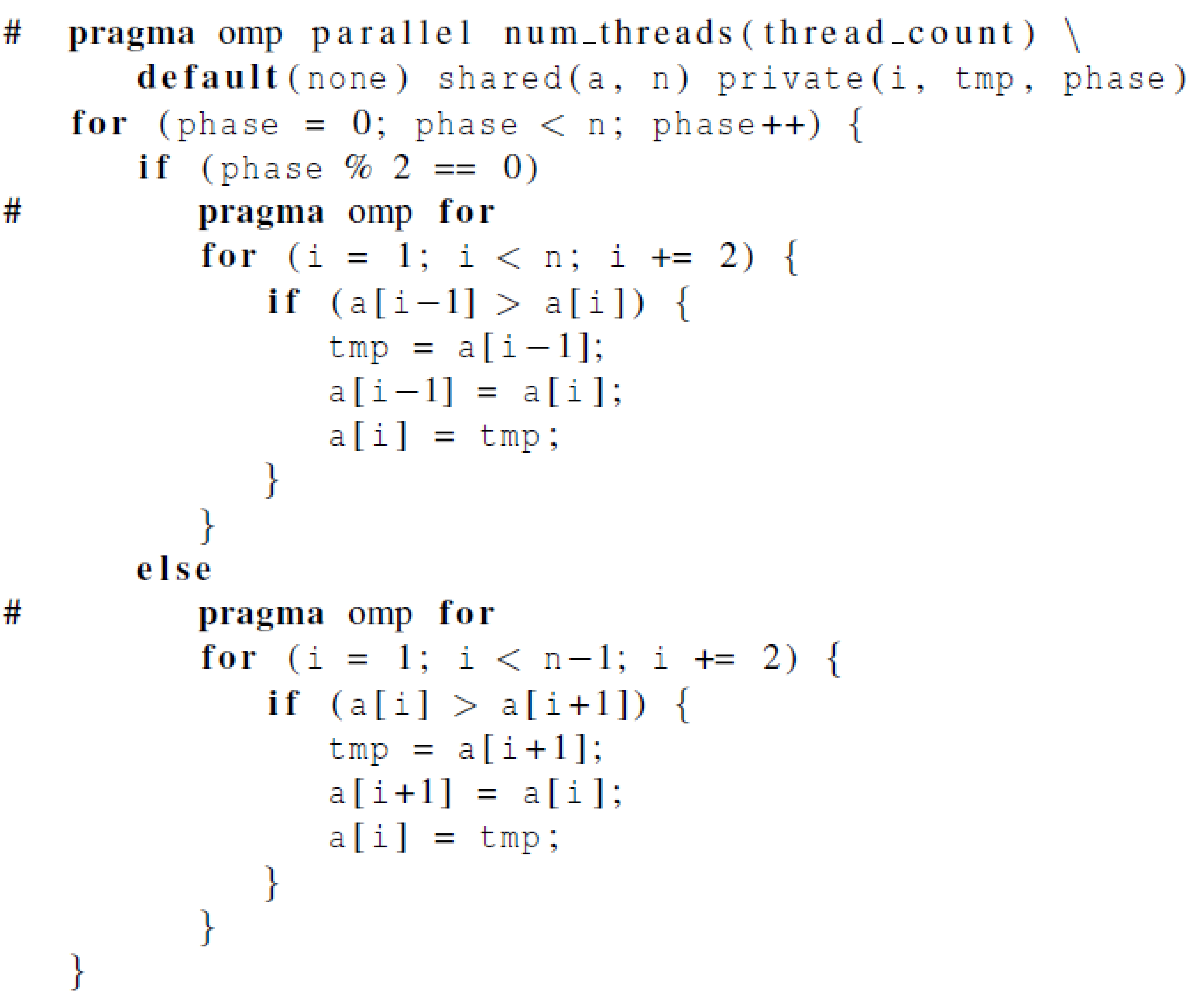
|  | Subscript in Array |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| Phase | 0 |  | 1 |  | 2 |  | 3 |
| 0 | 9 | ↔ | 7 |  | 8 | ↔ | 6 |
|  | 7 |  | 9 |  | 6 |  | 8 |
| 1 | 7 |  | 9 | ↔ | 6 |  | 8 |
|  | 7 |  | 6 |  | 9 |  | 8 |
| 2 | 7 | ↔ | 6 |  | 9 | ↔ | 8 |
|  | 6 |  | 7 |  | 8 |  | 9 |
| 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;
            }
         }
   }
```

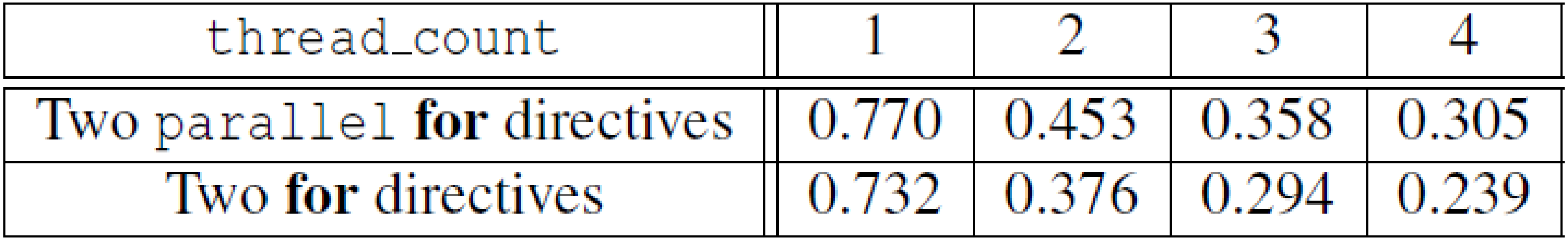
# Second OpenMP Odd-Even Sort



```
#  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;
            }
         }
   }
```

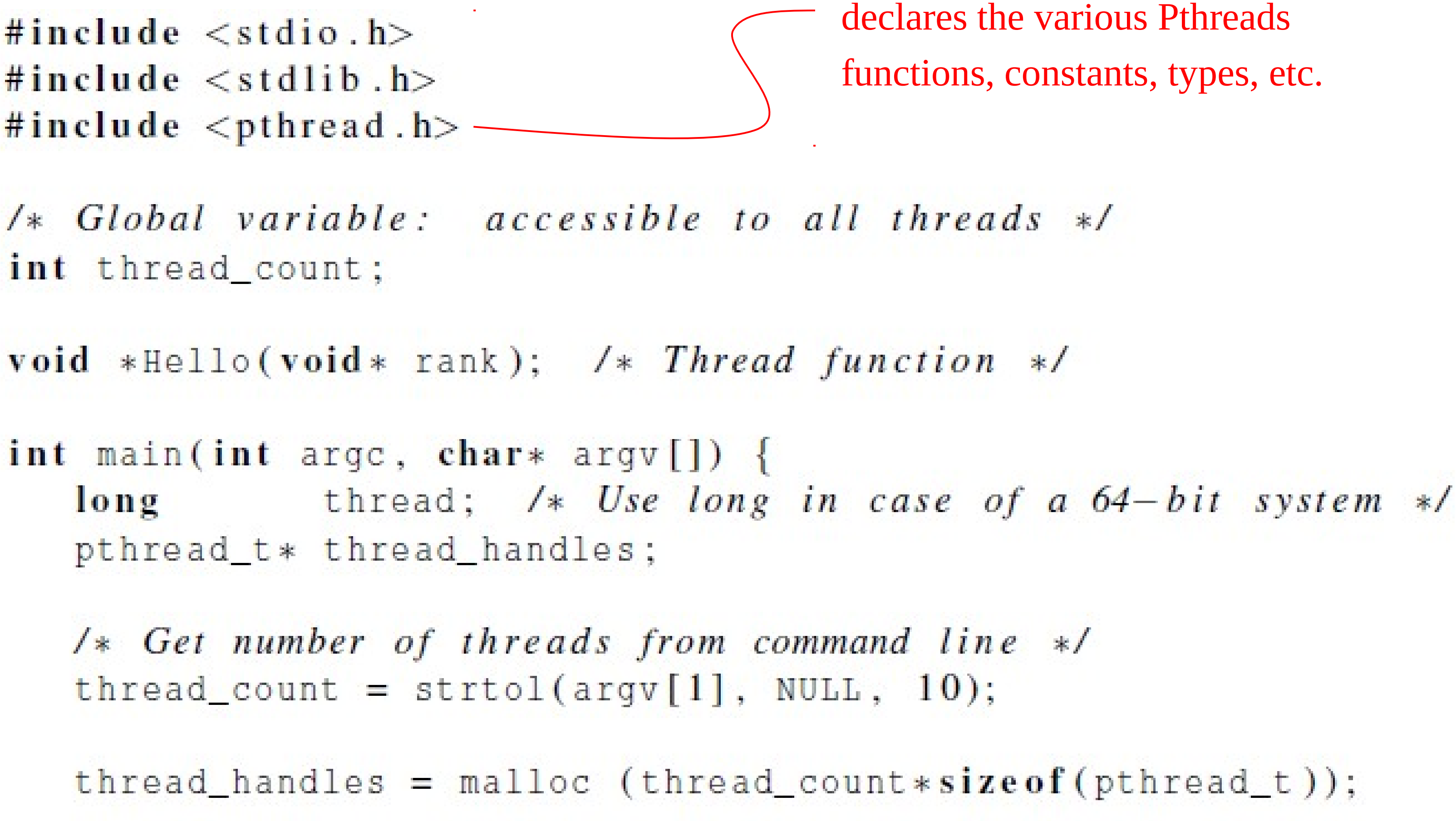
Odd-even sort with two parallel for directives and two for directives.

(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 |

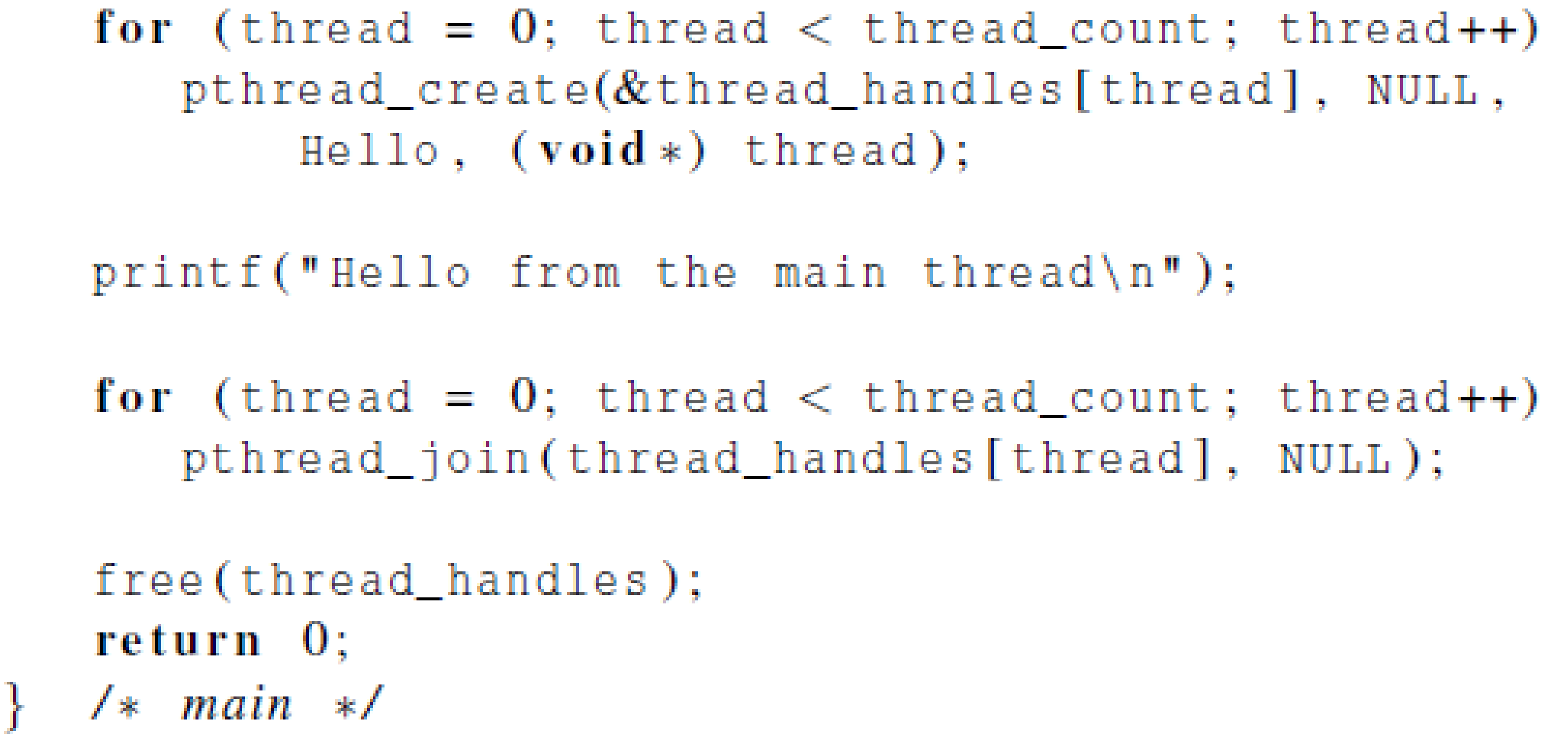
# Hello World! (1)



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

/* Global variable:  accessible to all threads */
int thread_count;

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

int main(int argc, char* argv[]) {
   long       thread;  /* Use long in case of a 64-bit system */
   pthread_t* thread_handles;

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

   thread_handles = malloc (thread_count*sizeof(pthread_t));
```

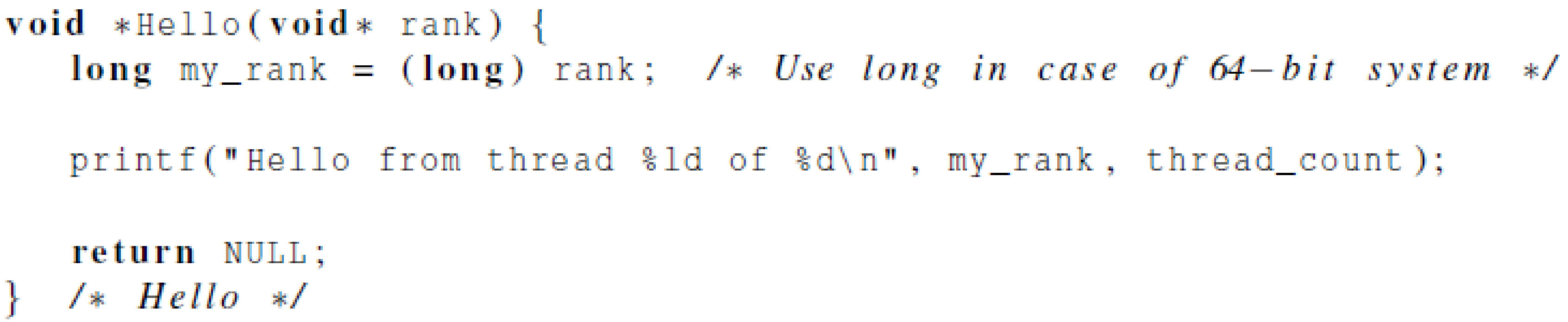
declares the various Pthreads functions, constants, types, etc.

# Hello World! (2)



```
   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,
          Hello, (void*) thread);

   printf("Hello from the main thread\n");

   for (thread = 0; thread < thread_count; thread++)
      pthread_join(thread_handles[thread], NULL);

   free(thread_handles);
   return 0;
}  /* main */
```

# Hello World! (3)



```
void *Hello(void* rank) {
   long my_rank = (long) rank;  /* Use long in case of 64−bit system */

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

   return NULL;
}  /* Hello */
```

# References

1.www.cs.fsu.edu/~xyuan/cda5125/lect9_openmp.ppt

2.http://eclass.uoa.gr/modules/document/file.php/D186/%CE%8E%CE%BB%CE%B7%202011-12/PachecoChapter_5.pdf

3. www.cs.utah.edu/~mhall/cs4961f11/CS4961-L5.ppt