# DTS202TC Foundation of Parallel Computing

## Lecture 4 OpenMP

Hong-Bin Liu

Xi'an Jiaotong-Liverpool University

September 27, 2021

# Group Assessment 1 realeased



- 20%, due October 10, 23:59pm
- 60 group marks + 40 individual marks
- Work in groups, discuss with your peers, every member should make contributions.
- One group member submit the group work.
- Peer review portal will be opened later.

# Week 5 CUDA



- 6 hours face-to-face lectures + lab (live streaming provided in case you are not in Suzhou)
- Mon. 11$^{th}$ Oct. 9-11am
- Tue. 12$^{th}$ Oct. 4-6pm
- Thurs. 14$^{th}$ Oct. 11am-1pm
- No recording due to the copyright

# This week's topic



- OpenMP Basics
  - Our First OpenMP Program
  - Fundamental Concepts and Library Functions
- The Trapezoidal Rule
- Scope of Variables
- Task Parallelism
  - The Reduction Clause
  - The `parallel for` Directive

# 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.

# OpenMP vs. Pthreads



- Pthreads requires that the programmer explicitly specify the behavior of each thread. OpenMP allows the compiler and run-time system to determine some of the details of thread behavior.
- Any Pthreads program can be used with any C compiler, provided the system has a Pthreads library. OpenMP 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.
- Pthreads is lower level. Cost: Specify every detail of the behavior of each thread. OpenMP can be simpler to code some parallel behaviors. Cost: Some low-level thread interactions can be more difficult to program.

# Pragma



- 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

# “Hello, World” Using OpenMP



```
#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 */
```

# Compiling and Running



```
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
```

# Outline



- OpenMP Basics
  - Our First OpenMP Program
  - Fundamental Concepts and Library Functions
- The Trapezoidal Rule
- Scope of Variables
- Task Parallelism
  - The Reduction Clause
  - The `parallel for` Directive

# OpenMP Syntax



- Most of the constructs in OpenMP are compiler directives
  - #pragma omp directive [clause list]
- A parallel directive: # pragma omp parallel
  - The number of threads that run the following structured block of code is determined by the run-time system.
- Function prototypes and types in the file
  - #include <omp.h>

# 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

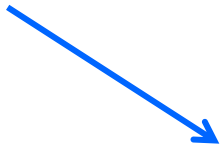


- In OpenMP parlance, the collection of threads executing the parallel block — the original thread and the new threads — is called a team, the original thread is called the master, and the additional threads are called slaves.

# 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
```

# OpenMP Library Functions



- **Control the number of threads and Processors:**

```
#include <omp.h>
void omp_set_num_threads (int num_threads);
int omp_get_num_threads ();
int omp_get_max_threads ();
int omp_get_thread_num ();
int omp_get_num_procs ();
int omp_in_parallel ();
```

- **Set and monitor thread creation:**

```
#include <omp.h>
void omp_set_dynamic(int dynamic_threads);
int omp_get_dynamic ();
void omp_set_nested (int nested);
int omp_get_nested ();
```

- **Mutex:**

```
#include <omp.h>
void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
int omp_test_lock(omp_lock_t *lock);
```

- **OpenMP also supports nested lock, which has similar semantics with simple lock.**

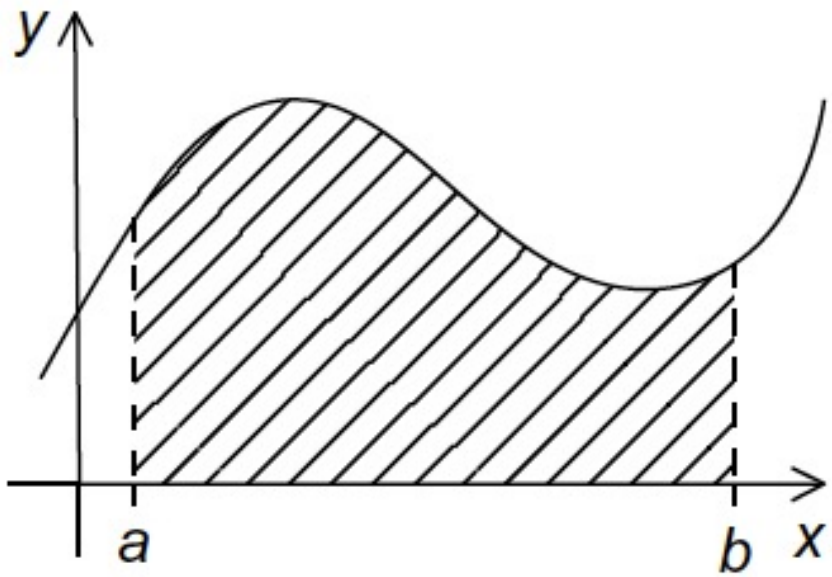
# Outline

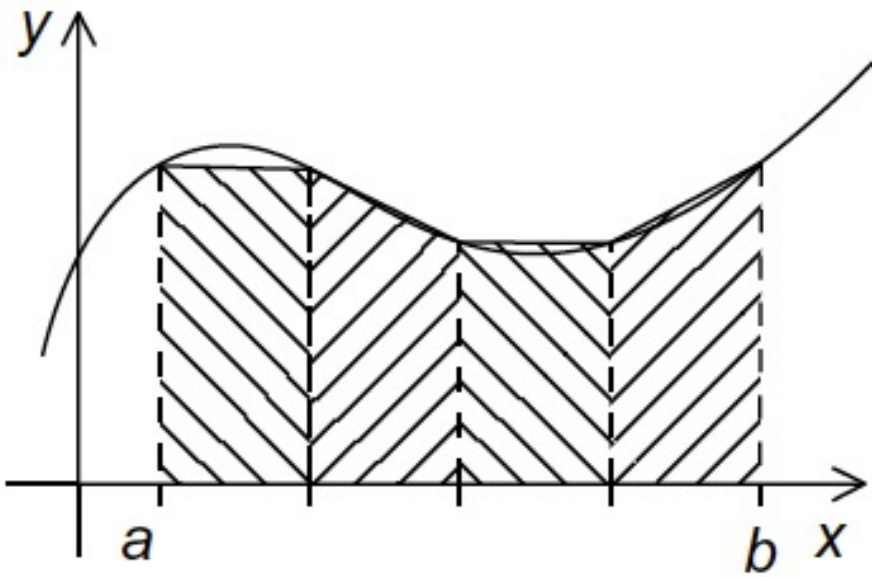


- OpenMP Basics
  - Our First OpenMP Program
  - Fundamental Concepts and Library Functions
- The Trapezoidal Rule
- Scope of Variables
- Task Parallelism
  - The Reduction Clause
  - The parallel for Directive

# The Trapezoidal Rule






If each subinterval has the same length $h$ and if we define $h=(b-a)/n$, $x_i=a+ih$, $i=0,1,2,\ldots,n$, then our approximation will be

$$h[f(x_0)/2+f(x_1)+f(x_2)+\cdots+f(x_{n-1})+f(x_n)/2]$$

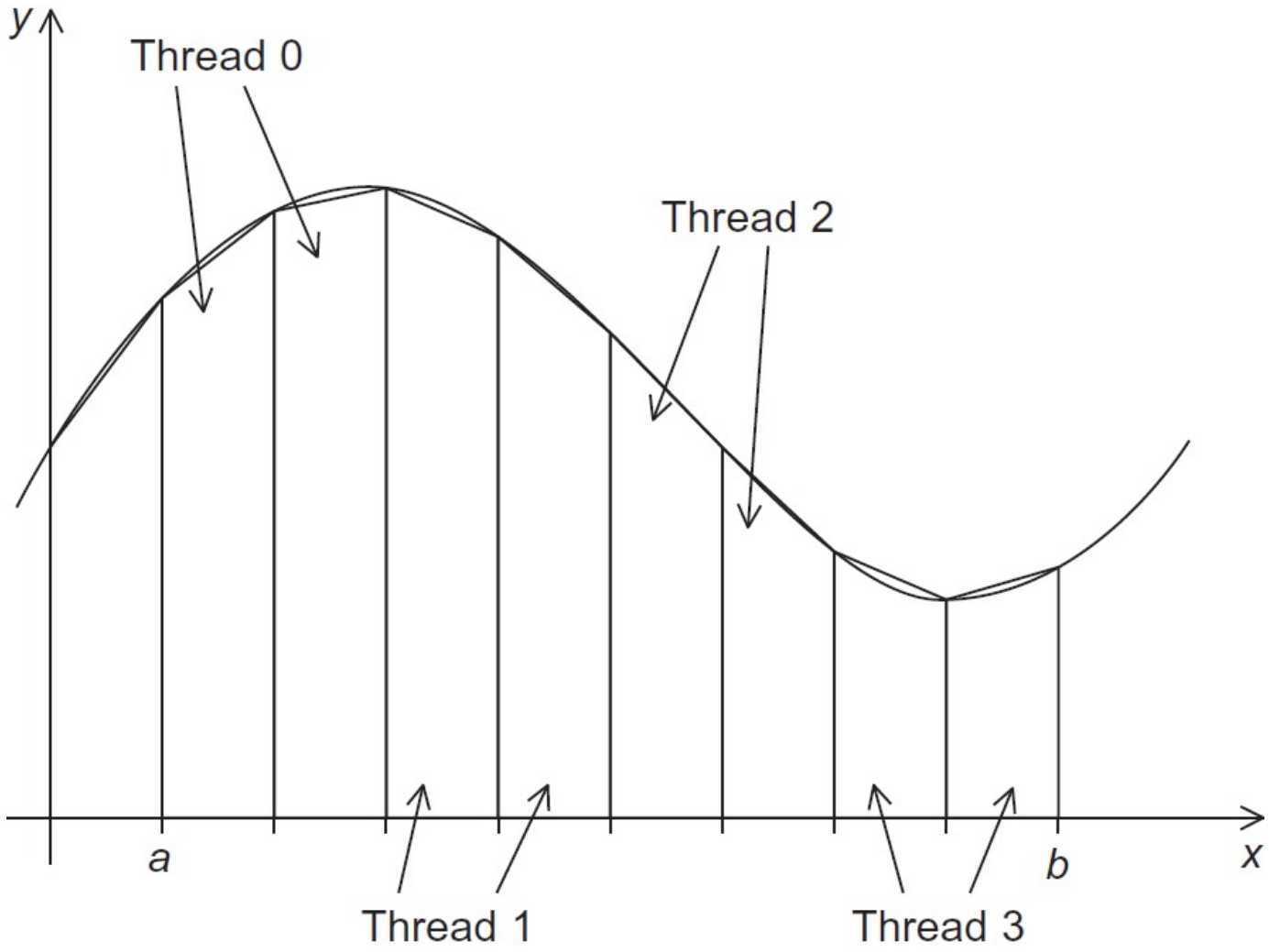
# Serial Algorithm



```
/* 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;
```

# 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 ;(critical section)

- Recall:

Race Condition, Critical Section

# Mutual Exclusion



Critical Directive:

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

only one thread can execute

the following structured block at a time

# First OpenMP Trapezoidal Rule Program(1)



```
#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 */
```

# First OpenMP Trapezoidal Rule Program(2)



```
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 */
```

# Outline



- OpenMP Basics
  - Our First OpenMP Program
  - Fundamental Concepts and Library Functions
- The Trapezoidal Rule
- Scope of Variables
- Task Parallelism
  - The Reduction Clause
  - The `parallel for` Directive

# Definition of 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.

# Example: Hello, World



```
#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 */
```

Private Scope

# Example: Trapezoidal Rule(1)



```
#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 */
```

Shared Scope

# Example: Trapezoidal Rule(2)



Private Scope

```
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 */
```

Shared Scope

# Outline



- OpenMP Basics
  - Our First OpenMP Program
  - Fundamental Concepts and Library Functions
- The Trapezoidal Rule
- Scope of Variables
- Task Parallelism
  - The Reduction Clause
  - The `parallel for` Directive

# Analysis of Trapezoidal Rule Program(1)



In the OpenMP program, this more complex version is used to get global_result by adding each thread's local calculation.

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

Although we'd prefer this for a serial implementation.

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

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

# Analysis of Trapezoidal Rule Program(2)



For the pointer version, we need to add each thread's local calculation to get global_result. If we use this, there's no critical section!

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

If we fix it like this, we force the threads to execute sequentially.

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

# Analysis of Trapezoidal Rule Program(3)



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;
   }
```

# Syntax of Reduction Clause



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

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

- 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



**A reduction clause can be added to a parallel directive.**

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

- 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.

# Outline



- OpenMP Basics
  - Our First OpenMP Program
  - Fundamental Concepts and Library Functions
- The Trapezoidal Rule
- Scope of Variables
- Task Parallelism
  - The Reduction Clause
  - The `parallel for` Directive

# Parallel for

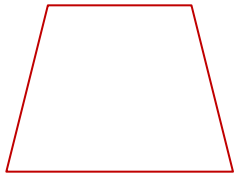


- Forks a team of threads to execute the following structured block.
- However, the structured block following the parallel for directive must be a for loop.
- With the parallel for directive, the system parallelizes the for loop by dividing the iterations of the loop among the threads.
- With just a parallel directive, in general, the work must be divided among the threads by the threads themselves.

# Example



```
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;
```

# Scope



- The default scope for all variables in a parallel directive is shared.
- In a loop that is parallelized with a parallel for directive, the default scope of the loop variable is private.

# Legal Forms for Parallelizable for Statements



- OpenMP will only parallelize for loops for which the number of iterations can be determined.
  - From the for statement itself
  - Prior to execution of the loop

# Examples of Illegal Forms



- The "infinite loop" cannot be parallelized.

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

- The following loop cannot be parallelized.

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

Since the number of iterations can't be determined from the `for` statement alone. This `for` loop is also not a structured block, since the `break` adds another point of exit from the loop.

# Legal Forms



- 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.
- 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

# Possible Case



thread 0: fibo[2] fibo[3] fibo[4] fibo[5]
thread 1: fibo[6] fibo[7] fibo[8] fibo[9]

- Correct: thread 0 finishes its computations before thread 1 starts.

- Incorrect: thread 0 has not computed fibo[4] and fibo[5], when thread 1 computes fibo[6]. It appears that the system has initialized the entries in fibo to 0, and thread 1 is using the values fibo[4] = 0 and fibo[5] = 0 to compute fibo[6] and so on.

# 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



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



Shared scope.

```
        double factor = 1.0;
        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);
        }
        pi_approx = 4.0*sum;
```

# 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.

# Summary



- Have a good holiday
- Looking forward to see you guys in person
- Don't forget to bring your laptop for the CUDA lectures.