Introduction to OpenMP

Chapter 5.1-5.

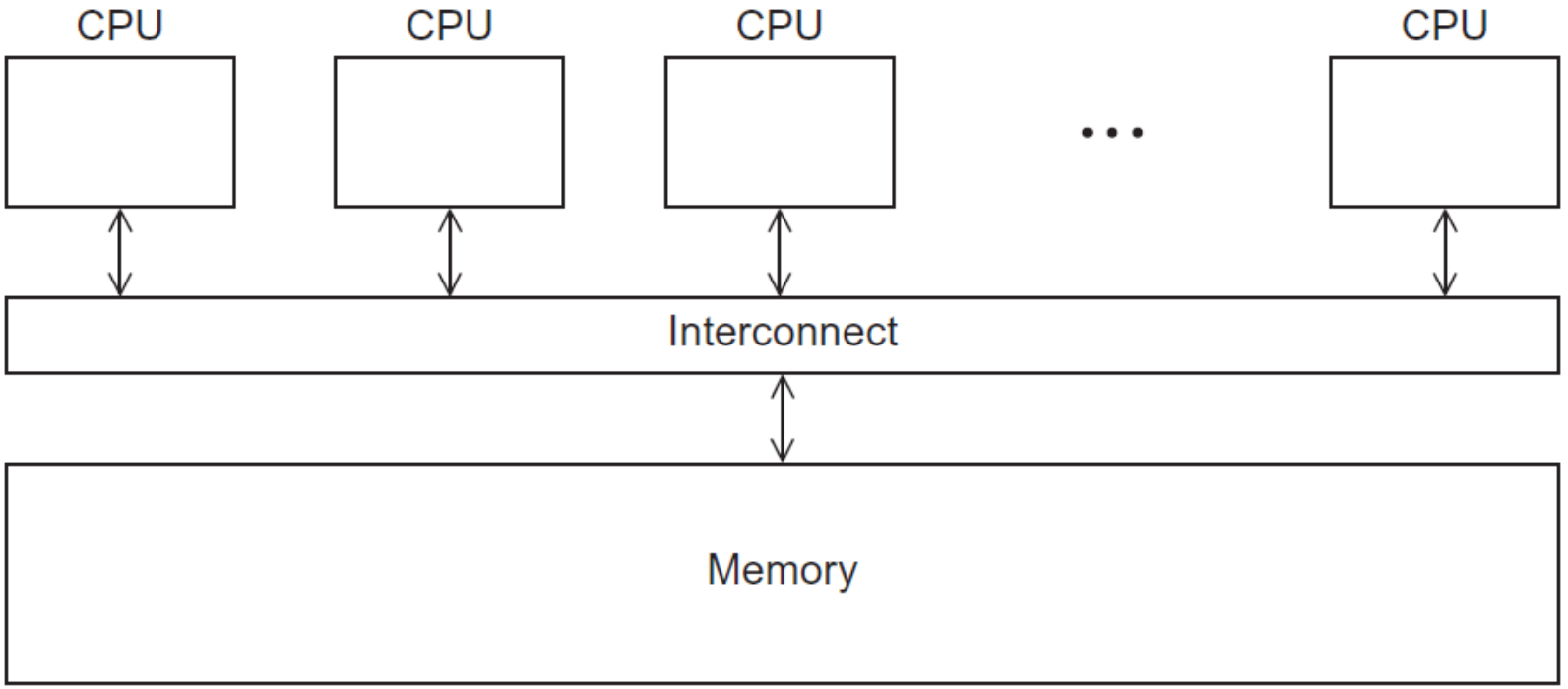
Bryan Mills, PhD

Spring 2017

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

```
#include<omp.h>

int main(){
  printf("The output:\n");
#pragma omp parallel num_threads(3)
  {
    printf("Hello World\n");
  }
  /* Resume Serial section*/
  printf("Done\n");
}
```

```
The output:
Hello World
Hello World
Hello World
Done
```

OpenMP

- API that abstracts away some of the details of pthreads
  - Simple library functions
  - Pragma directives (compiler hints)
  - Environment variables
- Compiler translates openmp code into pthreads calls
- Starts with a **master** thread, calls to openmp create a **team** of **slaves**.

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.
- Always starts with '#' in first column!

```
#pragma omp parallel num_threads(3)
```

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.

Pragma 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 )
```

Specifying number of threads

- Multiple ways of specifying the number of threads.
  - Explicitly in pragma

```
#pragma omp parallel num_threads(3)
```

  - Environment Variable

```
export OMP_NUM_THREADS=3
```

  - Call library function

```
omp_set_num_threads(int number)l
```

Easy to use

```
void hello() {
  int my_rank = omp_get_thread_num();
  int total = omp_get_num_threads();

  printf("Hello World from rank:%d total:%d\n",
      my_rank,
      total);
}

int main(){
  printf("The output:\n");

# pragma omp parallel
  hello();

  printf("Done\n");
}
```

Independent Threads

- Just like pthreads order is not guaranteed

```
Hello World from rank: 1 total: 4
Hello World from rank: 3 total: 4
Hello World from rank: 2 total: 4
Hello World from rank: 0 total: 4
```

Or any combination:

```
Hello World from rank: 0 total: 4
Hello World from rank: 2 total: 4
Hello World from rank: 3 total: 4
Hello World from rank: 1 total: 4
```

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.

Compiler Doesn't Support OpenMP?

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

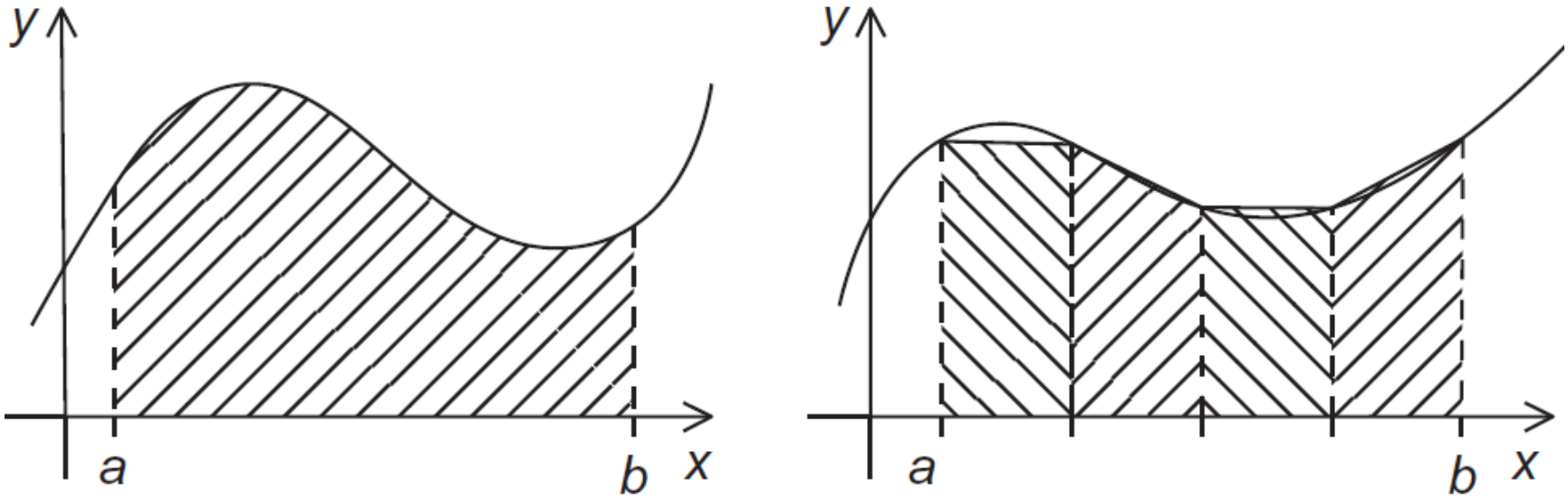


```
#ifdef _OPENMP
# include <omp.h>
#endif
```

```
int my_rank = omp_get_thread_num();
```

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

The trapezoidal rule



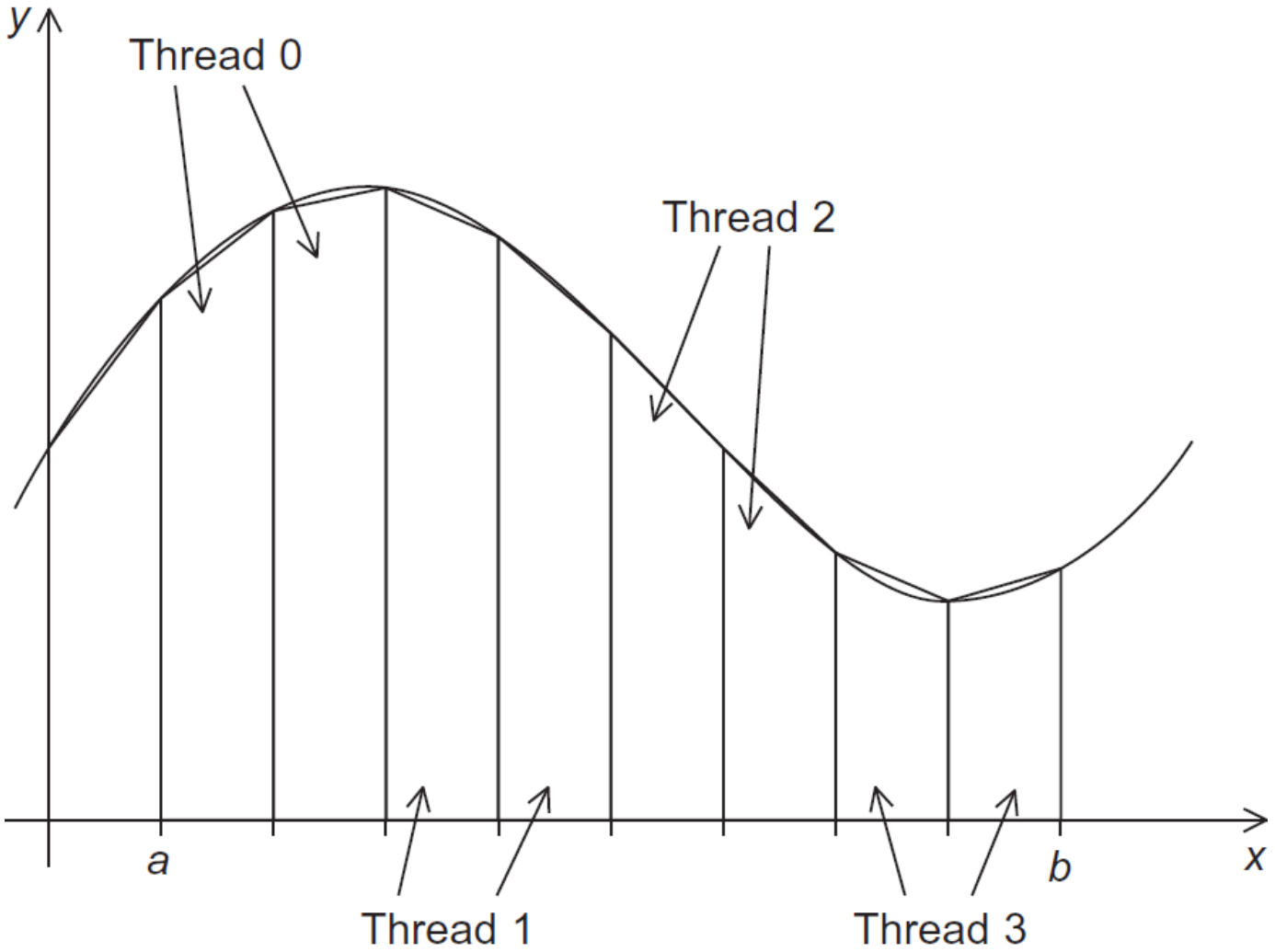

```
h = (b-a) / n;
approx = (f(a) - f(b)) / 2.0;
for(int i = 1; i < n-1; i++) {
  x_i = a + i*h;
  approx += f(x_i);
}
approx = h*approx;
```

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



Race Condition

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

```
int main(int argc, char* argv[]) {
  double  global_result = 0.0;  /* Store result in global_result */
  int a = 0;                    /* Left and right endpoints      */
  int b = 10.0;
  int n = 100000;               /* Total number of trapezoids    */
#  pragma omp parallel num_threads(thread_count)
  Trap(a, b, n, &global_result);
}
void Trap(double a, double b, int n, double* global_result_p) {
  double  my_result;
  int my_rank = omp_get_thread_num();
  int thread_count = omp_get_num_threads();

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

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

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.

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!

```
global_result += Trap(double a, double b, int n)
```

If we fix it like this…

```
global_result = 0.0;
#pragma omp parallel
{
  pragma omp critical
  global_result += Trap(a, b, 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.

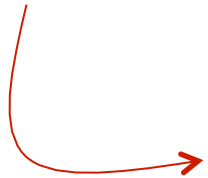
```
global_result = 0.0;
#pragma omp parallel
{
  double my_result = Trap(a, b, n);
  pragma omp critical
  global_result += my_result;
}
```

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>)
```



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

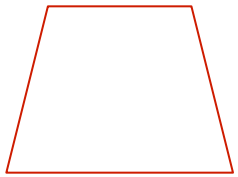
```
global_result = 0.0;
#pragma omp parallel reduction(+: global_result)
global_result += Trap(a, b, n);
```

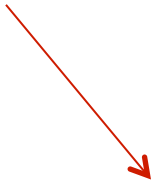
Parallel for

- 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 (int 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 reduction(+: approx)
for (int i = 1; i < n-1; i++)
  approx += f(a + i*h);
approx = h*approx;
```

Legal forms for parallelizable for statements

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

What happened?

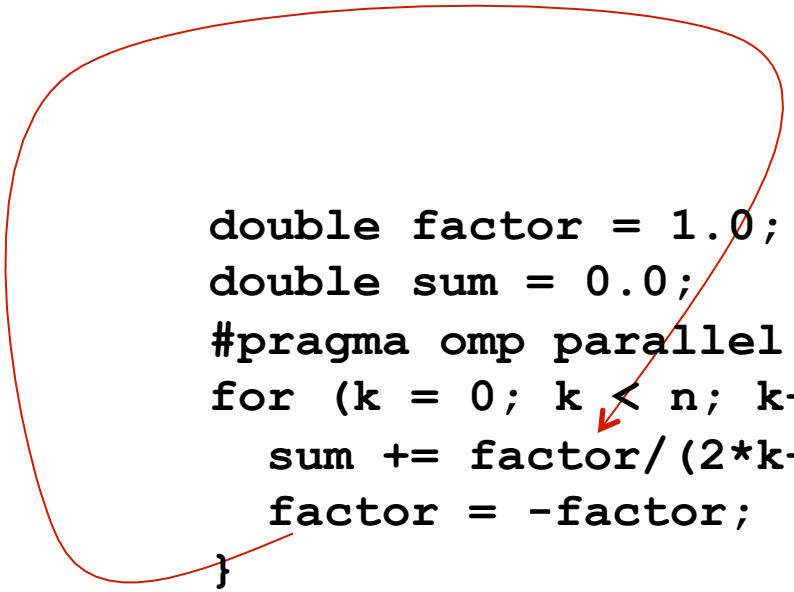
1. OpenMP compilers don't check for dependences among iterations in a loop that's being parallelized with a parallel for directive.
2. A loop in which the results of one or more iterations depend on other iterations cannot, in general, be correctly parallelized by OpenMP.

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;
```

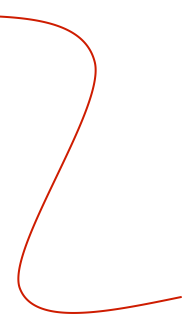
OpenMP solution #1




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

OpenMP solution #2

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



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 factor = 1.0;
double sum = 0.0;
#pragma omp parallel for reduction(+:sum) \
   default(none) private(factor)
for (k = 0; k < n; k++) {
  sum += factor/(2*k+1);
  factor = -factor;
}
pi_approx = 4.0 * sum;
```