

COMP 4901Q: High Performance Computing (HPC)

Lecture 5: Introduction to OpenMP

Instructor: Shaohuai SHI (shaohuais@cse.ust.hk)

Teaching assistants: Mingkai TANG (mtangag@connect.ust.hk)

Yazhou XING (yxingag@connect.ust.hk)

Course website: <https://course.cse.ust.hk/comp4901q/>

Outline

- ▶ Parallel programming with OpenMP
- ▶ Introduction to OpenMP
 - ▶ Creating parallelism
 - ▶ Parallel Loops
 - ▶ Synchronizing
 - ▶ Data sharing

What is OpenMP

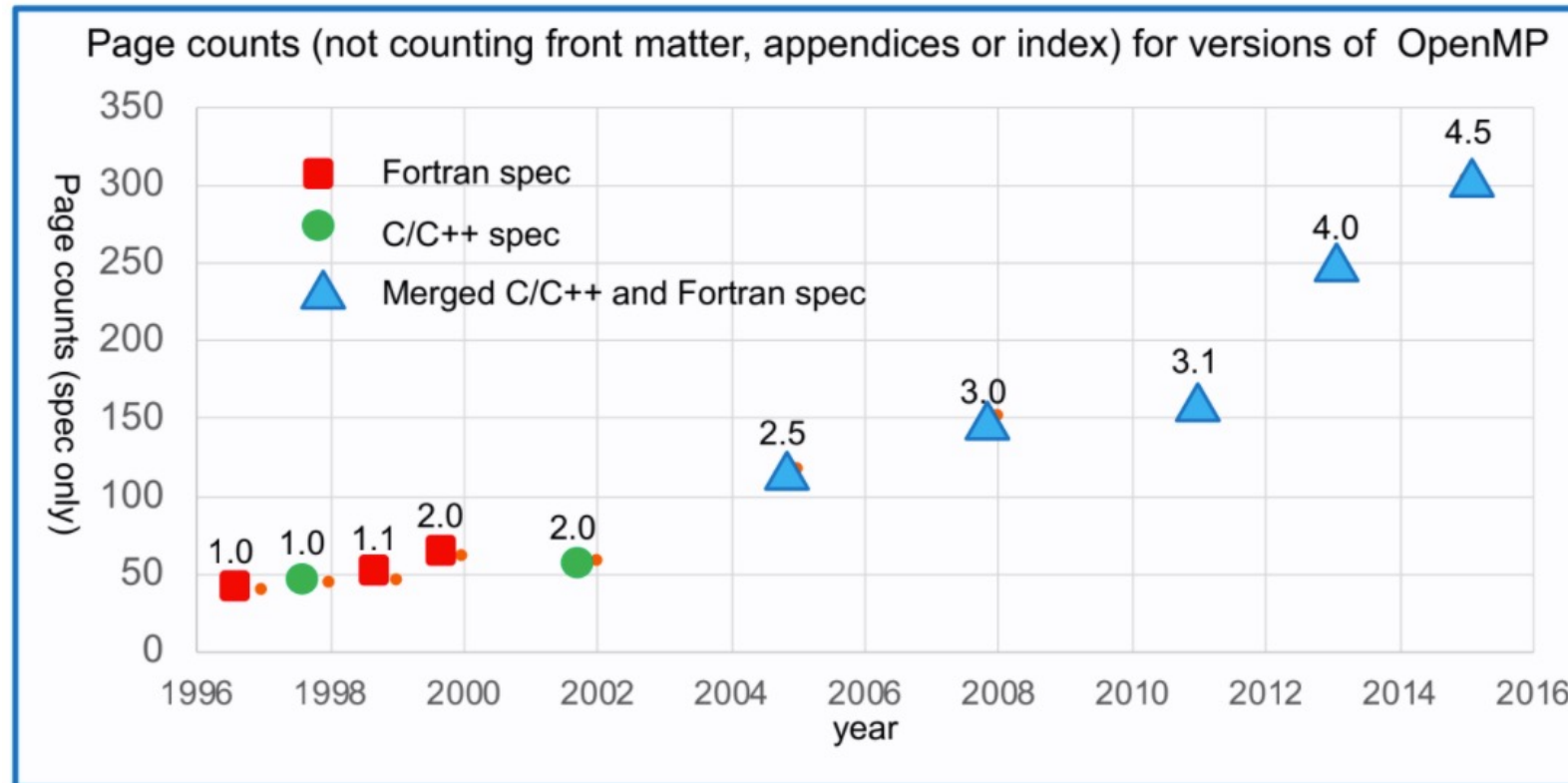
- ▶ OpenMP = Open specification for Multi-Processing
 - ▶ www.openmp.org – Talks, examples, forums, etc.
 - ▶ Specification controlled by the OpenMP Architecture Review Board (ARB)
 - ▶ A nonprofit organization that controls the OpenMP Spec
 - ▶ Latest spec: OpenMP 5.0 (Nov. 2018)
- ▶ Motivation: capture common usage and simplify programming
 - ▶ Designed for systems in which each thread or process can potentially have access to all available memory
- ▶ Thread based parallelism
 - ▶ OpenMP programs accomplish parallelism exclusively through the use of threads
 - ▶ Explicit parallelism
 - ▶ offering the programmer full control over parallelization
- ▶ Data scoping
 - ▶ All threads in a parallel region can access this shared data simultaneously

A Programmer's View of OpenMP

- ▶ OpenMP is a portable, threaded, shared-memory, programming specification with “light” syntax
 - ▶ Requires compiler support (C, C++ or Fortran)
- ▶ OpenMP will
 - ▶ allow a programmer to separate a program into serial regions and parallel regions, rather than P concurrently-executing threads.
 - ▶ hide stack management
 - ▶ provide synchronization constructs
- ▶ OpenMP will not
 - ▶ parallelize automatically
 - ▶ guarantee speedup
 - ▶ provide freedom from data races

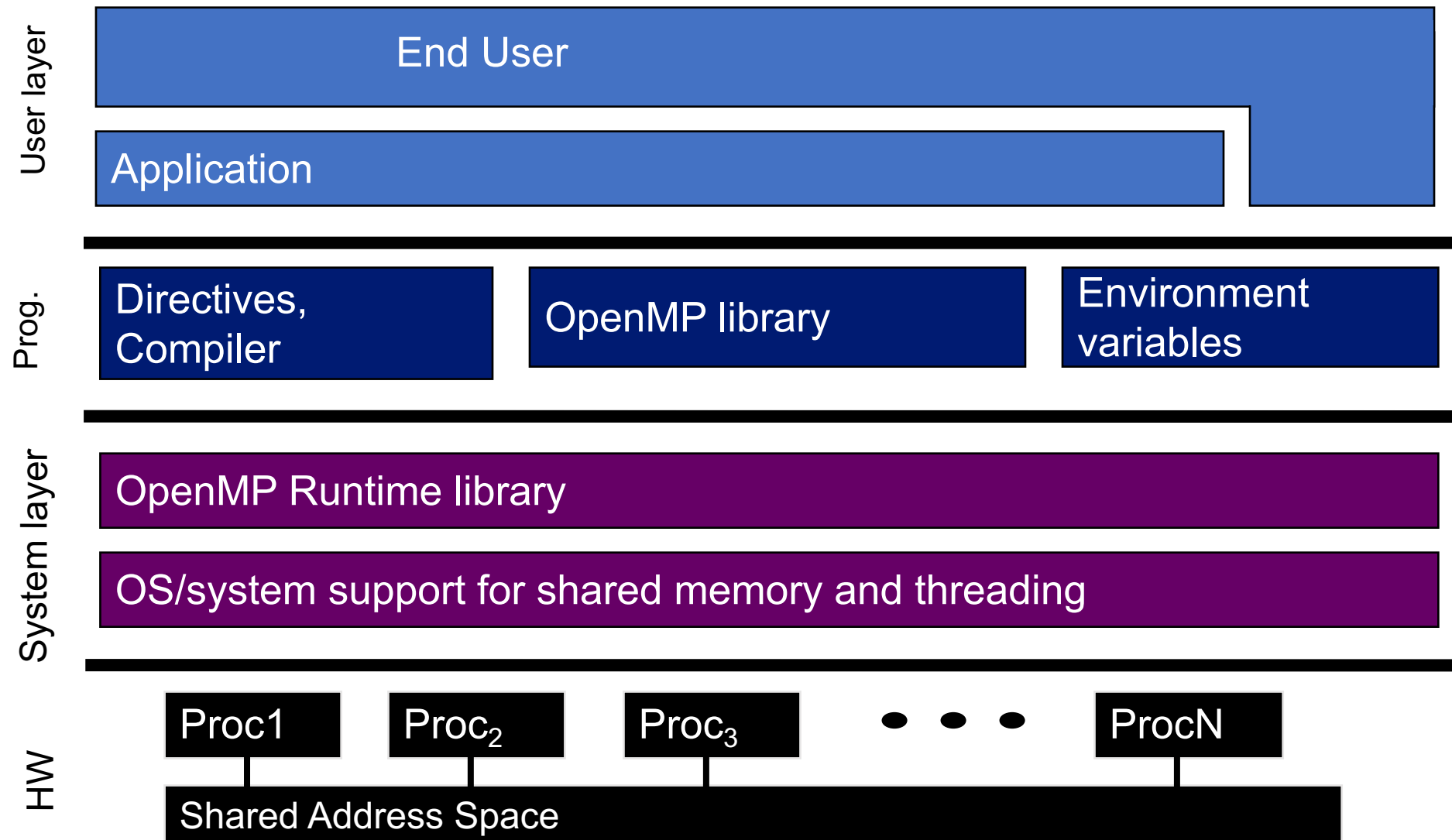
The Growth of OpenMP

- ▶ Started out in 1997 as a simple interface
- ▶ The complexity has grown considerably over the years

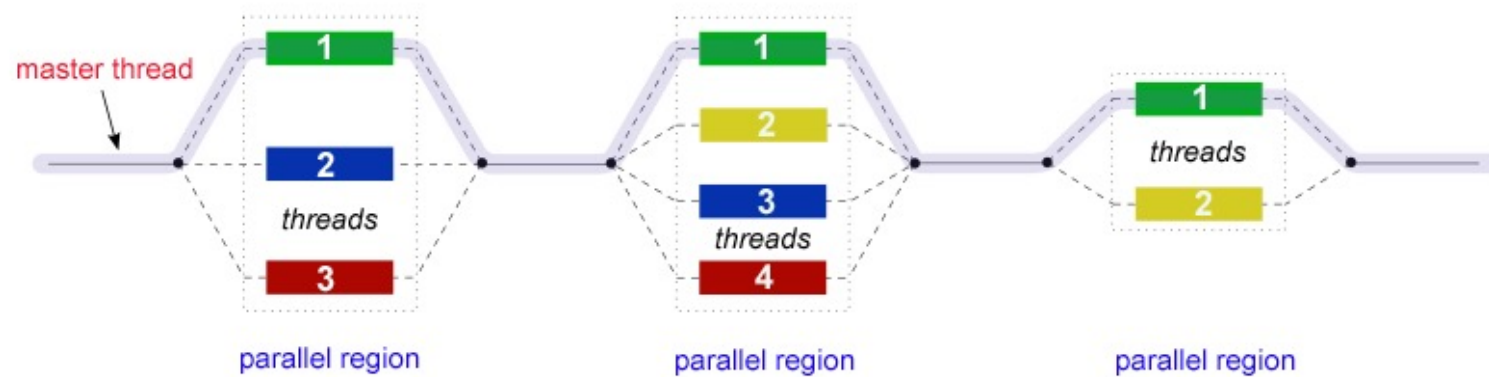


OpenMP 5.0 (Nov 2018) is actually **666** pages!

OpenMP Basic Definitions: Basic Solution Stack



Fork - Join Model



- ▶ All OpenMP programs begin as a single process: the master thread
 - ▶ **FORK**: the master thread then creates a team of parallel threads
 - ▶ **JOIN**: when the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread
- ▶ Some terminology
 - ▶ the original thread and the new threads — is called a **team**
 - ▶ the original thread is called the **master**
 - ▶ the additional threads are called **slaves**
- ▶ High-level API for programming in C/C++ and Fortran
 - ▶ Preprocessor (compiler) directives (~ 80%)
 - ▶ `#pragma omp directive-name [clause [clause ...]]`
 - ▶ Library Calls (~ 19%)
 - ▶ `#include <omp.h>`
 - ▶ Environment Variables (~ 1%)
 - ▶ All caps, added to `srun`, etc.

OpenMP Components

Directives

- ▶ Parallel regions
- ▶ Work sharing
- ▶ Synchronization
- ▶ Data-sharing attributes
 - ▶ Private
 - ▶ Firstprivate
 - ▶ Lastprivate
 - ▶ Shared
 - ▶ Reduction
- ▶ Orphaning

Runtime environment

- ▶ Number of threads
- ▶ Thread ID
- ▶ Dynamic thread adjustment
- ▶ Nested parallelism
- ▶ Timers
- ▶ API for locking

Environment variables

- ▶ Number of threads
- ▶ Scheduling type
- ▶ Dynamic thread adjustment
- ▶ Nested parallelism

Compiler Directives and Clauses

- ▶ Compiler directives have the following syntax

`#pragma omp directive [clause [clause] ...]`

- ▶ Example

```
#pragma omp parallel if( n > threshold) \
    shared(n,x,y) private(i)
{
    #pragma omp for
    for (i=0; i<n; i++) x[i] += y[i];
} /* End of parallel region */
```

- ▶ Spawning a parallel region
- ▶ Dividing blocks of code among threads
- ▶ Distributing loop iterations between threads
- ▶ Serializing sections of code
- ▶ Synchronization of work among threads

- ▶ **if (scalar expression)**

- ▶ Only execute in parallel if expression evaluates to true
- ▶ Otherwise, execute serially

- ▶ **private(list)**

- ▶ No storage association with original object
- ▶ All references are to the local object
- ▶ Values are undefined on entry and exit

- ▶ **shared(list)**

- ▶ Data is accessible by all threads in the team
- ▶ All threads access the same address space

- ▶ **firstprivate(list)**

- ▶ All variables in the list are initialized with the value the original object had before entering the parallel construct

- ▶ **lastprivate(list)**

- ▶ The thread that executes the sequentially last iteration or section updates the value of the objects in the list

Run-time Library Routines

- ▶ Calls pre-defined functions
 - ▶ Example
 - ▶ `#include <omp.h>`
 - ▶ `int omp_get_num_threads(void)`
- ▶ Setting and querying the number of threads
- ▶ Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
- ▶ Setting and querying the dynamic threads feature
- ▶ Querying if in a parallel region, and at what level
- ▶ Setting and querying nested parallelism
- ▶ Setting, initializing and terminating locks and nested locks
- ▶ Querying wall clock time and resolution

Environment Variables

- ▶ OpenMP provides several environment variables for controlling the execution of parallel code at run-time

- ▶ Example

csh/tcsh	setenv OMP_NUM_THREADS 8
sh/bash	export OMP_NUM_THREADS=8

- ▶ Setting the number of threads
- ▶ Specifying how loop iterations are divided
- ▶ Binding threads to processors
- ▶ Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
- ▶ Enabling/disabling dynamic threads
- ▶ Setting thread stack size
- ▶ Setting thread wait policy

The OpenMP Common Core

► Mostly used 19 items

OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads
int omp_get_thread_num() int omp_get_num_threads()	Create threads with a parallel region and split up the work using the number of threads and thread ID
double omp_get_wtime()	Speedup and Amdahl's law. False Sharing and other performance issues
setenv OMP_NUM_THREADS N	Internal control variables. Setting the default number of threads with an environment variable
#pragma omp barrier #pragma omp critical	Synchronization and race conditions. Revisit interleaved execution.
#pragma omp for #pragma omp parallel for	Worksharing, parallel loops, loop carried dependencies
reduction(op:list)	Reductions of values across a team of threads
schedule(dynamic [,chunk]) schedule (static [,chunk])	Loop schedules, loop overheads and load balance
private(list), firstprivate(list), shared(list)	Data environment
nowait	Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive)
#pragma omp single	Workshare with a single thread
#pragma omp task #pragma omp taskwait	Tasks including the data environment for tasks.

OpenMP – Hello World

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h> ← Runtime header

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) ← Compiler directive
    Hello();

    return 0;
} /* main */

void Hello(void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads(); ← Runtime routines

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

} /* Hello */

$gcc -fopenmp -o omp_hello omp_hello.c
$./omp_hello 4
```

OpenMP Directives

▶ Format

#pragma omp	directive-name	[clause, ...]	newline
Required for all OpenMP C/C++ directives.	A valid OpenMP directive. Must appear after the pragma and before any clauses.	Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.	Required. Precedes the structured block which is enclosed by this directive.

▶ General rules:

- ▶ Case sensitive
- ▶ Directives follow conventions of the C/C++ standards for compiler directives
- ▶ Only one directive-name may be specified per directive
- ▶ Each directive applies to at most one succeeding statement, which must be a structured block.
- ▶ Long directive lines can be "continued" on succeeding lines by escaping the newline character with a backslash ("\") at the end of a directive line.

▶ Example

- ▶ `#pragma omp parallel default(shared) private(beta,pi)`

OpenMP Directives

#pragma

- ▶ Special **preprocessor** instructions.
- ▶ Typically added to a system to allow behaviors that aren't part of the basic C/C++ specification.
- ▶ Compilers that don't support the pragma ignore them.

OpenMP Directives

#pragma omp parallel [clause[[,] clause] ...]

- ▶ Most basic parallel directive.
- ▶ The number of threads that run the following structured block of code is determined by the run-time system.
- ▶ Supports the following clauses
 - ▶ if (scalar expression)
 - ▶ private(list)
 - ▶ shared(list)
 - ▶ default(none/shared)
 - ▶ reduction(operator:list)
 - ▶ copyin(list)
 - ▶ firstprivate(list)
 - ▶ num_threads(scalar)

OpenMP Directives

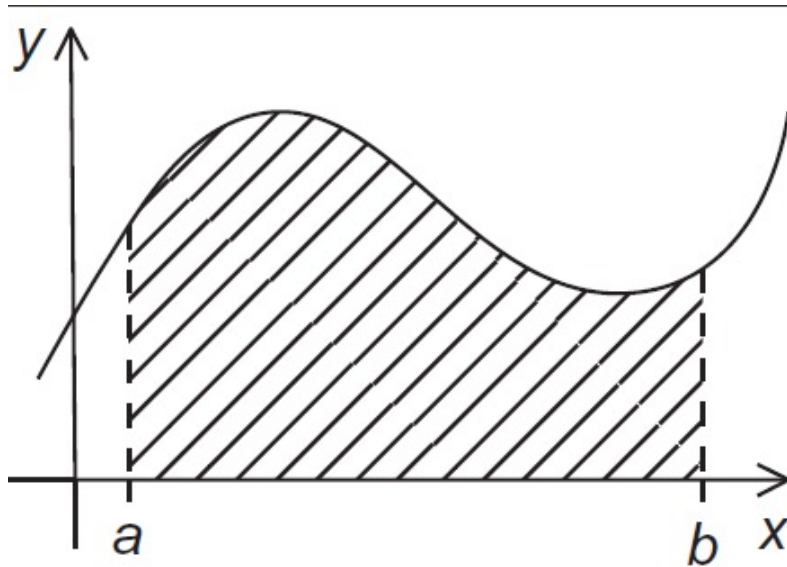
```
#pragma omp parallel num_threads ( thread_count )
```

▶ **Clause**

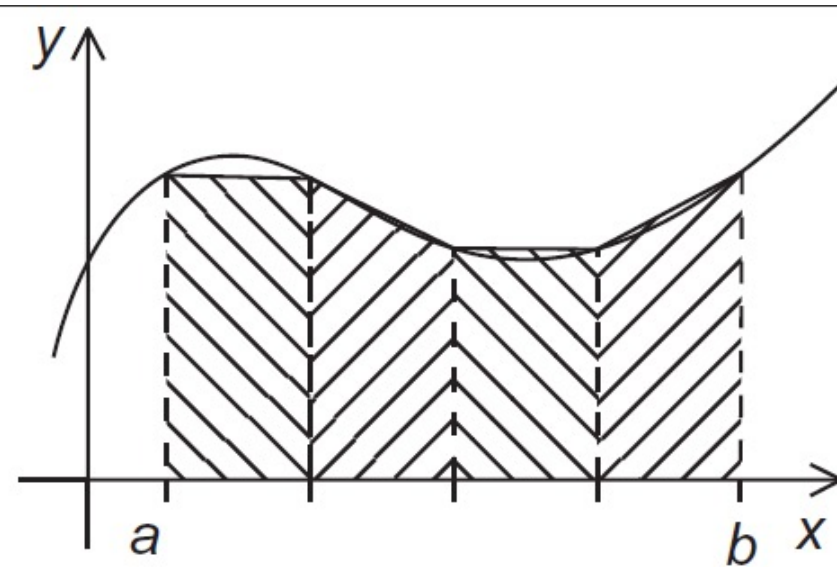
- ▶ Text that modifies a directive.
- ▶ The `num_threads` clause can be added to a parallel directive.
- ▶ It allows the programmer to specify the number of threads that should execute the following block.

Example - The Trapezoidal Rule

- ▶ Evaluates the area under the curves by dividing the total area into smaller trapezoids rather than using rectangles



$$\text{Area} = \int_a^b f(x) dx$$



$$\Delta x = \frac{b-a}{n}$$

$$a = x_0 < x_1 < x_2 < x_3 < \dots < x_n = b$$

$$\int_a^b f(x) dx \approx \frac{\Delta x}{2} (f(x_0) + 2f(x_1) + 2f(x_2) + \dots + 2f(x_{n-1}) + f(x_n))$$

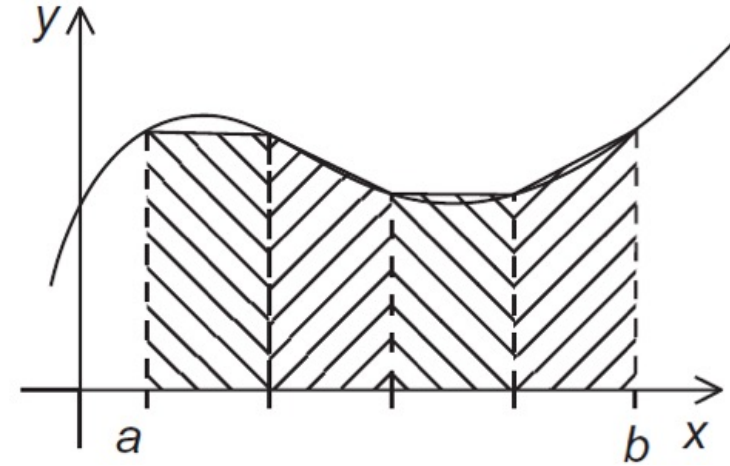
$$x_i = a + i\Delta x$$

The Trapezoidal Rule - A 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

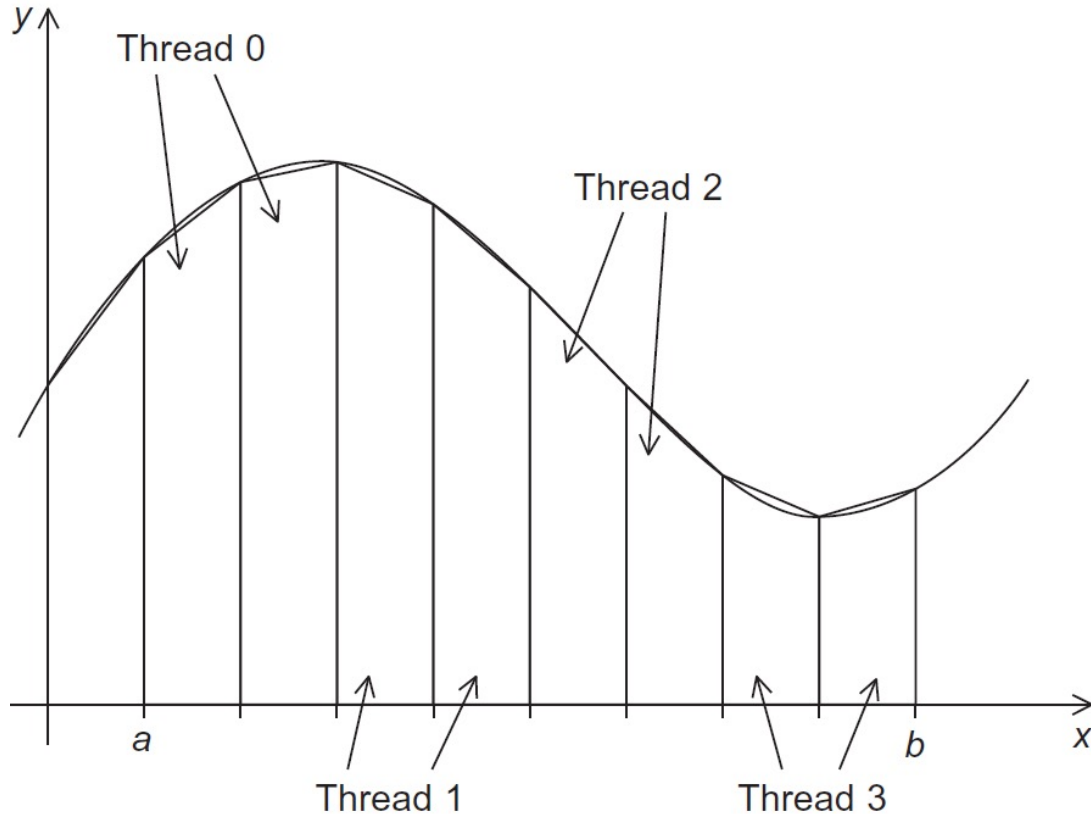
- ▶ 1) We identified two types of tasks:
 - ▶ a) computation of the areas of individual trapezoids, and
 - ▶ b) adding the areas of trapezoids
- ▶ 2) There is no communication among the tasks in the first collection, but each task in the first collection communicates with task 1b.



```
/* 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

- ▶ I.a) computation of the areas of individual trapezoids
- ▶ I.b) adding the areas of trapezoids



Time	Thread 0	Thread 1
0	<code>global_result = 0 to register</code>	<code>finish my_result</code>
1	<code>my_result = 1 to register</code>	<code>global_result = 0 to register</code>
2	<code>add my_result to global_result</code>	<code>my_result = 2 to register</code>
3	<code>store global_result = 1</code>	<code>add my_result to global_result</code>
4		<code>store global_result = 2</code>

- ▶ Unpredictable results when two (or more) threads attempt to simultaneously execute:

`global_result += my_result ;`

OpenMP Directives - Mutual Exclusion

- ▶ Critical sections
 - ▶ Only one thread can execute the following structured block at a time

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

The Trapezoidal Rule with OpenMP

```
#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
 - ▶ 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**.

Scope of Variables

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);
```

Scope of Variables

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.

Scope of Variables

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.

OpenMP Directives - Reduction Clause

- ▶ 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 - 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**.

OpenMP Directives - 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 of parallelizable for statements

for	{			index++
				++index
		index < end		index--
		index <= end		--index
		index = start ;	index >= end ;	index += incr
			index > end	index -= incr
				index = index + incr
				index = incr + index
		index = index - incr		
	}			

- ▶ 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

- ▶ The “parallel for” directive does **NOT** check data dependencies
- ▶ A loop in which the results of one or more iterations depend on other iterations cannot, in general, be correctly parallelized by OpenMP

```
fibonacci[ 0 ] = fibonacci[ 1 ] = 1;
```

```
for (i = 2; i < n; i++)
```

```
    fibonacci[ i ] = fibonacci[ i - 1 ] + fibonacci[ i - 2 ];
```

note 2 threads

```
fibonacci[ 0 ] = fibonacci[ 1 ] = 1;
```

```
# pragma omp parallel for num_threads(2)
```

```
for (i = 2; i < n; i++)
```

```
    fibonacci[ i ] = fibonacci[ i - 1 ] + fibonacci[ i - 2 ];
```

1 1 2 3 5 8 13 21 34 55

this is correct

1 1 2 3 5 8 0 0 0 0

but sometimes
we get this

Estimating π

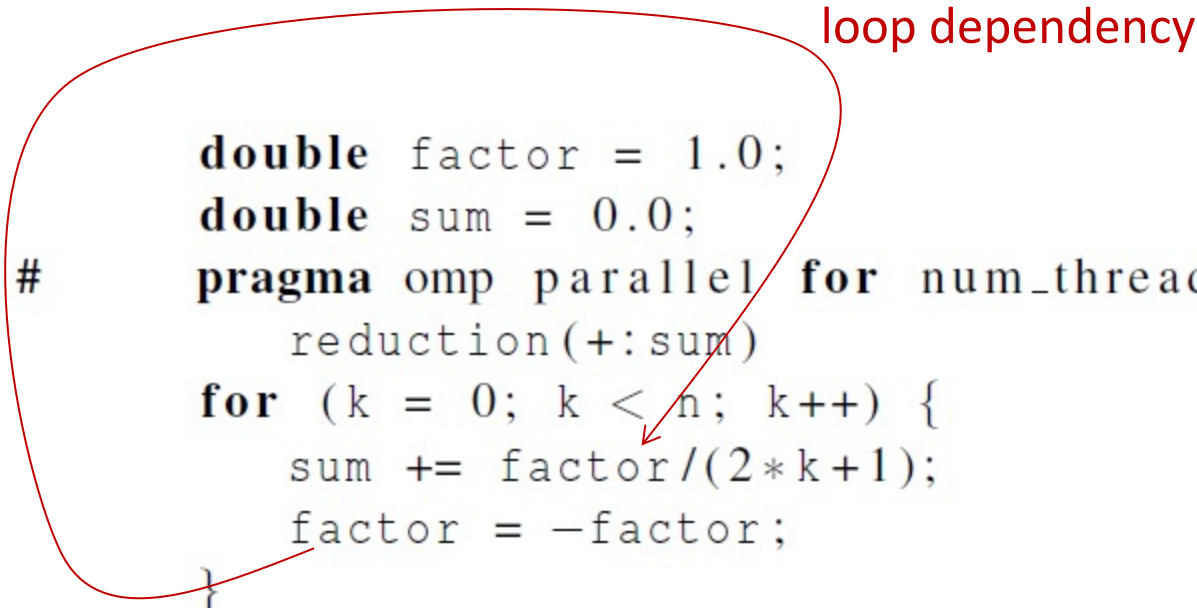
$$\pi = 4 \left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots \right] = 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

```
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);
}
```



Ensures factor has
private scope.

The Default Clause

- ▶ Let 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.

```
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);
}
```

OpenMP Directives - Scheduling Loops

- ▶ We want to parallelize a loop
 - ▶ assignment of work using **cyclic partitioning**

```
sum = 0.0;
for (i = 0; i <= n; i++)
    sum += f(i);
```

Thread	Iterations
0	0, n/t , $2n/t$, ...
1	1, $n/t + 1$, $2n/t + 1$, ...
\vdots	\vdots
$t - 1$	$t - 1$, $n/t + t - 1$, $2n/t + t - 1$, ...

```
double f(int i) {
    int j, start = i*(i+1)/2, finish = start + i;
    double return_val = 0.0;

    for (j = start; j <= finish; j++) {
        return_val += sin(j);
    }
    return return_val;
} /* f */
```

- ▶ $f(i)$ calls the sin function i times
 - ▶ Assume the time to execute $f(2i)$ requires approximately twice as much time as the time to execute $f(i)$
- ▶ $n=10,000$
 - ▶ One thread takes about 3.67 seconds

OpenMP Directives - The Schedule Clause

▶ Default schedule

```
sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
    reduction(+:sum)
for (i = 0; i <= n; i++)
    sum += f(i);
```

- ▶ **n = 10,000**
 - ▶ two threads
 - ▶ default assignment
 - ▶ run-time = **2.76** seconds
 - ▶ speedup = **1.33x**

▶ Cyclic schedule

```
sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
    reduction(+:sum) schedule(static,1)
for (i = 0; i <= n; i++)
    sum += f(i);
```

- ▶ **n = 10,000**
 - ▶ two threads
 - ▶ cyclic assignment
 - ▶ run-time = **1.84** seconds
 - ▶ speedup = **1.99x**

OpenMP Directives - The Schedule Clause

- ▶ `schedule (type , chunksize)`
 - ▶ The **type** can be:
 - ▶ **static**: the iterations can be assigned to the threads before the loop is executed.
 - ▶ **dynamic** or **guided**: the iterations are assigned to the threads while the loop is executing.
 - ▶ **auto**: the compiler and/or the run-time system determine the schedule.
 - ▶ **runtime**: the schedule is determined at run-time.
 - ▶ The **chunksize** is a positive integer.

The Schedule Clause - Type

```
    sum = 0.0;
#    pragma omp parallel for num_threads(thread_count) \
        reduction(+:sum) schedule(static,1)
    for (i = 0; i <= n; i++)
        sum += f(i);
```

▶ E.g., thread_count=3

▶ schedule(static, 1)

- ▶ **Thread 0:** 0, 3, 6, 9
- ▶ **Thread 1:** 1, 4, 7, 10
- ▶ **Thread 2:** 2, 5, 8, 11

▶ schedule(static, 2)

- ▶ **Thread 0:** 0, 1, 6, 7
- ▶ **Thread 1:** 2, 3, 8, 9
- ▶ **Thread 2:** 4, 5, 10, 11

The Schedule Clause - The Dynamic Schedule Type

- ▶ The iterations are also broken up into chunks of **chunksize** consecutive iterations.
- ▶ Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the run-time system.
- ▶ This continues until all the iterations are completed.
- ▶ The **chunksize** can be omitted. When it is omitted, a **chunksize** of 1 is used.

The Schedule Clause - The Guided Schedule Type

- ▶ Each thread also executes a chunk, and when a thread finishes a chunk, it requests another one.
- ▶ However, in a guided schedule, as chunks are completed the size of the new chunks decreases.
- ▶ If no **chunksize** is specified, the size of the chunks decreases down to 1.
- ▶ If **chunksize** is specified, it decreases down to **chunksize**, with the exception that the very last chunk can be smaller than **chunksize**.

The Schedule Clause - The Runtime Schedule

- ▶ The system uses the environment variable `OMP_SCHEDULE` to determine at run-time how to schedule the loop.
- ▶ The `OMP_SCHEDULE` environment variable can take on any of the values that can be used for a static, dynamic, or guided schedule.
 - ▶ `$setenv OMP_SCHEDULE "guided,4"`
 - ▶ `$setenv OMP_SCHEDULE "dynamic"`

OpenMP Directives - Atomic

```
# pragma omp atomic
```

- ▶ Unlike the critical directive, it can only protect critical sections that consist of a single assignment statement
- ▶ The statement must have one of the following forms

```
x <op>= <expression>;  
x++;  
++x;  
x--;  
--x;
```

- ▶ Here <op> can be one of the binary operators

```
+, *, -, /, &, ^, |, <<, or >>
```

- ▶ Many processors provide a special load-modify-store instruction.
- ▶ A critical section that only does a load-modify-store can be protected **much more efficiently** by using this special instruction rather than the constructs that are used to protect more general critical sections

Synchronization Clause

- ▶ High level synchronization
 - ▶ **critical**
 - ▶ **atomic**
 - ▶ **barrier**
 - ▶ ordered
- ▶ Low level synchronization
 - ▶ flush
 - ▶ locks

Matrix-vector Multiplication

$$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$$

a_{00}	a_{01}	\cdots	$a_{0,n-1}$		y_0
a_{10}	a_{11}	\cdots	$a_{1,n-1}$		y_1
\vdots	\vdots		\vdots		\vdots
a_{i0}	a_{i1}	\cdots	$a_{i,n-1}$	x_0	$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$
\vdots	\vdots		\vdots	x_1	\vdots
$a_{m-1,0}$	$a_{m-1,1}$	\cdots	$a_{m-1,n-1}$	\vdots	\vdots
				x_{n-1}	y_{m-1}

```

for (i = 0; i < m; i++) {
    y[i] = 0.0;
    for (j = 0; j < n; j++)
        y[i] += A[i][j]*x[j];
}

```

Matrix-vector Multiplication

```
# pragma omp parallel for num_threads(thread_count) \
    default(none) private(i, j) shared(A, x, y, m, n)
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    for (j = 0; j < n; j++)
        y[i] += A[i][j]*x[j];
}
```


Reading List

- ▶ <https://hpc-tutorials.llnl.gov/openmp/>
- ▶ <https://www.nersc.gov/users/training/events/openmp-common-core-february-2018/>

Summary

- ▶ OpenMP is a standard for programming shared-memory systems
- ▶ OpenMP uses both special functions and preprocessor directives called pragmas
 - ▶ Creating parallelism
 - ▶ Parallel Loops
 - ▶ Synchronizing
 - ▶ Data sharing