

PARALLEL COMPUTING		Semester	VII
Course Code	BCS702	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	3:0:2:0	SEE Marks	50
Total Hours of Pedagogy	40 hours Theory + 8-10 Lab slots	Total Marks	100
Credits	04	Exam Hours	03
Examination nature (SEE)	Theory/Practical		

Course outcomes (Course Skill Set):

At the end of the course, the student will be able to:

- Explain the need for parallel programming
- Demonstrate parallelism in MIMD system.
- Apply MPI library to parallelize the code to solve the given problem.
- Apply OpenMP pragma and directives to parallelize the code to solve the given problem
- Design a CUDA program for the given problem.

MODULE-4

Shared-memory programming with OpenMP – openmp pragmas and directives, The trapezoidal rule, Scope of variables, The reduction clause, loop carried dependency, scheduling, producers and consumers, Caches, cache coherence and false sharing in openmp, tasking, tasking, thread safety.

Shared Memory with MIMD

each cooking a different dish

chef 1

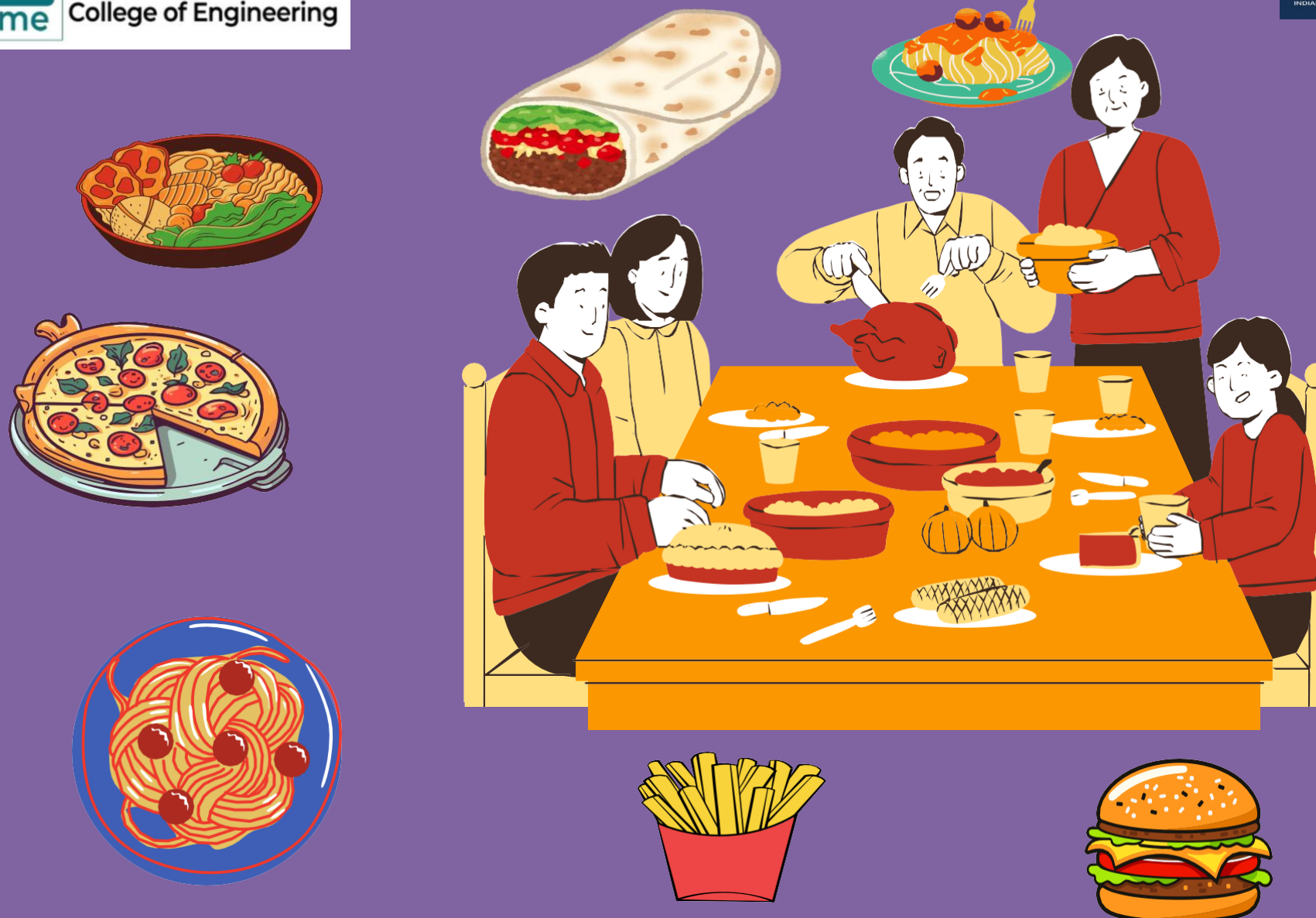
chef 2

chef 3





All chefs working in the same kitchen,
using the same fridge and pantry



OpenMP vs. Pthreads

Feature	OpenMP	Pthreads
Ease of Use	Very easy—just add <code>#pragma omp</code>	Complex—you manually create and manage threads
Control	Compiler handles thread creation & sync	You control everything (creation, sync, etc.)
Code Style	Looks like regular serial code	Requires lots of boilerplate and setup
Parallelism Type	Implicit (compiler decides)	Explicit (you decide everything)
Best For	Quick parallelization of loops & tasks	Fine-grained control over thread behavior
Debugging	Easier—less thread management	Harder—more chances for race conditions

Getting Started with OpenMP

OpenMP is **directives-based**.

In C/C++ → uses **pragmas** (special compiler instructions).

If a compiler doesn't support OpenMP, it just ignores them, and the program runs sequentially.
So, an OpenMP program can run both with and without OpenMP.

```
#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 */
```

“

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

”

OpenMP Terminology

Master thread: the original thread (thread 0).

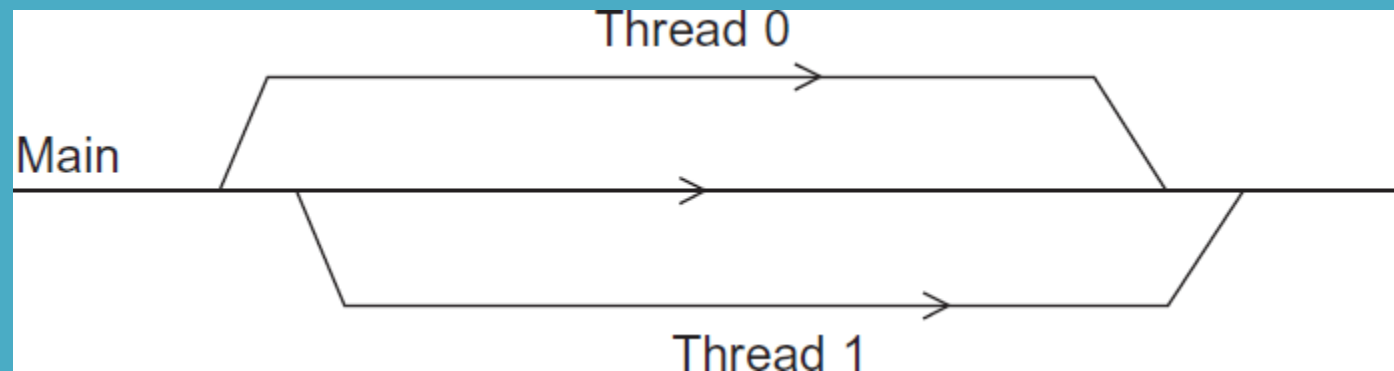
Parent thread: the one that started new threads.

Child threads: threads created by parent.
threads must finish before continuing.

Team: group of threads executing parallel block.

At the end of a parallel block → implicit **barrier**: all

Running the threads



Main thread forks and joins two threads.

Error checking

If **OpenMP is available**, each thread can compute the area of a chunk of trapezoids and then combine their results.

If **OpenMP is not available**, the code assumes a **single-threaded fallback**:

The thread rank is set to 0.

The total number of threads is 1.

The Hello() function runs just once.

```
if (argc != 2) {  
    fprintf(stderr, "Usage: %s <number of threads>\n", argv[0]);  
    exit(1);  
}  
long thread_count = strtol(argv[1], NULL, 10);  
if (thread_count <= 0) {  
    fprintf(stderr, "Thread count must be positive\n");  
    exit(1);  
}
```

Compiler Support Check with `_OPENMP`

Use preprocessor guards to avoid compilation errors on unsupported compilers:

Header Inclusion

```
#ifdef _OPENMP
```

Optional: Runtime Thread Count Check

```
if (omp_get_num_threads() != thread_count) {  
    fprintf(stderr, "Warning: Requested %ld threads, but got %d\n", thread_count, omp_get_num_threads());  
}
```

5.2 The Trapezoidal Rule

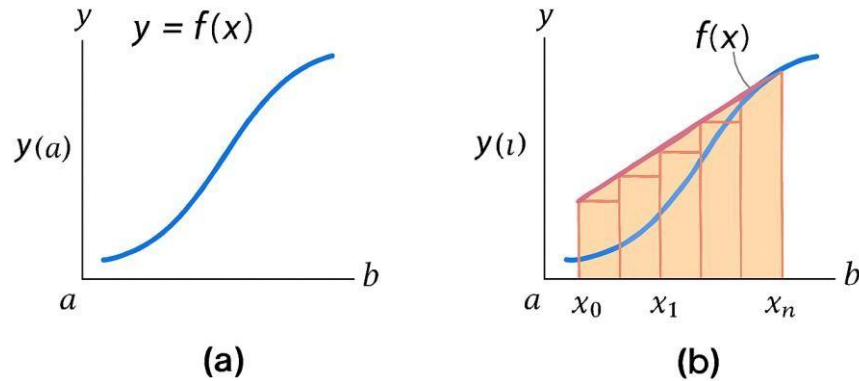


FIGURE 5.3: The trapezoidal rule.

```
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;
```

3. How does the formula work?

- Start with the boundary points: $\frac{f(a)+f(b)}{2}$
- Add values at all interior points: $f(x_1) + f(x_2) + \cdots + f(x_{n-1})$
- Multiply by step size h .

That gives the trapezoidal approximation.

$$x_i = a + i \cdot h \quad \text{for } i = 1, 2, \dots, n-1$$

- Example: If $a = 0$, $b = 1$, and $n = 4$:
 - $h = \frac{1-0}{4} = 0.25$
 - $x_1 = 0 + 1 \cdot 0.25 = 0.25$
 - $x_2 = 0 + 2 \cdot 0.25 = 0.50$
 - $x_3 = 0 + 3 \cdot 0.25 = 0.75$

These are the points where you evaluate $f(x)$ inside the interval.

Variable Scope in OpenMP vs. Serial Programming

Function-wide scope

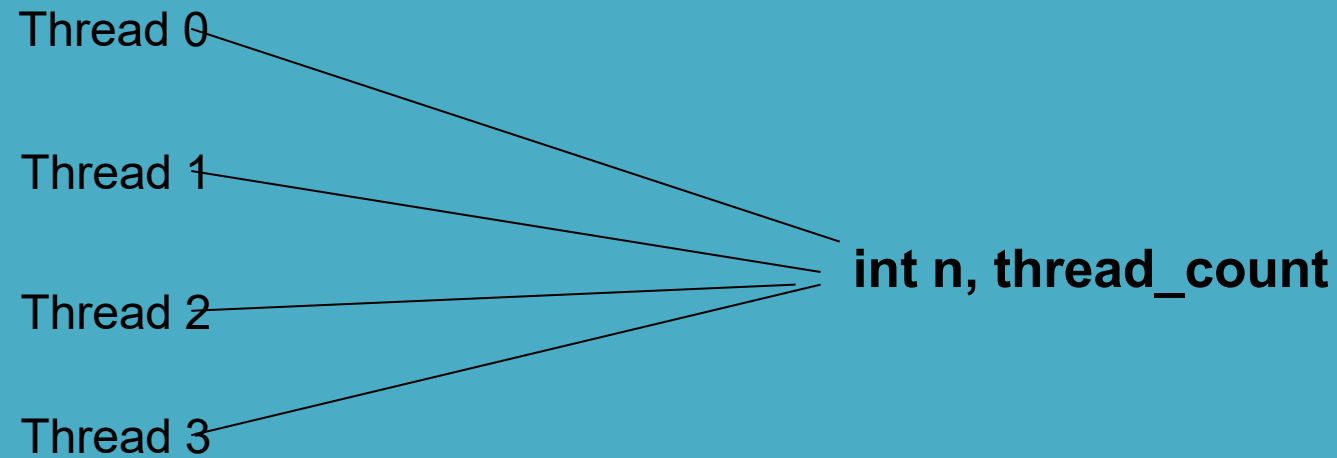
File-wide scope

OpenMP Scope: Thread Accessibility

```
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);  
}
```

Trapezoidal Rule Example

shared scope:



Variables in Trap():

Thread 0	double local_a, local_b, my_result; int local_n;
Thread 1	double local_a, local_b, my_result; int local_n;
Thread 2	double local_a, local_b, my_result; int local_n;
Thread 3	double local_a, local_b, my_result; int local_n;

Private Scope

Pointe———— Shared Variable

```
double* global_result_p;  
*global_result_p += my_result;
```

#pragma omp critical to avoid race conditions.

Why Scope Matters

If global_result were private:??

Before parallel block: variables are **shared** by default.

Inside parallel block or called functions: variables are **private** by default.

Shared variables: retain their values across the parallel block.

Private variables: are isolated per thread and discarded after the block.

Thank You