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):

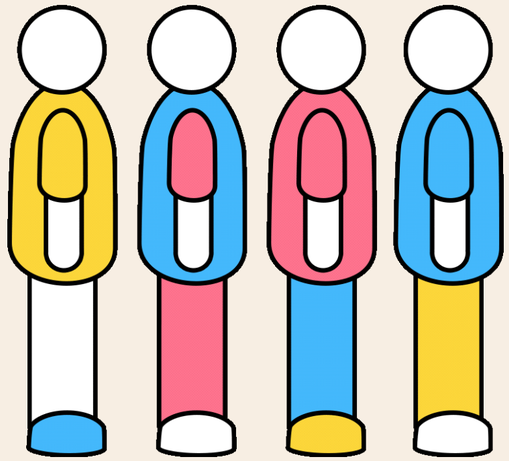
At the end of the course, the student will be able to:

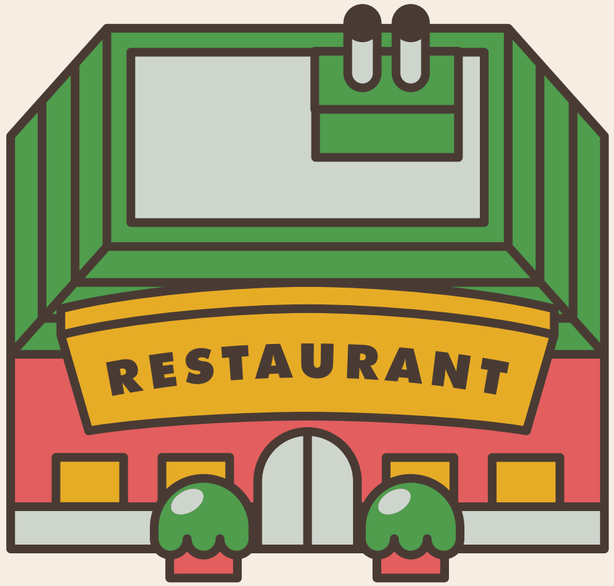
- Explain the need for parallel programming
- Demonstrate parallelism in MIMD system.
- Apply MPI library to parallelize the code to solve the given problem.
- Apply OpenMP pragma and directives to parallelize the code to solve the given problem
- Design a CUDA program for the given problem.

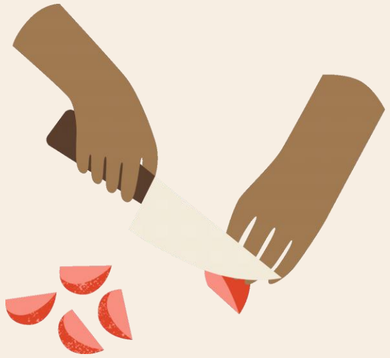
recall it.

MODULE-1

Introduction to parallel programming, Parallel hardware and parallel software – Classifications of parallel computers, SIMD systems, MIMD systems, Interconnection networks, Cache coherence, Shared-memory vs. distributed-memory, Coordinating the processes/threads, Shared-memory, Distributed-memory.







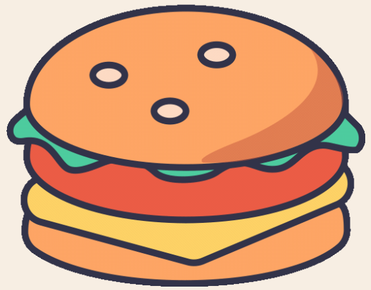
Pizza

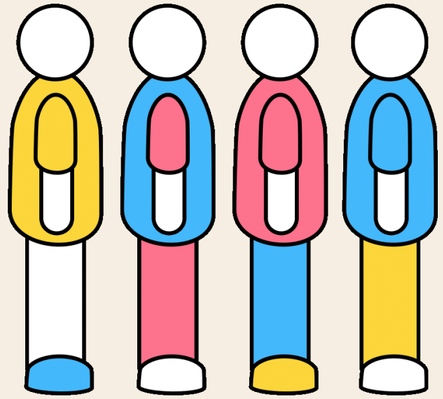
Burger

Pasta

Cut

fruits





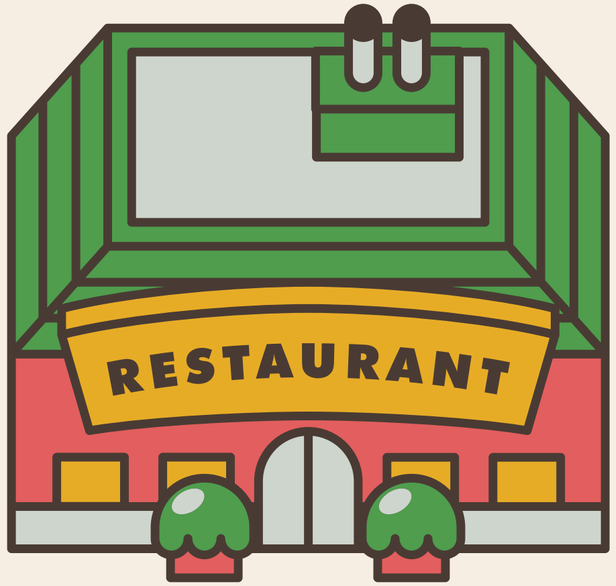
Pizza

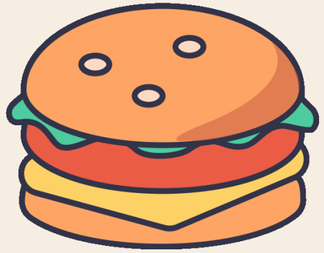
Burger

Pasta

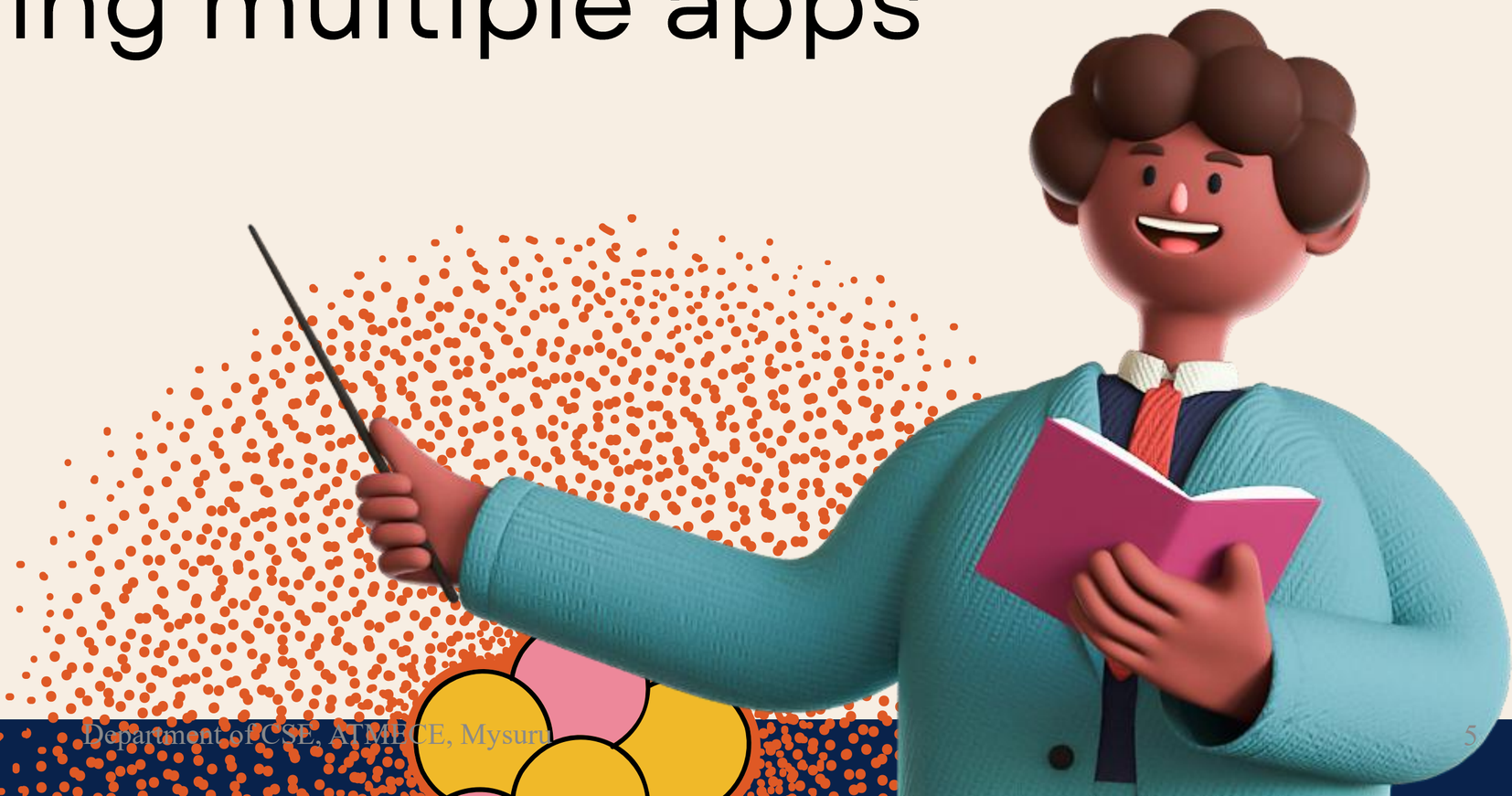
Cut

fruits





"Have you ever noticed your phone running multiple apps at once?"



Video streaming (Netflix downloading + playing simultaneously)

Gaming consoles with multiple players running on separate

AGENDA

1. Introduction to Parallel Computing
2. Why It’s Needed
3. History and Motivation
4. Real-World Applications
5. Basic Concepts

“Parallel Computing means performing many tasks simultaneously”

INCREASING DATA

CPU CLOCK SPEED REACHED LIMITS

HISTORY

REAL WORLD EXAMPLES

- Weather forecasting
- Space Exploration
- Online Platforms
- AI training chatgpt using GPUs
- autonomous vehicles

- how multiple apps can run simultaneously on a computer or phone and what happens internally

MULTITASKING

CPU CORE

OS SCHEDULER

HOW TASKS ARE ALLOCATED TO THE CORES OF THE CPU?

A task can be: A Process
A thread

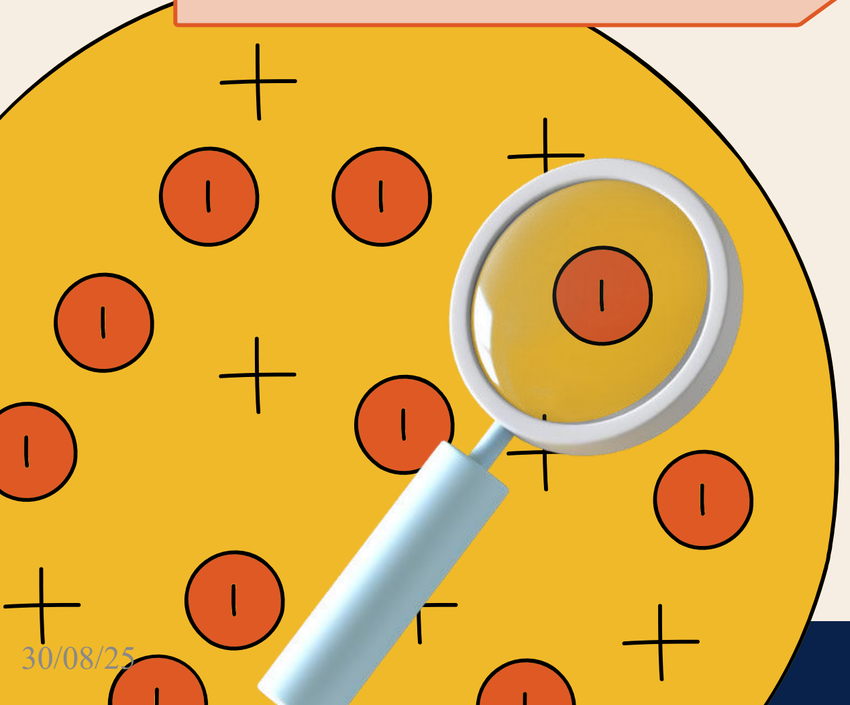
1. **TASK CREATED:**
2. **READY QUEUE:**
3. **CORE AVAILABILITY:**
4. **CONTEXT SWITCHING:**
5. **EXECUTION**
6. **MIGRATION**

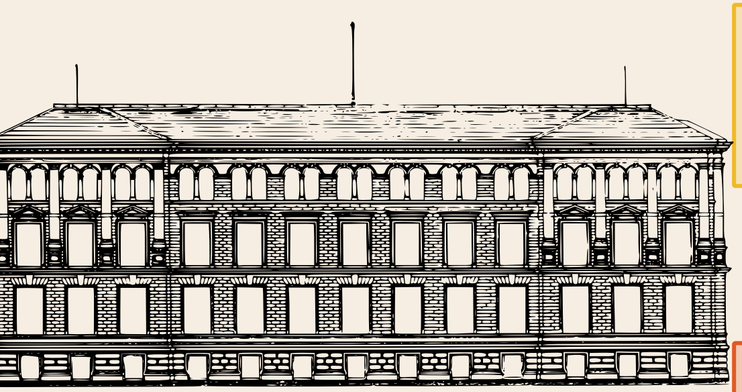
BASIC CONCEPTS

TYPES

Data parallelism

Task parallelism





PARALLEL ARCHITECTURES

- Shared Memory
- Distributed Memory

Why We Need to Write Parallel Programs

1. The problem with old (serial) programs

- Most existing programs were written for **single-core systems**.
- On a multicore system, you can run **multiple instances** of the same program (e.g., run 4 games at once), but that's **not useful** — users want one program to run **faster and better**, not more copies.
- Therefore: To use multiple cores effectively, programs must be **parallelized**.

2. Automatic conversion isn’t enough

- Researchers have tried to create compilers that **translate serial code to parallel code**.
- Success has been **limited** because:
  - Translating each step independently into parallel code often leads to **inefficiency**.
  - Sometimes the **best parallel solution requires a completely new algorithm**, not just a step-by-step parallelization of the serial one.
  - Example: Matrix multiplication — turning it into parallel dot-products may be inefficient compared to designing a new parallel matrix multiplication algorithm.

Serial code(one core): Summation

```
sum = 0;
for (i = 0; i < n; i++) {
    x = ComputeNextValue(...);
    sum += x;
}
```

The second method: Pairwise (tree-style) reduction

- Instead of all cores sending to the master, we **combine results in stages**:
  - **Stage 1**: Pair the cores:
    - Core 0 + Core 1, Core 2 + Core 3, Core 4 + Core 5, Core 6 + Core 7.
  - **Stage 2**: Pair the winners (even-numbered cores now hold results):
    - Core 0 + Core 2, Core 4 + Core 6.
  - **Stage 3**: Final combination:
    - Core 0 + Core 4.

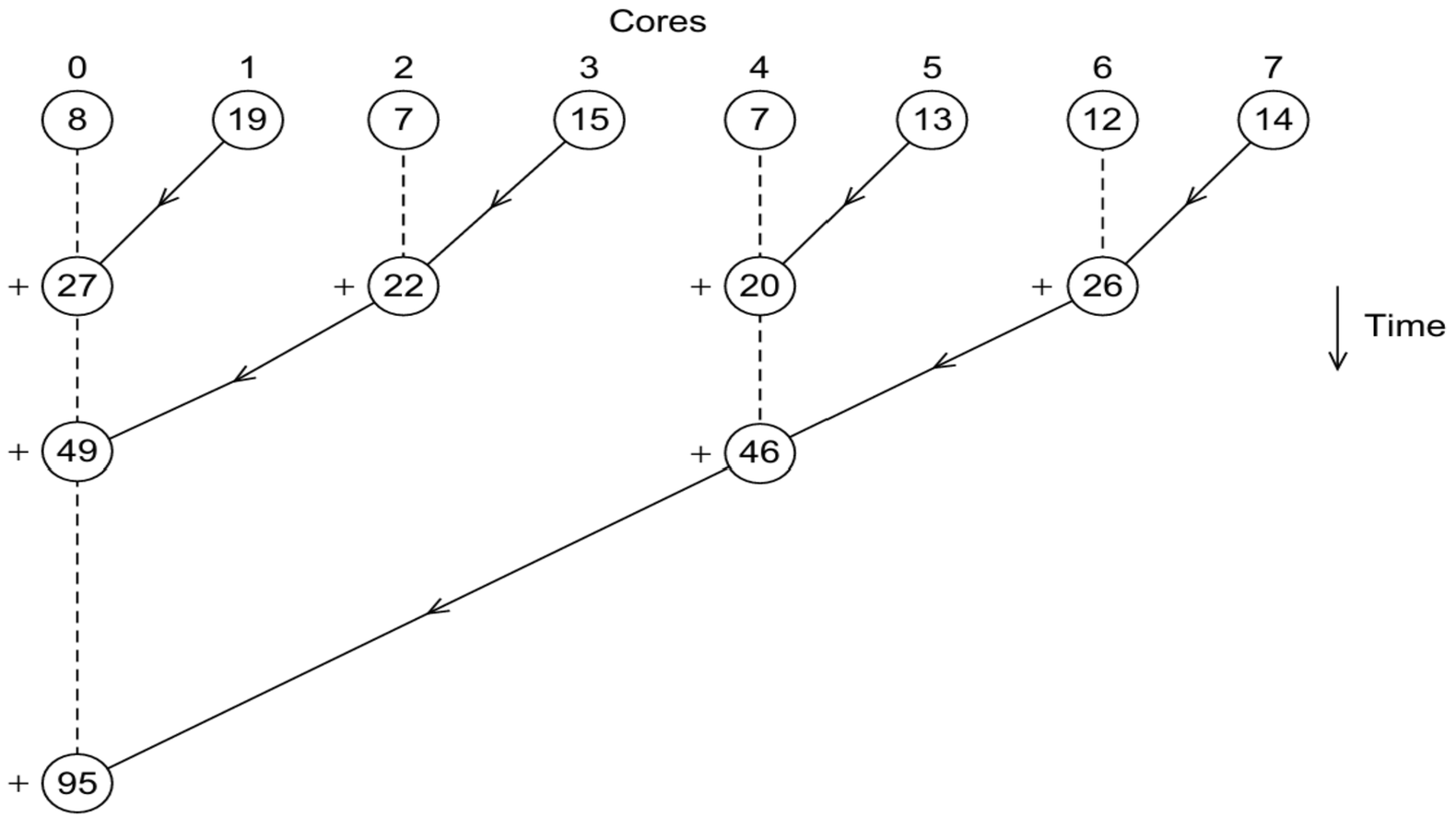


FIGURE 1.1

Multiple cores forming a global sum

Comparing the two global sum methods

- Method 1 (naïve / centralized):
  - Master adds up results from all cores.
  - Needs p − 1 operations (e.g., 999 adds for 1000 cores).

- Method 2 (tree reduction):
  - Results combined in pairs over stages.
  - Needs $\log_2(p)$ operations (e.g., only 10 adds for 1000 cores).
  - Much more efficient, especially as p grows.

how we actually write parallel programs and the main challenges

Two Main Approaches to Parallelism

1. Task Parallelism

- Different cores do *different tasks*.
- Example: In grading exams, one person grades only Question 1 (Shakespeare), another grades Question 2 (Milton), and so on.
- Each is doing a *different job*, so the instructions differ.

2. Data Parallelism

Different cores do the *same task* on *different pieces of data*.

Example: Split the 100 exam papers into 5 piles of 20. Each TA grades *all questions* on their pile. Same instructions, but applied to different data.

Von Neumann Architecture (Classical Computer Design)

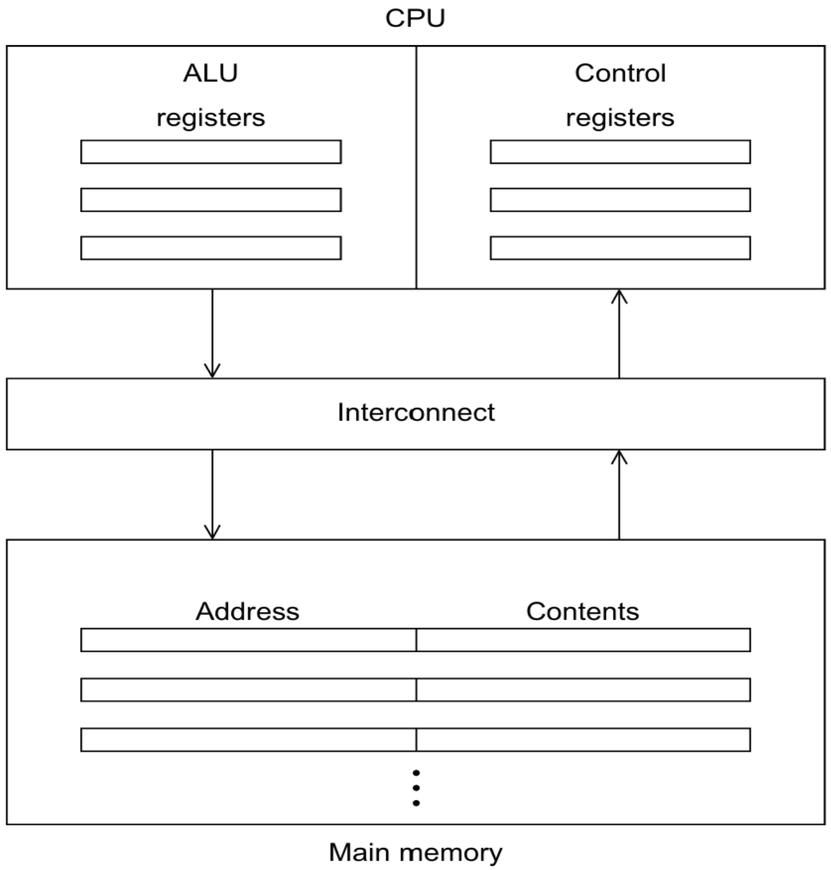


FIGURE 2.1

The von Neumann architecture

Von Neumann Bottleneck

- Problem: The CPU is much faster than the memory access speed.
- CPU may execute 100+ instructions in the time it takes to fetch one piece of data from memory.
- The bus/interconnect limits how quickly data & instructions travel.

Analogy:

- CPU is like factory making products.
- Memory is warehouse storing raw materials (data) and finished products (results).
- Road (bus) is the transport system between them.
- If the road is too narrow (limited bandwidth), the factory workers sit idle because raw materials arrive too slowly.

Modifications to the von Neumann model

The **von Neumann bottleneck** means the CPU is very fast, but memory (RAM) is much slower. Since the CPU often has to wait for memory, overall performance suffers.

To fix this, computer engineers added **caching, virtual memory, and parallelism**. This part is about **caching**.

What is caching?

Think of it like this:

- CPU = **factory**
- Main memory = **warehouse**
- Road between them = **slow, two-lane road**

The CPU constantly needs raw materials (data & instructions) from memory. If every time it has to go to the warehouse far away, it wastes time.

Solution is to Build a small storeroom (cache) right next to the CPU. Cache stores a small amount of data that the CPU is very likely to need soon. It's much faster to access than main memory.

Cache mappings

1. Fully Associative
2. Direct Mapped
3. N-way Set Associative

Locality:

- **Spatial locality**: if you use A[0][0], you'll probably use A[0][1], A[0][2] soon.
- **Temporal locality**: if you use a value once, you might use it again later.

```
for (i = 0; i < MAX; i++)
    for (j = 0; j < MAX; j++)
        y[i] += A[i][j] * x[j];
```

- Access pattern: row by row → contiguous memory.
- Example (MAX=4):
  - Access order: A[0][0], A[0][1], A[0][2], A[0][3] (all in **same cache line** → only **1 miss**).
  - Next row: A[1][0]...A[1][3] → again, just **1 miss**.

virtual memory

Virtual memory gives:

Illusion of large memory (even bigger than RAM, because disk is used as backup).

Protection (one program cannot overwrite another's memory).

Flexibility (any program can use any free RAM block).

Memory is divided into **pages** (usually 4 KB–16 KB).

Disk also has **swap space** divided into same-sized **pages**.

A program uses **virtual addresses** → these get mapped to **physical addresses** in RAM.

Table 2.3 Pipelined Addition. Numbers in the Table Are Subscripts of Operands/Results

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

Distributed-memory interconnects

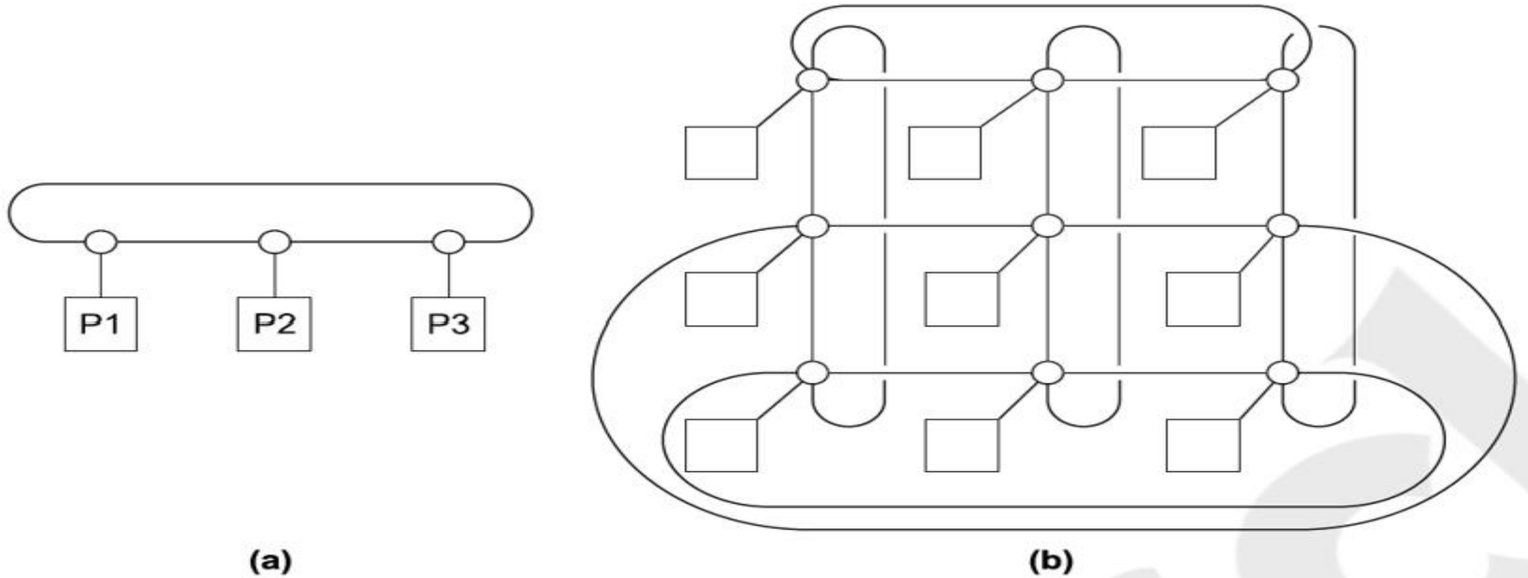


FIGURE 2.8

(a) A ring and (b) a toroidal mesh.

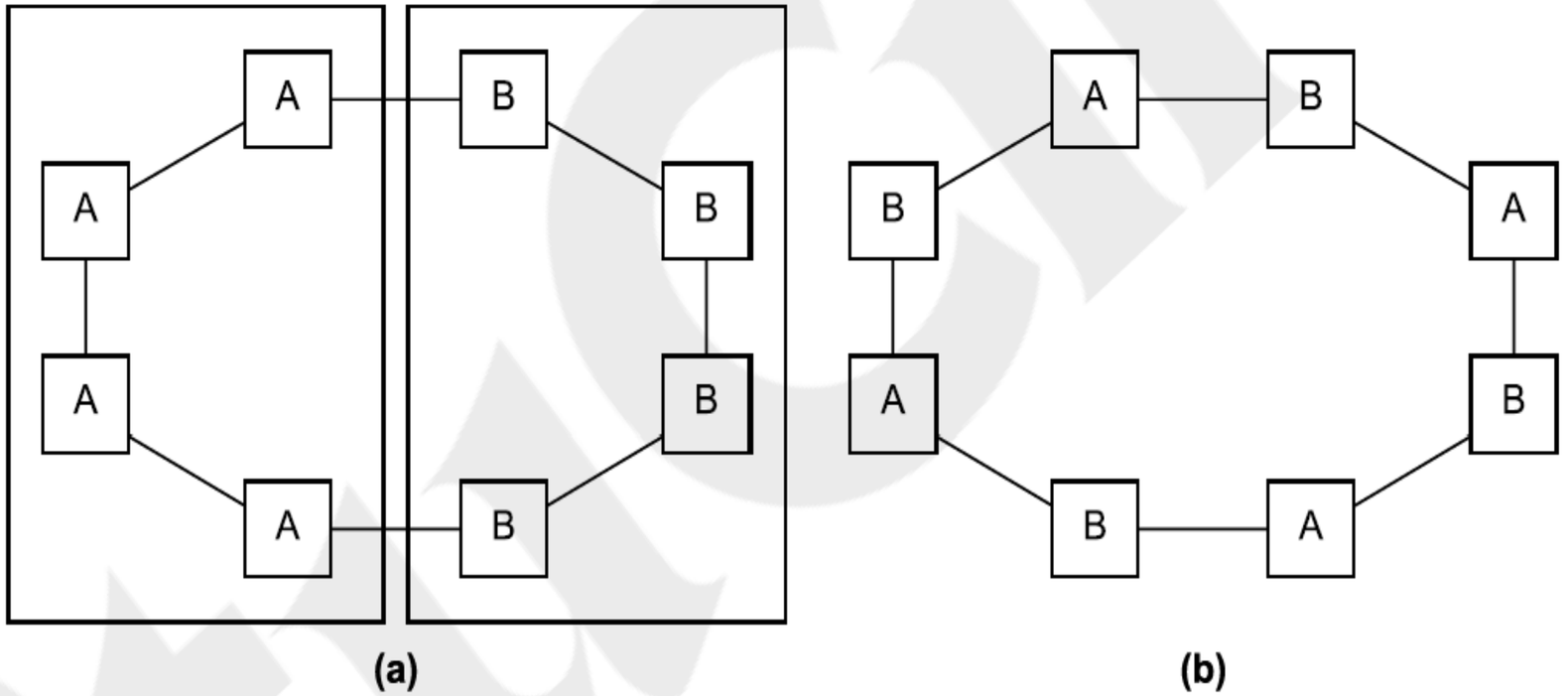


FIGURE 2.9

Two bisections of a ring: (a) only two communications can take place between the halves and (b) four simultaneous connections can take place.

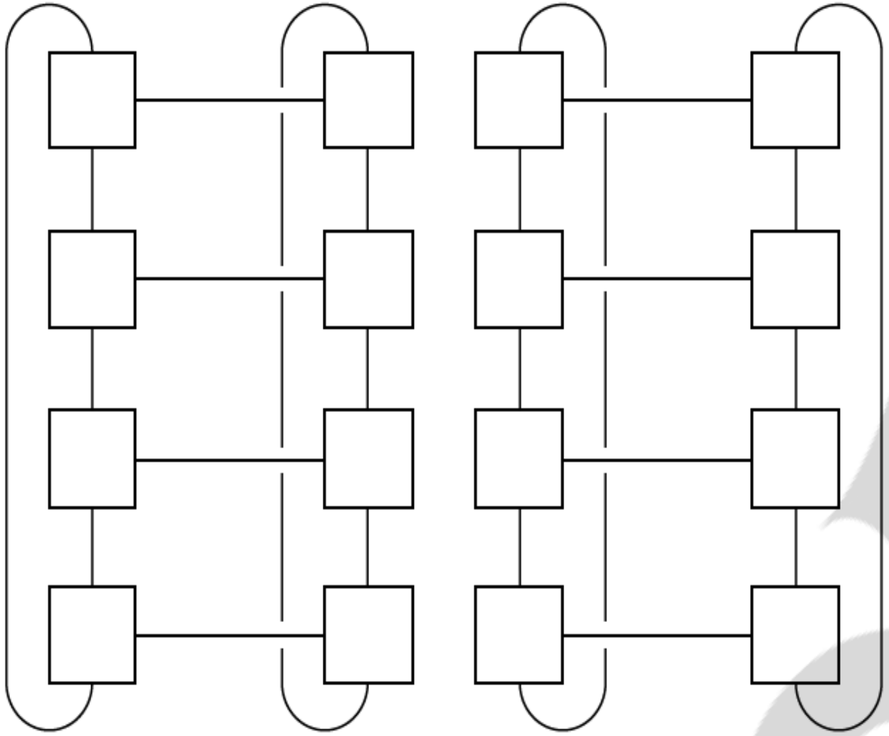


FIGURE 2.10

A bisection of a toroidal mesh.

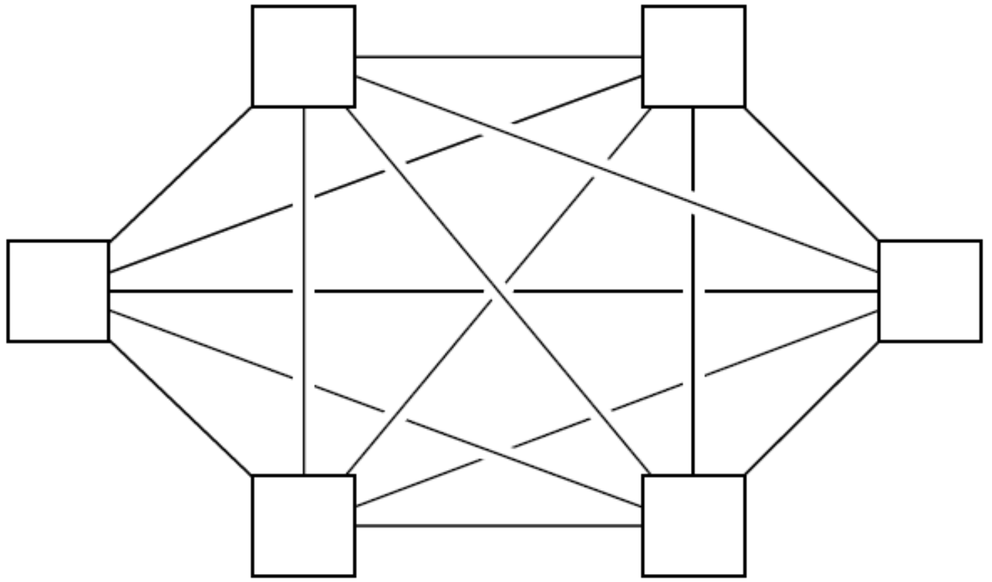


FIGURE 2.11

A fully connected network.

Cache coherence: CPU caches are managed by system hardware: programmers don't have direct control over them. This has several important consequences for shared-memory systems.

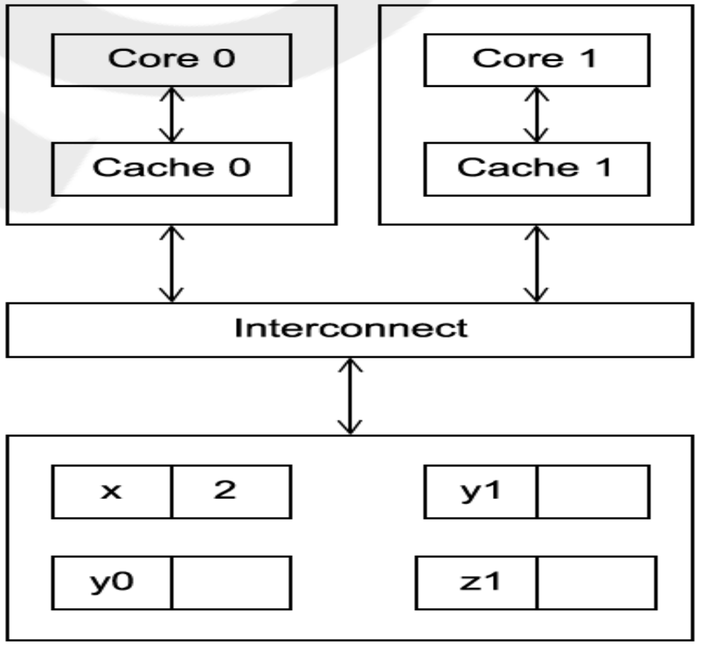


FIGURE 2.17

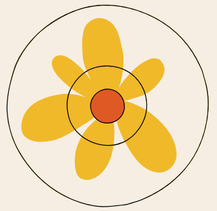
A shared-memory system with two cores and two caches.

Nondeterminism?

- In **MIMD systems** (Multiple Instruction, Multiple Data) where multiple processors (or threads) run at the same time, they usually don't stay perfectly in sync.

- This means **the same input** might produce **different outputs** depending on how the processors finish their tasks. This unpredictability is called **nondeterminism**.

Thank You



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-2

GPU programming, Programming hybrid systems, MIMD systems, GPUs, Performance – Speedup and efficiency in MIMD systems, Amdahl's law, Scalability in MIMD systems, Taking timings of MIMD programs, GPU performance.

GPU programming

GPUs are usually not “standalone” processors. They don’t ordinarily run an operating system and system services, such as direct access to secondary storage.

So, program ming a GPU also involves writing code for the CPU “host” system, which runs on an ordinary CPU. The memory for the CPU host and the GPU memory are usually separate. So, the code that runs on the host typically allocates and initializes storage on both the CPU and the GPU.

It will start the program on the GPU, and it is responsible for the output of the results of the GPU program. Thus, GPU programming is really heterogeneous programming, since it involves programming two different types of processors.

```
// Thread private variables
int rank_in_gp, my_x;
...
if (rank_in_gp < 16)
    my_x += 1;
else
    my_x -= 1;
```

Then the threads with rank < 16 will execute the first assignment, while the threads with rank ≥ 16areidle.

MIMD systems

We've generally avoided the issue of input and output. There are a couple of reasons.

First and foremost, parallel I/O, in which multiple cores access multiple disks or other devices, is a subject to which one could easily devote a book.

When we call printf from multiple processes, we, as developers, would like the output to appear on the console of a single system, the system on which we started the program.

GPUs

In most cases, the host code in our GPU programs will carry out all I/O. Since we'll only be running one process/thread on the host, the standard C I/O functions should behave as they do in ordinary serial C programs.

The exception to the rule that we use the host for I/O is that when we are debug ging our GPU code, we'll want to be able to write to stdout and/or stderr. In the systems we use, each thread can write to stdout, and, as with MIMD programs, the order of the output is nondeterministic. Also, in the systems we use, no GPU thread has access to stderr, stdin, or secondary storage.

PERFORMANCE

1. Speedup and efficiency in MIMD systems

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}},$$

This value, S/p, is sometimes called the efficiency of the parallel program. If we substitute the formula for S, we see that the efficiency is

$$E = \frac{S}{p} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{p} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}}.$$

Table 2.4 Speedups and Efficiencies of a Parallel Program

p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
E = S/p	1.0	0.95	0.90	0.81	0.68

Table 2.5 Speedups and Efficiencies of a Parallel Program on Different Problem Sizes

	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

Many parallel programs are developed by dividing the work of the serial program among the processes/threads and adding in the nec-essary “parallel overhead” such as mutual exclusion or communication.

Therefore, if *T*overhead denotes this parallel overhead, it’s often the case that

$$T_{\text{parallel}} = T_{\text{serial}}/p + T_{\text{overhead}}.$$

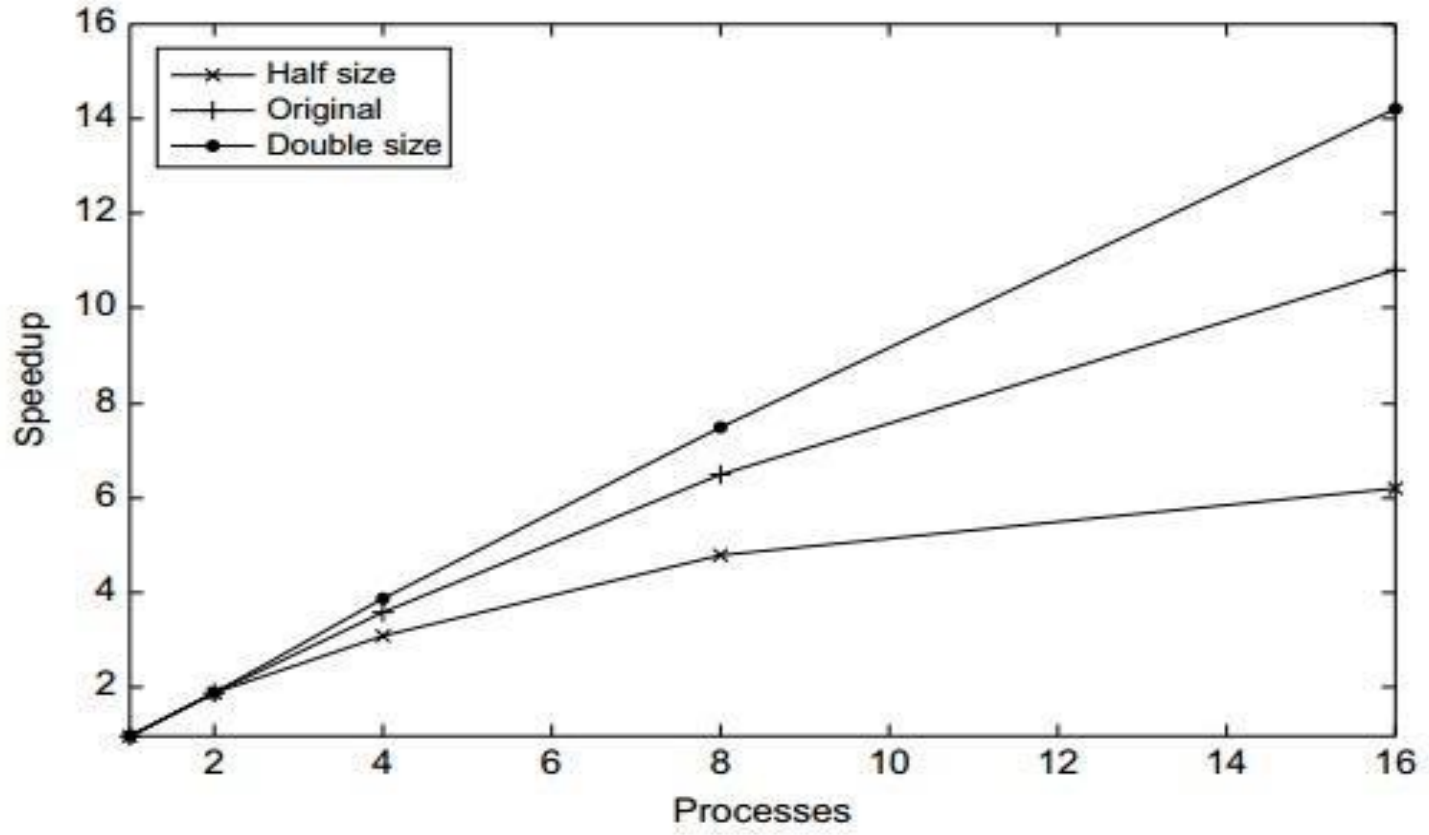


FIGURE 2.18

Speedups of parallel program on different problem sizes

2. Amdahl's law

$$T_{\text{parallel}} = 0.9 \times T_{\text{serial}}/p + 0.1 \times T_{\text{serial}} = 18/p + 2,$$

and the speedup will be

$$S = \frac{T_{\text{serial}}}{0.9 \times T_{\text{serial}}/p + 0.1 \times T_{\text{serial}}} = \frac{20}{18/p + 2}.$$

Now as p gets larger and larger, $0.9\ T_{serial}/p = 18/p$ gets closer and closer to 0, so the total parallel run-time can't be smaller than $0.1\ T_{serial} = 2$. That is, the denominator in S can't be

smaller than $0.1\ T_{serial} = 2$. The fraction S must therefore be smaller than

$$T_{parallel} = 0.9 \times T_{serial}/p + 0.1 \times T_{serial} = 18/p + 2,$$

and the speedup will be

$$S = \frac{T_{serial}}{0.9 \times T_{serial}/p + 0.1 \times T_{serial}} = \frac{20}{18/p + 2}.$$

To see if the program is scalable, we inc

3. Scalability in MIMD systems

As an example, suppose that *T*serial = *n*, where the units of *T*serial are in microseconds,

and *n* is also the problem size. Also suppose that *T*parallel = *n*/*p* + 1. Then

$$E = \frac{n}{p(n/p+1)} = \frac{n}{n+p}.$$

4. Taking timings of MIMD programs

The first thing to note is that there are at least two different reasons for taking timings. During program development we may take timings in order to determine if the program is behaving as we intend.

Second, we're usually *not* interested in the time that elapses between the pro-gram's start and the program's finish.

Third, we're usually *not* interested in "CPU time." This is the time reported by the standard C function clock.

5 GPU performance

It's quite common to see reported speedups of GPU programs over serial programs or parallel MIMD programs.

Since efficiency of a GPU program relative to a CPU program doesn't make sense, the formal definition of the scalability of a MIMD program can't be applied to a GPU program.

It should be noted that the same caveats that apply to Amdahl's law on MIMD systems also apply to Amdahl's law on GPUs:

Thank you

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):

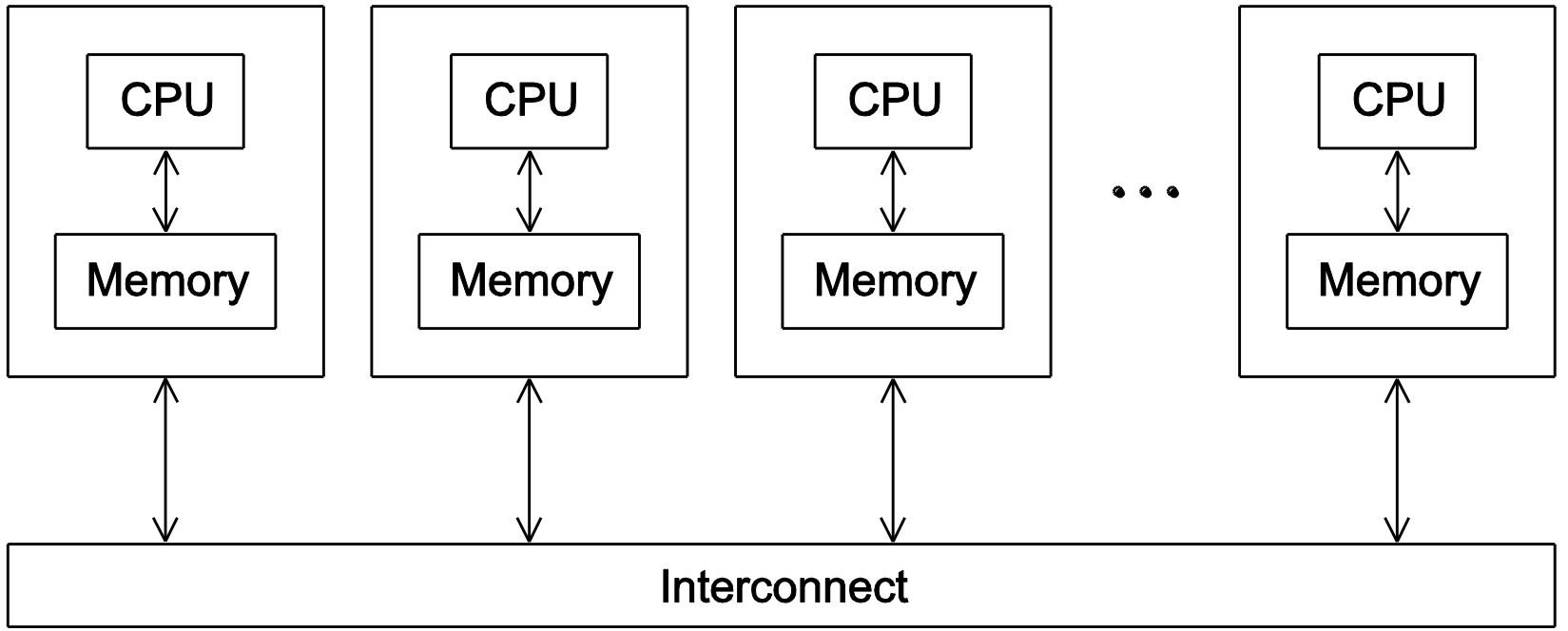
At the end of the course, the student will be able to:

- 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.

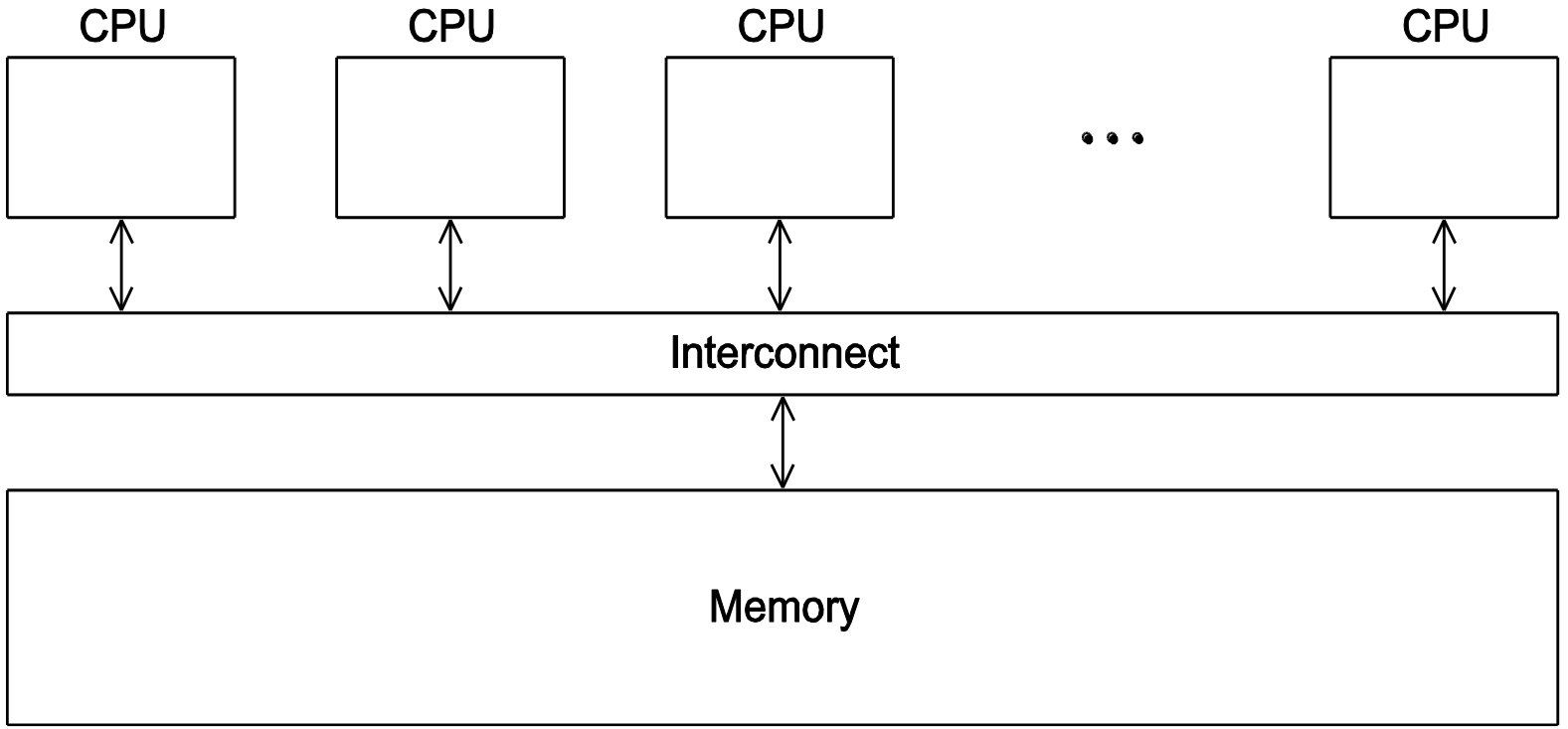
MODULE-3

Distributed memory programming with MPI – MPI functions, The trapezoidal rule in MPI, Dealing with I/O, Collective communication, MPI-derived datatypes, Performance evaluation of MPI programs, A parallel sorting algorithm.

A distributed memory system



A shared memory system



Hello World!

```c
#include <stdio.h>

int main(void) {
    printf("hello, world\n");

    return 0;
}
```

Identifying MPI processes

Common practice to identify processes by nonnegative integer ranks.

p processes are numbered *0, 1, 2, .. p-1*

Compilation

wrapper script to compile

source file

produce debugging information

create this executable file name (as opposed to default a.out)

turns on all warnings

mpiexec -n <number of processes> <executable>

---

mpiexec -n 1 ./mpi_hello

run with 1 process

Execution

mpiexec -n 4 ./mpi_hello

run with 4 processes

```
mpiexec  -n  1  ./mpi_hello

    Greetings from process 0 of 1 !
```

```
mpiexec  -n  4  ./mpi_hello

    Greetings from process 0 of 4 !
    Greetings from process 1 of 4 !
    Greetings from process 2 of 4 !
    Greetings from process 3 of 4 !
```

MPI Programs

Written in C.

- Has main.
- Uses stdio.h, string.h, etc.

Need to add mpi.h header file.

Identifiers defined by MPI start with “MPI_”.

First letter following underscore is uppercase.

- For function names and MPI-defined types.
- Helps to avoid confusion.

MPI Components

MPI_Init

Tells MPI to do all the necessary setup.

```
int MPI_Init(
      int*      argc_p  /* in/out */,
      char***   argv_p  /* in/out */);
```

MPI_Finalize

Tells MPI we're done, so clean up anything allocated for this program.

```
int MPI_Finalize(void);
```

```
. . .
#include <mpi.h>
. . .
int main(int argc, char* argv[]) {
   . . .
   /* No MPI calls before this */
   MPI_Init(&argc, &argv);
   . . .
   MPI_Finalize();
   /* No MPI calls after this */
   . . .
   return 0;
}
```

A collection of processes that can send messages to each other.

MPI_Init defines a communicator that consists of all the processes created when the program is started Called MPI_COMM_WORLD.

Communicators

```
int MPI_Comm_size(
      MPI_Comm   comm          /* in  */,
      int*       comm_sz_p     /* out */);
```

number of processes in the communicator

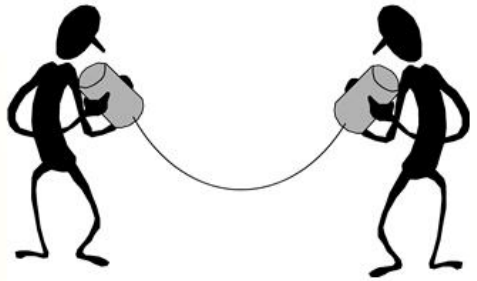
```
int MPI_Comm_rank(
      MPI_Comm   comm          /* in  */,
      int*       my_rank_p     /* out */);
```

my rank
(the process making this call)

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Data types

```
int MPI_Recv(
      void*          msg_buf_p     /* out */,
      int            buf_size      /* in  */,
      MPI_Datatype   buf_type      /* in  */,
      int            source        /* in  */,
      int            tag           /* in  */,
      MPI_Comm       communicator  /* in  */,
      MPI_Status*    status_p      /* out */);
```



```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,
         send_comm);
```

r

MPI_Send

src = q

MPI_Recv

dest = r

Message matching

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

q

Receiving messages

A receiver can get a message without knowing:

the amount of data in the message,

the sender of the message,

or the tag of the message.

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

MPI_Status*

MPI_SOURCE
MPI_TAG
MPI_ERROR

```
MPI_Status*  status;

status.MPI_SOURCE
status.MPI_TAG
```

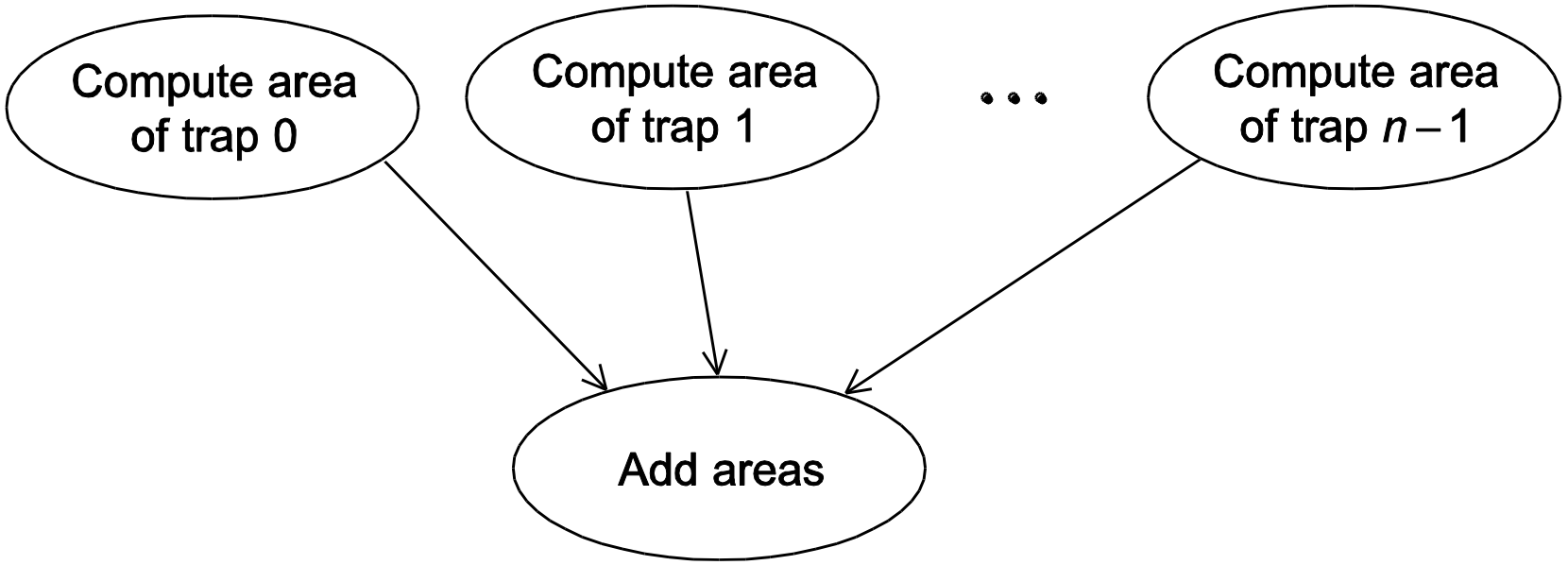
```
int MPI_Get_count(
      MPI_Status*   status_p   /* in  */,
      MPI_Datatype  type       /* in  */,
      int*          count_p    /* out */);
```

Exact behavior is determined by the MPI implementation.

MPI_Send may behave differently with regard to buffer size, cutoffs and blocking

```
/* 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;
```

Tasks and communications for Trapezoidal Rule

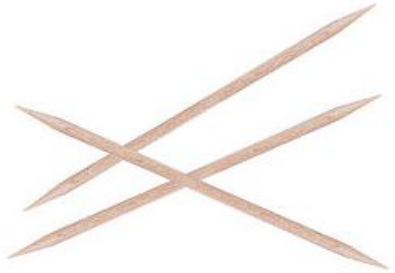


```c
#include <stdio.h>
#include <mpi.h>

int main(void) {
   int my_rank, comm_sz;

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   printf("Proc %d of %d > Does anyone have a toothpick?\n",
          my_rank, comm_sz);

   MPI_Finalize();
   return 0;
}  /* main */
```

Dealing with I/O

```
Proc 0 of 6 > Does anyone have a toothpick?
Proc 1 of 6 > Does anyone have a toothpick?
Proc 2 of 6 > Does anyone have a toothpick?
Proc 4 of 6 > Does anyone have a toothpick?
Proc 3 of 6 > Does anyone have a toothpick?
Proc 5 of 6 > Does anyone have a toothpick?
```

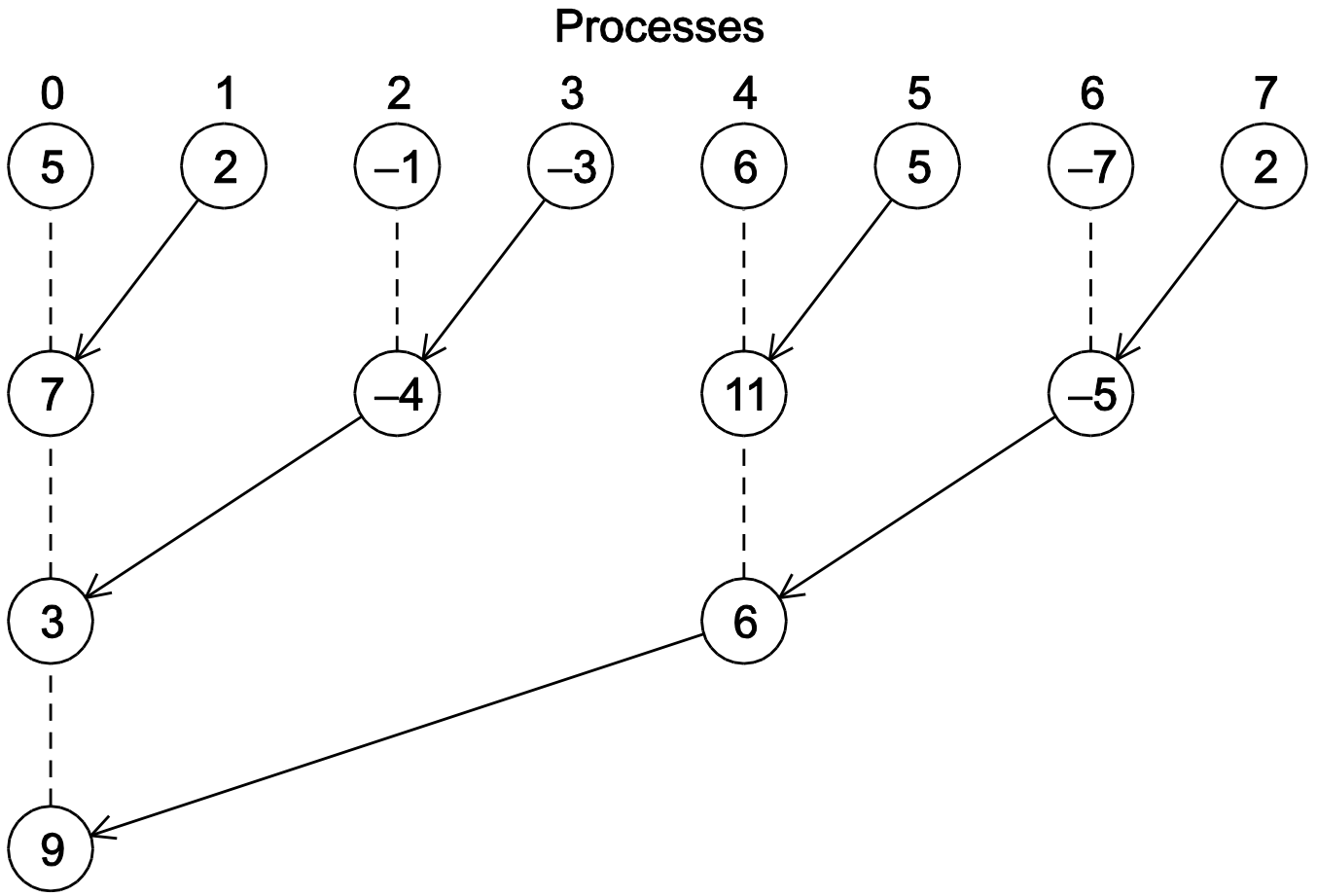
unpredictable output



```
. . .
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

Get_data(my_rank, comm_sz, &a, &b, &n);

h = (b-a)/n;
. . .
```

Input

Collective communication



A tree-structured global sum

MPI_Reduce

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

```
double local_x[N], sum[N];
. . .
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

Predefined reduction operators in MPI

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

Collective vs. Point-to-Point Communications

All the processes in the communicator must call the same collective function.

For example, a program that attempts to match a call to MPI_Reduce on one process with a call to MPI_Recv on another process is erroneous, and, in all likelihood, the program will hang or crash.

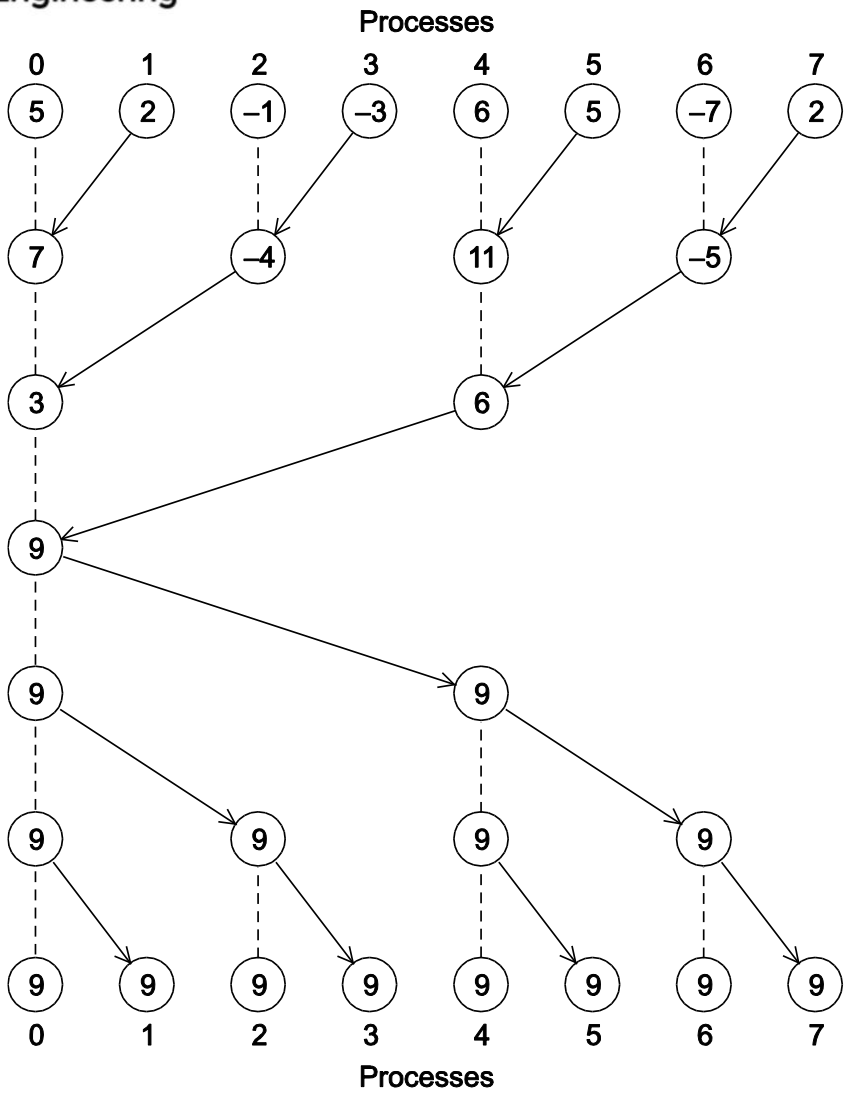
Example (1)

Time	Process 0	Process 1	Process 2
0	`a = 1; c = 2`	`a = 1; c = 2`	`a = 1; c = 2`
1	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`
2	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`

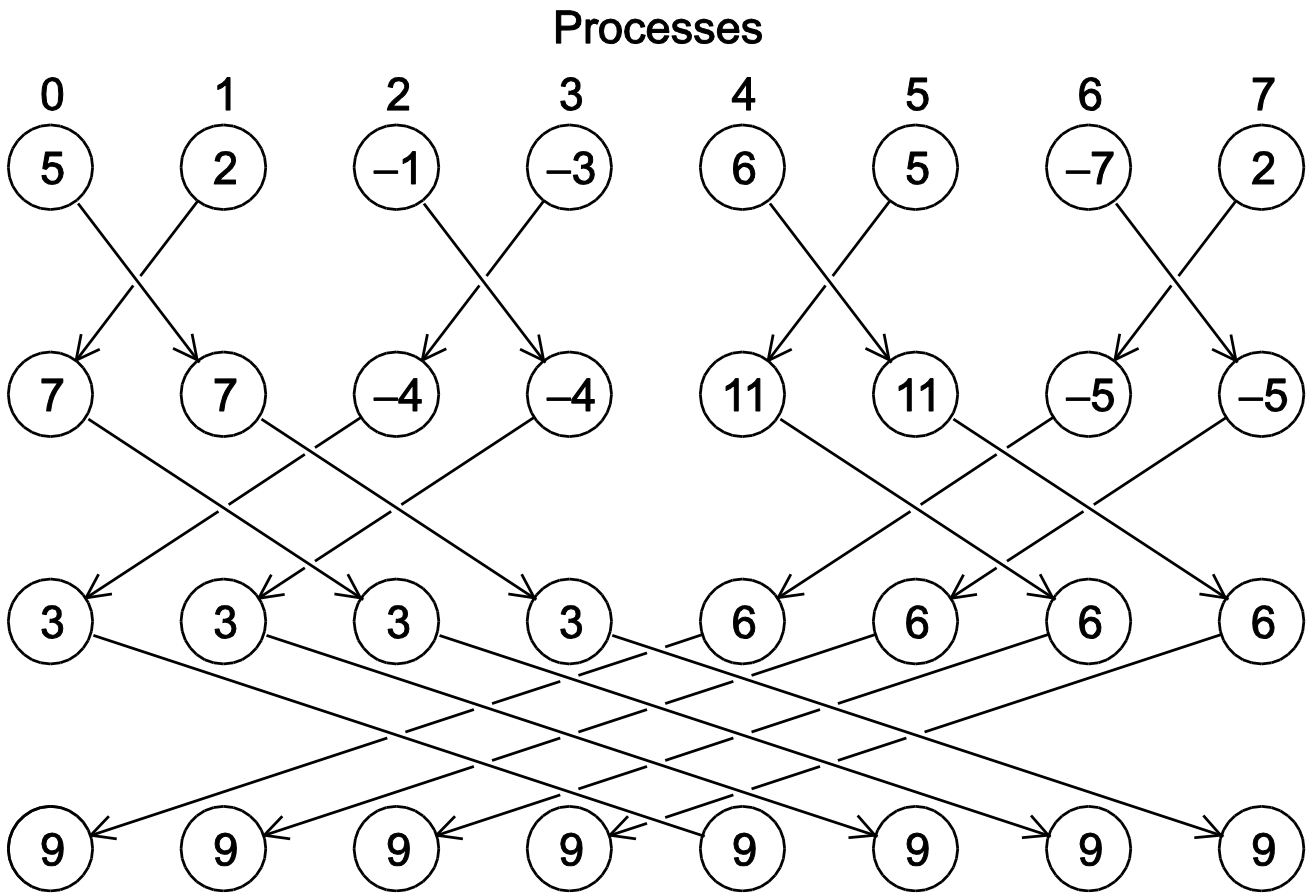
Multiple calls to MPI_Reduce

```
int MPI_Allreduce(
      void*         input_data_p   /* in  */,
      void*         output_data_p  /* out */,
      int           count          /* in  */,
      MPI_Datatype  datatype       /* in  */,
      MPI_Op        operator       /* in  */,
      MPI_Comm      comm           /* in  */);
```

MPI_Allreduce



A global sum followed by distribution of the result.

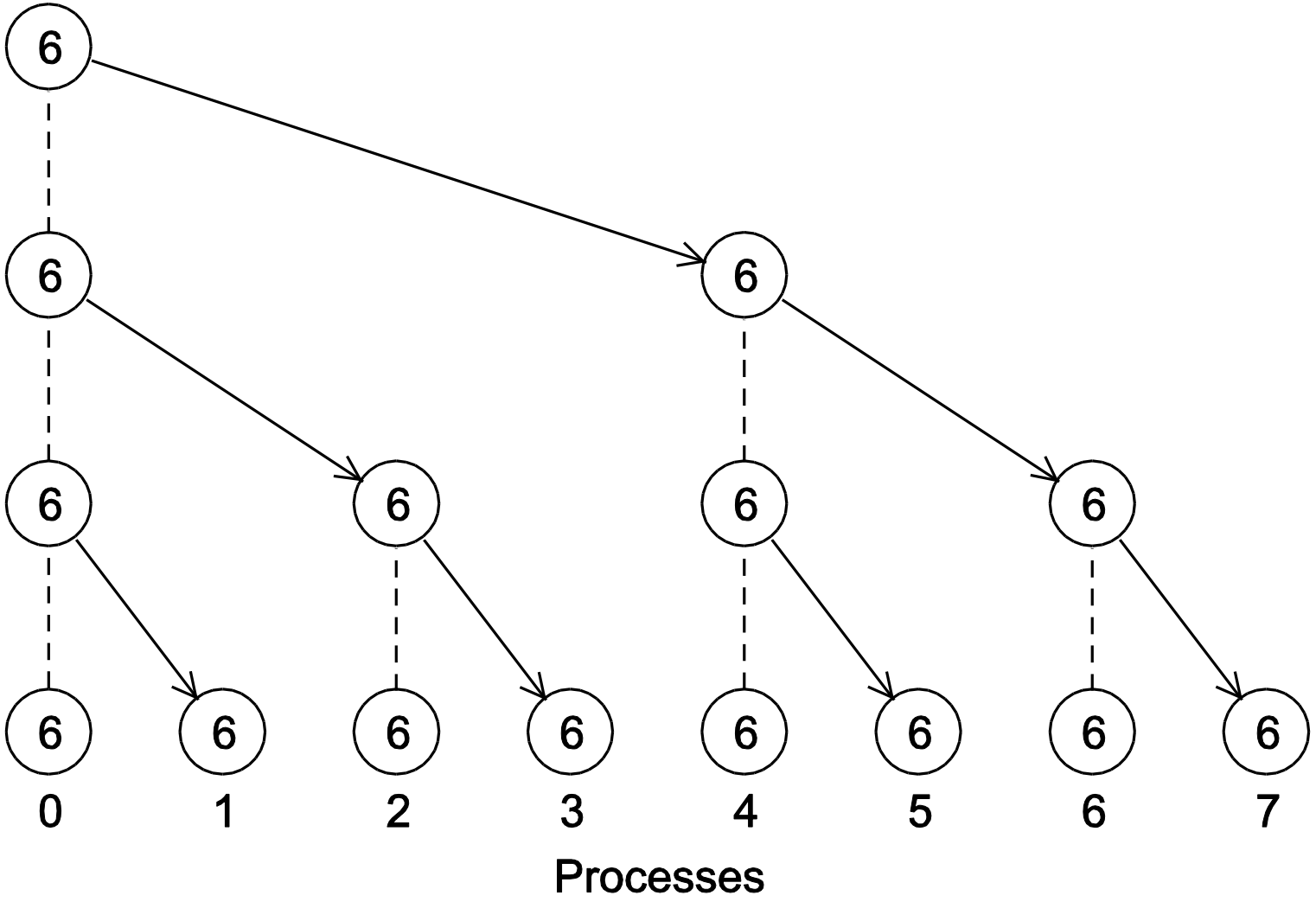


A butterfly-structured global sum.

Broadcast

```
int MPI_Bcast(
      void*         data_p       /* in/out */,
      int           count        /* in     */,
      MPI_Datatype  datatype     /* in     */,
      int           source_proc  /* in     */,
      MPI_Comm      comm         /* in     */);
```

A tree-structured broadcast.



$$
\begin{aligned}
\mathbf{x}+\mathbf{y} &= (x_0, x_1, \ldots, x_{n-1}) + (y_0, y_1, \ldots, y_{n-1}) \\
&= (x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1}) \\
&= (z_0, z_1, \ldots, z_{n-1}) \\
&= \mathbf{z}
\end{aligned}
$$

Compute a vector sum.

Different partitions of a 12-component vector among 3 processes

Process	Components											
	Block				Cyclic				Block-cyclic Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

Partitioning options

Block partitioning

Assign blocks of consecutive components to each process.

Cyclic partitioning

Assign components in a round robin fashion.

Block-cyclic partitioning

Use a cyclic distribution of blocks of components.

Scatter

```
int MPI_Scatter(
      void*         send_buf_p  /* in  */,
      int           send_count  /* in  */,
      MPI_Datatype  send_type   /* in  */,
      void*         recv_buf_p  /* out */,
      int           recv_count  /* in  */,
      MPI_Datatype  recv_type   /* in  */,
      int           src_proc    /* in  */,
      MPI_Comm      comm        /* in  */);
```

MPI_Scatter can be used in a function that reads in an entire vector on process 0 but only sends the needed components to each of the other processes.

```
int MPI_Gather(
      void*          send_buf_p   /* in  */,
      int            send_count   /* in  */,
      MPI_Datatype   send_type    /* in  */,
      void*          recv_buf_p   /* out */,
      int            recv_count   /* in  */,
      MPI_Datatype   recv_type    /* in  */,
      int            dest_proc    /* in  */,
      MPI_Comm       comm         /* in  */);
```

Collect all of the components of the vector onto process 0, and then process 0 can process all of the components.

Matrix-vector multiplication

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

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix}$$

stored as

0 1 2 3 4 5 6 7 8 9 10 11

Mpi derived datatypes

```
int MPI_Type_create_struct(
      int              count                          /* in  */,
      int              array_of_blocklengths[]        /* in  */,
      MPI_Aint         array_of_displacements[]       /* in  */,
      MPI_Datatype     array_of_types[]               /* in  */,
      MPI_Datatype*    new_type_p                     /* out */);
```

Performance evaluation

Elapsed parallel time

Elapsed serial time

MPI_Barrier

```
int MPI_Barrier(MPI_Comm    comm    /* in */);
```

Run-times of serial and parallel matrix-vector multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

$$S(n,p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n,p)}$$

Speedup

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_{\text{serial}}(n)}{p \times T_{\text{parallel}}(n,p)}$$

Efficiency

A parallel sorting algorithm

Serial odd-even transposition sort

Parallel odd-even transposition sort

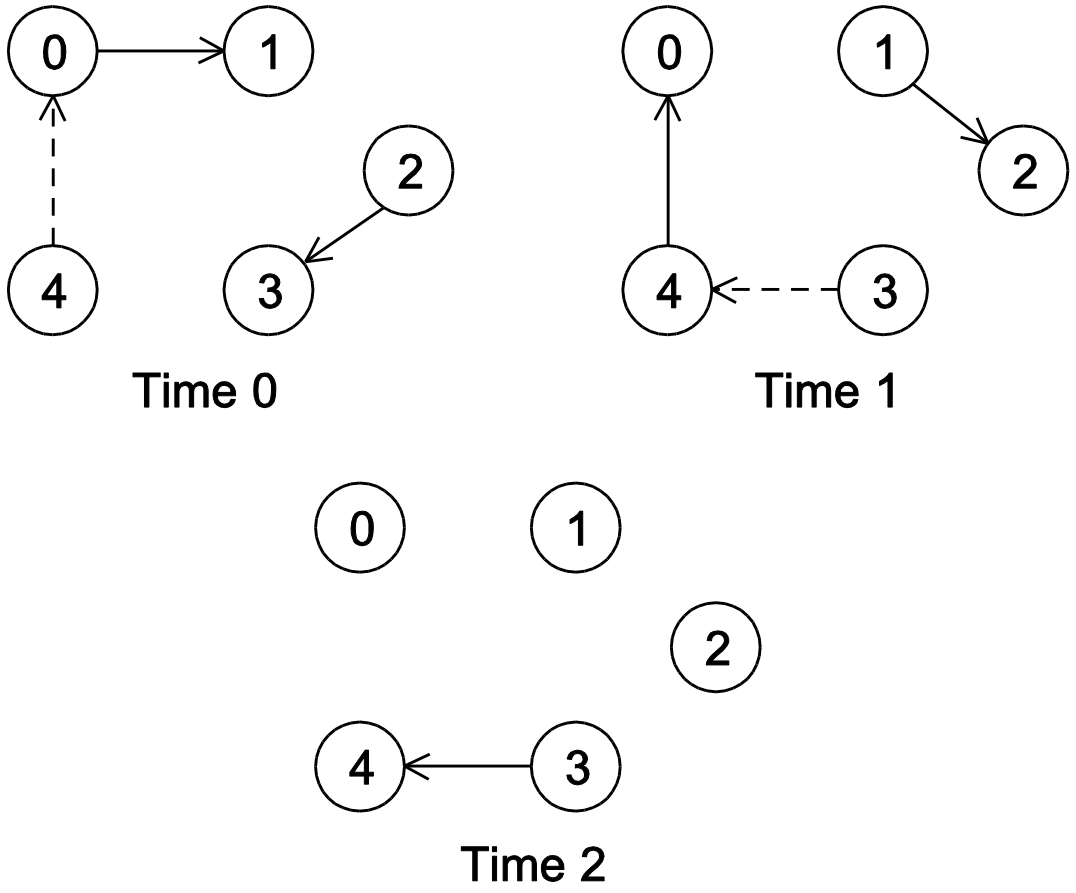
Time	Process			
	0	1	2	3
Start	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
After Local Sort	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13
After Phase 0	3, 7, 8, 9	11, 14, 15, 16	1, 2, 4, 5	6, 10, 12, 13
After Phase 1	3, 7, 8, 9	1, 2, 4, 5	11, 14, 15, 16	6, 10, 12, 13
After Phase 2	1, 2, 3, 4	5, 7, 8, 9	6, 10, 11, 12	13, 14, 15, 16
After Phase 3	1, 2, 3, 4	5, 6, 7, 8	9, 10, 11, 12	13, 14, 15, 16

```
int MPI_Ssend(
      void*          msg_buf_p      /* in */,
      int            msg_size       /* in */,
      MPI_Datatype   msg_type       /* in */,
      int            dest           /* in */,
      int            tag            /* in */,
      MPI_Comm       communicator   /* in */);
```

MPI_Ssend

```
int MPI_Sendrecv(
      void*          send_buf_p      /* in  */,
      int            send_buf_size   /* in  */,
      MPI_Datatype   send_buf_type   /* in  */,
      int            dest            /* in  */,
      int            send_tag        /* in  */,
      void*          recv_buf_p      /* out */,
      int            recv_buf_size   /* in  */,
      MPI_Datatype   recv_buf_type   /* in  */,
      int            source          /* in  */,
      int            recv_tag        /* in  */,
      MPI_Comm       communicator    /* in  */,
      MPI_Status*    status_p        /* in  */);
```

MPI_Sendrecv



Safe communication with five processes

Thank you

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

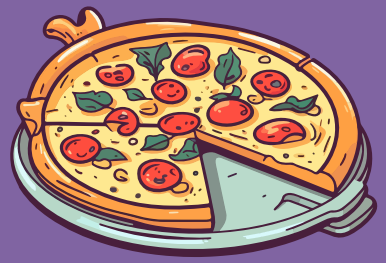
Shared-memory programming with OpenMP – openmp pragmas and directives, The trapezoidal rule, Scope of variables, The reduction clause, loop carried dependency, scheduling, producers and consumers, Caches, cache coherence and false sharing in openmp, tasking, tasking, thread safety.

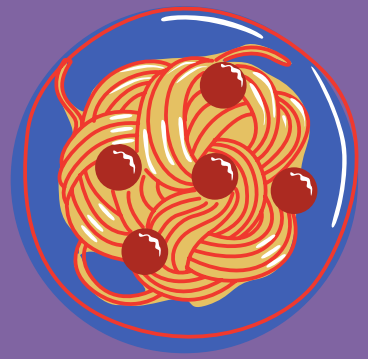
Shared Memory with MIMD

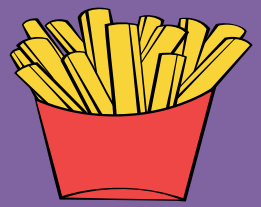
each cooking a different dish

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 `#pragma omp`	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.

```c
#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
```

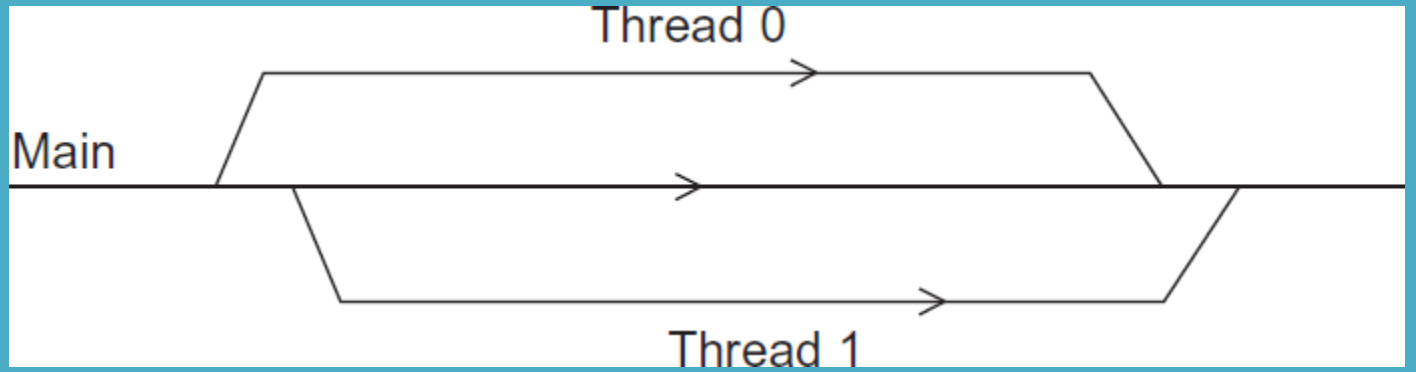
```
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
Hello from thread 1 of 4
```

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());
}
```

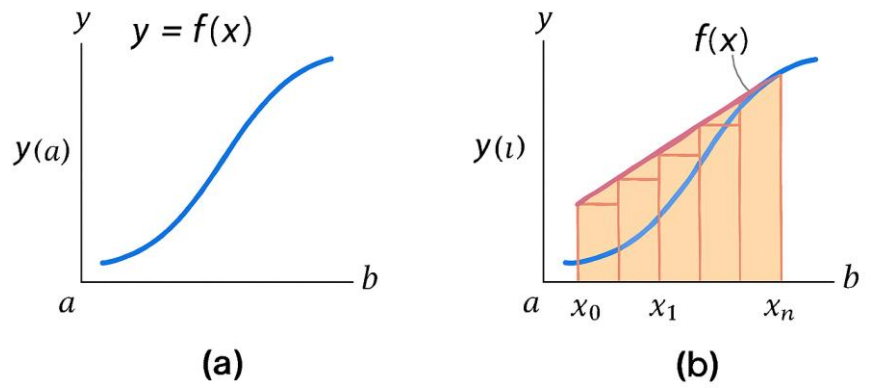


FIGURE 5.3: The trapezoidal rule.

```
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.

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

- 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

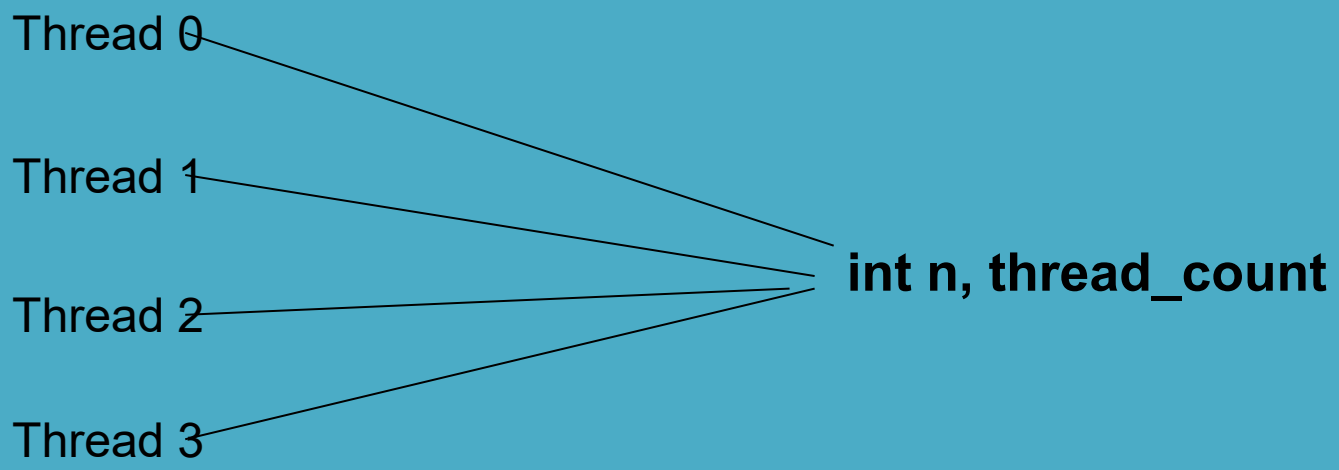
File-wide scope

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

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-5

GPU programming with CUDA - GPUs and GPGPU, GPU architectures, Heterogeneous computing, Threads, blocks, and grids Nvidia compute capabilities and device architectures, Vector addition, Returning results from CUDA kernels, CUDA trapezoidal rule I, CUDA trapezoidal rule II: improving performance, CUDA trapezoidal rule III: blocks with more than one warp.

GPUs and GPGPU

In the late 1990s and early 2000s, the computer industry responded to the demand for highly realistic computer video games and video animations by developing extremely powerful **graphics processing units** or **GPUs**.

These processors, as their name suggests, are designed to improve the performance of programs that need to render many detailed images.

One of the biggest difficulties faced by the early developers of GPGPU was that the GPUs of the time could only be programmed using computer graphics APIs, such as Direct3D and OpenGL.

GPU architectures

```
if ( x [ i ] >= 0 )
x [ i ] += 1 ;
else
x [ i ] -= 2 ;
```

In a typical SIMD system, each datapath carries out the test x[i] >= 0. Then the datapaths for which the test is true execute x[i] += 1, while those for which x[i] < 0 are *idle*. Then the roles of the datapaths are reversed: those for which x[i] >= 0 are idle while the other datapaths execute x[i] = 2.

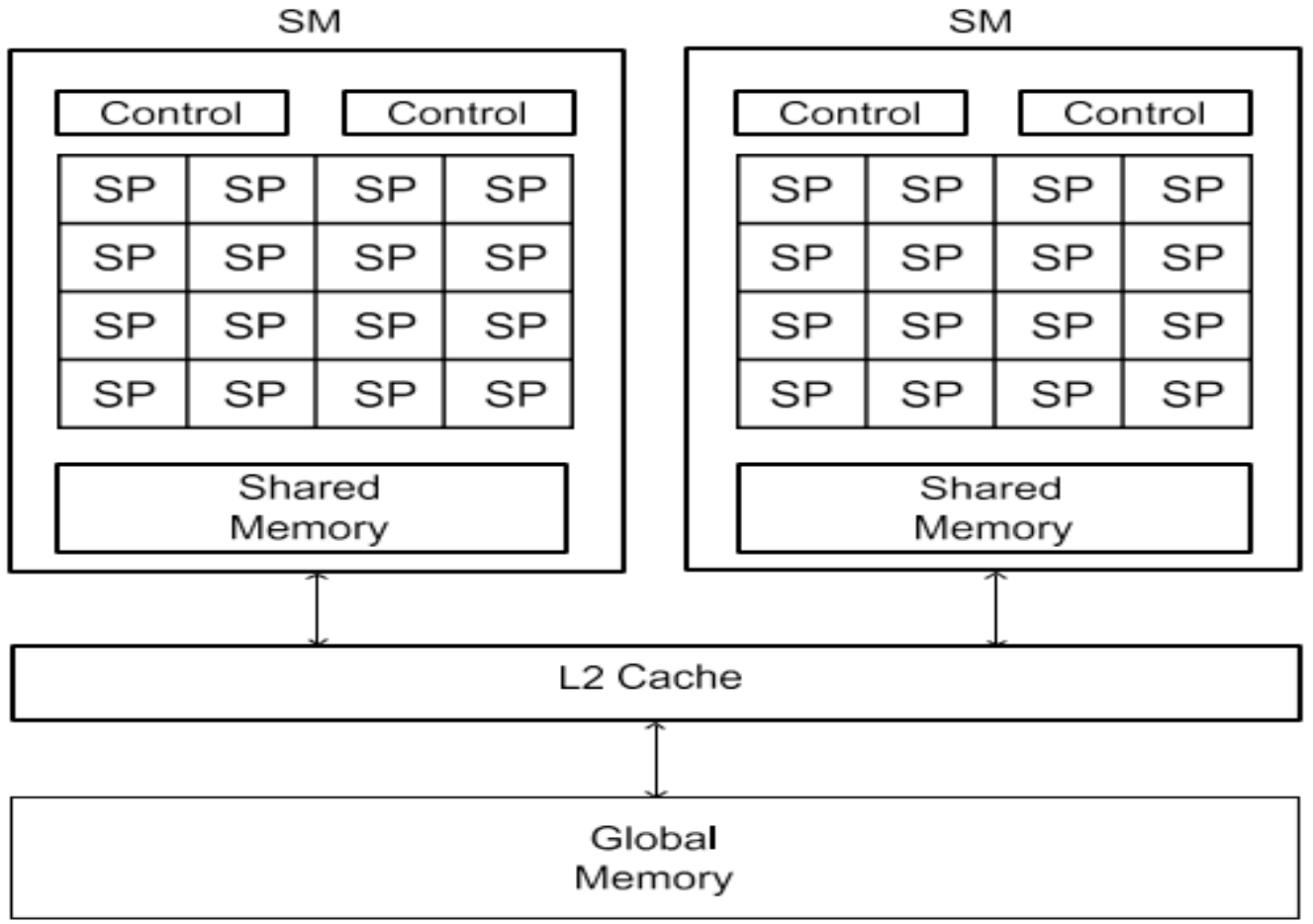


FIGURE 6.1

Simplified block diagram of a GPU.

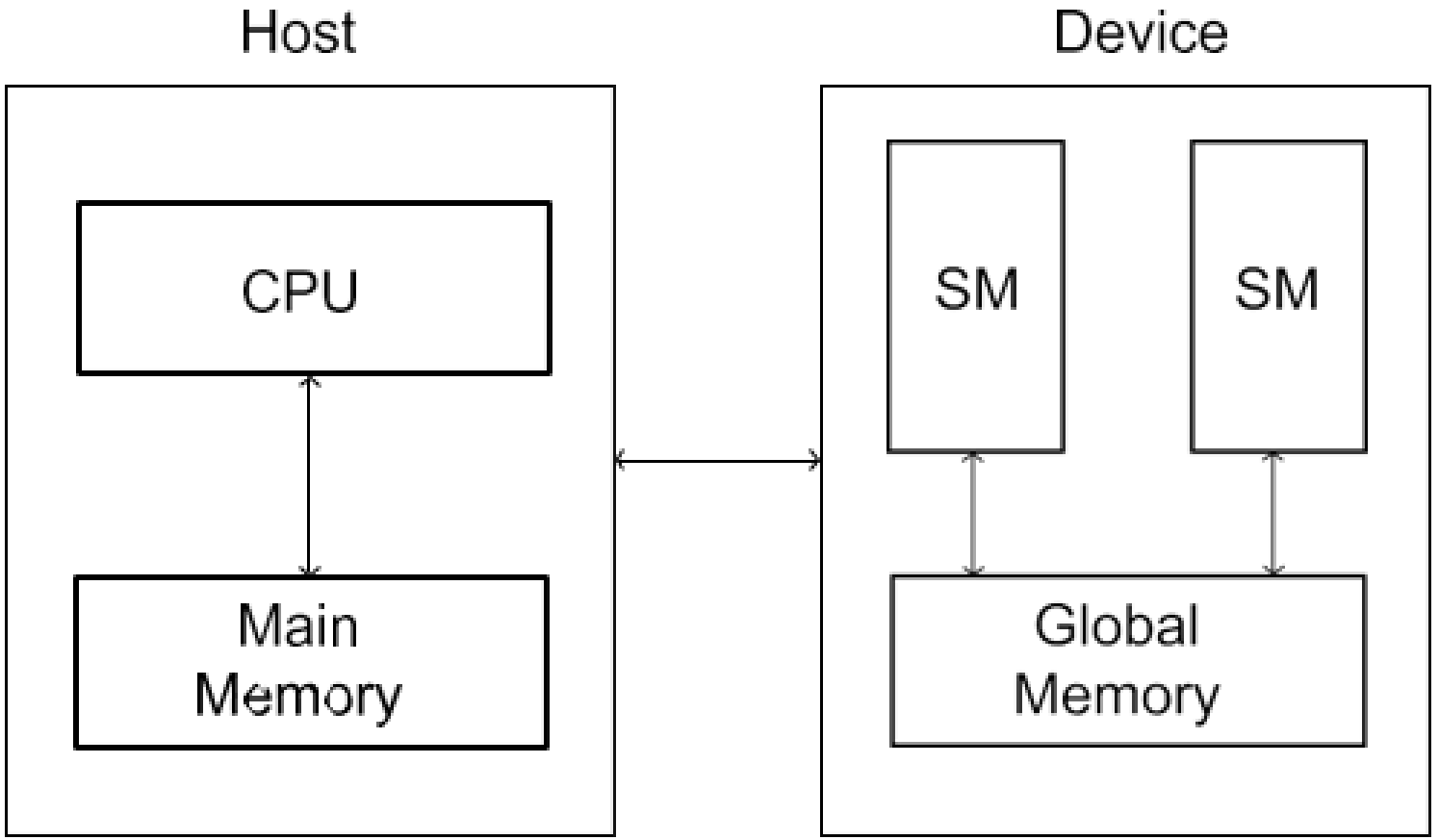


FIGURE 6.2

Simplified block diagram of a CPU and a GPU.

Heterogeneous computing

Writing a program that runs on a GPU is an example of **heterogeneous** computing.

Field Programmable Gate Arrays or FPGAs, and Digital Signal Processors or DSPs. FPGAs contain programmable logic blocks and interconnects that can be configured prior to program execution.

DSPs contain special circuitry for manipulating (e.g., compressing, filter- ing) signals, especially “real-world” analog signals.

CUDA hello

CUDA is a software platform that can be used to write GPGPU programs for heterogeneous systems equipped with an Nvidia GPU.

```
float *x = malloc ( n * sizeof ( float ));
```

However, in C++ a cast is required

```
float *x = ( float *) malloc ( n * sizeof ( float ));
```

```
1  #include <stdio.h>
2  #include <cuda.h>    /* Header file for CUDA */
3
4  /* Device code:  runs on GPU */
5  __global__ void Hello(void) {
6
7      printf("Hello from thread %d!\n", threadIdx.x);
8  }  /* Hello */
9
10
11 /* Host code:  Runs on CPU */
12 int main(int argc, char* argv[]) {
13     int thread_count;          /* Number of threads to run on GPU */
14
15     thread_count = strtol(argv[1], NULL, 10);
16                         /* Get thread_count from command line */
17
18     Hello <<<1, thread_count>>>();
19                         /* Start thread_count threads on GPU, */
20
21     cudaDeviceSynchronize();              /* Wait for GPU to finish */
22
23     return 0;
24 }  /* main */
```

Program 6.1: CUDA program that prints greetings from the threads.

Compiling and running the program

$ nvcc -o cuda_hello cuda_hello.cu

If we want to run one thread on the GPU, we can type
$ ./cuda_hello 1

and the output will be

```
Hello from thread 0!
```

If we want to run ten threads on the GPU, we can type

```
$ ./cuda_hello 10
```

and the output of will be

```
Hello from thread 0! Hello from thread 1! Hello from thread 2! Hello from thread 3! Hello from thread 4! Hello from thread 5! Hello from thread 6! Hello from thread 7! Hello from thread 8! Hello from thread 9!
```

Threads, blocks, and grids

Hello:

Hello <<<1, thread_count > > >();

Hello <<<2, thread_count / 2 >> >();

If thread_count is even, this kernel call will start a total of thread_count threads, and the threads will be divided between the two SMs: thread_count/2 threads will run on each SM. (What happens if thread_count is odd?)

```
1  #include <stdio.h>
2  #include <cuda.h>    /* Header file for CUDA */
3
4  /* Device code:  runs on GPU */
5  __global__ void Hello(void) {
6
7      printf("Hello from thread %d in block %d\n",
8              threadIdx.x, blockIdx.x);
9  }  /* Hello */
10
11
12 /* Host code:  Runs on CPU */
13 int main(int argc, char* argv[]) {
14     int blk_ct;                    /* Number of thread blocks */
15     int th_per_blk;          /* Number of threads in each block */
16
17     blk_ct = strtol(argv[1], NULL, 10);
18                     /* Get number of blocks from command line */
19     th_per_blk = strtol(argv[2], NULL, 10);
20          /* Get number of threads per block from command line */
21
22     Hello <<<blk_ct, th_per_blk>>>();
23             /* Start blk_ct*th_per_blk threads on GPU, */
24
25      cudaDeviceSynchronize();              /* Wait for GPU to finish */
26
27     return 0;
28 }  /* main */
```

Program 6.2: CUDA program that prints greetings from threads in multiple blocks.

Nvidia compute capabilities and device architectures

There are a number of versions of the CUDA API, and they do *not* correspond to the compute capabilities of the different GPUs.

Name	Ampere	Tesla	Fermi	Kepler	Maxwell	Pascal	Volta	Turing
Compute capability	8.0	1.b	2.b	3.b	5.b	6.b	7.0	7.5

GPU architectures and compute capabilities.

Vector addition

We'll initialize x and y on the host. Then a kernel can start at least *n* threads, and the *i*th thread will add

z [i] = x [i] + y [i];

float *x , *y , *z ;

After allocating and initializing the arrays,
we'll call the kernel, and after the kernel completes execution,
the program checks the result, frees memory, and quits.

```
1  void Serial_vec_add(
2          const float  x[]    /* in  */,
3          const float  y[]    /* in  */,
4          float        cz[]   /* out */,
5          const int    n      /* in  */) {
6
7      for (int i = 0; i < n; i++)
8          cz[i] = x[i] + y[i];
9  }  /* Serial_vec_add */
```

Program 6.4: Serial vector addition function.

```
1  void Get_args(
2          const int argc              /* in    */,
3          char*     argv[]            /* in    */,
4          int*      n_p               /* out   */,
5          int*      blk_ct_p          /* out   */,
6          int*      th_per_blk_p      /* out   */,
7          char*     i_g               /* out   */) {
8      if (argc != 5) {
9          /* Print an error message and exit */
10         ...
11     }
12
13     *n_p = strtol(argv[1], NULL, 10);
14     *blk_ct_p = strtol(argv[2], NULL, 10);
15     *th_per_blk_p = strtol(argv[3], NULL, 10);
16     *i_g = argv[4][0];
17
18     /* Is n > total thread count = blk_ct*th_per_blk? */
19     if (*n_p > (*blk_ct_p)*(*th_per_blk_p)) {
20         /* Print an error message and exit */
21         ...
22     }
23 }  /* Get_args */
```

Program 6.5: `Get_args` function from CUDA program that adds two vectors.

```
1  void  Allocate_vectors(
2          float** x_p       /* out */,
3          float** y_p       /* out */,
4          float** z_p       /* out */,
5          float** cz_p      /* out */,
6          int     n         /* in  */) {
7
8      /* x, y, and z are used on host and device */
9      cudaMallocManaged(x_p, n*sizeof(float));
10     cudaMallocManaged(y_p, n*sizeof(float));
11     cudaMallocManaged(z_p, n*sizeof(float));
12
13     /* cz is only used on host */
14     *cz_p = (float*) malloc(n*sizeof(float));
15 }   /* Allocate_vectors */
```

Program 6.6: Array allocation function of CUDA program that adds two vectors.

```
1  double Two_norm_diff(
2         const float  z[]     /* in */,
3         const float  cz[]    /* in */,
4         const int    n       /* in */) {
5     double diff,    sum = 0.0;
6
7     for (int i = 0; i < n; i++) {
8         diff = z[i] - cz[i];
9         sum += diff*diff;
10    }
11    return sqrt(sum);
12 } /* Two_norm_diff */
```

Program 6.7: C function that finds the distance between two vectors.

```
1  void  Free_vectors(
2          float *  x      /* in/out */,
3          float *  y      /* in/out */,
4          float *  z      /* in/out */,
5          float *  cz     /* in/out */) {
6
7      /* Allocated with cudaMallocManaged */
8      cudaFree(x);
9      cudaFree(y);
10     cudaFree(z);
11
12     /* Allocated with malloc */
13     free(cz);
14 }   /* Free_vectors */
```

Program 6.8: CUDA function that frees four arrays.

```
1  void  Allocate_vectors(
2          float **  hx_p     /* out */,
3          float **  hy_p     /* out */,
4          float **  hz_p     /* out */,
5          float **  cz_p     /* out */,
6          float **  dx_p     /* out */,
7          float **  dy_p     /* out */,
8          float **  dz_p     /* out */,
9          int       n        /* in  */)   {
10
11     /* dx, dy, and dz are used on device */
12     cudaMalloc(dx_p, n*sizeof(float));
13     cudaMalloc(dy_p, n*sizeof(float));
14     cudaMalloc(dz_p, n*sizeof(float));
15
16     /* hx, hy, hz, cz are used on host */
17     *hx_p = (float *) malloc(n*sizeof(float));
18     *hy_p = (float *) malloc(n*sizeof(float));
19     *hz_p = (float *) malloc(n*sizeof(float));
20     *cz_p = (float *) malloc(n*sizeof(float));
21 }   /* Allocate_vectors */
```

Program 6.10: Allocate_vectors function for CUDA vector addition program that doesn't use unified memory.

Returning results from CUDA kernels

```
__global__ void Add(int x, int y, int *sum_p) {
    *sum_p = x + y;
}  /* Add */

int main(void) {
    int sum = _5;
    Add <<<1, 1>>> (2, 3, &sum);
    cudaDeviceSynchronize();
    printf("The sum is %d\n", sum);

    return 0;
}
```

```
int main(void) {
    int *sum_p;
    cudaMallocManaged(&sum_p, sizeof(int));
    *sum_p = _5;
    Add <<<1, 1>>> (2, 3, sum_p);
    cudaDeviceSynchronize();
    printf("The sum is %d\n", *sum_p);
    cudaFree(sum_p);

    return 0;
}
```

```
__managed__ int sum;

__global__ void Add(int x, int y) {
    sum = x + y;
} /* Add */

int main(void) {
    sum = _5;
    Add <<<1, 1>>> (2, 3);
```

CUDA trapezoidal rule I

This gives us a total approximation of the area between the graph and the x-axis as

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

and we can rewrite this as

$$h\left[\frac{1}{2}(f(a)+f(b))+(f(x_1)+f(x_2)+\cdots+f(x_{n-1}))\right].$$

We can implement this with the serial function shown in Program 6.11.

```
1  float Serial_trap(
2        const float  a   /* in */,
3        const float  b   /* in */,
4        const int    n   /* in */) {
5     float x, h = (b-a)/n;
6     float trap = 0.5*(f(a) + f(b));
7
8     for (int i = 1; i <= n-1; i++) {
9        x = a + i*h;
10       trap += f(x);
11    }
12    trap = trap*h;
13
14    return trap;
15 }  /* Serial_trap */
```

Program 6.11: A serial function implementing the trapezoidal rule for a single CPU.

Initialization, return value, and final update

To deal with the initialization and the final update (Items 1 and 5), we could try to select a single thread—say, thread 0 in block 0—to carry out the operations:

```
int  my_i = blockDim . x blockIdx . x * + threadIdx . x ;
if  ( my_i == 0 ) {
h = ( b−a )/ n ;

trap = 0 . 5 ∗ ( f ( a ) + f ( b ));
}
. . .
if  ( my_i == 0 )
trap = trap ∗h ;
```

CUDA trapezoidal rule II: improving performance

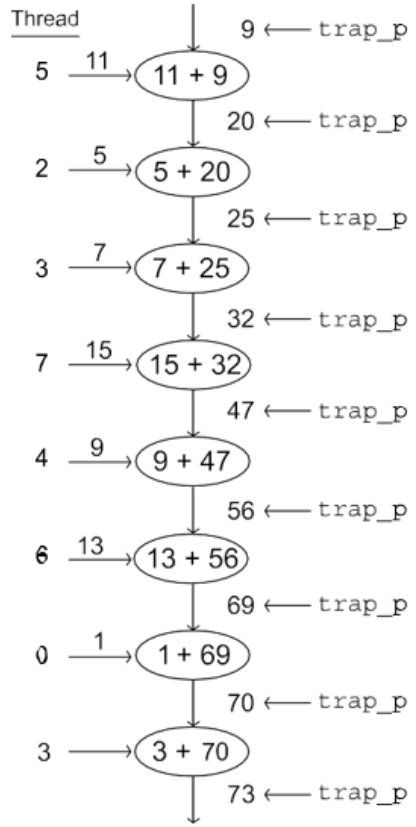


FIGURE 6.3
Basic sum.

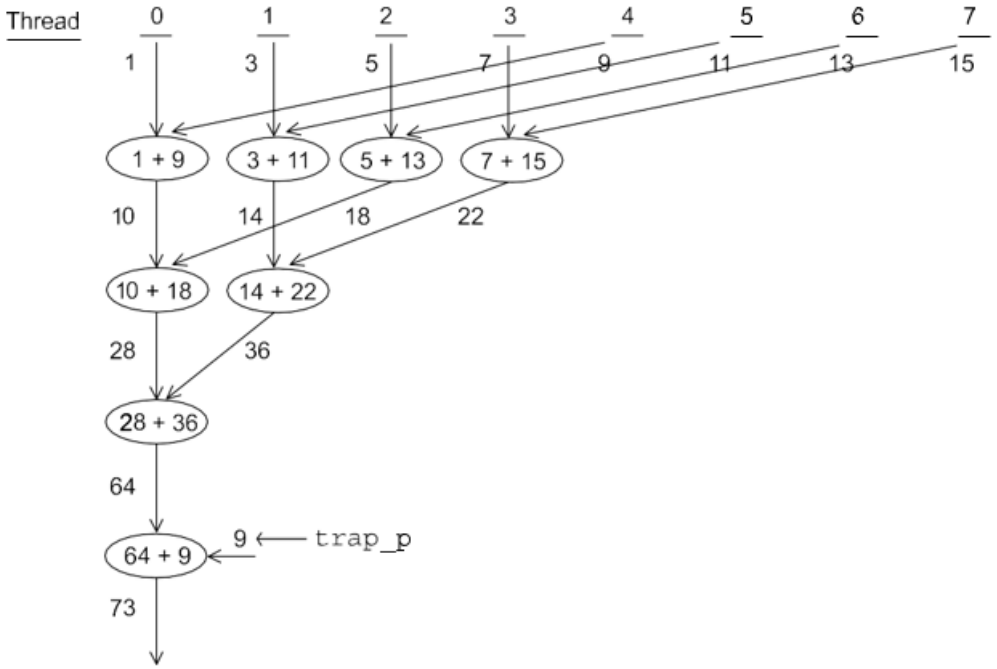


FIGURE 6.4
Tree-structured sum.

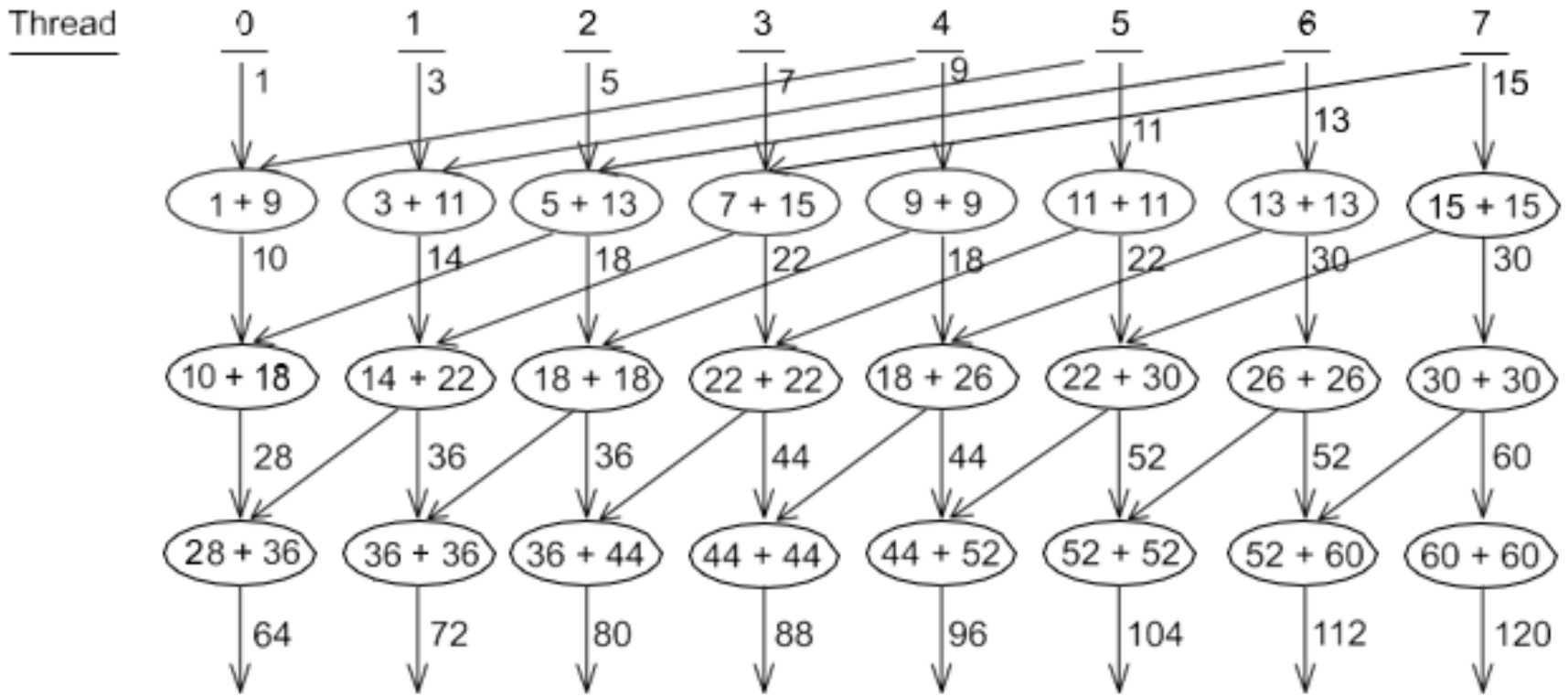


FIGURE 6.5

Tree-structured sum using warp shuffle.

```
__device__ float Shared_mem_sum(float shared_vals[]) {
int my_lane = threadIdx.x % warpSize;

for (int diff = warpSize/2; diff > 0; diff = diff/2) {
    /* Make sure 0 <= source < warpSize */
    int source = (my_lane + diff) % warpSize;
    shared_vals[my_lane] += shared_vals[source];
}
return shared_vals[my_lane];
}
```

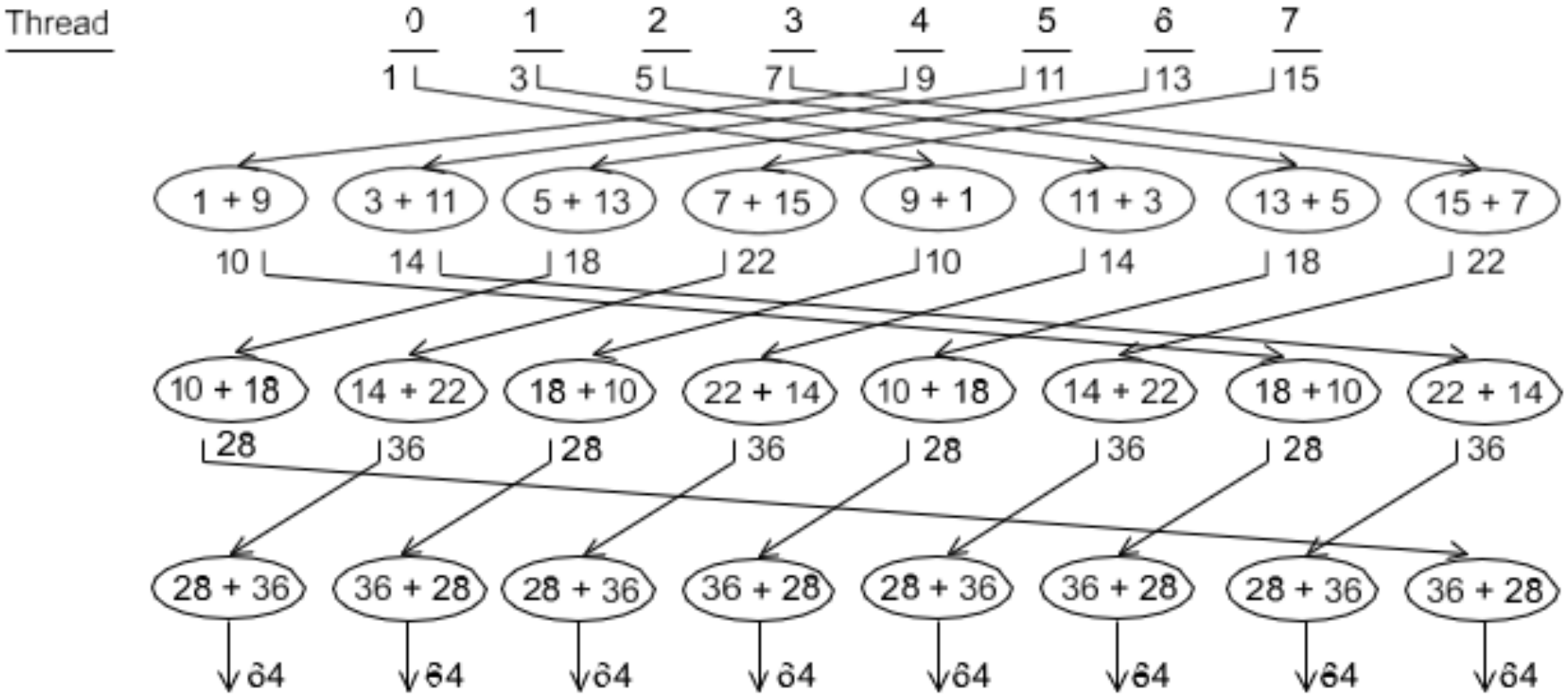


FIGURE 6.6

Dissemination sum using shared memory.

```
1  __global__ void Dev_trap(
2          const float  a          /*   in     */,
3          const float  b          /*   in     */,
4          const float  h          /*   in     */,
5          const int    n          /*   in     */,
6          float *      trap_p     /* in/out */) {
7      int my_i = blockDim.x * blockIdx.x + threadIdx.x;
8
9      float my_trap = 0.0f;
10     if (0 < my_i && my_i < n) {
11         float my_x = a + my_i*h;
12         my_trap = f(my_x);
13     }
14
15     float result = Warp_sum(my_trap);
16
17     /* result is correct only on thread 0 */
18     if (threadIdx.x == 0) atomicAdd(trap_p, result);
19 } /* Dev_trap */
```

Program 6.13: CUDA kernel implementing trapezoidal rule and using Warp_sum.

```
1  __global__ void Dev_trap(
2          const float  a           /* in  */,
3          const float  b           /* in  */,
4          const float  h           /* in  */,
5          const int    n           /* in  */,
6          float *      trap_p      /* out */) {
7      __shared__  float  shared_vals[WARPSZ];
8      int  my_i = blockDim.x * blockIdx.x + threadIdx.x;
9      int  my_lane = threadIdx.x % warpSize;
10
11     shared_vals[my_lane] = 0.0f;
12     if (0 < my_i && my_i < n) {
13         float my_x = a + my_i*h;
14         shared_vals[my_lane] = f(my_x);
15     }
16
17     float result = Shared_mem_sum(shared_vals);
18
19     /* result is the same on all threads in a block. */
20     if (threadIdx.x == 0) atomicAdd(trap_p, result);
21 }  /* Dev_trap */
```

Program 6.14: CUDA kernel implementing trapezoidal rule and using shared memory.

Performance

Table 6.8 Mean run-times for trapezoidal rule using block size of 32 threads (times in ms).

System	ARM Cortex-A15	Nvidia GK20A	Intel Core i7	Nvidia GeForce GTX Titan X
Clock	2.3 GHz	852 MHz	3.5 GHz	1.08 GHz
SMs, SPs		1, 192		24, 3072
Original	33.6	20.7	4.48	3.08
Warp Shuffle		14.4		0.210
Shared Memory		15.0		0.206

The remaining codes for the warp sum kernel and the shared memory sum kernel are very similar. First warp 0 computes the sum of the elements in warp_sum_arr. Then thread 0 in the block adds the block sum into the total across all the threads in the grid using atomicAdd. Here's the code for the shared memory sum:

```
if ( my_warp == 0 ) {
    if ( threadIdx . x >= blockDim . x / warpSize )
       warp_sum_arr [ threadIdx . x ] = 0.0;
    blk_result = Shared_mem_sum ( warp_sum_arr );
}
if ( threadIdx . x == 0 ) atomicAdd ( trap_p , blk_result );
```

Thank you