### Process

An instance of the computer that is being executed. These are its components:  
Executable machine language program. Block of memory. Descriptor of the OS resources allocated to it. Security info. Information about the state of the process.

### Threads

They allow programmers to divide their programs into independent tasks. A stream of instructions that can be scheduled to run independently of its main program. The hope is that when one thread blocks because it is waiting for resources, the other can run.

### Processes vs Threads

* Threads exist within a process; they’re like the children of the process.
* A process has at least one thread.
* If a process has more than one thread, it is multithreaded.
* Starting a thread within a process is known as **forking**.
* Terminating a thread is known as **joining**.
* Both threads and processes are units of execution or **tasks**.
* **Processes do not share memory** (each gets its own block of memory from the system).
* **Threads within a process share memory** (since they are children of the process, they have access to its resources).
* Data stored in a process’s memory can be **shared or private**:
  + If **private**, only the thread that owns it can use it.

### Shared Memory

* Allows processors to have access to a global address space.
* Multiple processes can operate independently but share the same memory resources.
* Changes in a memory location affected by one task are visible to others.

### Uniform Memory Access (UMA) vs Non-Uniform Memory Access (NUMA)

* The time to access all memory locations is the same for all cores.
* A memory location a core is directly connected to can be accessed faster than a memory location that must be accessed through another chip.

### Task Scheduling

* A **scheduler** is a program that uses a **scheduling policy** to decide which process should run next.
* It uses a selection function. The selection function considers:
  + Resources the process requires. Time the process has been waiting. The process’s priority.
* Scheduling policy should try to optimize:
  + **Responsiveness** of interactive processes, **Turnaround time**, **Resource utilization**, **Fairness**

### Non-Preemptive Scheduling Policies

* Each task runs to completion before the next one can run.
* **First In First Out (FIFO)**, **Shortest-Job-First (SJF)**.

### Preemptive Scheduling Policies

* **Round-Robin**: Each task is assigned a fixed time before it is required to give way to the next task and move back to the queue.
* **Earliest-Deadline-First**. **Shortest Remaining Time First**

### Key Terms

* **Shared resource**: A resource available to all processes in the concurrent program.
* **Critical section**: Sections of code within a process that require access to shared resources. Cannot be executed while another process is in a corresponding section of code.
* **Condition synchronization**: A mechanism ensuring that a process does not proceed until a certain condition is satisfied.
* **Starvation**: A situation where a runnable process is overlooked indefinitely by the scheduler.

### Dead or Alive(lock)

Concurrent programs must be safe, meaning the program cannot enter a bad state, and alive meaning they have to make progress Two problems that can occur with concurrency: - **Deadlock**: A process is waiting for a shared resource that will never be available (e.g., another process is waiting for this process to act). - **Livelock**: Multiple processes continuously change state in response to each other without making progress.

### Conditions for Deadlock

For deadlock to occur, **four conditions must hold**: 1. **Mutual Exclusion**: The program involves a shared resource protected by mutual exclusion. 2. **Hold While Waiting**: A process can hold a resource while waiting for others. 3. **No Preemption**: The OS cannot force a process to deallocate a resource it holds. 4. **Circular Wait**: P1 is waiting for a resource held by P2, and P2 is waiting for a resource held by P1.

### Preventing Deadlock

To prevent deadlock, **prevent at least one of the four conditions** from occurring.

### POSIX Threads

A **POSIX thread** is a thread associated with a process’s shared resources. Each thread has its own: **Stack**, **Program counter**, **Registers**, **Thread ID**

### Races

A **race condition** occurs when the **parent process exits before its child threads complete**. This does not allow enough time for child threads to finish execution. - Best fix for race conditions: use mutual exclusions and join the threads

### Task Parallelism vs. Data Parallelism

* Task Parallelism shares the tasks among each core ie on core does the tasks on all data
* Data parallelism shares the data among each core

// Task Parallelism  
#pragma omp parallel num\_threads(4)  
{   
 int id = omp\_get\_thread\_num();  
 printf("T%d:A\n", id);  
 printf("T%d:B\n", id);  
   
 if (id == 0)  
 printf("T0:special task\n");  
   
 if (id == 1)  
 printf("T1:special task\n");  
   
 if(id == 2)  
 printf("T2:special task\n");  
}  
  
// Data Parallelism  
#pragma omp parallel num\_threads(2)  
{  
 int id = get\_thread\_num()  
 int my\_a = id \* 3; // where you want the thread to start doing work  
 int my\_b = id \* 3 + 3; // where it should stop doing work  
   
 printf("T%d will process indexes %d to ");  
   
 for (int index = my\_a; index < my\_b; index++)   
 printf("do work\n");  
}  
return 0;

### Race Conditions

A **race condition** occurs when multiple threads **simultaneously access and modify shared data**, leading to **unpredictable behavior**.

#pragma omp parallel  
{  
 global\_sum += my\_sum; // Potential race condition  
}

To prevent this, we use **mutual exclusion** techniques.

### Barriers

**Barriers** ensure that all threads reach a synchronization point before continuing execution.

### Types of Barriers:

**Implicit Barriers** - Automatically added at the end of parallel regions. **Explicit Barriers** - Defined using #pragma omp barrier.

#pragma omp parallel  
{  
 compute\_part();  
 #pragma omp barrier // Ensures all threads finish before proceeding  
 finalize\_part();  
}

### Barrier Limitations:

* All threads must encounter the barrier.
* Conditional execution may lead to **illegal barriers**.

### nowait Clause

Using nowait allows threads **to skip synchronization** when it is unnecessary, improving performance.

#pragma omp single nowait  
{  
 expensive\_task();  
}  
// Other threads continue execution without waiting.

### OpenMP Mutual Exclusion Mechanisms:

1. **Critical Directive** - Ensures exclusive execution.
2. **Atomic Directive** - Ensures atomic updates to a shared variable.
3. **Locks** - Explicit locking mechanisms.

### Mutual exclusions

**Mutual exclusion** ensures that only **one thread at a time** accesses a critical section.

// critical: protects critical section, kinda slow, especially if overused  
// Allows \*\*simultaneous execution\*\* of \*\*different\*\* critical sections.  
#pragma omp critical  
{  
 shared\_var += local\_val;  
}  
// Named Critical Sections  
#pragma omp critical(name1)  
x = compute\_x();  
#pragma omp critical(name2)  
y = compute\_y();  
  
// atomic  
// only supports simple operations `x++`, `x--`, `x += expr`, `x = x + expr`  
#pragma omp atomic  
sum += value;  
  
// locking  
#include <omp.h>  
static omp\_lock\_t mylock;  
  
int main() {  
 omp\_init\_lock(&mylock);  
  
 #pragma omp parallel  
 {  
 omp\_set\_lock(&mylock);  
 critical\_section();  
 omp\_unset\_lock(&mylock);  
 }  
  
 omp\_destroy\_lock(&mylock);  
 return 0;  
}

**Key Lock Functions:** - omp\_init\_lock(&lock); - omp\_set\_lock(&lock); - omp\_unset\_lock(&lock); - omp\_destroy\_lock(&lock);

### When to Use Which?

**Atomic**, Single-variable updates (fastest). **Critical**, Protects complex code sections. **Locks**, Fine-grained control over execution.  
**Avoid Mixing** different mutual exclusion methods. **Fairness is NOT guaranteed** - Some threads may starve. **Avoid Nesting** critical sections (deadlocks possible).

### Shared Variables

* Exist in **one memory location**, accessible by all threads.

int x = 5;  
#pragma omp parallel  
{  
 // All threads access the same x  
}  
int y = 5;  
#pragma omp parallel private(y)  
{  
 // Each thread gets its own y (uninitialized)  
}  
int z = 5;  
#pragma omp parallel firstprivate(z)  
{  
 // Like `private`, but initialized with the original value.  
 // Each thread gets its own z, initialized to 5  
}

### Default Clause

Sets the default scope for all variables.

int x = 0, y = 0;  
#pragma omp parallel num\_threads(4) default(none) private(x) shared(y)  
{  
 x = omp\_get\_thread\_num();  
 #pragma omp atomic  
 y += x;  
}

### Reductions

**Reduction** operations allow threads to **aggregate results** safely without manual synchronization.

### Reduction Examples

// syntax  
#pragma omp parallel reduction(<operator> : <variable list>)  
int sum = 0;  
#pragma omp parallel reduction(+:sum)  
{  
 sum += omp\_get\_thread\_num();  
}  
  
// Multiple Variables  
int x = 10, y = 10;  
#pragma omp parallel reduction(+:x, y)  
{  
 x = omp\_get\_thread\_num();  
 y = 5;  
}

### Reduction Operations

+ Summation. \* Multiplication. & Bitwise AND. | Bitwise OR. ^ Bitwise XOR. && Logical AND. || Logical OR. max/min

### Parallel Summation with Reduction

Instead of using a **critical section**, reductions optimize aggregation.

double global\_sum = 0;  
#pragma omp parallel num\_threads(4) reduction(+:global\_sum)  
{  
 global\_sum += compute\_value(omp\_get\_thread\_num());  
}

### Area Under a Curve (Trapezoidal Rule)

Using **reduction** to integrate a function:

double global\_result = 0.0;  
#pragma omp parallel num\_threads(4) reduction(+:global\_result)  
{  
 global\_result += Local\_trap(a, b, n);  
}  
printf("Approximate area: %f\n", global\_result);

### **Work-Sharing Constructs**

* Used to distribute work among threads inside a parallel region.
* **Types:**
  + for – Divides loop iterations across threads.
  + single – Assigns work to a single thread.
  + sections – Splits tasks into sections executed by different threads.
* There is an **implied barrier** at the exit unless nowait is specified.

### **Parallel For**

* Loop iterations are divided across threads dynamically.
* The loop variable is **private** by default.
* The execution order is **non-deterministic**.

### **Syntax Options:**

**Inside an existing parallel region:** ```c #pragma omp for for(i = start; i < end; i += step) { // Loop body }

#pragma omp parallel for for(i = start; i < end; i += step) { // Loop body } ```

### **Data Dependency & Loop-Carried Dependencies**

* Parallel loops should avoid **loop-carried dependencies** (when one iteration depends on results from another).
* fibo[0] = fibo[1] = 1;  
  #pragma omp parallel for  
  for (i = 2; i < n; i++)  
   fibo[i] = fibo[i-1] + fibo[i-2];

### **Reduction in Parallel Loops**

* Reduction avoids data races when accumulating results.
* **Example: Summing values in an array**
* double sum = 0.0;  
  #pragma omp parallel for reduction(+:sum)  
  for (i = 0; i < n; i++)  
   sum += array[i];

### **Assigning Work to a Single Thread**

* Use #pragma omp single for operations that should only be done once.
* #pragma omp parallel  
  {  
   printf("Hi from T%d\n", omp\_get\_thread\_num());  
   #pragma omp single  
   printf("One Hi from T%d\n", omp\_get\_thread\_num());  
  }
* *Only one thread will execute the single block.*

### **Parallel Sections**

* #pragma omp sections allows different sections of code to be executed by different threads.
* #pragma omp parallel sections  
  {  
   #pragma omp section  
   {  
   printf("Section 1 executed by thread %d\n", omp\_get\_thread\_num());  
   }  
   #pragma omp section  
   {  
   printf("Section 2 executed by thread %d\n", omp\_get\_thread\_num());  
   }  
  }
* There is an **implicit barrier** at the end of the sections unless nowait is used.

### **Loop Scheduling**

* The schedule clause determines how loop iterations are assigned to threads.
* static, Equal chunks assigned at compile time. dynamic, Threads take chunks dynamically. guided, Starts with large chunks, then reduces. auto, Compiler decides the best method.
* #pragma omp parallel for schedule(dynamic,2)  
  for(int i = 0; i<8; i++)  
   printf("T%d: %d\n", omp\_get\_thread\_num(), i);

### **Ordered Iterations**

* Ensures that iterations follow a strict order when needed.
* #pragma omp for ordered schedule(dynamic)  
  for(int i=0; i<100; i++) {  
   f(a[i]); // Can run in parallel  
   #pragma omp ordered  
   g(a[i]); // Runs in order  
  }

### **Parallel Matrix Multiplication**

#pragma omp parallel for collapse(2)  
for (i = 0; i < N; i++)  
 for (j = 0; j < N; j++) {  
 C[i][j] = 0;  
 for (k = 0; k < N; k++)  
 C[i][j] += A[i][k] \* B[k][j];  
 }

### **Finding the Maximum Value**

int max\_parallel(int \*arr){  
 int i, m = arr[0];  
 #pragma omp parallel for reduction(max:m)  
 for (i = 0; i < N; i++)  
 if (m < arr[i])  
 m = arr[i];  
 return m;  
}

### **Producer-Consumer Model**

void produce() {  
 while (i < NUM\_ITEMS) {  
 #pragma omp critical(one)  
 if (!full) {  
 put(item);  
 i++;  
 }  
 }  
}  
  
void consume() {  
 while (j < NUM\_ITEMS) {  
 #pragma omp critical(two)  
 if (!empty) {  
 get();  
 j++;  
 }  
 }  
}

*Ensures only one thread modifies shared data at a time.*

### Performance Metrics

* **Response Time** The time taken to complete one task
* **Throughput** is the number of tasks completed per unit of time
* **CPU Time breakdown**
  + User time: Time the CPU spends running the user’s code, System time: Time CPU spent running the OS’s code, Wait time: Time spent waiting for I/O or other services ### Instruction-Level Metrics
* **IPS (Instructions Per Second):** Approximate speed of CPU execution.
* **CPI (Cycles Per Instruction):** CPU Time = (CPI × Instruction Count) / Clock Rate ### Overhead in Parallelism
* **Overhead includes:**
  + Thread creation/destruction, Synchronization, Communication, Waiting due to load imbalance or mutual exclusion ### Speedup and Efficiency formulas:
* Speedup = T\_serial / T\_parallel  
  Efficiency = E = S / p // p is the number
* As p increases then E decreases due to overhead. If the problem size increases then both Speedup (S) and Efficiency (E) increase. due to less overhead ### Amdahl’s Law S is the max speedup, r is the percentage/fraction of the program that is parallelizable, p is the number of cores. r/p is also the parallel speedup
* S = 1 / ((1 - r) + (r/p))  
  S = 1 / ((1 - r) + r) // as p approaches infinity

### Gustafson’s law

* This formula is for scalable/large problem sizes. If ‘r’ or the parallelizable portion is 100% then S = P **Strong scalability** if E remains constant as p increases (that means the problem size is fixed). **Weak scalability** if E remains constant as both p and problem size increase. ### CPU vs. GPU Architecture (Feature, CPU, GPU), (Control logic, Complex, Simple), (Threads, Few, Thousand), (Memory bandwidth, Lower, higher), (Latency, Optimized, Higher), (User Case, Serial Work, Parallel Work) GPU uses SIMD (single instruction multiple data). That’s why GPU’s are optimized for parallelism. Host is the CPU, Device is the GPU, Kernel is the function run on the device (executed in parallel by many threads). A grid is a collection of blocks, a block is a collection of threads **Latency** is the time taken to complete one task. **Throughput** is the number of tasks completed per unit of time. ### Function Qualifiers (Qualifier, Runs on, Callable from), (**global**, Device, Host), (**device**, Device, Device), (**host**, Host, Host), (**host** **device**, Both, Both) ### cudaDeviceSynchronize()
* CUDA and OpenMP are async. Use cudaDeviceSynchronize(); to wait until all launched kernels finish.
* Threads are organized into 1D/2D/3D blocks.
* Blocks are organized into 1D/2D/3D grids.
* Each thread/block has its own ID:

To compute total threads and blocks:

* // computing thread idx  
  int i = blockIdx.x \* blockDim.x + threadIdx.x;  
  // 2d grid  
  int x = blockIdx.x \* blockDim.x + threadIdx.x;  
  int y = blockIdx.y \* blockDim.y + threadIdx.y;  
    
  int nthreads = 256;  
  int nblocks = (N + nthreads - 1) / nthreads;  
  vectorAdd<<<nblocks, nthreads>>>(...);  
  // block fimension  
  dim3 blockDim(16, 16);  
  //blocks in grid  
  dim3 gridDim((width+15)/16, (height+15)/16);  
  //thread indexing  
  int y = blockIdx.y \* blockDim.y + threadIdx.y;  
  int x = blockIdx.x \* blockDim.x + threadIdx.x;  
  // Always check bounds!  
  if (x < width && y < height) {  
   // safe access  
  }

### Matrix Multiplication: One Block

* Threads in a **single block** compute a matrix P = M × N
* Each thread computes one element in result P.

\_\_global\_\_ void MatrixMulKernel(float\* d\_M, float\* d\_N, float\* d\_P, int width) {  
 int r = threadIdx.y;  
 int c = threadIdx.x;  
 if (r < width && c < width) {  
 float value = 0;  
 for (int k = 0; k < width; k++)  
 value += d\_M[r\*width + k] \* d\_N[k\*width + c];  
 d\_P[r\*width + c] = value;  
 }  
}

* Threads perform computation on shared data. ### Combined CUDA Memory Summary

| Type | Memory | Scope | Lifetime | Speed | Notes |
| --- | --- | --- | --- | --- | --- |
| int x | Register | Thread | Thread | Very fast | Private |
| int arr[10] | Local | Thread | Thread | Slow (cached) | Used when registers spill |
| \_\_shared\_\_ | Shared | Block | Block | Fast | Shared across threads in a block |
| \_\_device\_\_ | Global | All grids | Application | Slow | Accessible by all threads |
| \_\_constant\_\_ | Constant | All grids | Application | Fast (cached) | Read-only, limited size (64 KB) |

### Memory Optimization Guidelines

1. **Minimize Host-Device Transfers:** Batch small transfers, Keep intermediate structures on the device
2. **Use Fast Memory Types:** (Register, Thread), (Shared, Fast), (Constant, Grid), (Global, Slow), (Local, Slow)
3. **Reduce Global Memory Traffic:** Use **tiling**: load data into shared memory, compute, write back, Coalesced access patterns ### Tiling and Shared Memory Example

### Synchronization Example Fix

array[i] = array[i-1]; // data race!  
//fix:   
\_\_syncthreads();  
int temp = array[i-1];  
\_\_syncthreads();  
array[i] = temp;  
//atomic sections  
\_\_global\_\_ void increment\_atomic(int\* x) {  
 atomicAdd(x, 1);  
}  
//critical sections  
\_\_device\_\_ void lock() {  
 while (atomicCAS(mutex, 0, 1) != 0);  
 \_\_threadfence();  
}  
\_\_device\_\_ void unlock() {  
 atomicExch(mutex, 0);  
 \_\_threadfence();  
}

### Coalesced Global Memory Access

**Access pattern matters!** - Best: contiguous access pattern (stride 1) - Avoid: Strided access, Random access, Misaligned blocks ### Access Pattern Examples

x = A[i]; x = A[2 \* i]; // Coalesced, Strided  
x = A[128 - i]; A[A[i]] = 7; // Strided, Random

### L1 and L2 Caches

* L1: Per SM, fast but **not coherent**
* L2: Shared across GPU, coherent
* Reads use both caches; writes go through L2 only ### Constant Memory Limited (64 KB), cached, read-only from device, Great for parameters or lookup tables ### Local Memory Private to thread, Physically located in global memory (but cached), Used when registers are insufficient

#include <stdio.h>  
#include <stdlib.h>  
#include "cuda\_runtime.h"  
#include "device\_launch\_parameters.h"  
#define BLOCK\_SIZE 1024  
\_\_global\_\_ void gpu\_sqr(int \*d\_in, int \*d\_out, int size) {  
 int idx = blockIdx.x \* blockDim.x + threadIdx.x;  
 if (idx < size) {  
 d\_out[idx] = d\_in[idx] \* d\_in[idx];  
 }  
}  
void cpu\_sqr(int \*data\_in, int \*data\_out, int size) {  
 for (int i = 0; i < size; ++i) {  
 data\_out[i] = data\_in[i] \* data\_in[i];  
 }  
}  
#define CHECK\_CUDA(call) \  
 { \  
 cudaError\_t err = call; \  
 if (err != cudaSuccess) { \  
 fprintf(stderr, "CUDA error at %s:%d: %s\n", \_\_FILE\_\_, \_\_LINE\_\_, \  
 cudaGetErrorString(err)); \  
 exit(EXIT\_FAILURE); \  
 } \  
 }  
int main() {  
 const int N = 10000;  
 int \*h\_in, \*h\_out; int \*d\_in, \*d\_out;  
 dim3 grid;  
 h\_in = (int\*)malloc(N \* sizeof(int));  
 h\_out = (int\*)malloc(N \* sizeof(int));  
 for (int i = 0; i < N; ++i) {  
 h\_in[i] = i;  
 }  
 CHECK\_CUDA(cudaMalloc((void\*\*)&d\_in, N \* sizeof(int)));  
 CHECK\_CUDA(cudaMalloc((void\*\*)&d\_out, N \* sizeof(int)));  
 CHECK\_CUDA(cudaMemcpy(d\_in, h\_in, N \* sizeof(int), cudaMemcpyHostToDevice));  
 grid.x = (N + BLOCK\_SIZE - 1) / BLOCK\_SIZE;  
 gpu\_sqr<<<grid, BLOCK\_SIZE>>>(d\_in, d\_out, N);  
 CHECK\_CUDA(cudaGetLastError());  
 CHECK\_CUDA(cudaMemcpy(h\_out, d\_out, N \* sizeof(int), cudaMemcpyDeviceToHost));  
 for (int i = 0; i < 10; ++i) {  
 printf("h\_out[%d] = %d\n", i, h\_out[i]);  
 }  
 free(h\_in);  
 free(h\_out);  
 CHECK\_CUDA(cudaFree(d\_in));  
 CHECK\_CUDA(cudaFree(d\_out));  
 return 0;  
}

### Parallel Programming Overview

* **Shared Memory Systems**:
  + Multiple cores on a chip share memory (e.g., CPUs with OpenMP).
  + Multiple chips (e.g., GPUs using CUDA).
* **Distributed Memory Systems**:
  + No shared memory; connected over a network.
  + Communicate via **messages** (MPI).

### Distributed Memory Programming Models

| Model | Description |
| --- | --- |
| Hadoop | Uses HDFS and MapReduce. High fault tolerance, low performance. |
| Spark | Better performance than Hadoop. Similar concept. |
| MPI | High-performance. Scales well. Suitable for most parallel code. |

### MPI Concepts

* **MPI** = Message Passing Interface.
* **SPMD** = Single-Program Multiple-Data.
* All nodes run the same program, but behavior depends on rank (e.g., if (my\_rank == 0)).

### MPI Program Structure (Boilerplate)

#include <mpi.h>  
  
int main(int argc, char \*argv[]) {  
 int my\_rank, comm\_sz;  
 MPI\_Init(&argc, &argv);  
 MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank);  
 MPI\_Comm\_size(MPI\_COMM\_WORLD, &comm\_sz);  
 // work here  
 MPI\_Finalize();  
 return 0;  
}

### Basic MPI Functions

| Function | Purpose |
| --- | --- |
| MPI\_Init | Initialize MPI |
| MPI\_Finalize | Finalize MPI |
| MPI\_Comm\_size | Get number of processes |
| MPI\_Comm\_rank | Get current process’s rank |
| MPI\_Send | Send message (blocking) |
| MPI\_Recv | Receive message (blocking) |
| MPI\_Isend/Irecv | Non-blocking versions |

### MPI Data Types

* Examples: MPI\_INT, MPI\_FLOAT, MPI\_DOUBLE, MPI\_CHAR, etc.

### Point-to-Point Communication

MPI\_Send(msg, size, type, dest, tag, comm);  
MPI\_Recv(msg, size, type, src, tag, comm, &status);

* Wildcards: MPI\_ANY\_SOURCE, MPI\_ANY\_TAG
* Use MPI\_STATUS\_IGNORE if status not needed.

### Wildcard Receives

MPI\_Recv(msg, size, type, MPI\_ANY\_SOURCE, MPI\_ANY\_TAG, comm, &status);

### Status Object

MPI\_Status status;  
status.MPI\_SOURCE; // source ID  
status.MPI\_TAG; // message tag  
status.MPI\_ERROR; // error code  
MPI\_Get\_count(&status, type, &count);

### Serial vs Parallel Trapezoidal Rule

#### Serial

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

#### Parallel (Manual Reduction)

Each process: - Computes its local interval and my\_sum. - Sends to rank 0. - Rank 0 gathers and sums.

### Communication Types

| Type | Description |
| --- | --- |
| Point-to-point | 1 sender, 1 receiver (MPI\_Send, MPI\_Recv) |
| Collective | Involves all processes in communicator |

### Collective Operations Summary

| Operation | Description |
| --- | --- |
| MPI\_Bcast | One to all (broadcast) |
| MPI\_Scatter | Split array among processes |
| MPI\_Gather | Combine chunks into array |
| MPI\_Allgather | Like gather, but all receive |
| MPI\_Reduce | All to one reduction |
| MPI\_Allreduce | All to all reduction |

### MPI\_Reduce

MPI\_Reduce(&input, &output, count, datatype, op, dest, comm);

* Example: MPI\_SUM, MPI\_MAX, etc.
* Output only valid on dest process.

### MPI\_Allreduce

MPI\_Allreduce(&input, &output, count, datatype, op, comm);

* All processes receive the result.

### MPI\_Bcast

MPI\_Bcast(&data, count, datatype, source, comm);

* All processes must participate.

### MPI\_Scatter

MPI\_Scatter(sendbuf, sendcount, sendtype,  
 recvbuf, recvcount, recvtype,  
 root, comm);

* Sends chunks to each process from root.

### MPI\_Gather

MPI\_Gather(sendbuf, sendcount, sendtype,  
 recvbuf, recvcount, recvtype,  
 root, comm);

* Opposite of scatter.

### MPI\_Allgather

MPI\_Allgather(sendbuf, sendcount, sendtype,  
 recvbuf, recvcount, recvtype,  
 comm);

* All processes get full combined result.

### Collective vs Point-to-Point

1. All processes must call the collective function.
2. Arguments must be compatible.
3. Output buffers must exist even on unused processes.
4. No tags in collectives → order matters!

### Efficiency Notes

* MPI implementations optimize tree structures and internal reductions.
* You should **prefer MPI\_Reduce/MPI\_Allreduce** over manual gathering when possible.

### Tree-Structured Reduction (Manual)

* Step-wise reductions from leaves to root.
* More balanced but requires more code.

### Comparison: Manual vs MPI\_Reduce

| Approach | Pros | Cons |
| --- | --- | --- |
| Manual Reduction | Full control | Complex, error-prone |
| MPI\_Reduce | Simple, optimized | Less flexible |

### Broadcast Example

if (rank == 0)  
 scanf(...); // root reads  
MPI\_Bcast(...); // everyone receives

### Scatter + Gather Use Case

1. Scatter chunks of data (e.g., subarrays).
2. Each process works independently.
3. Gather results back to one process.

### Final Notes

* MPI communication is **explicit** and **requires coordination**.
* Always ensure every process makes matching collective calls.
* Know when to use:
  + MPI\_Send / MPI\_Recv
  + MPI\_Bcast / MPI\_Scatter
  + MPI\_Reduce / MPI\_Allreduce