Nice set of questions — I’ll answer each one clearly and with working code examples where appropriate so you can use / submit them directly. I’ll label answers Q1…Q10 to match the paper.

**Q1. What are the Directives in OpenMP and summarize their roles. (3M)**

**OpenMP directives** (pragma in C/C++) are hints / commands to the compiler and runtime to express parallelism, data scoping, work-sharing, synchronization, and tasks. They are the primary interface between your sequential source and OpenMP parallel execution.

Important directives and short roles:

* #pragma omp parallel  
  Create a parallel region: threads are spawned and execute the structured block in parallel.
* #pragma omp for (or combined #pragma omp parallel for)  
  Work-sharing: distribute loop iterations among threads.
* #pragma omp sections + #pragma omp section  
  Work-sharing: divide independent blocks of code (sections) among threads.
* #pragma omp single  
  Make one thread execute the block; others skip (can include nowait).
* #pragma omp master  
  Only the master thread executes the block; cheaper than single because no implicit barrier.
* #pragma omp critical  
  Mutual exclusion: only one thread can execute the critical region at a time.
* #pragma omp atomic  
  Atomic update of a single memory location (faster / lighter than critical when applicable).
* #pragma omp barrier  
  Synchronize: all threads in the team wait until everyone reaches the barrier.
* #pragma omp task / #pragma omp taskwait  
  Task-based parallelism: create tasks that may execute later; taskwait waits for child tasks.
* #pragma omp reduction(...) (clause)  
  Specify reduction variables for +, \*, max, custom ops etc. Combined with parallel/for.
* #pragma omp simd  
  Request vectorization (SIMD) of the loop.
* #pragma omp ordered  
  For ordered loops, ensure sections of the loop execute in iteration order.
* #pragma omp threadprivate(...)  
  Make global variables private per thread (like TLS).
* #pragma omp flush  
  Enforce memory consistency / visibility between threads.

Each directive usually accepts *clauses* (e.g., private, shared, reduction, schedule, collapse, num\_threads) which control scoping, scheduling, thread counts, etc.

**Q2. Interpret and explain the role of for and sections directives in OpenMP. Give usage structure. (3M)**

**for (work-sharing for loops)**

Role: split iterations of a canonical loop among threads so iterations run in parallel.

Usage forms (C/C++):

#pragma omp parallel for [clauses]

for (int i = 0; i < N; ++i) {

// loop body; iterations can run concurrently

}

Or split parallel and for:

#pragma omp parallel

{

#pragma omp for

for (...) { ... }

}

Common clauses:

* private(varlist) — loop-private variables.
* shared(varlist) — shared variables.
* reduction(op:varlist) — perform reduction (sum/product/max).
* schedule(static|dynamic|guided, chunk) — controls iteration assignment.
* collapse(k) — collapse nested loops into one larger iteration space (useful when inner loops are small).
* ordered — use with #pragma omp ordered inside loop to enforce serial order in parts.

Example (sum with reduction):

double sum = 0.0;

#pragma omp parallel for reduction(+:sum)

for (int i=0;i<N;i++) sum += a[i];

**sections (work-sharing for independent blocks)**

Role: divide independent code blocks (not loop iterations) among threads. Good when you have several independent tasks with different workloads and you want different threads to execute different blocks.

Usage:

#pragma omp parallel sections [clauses]

{

#pragma omp section

{ /\* code block A \*/ }

#pragma omp section

{ /\* code block B \*/ }

#pragma omp section

{ /\* code block C \*/ }

}

Each section block is assigned to one thread; number of sections can be ≠ number of threads. There is an implicit barrier at the end of parallel sections unless nowait used.

Example:

#pragma omp parallel sections

{

#pragma omp section

{ process\_data(); }

#pragma omp section

{ write\_log(); }

}

**Key differences**:

* for partitions iterations of a loop evenly (or according to schedule). Useful for regular, repeated tasks.
* sections assigns whole independent code blocks; useful for a small number of different tasks.

**Q3. MPI routine that combines data from all processes using a specified operation, store result in root and distribute to all processes. (2M)**

Two standard MPI patterns:

1. **MPI\_Reduce** — combine values from all processes using an operation and **store the result on a designated root** (only the root gets the final result).

int MPI\_Reduce(const void \*sendbuf, void \*recvbuf, int count,

MPI\_Datatype datatype, MPI\_Op op, int root, MPI\_Comm comm);

* sendbuf: pointer to local data
* recvbuf: pointer to buffer on root (ignored on others)
* count: number of elements
* datatype: MPI datatype (MPI\_INT, MPI\_DOUBLE…)
* op: predefined or user op (MPI\_SUM, MPI\_MAX, ...)
* root: rank that receives the result
* comm: communicator (e.g. MPI\_COMM\_WORLD)

Usage: compute global sum at root:

int local = ...;

int result;

MPI\_Reduce(&local, &result, 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (rank == 0) printf("sum = %d\n", result);

1. **MPI\_Allreduce** — same combine operation but **the result is returned on all processes** (convenient if all ranks need the reduced value):

int MPI\_Allreduce(const void \*sendbuf, void \*recvbuf, int count,

MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm);

If you specifically need “store in root and then distribute,” you can:

* do MPI\_Reduce(..., root) and then MPI\_Bcast(recvbuf on root) to share the result to all processes. Or simply use MPI\_Allreduce which does both in one call (internally optimized).

**Q4. MPI program of 2 processes: P0 sends a number and a text message to P1. P1 receives them, sleeps 1s, then sends text back. Measure time taken by receiver to send the text back. Implement using standard send/receive. (3M)**

Below is a compact, standard MPI C program that does exactly that. Receiver measures the time taken to perform the send back (uses MPI\_Wtime()).

/\* mpi\_roundtrip.c \*/

#include <mpi.h>

#include <stdio.h>

#include <string.h>

#include <unistd.h> // sleep

#include <stdlib.h>

int main(int argc, char \*\*argv) {

MPI\_Init(&argc, &argv);

int rank, size;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

if (size != 2) {

if (rank==0) fprintf(stderr, "This program requires exactly 2 processes.\n");

MPI\_Finalize();

return 1;

}

const int TAG\_NUM = 0, TAG\_TEXT = 1, TAG\_REPLY = 2;

if (rank == 0) {

int number = 42;

char text[] = "Hello from P0";

printf("[P0] Sending number=%d and text=\"%s\" to P1\n", number, text);

MPI\_Send(&number, 1, MPI\_INT, 1, TAG\_NUM, MPI\_COMM\_WORLD);

MPI\_Send(text, (int)strlen(text)+1, MPI\_CHAR, 1, TAG\_TEXT, MPI\_COMM\_WORLD);

// wait for reply

char recvbuf[256];

MPI\_Recv(recvbuf, sizeof(recvbuf), MPI\_CHAR, 1, TAG\_REPLY, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

printf("[P0] Received reply from P1: \"%s\"\n", recvbuf);

} else if (rank == 1) {

int recv\_number;

char recvtext[256];

MPI\_Recv(&recv\_number, 1, MPI\_INT, 0, TAG\_NUM, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

MPI\_Recv(recvtext, sizeof(recvtext), MPI\_CHAR, 0, TAG\_TEXT, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

printf("[P1] Received number=%d and text=\"%s\" from P0\n", recv\_number, recvtext);

// simulate processing delay

sleep(1);

// measure time taken to send the text back

double t0 = MPI\_Wtime();

MPI\_Send(recvtext, (int)strlen(recvtext)+1, MPI\_CHAR, 0, TAG\_REPLY, MPI\_COMM\_WORLD);

double t1 = MPI\_Wtime();

printf("[P1] Time taken to MPI\_Send the reply: %.9f seconds\n", t1 - t0);

}

MPI\_Finalize();

return 0;

}

**Notes:**

* MPI\_Send is blocking semantics (it returns when the send buffer can be reused) — timing MPI\_Send on the sender side gives the time that the MPI call took locally. For full round trip timing, measure at P0 before and after round-trip.
* We used separate tags for clarity.

**Q5. Efficient MPI program: root sends a word to its slaves. All processes (including root) reverse this word. Display time taken by each slave; each process prints the reversed word. No user-defined functions. (4M)**

Approach: Use MPI\_Bcast to broadcast the word from root to all processes. Each process times how long it takes to reverse the word (using MPI\_Wtime()), prints its ID, the reversed word, and the elapsed time. All code in main.

Example C code:

/\* mpi\_reverse\_word.c \*/

#include <mpi.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

int main(int argc, char \*\*argv) {

MPI\_Init(&argc, &argv);

int rank, size;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

const int MAXLEN = 256;

char word[MAXLEN];

if (rank == 0) {

printf("Root: enter a word (no spaces): ");

fflush(stdout);

if (scanf("%255s", word) != 1) {

strcpy(word, "default");

}

}

// broadcast the word to all processes

MPI\_Bcast(word, MAXLEN, MPI\_CHAR, 0, MPI\_COMM\_WORLD);

// synchronize all processes before timing

MPI\_Barrier(MPI\_COMM\_WORLD);

// measure time to reverse locally

double t0 = MPI\_Wtime();

int len = (int)strlen(word);

char rev[MAXLEN];

for (int i = 0; i < len; ++i) rev[i] = word[len - 1 - i];

rev[len] = '\0';

double t1 = MPI\_Wtime();

double elapsed = t1 - t0;

// print results: let every process display reversed word and its time

printf("[rank %d] received \"%s\" -> reversed \"%s\"; time to reverse = %.9f s\n",

rank, word, rev, elapsed);

MPI\_Finalize();

return 0;

}

**Comments / Efficiency:**

* MPI\_Bcast is an efficient collective to distribute the root’s data to all processes.
* Reversal time is trivially small — MPI\_Wtime() still measures it; for very short strings timing noise dominates.
* If the root must measure *only* slaves, simply print only for ranks > 0 — code shows both so professor sees root also reversed.

**Q6. Interpret the use of threads, blocks, and grids in CUDA. (3M)**

CUDA’s execution hierarchy maps programmer-specified parallelism to GPU hardware:

* **Thread** (fine-grain): the smallest unit of execution. Each thread executes the kernel code for a single data element (e.g., one matrix element, one pixel). Threads have private registers and local memory.
* **Warp** (hardware concept): group of 32 threads that execute the same instruction in lock-step (SIMT). Branch divergence inside a warp hurts performance.
* **Block** (thread block): a group of threads that can cooperate:
  + Threads within a block can synchronize using \_\_syncthreads() and share fast **shared memory**.
  + Block size is limited (e.g., up to 1024 threads) and preferable dimensions are (x,y) or (x,y,z) for mapping 2D/3D problems.
  + Blocks are scheduled onto Streaming Multiprocessors (SMs). Multiple blocks can run on an SM depending on resource usage (registers/shared mem).
* **Grid**: collection of all blocks launched for a kernel. A kernel launch specifies the grid dimension and block dimension (<<<gridDim, blockDim>>>). Grids are independent; there is no implicit global synchronization between blocks in one kernel (except kernel launch boundary). Use multiple kernels or atomic ops to coordinate globally.

**Typical mapping**:

* For a 2D image: assign each thread to one pixel: blockDim = (16,16), gridDim = ((W+15)/16, (H+15)/16).

**Memory & Synchronization**:

* Shared memory: per-block low-latency scratchpad for inter-thread cooperation.
* Global memory: visible to all threads; higher latency; coalesced access matters.
* Synchronization:
  + Threads in a block → \_\_syncthreads().
  + No direct sync across blocks within one kernel; use successive kernels or atomics/locks.

**Best practice**:

* Use 2D/3D thread/block layouts to match data topology.
* Choose block sizes that are multiples of warp size (32), e.g., 128, 256, or a 16×16 tile (256 threads).
* Maximize occupancy but balance with register and shared memory usage.
* Avoid warp divergence and aim for coalesced global memory access.

**Q7. Construct an efficient CUDA kernel to generate an output matrix of size N×N from given input matrix N×N. (4M)**

If the transformation is element-wise (copy or simple per-element function), the most efficient approach is a 2D-block/2D-grid kernel so each thread writes one output element. Below are two kernels:

1. **Simple element-wise copy kernel** (easy, coalesced access if row-major):

// Kernel: copy A -> B

\_\_global\_\_ void copyKernel(const float \*A, float \*B, int N) {

int col = blockIdx.x \* blockDim.x + threadIdx.x;

int row = blockIdx.y \* blockDim.y + threadIdx.y;

if (row < N && col < N) {

int idx = row \* N + col; // row-major

B[idx] = A[idx];

}

}

1. **Tiled (shared-memory) approach** — useful when you later do neighborhood operations (transpose, convolution). For a simple copy it may be unnecessary, but here's a tiled pattern (tile size 16 recommended):

#define TILE 16

\_\_global\_\_ void copyTiledKernel(const float \*A, float \*B, int N) {

\_\_shared\_\_ float tile[TILE][TILE];

int tx = threadIdx.x, ty = threadIdx.y;

int col = blockIdx.x \* TILE + tx;

int row = blockIdx.y \* TILE + ty;

// load to shared memory

if (row < N && col < N) tile[ty][tx] = A[row \* N + col];

\_\_syncthreads();

// write back from tile

if (row < N && col < N) B[row \* N + col] = tile[ty][tx];

}

**Launch parameters recommendation**:

* dim3 block(16, 16);
* dim3 grid( (N + 15)/16, (N + 15)/16 );

This maps each thread to one matrix element, keeps memory accesses contiguous (row-major), and avoids divergence.

**Q8. Construct the CUDA main() required to use the kernel you wrote in Q7. (4M)**

A working host program (CUDA C) that allocates memory, copies data, launches the kernel, and copies back the result:

/\* cuda\_matrix\_copy.cu \*/

#include <stdio.h>

#include <cuda\_runtime.h>

\_\_global\_\_ void copyKernel(const float \*A, float \*B, int N) {

int col = blockIdx.x \* blockDim.x + threadIdx.x;

int row = blockIdx.y \* blockDim.y + threadIdx.y;

if (row < N && col < N) {

int idx = row \* N + col;

B[idx] = A[idx];

}

}

int main() {

const int N = 3; // example, change as needed

size\_t bytes = N \* N \* sizeof(float);

// host data

float h\_A[N\*N] = {1,2,3, 4,5,6, 7,8,9}; // row-major

float h\_B[N\*N];

// device pointers

float \*d\_A = nullptr, \*d\_B = nullptr;

cudaMalloc((void\*\*)&d\_A, bytes);

cudaMalloc((void\*\*)&d\_B, bytes);

cudaMemcpy(d\_A, h\_A, bytes, cudaMemcpyHostToDevice);

dim3 block(16,16);

dim3 grid( (N + block.x - 1) / block.x, (N + block.y - 1) / block.y );

copyKernel<<<grid, block>>>(d\_A, d\_B, N);

cudaError\_t err = cudaGetLastError();

if (err != cudaSuccess) {

printf("Kernel launch error: %s\n", cudaGetErrorString(err));

return 1;

}

cudaDeviceSynchronize();

cudaMemcpy(h\_B, d\_B, bytes, cudaMemcpyDeviceToHost);

// print output matrix

printf("Output matrix:\n");

for (int r=0; r<N; ++r) {

for (int c=0; c<N; ++c) printf("%f ", h\_B[r \* N + c]);

printf("\n");

}

cudaFree(d\_A);

cudaFree(d\_B);

return 0;

}

**How to compile:**

nvcc cuda\_matrix\_copy.cu -o cuda\_matrix\_copy

**Notes:**

* For large N, choose block sizes tuned for occupancy (e.g., 16×16).
* If the real transformation is more than a copy (e.g., transpose), adjust kernel accordingly to avoid bank conflicts and use tiled shared memory patterns.

**Q9. Sketch a block diagram and explain its architectural class scheme based on multiplicity of instruction streams and data streams (Flynn’s taxonomy). (2M)**

**Flynn’s taxonomy** classifies computers by instruction-stream and data-stream multiplicity:

1. **SISD (Single Instruction, Single Data)**
   * Traditional sequential processors: one instruction stream operating on one data stream.
   * Diagram: single CPU -> memory -> I/O.
2. **SIMD (Single Instruction, Multiple Data)**
   * One control unit issues the same instruction to multiple processing elements operating on different data (vector processors, GPUs at warp level).
   * Diagram (simplified):
   * Control Unit
   * |
   * -------------------------------
   * | PE0 | PE1 | PE2 |
   * | (data0) | (data1) | (data2) |
   * Use case: apply same operation across array (e.g., add scalar to vector).
3. **MISD (Multiple Instruction, Single Data)**
   * Multiple instruction streams operate on the same data stream (rare). Typical example: redundant pipelines for fault tolerance, or some streaming DSP pipelines where different transformations happen serially on same data stream.
   * Diagram: same input stream feeds several different functional units each executing different programs (or different operations in pipeline).
4. **MIMD (Multiple Instruction, Multiple Data)**
   * Each processor executes its own instruction stream on its own data. (Multi-core CPUs, clusters). Also SPMD (Single Program, Multiple Data) is a common programming approach under MIMD where each processor runs the same program but on different data and may take different control paths based on rank.
   * Diagram:
   * CPU0 (instr stream 0) -- local mem0
   * CPU1 (instr stream 1) -- local mem1
   * ...
   * Interconnect / Network between them

**Specific phrase from question:** “distinct instructions operating over the same data stream and its derivatives” — that describes **MISD** (though MISD is uncommon in general computing). Example: pipeline of different filters applied to the same incoming data stream (e.g., digital signal pipeline), or heterogeneous pipeline stages.

**Modern mapping examples:**

* GPUs: often described as SIMT (Single-Instruction Multiple-Threads) — a hybrid: within a warp behaves like SIMD, but different warps/blocks can execute different instruction streams (MIMD at a coarser level).
* Clusters / MPI: MIMD with message passing.
* Vector processors / DSPs: SIMD.

**Q10. Differentiate the role of hardware and software in programmatic levels of parallel processing in parallel computer systems. (2M)**

Parallel processing success depends on both hardware and software; they have distinct but complementary responsibilities.

**Hardware responsibilities**

* **Physical compute resources**: cores, ALUs, vector units, GPU SMs.
* **Memory hierarchy**: registers, L1/L2/L3 caches, shared memory, DRAM, and memory bandwidth.
* **Interconnect**: on-chip interconnects, NUMA topology, network topology (Infiniband, Ethernet) for distributed systems.
* **Synchronization primitives**: atomics, hardware barriers, fences, cache-coherence mechanisms.
* **DMA engines / HWA**: offload, specialized accelerators.
* **Performance characteristics**: latency, bandwidth, warp/divergence behavior, cache sizes—these determine effective parallel strategies.

Hardware goal: provide low-latency, high-throughput primitives and resources so software can efficiently map parallelism.

**Software responsibilities**

* **Parallel decomposition & mapping**: decide how to split work among threads/processes (domain decomposition, task graph).
* **Scheduling and load balancing**: ensure all processing elements have useful work (OS scheduler, runtime like OpenMP or MPI libraries).
* **Communication and synchronization**: implement message passing, barriers, locks at the software level, leveraging hardware primitives.
* **Memory management & data placement**: place data to exploit locality (NUMA-aware allocation, GPU host->device transfers).
* **Algorithmic adaptation**: redesign algorithms to exploit parallelism (blocking, pipelining, asynchronous tasks).
* **Fault tolerance & reliability**: checkpointing, redundant computation, error handling.
* **Compiler/runtime optimizations**: auto-vectorization, tiling, fusion, loop unrolling; runtime decides threads, tasks, and schedules.

Hardware provides raw building blocks; software organizes computation, coordinates resources, and adapts algorithms to the hardware. The best performance results from co-design — algorithm and software tuned to hardware characteristics (cache sizes, memory bandwidth, warp size, network topology).

**Example contrast**:

* A GPU has very high arithmetic throughput but limited single-thread performance and complex memory hierarchy. The **hardware** provides thousands of threads and shared memory; the **software** must provide massive parallelism, ensure coalesced memory accesses, and manage host-device transfers.
* A multi-socket CPU provides coherent caches and complex memory. The **hardware** offers coherence and fast context switches; the **software** (OS + runtime) must be NUMA-aware and schedule threads to CPUs to avoid remote memory penalties.

If you want, I can:

* Provide runnable MPI binaries/test instructions for the MPI code, or
* Adjust CUDA example to do a non-trivial transformation (transpose, convolution), or
* Give a concise 1-page printable answer sheet formatted for submission.

Which of those do you want next? (I'll just produce it now if you'd like — no waiting.)

Nice — I’ll answer each sub-question carefully with runnable code, clear explanations, diagrams (ASCII), and notes about correctness/limitations. Where you asked for programs I give complete C/C++/CUDA code you can compile and run. Let’s go.

**1A — OpenMP program using sections**

**Goal:** split work into sections: one section computes the product of the first N natural numbers, another computes the sum. After the parallel sections combine the results.

**Notes before code**

* The OpenMP parallel sections construct creates a team of threads and assigns each #pragma omp section block to one of the threads.
* There is an *implicit barrier* at the end of parallel sections (unless nowait is used), so it’s safe for sections to store results into shared variables and then combine after the region.
* Beware: the factorial (product) grows very fast — it overflows 64-bit for N > ~20. If you need large N, use big-integer libraries (GMP, boost::multiprecision) or compute in log-space.

**Code (C, OpenMP)**

/\* omp\_sections\_sum\_prod.c

Compile: gcc -fopenmp omp\_sections\_sum\_prod.c -o omp\_sections\_sum\_prod

\*/

#include <stdio.h>

#include <omp.h>

int main(void) {

int N = 12; // example; change as required

long long sum = 0;

unsigned long long product = 1ULL;

printf("Computing sum and product of first %d natural numbers using OpenMP sections\n", N);

#pragma omp parallel sections shared(sum,product)

{

#pragma omp section

{

long long local\_sum = 0;

for (int i = 1; i <= N; ++i) local\_sum += i;

sum = local\_sum; // safe because implicit barrier

printf("[section-sum] thread %d computed sum = %lld\n", omp\_get\_thread\_num(), sum);

}

#pragma omp section

{

unsigned long long local\_prod = 1ULL;

for (int i = 1; i <= N; ++i) local\_prod \*= (unsigned long long)i;

product = local\_prod; // safe because implicit barrier

printf("[section-prod] thread %d computed product = %llu\n", omp\_get\_thread\_num(), product);

}

} // implicit barrier: both sections finished here

// final combine or display

printf("Final results: sum = %lld, product = %llu\n", sum, product);

return 0;

}

**Run example:** ./omp\_sections\_sum\_prod  
If you want to control number of threads: export OMP\_NUM\_THREADS=2.

**1B — Time-shared common bus interconnection network**

**Goal:** show diagram; explain importance, advantages & disadvantages.

**Neat diagram (ASCII)**

+------------+

| Memory |

+------------+

|

+-------------------+

| Shared Bus (time-shared)

+-------------------+

+---------+ +--------+--------+ +----------+

| CPU 0 |-------| Arbiter (grant) |------| I/O dev |

+---------+ +-----------------+ +----------+

| |

+---------+ +---------+

| CPU 1 |-----------| Cache |

+---------+ +---------+

* All processors (CPUs), memory, and I/O devices connect to a single common bus.
* **Time-shared** means an arbiter grants bus access to one device at a time for a timeslot; devices take turns to use the bus.

**Why it’s important (use cases & impact)**

* **Simplicity & low cost**: a single electrical bus is simple to wire and cheap to implement for small-scale systems.
* **Broadcast capability**: easy to broadcast data or control signals to all participants (useful for cache coherence protocols in SMP).
* **Works well for small SMP systems and simple multiprocessor boards**.

**Arbitration methods**

* **Centralized** (single arbiter): round-robin, priority-based.
* **Distributed**: peer-to-peer token passing, daisy-chain.

**Advantages**

1. **Simple and inexpensive** implementation.
2. **Easy to broadcast** signals and implement cache coherence.
3. **Low protocol complexity** relative to switched networks.
4. **Predictable behavior** under light load.

**Disadvantages**

1. **Limited scalability** — bus bandwidth is shared: as node count grows, contention increases and throughput per node falls.
2. **Single point of contention** — performance degrades sharply with load.
3. **Single point of failure** (bus hardware failure affects entire system).
4. **Latency & collision**: average access latency rises with number of devices waiting.
5. **Physical limitations**: bus length and electrical loading limit number of devices and clock speed.

**Performance intuition**

* If bus bandwidth = B and there are P active devices contending, worst-case throughput per device ≈ B/P. High P drastically reduces effective parallel bandwidth.

**Conclusion:** time-shared common bus is excellent for small systems (SMP with few CPUs). For large parallel machines, switched interconnects (multistage networks, meshes, torus, fat-tree) are used instead.

**1C — Applications of parallel processing in Engineering and Cybersecurity**

**Engineering (detailed examples)**

1. **Computational Fluid Dynamics (CFD)** — solve Navier–Stokes PDEs using domain decomposition across many CPUs/GPUs (finite volume / finite element). Parallelization reduces simulation time from weeks to hours.
2. **Finite Element Analysis (FEA)** — structural analysis of large models; parallel sparse linear solvers (PETSc, Trilinos) and parallel meshing.
3. **Signal & Image Processing** — real-time processing on many channels (radar, sonar), 2D/3D image filters, tomography; GPUs accelerate convolution and FFTs.
4. **Control & Robotics** — parallel sensory data fusion, multi-sensor Kalman filters running on multicore/FPGA for low latency.
5. **Optimization & Parameter Sweeps** — large-scale parameter studies (Monte Carlo, hyperparameter search) distributed over clusters.
6. **CAD/CAM & Rendering** — parallel rendering, ray-tracing on GPU farms; parallel geometry processing.
7. **Machine Learning for Engineering** — training deep neural networks on GPUs (distributed data-parallel or model-parallel training).
8. **Real-time Simulation & HIL (Hardware-in-loop)** — parallel compute to meet strict real-time deadlines.

**Cybersecurity (detailed examples & ethical note)**

**Important:** parallel methods can be used defensively (logging, detection, analysis) — use only for lawful and authorized security testing.

1. **Network Intrusion Detection / Packet Inspection**
   * Inspect multiple packet flows in parallel using multicore systems or GPUs (pattern matching and regular expressions across flows).
   * Example: parallel Aho–Corasick on GPUs to detect signatures at line speed.
2. **Large-scale Log Analysis / SIEM**
   * Process terabytes of logs (MapReduce / Spark) to detect anomalies, correlate events, and extract indicators of compromise.
3. **Malware analysis & sandboxing**
   * Parallelize dynamic analysis across many sandboxes/VMs to test different payloads, configurations, or input stimuli quickly.
4. **Threat Hunting / Graph Analysis**
   * Large graphs (host–process–file relationships) processed with parallel graph algorithms to find lateral movement patterns.
5. **Cryptography & Crypto-analysis (defensive use)**
   * High-performance computing used to analyze cryptographic protocols and test strength of keys — only performed for authorized testing and security research.
6. **Password hash testing (authorized penetration testing)**
   * GPU clusters accelerate hashing algorithms. **Do not** use for unauthorized password cracking — only for permitted audits.
7. **Real-time DDoS mitigation**
   * Distributed detection and blackholing decisions across many nodes for rapid response.

**Common parallel tools & stacks used**

* **HPC libraries**: MPI, OpenMP, CUDA, OpenCL.
* **Big data**: Hadoop, Spark, Flink.
* **GPU-accelerated NIDS / pattern matching**.
* **Parallel graph frameworks**: GraphX, Gunrock (GPU), etc.

**2A — MPI program: root sends array parts to two slaves; slaves compute partial sums; root collects and outputs final sum (point-to-point only)**

**Program design**

* Run with **3 processes**: rank 0 = root, ranks 1 & 2 = slaves.
* Root initializes array, computes chunk sizes, sends a count then chunk data to each slave using MPI\_Send.
* Slave MPI\_Recv count, allocate buffer, receive chunk, compute sum, send partial sum back via MPI\_Send.
* Root receives two partial sums and prints final sum.

**Code (C + MPI)**

/\* mpi\_sum\_three\_processes.c

Compile: mpicc mpi\_sum\_three\_processes.c -o mpi\_sum

Run: mpirun -np 3 ./mpi\_sum

\*/

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \*\*argv) {

MPI\_Init(&argc, &argv);

int rank, size;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

if (size != 3) {

if (rank == 0) fprintf(stderr, "This program requires exactly 3 processes (1 root + 2 slaves)\n");

MPI\_Finalize();

return 1;

}

const int N = 11; // example array size (can change)

if (rank == 0) {

int \*arr = malloc(N \* sizeof(int));

for (int i = 0; i < N; ++i) arr[i] = i + 1; // 1..N

int counts[2] = { N/2, N - N/2 }; // split roughly

// send count and data to slave 1

MPI\_Send(&counts[0], 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);

MPI\_Send(arr, counts[0], MPI\_INT, 1, 1, MPI\_COMM\_WORLD);

// send count and data to slave 2

MPI\_Send(&counts[1], 1, MPI\_INT, 2, 0, MPI\_COMM\_WORLD);

MPI\_Send(arr + counts[0], counts[1], MPI\_INT, 2, 1, MPI\_COMM\_WORLD);

// receive partial sums

int sum1 = 0, sum2 = 0;

MPI\_Recv(&sum1, 1, MPI\_INT, 1, 2, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

MPI\_Recv(&sum2, 1, MPI\_INT, 2, 2, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

printf("[root] partial sums: %d, %d; final sum = %d\n", sum1, sum2, sum1 + sum2);

free(arr);

} else {

// slave: receive count, buffer, compute sum, send back

int count = 0;

MPI\_Recv(&count, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

int \*buf = malloc(count \* sizeof(int));

MPI\_Recv(buf, count, MPI\_INT, 0, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

int partial = 0;

for (int i = 0; i < count; ++i) partial += buf[i];

MPI\_Send(&partial, 1, MPI\_INT, 0, 2, MPI\_COMM\_WORLD);

printf("[rank %d] computed partial sum = %d\n", rank, partial);

free(buf);

}

MPI\_Finalize();

return 0;

}

**2B — Odd–Even transposition sort: two phases in each iteration**

**Odd–Even transposition sort** (parallel version of bubble sort) performs N iterations (for N items). Each iteration has two phases:

1. **Odd phase** — compare and possibly swap adjacent pairs whose first index is odd: pairs (1,2), (3,4), (5,6), ...
   * All compare-swaps among those pairs can be done in parallel because pairs are disjoint.
2. **Even phase** — compare and possibly swap adjacent pairs whose first index is even: pairs (0,1), (2,3), (4,5), ...
   * Again, pairs are disjoint and can be done in parallel.

Repeat (odd-phase then even-phase) typically N times (or until sorted). After at most N full iterations, the list is sorted.

**Pseudocode (indexing 0..N-1)**

for iter = 0 to N-1:

// odd phase (compare indices 1,2 ; 3,4 ; ...)

for i = 1; i+1 < N; i += 2 in parallel:

if A[i] > A[i+1] swap(A[i],A[i+1])

// even phase (compare indices 0,1 ; 2,3 ; ...)

for i = 0; i+1 < N; i += 2 in parallel:

if A[i] > A[i+1] swap(A[i],A[i+1])

**Complexity:** O(N) iterations × O(N/2) compares per phase → worst-case O(N^2) comparisons, but works well for parallel hardware because disjoint compares are independent in each phase.

**2C — MPI APIs (identify & example)**

**(i) One process (root) sends data to all other processes**

* **API**: MPI\_Bcast
* **Signature (C)**:

int MPI\_Bcast(void \*buffer, int count, MPI\_Datatype datatype, int root, MPI\_Comm comm);

* **Behavior**: root’s buffer is sent to all ranks in comm. After call, all processes have same buffer contents.
* **Example**:

int N = 100;

double data[100]; // initialized on root

MPI\_Bcast(data, N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

**(ii) All processes collect data from all others and perform an operation on the data**

* If you want **each process to get the reduced (aggregated) result**, use MPI\_Allreduce.
* **API**: MPI\_Allreduce
* **Signature (C)**:

int MPI\_Allreduce(const void \*sendbuf, void \*recvbuf, int count,

MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm);

* **Behavior**: perform reduction (e.g., MPI\_SUM, MPI\_MAX) across all sendbuf values; result written to recvbuf on all processes.
* **Example (global sum of 1 int per process)**:

int myval = rank; // each rank provides an int

int totalsum;

MPI\_Allreduce(&myval, &totalsum, 1, MPI\_INT, MPI\_SUM, MPI\_COMM\_WORLD);

// now totalsum is sum of all ranks on every process

(If the need is to gather all raw data without reducing, use MPI\_Allgather/MPI\_Gather.)

**3A — Mesh interconnection network (3×3) AND chordal ring derived from it**

**3×3 mesh (node indexing & ASCII)**

Index the mesh row-major (row 0..2, col 0..2):

(0,0)[0] --- [1] --- [2]

| | |

[3] --- [4] --- [5]

| | |

[6] --- [7] --- [8]

* **Adjacency list**:
  + 0: neighbors {1,3}
  + 1: neighbors {0,2,4}
  + 2: neighbors {1,5}
  + 3: neighbors {0,4,6}
  + 4: neighbors {1,3,5,7}
  + 5: neighbors {2,4,8}
  + 6: neighbors {3,7}
  + 7: neighbors {4,6,8}
  + 8: neighbors {5,7}

**Mesh properties:** degree ≤ 4, diameter = (rows-1)+(cols-1) = 4 for 3×3 if Manhattan distance between opposite corners.

**Constructing a Chordal Ring from the same 9 nodes**

A **chordal ring** is typically built on n nodes arranged in a ring (cycle) and then adding “chords” connecting nodes i and i + s (mod n) for one or more skip distances s. This reduces diameter while keeping low degree.

One simple construction for n=9:

* Start with a ring ordering of nodes: 0—1—2—3—4—5—6—7—8—0.
* Add chord of length 3: connect each i to (i+3) mod 9. (This corresponds to connecting nodes that were vertically separated in the mesh.)

Resulting edges:

* Ring edges: (0,1),(1,2),(2,3),(3,4),(4,5),(5,6),(6,7),(7,8),(8,0)
* Chord edges (skip 3): (0,3),(1,4),(2,5),(3,6),(4,7),(5,8),(6,0),(7,1),(8,2)

**ASCII (circular layout with chords):**

0

/ \

8 1

/ \

7 2

\ /

6 --- 5

|

4

(Chords are connections like 0–3,1–4,2–5 etc.)

**Why this maps to mesh:** with chord length 3 you connect nodes that were three steps away on the ring — this corresponds to vertical connections in the original 3×3 mesh if you list nodes row-major. So chordal ring approximates the mesh connectivity but as a ring plus long-distance shortcuts (chords).

**Properties comparison**

* Mesh: small node degree, simple planar layout, local neighbors → good for locality.
* Chordal ring: keeps low node degree but smaller diameter than ring thanks to chords. Good compromise between ring and more connected topologies.

**3B — CUDA APIs and code structure for timing a kernel execution**

**Which APIs to use to measure GPU kernel execution time**

* cudaEventCreate(cudaEvent\_t \*event) — create event object.
* cudaEventRecord(cudaEvent\_t event, cudaStream\_t stream=0) — record event on a stream (start/stop).
* cudaEventSynchronize(cudaEvent\_t event) — wait until event is complete.
* cudaEventElapsedTime(float \*ms, cudaEvent\_t start, cudaEvent\_t stop) — returns elapsed time in **milliseconds** (float) between two events.
* cudaDeviceSynchronize() — forces host to wait until all device tasks are complete (useful but coarser).
* cudaMemcpy, cudaMemcpyAsync — host ↔ device copy functions.
* cudaMalloc, cudaFree — allocate/free device memory.
* cudaGetLastError() — check kernel launch errors.
* cudaStreamCreate/Destroy — optional: measure on custom stream.

**Typical measurement patterns**

* **Measure only kernel execution**: record start after H2D copy is done; launch kernel; record stop; synchronize and compute elapsed time.
* **Measure end-to-end (including memcopies)**: place start before cudaMemcpy & stop after D2H; but GPU events measure only GPU work queued on a stream — host memcpy (synchronous) must be counted separately or with events if using async copies and streams.

**Minimal example (measure kernel only)**

/\* kernel\_timing\_example.cu \*/

#include <stdio.h>

#include <cuda\_runtime.h>

\_\_global\_\_ void simpleKernel(float \*d, int N) {

int idx = blockIdx.x \* blockDim.x + threadIdx.x;

if (idx < N) d[idx] = idx \* 2.0f;

}

int main() {

const int N = 1<<20;

size\_t bytes = N \* sizeof(float);

float \*h = (float\*)malloc(bytes);

float \*d = NULL;

cudaMalloc(&d, bytes);

// host->device copy (synchronous)

cudaMemcpy(d, h, bytes, cudaMemcpyHostToDevice);

// create events

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

// launch config

int block = 256;

int grid = (N + block - 1) / block;

// record start, launch kernel, record stop

cudaEventRecord(start, 0);

simpleKernel<<<grid, block>>>(d, N);

cudaEventRecord(stop, 0);

// wait for stop (kernel completion)

cudaEventSynchronize(stop);

float elapsed\_ms = 0.0f;

cudaEventElapsedTime(&elapsed\_ms, start, stop);

printf("Kernel execution time: %f ms\n", elapsed\_ms);

// cleanup

cudaEventDestroy(start);

cudaEventDestroy(stop);

cudaMemcpy(h, d, bytes, cudaMemcpyDeviceToHost);

cudaFree(d);

free(h);

return 0;

}

**Notes & tips**

* Use cudaEventRecord on the same stream you launch the kernel (default stream 0 is fine).
* cudaEventElapsedTime returns **milliseconds** with millisecond (sub-ms) resolution depending on the GPU.
* For precise profiling (memory vs compute), measure H2D, kernel, D2H separately using events placed before and after each operation.
* If you call cudaDeviceSynchronize() instead of events, you can measure wall clock on the host, but this includes host scheduling delays and is coarser.

If you want, I can:

* Convert the OpenMP program to use reduction clauses or demonstrate a big-integer factorial using boost::multiprecision.
* Provide an MPI implementation of **odd–even transposition** (parallel sort) across processes.
* Create a PNG of the bus/mesh/chordal-ring diagrams or produce a LaTeX-ready diagram.

Which of those would help you most next?

Good — I’ll answer each sub-question carefully and in depth. For each programming question I’ll give complete, compilable code and explain how it works and why. Where diagrams are requested I’ll use clear ASCII diagrams and explain meanings and trade-offs.

**1A — OpenMP reduction clause: explanation + program**

**What reduction does (explanation)**

* The OpenMP reduction(op:varlist) clause instructs the runtime to give each thread a private copy of each variable in varlist, initialize that private copy to the *identity* of op (for + identity is 0, for \* identity is 1, for max identity is -∞, etc.), let the thread update its private copy while executing the parallel region, and when the region ends (or the construct finishes) combine all private copies into a single result using op. The combination is done with a thread-safe reduction operation.
* Benefits: avoids race conditions without explicit locks; typically faster than a critical or atomic when many updates are needed.
* Caveats: for non-associative ops (floating-point +), results depend on reduction order — small rounding differences can appear compared to sequential sum.

**Example task**

Read an array and compute the sum of elements using #pragma omp parallel for reduction(+:sum).

**Code (C, OpenMP)**

Compile: gcc -fopenmp omp\_reduction\_sum.c -o omp\_reduction\_sum

/\* omp\_reduction\_sum.c

Usage: ./omp\_reduction\_sum

Input: first line N, next N integers (space or newline separated)

\*/

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

int main(void) {

int N;

if (scanf("%d", &N) != 1) {

fprintf(stderr, "Failed to read N\n");

return 1;

}

int \*a = malloc(sizeof(int) \* N);

if (!a) { perror("malloc"); return 1; }

for (int i = 0; i < N; ++i) {

if (scanf("%d", &a[i]) != 1) { fprintf(stderr,"Bad input\n"); free(a); return 1; }

}

long long sum = 0;

// The reduction clause ensures thread-local accumulators are used,

// then combined into 'sum' at the end.

#pragma omp parallel for reduction(+:sum)

for (int i = 0; i < N; ++i) {

sum += a[i];

}

printf("Sum of %d elements = %lld\n", N, sum);

free(a);

return 0;

}

**Notes & tips**

* To control number of threads: set environment variable OMP\_NUM\_THREADS or call omp\_set\_num\_threads(...).
* The reduction clause supports many operators: +, \*, - (order-dependent), max, min, bitwise ops, logical ops. For custom user-defined reductions you need OpenMP 4.0+ custom reduction support.

**1B — Block diagram & explanation of shared memory in Symmetric Multiprocessor (SMP)**

**ASCII block diagram (simple SMP)**

+--------------------------------+

| Shared Main Memory |

+--------------------------------+

^ ^ ^ ^

| | | |

+----+--+ +---+--+ +--+---+ +--+---+

| CPU0 | | CPU1 | | CPU2 | | CPU3 |

| Core0 | | Core0| | Core0| | Core0|

+---+---+ +---+--+ +---+--+ +---+---+

\ | | /

\ | | /

+-----------------------+

| Cache-coherence bus |

+-----------------------+

**Explanation (conceptual)**

* **SMP (Symmetric Multiprocessor)**: multiple identical processors/cores share a single, coherent physical memory address space. Each processor runs its own thread(s) and can directly load/store the same global memory addresses.
* **Cache coherence**: each processor typically has private caches (L1, maybe L2). Coherence protocol (MESI, MOESI, etc.) ensures that writes by one processor become visible to others in a consistent manner.
* **Interconnect**: processors and memory communicate through a bus or point-to-point interconnect. For small SMPs a common bus is used; for larger ones a crossbar or NUMA interconnect is used.

**Key points**

* Shared memory simplifies programming: threads access shared variables directly and can synchronize with locks, barriers, atomics.
* Performance depends on memory contention, cache coherence traffic, and locality. For high contention, throughput can drop.
* For large systems NUMA effects matter: memory access time depends on which CPU’s local memory bank holds the data.

**1C — Definitions: node degree and network diameter (static interconnection networks)**

**Node degree**

* **Definition:** For a static interconnection network (graph) the *node degree* is the number of direct links/edges connected to a node. If every node has d links, the network is d-regular.
* **Interpretation:** Node degree measures hardware cost per node (more links → higher cost/complexity) and also the parallel network’s potential to route traffic (higher degree typically reduces congestion).

**Examples**

* Ring: degree = 2 (each node connected to two neighbors).
* 2D mesh (interior node): degree = 4. Corner nodes have degree 2, edge nodes 3.
* Hypercube of dimension k: degree = k (since each node differs in exactly k bit positions and connected to k neighbors).

**Network diameter**

* **Definition:** The *network diameter* is the maximum over all pairs of nodes of the length (number of hops) of the shortest path between them. It’s the worst-case minimum distance between any two nodes.
* **Interpretation:** Diameter approximates the worst-case communication latency (in hops). Smaller diameter generally means lower worst-case latency.

**Examples**

* Ring of n nodes: diameter = ⌊n/2⌋.
* 2D r × c mesh: diameter = (r-1) + (c-1) (Manhattan distance between opposite corners). For a 3×3 mesh, diameter = 4.
* k-dimensional hypercube with n = 2^k nodes: diameter = k = log2(n) — excellent scaling.

**Trade-offs**: Increasing node degree typically reduces diameter (better connectivity) but increases hardware cost and routing complexity.

**2A — MPI program: scatter equal portions, each process modifies elements and computes partial sum, collect partial sums (use MPI\_Reduce)**

**Assumptions**

* Number of processes P evenly divides array length N (i.e., N % P == 0).
* Each process will add a fixed value (for example add\_val = 1) to each element in its portion and then compute the sum of its modified portion. (If you want a different per-element addition, replace add\_val accordingly.)

**Code (C + MPI) — uses MPI\_Scatter and MPI\_Reduce**

Compile: mpicc mpi\_scatter\_reduce.c -o mpi\_scatter\_reduce  
Run: mpirun -np P ./mpi\_scatter\_reduce (ensure P divides N)

/\* mpi\_scatter\_reduce.c \*/

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \*\*argv) {

MPI\_Init(&argc, &argv);

int rank, size;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

const int N = 12; // total elements (must be divisible by size)

const int add\_val = 1; // value to add to each element

if (N % size != 0) {

if (rank == 0) fprintf(stderr,"N must be divisible by number of processes\n");

MPI\_Finalize();

return 1;

}

int chunk = N / size;

int \*array = NULL;

if (rank == 0) {

array = malloc(N \* sizeof(int));

for (int i = 0; i < N; ++i) array[i] = i + 1; // example initialization 1..N

printf("[root] initial array: ");

for (int i=0;i<N;++i) printf("%d ", array[i]);

printf("\n");

}

// allocate local buffer

int \*local = malloc(chunk \* sizeof(int));

// Scatter chunk elements to all processes

MPI\_Scatter(array, chunk, MPI\_INT, local, chunk, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Each process performs addition to each element and computes partial sum

int partial\_sum = 0;

for (int i = 0; i < chunk; ++i) {

local[i] += add\_val; // modify element (example operation)

partial\_sum += local[i];

}

printf("[rank %d] partial sum = %d\n", rank, partial\_sum);

// Collect partial sums on root via reduction

int global\_sum = 0;

MPI\_Reduce(&partial\_sum, &global\_sum, 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (rank == 0) {

printf("[root] global sum after adding %d to each element = %d\n", add\_val, global\_sum);

}

free(local);

if (array) free(array);

MPI\_Finalize();

return 0;

}

**Key MPI calls used**

* MPI\_Scatter: distribute contiguous chunks of the array to each process.
* MPI\_Reduce(..., MPI\_SUM, root): collect and sum partial sums on root.

**2B — Built-in MPI reduction operators: interpretation + example API**

**Common built-in reduction ops (semantics)**

* MPI\_SUM — sum of values.
* MPI\_PROD — product of values.
* MPI\_MAX / MPI\_MIN — maximum / minimum of values.
* MPI\_MAXLOC / MPI\_MINLOC — return value+rank of max/min (used with pair datatypes, e.g. MPI\_2INT).
* Bitwise logical/bitwise ops:
  + MPI\_LAND, MPI\_LOR, MPI\_LXOR — logical AND/OR/XOR (treat non-zero as true).
  + MPI\_BAND, MPI\_BOR, MPI\_BXOR — bitwise AND/OR/XOR.
* MPI\_REPLACE — overwrite: the result is root’s own sendbuf (useful with MPI\_Allreduce alternative patterns).

**Notes**

* The operation semantics depend on datatype (e.g., integer vs floating) and the operation.
* MPI\_MAXLOC/MPI\_MINLOC require special handling with location pairs (value + rank) to identify where extrema occur.

**Example: Using MPI\_Allreduce with MPI\_MAX**

* Suppose every process has a double local\_val. To find the maximum across all processes and have the max available on every process:

double local\_val = ...;

double global\_max = 0.0;

MPI\_Allreduce(&local\_val, &global\_max, 1, MPI\_DOUBLE, MPI\_MAX, MPI\_COMM\_WORLD);

// now global\_max contains the maximum local\_val across all ranks, on every process

**2C — Skeleton showing potential deadlock using blocking point-to-point calls**

**Scenario that can deadlock**

Two processes (rank 0 and rank 1) both call blocking MPI\_Send to each other with large messages before posting receives. If the MPI implementation uses rendezvous protocol for large messages (i.e., MPI\_Send blocks until the receiver posts MPI\_Recv), both sends wait forever → deadlock.

**Deadlock skeleton (C + MPI)**

/\* deadlock\_skeleton.c \*/

#include <mpi.h>

int main(int argc, char \*\*argv) {

MPI\_Init(&argc, &argv);

int rank;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

const int LARGE = 1000000;

int \*buf = malloc(LARGE \* sizeof(int));

for (int i=0;i<LARGE;i++) buf[i]=rank; // big message

if (rank == 0) {

// If MPI\_Send blocks until receiver calls MPI\_Recv, and rank 1 does the same,

// this will deadlock.

MPI\_Send(buf, LARGE, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);

MPI\_Recv(buf, LARGE, MPI\_INT, 1, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

} else if (rank == 1) {

MPI\_Send(buf, LARGE, MPI\_INT, 0, 1, MPI\_COMM\_WORLD);

MPI\_Recv(buf, LARGE, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

}

free(buf);

MPI\_Finalize();

return 0;

}

**How to avoid deadlock**

* Use MPI\_Sendrecv which performs send and receive atomically:

MPI\_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag,

recvbuf, recvcount, recvtype, src, recvtag,

MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

* Or use non-blocking MPI\_Isend / MPI\_Irecv and MPI\_Waitall.
* Or ensure ordering: one process does Send then Recv, the other does Recv then Send.

**4C — Message-passing vs Shared-memory models (comparison & detailed discussion)**

**Models (brief)**

* **Shared-memory model:** multiple threads (lightweight processes) share a common address space and communicate by reading/writing shared variables. Synchronization via mutexes/locks, semaphores, atomics, barriers. Example APIs: POSIX threads (pthreads), OpenMP, C++ threads.
* **Message-passing model:** processes have private memory and communicate explicitly by sending/receiving messages (copy semantics). No implicit shared state. Example API: MPI.

**Programming differences**

* **Data visibility:**
  + Shared-memory: any thread can directly access variables; care needed for synchronization and memory ordering.
  + Message-passing: data is explicitly transferred; no shared-memory coherence issues.
* **Synchronization primitives:**
  + Shared: locks, condition variables, atomics, barriers (\_\_syncthreads() in CUDA for block).
  + Message-passing: send/recv synchronization, collective operations (broadcast/reduce/barrier).
* **Decomposition style:**
  + Shared: fine-grained parallelism, dynamic work-stealing, easily share complex data structures.
  + Message-passing: often coarse-grained with explicit data decomposition; good for distributed memory (clusters).
* **Scalability:**
  + Shared-memory: relatively easier to program but limited by memory bandwidth and coherence traffic. Large-scale shared-memory systems are complex and expensive.
  + Message-passing: scales to many nodes (clusters); programmer must manage communication and data distribution.
* **Performance characteristics:**
  + Shared-memory: lower latency for data access (local loads/stores) but contention on global data and cache-coherence overhead.
  + Message-passing: communication cost = message latency + (message\_size / bandwidth); often higher latency but predictable and scalable.
* **Failure model / robustness:**
  + Message-passing systems (distributed) may need explicit fault-tolerance logic; shared-memory often fails as a whole if a process crashes.
* **Ease of debugging:**
  + Shared-memory concurrency bugs include data races and deadlocks (hard to detect).
  + Message-passing bugs include mismatched sends/receives, deadlocks due to blocking patterns.

**When to choose which**

* **Single machine / multi-core / simple sharing** → shared-memory (OpenMP, pthreads) is convenient and efficient.
* **Cluster / distributed memory / large-scale HPC** → message-passing (MPI) is preferred for scalability.
* **Hybrid**: Many HPC codes use a hybrid model (MPI between nodes + OpenMP or CUDA within a node/GPU) to exploit both distributed and shared-memory parallelism.

**5B — CUDA kernel: Multiply A (M×N) and B (N×P) → C (M×P)**

**As requested:** create P threads where each thread computes one column of the result. That means each thread computes all M entries of a single column (so degree of parallelism = number of columns P).

Important note: launching only P threads seriously underutilizes the GPU if P is small. The canonical efficient approach uses a 2D grid of threads with each thread computing one element C[row][col] and uses tiling/shared memory for performance. Below I implement the requested single-thread-per-column approach and then give notes on a more efficient method.

**CUDA kernel (one thread per column)**

* Each thread col loops over rows r = 0..M-1.
* For each row computes dot product of A[row,*] with B[*,col].
* Memory layout: row-major: A[row\*N + k], B is k\*P + col, C is row\*P + col.

**Full example (single-file .cu)**

Compile: nvcc matmul\_oneThreadPerCol.cu -o matmul\_col  
Run: ./matmul\_col

// matmul\_oneThreadPerCol.cu

#include <stdio.h>

#include <cuda\_runtime.h>

#include <stdlib.h>

\_\_global\_\_ void matmul\_col\_kernel(const float \*A, const float \*B, float \*C, int M, int N, int P) {

int col = blockIdx.x \* blockDim.x + threadIdx.x;

if (col >= P) return;

// compute column 'col' of C (all rows 0..M-1)

for (int row = 0; row < M; ++row) {

float sum = 0.0f;

int a\_base = row \* N;

int b\_base = col;

// dot product of row of A and column of B

for (int k = 0; k < N; ++k) {

sum += A[a\_base + k] \* B[k \* P + col];

}

C[row \* P + col] = sum;

}

}

int main() {

// example sizes (change as needed)

const int M = 4, N = 3, P = 5;

size\_t Asz = M \* N \* sizeof(float);

size\_t Bsz = N \* P \* sizeof(float);

size\_t Csz = M \* P \* sizeof(float);

float \*h\_A = (float\*)malloc(Asz);

float \*h\_B = (float\*)malloc(Bsz);

float \*h\_C = (float\*)malloc(Csz);

// initialize (A row-major) and (B row-major where B[k\*P + j] is element)

for (int i = 0; i < M \* N; ++i) h\_A[i] = (float)(i + 1); // 1..M\*N

for (int i = 0; i < N \* P; ++i) h\_B[i] = (float)(i + 1);

float \*d\_A, \*d\_B, \*d\_C;

cudaMalloc(&d\_A, Asz);

cudaMalloc(&d\_B, Bsz);

cudaMalloc(&d\_C, Csz);

cudaMemcpy(d\_A, h\_A, Asz, cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, h\_B, Bsz, cudaMemcpyHostToDevice);

// Launch P threads (one per column). Use block size up to 1024 threads.

int block = 256;

int grid = (P + block - 1) / block;

matmul\_col\_kernel<<<grid, block>>>(d\_A, d\_B, d\_C, M, N, P);

cudaDeviceSynchronize();

cudaMemcpy(h\_C, d\_C, Csz, cudaMemcpyDeviceToHost);

printf("C (M x P) =\n");

for (int r = 0; r < M; ++r) {

for (int c = 0; c < P; ++c) printf("%8.3f ", h\_C[r \* P + c]);

printf("\n");

}

cudaFree(d\_A); cudaFree(d\_B); cudaFree(d\_C);

free(h\_A); free(h\_B); free(h\_C);

return 0;

}

**Explanation & complexity**

* Each thread computes M × N multiply-add operations (for its column). Total FLOPs = M × N × P as expected.
* If P is small, GPU threads are underutilized because GPUs demand thousands of threads to hide memory/latency. Use this kernel only when required by assignment; for performance, prefer many threads (one per output element or blocks of tiles).

**Better (more efficient) approach (summary)**

* Launch a 2D grid with gridDim.x = (P+TX-1)/TX, gridDim.y = (M+TY-1)/TY and blockDim = (TX,TY). Let each thread compute one element C[r, c]. Use tiling and \_\_shared\_\_ memory to load tiles of A and B so global memory access is coalesced and arithmetic intensity is increased — this is the standard high-performance GEMM pattern (cuBLAS uses highly optimized kernels).

**5C — What is GPU computing?**

**Short definition**

**GPU computing** uses Graphics Processing Units (GPUs) — originally designed for rendering graphics — as general-purpose parallel processors to accelerate computationally intensive tasks. It maps massive data-parallel work to hundreds or thousands of GPU cores to achieve high throughput.

**Key characteristics**

* **Massive parallelism**: GPUs have many cores (CUDA cores / stream processors) organized into Streaming Multiprocessors (SMs). Best for workloads with high data parallelism (same operation applied to many data items).
* **SIMT (Single Instruction, Multiple Threads)**: GPU executes groups of threads (warps) in lock-step on the same instruction; divergence reduces efficiency.
* **Memory hierarchy**: registers → shared memory (per-block low-latency) → L1/L2 caches → global DRAM (GDDR/HBM). Proper use of shared memory and coalesced global memory accesses is critical for performance.
* **Programming models & APIs**: CUDA (NVIDIA), HIP (AMD), OpenCL, SYCL. These allow writing kernels (device functions) and launching them with many threads.
* **High arithmetic throughput**: GPUs provide very large FLOP/s for floating-point workloads — excellent for linear algebra, CNNs, simulations, image processing, signal processing.
* **Limitations**: high host-device transfer overhead (PCIe or NVLink), limited per-thread memory, warp divergence and memory-bound workloads reduce efficiency.

**Typical use-cases**

* Deep learning (training and inference)
* Dense linear algebra (matrix multiply, solvers)
* Image & video processing
* Scientific simulations (CFD, molecular dynamics)
* Monte-Carlo simulations and data analytics
* Encryption/hashing (authorized use)

**Summary**

GPU computing is about structuring work to exploit massive parallelism and memory hierarchy of GPUs; when done well it yields orders-of-magnitude speedups for suitable algorithms compared to CPU-only implementations.

If you want, I can now:

* Convert any of the example programs into a ready-to-run tarball, or
* Provide a high-performance CUDA matrix multiply using tiling and \_\_shared\_\_ memory (fast, but longer), or
* Provide an MPI version of odd–even transposition sort.

Which follow-up would be most useful?