Summer School on Supercomputing & AI for Tech Entrepreneurs
29-31 August 2025 Namal University Mianwali

Think Parallel: The Art of Parallel Programming

PRESENTER: PROF. DR. TASSADAQ HUSSAIN

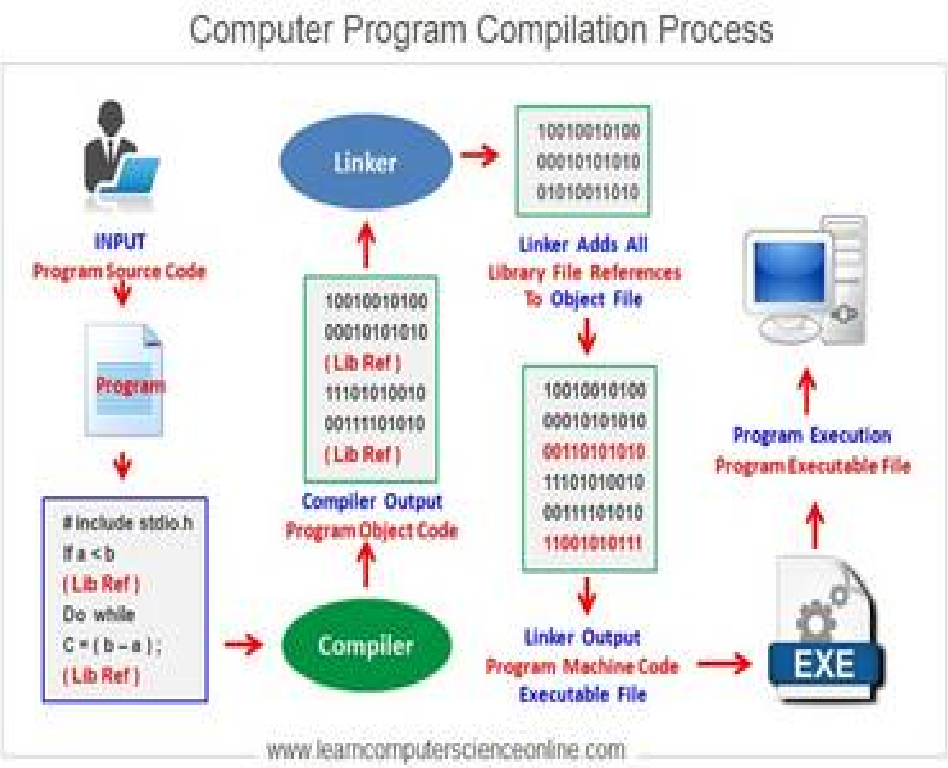
- **Introduction to Programming**
- **Parallel Approaches**
- **Shared Memory Programming**
  - Multi-core
  - Accelerator Offloading
- **Distributed Memory Programming**
  - MPI
  - AI
- **HPC and Cloud**

What is Programming?

The art/science of giving precise instructions to a computer to perform tasks.

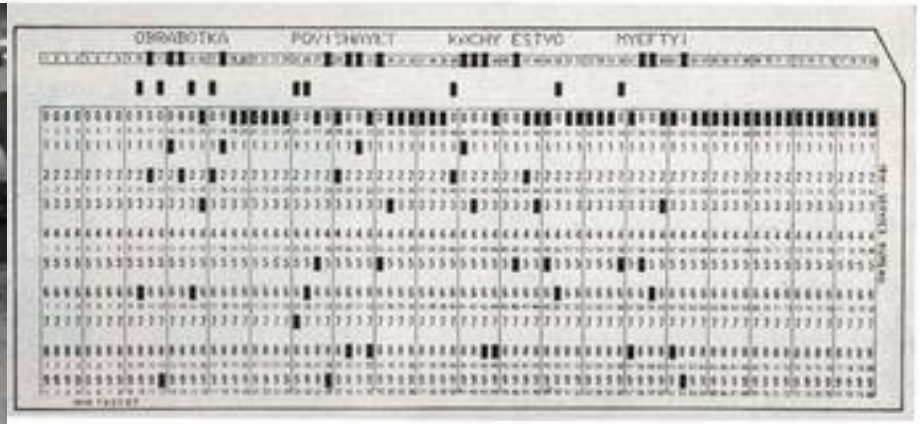
Analogy: Like teaching a robot step-by-step instructions for cooking or driving.

Goal: Translate human thought into machine-executable commands.



Programming in the Early Mainframe Era

- 1800s – Mechanical Computer (Manually Programmed)
- 1940s–1950s: Programs written in machine language (0s and 1s).
  - Very tedious and error-prone.



Before keyboards and screens: Early mainframes were programmed by toggling switches and monitoring lights.

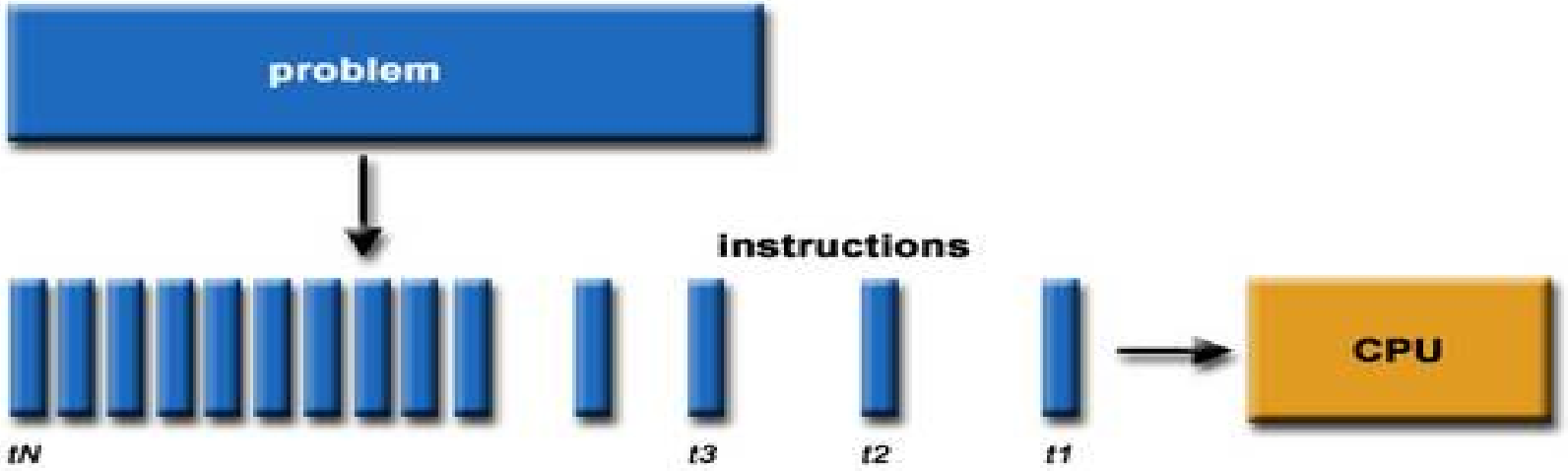
Programming Languages and Compilers

- 1950s–1960s:: Assembly language introduced – mnemonics instead of binary.
  - Still tied closely to hardware.
- Procedural Era (1960s–1980s): Languages: Fortran, C, Pascal.
  - Step-by-step instructions, good for scientific/engineering problems.
- Object-Oriented Era (1980s–2000s): Languages.
  - C++, Java. Organize programs into “objects” → easier to manage complexity.
- Scripting/Web Era (1990s–2000s): Languages.
  - Python, PHP, JavaScript. Focused on quick development, web interactivity.
- Data/AI Era (2010s–Present): Python, R, Julia, CUDA.
  - Heavy reliance on libraries, frameworks, GPUs.Parallel/Concurrent Era (Ongoing)
  - Languages & models: MPI, OpenMP, CUDA, TensorFlow, Spark.
  - Designed for multi-core, clusters, cloud, GPUs.

- Introduction to Programming
- **Think Parallel**
- Shared Memory Programming
- Distributed Memory Programming
- HPC and Cloud

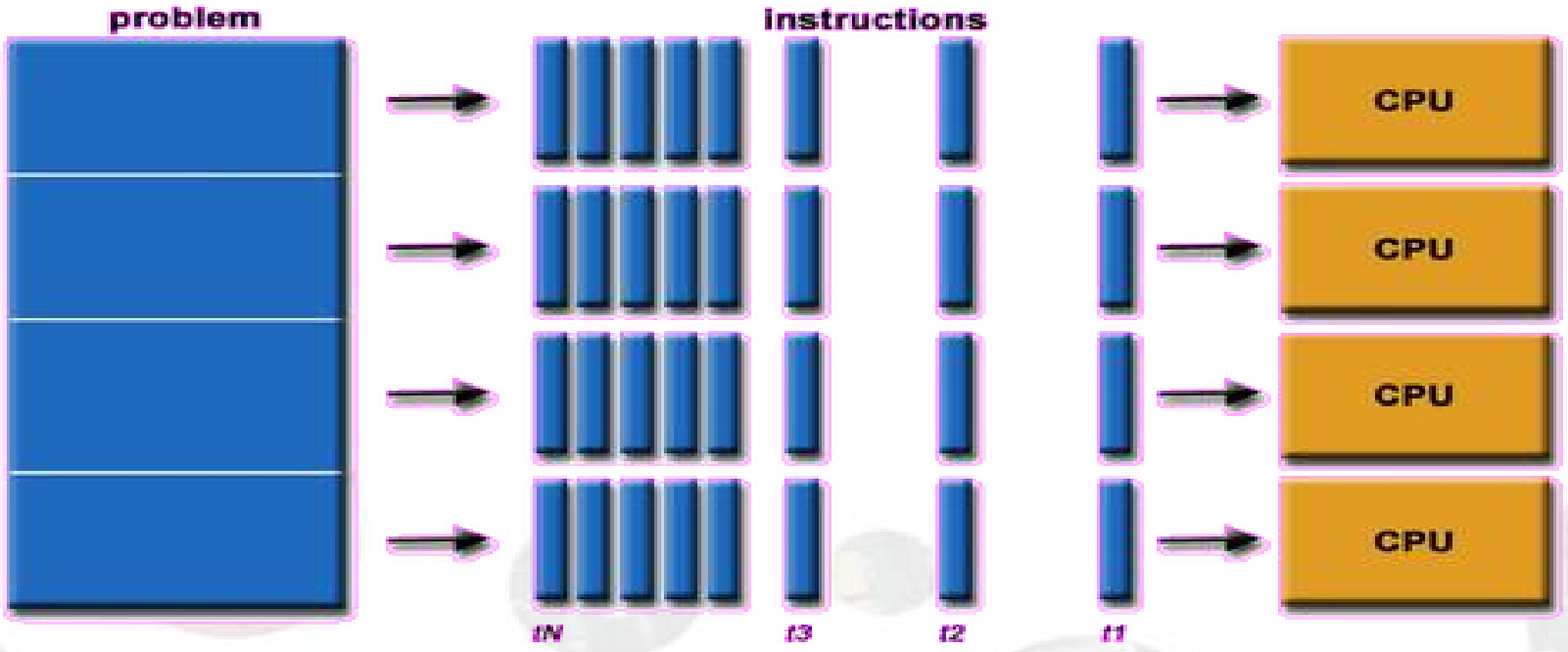
Programming ?

- C is fundamentally a sequential programming language.
- Traditionally, software has been written for ***serial*** computation:
  - To be run on a single computer having a single Central Processing Unit (CPU);
  - A problem is broken into a discrete series of instructions.
  - Instructions are executed one after another.
  - Only one instruction may execute at any moment in time.



Parallel Computing

- Parallel computing simultaneous use of multiple compute resources to solve a computational problem.
  - To be run using multiple CPUs
  - A problem is broken into discrete parts that can be solved concurrently
  - Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs

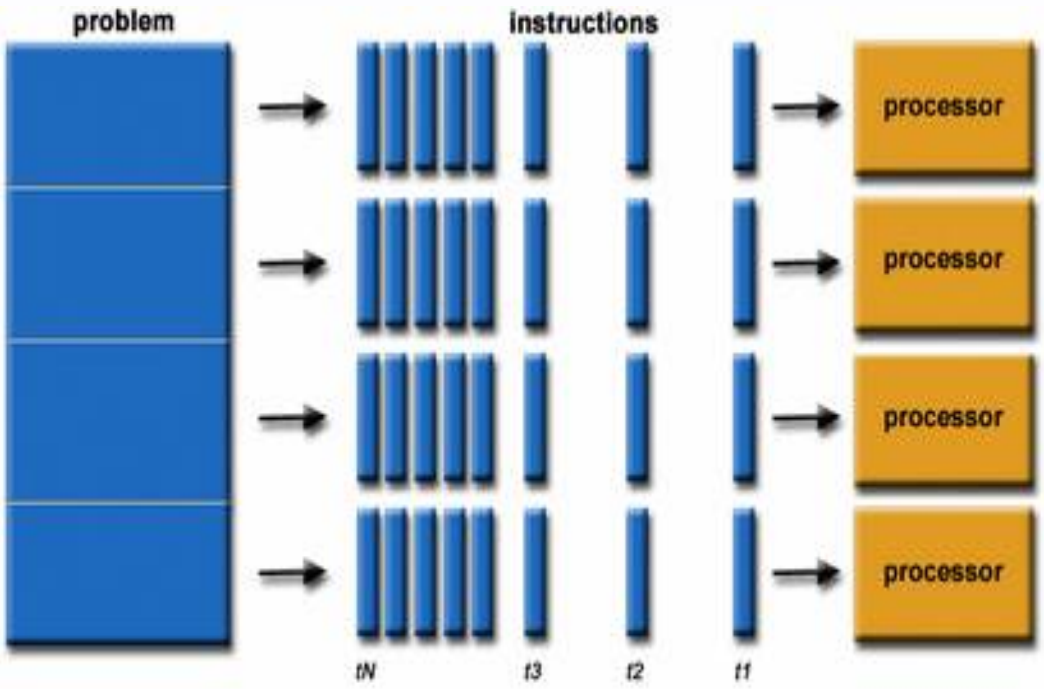


Introduction to Parallel Programming

What is Parallel Programming?: The practice of executing multiple computations simultaneously to improve performance.

Evolution: Shift from single-core to multi-core processors necessitates parallel approaches.

Relevance in 2025: With AI and big data, parallelism is essential for efficiency in cloud, edge, and HPC environments.



Why Think Parallel?

Core Motivation

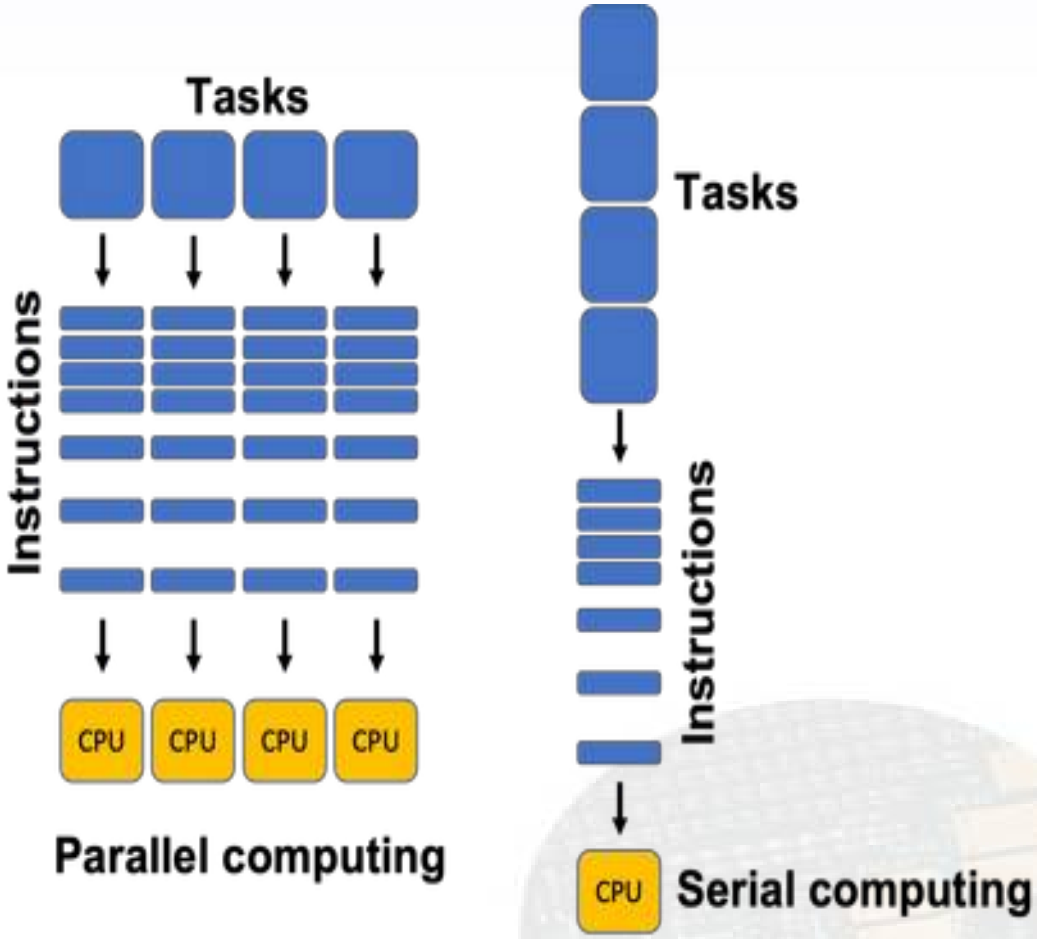
Understand the need for parallelism in today's computing landscape.

Performance Bottlenecks: Sequential code can't exploit multi-core hardware.

Scalability: Handles larger datasets and complex simulations faster.

Real-World Applications: AI training, scientific modeling, real-time data processing.

Challenges Addressed: Overcomes Moore's Law slowdown by leveraging concurrency.



Amdahl's Law in Practice

How much faster a program can get if we use multiple processors/cores.

Sequential parts bottleneck overall performance.

Formula: =S+N(1−S)1

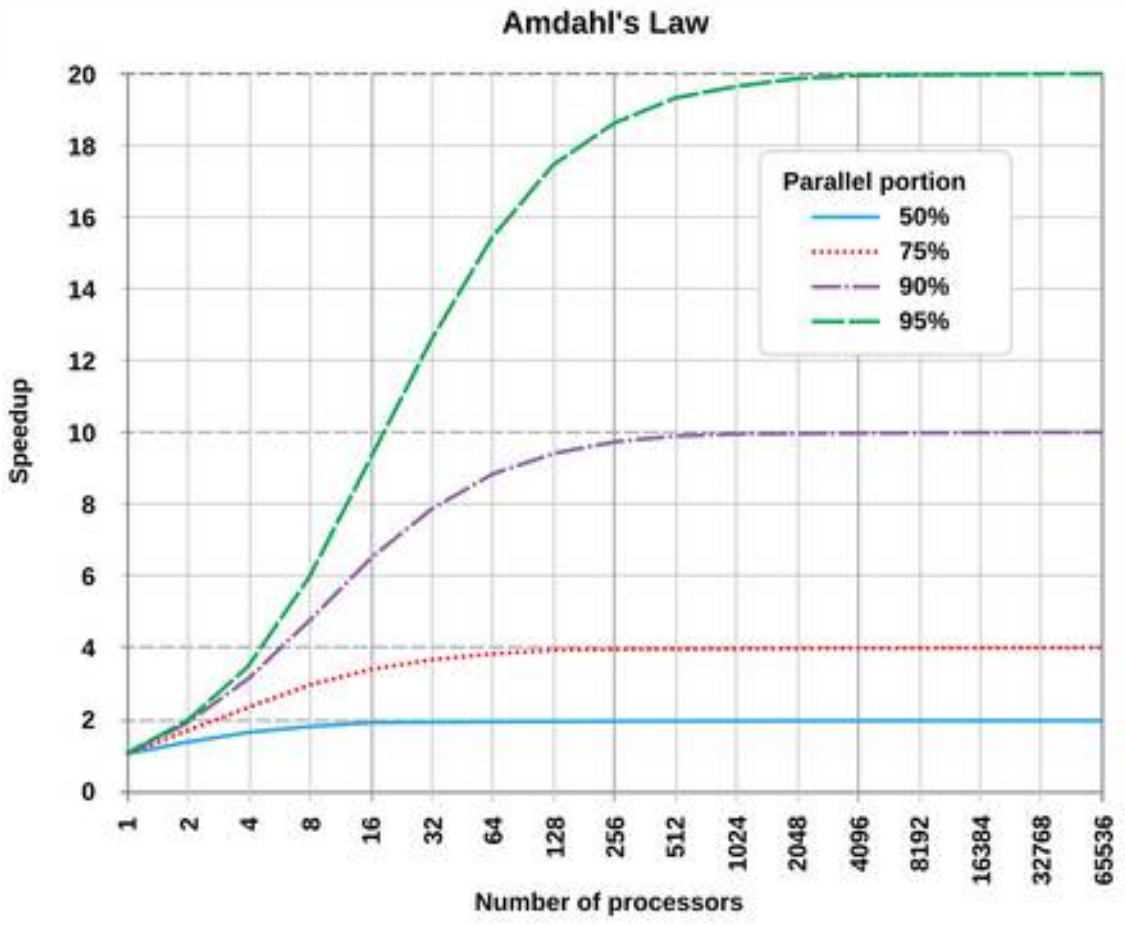
S = fraction of program that is sequential.

(1–S) = fraction that can be parallelized.

N = number of processors.

If 10% is sequential, maximum speedup is 10x, even with infinite processors.

If 1% is sequential, maximum speedup is 100x.



Processing Cores: CPU/GPU

Key Drivers

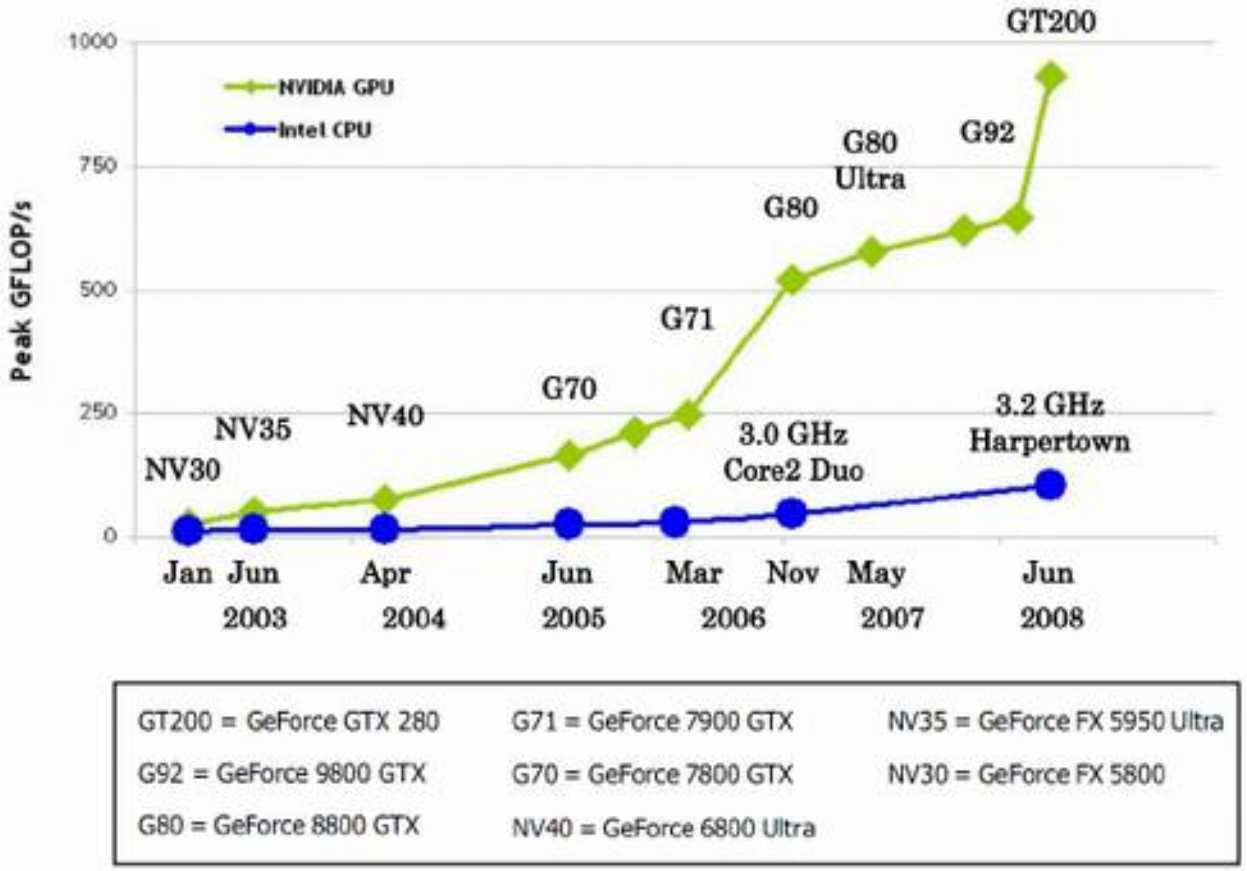
Real CPU/GPU trends: speed, cost, energy

Speed: Single-core frequency stalled; multi-core and GPU parallelism dominate.

Cost: Affordable multi-core chips reduce hardware expenses for high performance.

Energy Efficiency: Parallelism minimizes power usage per computation (e.g., GPUs for AI workloads).

2025 Insights: Quantum-assisted parallelism emerging, but classical multi-core still core.



Memory Hierarchy & NUMA

Why Locality Dominates

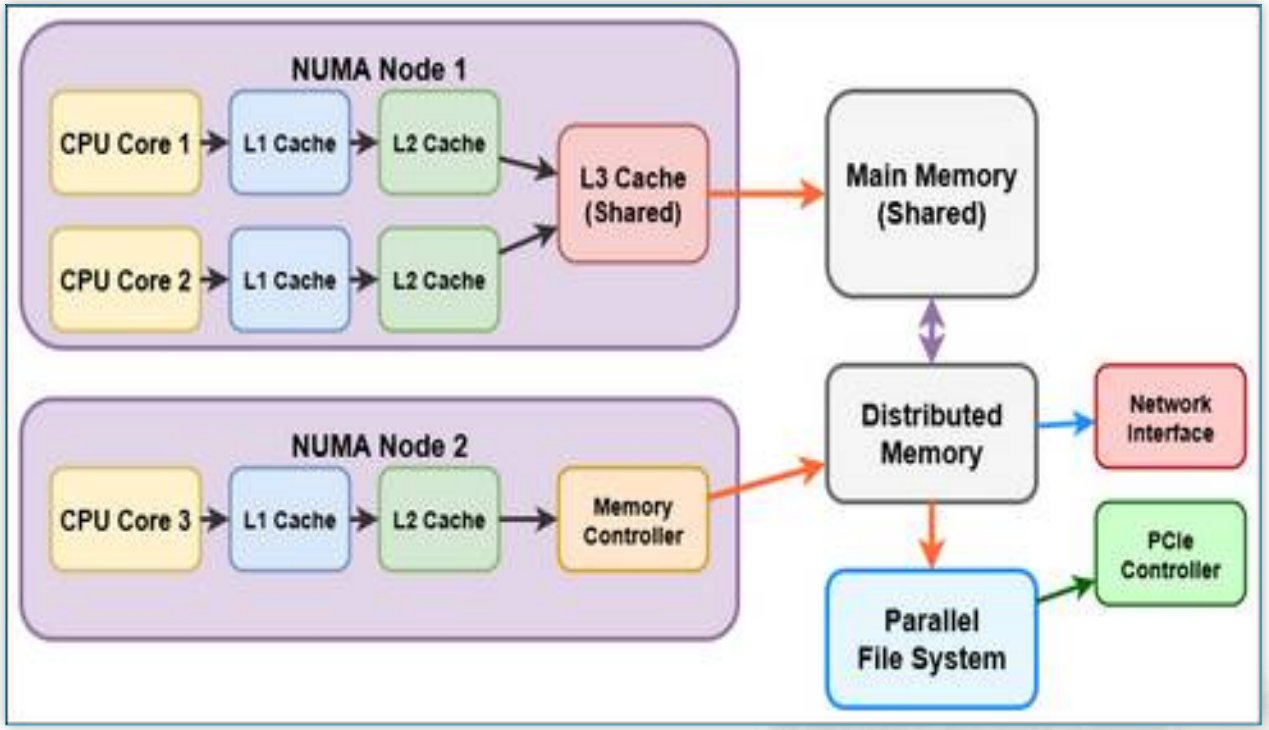
Caches, bandwidth, affinity.

Hierarchy Levels: Registers → L1/L2/L3 Cache → RAM → Storage.

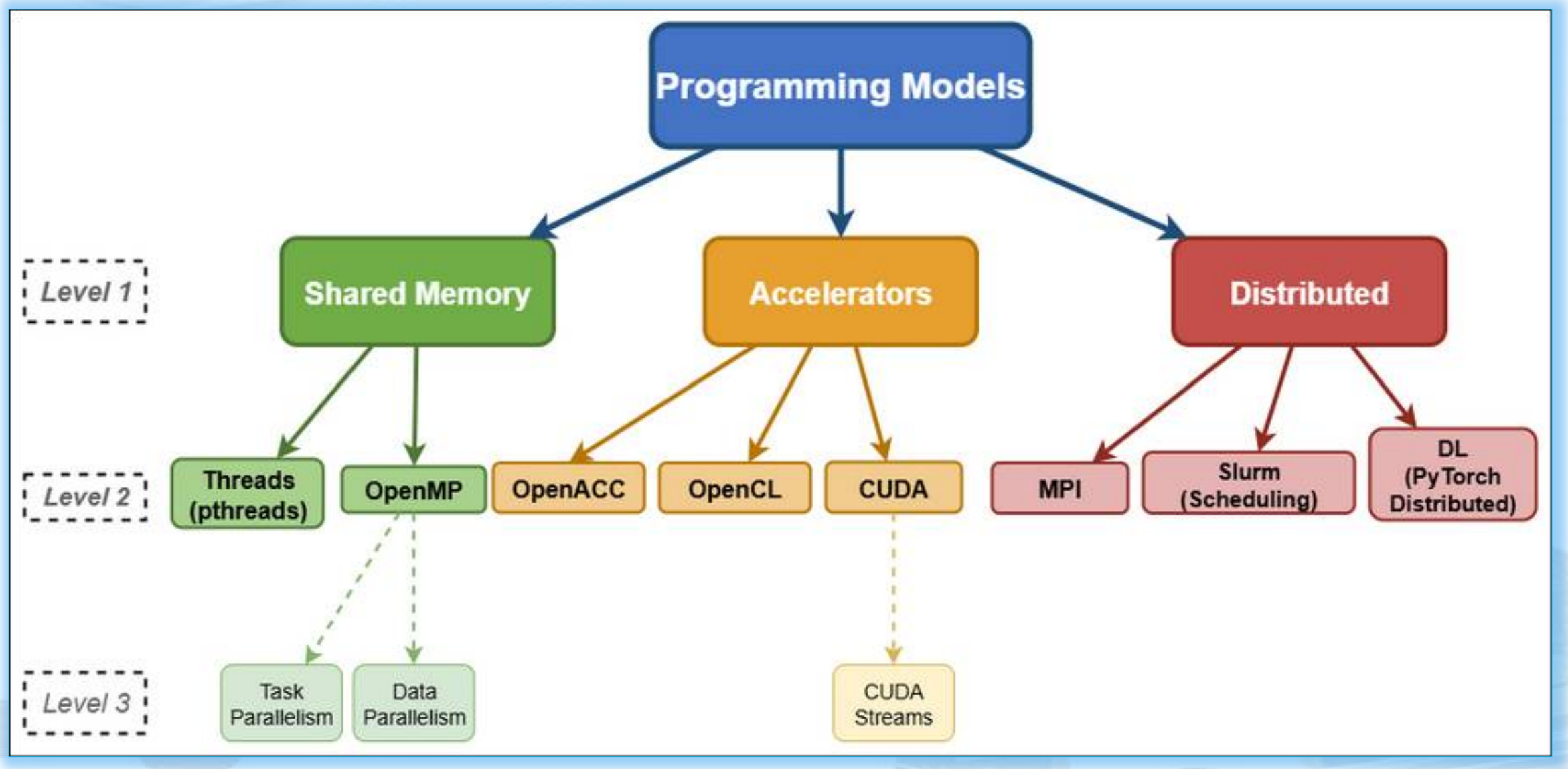
NUMA: Local memory faster than remote.

Importance: Poor locality causes cache misses, increasing latency.

Tips: Use affinity pinning to bind threads to cores for better performance.



Programming Models



Programming Models - Shared Memory

Taxonomy Part 1

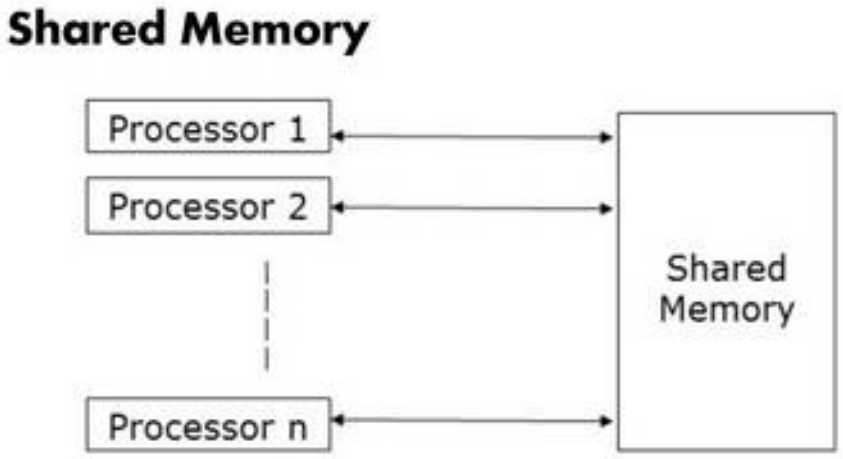
Threads, OpenMP.

Threads: POSIX threads (pthreads) for fine-grained parallelism in shared memory.

OpenMP: Compiler directives for loop parallelization on multi-core CPUs.

Advantages: Easy data sharing, low overhead.

Use Cases: Multi-threaded applications like image processing.



Programming Models – Offloading Accelerators

Taxonomy Part 2

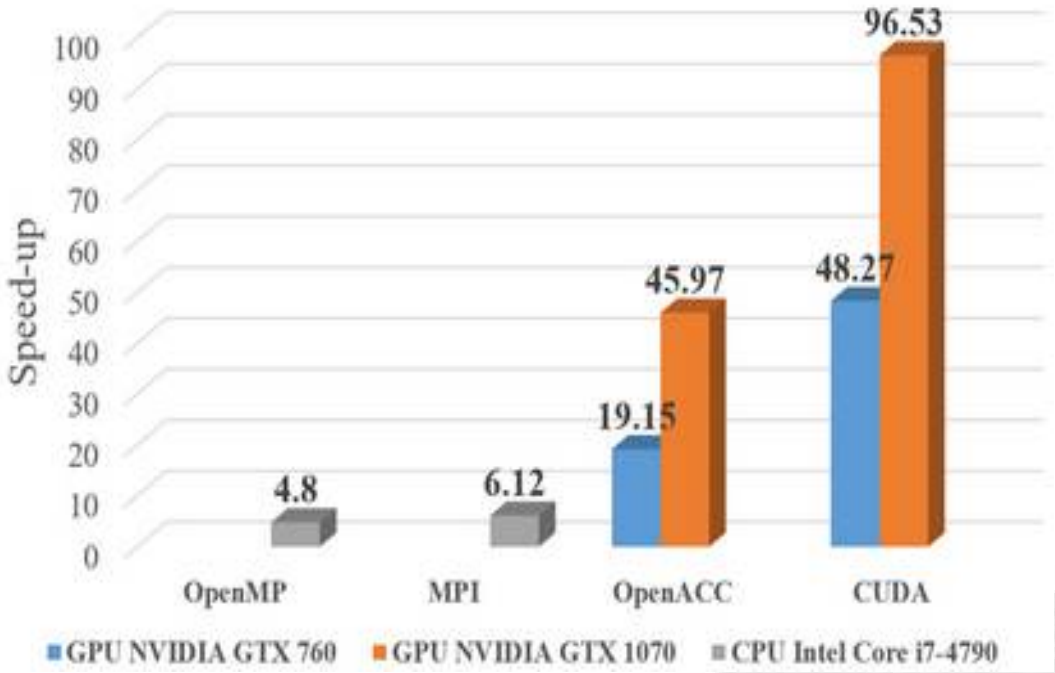
OpenACC/CUDA.

OpenACC: Directive-based for GPUs, similar to OpenMP.

CUDA: NVIDIA-specific for massive parallelism on GPUs.

Benefits: Handles thousands of threads for compute-intensive tasks.

2025 Trends: Integration with AI frameworks like TensorFlow.



Programming Models – Distributed Memory

Taxonomy Part 3

MPI, DL, Slurm.

MPI: Message Passing Interface for cluster computing.

DL: Distributed Learning in frameworks like PyTorch Distributed.

Slurm: Workload manager for job scheduling on HPC clusters.

Advantages: Scales to thousands of nodes.

Understand Program

Program Flow

Processing

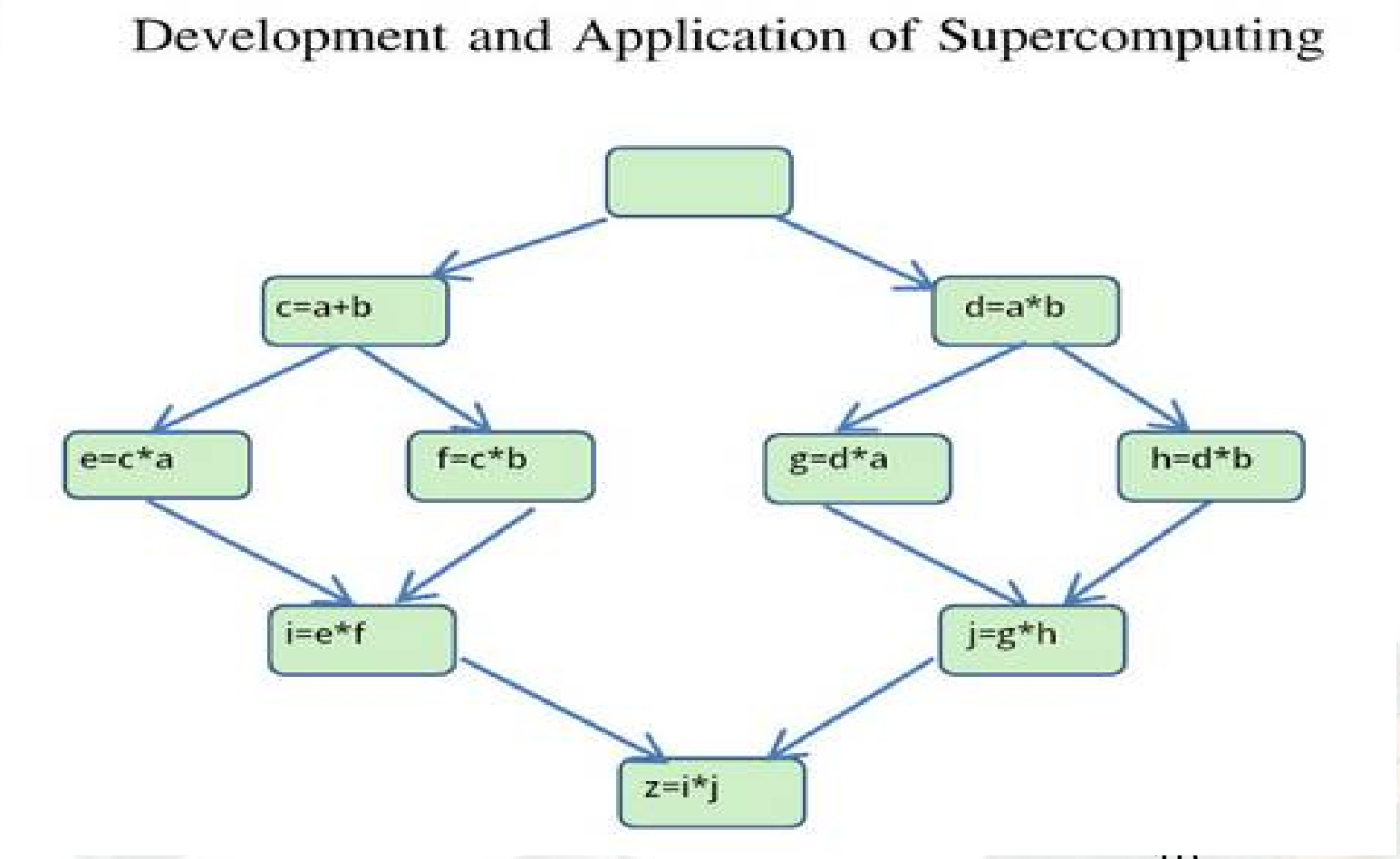
Memory Management

Concurrency and Synchronization

Inter-Process Communication (IPC)

Application Program Architecture

```
#include <stdio.h>

int main() {
    int A, B, C, D, E, F, G, H, I, J, Z;

    A = 10;
    B = 20;
    C = A + B;
    D = A * B;
    E = C * A;
    F = C * B;
    G = D * A;
    H = D * B;
    I = E * F;
    J = G * H;
    Z = I * J;
    return 0;
}
```

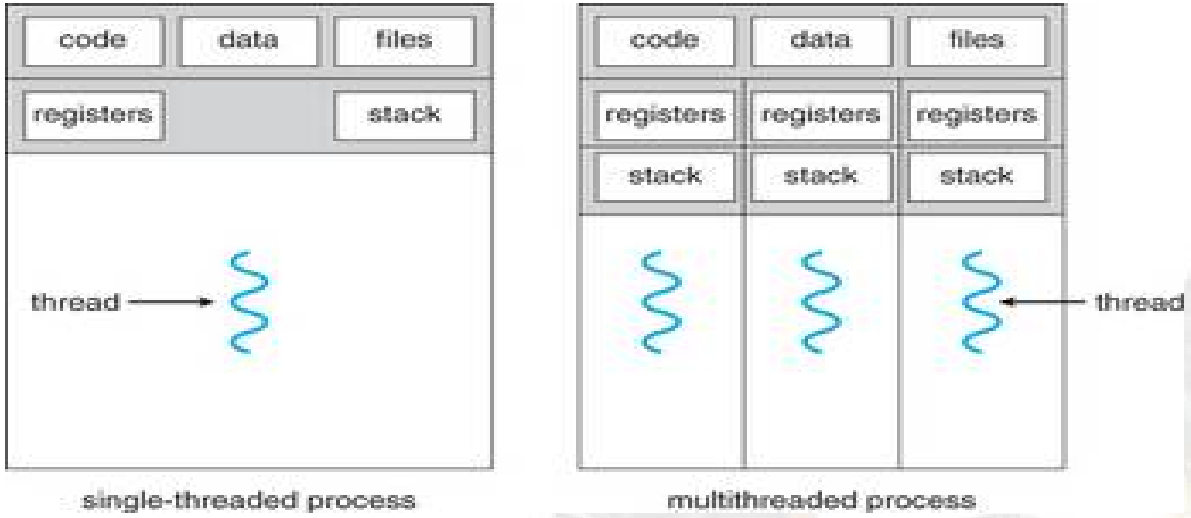


Processing

Process: Start by understanding that a program's execution begins as a process, which represents an instance of the program running on the operating system.

Thread: Threads allow concurrent execution of tasks within the same process.

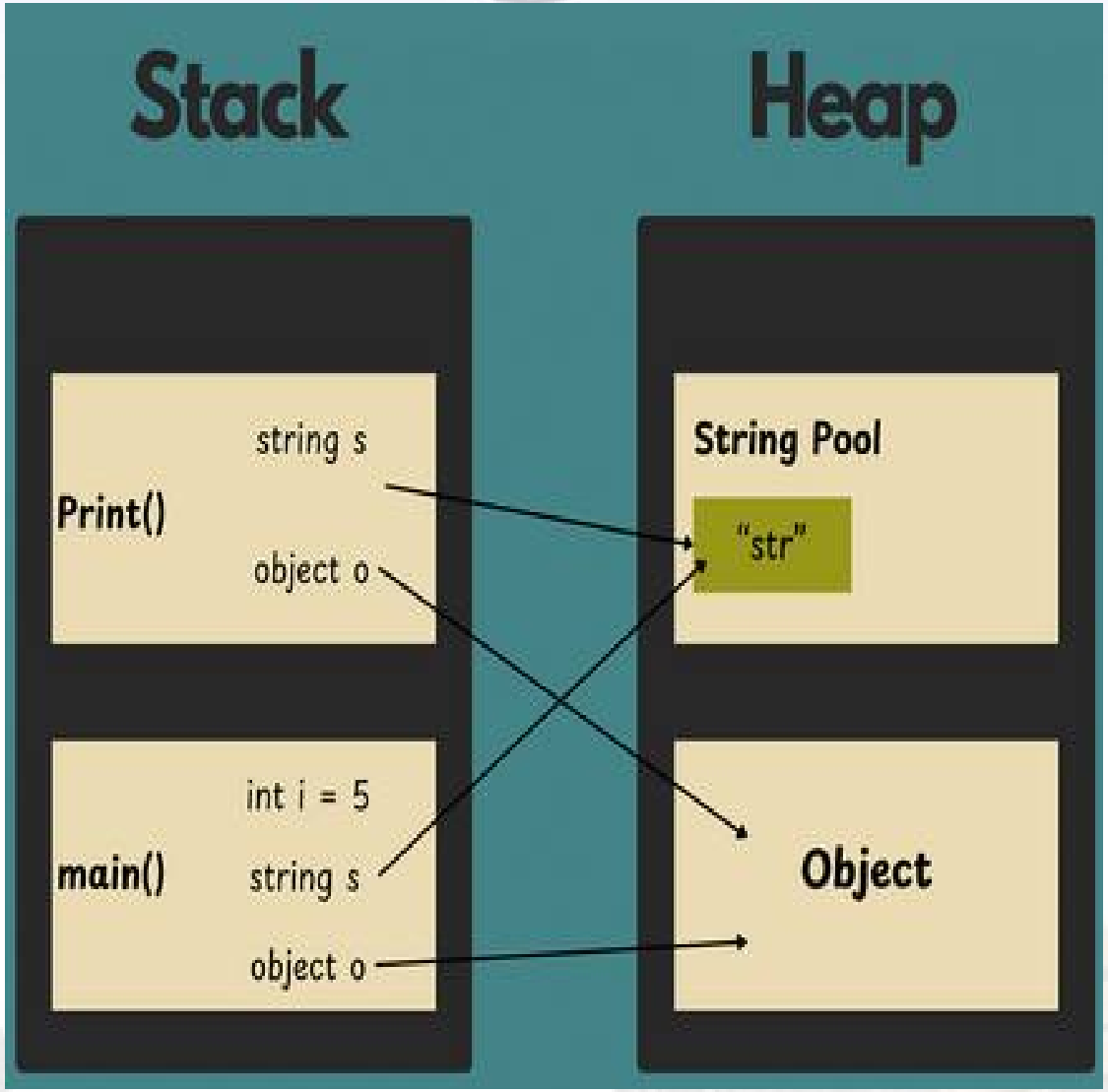
Task: Define specific functions of work or tasks within the program that can be executed independently to take benefit of parallelism effectively.



Memory Management

Heap: Utilize the heap for dynamically allocating memory required by data structures or objects that are shared among threads or tasks.

Stack: Each thread has its own stack for managing local variables, function call information, and return addresses.



Concurrency and Synchronization

Scheduler: Understand how the scheduler manages the execution of threads or tasks on the CPU, considering factors such as priority and time slicing.

Synchronization: Implement synchronization mechanisms (e.g., mutexes, semaphores) to coordinate access to shared resources and ensure data integrity in concurrent execution.

Inter-Process Communication (IPC):

IPC mechanisms (e.g., pipes, shared memory, message queues) for communication and data exchange between processes or threads.

Considerations for Parallel Programs

Understand the Problem and the Program

Data Dependencies

Partitioning (Operations)

Communications (Distribution)

Synchronization

Load Balancing

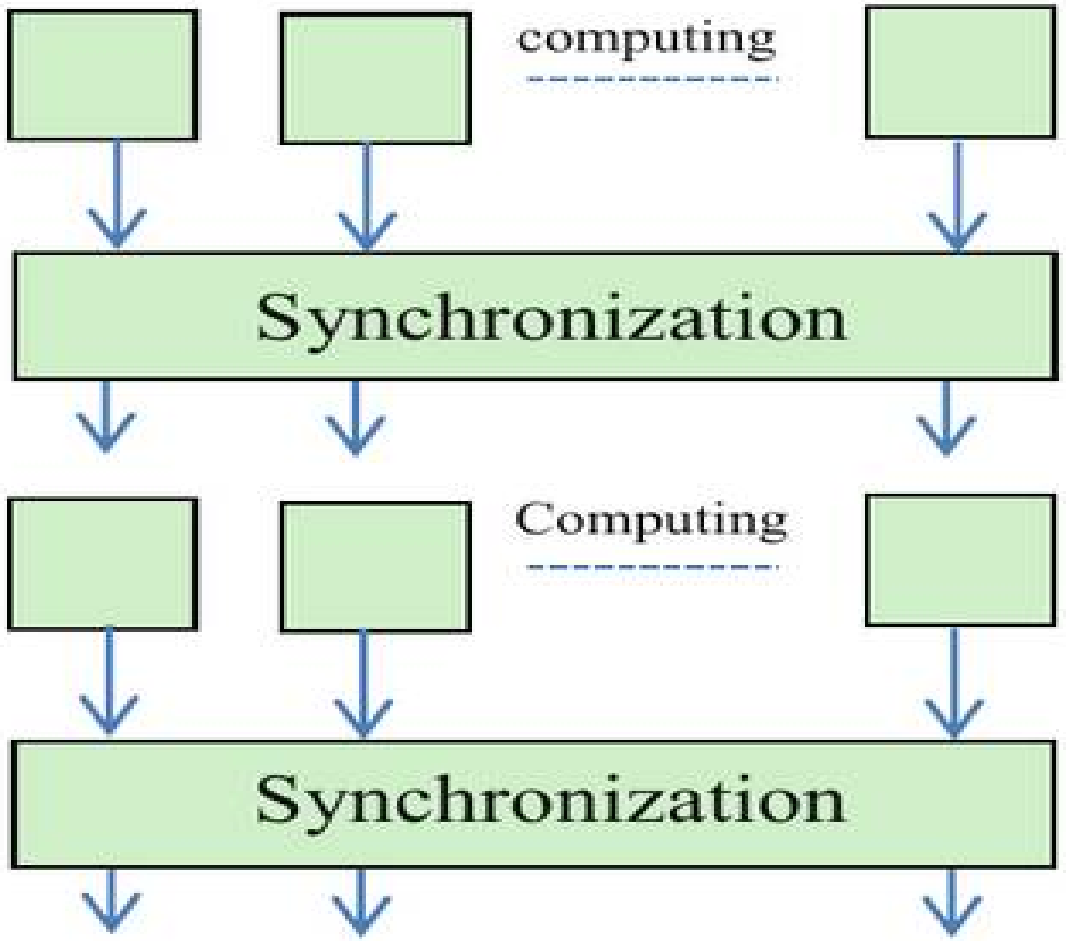
Granularity (Bit, Task, Thread, Nodes)

- **OpenMP** is primarily used for shared-memory parallelism, where multiple threads work together within a single process, accessing shared resources.
- **MPI,** is designed for distributed-memory parallelism, enabling communication and coordination between separate processes running on different nodes of a cluster or supercomputer.

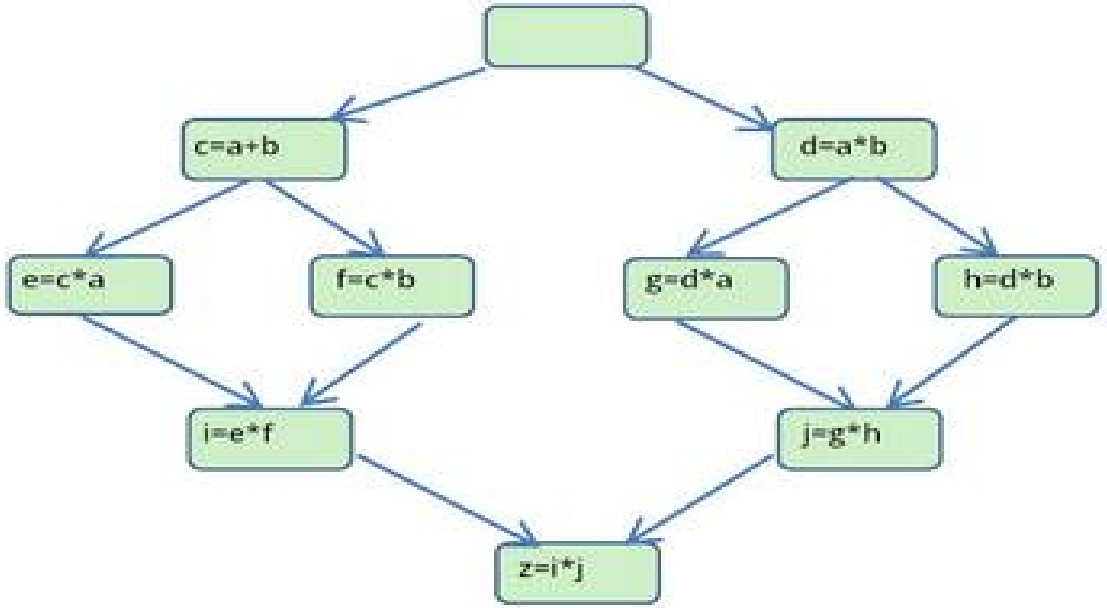
Limits and Costs of Parallel Programming

Automatic vs. Manual Parallelization

Performance Analysis and Tuning



Development and Application of Supercomputing



A 5-Step Recipe Overview

Systematic Approach

Profile — decompose — choose — synchronize — optimize/validate.

Overview: A practical guide to parallelize code effectively.

Benefits: Ensures correctness, efficiency, and scalability.

Application: Applies to any parallel programming task.

5-Step Recipe Details: A practical guide to parallelize code effectively.

Confirms correctness, efficiency, and scalability. Applies to any parallel programming task.

- **Profile**: Analyze sequential code with tools like gprof or VTune.
- **Decompose**: Identify independent tasks (data/task parallelism).
- **Choose Model**: Match to hardware (e.g., OpenMP for shared, MPI for distributed).
- **Synchronize**: Use barriers, locks to avoid races.
- **Optimize/Validate**: Tune, test for speedup and correctness

Challenges and Best Practices

- **Challenges**: Race conditions, load imbalance, overhead.
- **Best Practices**: Start small, use debugging tools, focus on locality.
- **Tools**: Valgrind for races, TAU for profiling.
- **Future-Proofing**: Design for hybrid CPU-GPU systems.

Example: Reverse Time Migration Kernel

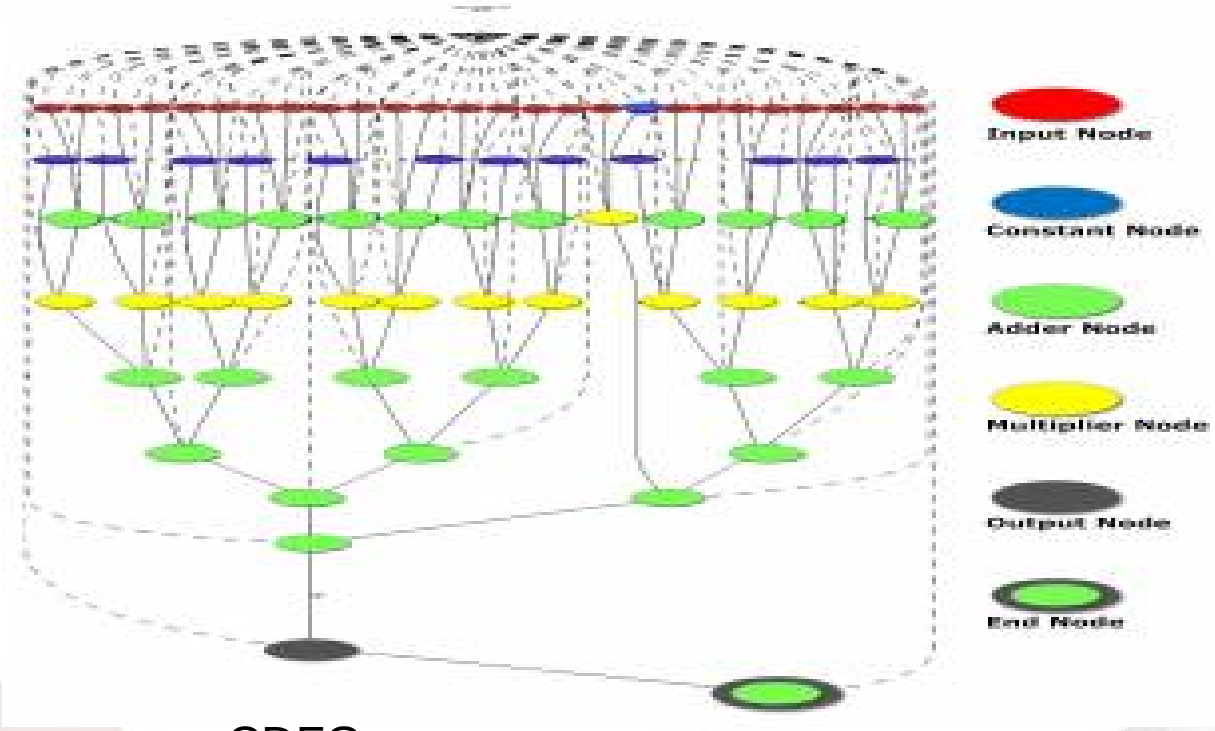
- Sequential application program and converts into parallel program.
- Understand Algorithm/Application data access, data structure, data dependencies and CFG.

$$for\ (y = stencil;\ y < NY - stencil;\ y++)$$
$$for\ (x = stencil;\ x < NX - stencil;\ x++)$$
$$for\ (z = stencil;\ z < NZ - stencil;\ z++)$$
$$P_3(x,y,z) = \sum_{l}^{s} w_l^1\ [P_2(x-l,y,z) + P_2(x+l,y,z)]$$
$$+ \sum_{l}^{s} w_l^2\ [P_2(x,y-l,z) + P_2(x,y+l,z)]$$
$$+ \sum_{l}^{s} w_l^3\ [P_2(x,y,z-l) + P_2(x,y,z+l)] + c^{\circ} P_2(x,y,z))$$
$$+(V(x,y,z) \times dt)^2 + (2 \times P_2(x,y,z)) - P_1(x,y,z)$$

Mathematical Model

RTM: Control and Data Flow Graphs

- ➔ Sequential application program and converts into parallel program.
- ➔ Understand Algorithm/Application data access, data structure, data dependencies and CFG.



CDFG

RTM

- Sequential application program and converts into parallel program.
- Understand Algorithm/Application data access, data structure, data dependencies and CFG.

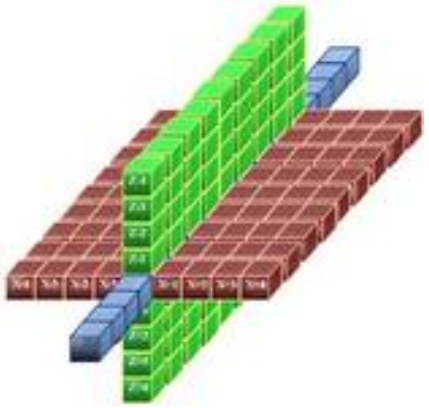
```
#define MX 64
#define MY 64
#define MZ 64
for ( k = Stencil ; k < MY - Stencil ; k++ )
for ( j = Stencil ; j < MZ - Stencil ; j++ )
for ( i = Stencil ; i < MX - Stencil ; i++ )
{
iter = k*(MX*MZ) + (j*MX) + i;
tmp =
Y1*(P2_linear[i+j*iter_j+(k-1)*iter_k] + P2_linear[i+j*iter_j+(k+1)*iter_k]) +
Y2*(P2_linear[i+j*iter_j+(k-2)*iter_k] + P2_linear[i+j*iter_j+(k+2)*iter_k]) +
Y3*(P2_linear[i+j*iter_j+(k-3)*iter_k] + P2_linear[i+j*iter_j+(k+3)*iter_k]) +
Y4*(P2_linear[i+j*iter_j+(k-4)*iter_k] + P2_linear[i+j*iter_j+(k+4)*iter_k]) +
c00 * P2_linear[iter] +
X4*(P2_linear[i+(j-4)*iter_j+k*iter_k] + P2_linear[i+(j+4)*iter_j+k*iter_k]) +
X3*(P2_linear[i+(j-3)*iter_j+k*iter_k] + P2_linear[i+(j+3)*iter_j+k*iter_k]) +
X2*(P2_linear[i+(j-2)*iter_j+k*iter_k] + P2_linear[i+(j+2)*iter_j+k*iter_k]) +
X1*(P2_linear[i+(j-1)*iter_j+k*iter_k] + P2_linear[i+(j+1)*iter_j+k*iter_k]) +
Z4*(P2_linear[(i-4)+j*iter_j+k*iter_k] + P2_linear[(i+4)+j*iter_j+k*iter_k]) +
Z3*(P2_linear[(i-3)+j*iter_j+k*iter_k] + P2_linear[(i+3)+j*iter_j+k*iter_k]) +
Z2*(P2_linear[(i-2)+j*iter_j+k*iter_k] + P2_linear[(i+2)+j*iter_j+k*iter_k]) +
Z1*(P2_linear[(i-1)+j*iter_j+k*iter_k] + P2_linear[(i+1)+j*iter_j+k*iter_k]);
P3_linear[iter] = tmp ;
}
```

C/C++ Program

Programming Example: 3D-Stencil

```
// Stencil Structure
#define Sten_size 4
// 128x128x128 Main Memory Data Set
#define WIDTH 128
#define HEIGHT 128
#define BANK 128
main () {
int X,Y,Z;
X = HEIGHT;
Y = WIDTH*HEIGHT;
Z =0;
float Sten[WIDTH*HEIGHT*BANK];
for ( k = Stencil_size ; k < BANK - Sten_size ; k++ )
  for ( j = Stencil_size ; j < HEIGHT - Sten_size ; j++ )
    for ( i = Stencil_size ; i < WIDTH - Sten_size ; i++ )
{
   Z = k*(WIDTH*HEIGHT) + (j*WIDTH) + i;
   Sten[i+j*X+(k-1)*Y] + Sten[i+j*X+(k+1)*Y] +
   Sten[i+j*X+(k-2)*Y] + Sten[i+j*X+(k+2)*Y] +
   Sten[i+j*X+(k-3)*Y] + Sten[i+j*X+(k+3)*Y] +
   Sten[i+j*X+(k-4)*Y] + Sten[i+j*X+(k+4)*Y] +
   Sten[Z] +
   Sten[i+(j-4)*X+k*Y] + Sten[i+(j+4)*X+k*Y] +
   Sten[i+(j-3)*X+k*Y] + Sten[i+(j+3)*X+k*Y] +
   Sten[i+(j-2)*X+k*Y] + Sten[i+(j+2)*X+k*Y] +
   Sten[i+(j-1)*X+k*Y] + Sten[i+(j+1)*X+k*Y] +
   Sten[(i-4)+j*X+k*Y] + Sten[(i+4)+j*X+k*Y] +
   Sten[(i-3)+j*X+k*Y] + Sten[(i+3)+j*X+k*Y] +
   Sten[(i-2)+j*X+k*Y] + Sten[(i+2)+j*X+k*Y] +
   Sten[(i-1)+j*X+k*Y] + Sten[(i+1)+j*X+k*Y];}
}
```

Conventional 3D stencil access

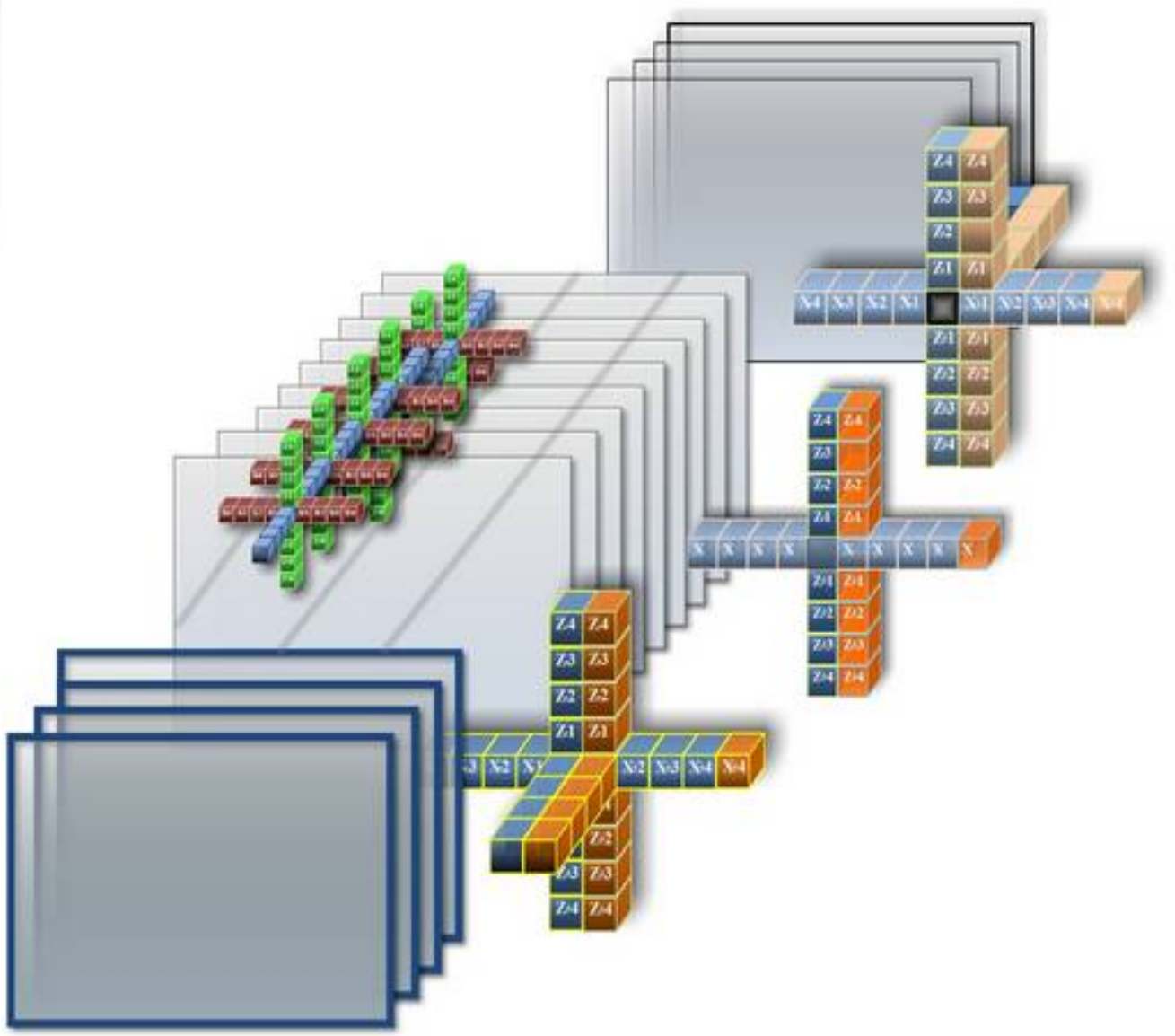


```
#define stencil_size 4
#define PRIORITY1 1
#define PRIORITY2 2
// Main Program
   PMC_SCRATCHPAD STENCIL;
   PMC_SCRATCHPAD SSM_3D;
   MAIN_MEMORY DATASET_3D;
// Part I : Local SSM
// Single Stencil Buffer
   STENCIL.ADDRESS=0X10000000;
   STENCIL.WIDTH=9;
   STENCIL.HEIGHT=3;
   STENCIL.BANK=1;
// 3D 32x32x32 SSM
   SSM_3D.ADDRESS=0X11000000;
   SSM_3D.WIDTH=32;
   SSM_3D.HEIGHT=32;
   SSM_3D.BANK=32;
// Part II : Main Memory
// 3D-Data set
   DATASET_3D.ADDRESS=0X00100000;
   DATASET_3D.WIDTH=128;
   DATASET_3D.HEIGHT=128;
   DATASET_3D.BANK=128;
//PART III : DATA TRANSFER

3D_STENCIL (STENCIL, DATASET_3D, PRIORITY1);

3D_STENCIL_VECTOR (SSM_3D, DATASET_3D, PRIORITY2);
```

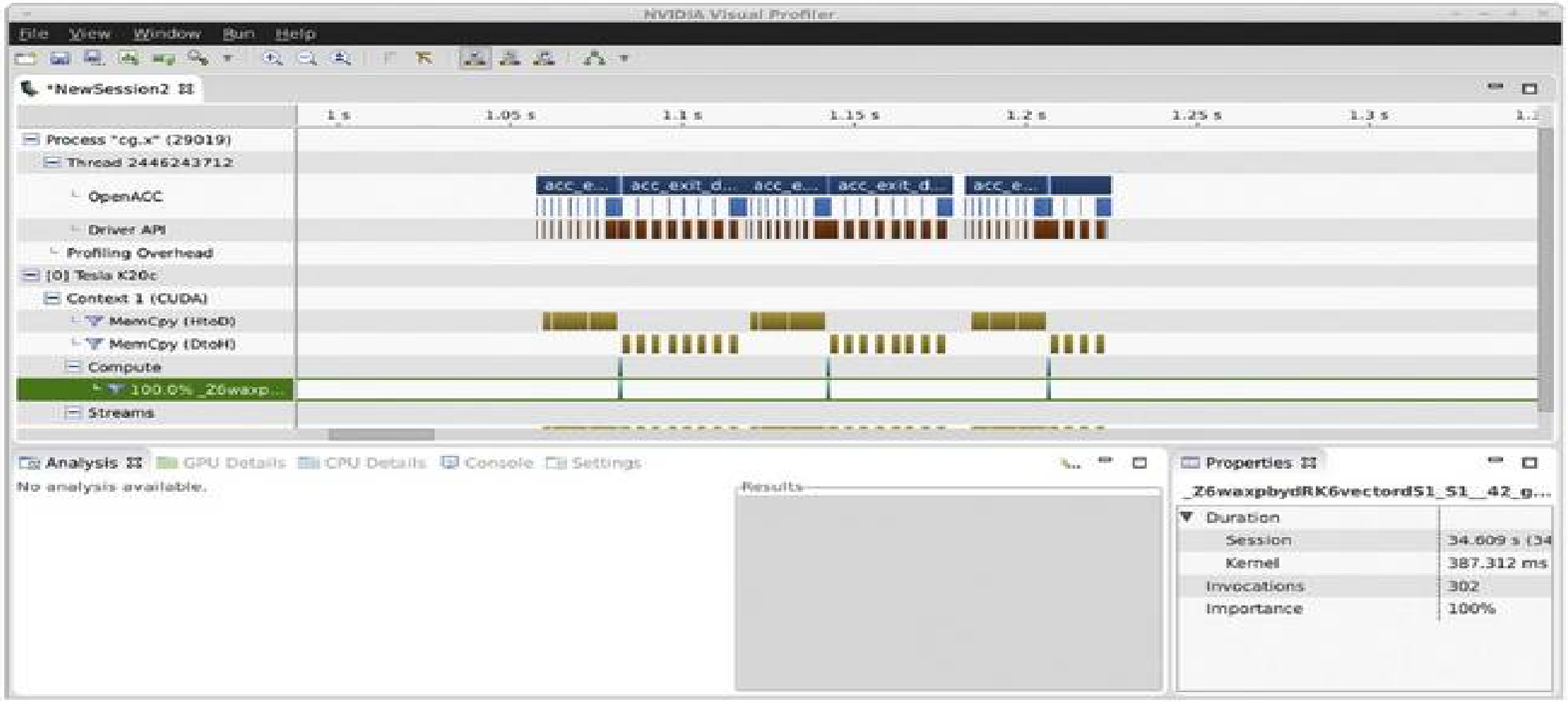
Parallel 3D stencil access

3D Memory Architecture

- Noncontiguous data access to contiguous formate
- Plane size = Nx * Nz
- Number of parallel ports = Ny * 2



Performance Analysis And Tuning



- Introduction to Programming
- Think Parallel Approaches
- **Shared Memory Programming**
- Distributed Memory Programming
- HPC and Cloud

Part - I

(Shared Memory Single Node)

Pthreads

- **Threads**: Lightweight processes sharing memory for parallel execution.
- **Shared Address Space**: All threads access the same memory, requiring synchronization.
- **Locks**: Use mutexes to prevent data races (e.g., pthread_mutex_lock).
- **Use Case**: Ideal for multi-core CPU tasks like concurrent data processing.

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

void* thread_func(void* arg) {
    printf("Thread running\n");
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, thread_func, NULL);
    pthread_join(thread, NULL);
    printf("Main exiting\n");
    return 0;
}
```

Output

```
Thread running
Main exiting
```

Pthreads Demo & Pitfalls

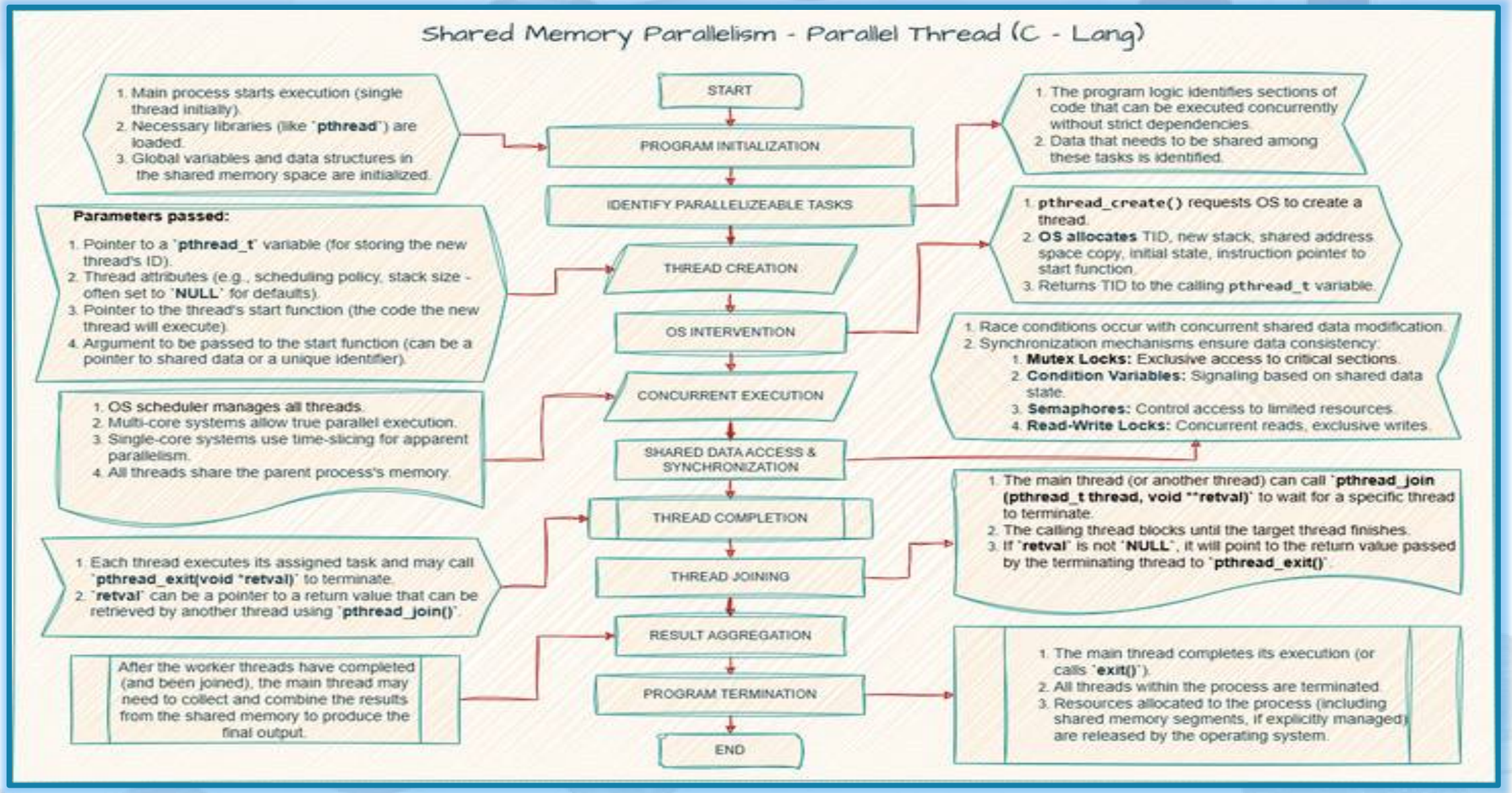
- **Demo**: Counter increment with/without mutex.
- **Pitfalls**: Race conditions, deadlock risks.
- **Speedup**: ~1.67x (250/150) with mutex vs. single-threaded; slight gain without mutex due to race.
- **Correctness**: Mutex ensures 2000000; without mutex, varies (e.g., 1800000-1900000).

Output

```
Single-threaded time: 250.00 ms, Counter: 1000000
With mutex time: 150.00 ms, Counter: 2000000 (Expected: 2000000)
Without mutex time: 140.00 ms, Counter: 1876543 (Incorrect)
```

```
    // With mutex
    counter = 0;
    start = clock();
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);
    pthread_join(t1, NULL); pthread_join(t2, NULL);
    end = clock();
    printf("With mutex time: %.2f ms, Counter: %d (Expected: 2000000)\n",
           1000.0 * (end - start) / CLOCKS_PER_SEC, counter);

    // Without mutex
    counter = 0;
    start = clock();
    pthread_create(&t1, NULL, increment_race, NULL);
    pthread_create(&t2, NULL, increment_race, NULL);
    pthread_join(t1, NULL); pthread_join(t2, NULL);
    end = clock();
    printf("Without mutex time: %.2f ms, Counter: %d (Incorrect)\n",
           1000.0 * (end - start) / CLOCKS_PER_SEC, counter);

    pthread_mutex_destroy(&mutex);
    return 0;
}
```

Pthreads Execution Flow



Python Multiprocessing

- **GIL Limitation**: Global Interpreter Lock (GIL) restricts multi-threading in CPython, limiting CPU-bound performance.
- **Processes**: Multiprocessing bypasses GIL using separate memory spaces, leveraging multiple cores.
- **Queues/Pools**: Enable task distribution and result collection (e.g., Queue, Pool).
- **When it Wins**: Excels in CPU-bound tasks (e.g., numerical computations) over I/O-bound, showing better speedup.

```
def compute_squares(n, q):
    start = time.time()
    result = sum(i * i for i in range(n))
    q.put((result, time.time() - start))

if __name__ == '__main__':
    queue = Queue()
    n = 1000000
    # Single process (simulating GIL-limited threading)
    start = time.time()
    result_single = sum(i * i for i in range(n))
    time_single = time.time() - start
    # Multiprocessing
    p = Process(target=compute_squares, args=(n, queue))
    p.start()
    p.join()
    result_multi, time_multi = queue.get()
    print(f"Single (GIL-limited): Result={result_single:.2e}, Time={time_single:.2f}s")
    print(f"Multi (Process): Result={result_multi:.2e}, Time={time_multi:.2f}s")
```

Output

```
Single (GIL-limited): Result=3.33e+13, Time=0.85s
Multi (Process): Result=3.33e+13, Time=0.25s
```

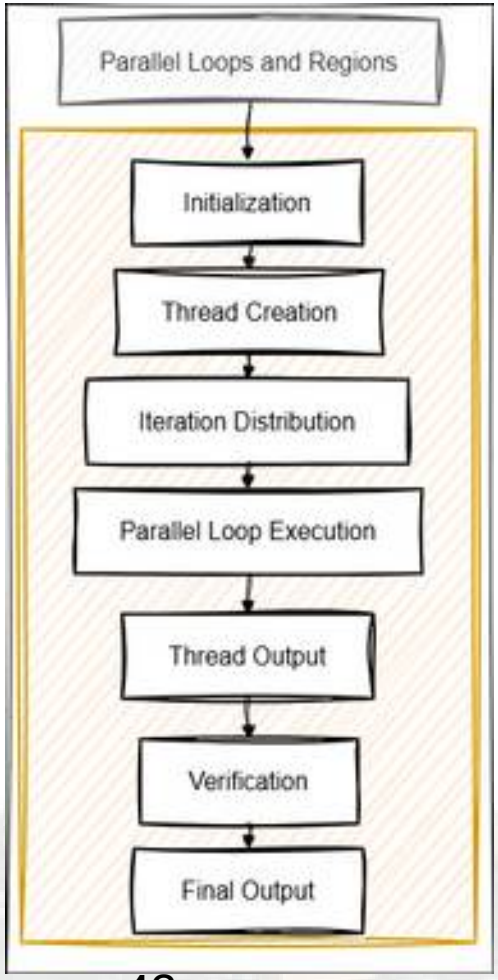
Part – II

OpenMP Essentials

Parallel Regions & Loops

- **Parallel Regions**: #pragma omp parallel creates thread teams.
- **Loops**: #pragma omp parallel for distributes iterations across threads.
- **Schedule**: Options like static for load balancing.
- **Use Case**: Parallel array summation on multi-core CPUs.

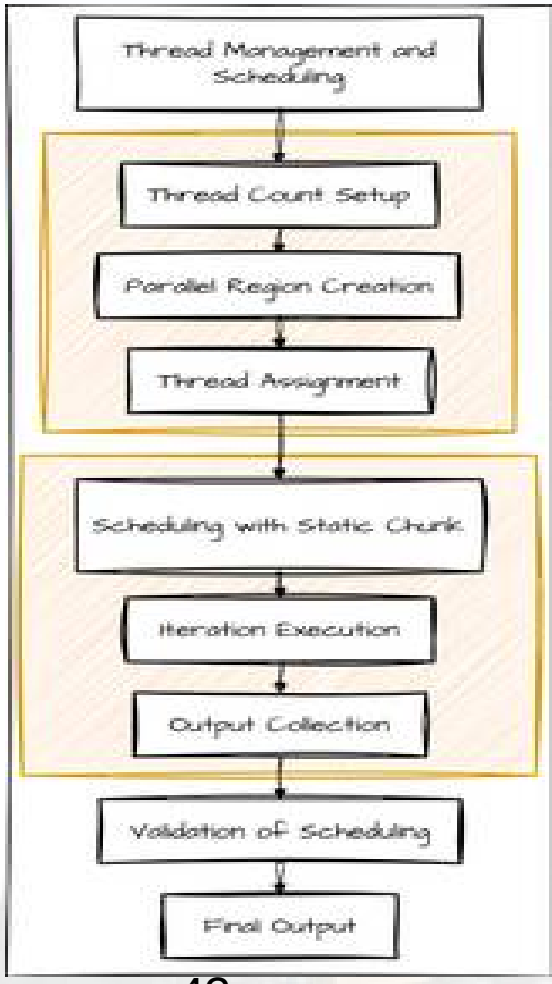
```c
#include <stdio.h>
#include <omp.h>

int main() {
    int i;
#pragma omp parallel for
    for (i = 0; i < 10; i++) {
        printf("Thread %d processing iteration %d\n", omp_get_thread_num(), i);
    }
    return 0;
}
```



Threads & Scheduling

- **Scheduling Types**: static (even split), dynamic (adaptive), guided (decreasing chunks).
- **Chunk Sizes**: Affect load balance and overhead.
- **Affinity**: Bind threads to cores for cache efficiency.
- **Use Case**: Parallel loop with varying schedules.

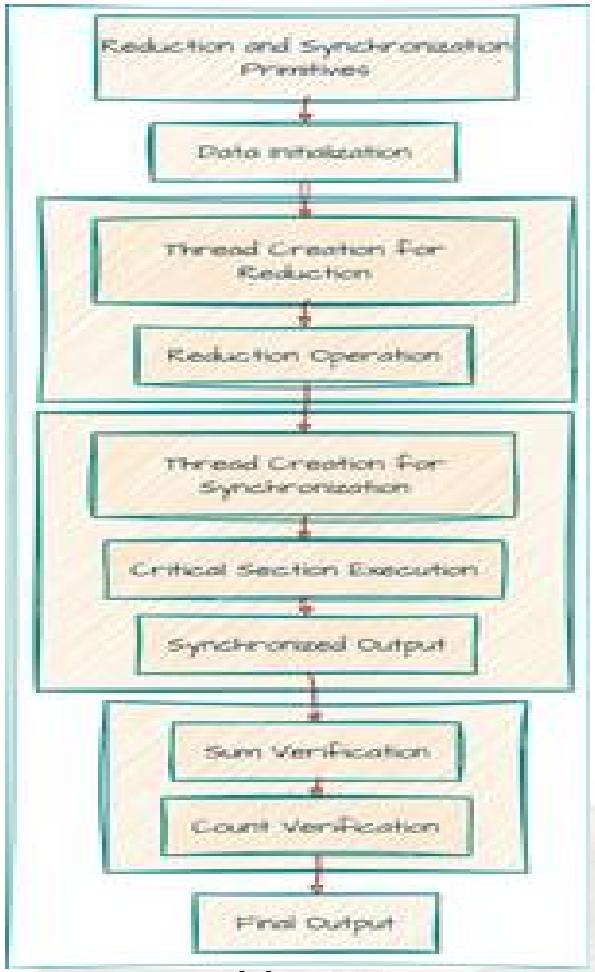
```c
#include <stdio.h>
#include <omp.h>

int main() {
    omp_set_num_threads(4);  // Set the number of threads

#pragma omp parallel
    {
        int tid = omp_get_thread_num();

#pragma omp for schedule(static, 2)
        for (int i = 0; i < 8; i++) {
            printf("Thread %d handling iteration %d\n", tid, i);
        }
    }
    return 0;
}
```



Reductions & Sync

- **Reduction**: Combines thread results (e.g., sum) safely.
- **Synchronization**: critical, atomic, barrier prevent race conditions.
- **Use Case**: Parallel sum with sync to ensure correctness.
- **Benefit**: Reduction avoids manual sync overhead.

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

int main() {
    int sum = 0;

#pragma omp parallel for reduction(+:sum)
    for (int i = 1; i <= 10; i++) {
        sum += i;  // Safe reduction
    }

    printf("Sum = %d\n", sum);

    int count = 0;
#pragma omp parallel
    {
#pragma omp critical
        {
            count++;
        }
    }

    printf("Critical section count = %d\n", count);
    return 0;
}
```

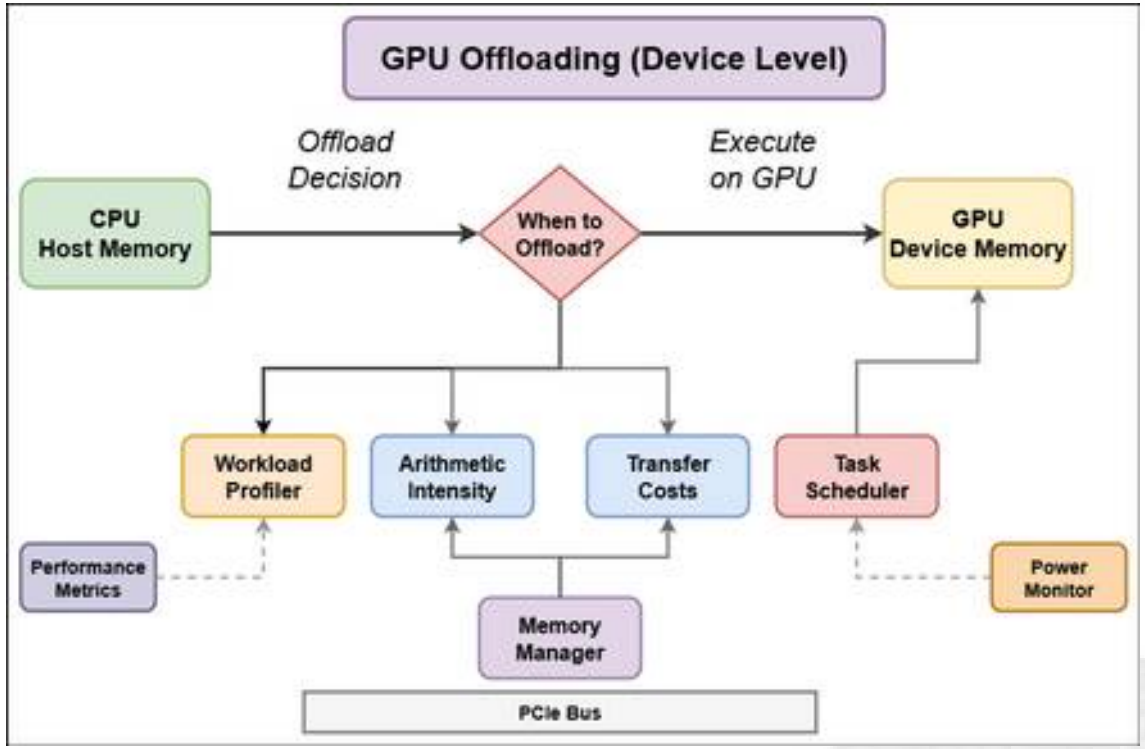


Part - III

GPU Offloading (Device Level)

When to Offload?

- **Arithmetic Intensity**: Ratio of computations to memory access; high intensity favors GPU offload.
- **Transfer Costs**: Data movement (H2D/D2H) can negate GPU benefits if low intensity.
- **Decision Point**: Offload when compute exceeds transfer overhead.
- **Use Case**: Evaluate offload for matrix multiply.



CUDA Kernel Anatomy

- **Grid/Block/Thread**: Hierarchical structure for GPU execution.
- **Memory Spaces**: Global for data, shared for fast access, registers per-thread.
- **Use Case**: Vector addition with thread indexing.

```
#include <stdio.h>
#include <cuda_runtime.h>

__global__ void addKernel(float *c, float *a, float *b, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) c[idx] = a[idx] + b[idx];
}
```

- **Kernel**: Adds vectors a and b into c using thread idx.
- **Grid/Block**: Maps threads across blocks (e.g., 256/block).

```
int main() {
    int n = 1000; float *a, *b, *c, *d_a, *d_b, *d_c;
    a = (float*)malloc(n * sizeof(float)); b = (float*)malloc(n * sizeof(float)); c = (float*)malloc(n * si
zeof(float));
    for(int i = 0; i < n; i++) { a[i] = 1.0f; b[i] = 2.0f; }
    cudaMalloc(&d_a, n * sizeof(float)); cudaMalloc(&d_b, n * sizeof(float)); cudaMalloc(&d_c, n * sizeof(f
loat));
    cudaMemcpy(d_a, a, n * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, n * sizeof(float), cudaMemcpyHostToDevice);
    addKernel<<<(n+255)/256, 256>>>(d_c, d_a, d_b, n);
    cudaMemcpy(c, d_c, n * sizeof(float), cudaMemcpyDeviceToHost);
    printf("c[0] = %f\n", c[0]);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c); free(a); free(b); free(c);
}
```

- **Memory**: Host allocates a, b, c; device uses global memory (d_a, d_b, d_c).
- **Launch**: 4 blocks of 256 threads cover 1000 elements.
- **Result**: Copies back and verifies addition.

CUDA Memory & Perf

- **Memory Types**: Global (large/slow), shared (fast/small), registers (per-thread).
- **Coalescing**: Aligns memory access for bandwidth efficiency.
- **Occupancy**: Maximizes thread usage per SM.
- **Use Case**: Optimize matrix multiply with memory types.

```
#include <stdio.h>
#include <cuda_runtime.h>

__global__ void multiplyKernel(float *a, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        a[idx] *= 2.0f;
    }
}

int main() {
    int n = 1000;
    float *a, *d_a;

    // Allocate host memory
    a = (float*)malloc(n * sizeof(float));

    // Initialize array
    for (int i = 0; i < n; i++) {
        a[i] = (float)i;
    }

    // Allocate device memory
    cudaMalloc(&d_a, n * sizeof(float));

    // Copy data to device
    cudaMemcpy(d_a, a, n * sizeof(float), cudaMemcpyHostToDevice);

    // Launch kernel with optimized grid/block sizes
    int threadsPerBlock = 256;
    int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;
    multiplyKernel<<<blocksPerGrid, threadsPerBlock>>>(d_a, n);

    // Copy result back to host
    cudaMemcpy(a, d_a, n * sizeof(float), cudaMemcpyDeviceToHost);

    // Verify result
    printf("a[1] = %f\n", a[1]); // Should be 2.0

    // Free memory
    cudaFree(d_a);
    free(a);
    return 0;
}
```

CUDA Streams & Overlap

- **Streams**: Independent queues for overlapping tasks.
- **Overlap**: Hides data transfer latency with computation.
- **Concurrency**: Multiple streams enable pipelined execution.
- **Use Case**: Overlap data transfer with kernel execution.
- **Output:**

```
c[0] = 3.000000, Overlap achieved
```

- **Overlap**: H2D and compute/D2H run concurrently, reducing total time.
- **Concurrency**: Two streams enable pipelined execution, confirmed by correct result (3.0).

```
#include <cuda.h>
#include <stdio.h>

__global__ void addKernel(float *c, float *a, float *b) {
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}

int main() {
    float a[1000], b[1000], c[1000];
    float *d_a, *d_b, *d_c;
    cudaStream_t stream1, stream2;
    cudaStreamCreate(&stream1); cudaStreamCreate(&stream2);
    cudaMalloc(&d_a, 1000 * sizeof(float));
    cudaMalloc(&d_b, 1000 * sizeof(float));
    cudaMalloc(&d_c, 1000 * sizeof(float));
    for (int i = 0; i < 1000; i++) { a[i] = 1.0f; b[i] = 2.0f; }
    cudaMemcpyAsync(d_a, a, 1000 * sizeof(float), cudaMemcpyHostToDevice, stream1);
    cudaMemcpyAsync(d_b, b, 1000 * sizeof(float), cudaMemcpyHostToDevice, stream2);
    addKernel<<<1, 1000, 0, stream1>>>(d_c, d_a, d_b);
    cudaMemcpyAsync(c, d_c, 1000 * sizeof(float), cudaMemcpyDeviceToHost, stream1);
    cudaStreamSynchronize(stream1);
    printf("c[0] = %f, Overlap achieved\n", c[0]);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Open-ACC Directives

- **Directives**: #pragma acc parallel loop enables GPU offload with minimal changes.
- **Minimal Changes**: Annotate existing loops without major refactoring.
- **Loop Offload**: Distributes work to GPU threads.
- **Use Case**: Offload a simple array operation.

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

int main() {
    int n = 1000;
    float a[n], b[n], c[n];

    // Initialize arrays
    for (int i = 0; i < n; i++) {
        a[i] = 1.0f;
        b[i] = 2.0f;
    }

    // Parallelize vector addition on GPU
    #pragma acc parallel loop
    for (int i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }

    // Verify result
    printf("c[0] = %f\n", c[0]); // Should be 3.0
    return 0;
}
```

Open-ACC Tasks/Loops

- **Pipelines**: Overlap data transfer with computation for efficiency.
- **Kernels vs Parallel**: Kernels offer fine control, parallel simplifies loop offload.
- **Use Case**: Pipeline a sequence of array operations.

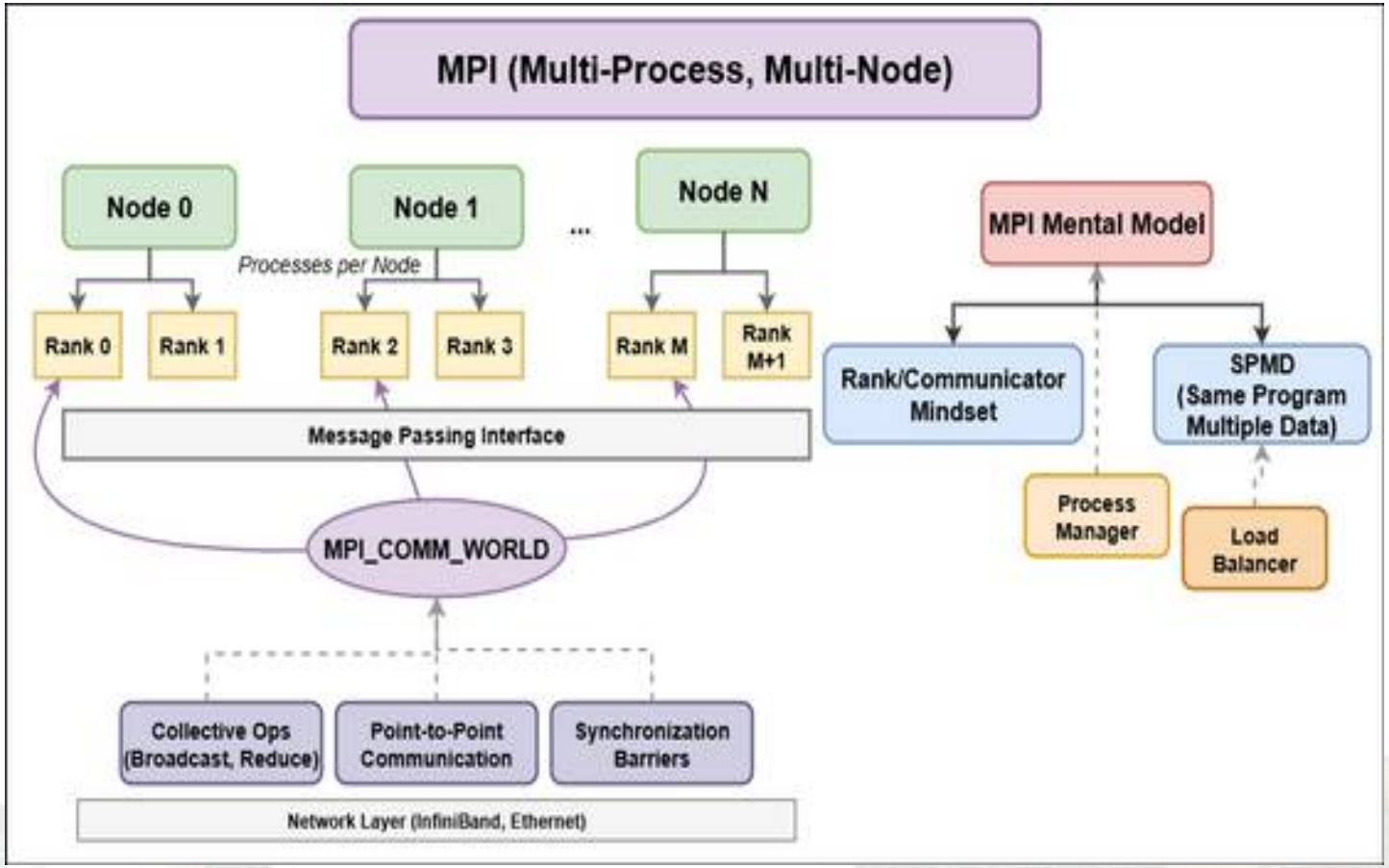
```c
#include <stdio.h>
#include <openacc.h>

int main() {
    int n = 1000;
    float a[n];

    // Initialize array
    for (int i = 0; i < n; i++) {
        a[i] = (float)i;
    }

    // Offload computation to GPU
    #pragma acc data copy(a)
    {
        #pragma acc parallel loop
        for (int i = 0; i < n; i++) {
            a[i] = a[i] * 2.0f;
        }
    }

    // Verify result
    printf("a[1] = %f\n", a[1]); // Should be 2.0
    return 0;
}
```

Part - IV

MPI (Multi-Process, Multi-Node)

MPI Mental Model

- **Ranks**: Unique IDs for processes in a communicator.
- **Communicators**: Groups for message passing (e.g., MPI_COMM_WORLD).
- **SPMD**: Single Program Multiple Data execution model.
- **Use Case**: Distributed computing across nodes.



Point-to-Point

- **Send/Recv**: MPI_Send and MPI_Recv for direct communication.
- **Tags**: Label messages for selective receive.
- **Matching Pitfalls**: Mismatched tags/buffers cause deadlocks.
- **Use Case**: Data exchange between two ranks.

```c
#include <mpi.h>
#include <stdio.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) printf("Need at least 2 processes\n");
        MPI_Finalize();
        return 1;
    }

    int data;
    if (rank == 0) {
        data = 42;
        MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Rank %d sent %d\n", rank, data);
    } else if (rank == 1) {
        MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Rank %d received %d\n", rank, data);
    }

    MPI_Finalize();
    return 0;
}
```

Broadcast/Scatter/Gather

- **Broadcast**: MPI_Bcast sends data from one to all.
- **Scatter**: MPI_Scatter distributes chunks to ranks.
- **Gather**: MPI_Gather collects data to one rank.
- **Use Case**: Data distribution in parallel computation.

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

    int data[4] = {0};
    int local_data;

    if (rank == 0) {
        for (int i = 0; i < 4; i++) data[i] = i + 1;
    }

    // Broadcast
    MPI_Bcast(data, 4, MPI_INT, 0, MPI_COMM_WORLD);
    printf("Rank %d after bcast: data[0] = %d\n", rank, data[0]);

    // Scatter
    MPI_Scatter(data, 1, MPI_INT, &local_data, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf("Rank %d after scatter: local_data = %d\n", rank, local_data);

    // Gather
    int gather_data[4];
    MPI_Gather(&local_data, 1, MPI_INT, gather_data, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        printf("Rank 0 gathered: ");
        for (int i = 0; i < 4; i++) printf("%d ", gather_data[i]);
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

Collectives & Sync

- **Allreduce**: MPI_Allreduce combines and distributes results.
- **Barrier**: MPI_Barrier synchronizes processes.
- **Costs/Benefits**: Latency cost for sync; benefit in consistent data.
- **Use Case**: Global sum with synchronization.

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

    int local_value = rank + 1;
    int sum;

    // Reduce to sum all values
    MPI_Reduce(&local_value, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Sum of all ranks: %d\n", sum);
    }

    // Synchronize processes
    MPI_Barrier(MPI_COMM_WORLD);
    printf("Rank %d passed barrier\n", rank);

    MPI_Finalize();
    return 0;
}
```

Topology & Scaling

- **Cartesian Topology**: MPI_Cart_create defines process grid.
- **Topology Mapping**: Optimizes communication patterns.
- **Strong/Weak Scaling**: Fixed/growing problem size with more processes.
- **Use Case**: 2D grid for scaled computation.

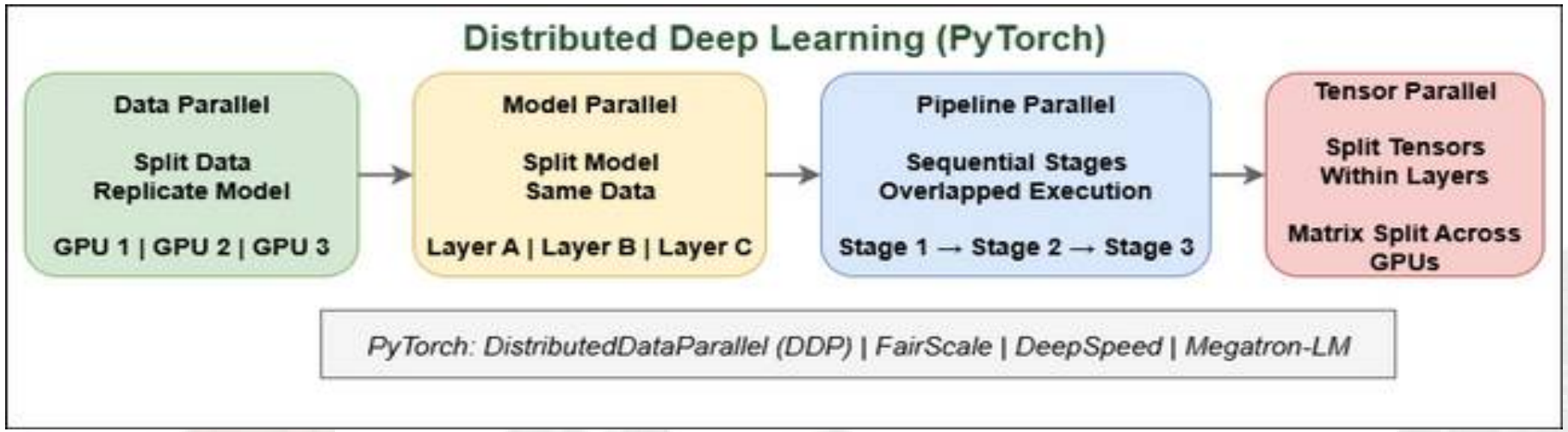
```c
#include <mpi.h>
#include <stdio.h>
#include <math.h>

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);

    // Ensure enough processes for a meaningful grid
    if (size < 4) {
        if (rank == 0) printf("Need at least 4 processes for a 2x2 grid\n");
        MPI_Finalize();
        return 1;
    }

    // Create a 2D Cartesian topology
    int dims[2] = {0, 0}; // Let MPI decide dimensions
    int periods[2] = {0, 0}; // Non-periodic boundaries
    MPI_Comm cart_comm;
    MPI_Dims_create(size, 2, dims); // Automatically compute grid dimensions
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &cart_comm);

    // Get process coordinates in the grid
    int coords[2];
    MPI_Cart_coords(cart_comm, rank, 2, coords);
    printf("Rank %d mapped to coordinates (%d, %d) in %dx%d grid\n",
           rank, coords[0], coords[1], dims[0], dims[1]);

    // Get neighbors (up, down, left, right)
    int up, down, left, right;
    MPI_Cart_shift(cart_comm, 0, 1, &up, &down); // Shift along first dimension (rows)
    MPI_Cart_shift(cart_comm, 1, 1, &left, &right); // Shift along second dimension (columns)
    printf("Rank %d neighbors: up=%d, down=%d, left=%d, right=%d\n",
           rank, up, down, left, right);

    // Scalable computation: Each process contributes its rank to a global sum
    int local_value = rank;
    int global_sum;
    MPI_Reduce(&local_value, &global_sum, 1, MPI_INT, MPI_SUM, 0, cart_comm);
    if (rank == 0) {
        printf("Global sum of ranks = %d\n", global_sum);
    }

    MPI_Comm_free(&cart_comm);
    MPI_Finalize();
    return 0;
}
```

Part-V

Distributed Deep Learning (PyTorch)

DL Parallelism Landscape

- **Data Parallel**: Replicates model, splits data (e.g., mini-batches).
- **Model Parallel**: Splits model layers across devices.
- **Pipeline Parallel**: Stages model execution.
- **Tensor Parallel**: Splits tensor computations.



Streams & Task Parallel (Single Node)

- **Streams**: Overlap data preparation with computation.
- **Task Parallel**: Concurrent prep and compute tasks.
- **Non-blocking Ops**: Improves throughput with async execution.
- **Use Case**: Stream data loading in PyTorch.

```python
import torch

# Simple example using CUDA streams
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
stream1 = torch.cuda.Stream()
stream2 = torch.cuda.Stream()

# Create tensors
a = torch.randn(1000, 1000, device=device)
b = torch.randn(1000, 1000, device=device)
c = torch.zeros(1000, 1000, device=device)

# Perform matrix multiplications in parallel streams
with torch.cuda.stream(stream1):
    c += torch.matmul(a, b)

with torch.cuda.stream(stream2):
    c += torch.matmul(b, a)

# Synchronize
torch.cuda.synchronize()
print(c[0, 0].item())  # Output result
```

Data-Parallel vs DDP

- **Data-Parallel**: Single-process, limited scalability.
- **DDP**: Multi-process, better scaling with NCCL.
- **Buckets**: Optimizes gradient aggregation.
- **Use Case**: Distributed training comparison.

```python
import torch
import torch.nn as nn
import torch.distributed as dist
import torch.multiprocessing as mp
from torch.nn.parallel import DistributedDataParallel as DDP

def setup(rank, world_size):
    dist.init_process_group("nccl", rank=rank, world_size=world_size)

def train(rank, world_size):
    setup(rank, world_size)
    model = nn.Linear(10, 10).cuda(rank)
    model = DDP(model, device_ids=[rank])
    input = torch.randn(20, 10).cuda(rank)
    output = model(input)
    print(f"Rank {rank} output shape: {output.shape}")
    dist.destroy_process_group()

if __name__ == "__main__":
    world_size = torch.cuda.device_count()
    mp.spawn(train, args=(world_size,), nprocs=world_size, join=True)
```

Model Parallelism

- **Layer Split**: Distributes layers across devices.
- **Tensor Split**: Parallelizes tensor operations.
- **Pipeline Bubbles**: Idle time between stages.
- **Use Case**: Split transformer layers.

```python
import torch
import torch.nn as nn

class ModelParallel(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Linear(10, 10).cuda(0)
        self.layer2 = nn.Linear(10, 10).cuda(1)

    def forward(self, x):
        x = self.layer1(x.cuda(0))
        x = self.layer2(x.cuda(1))
        return x

model = ModelParallel()
input = torch.randn(20, 10)
output = model(input)
print(output.shape)
```

Sync vs Async Training

- **Synchronous**: Consistent gradients, lower staleness, higher sync cost.
- **Asynchronous**: Higher throughput, potential staleness.
- **When to Use**: Sync for small clusters, async for large-scale.
- **Use Case**: Compare gradient update strategies.

```python
import torch
import torch.nn as nn
import torch.distributed as dist
import torch.multiprocessing as mp

def setup(rank, world_size):
    dist.init_process_group("nccl", rank=rank, world_size=world_size)

def train(rank, world_size):
    setup(rank, world_size)
    model = nn.Linear(10, 10).cuda(rank)
    model = torch.nn.parallel.DistributedDataParallel(model, device_ids=[rank])
    optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
    input = torch.randn(20, 10).cuda(rank)
    target = torch.randn(20, 10).cuda(rank)
    output = model(input)
    loss = nn.MSELoss()(output, target)
    loss.backward()
    optimizer.step()  # Synchronous update
    print(f"Rank {rank} loss: {loss.item()}")
    dist.destroy_process_group()

if __name__ == "__main__":
    world_size = torch.cuda.device_count()
    mp.spawn(train, args=(world_size,), nprocs=world_size, join=True)
```

Multi-Node Setup

- **Init Methods**: File or TCP for process coordination.
- **NCCL/Gloo**: Backends for GPU/CPU communication.
- **Rendezvous**: Ensures multi-node startup sync.
- **Use Case**: Multi-node PyTorch training.

```python
import torch
import torch.nn as nn
import torch.distributed as dist
import os

def setup(rank, world_size):
    os.environ['MASTER_ADDR'] = 'localhost'
    os.environ['MASTER_PORT'] = '12355'
    dist.init_process_group("nccl", rank=rank, world_size=world_size)

def train(rank, world_size):
    setup(rank, world_size)
    model = nn.Linear(10, 10).cuda(rank)
    model = torch.nn.parallel.DistributedDataParallel(model, device_ids=[rank])
    input = torch.randn(20, 10).cuda(rank)
    output = model(input)
    print(f"Rank {rank} output shape: {output.shape}")
    dist.destroy_process_group()

if __name__ == "__main__":
    world_size = 2  # Example: 2 processes
    torch.multiprocessing.spawn(train, args=(world_size,), nprocs=world_size, join=True)
```

Sharding & Grad Sync

- **Sharding**: Splits model parameters for memory efficiency.
- **Grad Sync**: Overlaps communication with computation.
- **Use Case**: Shard large model training.
- **Sharding**: DDP shards model across 2 ranks, reducing memory use.
- **Overlap**: Comm-compute overlap during gradient sync improves efficiency.

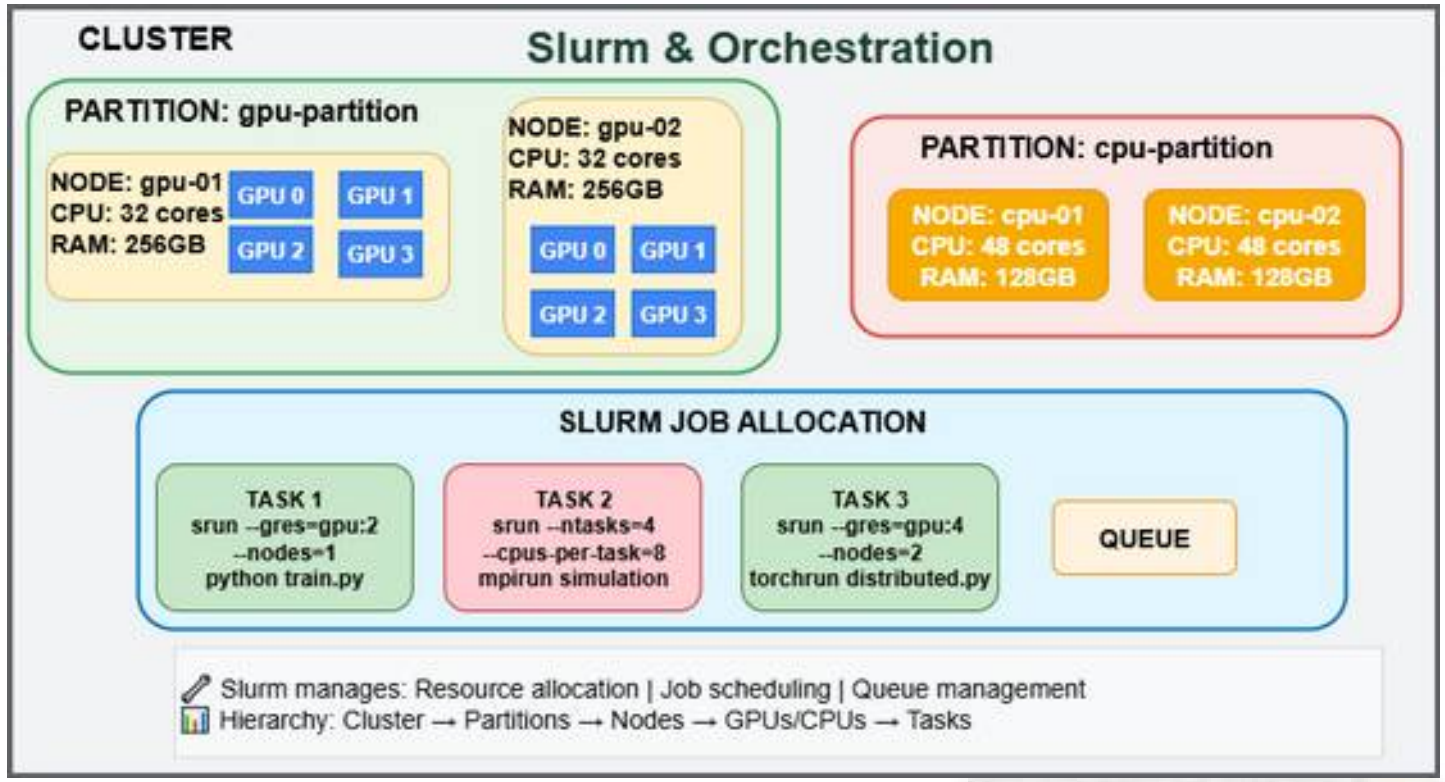
```python
import torch
import torch.nn as nn
import torch.distributed as dist

def setup(rank, world_size):
    dist.init_process_group("nccl", rank=rank, world_size=world_size)

def train(rank, world_size):
    setup(rank, world_size)
    # Simple model sharding: each rank handles a portion of the model
    model = nn.Linear(10, 10).cuda(rank)
    model = torch.nn.parallel.DistributedDataParallel(model, device_ids=[rank])
    optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
    input = torch.randn(20, 10).cuda(rank)
    target = torch.randn(20, 10).cuda(rank)
    output = model(input)
    loss = nn.MSELoss()(output, target)
    loss.backward()
    optimizer.step()  # Gradients synchronized via DDP
    print(f"Rank {rank} loss: {loss.item()}")
    dist.destroy_process_group()

if __name__ == "__main__":
    world_size = 2
    torch.multiprocessing.spawn(train, args=(world_size,), nprocs=world_size, join=True)
```

Part - VI

Slurm & Orchestration

Slurm Mental Model

- **Partitions**: Logical node groupings (e.g., gpu, cpu).
- **Nodes**: Compute resources in the cluster.
- **Tasks**: Jobs or processes to execute.
- **GPUs**: Requested via gres for GPU tasks.



Cluster-wide Shell Fan-out

- **srun**: Launches tasks across nodes.
- **pdsh**: Parallel shell for command execution.
- **Patterns**: Fan-out for setup or monitoring.
- **Use Case**: Run health checks cluster-wide.

```
#!/bin/bash
#SBATCH --job-name=shell_commands
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=1
#SBATCH --time=00:05:00

srun hostname
srun echo "Hello from $SLURM_PROCID"
```

Requesting Resources

- **CPUs**: Specify with --ntasks or --cpus-per-task.
- **GPUs**: Use --gres=gpu:X.
- **Memory**: Request with --mem.
- **Constraints**: Match node features (e.g., GPU type).

```
#!/bin/bash
#SBATCH --job-name=gpu_test
#SBATCH --output=%j.out
#SBATCH --error=%j.err
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --gres=gpu:1
#SBATCH --partition=compute    # Explicit partition setting

# Run your job commands here

echo "=== Hostname ==="
hostname

echo "=== CPU Info ==="
lscpu | grep "Model name"

echo "=== GPU Info ==="
nvidia-smi

echo "=== GPU UUID ==="
nvidia-smi --query-gpu=uuid --format=csv
```

Multi-node GPU Jobs

- **Nodes**: Specify with --nodes.
- **gres**: Request GPUs per node.
- **Exclusive**: Allocates nodes fully.
- **Timeouts**: Limits job duration with --time.

```
#!/bin/bash
#SBATCH --job-name=gpu_test
#SBATCH --output=%j.out
#SBATCH --error=%j.err
#SBATCH --nodes=6                    # Set the number of GPU nodes you want
#SBATCH --ntasks=6                   # Each Task on A node
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=4
#SBATCH --gres=gpu:1
#SBATCH --distribution=block

# Single srun will launch in parallel on all allocated nodes
srun bash -c '
    echo "=== Running on $(hostname) ==="
    nvidia-smi
    nvidia-smi --query-gpu=uuid --format=csv
```

sbatch for PyTorch

- **Env**: Sets up Python and dependencies.
- **srun vs torchrun**: srun for simple, torchrun for distributed.
- **Arrays**: Runs multiple experiments.
- **Use Case**: Train a model with sbatch.

```
#!/bin/bash
#SBATCH --job-name=pytorch_train
#SBATCH --nodes=1
#SBATCH --gpus=1
#SBATCH --time=00:10:00
#SBATCH --mem=8GB

module load python
module load pytorch

python train.py
```

```
import torch
import torch.nn as nn
import torch.optim as optim

# Define a simple model
model = nn.Sequential(
    nn.Linear(10, 5),
    nn.ReLU(),
    nn.Linear(5, 2)
).cuda()

# Loss and optimizer
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

inputs = torch.randn(32, 10).cuda()
targets = torch.randn(32, 2).cuda()

# Training loop
for epoch in range(5):
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    loss.backward()
    optimizer.step()
    print(f"Epoch {epoch+1}, Loss: {loss.item():.4f}")
```

Job Distribution Knobs

- **--ntasks**: Number of tasks to launch.
- **--cpus-per-task**: CPUs per task.
- **Partitions**: Select specific cluster partitions.
- **Use Case**: Distribute workload across nodes.

```
#!/bin/bash
#SBATCH --job-name=job_dist
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=2
#SBATCH --time=00:05:00
#SBATCH --partition=gpu
#SBATCH --mem=4GB

srun python -c "import torch; print(torch.cuda.device_count())"
```

Distributed Launch

- **torchrun**: Launches multi-node PyTorch jobs.
- **Rendezvous**: Coordinates process startup.
- **Env Vars**: Configures NCCL or Gloo settings.
- **Use Case**: Distributed training setup.

```
#!/bin/bash
#SBATCH --job-name=dist_pytorch
#SBATCH --nodes=2
#SBATCH --gpus-per-node=1
#SBATCH --ntasks-per-node=1
#SBATCH --time=00:10:00

module load pytorch

srun python distributed_train.py
```

```
import torch
import torch.distributed as dist

def setup(rank, world_size):
    dist.init_process_group("gloo", rank=rank, world_size=world_size)

def main():
    rank = int(os.environ["SLURM_PROCID"])
    world_size = int(os.environ["SLURM_NTASKS"])
    setup(rank, world_size)
    print(f"Rank {rank} initialized")
    dist.destroy_process_group()

if __name__ == "__main__":
    main()
```

Ports & Networking

- **NCCL Envs**: Configures network performance.
- **OOB**: Out-of-band communication setup.
- **Firewall/Ports**: Opens ports (e.g., 29500-29600).
- **Use Case**: Multi-node GPU networking.

```
#!/bin/bash
#SBATCH --job-name=port_mgmt
#SBATCH --nodes=2
#SBATCH --gpus-per-node=1
#SBATCH --ntasks-per-node=1
#SBATCH --time=00:10:00

MASTER_ADDR=$(scontrol show hostnames $SLURM_JOB_NODELIST | head -n 1)
MASTER_PORT=29500

srun python distributed_train.py --master-addr $MASTER_ADDR --master-port $MASTER_PORT
```

```
import torch
import torch.distributed as dist
import argparse

def setup(rank, world_size, master_addr, master_port):
    dist.init_process_group("nccl", rank=rank, world_size=world_size,
                            init_method=f"tcp://{master_addr}:{master_port}")

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--master-addr", type=str)
    parser.add_argument("--master-port", type=str)
    args = parser.parse_args()

    rank = int(os.environ["SLURM_PROCID"])
    world_size = int(os.environ["SLURM_NTASKS"])
    setup(rank, world_size, args.master_addr, args.master_port)
    print(f"Rank {rank} connected")
    dist.destroy_process_group()

if __name__ == "__main__":
    main()
```

Engineering & Wrap-up

Profiling & Roofline

- **Profiling**: Identifies CPU/GPU/MPI time bottlenecks (e.g., NVIDIA Nsight, MPI profiling).
- **Roofline**: Models performance ceiling based on arithmetic intensity.
- **Use Case**: Profile a matrix multiply kernel.

```
#include <cuda.h>
#include <stdio.h>

__global__ void matMul(float *c, float *a, float *b, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) c[i] = a[i] * b[i];
}

int main() {
    int n = 1000; float *a, *b, *c, *d_a, *d_b, *d_c;
    cudaMalloc(&d_a, n * sizeof(float)); cudaMalloc(&d_b, n * sizeof(float)); cudaMalloc(&d_c, n * sizeof(f
loat));
    matMul<<<(n+255)/256, 256>>>(d_a, d_b, d_c, n);
    printf("Kernel executed\n");
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Correctness & Debugging

- **Data Races**: Concurrent access issues (e.g., OpenMP critical).
- **Deadlocks**: Blocking waits (e.g., MPI mismatch).
- **Tools**: Valgrind, GDB, MPI debuggers for determinism.
- **Use Case**: Check a parallel sum.

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

int main() {
    int sum = 0;
    #pragma omp parallel for reduction(+:sum)
    for(int i = 0; i < 100; i++) sum += i;
    printf("Sum: %d\n", sum);
    return 0;
}
```

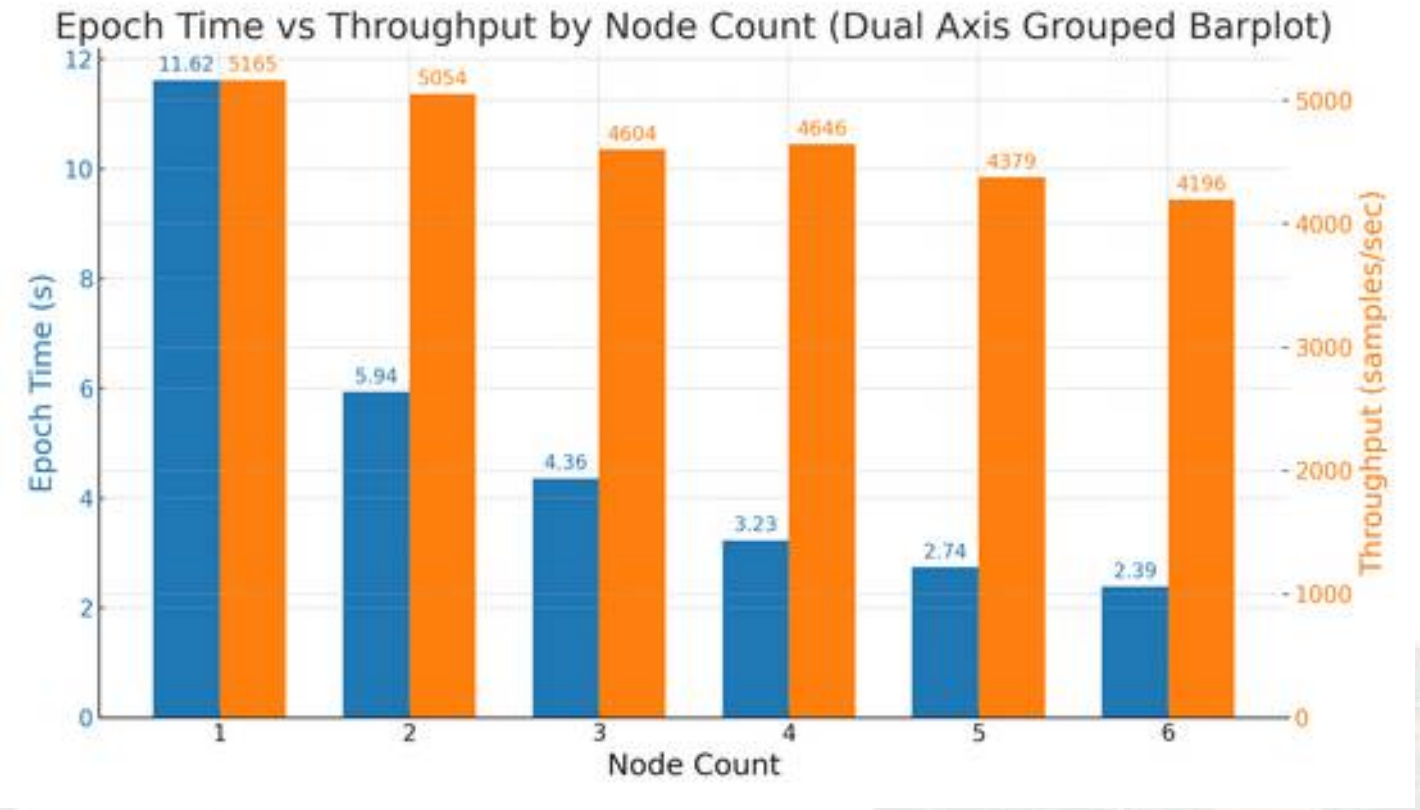
Reproducibility & Checkpoints

- **Env Capture**: Record versions (e.g., Python, PyTorch).
- **Seeds**: Ensure random consistency.
- **Logging**: Track metrics.
- **Resume**: Load checkpoints for continuity.

```
import torch
import torch.distributed as dist

def train(rank, world_size):
    dist.init_process_group("nccl", rank=rank, world_size=world_size)
    torch.manual_seed(42 + rank)  # Seed per rank
    model = torch.nn.Linear(10, 10).cuda(rank)
    torch.save(model.state_dict(), f"checkpoint_{rank}.pt")
    print(f"Rank {rank} seed: {torch.initial_seed()}, Checkpoint saved")

if __name__ == "__main__":
    world_size = 2
    torch.multiprocessing.spawn(train, args=(world_size,), nprocs=world_size, join=True)
```

Case Study: End-to-End

- **Single-Node**: Baseline performance.
- **Multi-Node**: Scalability with GPUs.
- **Speedups**: Measured from reports.
- **Use Case**: Distributed training speedup.



Takeaways & Next Steps

- **Pthreads/OpenMP**: Best for single-node, shared-memory parallelism; use for CPU-bound tasks
- **OpenACC/CUDA**: Ideal for GPU offloading; prioritize high arithmetic intensity
- **MPI**: Scales to multi-node; optimize collectives for distributed workloads
- **PyTorch DDP/FSDP**: Use for distributed deep learning; balance data vs. model parallelism
- **Slurm**: Orchestrate multi-node jobs; tune resource requests for efficiency

Decision Tree:

- Single node? → Pthreads/OpenMP for CPU, OpenACC/CUDA for GPU
- Multi-node? → MPI for general HPC, PyTorch DDP/FSDP for DL
- GPU-heavy? → CUDA for control, OpenACC for simplicity
- DL-specific? → DataParallel (single node) or DDP/FSDP (multi-node)

Next Steps:

- Profile your workload to identify bottlenecks
- Experiment with hybrid models (e.g., MPI + OpenMP)
- Use provided code samples to start prototyping
- Monitor Slurm jobs for resource efficiency

Q&A + Resources

Code: All examples (pthreads_example.c, p_L_R.c, streams.cu, multi_node.py, pytorch_sbatch.slurm) available at dropbox.

Slides: Full deck at dropbox.

Reports: Performance reports and case studies at () .

Thank You!