HPC Lecture Notes

Supercomputers

ASK: who built a PC, what CPU? Write specs. Compare: AMD EPYC Gen4, 1.8 GHz. "Which is faster?" Reveal: this is the fastest supercomputer's CPU.

- 1 FLOP = one floating-point op.
- MacBook Pro Laptop ~6 TFLOPS (CPU + GPU).
- Supercomputer ~ExaFLOPS (10^{18}).

Architecture Stack

- Core: executes instructions
- CPU: multiple cores
- Node: shared memory
- Cluster: distributed memory
- GPU (Accelerator): separate memory
- Interconnect: InifiniBand (~1-2 μs latency)

Example Problem: Computing Pi

Common HPC Workloads

- Scientific simulations: weather, climate, astrophysics, molecular dynamics
- Life sciences: genomics, drug discovery, protein folding
- Data-intensive science: high-energy physics, radio astronomy, cosmology
- AI/ML: training large models, inference at scale
- Material science: quantum chemistry, materials design
- Engineering: CFD, structural analysis, optimization
- Financial modeling: risk analysis, option pricing

A Simple Problem: calculating π

Circle of radius 1:

$$x^2 + y^2 = 1$$

Area of quarter circle:

$$\frac{\pi}{4} = \int_0^1 \sqrt{1 - x^2}\, dx$$

N rectangles, $\Delta x = \frac{1}{N}$:

$$\pi \approx 4 \sum_{i=0}^{N-1} \sqrt{1 - x_i^2} \cdot \Delta x, \quad x_i = (i + 0.5) \cdot \Delta x$$

Serial Baseline

```
double sum = 0.0, dx = 1.0 / N;
for (long i = 0; i < N; i++) {
    double x = (i + 0.5) * dx;
    sum += sqrt(1.0 - x * x);
}
double pi = 4 * sum * dx;
```

Embarrassingly Parallel

- Every iteration independent – no data dependency between rectangles
- Ideal speedup = P with P processors

Shared-Memory Parallelism with Pthreads

Naive Version (Buggy)

```
// shared global
double sum = 0.0;

// inside each thread:
for (long i = start; i < end; i++) {
    double x = (i + 0.5) * dx;
    sum += sqrt(1.0 - x * x);   // BUG
}
```

Demo: run 3 times – wrong answer each time, slower than serial.

Race Condition

`sum += ...` is load-add-store, not atomic. Concurrent threads: both load same value, both store – one update lost.

Cache Performance Issue

- `sum` sits on one cache line (64 bytes)
- Every write invalidates that line on all other cores (MESI protocol)
- Cache line ping-pongs between cores – destroys performance

Fix: Local Accumulation

```
// inside each thread:
double local_sum = 0.0;
for (long i = start; i < end; i++) {
    double x = (i + 0.5) * dx;
    local_sum += sqrt(1.0 - x * x);
```

```
}
partial_sums[tid] = local_sum;

// after join, main thread:
for (int i = 0; i < num_threads; i++)
    sum += partial_sums[i];
double pi = sum * dx;
```

THE PATTERN: partition, compute locally, reduce.

Demo: 1, 2, 4, 8 threads. ASK: "Why not 8x speedup?"

Scaling

Reason: serial fraction in the program.

ASK: "What are some serial parts of this code?" Reduction step, thread creation/join, loop overhead.

Demo tiny N (1000), 8 threads – parallel slower.

$$T_{total} = T_{serial} + \frac{T_{parallel}}{P}$$

Amdahl's Law:

$$T = \frac{1}{s + \frac{1-s}{p}}$$

Why MPI

Scale beyond one node.

Three limits of shared memory: core count (~64-128), RAM (~512 GB-1 TB), single point of failure.

"Nodes can't see each other's RAM. Must send data explicitly."

MPI

Concepts

ASK: "Process vs thread?" They know this. MPI process = own address space, own stack, own heap. Nothing shared.

Communicators: group of processes that can talk to each other. MPI_COMM_WORLD = all processes.

Ranks: Id number of process within communicator. From 0 to size-1.

Messages: atomic operations

Data	Count	Datatype	Dest/Src	Tag	Communicator
`buf`	`count`	`MPI_DOUBLE`, ...	rank id	tag	`MPI_COMM_WORLD`

Point-to-point Communication:

- Send: `MPI_Send(buf, count, datatype, dest, tag, comm)`
- Recv: `MPI_Recv(buf, count, datatype, source, tag, comm, &status)`

Collective operations: all processes call together. Examples:

- Broadcast: one rank sends same data to all
- Scatter: one rank sends different chunks to each
- Gather: all ranks send to one
- Reduce: all ranks contribute, combined with op (+, max, etc), result on one rank

SPMD: same program all ranks, rank determines work. Exam analogy: same paper, answer your seat's questions.

Six Functions

```
MPI_Init(&argc, &argv)
MPI_Comm_size(MPI_COMM_WORLD, &size)
MPI_Comm_rank(MPI_COMM_WORLD, &rank)
MPI_Send(buf, count, datatype, dest, tag, comm)
MPI_Recv(buf, count, datatype, source, tag, comm, &status)
MPI_Finalize()
```

Code Walkthrough

pi-mpi.c. Each rank computes its slice of rectangles. MPI_Reduce(MPI_SUM) to rank 0. Rank 0 prints.

```
int rank, size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

// Each process handles a subset of iterations
for (long i = rank; i < N; i += size) {
    double x = (i + 0.5) * step;
    local_sum += sqrt(1.0 - x * x);
}

double sum;
MPI_Reduce(&local_sum, &sum, 1, MPI_DOUBLE,
```

```
            MPI_SUM, 0, MPI_COMM_WORLD);
double pi = 4 * sum * step;
```

Demo: mpirun -np 4 ./pi-mpi 10000000.

What's Missing (brief)

- Real programs: rank 0 reads input, MPI_Bcast to all
- Computation vs communication: if computation is fast, communication overhead dominates
- Load imbalance: everyone waits for slowest rank (stragglers)

Job Schedulers

ASK: "How do you actually run on 10,000 nodes?"

SLURM: submit job script requesting nodes, cores, wall time. Scheduler queues and allocates. sbatch, srun. OS process scheduling scaled to a datacenter. - Interactive: `srun --nodes=4 --ntasks-per-node=48 --time=00:10:00 --pty bash` - Batch: write job script:

```
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=32
#SBATCH --time=00:10:00

mpirun -n 128 ./pi-mpi 10000000
```

GPU and CUDA

Why GPUs

- CPU: few big cores, big cache, complex control – optimized for latency
- GPU: thousands of tiny cores, small cache – optimized for throughput
- "Professor vs 10,000 students grading exams"
- Data parallelism: same op on every element. Semicircle = perfect fit.
- More and more of supercomputing is GPU-accelerated.
- Power efficiency: GPU FLOPS/Watt much higher than CPU.

GPU Programming Libraries

- CUDA: NVIDIA's proprietary API for programming their GPUs. C/C++ extensions, low-level control.
- OpenCL: open standard for heterogeneous computing. More portable, but less optimized.
- HIP: AMD's CUDA-compatible API. Port CUDA code to run on AMD GPUs.

Host/Device Model

CPU (host) and GPU (device) are separate computers with separate memory. Connected by PCIe bus. Must copy data explicitly.

- GPU memory bandwidth: ~1000 GB/s
- PCIe bus: ~25 GB/s (40x slower – biggest bottleneck)

Memory hierarchy:

$$\text{Registers} \to \text{L1} \to \text{L2} \to \text{L3} \to \text{RAM} \to \text{PCIe}$$

Each level ~10x slower, ~10x bigger.

ASK: "Why was buggy pthreads code slow, in terms of this hierarchy?" Cache line bouncing = traffic up and down.

CUDA Thread Hierarchy

```
Software                      Hardware
-----------------------------------------
Grid                          Entire GPU
  Block 0  -------------->    SM 0 (or any SM)
    Warp 0 (threads 0-31)       32 CUDA cores execute in lockstep
    Warp 1 (threads 32-63)      ...
  Block 1  -------------->    SM 1 (or any SM)
    ...
  Block N  -------------->    scheduled onto available SM
```

Thread hierarchy:

- Threads: basic unit of execution. Each thread executes same code (SIMT). Each thread has its own registers.
- Warps: group of 32 threads that execute in lockstep. If threads in a warp diverge (if/else), GPU serializes paths.
- Blocks: independent units of work, scheduled on any SM. Multiple blocks can run concurrently on same SM if resources allow.
- Grid: collection of blocks. Launched by CPU, runs on GPU.

Global thread index:

```
idx = blockIdx.x * blockDim.x + threadIdx.x
```

Example: block 3, blockDim 256, thread 7: $3 \times 256 + 7 = 775$

CUDA Workflow

```
cudaMalloc(&d_ptr, size)              // 1. allocate on GPU
cudaMemcpy(d_ptr, h_ptr, size, H2D)  // 2. copy host -> device
kernel<<<gridDim, blockDim>>>(args)   // 3. launch kernel
```

```
cudaMemcpy(h_ptr, d_ptr, size, D2H)  // 4. copy device -> host
cudaFree(d_ptr)                       // 5. free GPU memory
```

Kernel Launch Syntax

`kernel<<<gridDim, blockDim>>>(args)`

gridDim = number of blocks

blockDim = threads per block

Total threads = gridDim * blockDim.

Shared Memory and Reduction

Naive (pi-cuda.cu): each thread writes partial[idx], CPU sums 65,536 values.

```
__global__ void pi_kernel(long long N, double step, double *partial) {
    long long idx = blockIdx.x * blockDim.x + threadIdx.x;
    long long stride = (long long)blockDim.x * gridDim.x;

    double local = 0.0;
    for (long long i = idx; i < N; i += stride) {
        double x = (i + 0.5) * step;
        local += sqrt(1.0 - x * x);
    }

    partial[idx] = local;
}
```

Run pi-cuda.

When GPUs Struggle

1. Not enough work – overhead dominates
2. Branchy code – warp divergence (threads in same warp diverge on if/else, GPU serializes paths)
3. Sequential dependencies
4. Global memory vs shared memory: global memory (RAM) is slow, shared memory (on-chip) is fast but limited. Need to optimize access patterns.
5. PCIe bottleneck: if you have to copy data back and forth, GPU may not help.
6. Synchronization: threads in same block can sync with ___syncthreads(), but no sync across blocks. Need to design algorithms carefully.

Multi-GPU: NCCL

CUDA covers one GPU. Real systems have multiple GPUs per node, multiple nodes.

NCCL (NVIDIA Collective Communications Library, pronounced "nickel") – MPI-like collectives but optimized for GPU-to-GPU communication.

Provides AllReduce, Broadcast, AllGather, ReduceScatter – same concepts as MPI

Exploits fast interconnects: NVLink (GPU-to-GPU within node, ~900 GB/s) vs PCIe (~25 GB/s)

Handles multi-node GPU communication over network (InfiniBand)

Data stays on GPU – no detour through CPU/host RAM

Why it matters: this is the communication backbone for distributed deep learning. When you hear "training on 1,000 GPUs," NCCL is doing the gradient synchronization.

Connection: MPI for CPU-to-CPU across nodes, NCCL for GPU-to-GPU across nodes. Modern HPC often uses both together.

Hybrid Model

Real supercomputers use MPI + X (X = pthreads, OpenMP, CUDA). MPI for inter-node communication, X for intra-node parallelism.

Wrap-Up

Comparison table:

	pthreads	MPI	CUDA
Memory	shared	distributed	separate host/device
Unit	thread	process	thread in blocks
Scale	1 node	many nodes	1 GPU
Communication	shared vars	messages	shared mem + global mem
Bottleneck	contention	network latency	PCIe transfer

Closing: "One problem, three ways. Same pattern: partition, compute locally, reduce. Differences: where memory lives, how workers communicate."

Resources: top500.org, LLNL pthreads tutorial, openmp.org, mpitutorial.com, developer.nvidia.com