CMSC416 Instructor Notes Fall 2025

Abhinav Bhatele

Contents

Chapter 1

Message Passing and MPI

1.1 Parallel Programming Models and Runtimes

1.1.1 Shared vs. Distributed vs. Hybrid Memory

Shared memory systems All processors can access a common memory space. This may be:

- UMA (Uniform Memory Access): equal latency to all memory locations.
- NUMA (Non-Uniform Memory Access): access time depends on location.

Programming model: threads with a shared address space (e.g., Pthreads, OpenMP).

Distributed memory systems Each processor has its own private memory. Processes communicate by *explicit* message passing. This is the model targeted by MPI.

Hybrid systems Combine shared memory within a node and distributed memory across nodes; for example, use OpenMP threads inside each node and MPI across nodes.

1.1.2 User Code, Parallel Runtimes, and Communication Libraries

Conceptual software stack:

1. **User code:** The program written by the student.
2. **Parallel runtime:** Implementation of a programming model:
   - MPI runtime (for MPI).
   - OpenMP runtime (for OpenMP).
   - Charm++ runtime (for Charm++).
3. **Vendor communication libraries and OS:**
   - Libraries like UCX, PSM, etc.
   - Operating system services.

The runtime is responsible for:

- Receiving incoming messages on behalf of processes.
- Buffering data until the receiving process posts a matching receive.
- Delivering the data to the right process at the right time.
- Bookkeeping and correctness (e.g., avoiding memory corruption).

Important points:

- The runtime runs on every process. There is no dedicated "runtime process"; its code is linked into each MPI rank.
- You do not write a separate runtime program; you link your code with the MPI/OpenMP library provided by the system.

1.2 Message Passing and MPI Basics

1.2.1 Distributed Memory and SPMD Style

In message passing:

- The program consists of multiple independent *processes*.
- Processes communicate by explicitly sending and receiving messages.
- Processes are typically created by a launcher such as `mpirun` or a job script, not via `fork()` inside the program.

MPI programs follow the **Single Program, Multiple Data (SPMD)** model:

- All processes start from the same executable.
- Each process has a unique *rank* (integer ID).
- Processes may take different branches and operate on different data based on their rank.

Parallel speedup arises from processes computing on different parts of the data *concurrently*.

1.2.2 MPI History and Standardization

Before MPI:

- PVM (Parallel Virtual Machine) from Oak Ridge was an influential message-passing system.
- Between roughly 1989–1993 many incompatible message-passing systems existed.

The **MPI Forum** formed in 1992 to standardize message passing:

- MPI 1.0 released in 1994.
- Later versions: MPI 2.0 (1997), MPI 3.0 (2012), MPI 4.0 (2021), each adding functionality.

MPI is a *specification*, not a single library. Implementations include:

- MPICH (Argonne).
- MVAPICH (Ohio State).
- Open MPI (multi-institution open source).
- Vendor-tuned MPIs: Intel MPI, Cray MPI, NVIDIA, AMD, etc.

1.2.3 Portability

MPI's design goal is **portability**:

- The same source code can run on laptops, university clusters, DOE supercomputers, or Ethernet-connected machines, as long as an MPI implementation is available.
- Implementations hide hardware differences (network types, messaging libraries, CPU architectures) behind the standard API.

1.3 Structure of a Simple MPI Program

1.3.1 `Hello World` in MPI

A minimal MPI "Hello World" in C:

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

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);  // initialize MPI

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // my rank
    MPI_Comm_size(MPI_COMM_WORLD, &size); // total number of ranks

    printf("Hello world, I am process %d of %d\n", rank, size);

    MPI_Finalize(); // clean up
    return 0;
}
```

Listing 1.1: Simple MPI Hello World

Important points:

- `MPI_Init` must be called before any other MPI routine.
- `MPI_Finalize` should be the last MPI call.
- `MPI_COMM_WORLD` is the default communicator containing all processes.
- Each process prints a line; the order is not guaranteed.

1.3.2 Compiling and Running

Compilation. Use MPI compiler wrappers such as `mpicc`:

```
mpicc -o hello hello.c
```

Running. Use an MPI launcher:

```
mpirun -np 4 ./hello
```

`-np 4` (or `-n 4`) requests 4 MPI processes. Each process gets a unique rank in `MPI_COMM_WORLD`. In a batch environment, `mpirun` is called from within the job script on the allocated nodes/cores.

1.4 Cluster Resources and Job Scheduling

On a cluster node with, say, 128 cores:

- Job scripts request resources (nodes/cores, wall time).
- Example request: 128 cores for 1 hour.
- The scheduler charges in core-hours (e.g., 128 cores $\times$ 1 hour = 128 core-hours), regardless of how many cores your program actually uses.

If you:

- Allocate 128 cores and run 129 MPI processes, the job will fail.
- Allocate 128 cores but only run 4 processes, the job will run but waste resources; you still pay for all 128 cores.

Clean termination is critical:

- If the MPI program calls `MPI_Finalize` and exits, the scheduler can reclaim resources early.
- If it deadlocks or crashes but does not exit, cores may be held until the requested wall time ends.

1.5 Communicators, Ranks, and Datatypes

1.5.1 Communicators and Ranks

- A **communicator** is a group of processes plus a communication context.
- `MPI_COMM_WORLD` is the default communicator containing all ranks.
- Within a communicator, each process has a **rank** (integer ID) from 0 to `size-1`.
- Many MPI routines take a communicator argument:
  - `MPI_Comm_rank(comm, &rank)`
  - `MPI_Comm_size(comm, &size)`
  - `MPI_Send(..., comm)`
  - `MPI_Recv(..., comm)`

MPI also supports creating *subcommunicators*. For example, in a 2D process grid one can create:

- Row communicators.
- Column communicators.

Each communicator has its own local rank numbering. For most course examples, only `MPI_COMM_WORLD` is used.

1.5.2 MPI Datatypes

MPI defines built-in datatypes:

- `MPI_INT`, `MPI_CHAR`, `MPI_DOUBLE`, etc.

Messages specify:

- `count`: number of elements.
- `datatype`: type of each element.

MPI uses this to compute how many bytes to transfer. MPI also allows user-defined datatypes (e.g., for C structs or non-contiguous data), though these are mostly beyond the introductory examples.

1.6 Blocking Point-to-Point Communication

1.6.1 `MPI_Send` and `MPI_Recv`

Conceptual prototype of `MPI_Send`:

```
MPI_Send(void *buf, int count, MPI_Datatype datatype,
         int dest, int tag, MPI_Comm comm);
```

- `buf`: pointer to first element of the send buffer.
- `count`: number of elements to send.
- `datatype`: MPI datatype (e.g., `MPI_INT`).
- `dest`: destination rank in the communicator.
- `tag`: integer message tag used to categorize messages.
- `comm`: communicator, typically `MPI_COMM_WORLD`.

`MPI_Recv` has a similar interface plus a status argument. MPI routines return an integer error code; the interesting data is returned via pointer arguments (e.g., `&rank` in `MPI_Comm_rank`).

1.6.2 Blocking Semantics

- **Send:** `MPI_Send` returns when it is safe for the user to reuse or overwrite the send buffer. Internally, MPI may copy the data into a system buffer or wait for a matching receive.
- **Receive:** `MPI_Recv` returns only after the requested message has arrived and been copied into the receive buffer.

1.6.3 Matching Rules and Ordering

Messages are matched based on:

- Source rank (on the receive side).
- Destination rank (on the send side).
- Communicator.
- Tag.

MPI guarantees ordering for a *single* sender/receiver pair:

- If process A sends two messages to process B in order, and process B posts matching receives in order, they will be received in the same order.

MPI does not automatically check that `count` and `datatype` match on both sides; mismatches lead to undefined behavior and are the programmer's responsibility to avoid.

1.6.4 Deadlock Example

Two-process deadlock:

- Process 0: `MPI_Send` to 1, then `MPI_Recv` from 1.
- Process 1: `MPI_Send` to 0, then `MPI_Recv` from 0.

Both processes block in `MPI_Send`, waiting for the other to post a matching receive. Neither reaches its `MPI_Recv`, so the program deadlocks.

Fix: have one process receive first:

- Process 0: `MPI_Recv` then `MPI_Send`.
- Process 1: `MPI_Send` then `MPI_Recv`.

1.7 Examples: Simple Sends and Rank-Based Logic

1.7.1 Two-Process Example

- Rank 0 sets `data = 7` and calls `MPI_Send` to rank 1.
- Rank 1 calls `MPI_Recv` from rank 0 into `data`.
- Rank 1 prints the received value; it will be 7.

If rank 1 printed `data` before the receive, it would print an uninitialized (garbage) value because each process has its own memory.

1.7.2 Even/Odd Ranks Example

Assume ranks 0–3:

- Even ranks (`rank % 2 == 0`) set `data = rank` and send to `rank + 1`.

- Odd ranks set `data = 2 * rank` and then receive from `rank - 1`.

Before communication:

Rank	`data` (local)
0	0
1	2
2	2
3	6

After communication:

- Rank 1 receives 0 from rank 0.
- Rank 3 receives 2 from rank 2.

Notice that ranks compute communication partners by simple arithmetic (`rank+1`, `rank−1`) instead of hard-coded IDs.

1.8 Non-Blocking Communication

1.8.1 `MPI_Isend`, `MPI_Irecv`, and Requests

Non-blocking operations initiate communication and return immediately, allowing computation to proceed while data is in transit.

- `MPI_Isend`: non-blocking send.
- `MPI_Irecv`: non-blocking receive.

Both take an extra `MPI_Request` argument:

- The request is an *opaque* handle managed by MPI.
- It links the non-blocking operation to its eventual completion.

Even if example code uses the same variable name (e.g., `req`) on all ranks, in SPMD each rank has its own copy of that variable in its own memory.

1.8.2 Completion Routines

To ensure correctness, a non-blocking operation must be completed by:

- `MPI_Wait(&req, &status)`: blocks until the operation completes.
- `MPI_Test(&req, &flag, &status)`: non-blocking poll.
- `MPI_Waitall`, `MPI_Waitany`: variants for arrays of requests.
- `MPI_Waitsome`: similar to `Waitany` but returns multiple completed requests (described but not required in the course).

Guidelines:

- Do not read data from a receive buffer until the corresponding wait has completed.
- Do not overwrite a send buffer until its send operation has completed.

1.8.3 Overlap of Communication and Computation

Non-blocking communication enables overlap:

1. Post `MPI_Isend` or `MPI_Irecv`.
2. Perform independent computation that does not depend on the communicated data.
3. Later, call `MPI_Wait` to ensure completion before using the buffer.

The goal is to reduce idle time waiting in blocking communication calls.

1.9 Halo Exchange and 2D Stencil / Game of Life

Consider a 2D stencil or Game of Life grid:

- The global grid is decomposed into subdomains, one per MPI process.
- Each process maintains **ghost cells** (halo cells) that store copies of its neighbors' boundary rows/-columns.

For a 1D decomposition by rows:

- Each process communicates with its neighbor above and below.
- It:
  - Sends its top row to the process above, bottom row to the process below.
  - Receives corresponding rows into its top and bottom ghost layers.

A typical non-blocking pattern:

1. Post `MPI_Irecv` calls for incoming boundary rows (from neighbors).
2. Post `MPI_Isend` calls for outgoing boundary rows.
3. Compute on interior cells that do not depend on ghost layers.
4. Call `MPI_Wait` (or variants) to ensure ghost data has arrived.
5. Update border cells using local and ghost data.

1.9.1 Memory Layout and Contiguity

Assuming C row-major arrays:

- Entire rows are stored contiguously in memory ⇒ can send an entire row with a single (I)send by pointing to the first element and setting `count = numCols`.
- Columns are *not* contiguous; elements are separated by row size. Options:
  - Manually pack a column into a contiguous buffer, send, and unpack.
  - Define an MPI derived datatype describing the strided layout.

1.10 Collective Operations

1.10.1 Point-to-Point vs. Collective

Point-to-point Communication between a specific pair of processes, using routines like `MPI_Send` and `MPI_Recv`.

Collective Operations involving *all* processes in a communicator. Every participating process must call the collective with matching arguments.

1.10.2 `MPI_Barrier`

- A barrier is a synchronization point.
- All processes calling `MPI_Barrier(comm)` block until every process in `comm` has reached the barrier.
- After that, they all continue execution.

1.10.3 `MPI_Bcast`

- `MPI_Bcast(buf, count, datatype, root, comm)`:
  - Root process: the data in `buf` is the source.
  - All other processes: `buf` is the destination for the broadcast data.
- Same argument on all ranks; MPI interprets its role based on whether the process is the root or not.
- Can broadcast scalars or arrays.

1.10.4 `MPI_Reduce` and `MPI_Allreduce`

- `MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm)`:
  - Each rank contributes data in `sendbuf`.
  - The root receives the aggregated result in `recvbuf`.
  - `op` is an operation (e.g., `MPI_SUM`, `MPI_MIN`, `MPI_MAX`) that must be associative (and usually commutative).
- `MPI_Allreduce` performs the same reduction but returns the result to *all* ranks.

1.10.5 `MPI_Scatter` and `MPI_Gather`

Scatter. `MPI_Scatter` distributes different segments of an array on the root to each process:

- Root: `sendbuf` points to a full array logically split into equal chunks.
- All ranks (including root): `recvbuf` receives its chunk.
- For variable chunk sizes, use `MPI_Scatterv`.

Gather. `MPI_Gather` collects data from all ranks onto the root:

- All ranks: `sendbuf` contains local data.
- Root: `recvbuf` collects all contributions in rank order.
- For variable sizes, use `MPI_Gatherv`.

1.10.6 `MPI_Scan`

- `MPI_Scan` performs a parallel prefix (scan) operation.
- With `MPI_SUM`, rank 0 gets its own value, rank 1 gets sum of ranks 0–1, rank 2 gets sum of ranks 0–2, etc.

1.10.7 Implementation Insight

Collectives are implemented internally using point-to-point operations:

- Naive broadcast: root sends an individual message to each rank.
- Efficient broadcast: tree-based algorithm achieving $O(\log P)$ steps.

Users do not need to implement these; MPI chooses efficient algorithms.

1.10.8 `MPI_Wtime` for Timing

- `MPI_Wtime` returns a wall-clock time in seconds since an arbitrary reference point.
- To time a code segment:
  - Call `t1 = MPI_Wtime()` before.
  - Call `t2 = MPI_Wtime()` after.
  - Elapsed time is `t2 - t1`.
- The underlying resolution depends on the system and MPI build.

1.11 Parallel π Computation with MPI

1.11.1 Sequential Algorithm

Sequential numerical integration to approximate π:

- Choose number of intervals n (larger $n \Rightarrow$ more accurate).
- Step size: $h = 1.0/n$.
- Loop over $i = 1, \ldots, n$:
  - $x_i = (i - 0.5) \times h$.
  - $\texttt{sum} += \frac{4}{1+x_i^2}$.
- $\pi \approx h \times \texttt{sum}$.

1.11.2 Parallel Decomposition

Iterations are independent, so the problem is "embarrassingly parallel".
Two main distribution strategies:

Round-robin. Each rank computes:

```
for (i = myRank + 1; i <= n; i += numRanks) {
    // accumulate local sum
}
```

Block. Divide $[1, n]$ into contiguous chunks per rank:

- Rank r gets iterations from some `start_i(r)` to `end_i(r)`.

In the lecture, round-robin is used for illustration.

1.11.3 Broadcasting Parameters

The example code includes a call to `MPI_Bcast` to distribute n:

- In many MPI setups, all ranks receive the same command-line arguments, so broadcasting n is not strictly necessary.
- It demonstrates how a single rank (e.g., root) can read input or compute a parameter and then broadcast it.

1.11.4 Combining Partial Results with `MPI_Reduce`

Each rank computes a partial sum and/or partial π value. To match the sequential program's output:

- Use `MPI_Reduce` with operation `MPI_SUM` to combine partial sums from all ranks into a single global sum (or global π) on the root.
- The root then prints the result.

Correctness requirement: the parallel program should produce the same π approximation as the sequential code (up to floating-point rounding).

1.12 MPI Internals and Message Protocols

1.12.1 Message Envelope and Data

Each `MPI_Send` creates a message consisting of:

- An **envelope** containing metadata:
  - Source rank, destination rank.
  - Tag.
  - Communicator ID.
  - Message length, etc.
- A **payload** containing the user data.

1.12.2 Eager, Rendezvous, and Short Protocols

MPI uses different internal protocols depending on message size:

Eager protocol. Used for small messages:

- Sender transmits the message immediately, without waiting for the receiver to post a matching receive.
- Assumes the receiver has sufficient buffering capacity.
- Low latency but can increase memory pressure if overused.

Rendezvous protocol. Used for larger messages:

- Two-phase handshake:
  1. Sender sends a control message: "I want to send N bytes."
  2. Receiver replies when a buffer is ready.
- Only then does the sender transmit the actual payload.

Short messages. For very small messages:

- MPI may pack the payload directly into the envelope.
- This can reduce overhead and the number of internal copies.

1.12.3 Tuning Protocol Thresholds

Many MPI implementations expose environment variables to control the cutoff between eager and rendezvous:

- Increasing the eager limit treats more messages as "small" and may improve latency.
- But it can increase memory usage and buffer contention on the receiver.

1.12.4 Reliability and Failures

In HPC environments:

- Underlying protocols typically ensure reliable delivery (with retries).
- If something goes wrong (e.g., hardware failure), the MPI job may hang or crash rather than silently losing messages.
- Programmers generally do not handle individual packet losses; the network and MPI handle these details.

1.12.5 Software Stack and Network Path

For an inter-node message:

1. Data moves from user memory into MPI buffers.
2. MPI calls into vendor communication libraries (e.g., UCX, PSM).
3. Data is handed to the NIC (Network Interface Card).
4. The NIC injects packets into the network.
5. Packets traverse switches/routers and are buffered/forwarded.
6. At the destination, the NIC receives packets.
7. MPI copies data from NIC buffers into the destination process's memory and completes the receive.

1.12.6 Packets, Flits, and Wormhole Routing

- High-level messages are split into *packets*.
- Packets are further composed of *flits* (flow control digits), the smallest units flowing through the network.
- In wormhole routing:
  - The header flits establish a path.
  - The rest of the message follows through the same path with minimal buffering in intermediate routers.

1.12.7 Heterogeneity and Endianness

MPI works across nodes as long as:

- The same executable format can run on all nodes.
- Endianness and basic data representations are compatible, or explicit conversions are performed.

MPI does not automatically solve arbitrary heterogeneity issues; it operates within the capabilities of the underlying systems.

Chapter 2

Performance Analysis in Parallel Computing

2.1 Performance Analysis and Scalability

2.1.1 Strong vs. Weak Scaling

Strong scaling. In strong scaling experiments:

- The global problem size is fixed.
- The number of processes P is increased.
- Ideally, runtime decreases as P increases.
- Example: a Game of Life simulation on a fixed 256×256 grid run with 1, 2, 4, and 8 processes.

Weak scaling. In weak scaling experiments:

- The problem size *per process* is held constant.
- When P doubles, the total problem size doubles.
- Ideally, runtime stays roughly constant because each process does the same amount of work.
- Example: each process holds a fixed number of cells; as we add more processes, the global grid grows (e.g., from 256×256 to 512×256 and beyond).

2.1.2 Amdahl's Law

Amdahl's law captures the effect of non-parallelizable code sections. Suppose:

- A fraction F of the execution time is parallelizable.
- The remaining fraction $1 - F$ is inherently serial.

On P processes, normalized execution time is:

$$T_P = (1 - F) + \frac{F}{P}.$$

The speedup is:

$$S(P) = \frac{1}{T_P} = \frac{1}{(1 - F) + \frac{F}{P}}.$$

As $P \to \infty$, the maximum speedup is:

$$S_{\max} = \frac{1}{1 - F}.$$

If 40% of the code is serial ($1 - F = 0.4$), the maximum speedup is $1/0.4 = 2.5$ regardless of how many processors are used. This underscores the importance of reducing serial or poorly parallelized portions of the code.

2.1.3 What Is Performance Analysis?

Performance analysis is the systematic study of where time is spent in a parallel program and why performance may fall short of expectations. Common contributors to poor performance include:

- Large serial or sequential sections that were never parallelized.
- Inefficient per-process sequential performance (e.g., cache-unfriendly loops).
- Communication overheads from excessive or poorly structured messaging.
- Idle time when processes or threads wait for data or synchronization.
- Insufficient overlap between communication and computation.

There are two broad approaches:

Analytical techniques use mathematical models to relate execution time to problem size N and number of processes P. They include time-complexity analysis, Amdahl's law, iso-efficiency analysis, and communication models such as the α–β and LogP models.

Empirical techniques use measurements from profilers, hardware counters, and tracing tools to observe actual runtime behavior. They provide concrete data on where time is spent and how performance behaves as P changes.

Assignment 2 uses both perspectives: analytical models for reasoning and profiling data for validation and deeper insight.

2.2 Parallel Prefix Sum (Scan) Complexity

The parallel prefix sum (scan) is a canonical example used to illustrate both computation and communication costs in a parallel program.

Consider N elements distributed across P processes, with each process holding N/P elements.

2.2.1 Local Computation

Each process performs a local prefix sum over its N/P elements. This is a sequential scan that requires $\mathcal{O}(N/P)$ additions:

$$T_{\text{local}} = \mathcal{O}\left(\frac{N}{P}\right).$$

2.2.2 Global Prefix Phases

After computing local prefix sums, the algorithm uses a tree-based communication pattern across processes in $\log P$ phases:

- In phase 0, each process communicates with a neighbor one step away.
- In phase 1, it communicates with a neighbor two steps away.
- In phase 2, four steps away, and so on, for a total of $\log P$ phases.

In each phase, a process that receives an offset value adds that offset to each of its N/P local elements. Thus, each phase requires $\mathcal{O}(N/P)$ operations per process, and across all phases:

$$T_{\text{phases}} = \mathcal{O}\left(\frac{N}{P}\log P\right).$$

2.2.3 Total Computation per Process

Combining the local scan and the updates across phases, the per-process computation is approximately:

$$T_{\text{comp}} = \mathcal{O}\left(\frac{N}{P}\right) + \mathcal{O}\left(\frac{N}{P}\log P\right) \approx \mathcal{O}\left(\frac{N}{P}\log P\right).$$

Note that we focus on per-process time (and ultimately the maximum across processes), not on the total number of operations over all processes.

2.3 Communication Cost Models

2.3.1 Coarse Message Count Model

In the prefix sum example:

- Each process sends one message per phase.
- There are $\log P$ phases.

If we treat each message as having a constant cost, then the per-process communication cost is:

$$T_{\text{comm}} = \mathcal{O}(\log P).$$

This is a simplified estimate that ignores message size and network details, but it is often a useful starting point.

2.3.2 α–β (Alpha–Beta) Model

A more refined model writes the cost of sending a message of size n as:

$$T(n) = \alpha + \beta n,$$

where:

- α is the startup cost (latency and initial overhead).

- β is the per-byte time (inverse bandwidth).
- n is the message size in bytes or words.

For an algorithm with many messages, the total communication time is the sum of such terms over all messages. A similar modeling idea can be applied to memory transfers (e.g., from RAM to registers).

2.3.3 LogP Model

The LogP model describes parallel communication using four parameters:

- L: latency to deliver a message across the network.
- o: per-message overhead during which a processor is engaged in sending/receiving and cannot do other work.
- g: gap between consecutive sends or receives, modeling network throughput.
- P: number of processes.

LogP can capture effects such as network contention and limited buffering more accurately than the simple α–β model, but it is also more complex.

2.3.4 Overlap and Bulk-Synchronous Execution

In bulk-synchronous programs, computation and communication phases are distinct:

$$T \approx T_{\text{comp}} + T_{\text{comm}}.$$

When non-blocking communication is used, some communication can be overlapped with computation. In that case, the visible communication cost is reduced, and the total time is closer to:

$$T \approx \max(T_{\text{comp}}, T_{\text{comm}}^{\text{non-overlapped}}).$$

Modeling overlap correctly is important for understanding the benefit of non-blocking MPI operations.

2.4 Overhead, Efficiency, and Isoefficiency

2.4.1 Decomposing Parallel Time

Let T_1 denote the sequential execution time for a problem of size N, and let T_P be the execution time on P processes for the same problem.

The total time spent across all processes in the parallel run is PT_P. This can be decomposed into:

$$PT_P = T_1 + T_O,$$

where T_1 is the useful work (equivalent to the sequential execution) and T_O is **overhead**, consisting of:

- Extra computation introduced by parallel algorithms (e.g., partial sums needed only in the parallel prefix).
- Communication time.
- Idle time due to waiting for messages or synchronization.
- Other parallel-specific overheads such as contention.

2.4.2 Efficiency in Terms of Overhead

Parallel efficiency is defined as:

$$E = \frac{\text{speedup}}{P} = \frac{T_1}{PT_P}.$$

Using the decomposition $PT_P = T_1 + T_O$, we obtain:

$$E = \frac{T_1}{T_1 + T_O} = \frac{1}{1 + \frac{T_O}{T_1}}.$$

Therefore, efficiency depends solely on the ratio T_O/T_1:

- If $T_O \ll T_1$, then $E \approx 1$, and the parallel program is efficient.
- If T_O is comparable to or larger than T_1, efficiency drops.

2.4.3 Isoefficiency Concept

The **iso-efficiency** idea asks: if we want to maintain a fixed efficiency E as we increase the number of processes P, how must the problem size (and hence T_1) grow?

To keep efficiency constant, we require the ratio T_O/T_1 to remain constant:

$$\frac{T_O}{T_1} = K \quad \Rightarrow \quad T_O = KT_1.$$

In practice, we:

1. Express useful work T_1 as a function of problem size N and processes P.
2. Express overhead T_O (often dominated by communication) as a function of N and P.
3. Solve $T_O = KT_1$ for the relationship between N and P.

The resulting $N(P)$ is the iso-efficiency function. Algorithms with slower growing $N(P)$ (e.g., $N = \Theta(P)$) are more scalable than those requiring faster growth (e.g., $N = \Theta(P^2)$).

2.5 Isoefficiency Example: 2D Stencil / Game of Life

To illustrate iso-efficiency, we use a 2D stencil / Game of Life computation on a grid of N elements, assumed to be arranged as $\sqrt{N} \times \sqrt{N}$. Two decompositions are considered: 1D and 2D.

In both decompositions, each process owns N/P elements, so the computation per process is proportional to N/P. The difference lies in the communication overhead.

2.5.1 1D Decomposition

In the 1D decomposition:

- Processes are arranged logically in a line.
- Each process owns $\sqrt{N}$ columns and $\sqrt{N}/P$ rows, so it holds $(\sqrt{N}/P) \cdot \sqrt{N} = N/P$ elements.
- To update boundary cells, each process exchanges halo rows with its neighbors above and below.

Communication per process:

- Two halo rows are sent (one up, one down), each of length $\sqrt{N}$.
- Thus the communication volume per process is proportional to $2\sqrt{N}$.

From the point of view of a single process:

- Useful work is $T_1' \propto N/P$.
- Overhead is $T_O' \propto 2\sqrt{N}$.

The ratio is:

$$\frac{T_O'}{T_1'} \propto \frac{2\sqrt{N}}{N/P} = \frac{2P}{\sqrt{N}}.$$

To maintain constant efficiency, we require:

$$\frac{2P}{\sqrt{N}} = K \quad \Rightarrow \quad P \propto \sqrt{N} \quad \Rightarrow \quad N \propto P^2.$$

Thus, for a 1D decomposition, the problem size must grow quadratically in P to keep efficiency constant. This is a relatively weak scalability property.

2.5.2 2D Decomposition

In the 2D decomposition:

- Processes are arranged in a $\sqrt{P} \times \sqrt{P}$ grid.
- Each process owns a sub-block of size $(\sqrt{N}/\sqrt{P}) \times (\sqrt{N}/\sqrt{P})$.
- The number of elements per process is $(\sqrt{N}/\sqrt{P})^2 = N/P$.

Each process now has up to four neighbors (up, down, left, right). For a halo exchange:

- Each boundary has length $\sqrt{N}/\sqrt{P}$.

- Each process sends four such boundaries.
- Therefore, communication volume per process is proportional to $4 \cdot \frac{\sqrt{N}}{\sqrt{P}}$.

Again, from one process's perspective:

- Useful work is $T_1' \propto N/P$.
- Overhead is $T_O' \propto 4\frac{\sqrt{N}}{\sqrt{P}}$.

The ratio is:

$$\frac{T_O'}{T_1'} \propto \frac{4\frac{\sqrt{N}}{\sqrt{P}}}{N/P} = 4 \cdot \frac{\sqrt{N}}{\sqrt{P}} \cdot \frac{P}{N} = 4 \cdot \frac{\sqrt{P}}{\sqrt{N}}.$$

To maintain constant efficiency:

$$4 \cdot \frac{\sqrt{P}}{\sqrt{N}} = K \quad \Rightarrow \quad \sqrt{N} \propto \sqrt{P} \quad \Rightarrow \quad N \propto P.$$

For the 2D decomposition, problem size only needs to grow linearly with P to maintain efficiency. This is significantly more scalable than the 1D case.

2.5.3 Interpretation

The comparison shows that 2D decomposition is more isoefficient and thus more scalable than 1D decomposition. In practice:

- For toy problems (e.g., Game of Life), it may be easy to increase problem sizes.
- For real scientific simulations, growing N may be constrained by physics, memory, or wall-clock time.

In such settings, choosing parallel decompositions and algorithms with better iso-efficiency is crucial for achieving good scalability.

2.6 Calling Context Trees, Call Graphs, and Profiling

After discussing analytical models, the lecture turns to practical performance analysis based on profiles collected during program execution.

2.6.1 Calling Context and CCTs

The **calling context** of a function is the path of calls from the root of execution (e.g., `main`) to that function. For example:

$$\texttt{main} \rightarrow \texttt{foo} \rightarrow \texttt{bar}$$

is one calling context for `bar`.

By collecting all call paths observed during a run, a profiling tool can build a **calling context tree (CCT)**:

- Each node corresponds to a function in a specific calling context.
- Edges represent caller–callee relationships.
- Each node can store metrics such as time, hardware counters, or call counts.

2.6.2 Call Graphs

A **call graph** is obtained by merging all CCT nodes that represent the same function into one node:

- There is a single node for each function (e.g., one node for `bar`).
- Edges indicate which functions call which others (e.g., `foo` calls `bar`, `third` calls `bar`, etc.).

This representation loses calling-context information. For instance, it no longer distinguishes between time spent in `bar` when called from `foo` versus `third`. However, it is more compact and may be easier to analyze in some workflows.

2.6.3 Flat Profiles vs. CCTs vs. Call Graphs

Profiling tools can present performance data at different levels:

- A **flat profile** aggregates time per function with no calling context.

- A **call graph** preserves caller–callee relationships but merges contexts.
- A **CCT** preserves full calling context and is often the most detailed (and largest) representation.

In parallel programs, each node may also be indexed by process or thread, so a node may carry a vector of metrics across ranks. This can lead to high dimensionality: function $\times$ calling context $\times$ process $\times$ thread $\times$ metric.

2.7 Programmatic Analysis with Hatchet and Pandas

Many HPC profiling tools (HPCToolkit, Caliper, TAU, etc.) provide GUI-based analysis. However, GUIs are limited to built-in operations and can make it hard to automate complex analyses. To address this, we utilize **Hatchet**, a Python-based library for programmatic performance analysis.

2.7.1 GraphFrame: Graph + DataFrame

Hatchet's core abstraction is the **GraphFrame**, which combines:

- A graph structure (CCT or call graph) representing the calling relationships.
- A Pandas DataFrame holding metrics and metadata for each node (and optionally each rank).

Typical usage:

- Import Hatchet in Python (e.g., `import hatchet as ht`).
- Use a constructor like `ht.GraphFrame.from_hpctoolkit()` to load an HPCToolkit performance database (e.g., `hpctoolkit-calculatepi-database`).
- Access the DataFrame via `gf.dataframe` and the tree or graph via helper methods such as `gf.tree()`.

The DataFrame typically uses a multi-index:

- The first index level identifies the node.
- The second index level may be the MPI rank.

Each row stores inclusive and exclusive time (e.g., `time.inc`, `time.exc`) and other metadata such as function name, file, and line number (when available).

2.7.2 Filtering and Squashing GraphFrames

Hatchet extends Pandas with operations that keep the graph and DataFrame in sync:

Filter applies a predicate to the DataFrame and keeps only rows that satisfy the condition (e.g., nodes whose exclusive time exceeds a threshold). Hatchet returns a new GraphFrame; the original remains unchanged.

Squash removes filtered-out nodes from the graph and rewires the remaining nodes so that parents are connected directly to children that survived filtering.

Older versions required explicitly calling `squash()` after filtering. Recent versions perform the squashing automatically as part of `filter`, so the returned GraphFrame has a consistent graph and DataFrame.

2.7.3 Unify and Subtracting Executions

To compare two runs of the same program, one can subtract their profiles. For example, given GraphFrames `gf1` and `gf2`, one can compute differences via `gf2 - gf1`.

However, the two graphs may differ structurally if runs follow different call paths. Hatchet addresses this by **unifying** the graphs:

- It constructs the union of nodes and edges from both graphs.
- For nodes that exist in only one run, it creates dummy nodes in the other run with zero time.
- Once both GraphFrames share the same structure, Hatchet subtracts metrics node by node.

For multi-process data, it is often convenient to aggregate across ranks before subtracting runs. The function `drop_index_levels` can:

- Collapse the rank dimension, using an aggregation such as mean, max, or sum.
- Produce a DataFrame where each node has a single time value for each run.

2.7.4 Visualization Utilities

Hatchet includes basic visualization tools:

- A textual tree view showing the call tree with metrics, often using colors (e.g., red for slower, green for faster).
- Export to GraphViz DOT format for generating static graphs using GraphViz.
- Support for generating data for flame graphs, which can be rendered by external tools.

These visualizations are most useful after the data has been filtered and reduced; Hatchet is not intended as a full-fledged visualization environment.

2.7.5 Higher-Level Analyses: Chopper

Building on Hatchet, the **Chopper** API implements common multi-run analyses, including speedup and efficiency per function:

- Users supply a list of GraphFrames from runs at different process counts (e.g., 128, 256, 512, 1024 ranks).
- They specify whether the experiment is strong or weak scaling.
- Chopper computes speedup and efficiency per node (function).

This yields a DataFrame with columns for speedup and efficiency, which can be sorted or filtered. For instance, one might identify functions whose efficiency falls below a threshold and whose absolute time is large, making them good optimization targets.

2.7.6 Load Imbalance Analysis

Load imbalance is another important aspect of performance. Hatchet can compute a load imbalance metric per node:

$$\text{imbalance} = \frac{\max \text{ time across ranks}}{\text{mean time across ranks}}.$$

If this ratio is close to 1, the function is well balanced. Larger values indicate severe imbalance. Hatchet's `load_imbalance()` function:

- Computes this ratio and stores it as a new column in the DataFrame.
- Optionally, with `verbose=True`, reports which ranks are most overloaded and other distribution details.

2.7.7 Pandas Primer

Pandas is a widely used Python library for data analysis. Its core abstractions are:

- **Series**: one-dimensional labeled arrays.
- **DataFrame**: two-dimensional tables of rows and columns, built from Series objects.

Pandas supports operations reminiscent of SQL:

- Filtering rows and selecting columns.
- Grouping and aggregating.
- Joining and merging tables.

Hatchet leverages Pandas to let users treat profiling data as a table, while preserving the graph structure via the GraphFrame abstraction. This enables flexible, script-based analysis of complex parallel performance data.

Chapter 3

OpenMP Data Sharing and Scheduling

3.1 OpenMP Refresher and Shared-Memory Programming

After the Hatchet discussion, the lecture transitions to **OpenMP**, a shared-memory parallel programming model.

3.1.1 Shared-Memory Architecture and Race Conditions

On a typical compute node (such as those in the Zaratan cluster), memory is logically shared:

- Multiple cores, possibly organized into NUMA domains, all see the same physical address space.
- Although access latency can differ across NUMA domains, conceptually all cores can read and write any memory location.

In shared-memory programming models:

- Multiple threads share the same address space.
- Any thread can read or write any variable (unless explicitly made private).
- Communication is implicit through reads and writes to shared variables, rather than explicit message passing as in MPI.

This can make it easier to write the first parallel version of a loop, but it shifts responsibility for correctness to the programmer. The main hazards are **race conditions**, where two or more threads access the same data without proper synchronization and at least one access is a write. The final result then depends on the unpredictable thread scheduling order.

A correct parallel program must produce the same results regardless of how the operating system schedules threads.

3.1.2 What Is OpenMP?

OpenMP is a shared-memory programming model and runtime for C, C++, and Fortran. It consists of:

- **Compiler directives** (pragmas in C/C++) that annotate code with parallel constructs.
- **Runtime library routines** declared in `omp.h`, such as `omp_set_num_threads()` and `omp_get_num_procs()`.
- **Environment variables** such as `OMP_NUM_THREADS` and `OMP_SCHEDULE` for configuring runtime behavior.

A key design feature is that OpenMP directives are *advisory*:

- If you compile without OpenMP support, the compiler ignores the pragmas and produces a sequential program.
- If you compile with OpenMP enabled (e.g., using `-fopenmp` in GCC), the compiler generates code to create threads and manage parallel regions.

In modern clusters, OpenMP is typically used for on-node parallelism, while MPI handles distributed-memory parallelism across nodes.

3.1.3 Fork–Join Model and Pragmas

OpenMP follows a **fork–join** execution model:

1. Execution begins with a single master thread.
2. Upon encountering a parallel region, the master thread *forks* a team of worker threads.
3. All threads (including the master) execute the code inside the parallel region.
4. At the end of the region, the threads *join* and execution continues with a single thread.

In C/C++, OpenMP uses pragmas:

```
#pragma omp <construct> [clauses...]
```

The lecture highlights two important constructs:

- `#pragma omp parallel`: marks a parallel region; the immediately following statement or block is executed by all threads.
- `#pragma omp parallel for`: combines a parallel region with distribution of the following `for` loop's iterations across threads.

Only the *immediately following* code element is affected. If you have nested loops, `parallel for` applies only to the outer loop directly after the pragma. If you have multiple loops in sequence, each must be annotated separately if all should be parallelized.

3.1.4 Controlling the Number of Threads

There are two main mechanisms to control the number of OpenMP threads:

Environment variable. Set `OMP_NUM_THREADS` before running the program:

```
export OMP_NUM_THREADS=4
./prog
```

This chooses four threads for parallel regions unless overridden. The setting applies to that process and persists in the shell until changed.

Library routine. Call `omp_set_num_threads()` inside the program before a parallel region:

```
#include <omp.h>

int main() {
    omp_set_num_threads(8);
    #pragma omp parallel
    {
        ...
    }
}
```

The number of available cores can be queried using `omp_get_num_procs()`, which returns a value such as 64 or 128 depending on the node.

3.2 OpenMP Data Sharing: Shared, Private, and Reductions

3.2.1 Shared vs Thread-Private Variables

Unlike MPI processes, which have separate virtual address spaces, OpenMP threads share (almost) all of memory. By default:

- Most variables in the enclosing scope are **shared**.
- Consequently, multiple threads may read and write the same memory location, creating potential race conditions.

There are two important exceptions where OpenMP automatically provides thread-private behavior:

Loop indices. In a `parallel for` loop, the loop index variable (e.g., `i` in `for (i = 0; i < N; i++)`) is **thread-private by default**. Each thread has its own copy of the loop index. This is essential to assign different iterations to different threads without interference.

Local variables in called functions. When a parallel region calls a function, each thread receives its own stack frame for that function. Local (stack) variables in the function are therefore thread-private. If they were shared, threads would overwrite each other's local state when calling the same function concurrently.

3.2.2 The `private` Clause

The `private` clause allows us to explicitly make variables thread-private:

```
#pragma omp parallel for private(val, x, y)
for (i = 0; i < N; i++) {
    ...
}
```

Semantics:

- Each thread gets its own private copy of each `private` variable.
- The private copy is **uninitialized** on entry to the parallel region for that thread.
- The private variable's value at the end of the region is not copied back to the original (master) variable.

Consequences:

- You cannot rely on values assigned before the parallel region. For example, if `val = 5;` is set before the loop, then `private(val)` does *not* guarantee that each thread's private `val` begins at 5; they start uninitialized.
- You cannot rely on updates inside the region being reflected in the master variable after the region; the master retains its old value.

3.2.3 `firstprivate`: Copy In from the Master

The `firstprivate` clause behaves like `private`, but copies the initial value from the master thread:

```
int val = 5;

#pragma omp parallel for firstprivate(val)
for (i = 0; i < N; i++) {
    // each thread's val starts at 5
}
```

Each thread gets a private copy of `val` initialized to the value it had in the master just before entering the region. There is still no automatic copy-back at the end of the region.

3.2.4 `lastprivate`: Copy Out from the Last Iteration

The `lastprivate` clause copies a value from the *last* logical iteration of the loop (according to sequential order) back to the master thread:

```
int val;

#pragma omp parallel for lastprivate(val)
for (i = 0; i < N; i++) {
    val = i + 1;
}
// after the loop, master val has the value from i = N-1
```

Semantics:

- Each thread has a private copy of `val` during the loop.
- When the loop ends, OpenMP identifies the last iteration that would have been executed in a sequential run (e.g., `i = N-1`).
- The value of `val` from that iteration is copied back to the master thread's `val`.

The definition of "last" is based on iteration order, not on which thread finishes last in wall-clock time.

3.2.5 Reductions

Many loops compute scalar aggregates, such as sums or products. A simple sequential example:

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

Naively adding `#pragma omp parallel for` yields a data race:

```
int val = 0;
```

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
    val = val + i;  // race on val
}
```

Multiple threads read and write `val` concurrently, leading to nondeterministic results.

Conceptually, the correct parallelization is to give each thread a private partial sum and then add the partial sums together at the end. OpenMP's `reduction` clause automates this pattern:

```
int val = 0;

#pragma omp parallel for reduction(+: val)
for (i = 0; i < N; i++) {
    val = val + i;
}
// val is now the correct sum
```

For `reduction(op: var)`, OpenMP:

1. Creates a private copy of `var` for each thread.
2. Initializes the private copy according to `op`:
   - For `+`, `-`, bitwise `|`, and `^`, the initial value is 0.
   - For `*` and bitwise `&`, the initial value is 1.
   - For logical `&&`, the initial value is true; for logical `||`, it is false.
3. Lets each thread update its private copy in the loop via the specified operator.
4. At the end of the region, combines all private copies into the master variable using `op`, similar to an MPI `Reduce`.

The important takeaway is that `reduction` provides both thread-private accumulators and a safe, deterministic combination step.

3.2.6 Synchronization and Data Races

When variables are shared across threads without proper data-sharing clauses or synchronization, uncontrolled updates lead to data races and inconsistent results. The combination of:

- Data-sharing clauses (e.g., `private`, `firstprivate`, `lastprivate`, `reduction`), and
- Explicit synchronization constructs (e.g., `critical`, `atomic`, `barrier`, covered later)

allows the programmer to avoid races and reason about correctness.

3.3 OpenMP Loop Scheduling

3.3.1 Scheduling Kinds

OpenMP's loop **schedule** determines how iterations of a `parallel for` loop are mapped to threads. Different schedules balance simplicity, overhead, and load balance.

The general syntax is:

```
#pragma omp parallel for schedule(kind[, chunk])
for (i = 0; i < N; i++) {
    ...
}
```

The lecture focuses on:

Static scheduling. With `schedule(static[, chunk])`:

- Iterations are divided into chunks and assigned to threads before the loop begins.
- Without an explicit `chunk`, iterations are divided into contiguous blocks; each thread gets one block.

- With `chunk`, the iteration range is split into blocks of size `chunk` and assigned to threads in round-robin fashion.

For example, with 4 threads, 12 iterations (0–11), and `schedule(static, 1)`:

- Thread 0: 0, 4, 8
- Thread 1: 1, 5, 9
- Thread 2: 2, 6, 10
- Thread 3: 3, 7, 11

Dynamic scheduling. With `schedule(dynamic, chunk)`:

- Iterations are divided into chunks of size `chunk`.
- Threads dynamically grab the next available chunk from a shared pool as they finish their current chunk.
- This reduces load imbalance when iterations vary significantly in cost, at the expense of higher scheduling overhead.

Guided, runtime, and auto. The lecture also mentions:

- **Guided**: similar to dynamic but with decreasing chunk sizes over time (large chunks at the beginning, smaller chunks later).
- **Runtime**: the schedule is chosen at runtime based on the `OMP_SCHEDULE` environment variable.
- **Auto**: the compiler and runtime choose a schedule they believe to be good.

3.3.2 The `OMP_SCHEDULE` Environment Variable

Rather than hard-coding a schedule in the pragma, we can use the environment variable `OMP_SCHEDULE`:

```
export OMP_SCHEDULE="dynamic,4"
```

If no `schedule` clause is provided in the pragma, the runtime consults `OMP_SCHEDULE` to determine scheduling. This allows:

- Experimenting with different schedules without recompiling.
- Systematically exploring schedule choices in scripts.

If a schedule is explicitly specified in the pragma, it overrides `OMP_SCHEDULE` for that loop.

3.3.3 `parallel for` vs `parallel`

- `#pragma omp parallel for`: must be immediately followed by a `for` loop. OpenMP handles the division of iterations among threads.
- `#pragma omp parallel`: can precede any code block. Threads execute the block in parallel, but the programmer must manually divide work among them, often using thread IDs.

For array-heavy scientific codes, `parallel for` is usually more convenient and will be the primary construct used in the course. The more general `parallel` construct is closer in spirit to MPI, where work distribution is explicit.

3.4 OpenMP Examples: SAXPY and π Computation

3.4.1 Race-Free SAXPY

The SAXPY operation (Single-precision A·X + Y) computes:

$$Z[i] = A \cdot X[i] + Y[i].$$

A simple loop is:

```
for (i = 0; i < N; i++) {
    Z[i] = A * X[i] + Y[i];
}
```

Here, `X` and `Y` are read-only in the loop, and `Z` is write-only, with each iteration writing a different index `Z[i]`. There are no cross-iteration dependencies.

Adding `#pragma omp parallel for`:

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
    Z[i] = A * X[i] + Y[i];
}
```

is safe and results in a race-free parallel loop.

3.4.2 π Computation with OpenMP

The lecture revisits the classic π computation used earlier with MPI. The sequential version approximates π by numeric integration over many intervals. A typical loop is:

```
for (i = 0; i < N; i++) {
    x = ...;            // depends on i and step size h
    sum += f(x);        // accumulate contribution
}
```

In the OpenMP version, we must:

- Avoid races on `sum` by using a reduction.
- Avoid races on temporary variables like `x` by making them private.
- Decide whether shared read-only variables (such as `h`) need special handling.

An OpenMP-parallel version looks like:

```
#pragma omp parallel for private(x) reduction(+: sum) firstprivate(h)
for (i = 0; i < N; i++) {
    x = ...;           // uses i and h
    sum += f(x);
}
```

Here:

- `sum` is a reduction variable, avoiding a race and combining partial sums.
- `x` is private, so threads do not overwrite each other's temporary values.
- `h` is read-only and shared by default; using `firstprivate(h)` is mainly illustrative in this context.

3.5 OpenMP Offloading to GPUs

The lecture briefly introduces **OpenMP offloading** to GPUs, building on high-level differences between CPU and GPU architectures.

3.5.1 CPU vs GPU Architecture (High Level)

CPUs. Relatively few, powerful cores:

- Each core has a deep cache hierarchy (L1, L2, L3).
- Designed for complex, latency-sensitive workloads.

GPUs. Many lightweight cores:

- Thousands of cores, each individually weaker than a CPU core.
- Designed for throughput and massive parallelism.
- Have their own on-device memory and caches.
- Typically act as accelerators attached to CPUs.

Data for GPU computation usually resides initially in CPU memory; it must be copied to GPU memory, processed there, and copied back.

3.5.2 OpenMP Target Directives

OpenMP 4 and later introduce directives for offloading code to accelerators, including GPUs. A typical offloaded loop is:

```
#pragma omp target teams distribute parallel for ...
for (i = 0; i < N; i++) {
    ...
}
```

Here:

- `target` indicates that the enclosed region should run on an accelerator (e.g., a GPU).
- `teams distribute parallel for` creates teams of threads on the device and distributes loop iterations across those teams and threads.

Semantically, this is similar to `parallel for`, but the threads execute on GPU cores instead of CPU cores. The lecture notes:

- If a GPU is present and the compiler/runtime supports it, the region runs on the GPU.
- If no GPU is available, a typical outcome is an error or fallback behavior (implementation-dependent).
- Data mapping between host and device memory (via `map` clauses) is required but is discussed in more detail in the GPU portion of the course.

Chapter 4

Matrix Multiplication and Communication Algorithms

4.1 Parallel Algorithms and AI Relevance

Before diving deeper into matrix multiplication, the lecture highlights the connection between scientific computing and AI:

- Many course examples come from classical HPC, but the same concepts and primitives are central in AI and deep learning.
- Collective operations such as broadcast, reduce, and allgather are heavily used when training large models across multiple GPUs.
- Libraries like NVIDIA's NCCL and AMD's RCCL implement GPU-optimized collectives with semantics similar to MPI collectives.
- Matrix multiplication is a central building block in both domains:
  - In scientific computing: used in solvers, PDE discretizations, and many numerical methods.
  - In AI: neural network layers and tensor operations are dominated by matrix and tensor multiplications.

Thus performance improvements and parallelization techniques for matrix multiply directly benefit both HPC and AI workloads.

4.2 Matrix Multiplication: Basics and Sequential Implementation

4.2.1 Dimensions and Compatibility

Given matrices A, B, and C:

- A has size $M \times L$.
- B has size $L \times N$.
- Their product $C = A \times B$ has size $M \times N$.

The compatibility requirement is:

$$\text{cols}(A) = \text{rows}(B) = L.$$

If this condition is not satisfied, matrix multiplication is not defined and the program should report an error.

4.2.2 Elementwise Definition

The element C_{ij} (row i, column j) is defined as:

$$C_{ij} = \sum_{k=0}^{L-1} A_{ik} B_{kj}.$$

This is the inner product of row i of A with column j of B.

4.2.3 Standard Triple-Loop Implementation

The conventional triple-nested loop in C is:

```
for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++) {
        C[i][j] = 0;
        for (k = 0; k < L; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Here:

- `i` indexes rows of A and C.

- `j` indexes columns of B and C.
- `k` indexes the inner dimension L.

Note again that the second index of A and the first index of B share the summation index k.

4.3 Performance and Blocking in Matrix Multiplication

4.3.1 Cache Behavior and Performance Issues

For large matrices (e.g., $10{,}000 \times 10{,}000$), naive multiplication is slow not only because of the large number of arithmetic operations but also because of poor cache behavior.

Assuming row-major storage (as in C/C++):

- Elements of a row are stored contiguously in memory.
- Accessing `A[i][k]` for increasing `k` is fairly cache-friendly.
- Accessing `B[k][j]` sweeps down a column; successive elements are far apart in memory.

As we traverse long columns of B:

- Each access to `B[k][j]` brings in part of a row containing that element.
- When we move to `B[k+1][j]`, we likely load a different cache line and evict the previously loaded row segment.
- Later, when we move to a different column, we may have to reload elements of A and B that were previously evicted.

Thus the naive triple-loop implementation is often memory-bound and fails to exploit cache reuse, especially for matrix B.

4.3.2 Blocking (Tiling) as a Sequential Optimization

To improve cache behavior, we use **blocking** (tiling). Conceptually, we partition A, B, and C into smaller submatrices (tiles):

- Denote blocks by A_{pq}, B_{qr}, C_{pr}, where each symbol represents a submatrix rather than a scalar.
- Choose a block size B (e.g., 32×32) such that blocks of A and B fit nicely in cache.

At the block level, we have:

$$C_{p,r} = \sum_q A_{p,q} B_{q,r},$$

which mirrors the scalar definition but at the submatrix level.

To implement blocking, we add three outer loops over block indices:

```
for (ii = 0; ii < M; ii += B)          // block index in i dimension
  for (jj = 0; jj < N; jj += B)        // block index in j dimension
    for (kk = 0; kk < L; kk += B) {    // block index in k dimension

      // Multiply A[ii:ii+B, kk:kk+B] and B[kk:kk+B, jj:jj+B]
      // Accumulate into C[ii:ii+B, jj:jj+B]

      for (i = ii; i < min(ii+B, M); i++)
        for (j = jj; j < min(jj+B, N); j++)
          for (k = kk; k < min(kk+B, L); k++)
            C[i][j] += A[i][k] * B[k][j];
    }
```

Blocking improves performance by:

- Restricting computation to small portions of A and B that can remain in cache.
- Allowing rows of A and columns of B within a block to be reused many times before eviction.
- Particularly helping the access pattern for columns of B, which is problematic in naive row-major implementations.

Blocking does not change the asymptotic arithmetic cost ($O(n^3)$), but it significantly improves cache efficiency in practice. Real BLAS libraries heavily rely on such blocking for high performance.

4.4 Parallel Matrix Multiplication: Cannon and Agarwal

The lecture then extends blocking concepts to parallel matrix multiplication on distributed-memory systems.

4.4.1 General Setup for Parallel Matrix Multiply

In a distributed-memory environment (e.g., MPI):

- Matrices A and B may be too large to fit in a single process's memory, or computation may be too slow if done on one process.
- The matrices are partitioned, and each process holds a portion of A, B, and C.
- Each process is responsible for computing disjoint subblocks of C (no overlap between processes).
- Because each C_{ij} depends on an entire row of A and an entire column of B, processes must communicate subblocks of A and B so that each process obtains the blocks it needs.

Two classical algorithms are discussed: Cannon's 2D algorithm and Agarwal's 3D algorithm.

4.4.2 Cannon's 2D Algorithm

Process Grid and Data Distribution

Cannon's algorithm arranges P processes in a 2D grid:

$$\sqrt{P} \times \sqrt{P}.$$

Matrices A, B, and C are partitioned into compatible $\sqrt{P} \times \sqrt{P}$ grids of subblocks. Each process initially holds one subblock of A and one subblock of B; process (i, j) is ultimately responsible for computing subblock C_{ij}.

However, the initial pairing of A and B subblocks on each process does not necessarily correspond to the pairings required for computing its C_{ij}.

Initial Skew (Displacement)

To align pairings, Cannon's algorithm performs an initial skew:

- For matrix A: in each row i, blocks of A are circularly shifted left by i positions.
- For matrix B: in each column j, blocks of B are circularly shifted upward by j positions.

This yields a configuration where each process (i, j) has a pair $(A_{i,k}, B_{k,j})$ that can be multiplied to contribute to C_{ij} for some k.

Iterative Shifts and Partial Products

Once properly skewed, each process multiplies its local A and B blocks and accumulates the result into its local C_{ij} block. To obtain the remaining terms in the sum over k:

- In each subsequent phase, all rows of A are shifted left by one position (circularly), and all columns of B are shifted up by one position.
- After each shift, each process obtains a new pair of blocks suited for the next k in the summation and multiplies and accumulates again.

After $\sqrt{P}$ phases (including the initial configuration), each process has computed all partial products required for its C_{ij} block.

Complexity and Trade-offs

The total arithmetic work remains $O(n^3)$ across all processes, as no redundant computation is performed. For communication:

- There is an initial skew phase (one shift of A and B).
- There are $\sqrt{P} - 1$ additional phases, each shifting A and B along rows and columns.

Each phase requires sending and receiving blocks of size proportional to the square of the submatrix dimension. Thus, the communication grows with both the number of processes and the block size. Cannon's algorithm uses memory efficiently (each block of A and B is stored on exactly one process) but requires multiple communication phases.

4.4.3 Agarwal's 3D Algorithm

3D Process Grid and Replication

Agarwal's 3D algorithm arranges processes in a 3D grid:

$$p_i \times p_j \times p_k.$$

The example in the slides uses an $3 \times 3 \times 2$ grid, totaling 18 processes. The algorithm introduces replication of blocks:

- One plane (e.g., an I-K plane at fixed j) is used to place unique subblocks of A. Each subblock is assigned to one process in that plane.
- Those A blocks are then broadcast along the remaining dimension (the j direction) so that every process with the same (i, k) coordinates receives the same A block.
- Similarly, another plane (e.g., a J-K plane at fixed i) is used to place unique subblocks of B, which are broadcast along the i dimension so that all processes with the same (j, k) share the same B block.

In the final layout, each process holds one A tile and one B tile, but these tiles are replicated across the 3D grid.

Local Computation and Reduction

Once data is distributed:

- Each process multiplies its local A and B tiles, producing a single partial product contributing to some C_{ij} block.
- For each pair (i, j), the partial products reside on the processes differing in their k coordinate. These processes perform a reduction (e.g., MPI_Reduce or MPI_Allreduce) along the k dimension to sum their partial results and obtain the final C_{ij} tile.

Thus, after initial broadcasts of A and B and a final reduction, no further communication is needed during core computation.

Comparison with Cannon's Algorithm

Key differences:

- **Memory usage**: In Cannon's algorithm, each block of A and B is stored exactly once. In Agarwal's algorithm, blocks of A and B are replicated along different dimensions, increasing memory usage.
- **Communication pattern**: Cannon's algorithm requires multiple shift phases, while Agarwal's algorithm uses a small number of broadcasts and a final reduction. This can reduce communication volume and latency.
- **Generality**: The 3D algorithm can be used with many different process counts and grid shapes (e.g., $3 \times 3 \times 3$, $4 \times 4 \times 4$, etc.), adjusting the amount of replication and per-process work.

Generalization to Other Process Counts

The lecture emphasizes that numbers like 18 (for $3 \times 3 \times 2$) are not fixed limits; they are merely examples. For instance:

- With 27 processes, one might use a $3 \times 3 \times 3$ grid, adding a third plane along the replication dimension.
- With 64 processes, one might choose a $4 \times 4 \times 4$ grid, again replicating A and B along appropriate dimensions and reducing along the third dimension.

In symmetric grids, the parameter k can be thought of as the number of processes per dimension, analogous to the branching factor in a k-ary tree.

4.5 Collective Communication Algorithms: Reduction and All-to-All

The lecture then shifts to general communication algorithms in MPI, focusing on reduction and all-to-all collectives.

4.5.1 Reduction: Scalar and Vector

A reduction operation combines values from all processes using an associative and commutative operation (e.g., sum, min, max):

- **Scalar reduction**: each process contributes a single value, such as a local timing. The collective combines these into a single value at a root.
- **Vector reduction**: each process contributes an array of values. The reduction operation is applied element-wise across the arrays.

From MPI's perspective, both scalar and vector reductions use the same interface; only the count and datatype differ.

4.5.2 Naive Reduction Algorithm

A naive implementation of MPI_Reduce is:

- Every non-root process sends its value (or vector) directly to the root.
- The root receives all messages and performs all the reductions.

Problems:

- The root becomes a major **communication bottleneck**, receiving messages from all processes.
- The root is also a **computation bottleneck**, performing all reduction operations, while other processes do very little.
- The scheme does not scale well as the number of processes grows.

4.5.3 Tree-Based Reduction (Spanning Tree)

A more scalable approach uses a spanning tree:

- Processes are arranged logically in a tree (binary, k-ary, etc.).
- Leaves send their values to their parents.
- Each internal node waits for values from its children, combines them with its own local value, and sends the result to its parent.
- This continues level by level until the root has the final result.

For a binary tree:

- The height is approximately $\log_2 P$.
- Each phase corresponds to one level in the tree; messages move only one level per phase.

For a k-ary tree, the height is roughly $\log_k P$. Using higher branching factors reduces the number of levels (phases) but increases per-process work and per-phase communication.

This tree-based approach:

- Balances communication load; the root receives data from only a few children rather than from all processes.
- Balances computation; internal nodes perform similar amounts of reduction work as the root.
- Scales much better than the naive scheme.

The same structure applies to vector reductions, where entire arrays are reduced at each node.

4.5.4 MPI_Alltoall

MPI_Alltoall is a collective that performs a global exchange of data such that every process sends potentially distinct data to every other process and receives data from every other process.

Semantics

With P processes:

- Each process's send buffer is conceptually divided into P segments.
- Segment j in process i's buffer is destined for process j.
- After MPI_Alltoall:
  - Process 0 holds all data destined for rank 0 (one segment from each process).
  - Process 1 holds all data destined for rank 1, and so on.

This is analogous to transposing a logical $P \times P$ matrix of data segments.

Naive Algorithm

A naive implementation:

- Each process sends a message to every other process.
- Each process receives a message from every other process.

With P processes, this yields on the order of $P(P-1) = O(P^2)$ messages, creating severe network congestion and overhead for large P.

Optimized Algorithm Using a 2D Process Grid

To reduce the number of messages, we arrange the P processes in a $\sqrt{P} \times \sqrt{P}$ virtual grid (assuming P is a perfect square for simplicity) and perform two structured phases:

Phase 1: Row Phase.

- Each process sends data destined for processes in its *column* to all processes in its *row*.
- Each process sends $\sqrt{P} - 1$ messages (one to each other process in the row).
- After this phase, each process holds all data destined for processes in its column (aggregated from its row peers).

Phase 2: Column Phase.

- Each process sends its aggregated data to all processes in its column.
- Again, each process sends $\sqrt{P} - 1$ messages (one to each other process in the column).
- After this phase, each process has received all data intended for it from every process in the system.

A conceptual barrier separates the two phases: processes must complete row communication before beginning column communication, so that they have all the data for their column targets.

Complexity Comparison.

- **Naive algorithm**: each process sends $P - 1$ messages; total messages are $O(P^2)$.
- **2D grid algorithm**: each process sends $\sqrt{P} - 1$ messages per phase; there are two phases. Across all processes, each phase uses $P(\sqrt{P} - 1) = O(P^{3/2})$ messages.

Thus, the optimized algorithm reduces the number of messages from $O(P^2)$ to $O(P^{3/2})$, significantly improving scalability and reducing network congestion.

Chapter 5

GPGPU and CUDA Introduction

5.1 Why GPUs Are "Hot" Right Now

There are three major drivers behind the current boom in GPUs:

5.1.1 Scientific and HPC Workloads

Traditional scientific and high-performance computing applications often operate on large arrays and apply the same operation across many elements. These data-parallel patterns, such as vector updates, stencil computations, and dense linear algebra, map naturally onto GPUs, which can perform the same operation on many data elements in parallel.

5.1.2 Cryptocurrency Mining

During the cryptocurrency mining surge, particularly around the COVID era, GPUs were heavily used for mining due to their high throughput in certain integer and hash-based workloads. This demand contributed to GPU shortages and significant price increases, affecting both gamers and researchers.

5.1.3 AI Training and Matrix Multiplication

Modern deep learning training workloads are dominated by matrix multiplications. Large models, especially Transformers, rely heavily on general matrix multiply (GEMM) operations in both the attention and feed-forward components of each encoder or decoder block.

In a typical Transformer encoder block:

- The attention mechanism uses matrix multiplications to compute queries, keys, and values, and to combine attention weights with value vectors.
- The feed-forward network uses matrix multiplications to transform hidden representations.
- During training, each layer performs matrix multiplications in the forward pass and multiple matrix multiplications in the backward pass (often described informally as "one in forward, two in backward").

As a result, overall training is "mostly GEMM." GPUs are extremely efficient at GEMM because they provide massive throughput for regular, parallel floating-point operations.

The lecture also notes a trend in numerical precision:

- Scientific computing often uses 32-bit floating point (FP32) or 64-bit floating point (FP64) for accuracy.
- AI training increasingly uses mixed precision with FP16 or BF16, especially on tensor cores designed to accelerate low-precision matrix multiplications.

5.2 GPU Hardware: How It Differs from CPUs

5.2.1 CPU Overview

CPUs have a relatively small number of powerful cores. Each core:

- Operates at a relatively high clock frequency.
- Has deep cache hierarchies (private L1 and often L2, with shared L3).
- Is optimized for low-latency execution and complex control flow, including branch-heavy and irregular workloads.

5.2.2 GPU Overview

GPUs, in contrast, contain many more cores, each of which is simpler:

- A GPU can have tens of thousands of "CUDA cores" (depending on the definition and the generation).
- GPU cores are individually less powerful and operate at lower clock speeds than CPU cores.
- GPUs trade single-thread latency and per-core sophistication for massive throughput via concurrency.

There are two key differences:

Core count CPUs may have tens to hundreds of cores, whereas GPUs can expose tens of thousands of lanes of parallelism.

Clock speed GPU clock frequencies are typically lower, in part due to power and thermal constraints. Packing many cores into a chip increases heat density, limiting achievable frequency.

5.3 Inside an NVIDIA GPU: SMs and Specialized Units

The lecture uses a Volta-era NVIDIA GPU (e.g., V100) as a concrete example and describes its internal organization.

5.3.1 Streaming Multiprocessors (SMs)

A GPU is composed of multiple **Streaming Multiprocessors (SMs)**. Each SM is a workhorse unit that contains multiple execution resources and manages many threads concurrently.

Within an SM, there are specialized units, including:

- **FP32 cores** for single-precision floating-point arithmetic.
- **FP64 cores** for double-precision floating-point arithmetic, which are important for many HPC and scientific workloads.
- **INT32 cores** for integer operations.
- **Tensor cores** designed for efficient mixed-precision matrix operations, crucial for AI workloads.

The discussion notes that GPU vendors have been allocating more silicon area to tensor and mixed-precision capabilities due to the enormous demand from AI. This can create tension with the HPC community, which often relies on high double-precision performance.

5.4 GPUs in a Compute Node: Host + Device Model

A typical node contains:

- One or more **CPUs** (the *hosts*).
- Several **GPUs** (the *devices*).
- High-bandwidth interconnects (e.g., NVLink) between CPUs and GPUs and often between GPUs themselves within the node.

Different systems may have different numbers of GPUs per node (e.g., 4 GPUs, 8 GPUs, etc.), but the conceptual model is the same:

- The CPU executes the main program, manages memory allocations on the GPU, and launches kernels.
- The GPU executes highly parallel kernels on data stored in its own device memory.

This leads to the standard CUDA terminology:

Host The CPU and its memory.
Device The GPU and its memory.

5.5 CUDA Programming Model: Threads, Blocks, and Grids

CUDA exposes a hierarchy of parallel execution:

- A **thread** is the smallest execution unit, representing a single serial control flow in the kernel.
- A **block** is a group of threads that execute on the same SM and can cooperate via shared memory and synchronization primitives.
- A **grid** is a collection of blocks launched for a particular kernel invocation.

This hierarchy maps onto hardware as follows (conceptually):

- A CUDA *thread* runs on a CUDA core.
- A thread *block* is scheduled onto a single SM; multiple blocks may reside on the same SM concurrently.
- A *grid* spans the entire GPU, utilizing blocks across all SMs.

CUDA places more responsibility on the programmer compared to, say, OpenMP. The programmer must decide:

- How many threads to launch per block.
- How many blocks to launch in the grid.

- How threads compute their indices and determine which data elements they operate on.

The lecture notes a common constraint: on many NVIDIA architectures, the maximum number of threads per block is 1024. This is a hardware constraint and must be respected when designing kernels.

5.6 CUDA Workflow: Memory, Kernel, Memory

A basic CUDA program typically follows a three-step pattern:

1. Allocate memory on the host (CPU) and the device (GPU) and copy input data from host to device.
2. Launch one or more kernels on the device to perform computation.
3. Copy results from device back to host and free memory on both sides.

Because CPU and GPU memories are separate in the traditional model, explicit memory management and data movement are essential.

5.6.1 Memory Allocation and Copying

The common pattern is:

- Allocate host memory (e.g., with `malloc`) and initialize it, potentially by reading from files or generating data.
- Allocate device memory using `cudaMalloc`.
- Copy input data from host to device using `cudaMemcpy` with `cudaMemcpyHostToDevice`.
- After kernel execution, copy result data from device to host using `cudaMemcpy` with `cudaMemcpyDeviceToHost`.
- Free device memory with `cudaFree` and free host memory with `free`.

```
float *h_A, *d_A;
size_t size = N * sizeof(float);

// Allocate and initialize host memory
h_A = (float*) malloc(size);
initialize(h_A, N);

// Allocate device memory
cudaMalloc((void**)&d_A, size);

// Copy host -> device
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);

// ... launch kernels that use d_A ...

// Copy device -> host
cudaMemcpy(h_A, d_A, size, cudaMemcpyDeviceToHost);

// Free memory
cudaFree(d_A);
free(h_A);
```

Listing 5.1: Simplified host-side CUDA memory management

A common point of confusion is that the device pointer (e.g., `d_A`) is stored in a CPU variable and is therefore visible in host code, but it refers to an address in GPU memory. You generally do not dereference such pointers on the CPU; instead, you pass them to CUDA runtime calls and kernel launches so that the GPU can use them.

5.6.2 Copy Directions

`cudaMemcpy` supports multiple directions:

- Host to device (`cudaMemcpyHostToDevice`).
- Device to host (`cudaMemcpyDeviceToHost`).
- Device to device (`cudaMemcpyDeviceToDevice`).

Device-to-device copies can be used to move data between GPUs without staging through host memory, depending on the system configuration.

5.7 Kernels and Kernel Launches: SAXPY Example

A CUDA **kernel** is a function that runs on the GPU. In code, kernels are typically annotated with the `__global__` qualifier and are invoked from the host using a special launch syntax.

5.7.1 SAXPY as an Example

The lecture uses SAXPY (Single-precision A·X + Y) as a running example. The mathematical operation is:

$$Y[i] = \alpha \cdot X[i] + Y[i]$$

for each index i.

A simple CUDA kernel for SAXPY might look like:

```
__global__ void saxpy_kernel(int N, float alpha, const float *X, float *Y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        Y[i] = alpha * X[i] + Y[i];
    }
}
```

Listing 5.2: Simple SAXPY kernel

5.7.2 Kernel Launch Syntax

From the host, the kernel is launched using the `<<<...>>>` syntax:

```
int threadsPerBlock = 256;
int numBlocks       = (N + threadsPerBlock - 1) / threadsPerBlock;

saxpy_kernel<<<numBlocks, threadsPerBlock>>>(N, alpha, d_X, d_Y);
```

Listing 5.3: Kernel launch from host

The two launch parameters specify:

- The number of *blocks* in the grid (`numBlocks`).
- The number of *threads* per block (`threadsPerBlock`).

Every thread executes the same kernel code (SPMD style), and the thread's unique index determines which data element(s) it processes.

5.8 Handling Large Arrays: Global Indexing

In practice, arrays often have many more elements than the maximum number of threads in a single block. CUDA addresses this by allowing multiple blocks in a grid. Each thread computes a **global index** based on its block and thread indices.

CUDA provides built-in variables:

- `threadIdx.x`: thread index within its block.
- `blockIdx.x`: block index within the grid.
- `blockDim.x`: number of threads per block.

The usual formula for the global index in a 1D configuration is:

$$i = \texttt{blockIdx.x} \times \texttt{blockDim.x} + \texttt{threadIdx.x}.$$

Inside the kernel, each thread computes this index and operates on element `i`, for example `X[i]` and `Y[i]`.

5.8.1 Bounds Checking

When the grid configuration results in more threads than array elements, some threads will have `i >= N`. To avoid out-of-bounds access, kernels should guard their work:

```
int i = blockIdx.x * blockDim.x + threadIdx.x;

if (i < N) {
    Y[i] = alpha * X[i] + Y[i];
}
```

Listing 5.4: Bounds checking in kernel

This pattern allows launching a convenient number of blocks and threads (often computed as $(N + \texttt{threadsPerBlock} - 1)/\texttt{threadsPerBlock}$) while ensuring safety.

If N is very small, launching many threads is inefficient because most threads do no work. For realistic workloads, N is often large enough that this overhead is negligible relative to kernel computation.

5.9 Compilation with NVCC

CUDA code is usually placed in files with a `.cu` extension and compiled with NVIDIA's CUDA compiler, `nvcc`. A typical compilation command is:

```
nvcc -arch=sm_80 -o myprog myprog.cu
```

The `-arch` flag specifies the target GPU architecture. For example:

- Volta GPUs might use an `sm_70`-class architecture.
- Ampere GPUs (such as A100) might use `sm_80`.
- Newer generations (Hopper, Blackwell, etc.) have their own `sm_XX`.

On a system like Zaratan, which has A100 (Ampere) GPUs, specifying an `sm_80` architecture is common to ensure that the generated code is optimized for the hardware.

5.10 Discussion Points and Common Questions

The lecture addresses several frequently asked questions:

5.10.1 Why the 1024 Threads-Per-Block Limit?

There is a hardware-imposed maximum on the number of threads per block, commonly 1024 on many architectures. This limit is tied to architectural constraints such as resource allocation per block (registers, shared memory) and hardware scheduling considerations.

5.10.2 What if Threads Outnumber Elements?

If the total number of launched threads exceeds the number of elements (N), threads use a bounds check (`if (i < N)`) and simply perform no work when their index is out of range. This is a safe and common pattern.

5.10.3 What if Elements Outnumber Threads?

If N is larger than the total number of threads in the grid, one can:

- Increase the number of blocks so that $\texttt{numBlocks} \times \texttt{threadsPerBlock} \geq N$, or
- Use a striding pattern where each thread processes multiple elements, e.g., `for (int i = globalIdx; i < N; i += stride)` inside the kernel.

5.10.4 Can Multiple Kernels Run at the Same Time?

Modern GPUs can overlap kernel execution using **streams** and asynchronous execution. The lecture mentions this only briefly and treats it as an advanced topic that will be covered later. For this introductory lecture, the focus remains on single-kernel execution and basic memory operations.

Chapter 6

Performance Issues and Variability

6.1 What "Performance" Means in HPC

The lecture begins by clarifying what people usually mean when they talk about performance in high-performance computing. The most basic and widely used metric is **wall-clock execution time**, also called **time to solution**. This is simply how long a program takes, from start to finish, to complete its scientific task—for example, running a molecular dynamics simulation to completion or producing a Fourier transform result.

Because many real HPC applications can run for hours, days, or even months, it is often impractical to time an entire production run just for performance studies. Instead, iterative codes frequently report **time per step** (or time per iteration). In this approach:

- The program is run for a small number of iterations (e.g., 10).
- The total time is measured.
- An average time per iteration is computed and reported.

This provides a tractable, repeatable performance metric that approximates production behavior without requiring extremely long runs.

The lecture also notes that some communities prefer **science-facing figures of merit** instead of raw execution time. In these cases:

- A scientific quantity is divided by computer time to yield a more intuitive metric.
- Examples include "simulated days of epidemic spread per second of runtime" or "simulated nanoseconds of molecular time per hour of compute."

This reframes performance as *how much scientific progress you achieve per unit of compute time*.

For compute-heavy codes, performance is sometimes expressed as **floating-point operations per second (FLOP/s)**. A higher FLOP/s value can indicate better performance, but only if the application is actually compute-bound. If it is limited by memory bandwidth or latency, FLOP/s may not be the most useful metric.

6.2 Speedup, Efficiency, and Scalability

The lecture connects basic performance metrics to scalability analysis through **speedup** and **efficiency**. By running a program with different numbers of threads or processes and measuring performance in each case, we can compute:

- Speedup, typically defined as T_1/T_P, where T_1 is the execution time on one process and T_P is the execution time on P processes.
- Efficiency, defined as speedup divided by the number of processes, $E = S/P$.

Plotting performance versus parallelism (e.g., speedup vs. number of processes) helps diagnose whether a code scales well. This empirical approach complements earlier analytical tools such as:

- Amdahl-style reasoning, which highlights the impact of serial fractions.
- Asymptotic complexity and isoefficiency analysis, which explain how problem size must grow with process count to maintain efficiency.

6.3 Peak Performance and "Is My Performance Good?"

A central question in performance analysis is whether observed performance is close to what the hardware can theoretically deliver. A common practice is to compare measured performance to **peak advertised performance**, the vendor's theoretical maximum (for example, in TFLOP/s). These peak numbers serve as an upper bound, but are usually achieved only by highly specialized benchmarks that:

- Minimize data movement.
- Maximize repeated computation on data that stays in cache or registers.

The lecture emphasizes that peak values are often *optimistic* relative to real applications. For many scientific codes:

- Achieving $\sim 20\% - 40\%$ of peak can be quite reasonable.

- Some deep learning workloads dominated by dense matrix multiplies can sometimes reach $\sim 60\% - 80\%$ of peak on a single accelerator.

However, as parallelism increases across multiple GPUs and nodes, the *overall* measured FLOP/s often drops because the timing includes communication, synchronization, and other overheads, not just raw computation.

To illustrate the gap between peak and achieved performance at the system level, the lecture references the **TOP500** list, which uses the HPL/LINPACK benchmark (LU factorization) to rank supercomputers. Systems usually report:

- A theoretical peak number.
- An achieved number (often called R_{max}).

The ratio R_{max}/peak varies by system and architecture, reinforcing that peak numbers are helpful as a reference but not a guarantee.

6.4 Where Time Goes Inside Programs

Before diving into parallel performance issues, the lecture reviews the basic components that consume time in *any* executable. Programs spend time in:

- Integer operations (loop counters, indexing, pointer arithmetic).
- Floating-point operations (additions, multiplications, divisions, transcendental functions).
- Branching and conditional behavior (e.g., `if/else` logic, termination tests, convergence checks).
- Loads and stores, which move data between main memory, caches, and registers.

Even in parallel programs, many performance problems originate in these sequential behaviors. For this reason, developers often optimize single-core or single-process performance before scaling out.

In parallel settings, additional sources of time appear due to **data movement beyond the local core**, including:

- CPU–GPU transfers and GPU–GPU transfers over PCIe, NVLink, or other links.
- Node-to-node communication over the interconnect.
- I/O to shared storage systems and file systems.

6.5 A Taxonomy of Performance Problems in Parallel Programs

The lecture presents a broad classification of performance problems commonly found in parallel codes.

6.5.1 Serial Code Performance Issues

These are inefficiencies that exist even in the sequential version of the code:

- Poor cache utilization and unnecessary data movement.
- Using expensive arithmetic operations where cheaper alternatives suffice.
- Unvectorized inner loops or function-call overhead in hot paths.

6.5.2 Load Imbalance

Load imbalance occurs when different processes or threads perform different amounts of work. Faster ranks reach synchronization points earlier and must wait for slower ones, reducing overall parallel efficiency. Load imbalance can arise from:

- Unequal data partitioning.
- Algorithmic asymmetry (e.g., some regions are harder than others).
- Input-dependent behavior.

6.5.3 Communication and Parallel Overheads

As processes are added, communication and synchronization costs grow. A program may scale well at small process counts but degrade at larger counts because:

- Message counts increase.
- Collective operations become more expensive.

- Per-message startup costs dominate when many small messages are used.

6.5.4 Replicated Work (Algorithmic Overhead)

Parallelization sometimes introduces extra work that is not present in a strictly sequential algorithm. Examples include:

- Redundant calculations in parallel prefix sums.
- Extra ghost-cell computations in domain decompositions.

In some cases this overhead is necessary to achieve parallelism; in others, it may be accidental duplication.

6.5.5 Speculative Work

Speculative work refers to computations performed in anticipation of potential future need. When speculation is correct, it can reduce latency. But when it is wrong, it wastes compute time and energy.

6.5.6 Critical Paths

Critical paths arise from dependency chains across processes or threads, where one step cannot proceed until data from a previous step arrives. Delays on the critical path determine the overall runtime, even if other parts of the program are highly parallel.

6.5.7 Insufficient Parallelism and Serial Bottlenecks

Insufficient parallelism occurs when too much of the code is fundamentally sequential (an Amdahl's-law limitation), making it impossible to scale beyond a certain point. Serial bottlenecks occur when a single process or thread becomes a hub (for example, performing all I/O or all reductions), effectively serializing progress despite the presence of many ranks.

6.6 Serial Performance: How to Find and Fix It

For serial performance problems, the lecture emphasizes the use of profiling and hardware-counter-based tools:

- Traditional profilers (e.g., GNU `gprof`) reveal where time is spent.
- Advanced tools provide cache-miss rates, floating-point unit utilization, branch-misprediction rates, and other microarchitectural signals.

Common optimization strategies include:

Improving data movement and reuse Reorganizing loops and data layouts to increase cache reuse, as in the blocking/tiling techniques used for matrix multiplication.

Arithmetic simplifications Replacing expensive operations with cheaper ones, such as multiplying by a constant instead of dividing, or using precomputed reciprocals.

Approximation Using approximate math functions or lower-precision arithmetic where exact computation is not required, trading a small accuracy loss for speed.

6.7 Communication Performance: Granularity, Overlap, and Synchronization

Communication inefficiency is partly a **granularity** problem. At a high level:

- Coarse-grained decompositions send fewer, larger messages, which can better utilize available bandwidth.
- Fine-grained decompositions may send many small messages, suffering high per-message overhead and limited effective bandwidth.

A key technique for improving communication efficiency is **overlapping communication with computation**. This is typically achieved with non-blocking MPI operations such as `MPI_Isend` and `MPI_Irecv`, followed by completion calls like `MPI_Wait` or `MPI_Test`. Older codes that rely heavily on blocking communication can often benefit by restructuring to hide communication latency behind useful computation.

The lecture also warns that communication costs may grow rapidly with scale if the communication pattern is asymptotically expensive (e.g., all-to-all communication with $O(P^2)$ messages). Remedies may include:

- Redesigning algorithms to communicate within subgroups.
- Using smaller MPI communicators.
- Exploiting topology-aware communication patterns.

Excessive synchronization—such as frequent barriers or heavy global collective operations—can also hurt performance by forcing all ranks to progress at the pace of the slowest. Reducing unnecessary synchronization and increasing asynchrony generally improves scalability.

6.8 Critical Path vs. Serial Bottleneck

The lecture distinguishes **critical paths** from **serial bottlenecks**:

- A *serial bottleneck* is typically a single process or thread that performs work on behalf of many others (for instance, rank 0 performing all file I/O or acting as the root in every reduction). This bottleneck serializes progress for everyone else.
- A *critical path* is a chain of dependent operations across multiple ranks. Delays propagate through this chain as messages and computations, even if no single rank is inherently special.

To analyze critical paths, one can:

- Start with the last process to finish.
- Examine what it was waiting for (e.g., a receive that depended on a send from another rank).
- Recursively trace back through dependencies to see how delays propagated.

Improvements often come from:

- Shortening the critical chain by reducing dependencies.
- Reordering operations to move work off the critical path.
- Introducing overlap so that not all computation lies on the dependency chain.

6.9 Performance Variability and Why It Is Hard

Near the end of the first lecture segment, the instructor introduces **performance variability**: the phenomenon that repeated runs of the same executable, with the same input, can exhibit significantly different runtimes. Variability can be large—sometimes a factor of two or worse.

This variability is problematic because it:

- Slows scientific progress by making some runs unexpectedly slow.
- Reduces overall system throughput.
- Increases energy consumption for the same amount of work.
- Makes performance debugging unreliable: if runtime fluctuates widely, it is hard to tell whether a code change helped or whether a particular run was simply "lucky" or "unlucky."

A particularly frustrating aspect is that variability often arises from factors outside the developer's direct control, such as operating system behavior and shared resource contention.

6.10 OS Noise (Jitter) as a Major Source of Variability

The next lecture segment resumes the discussion on performance variability, focusing first on **operating system (OS) noise** or **jitter**. OS jitter arises because:

- The operating system and various background services (daemons) wake up periodically to perform work.
- When the kernel schedules these OS tasks, the user application is paused.
- After the OS work completes, the application resumes.

Even if the application executes exactly the same instructions each time, these interruptions introduce variability in observed runtime.

6.10.1 Quantifying OS Noise

The lecture describes two general benchmarking strategies for quantifying OS noise:

- Run a **fixed amount of work** repeatedly and measure how much the completion time fluctuates across runs.
- Run for a **fixed duration** and measure how often and how long interruptions occur during that interval.

The slide example discussed used a fixed-work experiment across a large machine (on the order of ~ 8000 cores) to measure how long the same benchmark takes when repeatedly executed on many cores.

6.10.2 Comparison of Systems: Blue Gene/P vs. Cray

A key comparison is made between two systems:

- **Blue Gene/P**: an IBM system with a lightweight kernel and minimal services. Repeated measurements show almost no variability; runtimes are nearly identical from one run to the next.
- **Cray system**: a system with more OS activity. Measurements show visibly distinct "bands" of runtimes.

These bands are interpreted as signatures of different daemon behaviors. Each daemon wakes at some interval and runs for some duration, adding a specific amount of delay. Runtimes cluster around these delay values, producing bands instead of a smooth spread.

6.11 Classic Case Study: "The Case of the Missing Supercomputer Performance"

The lecture references a well-known 2003 paper, *"The Case of the Missing Supercomputer Performance"*, which studied performance variability on the ASCI Q system at Sandia National Laboratories.

The key observation was that nodes/cores exhibited significantly slower performance than expected due to OS noise. The paper identified multiple OS daemons, each with:

- A characteristic period (how often it wakes).
- A characteristic runtime (how long it runs when awakened).

For example:

- One daemon might wake every ~ 30 seconds and run for ~ 10 ms.
- Another might wake every ~ 125 seconds and run for ~ 100 ms.

Each daemon introduces a delay when it runs. The periodicity and duration of these services help explain the banded measurement patterns seen in practice.

6.12 How Compute Variability Cascades into Communication Variability

The lecture emphasizes that variability in computation does not remain isolated to individual processes. In many MPI-style programs with bulk synchronous structure (e.g., stencils, iterative solvers), each iteration has:

- A compute phase.
- A communication phase.

If OS noise lengthens the compute time for one process, that process reaches the communication phase later than others. Its neighbors then:

- Wait longer in receives or synchronization calls.
- Possibly delay their own progress in subsequent phases.

An illustrated example uses processes arranged in a logical ring (0 sends to 1, 1 to 2, . . . , 7 to 0). When one process experiences an unusually long delay due to OS noise:

- Its outgoing messages are sent late.
- The receiver cannot proceed until it receives its data.

- Delays propagate around the ring over multiple iterations.

In subsequent iterations, different processes may be the slow ones, creating a shifting pattern of waiting. Small OS-induced delays (tens or hundreds of milliseconds) can accumulate and amplify through communication and synchronization, producing much larger slowdowns at the application level.

This compounding effect also explains why running on different days or under different load conditions can produce qualitatively different variability patterns. Once delays interact with communication and synchronization, system behavior looks more stochastic from the application's viewpoint.

6.13 Mitigation Strategies for OS Noise

The lecture discusses several strategies for mitigating OS noise, from system-level configurations to user-level practices.

6.13.1 System-Level Approaches

- **Lightweight kernels:** Some supercomputers (e.g., Blue Gene) historically used lightweight OS kernels with fewer daemons, reducing background activity and improving timing stability.
- **Service reduction:** Even on general-purpose Linux, HPC centers often disable nonessential services to minimize interference.
- **Daemon frequency tuning:** For necessary services (such as Lustre or other file-system-related daemons), system administrators may tune their wake-up frequency and behavior to reduce interruption frequency.
- **Dedicated cores for OS work:** Many systems concentrate OS daemons on specific cores (often core 0), so that application cores are less frequently interrupted. This can be combined with CPU affinity to keep user processes away from OS-heavy cores.

6.13.2 User-Level Approaches

From the user side:

- **Thread and process binding** (pinning) prevents threads and processes from migrating across cores, avoiding migration overhead and making behavior more predictable.
- When possible, users can avoid running performance-critical threads on cores that are likely to host OS daemons (e.g., avoiding core 0).

6.14 Preview: Network and File System Contention

The lecture briefly introduces a second major source of variability beyond OS noise: **network** and **file system contention**. Even when a job has exclusive access to its compute nodes:

- Other jobs may share the interconnect fabric.
- Multiple jobs may share parallel file systems.

Because HPC networks often have structured topologies (e.g., fat-tree, dragonfly), traffic patterns can interact in complex ways, creating unpredictable queuing, congestion, and delays. The lecture notes that this topic will be covered in more detail later.

Chapter 7

Load Balancing

7.1 Introducing Load Imbalance: Definition, Symptoms, and Metric

The lecture next transitions to the topic of **load balancing**, which is motivated by performance issues in parallel programs.

7.1.1 Definition of Load Imbalance

Load imbalance occurs when **unequal amounts of work** are assigned to different workers, where "workers" might be:

- MPI processes.
- OpenMP threads.
- Charm++ objects (chares).

Imbalance can arise from differences in:

- Computation: some workers perform more operations.
- Communication: some workers handle more or larger messages.
- Both computation and communication simultaneously.

7.1.2 Symptoms: Slow Collectives

A key symptom of load imbalance is that **collectives appear slow**. In bulk-synchronous programs, the slowest worker constrains progress at synchronization points. When many ranks spend significant time inside `MPI_Barrier` or `MPI_Reduce`, the core issue is often that:

- Faster ranks arrive early to the collective.
- Slower ranks arrive late, forcing others to wait.

Thus, the barrier itself is not necessarily slow; it reflects the fact that some workers have more work or more communication to do than others.

7.1.3 A Simple Load Imbalance Metric

To quantify imbalance, we use the ratio:

$$\text{Load Imbalance} = \frac{\max(\text{load across workers})}{\text{mean(load across workers)}}.$$

If the ratio is close to 1, the program is well balanced. Larger values indicate greater imbalance. The notion of "load" can be adapted:

- Wall-clock time spent per worker.
- Operation counts.
- Any other proxy for work (e.g., number of particles, mesh cells).

7.2 The Load Balancing Workflow

After identifying that load imbalance exists, the lecture frames load balancing as a multi-step process.

7.2.1 Step 1: Determine Whether Load Balancing Is Needed

First, the programmer must decide whether the current imbalance merits intervention. This decision is based on:

- Profiling and timing information.
- Developer knowledge of where work lies.
- Analytical models estimating cost distribution.

7.2.2 Step 2: Decide When and How Often to Balance

Load balancing almost always involves **moving work and data**, which has cost. Thus, the programmer must decide:

- Whether a one-time rebalancing at startup (static) is sufficient.

- Whether the load changes over time and requires periodic rebalancing (dynamic).
- How frequently to rebalance so that the benefit outweighs the cost of migration and planning.

7.2.3 Step 3: Collect Information Required to Balance

At minimum, the algorithm needs:

- Per-worker load estimates.

For more sophisticated balancing, additional information is collected:

- Communication volume (number and size of messages).
- Communication patterns (who talks to whom).
- Topology information (node layout, network distances).

7.2.4 Step 4: Choose or Design a Load Balancing Algorithm

Finally, a load balancing algorithm is selected or designed. Common algorithm styles include:

- Greedy strategies: iteratively move work from overloaded workers to underloaded ones.
- Partitioning-based methods: compute new partitions of a graph or spatial domain.
- Hierarchical methods that operate at multiple scales (see below).

7.3 Static vs. Dynamic Load Balancing

The lecture distinguishes two broad classes of balancing:

7.3.1 Static (Startup) Load Balancing

Static load balancing applies when the workload distribution is effectively fixed. In this case:

- An initial redistribution at startup can be enough.
- Sometimes applications perform a short "learning run" to decide on a good partition, then use that partition in a subsequent production run.

7.3.2 Dynamic Load Balancing

Dynamic load balancing is required when the load changes over time, such as:

- Particle simulations where the number or distribution of particles changes.
- Weather or climate models where regional conditions evolve.
- Adaptive methods where meshes are refined or coarsened dynamically.

In these cases:

- Rebalancing must occur periodically.
- The program must weigh the ongoing cost of rebalance against the benefits of improved load distribution.

7.4 Gathering Load Information: Centralized, Distributed, Hierarchical

The lecture describes three organizational strategies for gathering load information and making balancing decisions.

7.4.1 Centralized Load Balancing

In the centralized approach:

- All workers send their load data to a single process.
- That process computes a global redistribution plan.
- It then communicates the new assignment back to the workers.

Advantages:

- The balancer has a global view of the system.

- Implementation is conceptually simple.

Disadvantages:

- Communication bottleneck from many-to-one data gathering.
- Computational bottleneck from one node doing all planning.
- Potentially poor scalability for very large process counts.

7.4.2 Fully Distributed Load Balancing

In fully distributed balancing:

- Each worker exchanges load information with only a small, fixed number of neighbors.
- Each worker makes local decisions about sending or receiving work.

Advantages:

- No single global decision-maker, so no centralized bottleneck.
- Communication is local, reducing global traffic.

Disadvantages:

- Each worker has only a partial view of the system.
- Local balancing decisions may not produce good global balance.
- Imbalance may persist across neighborhoods even when local neighborhoods are balanced.

7.4.3 Hierarchical (Hybrid) Load Balancing

Hierarchical load balancing is presented as a "sweet spot" between the two extremes. The idea is to:

- Group workers (e.g., in blocks of 1024 processes or by node).
- Within each group, perform a local centralized balance using a group leader.
- Have group leaders report aggregated load information up a hierarchy (e.g., a tree).

At higher levels:

- Leaders can decide to move some load between groups to correct large imbalances.
- Each group is then responsible for selecting specific work units to migrate in or out.

This structure:

- Reduces the cost of gathering global information compared to a single centralized node.
- Provides more global visibility than purely local balancing.

7.5 Information Beyond Compute Load

While compute load is the most common balancing metric, the lecture points out that other signals can significantly improve balancing decisions.

7.5.1 Communication Load

Communication load includes:

- The number of messages sent and received.
- The sizes of those messages.

Workers with high communication load might be more expensive to migrate, or might benefit from being placed closer to their communication partners.

7.5.2 Communication Graphs

Communication graphs encode which workers communicate frequently. Using this information, a load balancer might:

- Co-locate heavily communicating tasks on the same node or nearby nodes.
- Avoid placing heavily communicating tasks on distant parts of the network.

7.5.3 Topology Awareness

Topology-aware balancing uses knowledge of the machine's structure:

- Node-level: which cores share caches or memory channels.
- Network-level: which nodes are close in the interconnect topology.

The goal is to:

- Reduce communication distance.
- Avoid overloading particular links.
- Improve performance for communication-intensive applications.

7.6 Input and Output of a Load Balancing Algorithm

Conceptually, a load balancer takes as input:

- A set of work units (e.g., MPI ranks, tasks, chares).
- The current assignment of these work units to workers.
- Load metrics (compute and possibly communication).
- Optional topology information.

It produces as output:

- A **new assignment** of work units to workers.

In MPI programs, implementing this may be difficult because:

- The natural unit of work is tied to the process.
- Moving work may require moving entire sets of data among processes.

In Charm++, where work units are chares, the runtime:

- Supports migration of individual chares.
- Handles the mapping of chare indices to processing elements.

This makes it easier to move "partial work" without redesigning the entire decomposition.

7.7 Goals and Overheads of Load Balancing

The primary goal of load balancing is to reduce imbalance, ideally bringing the load ratio (max/mean) close to 1. However, several competing objectives and overheads must be considered:

- **Minimize migration volume:** Moving large amounts of data is expensive. Effective algorithms try to move just enough work to fix imbalance.
- **Minimize decision overhead:** The computation of a new assignment should be relatively cheap. Algorithms with very high complexity may not be practical at scale.
- **Account for communication costs:** If the application is communication-heavy, a good assignment should not just balance compute load but also reduce communication cost and avoid creating new hotspots.

Overall, the load balancer must improve performance more than it costs.

7.8 Static Load Balancing Strategies

The lecture concludes by describing two static (startup-time) load balancing strategies commonly used in practice.

7.8.1 Space-Filling Curves

Space-filling curves (e.g., Hilbert or Z-order curves):

- Map multi-dimensional coordinates (2D, 3D, etc.) into a one-dimensional ordering that preserves locality as much as possible.
- Are particularly useful for spatial decompositions.

A typical process:

- Assign a 1D index to each work unit (e.g., spatial cell) based on its position along the space-filling curve.
- Sort work units by this index.

- Partition the sorted list into contiguous segments, each with roughly equal load.

Because nearby points in the original space are often near in the 1D order, this approach tends to keep spatially adjacent work together while balancing load.

7.8.2 Orthogonal Recursive Bisection (ORB)

Orthogonal Recursive Bisection is commonly used in particle and N-body simulations. The idea is:

- Consider the computational domain in a multi-dimensional space.
- Recursively split the domain along coordinate axes (e.g., first along x, then along y, then along z).
- At each split, choose the partition location so that the two subdomains contain roughly equal load (often measured by particle counts or estimated cost).

This yields a tree of subdomains, each associated with a roughly equal amount of work. ORB produces partitions that:

- Have balanced load.
- Preserve spatial structure, which is beneficial for locality and communication.

Chapter 8

HPC Interconnects, Congestion, and Parallel File Systems

8.1 Overview and Motivation

The lecture begins by emphasizing that, even though HPC *compute nodes* are typically dedicated to a single job and user, two major shared system components heavily influence performance once you scale beyond a single node:

- The **interconnection network**, which carries all message traffic between nodes.
- The **parallel file system**, which stores user data and executables.

The interconnect becomes relevant as soon as an application uses multiple nodes because messages must traverse links between nodes via switches and routers. The parallel file system is always involved because the files and directories users interact with on login and compute nodes are usually *mounted views* of storage hosted on separate I/O servers. In this set of lectures, the primary focus is first on high-end HPC interconnection networks and then on parallel file systems; both subsystems play central roles in performance and variability.

8.2 High-End HPC Networks and Why They Matter

We contrast HPC interconnects with geographically distributed networks such as the public internet and cloud-wide networks. HPC systems are typically:

- **On-premise**, in a single machine room or data center.
- Engineered specifically for **low latency** and **high bandwidth**.

Because the network only spans a single facility, system designers can:

- Use specialized hardware and topologies tuned for latency and throughput.
- Avoid the high latencies and variability inherent in wide-area networks.

Low latency means that messages start arriving quickly after being sent. High bandwidth means the network can sustain large data transfer rates once transmission begins. These properties are essential to make communication efficient at scale in tightly coupled parallel applications.

8.3 Hardware Building Blocks of an Interconnect

The lecture introduces the core physical components that make up an HPC network.

8.3.1 Network Interface Cards (NICs)

A **network interface card (NIC)** is the device connecting a compute node's CPU and memory subsystem to the network. Conceptually, the NIC:

- Moves data between host memory and the network link.
- Packetizes and injects messages onto the wire.
- Receives packets from the network and delivers data to the host.

8.3.2 Switches and Routers

A **router** or **switch** is the infrastructure component that interconnects many endpoints:

- Many compute nodes, each via its NIC, plug into switches.
- Switches also connect to other switches, forming a multi-stage network.

The terms "switch" and "router" are often used interchangeably in this context, though router emphasizes the path-selection role.

8.3.3 Cabling: Copper and Optical

Physical connectivity is provided by cables. Two main categories are:

- **Copper (electrical)** cabling, which is cheaper but usually limited to shorter distances due to signal degradation.
- **Optical (fiber)** cabling, which is more expensive but supports much longer distances and is often used for links spanning racks or cabinets.

In HPC machine rooms, there is no Wi-Fi in the interconnect path. All high-performance connections between nodes and switches rely on these physical cables.

8.4 Logical Topology vs. Physical Cabling

A recurring theme is the difference between:

- The **physical layout**: the actual arrangement of racks and cables in the machine room.
- The **logical (virtual) topology**: the abstract graph that describes how switches and nodes are connected.

Although you may see huge bundles of cables, you cannot deduce the logical topology directly by looking at them. Network architects first choose a virtual topology (mesh, torus, fat-tree, Dragonfly, etc.), then cable the system to implement that connectivity. When we analyze performance, we reason about the logical topology and how messages traverse it, not the visual tangle of cables.

8.5 Core Network Concepts: Hops, Diameter, and Radix

To reason precisely about network structure, the lecture defines several key terms.

8.5.1 Hop

A **hop** is a single step across a link from one router (switch) to another. Depending on convention, one may also count the hop from the NIC to its attached switch. In performance models, we often focus on:

- How many switching stages a message traverses.
- How hop count influences latency and contention.

8.5.2 Network Diameter

The **network diameter** is the longest shortest-path distance between any pair of nodes (or switches) in the network graph. Formally:

- For each pair of endpoints, we consider the shortest path between them.
- The diameter is the maximum of these shortest-path lengths.

Because most networks have multiple routes between endpoints, the "shortest path" qualifier is important: we use the minimal hop count for each pair.

8.5.3 Router Radix

The **radix** of a router or switch is its number of ports. Radix is central to network design:

- It determines how many compute nodes can attach to a switch.
- It determines how many inter-switch connections can be supported.

Later, we will see that radix essentially governs how many levels (and how many endpoints) a given topology can scale to.

8.6 Mesh Networks

The lecture uses the **mesh** as the first detailed example of a topology. In a mesh:

- Switches are arranged in a Cartesian grid.
- Each switch is connected to its nearest neighbors in each dimension.

8.6.1 N-Dimensional Mesh Structure

In a 2D mesh:

- Interior switches connect to four neighbors: north, south, east, and west.

In a 3D mesh:

- Interior switches connect to six neighbors: $\pm x, \pm y, \pm z$.

In general, an N-dimensional mesh has switches with links to $2N$ neighbors (along each positive and negative direction of each dimension).

A modeling note is that diagrams often label circles as "nodes," but topology discussions treat those circles as **switches**. Compute nodes attach to these switches, sometimes one per switch in older designs and multiple per switch in newer systems.

8.6.2 Bidirectional Links

In drawings, a single line between two switches is understood to be a **bidirectional** link: traffic can flow in both directions. Physically, this might be implemented as two unidirectional pairs, but at our level we treat the link as capable of sending and receiving concurrently.

8.6.3 Router Port Budgeting Example

A concrete example illustrates how topology drives hardware requirements. In a 3D mesh:

- Each switch needs six ports to connect to its neighboring switches ($\pm x, \pm y, \pm z$).
- If we attach four compute nodes per switch, we need four additional ports.

This yields a total of ten ports per switch. In real designs, architects often start from available router radix and choose topologies and node-per-switch ratios accordingly.

8.7 Torus Networks: Meshes with Wraparound

A **torus** network is a modified mesh where wraparound links connect opposite edges of the mesh in each dimension. For example:

- In a 2D torus, you connect both horizontal edges and both vertical edges, so the grid "wraps around" in both directions.
- In a 3D torus, wraparound links are added in each of the three dimensions.

The name comes from the geometric shape: if you embed the wraparound grid on a surface, it resembles a donut (torus). The term "wraparound" connects to the wraparound boundaries that students often implement in Game of Life or stencil examples.

Historically, IBM's **Blue Gene** systems and several generations of Cray systems used mesh or torus interconnects.

8.7.1 Physical Realization of Wraparound Links

While logical diagrams depict a single long wraparound cable, real installations avoid extreme cable-length outliers. Instead:

- Switches and racks are physically arranged so wraparound connections correspond to moderate-length cables.
- Cabling patterns create the logical wraparound while maintaining manageable physical distances.

8.8 Network Diameter in Meshes and Tori

8.8.1 Mesh Diameter in 2D

For an $X \times Y$ 2D mesh, the farthest switches are typically opposite corners. Using Manhattan routing (moving along x and y dimensions), the shortest distance between opposite corners is:

- $(X - 1) + (Y - 1)$ hops if we count only inter-switch hops.
- $X + Y$ hops if we also count the endpoint link from NIC to switch (depending on convention).

This is essentially the **Manhattan distance** on a grid; multiple shortest paths exist, but they all have the same total hop count.

8.8.2 Torus Diameter: Intuition

For a torus, wraparound links reduce the maximum shortest-path distance. In a 1D example:

- On an 8-node line, node 0 to node 7 is 7 hops away.

- On an 8-node ring (1D torus), node 7 is just 1 hop away from node 0, and the farthest node becomes node 4 at 4 hops.

This suggests that adding wraparound links roughly cuts the diameter in half. Extending to 2D, the lecture notes that the torus diameter can be thought of as roughly $\frac{X+Y}{2}$, and similarly in higher dimensions the sum of dimension lengths is effectively halved. The important conceptual takeaway is that wraparound connectivity *short-circuits* long mesh paths and reduces worst-case distances.

8.9 Cable Length, Latency, and Routing Assumptions

A student asks whether cable length affects message time. We distinguish copper and optical cables by distance and cost, but then notes a common modeling assumption in HPC:

- Within a single machine room, systems and routing techniques (e.g., wormhole routing) are designed so that moderate cable-length differences are relatively small compared to queuing and hop-based delays.

For course-level performance reasoning:

- Hop count and **contention** are usually more important than small differences in propagation time.
- We often treat each link as having a similar base latency, and focus on how many links and how congested they are.

8.10 Fat-Tree Networks: Construction and Intuition

The lecture then introduces **fat tree** networks, better known as fat-tree or Clos-like networks. These are hierarchical multi-level switch networks widely used in modern interconnects.

8.10.1 Basic Idea and Terminology

The core idea is to:

- Attach compute nodes to **leaf switches**.
- Connect groups of leaf switches to higher-level **aggregation switches**.
- Add additional levels as needed to scale to more nodes.

A central design parameter is the **switch radix** K, the number of ports per switch. In the lecture's drawings, $K = 4$ is used for pedagogy, but real systems use radix values like 36, 40, or 64.

8.10.2 Port Splitting: Down vs. Up

At intermediate levels, a common simplifying assumption is:

- $K/2$ ports face **down** (to lower-level switches or nodes).
- $K/2$ ports face **up** (to higher-level switches).

At the top level, all ports can face downward because there is no higher level.
High radix is attractive because:

- One switch can directly attach many compute nodes (e.g., with $K = 40$, roughly 20 nodes per leaf switch under the $K/2$ rule).
- The total number of switches required is greatly reduced compared to low-radix designs.

8.10.3 Pods and Scaling

A **pod** is a repeating building block, typically containing:

- A set of leaf (bottom-level) switches.
- A set of aggregation (middle-level) switches.
- Rich connectivity between leaves and aggregation switches.

Under the simplified $K/2$ up / $K/2$ down assumption:

- A pod can contain up to $K/2$ leaf switches and $K/2$ aggregation switches.
- Each leaf switch can connect to $K/2$ nodes.

Hence, the number of compute nodes per pod is approximately:

$$\left(\frac{K}{2}\right) \cdot \left(\frac{K}{2}\right) = \frac{K^2}{4}.$$

When you exceed the capacity of a single pod, you add more pods and add a top-level of switches that connect pods to each other.

The lecture mentions some confusion in small-number examples about the exact count of top-level switches (e.g., "$K/4$" appears at one point). The key message is that the *principles* of scaling are determined by radix and how ports are allocated to up/down connections at each level.

8.10.4 Fat-tree Network Diameter

A major advantage of fat-tree networks is that the diameter is nearly constant once the number of levels is fixed. In a three-level arrangement (leaf $\rightarrow$ middle $\rightarrow$ top):

- The farthest endpoints are usually in different pods.
- Traffic goes up from the source leaf to the top, then down to the destination leaf.

The worst-case switch-to-switch hop count is on the order of twice the number of levels traversed (up and down). For a 3-level fat-tree:

- Endpoints on the *same* leaf switch may be only 2 hops apart (counting NIC-to-switch links).
- Endpoints in the *same pod* but different leaves may be 4 hops apart.
- Endpoints in *different pods* may be around 6 hops apart.

Notably, K does not appear directly in these hop counts; the diameter depends primarily on the *height* of the tree, which grows slowly because high-radix switches let you scale to many nodes with only 2–3 levels.

8.10.5 Capacity Example

Using a simplified scaling approximation, the maximum number of nodes in a 3-level fat-tree is roughly:

$$\frac{K^3}{4}.$$

For $K = 40$, this yields on the order of 16,000 compute nodes. If each node contains multiple GPUs (e.g., 4 GPUs per node), this corresponds to very large accelerator counts without needing an additional network level.

8.11 Dragonfly Networks

8.11.1 High-Level Structure

Dragonfly is another modern topology that:

- Uses high-radix routers (e.g., 48 ports).
- Provides very low diameter (often even smaller than fat-tree).
- Organizes routers into two levels of hierarchy:
  - **Intra-group** connectivity: routers in a group are densely interconnected.
  - **Inter-group** connectivity: groups are connected by global links.

Each **group** (also called a clique or supernode) contains multiple switches and compute nodes. Within a group, the design aims for rich connectivity to keep local communication cheap.

8.11.2 Intra-Group Variations

Different vendors implement intra-group connectivity differently:

- One style (IBM-like) uses nearly *all-to-all* links, where each switch in the group has a direct connection to every other switch. Diagrams often omit some lines because drawing every cable would be visually overwhelming.

- Another style (Cray-like) uses row/column structures, where switches are strongly connected along rows and columns, but not fully all-to-all. This reduces the number of cables while retaining strong local connectivity.

8.11.3 Global Links and Redundancy

At the global level, groups must be connected so that the overall network is connected. Having only a single link between groups is undesirable because:

- It creates a serious bottleneck for traffic between those groups.
- It reduces fault tolerance.

Therefore, Dragonfly designs use multiple global links distributed across switches so that:

- Traffic has multiple paths between groups.
- Load can be spread across global links.

Routers with radix K (e.g., 48 ports) must allocate:

- Some ports to compute nodes.
- Many ports to intra-group links.
- The remainder to global links across groups.

8.11.4 Dragonfly Diameter

The lecture describes both best-case and worst-case hop counts:

- **Best case:** endpoints within the same group or on switches connected by a direct global link may have very short paths.
- **Worst case:** at the switch level, a packet may need:
  1. One hop within the source group to reach a switch with an appropriate global link.
  2. One global hop between groups.
  3. One hop within the destination group to reach the target switch.

This yields about 3 switch hops in the worst case. Including NIC-to-switch links at the endpoints adds roughly two more hops, yielding about 5 hops in total for endpoint-to-endpoint paths.

8.12 Message Lifecycle and Latency Components

The lecture next describes how a message travels through the network:

1. The source process passes data to its NIC.
2. The NIC breaks the message into packets suitable for the link.
3. Packets are injected from the NIC onto the first switch.
4. Packets traverse intermediate routers/switches, where each router decides the next hop based on routing tables or algorithms.
5. The destination NIC receives packets and reassembles the message.
6. The data is delivered to the destination process.

Various factors contribute to total message latency:

- Memory-to-NIC transfer time at the source.
- Injection rate limits at the NIC.
- Per-router processing or forwarding delays.
- Queuing delays due to congestion.
- NIC-to-memory transfer time at the destination.

Total communication time integrates all these components along the path.

8.13 Routing: Static, Dynamic, and Adaptive

Routing algorithms determine which sequence of switches a message follows between a given source and destination.

8.13.1 Static Routing

Static routing precomputes routes at boot time or configuration time. Each router:

- Receives a routing table mapping destinations to next hops.
- Uses these rules for the lifetime of the job or system boot.

Static routing is simple but cannot respond to changing traffic patterns or hotspots.

8.13.2 Dynamic and Adaptive Routing

Dynamic routing allows routes to change at runtime. However, dynamic does not necessarily mean congestion-aware; routes could be updated for other reasons (e.g., failures, reconfiguration).

Adaptive routing is dynamic routing that reacts to congestion:

- It monitors link loads or error conditions.
- It chooses alternatives when some paths are heavily loaded.

8.13.3 Routing in Specific Topologies

Torus Routing. Torus networks often use shortest-path routing based on Manhattan distance. The routing algorithm specifies an order of dimension traversal (e.g., first X, then Y, then Z) to reach the destination.

Fat-tree Routing. Fat-tree routing has historically been static, with equal-cost paths encoded in switch tables. More recently, adaptive routing is used to route around hotspots and distribute traffic.

Dragonfly and UGAL. Dragonfly networks often use UGAL-style adaptive routing. The key idea:

- Instead of always taking the direct global path between source and destination groups, the router can send traffic through an intermediate group.
- For example, *source group* $\rightarrow$ *random/intermediate group* $\rightarrow$ *destination group*.

This randomization can:

- Add one extra hop.
- Significantly reduce contention by spreading traffic across global links rather than overusing a single direct shortcut.

8.14 Network Congestion and Why It Matters

8.14.1 Defining Network Congestion

Network congestion occurs when multiple message streams attempt to use the same physical link simultaneously. For example:

- If three packets need the same link at once, only one (or some subset) can be forwarded immediately.
- The others must wait in buffers, which introduces stalling and increases message latency.

This is a major source of performance variability:

- Running the same executable on the same input can yield different runtimes depending on how congested the network is during each run.
- Variability can arise from both compute-side causes (e.g., OS jitter) and communication-side causes (congestion and queuing delays).

8.14.2 Identifying Whether Variability is Compute or Communication

To determine whether variability is dominated by compute or communication, we can:

- Break total runtime into compute time and communication time using performance tools.
- Compare variability across repeated runs for each component.

In one example described in the lecture:

- Compute time shows little variability, even across systems with different OS configurations.
- Communication time varies significantly.

This indicates that, for that case, variability is largely driven by network effects, not by compute jitter.

8.15 Placement-Induced Congestion on Torus Networks

The lecture presents a case study from an older torus system (e.g., Cray Hopper era), comparing:

- A **fast run**.
- A **slow run**.

By mapping job placement on the torus, we observe that:

- In the *slow run*, the job under study (drawn as "blue") is surrounded by many other jobs that also communicate heavily.
- Torus shortest-path routing tends to route traffic through the same limited set of links, so the blue job shares critical links with neighbors' traffic, increasing contention.
- In the *fast run*, fewer nearby jobs share those links, so the blue job experiences less interference and better performance.

Historically:

- IBM systems often allocated jobs as **convex contiguous partitions** (rectangular prisms in the torus) to minimize cross-job interference. This reduces congestion but can cause fragmentation and longer queue times.
- Cray systems explored more flexible placement policies, which can increase variability by mixing jobs more closely.

8.16 Mitigating Congestion: Three Approaches

There are three broad strategies for mitigating congestion.

8.16.1 Topology-Aware Node Allocation (System-Level)

Schedulers can allocate nodes in ways that reduce interference between jobs.

For torus networks:

- Jobs are assigned convex shapes in the coordinate space to keep their traffic within local regions.

For fat-tree networks, simple heuristics include:

Heuristic 1 Place small jobs within a single leaf switch whenever possible. If a leaf has, say, 20 down-facing ports for nodes, a job that requests 16 nodes can be entirely hosted on that switch, keeping communication local.

Heuristic 2 If a job must span multiple switches, keep it within the same pod as much as possible. This avoids sending traffic up into higher levels, which are more heavily shared.

For larger jobs:

- Allocate whole pods when possible.
- Minimize the fraction of a pod that is shared between unrelated jobs.

The guiding principle is to make allocations as *compact* as possible in the topology to lessen interference.

8.16.2 Congestion-Aware Adaptive Routing (System-Level)

Adaptive routing can respond dynamically to congestion. Conceptually, the routing system:

- Maintains estimates of current link load (e.g., packet counts).
- Identifies the most heavily loaded links.
- Selects some flows crossing those links and reroutes them to underutilized links, where alternative paths exist.

In fat-tree networks with many equal-cost paths, this process can reduce the peak link load by spreading traffic. The adaptation may be rerun:

- When new jobs are scheduled.
- Periodically for long-running jobs with changing communication patterns.

This logic typically runs on management or service nodes and requires system administrator-level control.

8.16.3 Topology-Aware MPI Process Mapping (User-Level)

Even when the system's node allocation is not topology-optimal, users can help by mapping MPI ranks so that:

- Frequently communicating ranks are placed close together in the network.

This is graph embedding: mapping the application's communication graph onto the machine's topology graph. Since optimal embedding is NP-hard, practical solutions rely on heuristics and approximations. Good process mapping can:

- Reduce hop counts.
- Reduce contention by avoiding placing communicating ranks on opposite sides of the network.

8.17 Bidirectional Links

The lecture explicitly notes that interconnect links are **bidirectional**:

- Each direction typically has its own physical lanes or wires.
- Messages are directional; for each link, there can be traffic in both directions if both endpoints send data.

Switching logic must arbitrate traffic in each direction, manage buffers, and avoid deadlock while sustaining high throughput.

8.18 Transition to Parallel I/O and the File System

After discussing networks, the lecture shifts focus to **parallel I/O**. Parallel programs perform I/O when:

- Reading input datasets.
- Writing results or logs.
- Producing checkpoints for restart/recovery.

A trivial but non-scalable pattern is to have a single process (often rank 0) handle all I/O, redistributing data to other ranks through MPI. This approach quickly becomes a bottleneck.

To understand scalable parallel I/O, the lecture first describes the hardware-software architecture of the I/O subsystem and parallel file systems.

8.19 Parallel File System Architecture: Metadata vs. Data

Parallel file systems separate:

- **Metadata management**: file names, directory structure, permissions, ownership, timestamps, sizes, and layout.
- **Data storage**: the actual file contents on disks.

8.19.1 Metadata Server (MDS)

Metadata is managed by dedicated servers:

- A **metadata server (MDS)** holds information about files and directories.
- In some systems, a separate management server coordinates other operations, but conceptually the MDS is the primary metadata component.

8.19.2 Object Storage Servers (OSS) and Targets (OST)

Data is stored on many disks known as:

- **Object storage targets (OSTs)**, which are the disk arrays.

These OSTs reside behind:

- **Object storage servers (OSSs)**, which are the nodes that physically host the disks and serve read-/write requests.

Compute and login nodes generally do not store user files locally. Instead they:

- Mount the parallel file system so that the same directory tree is visible from all nodes.

This allows workflows such as compiling on a login node and running the same binary on compute nodes.

8.19.3 File System Clients on Compute Nodes

Compute nodes run file system client components (daemons). For Lustre, this is the Lustre client. When an application performs I/O:

- The client contacts the MDS to resolve which OSTs hold the file's data.
- The client then issues read/write requests to the OSSs that host the relevant OSTs.

8.20 Connecting Compute and I/O Networks

Many systems effectively have two network "zones":

- One for the compute cluster interconnect.
- One for the storage cluster (I/O network).

To move data between them, special **bridging nodes** are used. In the Lustre environment:

- These nodes are called **LNet routers**.
- They have interfaces both on the compute interconnect and the storage network.

When a compute node issues I/O:

- Traffic traverses the compute interconnect to an LNet router.
- Then it crosses into the storage network.
- Finally it reaches the OSS node holding the relevant OST.

8.21 Storage Tiers: Scratch, Burst Buffers, and Tape

The lecture also covers multiple tiers of storage in HPC environments.

8.21.1 Scratch and Burst Buffers

The term **scratch** can refer to different things:

- A scratch directory on the parallel file system, used for temporary high-volume data.
- A separate storage tier called a **burst buffer**, typically comprising fast NVMe/flash devices.

Burst buffers:

- Provide high performance for temporary data during a job.
- Are typically not persistent beyond job lifetime; users are expected to move important results to the parallel file system before the job ends.

8.21.2 Tape Storage

Tape storage provides:

- Cheaper, higher-capacity storage for long-term archiving.
- Much slower access compared to disk-based parallel file systems.

Users often:

- Keep active working data on the parallel file system.
- Move older or less frequently used data to tape to free expensive disk space.

8.22 Parallel File Systems and Striping

The lecture mentions several common parallel file systems:

- **Lustre**.
- **BeeGFS**.
- **GPFS/Spectrum Scale**.
- **PVFS** and others.

The main performance idea they share is **striping**. Striping means:

- A single large file can be split and distributed across multiple OSTs and OSS nodes.
- When reading or writing such a file, multiple disks and servers can participate in the operation.

This increases achievable:

- Aggregate I/O bandwidth.
- Parallelism in serving requests.

This is why parallel I/O can scale on large systems: it leverages many disks in parallel rather than relying on a single disk or server.

8.23 Compute, Network, and I/O as Cost Drivers

The lecture closes by emphasizing that when designing or purchasing an HPC cluster, three major subsystems dominate cost:

- **Compute** (CPUs, GPUs, memory).
- **Network** (interconnect switches, cables, NICs).
- **I/O subsystem** (parallel file systems, OSSs, disks, tape, and any burst buffers).

Centers must choose how to allocate budget among them, based on whether they prioritize:

- Peak performance vs. predictability.
- Capacity vs. bandwidth.
- Utilization and queue times vs. isolation and reduced variability.

Understanding interconnect topologies, congestion behavior, and parallel file system architecture is thus essential to both system designers and application developers who want to achieve good performance and scalability.

Chapter 9

CSE Applications

9.1 Overview and Motivation

This lecture segment introduces **molecular dynamics (MD)** as the first major computational domain used to motivate why large-scale parallel computing matters. We can frame parallel computing as a cross-cutting tool used across many domains:

- Nanoscale science (atoms and molecules).
- Cosmology and astrophysics.
- Industrial and public-impact domains such as finance, epidemiology, weather and climate modeling, and weapons design.

The goal is not to survey every possible application but to explore a few representative scientific workloads that push computational and parallel capabilities. These case studies illustrate what scientists and engineers can do when they have access to powerful supercomputers.

9.2 Molecular Dynamics: What It Is and Why People Use It

9.2.1 Basic Definition

Molecular dynamics (MD) is the computational study of **how atoms and molecules move over time** in a simulated physical region, often called a *simulation box*. The simulation tracks positions and velocities of atoms and uses physical models to evolve them forward in time.

9.2.2 Application Areas

There are two major application areas:

Drug Design. Modern pharmaceutical workflows increasingly use MD to:

- Evaluate whether a candidate drug molecule plausibly binds to a target protein or biological structure.
- Explore whether such binding could inhibit or alter a biological function (e.g., inhibiting an HIV enzyme).

This computational stage does *not* replace wet-lab experiments, but it:

- Narrows down candidate molecules.
- Helps prioritize which compounds to test experimentally.

Materials Design. MD is also used to study how atoms and molecules:

- Bind to form materials with specific mechanical or electronic properties.
- Behave in contexts such as chip fabrication, construction materials, and engineered compounds where strength, flexibility, and bonding behavior matter.

While some companies own large computing resources, many also use outsourced infrastructure (including cloud services) because operating supercomputers carries substantial overhead.

9.3 Physical Model and Forces in MD

MD simulations evolve a molecular system according to **Newton's equations of motion**, conceptually summarized as "force determines acceleration," and acceleration determines changes in velocity and position.

At each time step, the simulation computes the **net force** on each atom. The net force is composed of:

9.3.1 Bonded Interactions

Bonded interactions arise from explicit chemical structure. Common bonded terms include:

- A **bond** between two atoms.
- A **bond angle** involving three atoms.
- A **dihedral angle** involving four atoms.

These terms act to maintain approximate bond lengths and angles and generate forces that pull atoms toward these preferred structural configurations.

9.3.2 Non-Bonded Interactions

Non-bonded interactions act between atoms that are not explicitly bonded. They include:

- **Electrostatic** forces due to partial or full charges on atoms.
- **Van der Waals** forces capturing short-range repulsion and attraction not explained by bonds or charges.

The core computational task at each time step is thus:

1. Compute all relevant bonded forces.
2. Compute non-bonded forces for interacting atom pairs.
3. Sum contributions to obtain the net force on each atom.
4. Use these forces to update accelerations, velocities, and positions.

9.4 Simulation Sizes and Time Scales

The lecture provides rough orders of magnitude for modern MD:

- Simulations commonly involve **thousands to millions of atoms**.
- Large research simulations can reach **tens to hundreds of millions of atoms** on top-end machines.

Time integration uses extremely small steps:

- Typical step size is on the order of ~ 1 femtosecond (10^{-15} seconds).

Even if the desired real-world simulation time is small (e.g., microseconds to milliseconds), the required number of steps is enormous:

- Number of steps = (real time) / (time step).
- For microseconds at 1 fs per step, this means billions of time steps.

Since each step involves many force calculations, the combination of:

- A large number of atoms.
- A large number of time steps.

makes MD an ideal driver for large parallel computing resources. It also explains why we still cannot simulate extremely large biological systems with full detail over long real-time durations, despite substantial computing power.

9.5 Sequential MD Algorithm

The lecture outlines the high-level structure of a sequential MD code as a loop over time steps:

1. **Compute short-range bonded and non-bonded forces.**
   At every time step, the code computes:
   - Bonded forces (bonds, angles, dihedrals).
   - Short-range non-bonded forces (within a cutoff region).
2. **Compute long-range forces less frequently.**
   Long-range forces (especially electrostatics) are:
   - Computed using approximate methods like Particle Mesh Ewald (PME).
   - Re-evaluated every few steps rather than every step to save time.
3. **Integrate equations of motion.**
   Using the net force on each atom, the code:
   - Updates accelerations and velocities.
   - Updates positions to move atoms.
4. **Repeat for many time steps.**
   This loop runs for a very large number of iterations to simulate the desired real time.

The outer time-step loop is the dominant repeated structure and is the focus of both algorithmic and parallelization efforts.

9.6 From O(n^2) to O($n \log n$): Short-Range vs. Long-Range

A central performance challenge is that if every atom interacted explicitly with every other atom, force computation would be:

- $O(n^2)$ in the number of atoms.

This is too expensive for realistic system sizes.

9.6.1 Short-Range Forces with Cutoffs

To reduce cost, MD codes:

- Compute only interactions within a cutoff radius for each atom.
- Ignore or approximate interactions beyond that cutoff at this stage.

This reduces the number of interactions per atom and hence the total work.

9.6.2 Long-Range Forces with PME

Long-range electrostatic interactions beyond the cutoff are computed using approximate methods. One widely used algorithm is **Particle Mesh Ewald (PME)**:

- It decomposes the potential into short-range and long-range parts.
- The long-range part is computed in Fourier space using FFT-based techniques.
- This reduces the complexity from $O(n^2)$ to approximately $O(n \log n)$.

Long-range calculations are often:

- Performed less frequently than short-range evaluations.
- More expensive per evaluation but amortized over several time steps.

9.7 Parallelizing Short-Range MD: Four Approaches

The lecture discusses four main strategies for parallelizing short-range forces. Each strategy reflects a different choice of:

- **Data distribution**: which atoms live on which processes.
- **Work distribution**: which process computes which forces.

9.7.1 Atom Decomposition

In **atom decomposition**:

- Each process is assigned roughly N/P atoms.
- Each process computes forces for the atoms it owns.

Drawbacks:

- **Load imbalance**: different atoms may have different numbers of neighbors or bonded configurations, leading to unequal work.
- **High communication**: atoms that are spatially close may be owned by different processes, so force computation requires exchanging position/state information extensively.

This approach is conceptually simple but tends to scale poorly when spatial locality is not respected.

9.7.2 Force Decomposition via a Force Matrix

In **force decomposition**, we conceptually form an interaction matrix:

- Rows and columns correspond to atoms.
- Non-zero entries indicate pairs that interact (within cutoff).

Processes are assigned subsets of this matrix, i.e., subsets of pairwise interactions, rather than subsets of atoms. This:

- Helps balance computation by assigning an equal number of interactions per process.
- Does not automatically solve communication problems, because the process computing a given interaction may not own the corresponding atoms and must fetch their data.

9.7.3 Spatial (Domain) Decomposition

Spatial decomposition uses the geometry of the simulation region:

- The simulation box is divided into sub-boxes.
- Box size is chosen relative to the cutoff radius (e.g., 16 Å in the lecture example) so that atoms in one box only need data from that box and neighboring boxes.

Mechanics:

- Each box is assigned to a process, which stores the atoms in that region and computes forces for its atoms.
- Processes exchange atom information with neighboring boxes to obtain local neighbors for force computations.
- Atoms can **migrate** across box boundaries over time; processes transfer atoms to neighbors when they move into new boxes.

Neighbor counts:

- In 2D, a box's cutoff region overlaps a 3×3 neighborhood, i.e., 8 neighboring boxes (minus itself).
- In 3D, the analogous region is $3 \times 3 \times 3$ minus itself, i.e., 26 neighbors.

Scalability limitation:

- The number of boxes is limited by the global simulation size and box size.
- For example, if the simulation is 64 Å per side and boxes are 16 Å, only 4 boxes fit per dimension; in 2D that is 16 boxes total, limiting the number of processes if each process owns one box.

Smaller Boxes and Communication Trade-Offs

A tempting approach is to split boxes further to use more processes. However:

- Reducing box size increases the number of boxes that cover a given cutoff region.
- A 3×3 neighbor region can become a 5×5 region (in 2D) for very small boxes, increasing the number of neighbors and thus the communication volume.

This shows a fundamental trade-off:

- Finer decomposition allows more parallelism.
- But it also increases communication overhead and complexity.

9.7.4 Hybrid Parallelization: Decoupling Data and Work

To address limitations of pure spatial decomposition, the lecture presents hybrid methods that **decouple** data ownership from force computation.

The idea:

- One set of objects is responsible for **holding atoms** in spatial boxes (data objects).
- Another set of objects is responsible for **computing forces** between pairs of boxes (compute objects). Each compute object handles interactions for one pair of boxes (or a subset of atom pairs).

Because there can be many more box-pairs than boxes, this approach:

- Provides many more units of parallel work.
- Increases scalability beyond the number of boxes.

The cost is:

- Increased communication, since compute objects may need to fetch atoms from one or both boxes they refer to.

The lecture notes that this style is associated with **NAMD**, and that NAMD is built on **Charm++**. In Charm++, one can represent:

- Spatial boxes as one chare array.
- Force-compute objects (box pairs) as another chare array.

The runtime then maps and load-balances these arrays across processes.

9.7.5 Neutral Territory (Midpoint) Method

Another hybrid approach is the **neutral territory method**, also called a midpoint method, associated with the MD package **Desmond** (from D. E. Shaw Research) and the specialized Anton hardware.

Rule:

- For each interacting atom pair in different boxes, find the midpoint of their positions.
- The process (or box) containing that midpoint is responsible for computing the force between them, even if it does not own either atom.

This "neutral" assignment:

- Spreads force-computation work more evenly across processes.
- Improves load balance at very large process counts.

As with other hybrid methods, this comes with extra communication cost because the midpoint owner must fetch state information for both atoms. These methods are most beneficial at large scale (thousands of processes), where better scalability can outweigh extra communication.

9.8 Computational Epidemiology: Motivation and Challenges

9.8.1 Why Computational Epidemiology Matters

The lecture resumes with **computational epidemiology** as another motivating domain. Epidemiological simulations help society:

- Understand and control the spread of infectious diseases.
- Make informed public policy decisions during outbreaks.

Simulations can inform:

- When to close schools or workplaces.
- When to start vaccination campaigns.
- How other mitigation measures might affect disease spread.

Modern conditions complicate disease control:

- Dense urbanization increases contact rates.
- Local and global travel rapidly move infections across regions.
- Population aging changes vulnerability and healthcare demands.

These factors make large-scale, data-driven modeling both more important and more challenging.

9.8.2 How Epidemiology Differs from Traditional HPC Workloads

Traditional computational science simulations (e.g., MD, computational astronomy) often:

- Spend most time in floating-point-heavy kernels.
- Rely on regular grid-based computations or structured neighbor patterns.

By contrast, epidemiology simulations are dominated by:

- Integer state updates.
- Branching and control flow (e.g., deciding whether a person visits a location, whether two people meet, whether infection occurs).
- Irregular memory access patterns.

While some floating-point arithmetic may appear, the dominant cost is often control flow and irregular communication, making optimization and scaling more difficult.

9.8.3 Why Parallel Epidemiology Is Hard

Challenges include:

- **Massive scale**: social contact networks may represent city-, country-, or world-scale populations (hundreds of millions to billions of individuals), exceeding single-node memory.
- **Irregular graphs**: degree distributions vary widely across people and locations. Some individuals or locations are hubs; others are sparsely connected.

- **High communication**: interactions are message-heavy. People send "visit" messages to locations; locations send exposure or infection results back. This message volume can dominate runtime and stress the interconnect.

9.9 Time-Stepped Simulation Loop and Synchronization

Epidemiology simulations often use a **time-stepped** approach:

- The main loop iterates over discrete time steps (e.g., days or smaller intervals).
- At each step, people visit locations, contacts occur, and infection state updates may happen.

Parallel detail:

- A synchronization point at the end of each time step is usually required.
- All infection-state updates must be complete before the next step so that visits reference consistent states.

This sequential dependency across time steps constrains how far we can reorder or pipeline computation.

9.10 Charm++ Implementation Idea (LOIMOS) and Bipartite Modeling

To make parallelization more concrete, the lecture introduces a Charm++-based implementation called **LOIMOS**, developed in the instructor's group. The model is naturally expressed as a **bipartite graph**:

- One partition represents **people**.
- The other partition represents **locations**.
- Edges represent visit relationships or interactions.

In Charm++, this is represented as:

- An array of *people chares*.
- An array of *location chares*.

Each chare can encapsulate many logical entities:

- A people chare can store many individual people.
- A location chare can store many locations.

The programmer can choose:

- How many chares to create (e.g., 100 location chares for 10,000 locations, 1,000 people chares for 1 million people).
- This decouples the number of logical objects from the number of physical processors.

The Charm++ runtime decides how to map chares to processors or cores.
By contrast, in MPI:

- Developers must manually co-locate people and locations on ranks.
- They must explicitly manage who sends messages to whom based on rank ownership.

With Charm++, once people and locations are decomposed into chares, the parallel algorithm:

- Mirrors the sequential structure.
- Adds an outer loop over people chares and location chares.
- Uses remote method invocations between people chares and location chares to represent visits and exposures.

9.11 Using Libraries Instead of Rewriting Everything

After epidemiology, the lecture shifts to a general HPC programming principle: **use libraries** whenever possible.

9.11.1 The Role of Libraries

As applications become more complex, it is rarely efficient to:

- Implement all core numerical and I/O routines from scratch.

Libraries:

- Are typically developed by domain experts.
- Are heavily optimized for performance on different architectures.
- Are widely tested and maintained.
- Improve portability across machines and vendors.

9.11.2 Library Ecosystem Examples

The lecture mentions several library categories:

- **I/O**: HDF5, parallel NetCDF.
- **Dense linear algebra and FFTs**: BLAS, LAPACK, FFTW, and vendor-optimized suites like Intel MKL.
- **Sparse linear algebra and solvers**: Hypre, PETSc, Trilinos family libraries.
- **Partitioning and load balancing**: METIS, Scotch, Zoltan.

Libraries come from:

- Academia.
- National laboratories.
- Industry vendors.

There is no single canonical index of all libraries. Practitioners:

- Discover them through community references, documentation, and targeted Google searches.

9.12 N-Body Problems in Computational Astronomy

9.12.1 Problem Definition

The final topic is **n-body simulation** from computational astronomy and astrophysics. Here, we:

- Simulate many particles (e.g., stars, galaxies, dark-matter parcels).
- Model their motion under gravitational interactions.

Key distinctions from MD:

- Distances can span light years or larger.
- Particle counts can be extremely large (billions or more).

The naive approach:

- Computes pairwise forces between every pair of particles.
- Has $O(n^2)$ complexity, which is infeasible at these scales.

9.13 Data Decomposition Strategies for Parallel N-Body

9.13.1 Basic Decomposition

A simple parallel strategy:

- Assign roughly n/p particles to each of p processes.
- Each process computes forces for its own particles.

However, a naive 1D split of the particle array ignores spatial structure, leading to poor locality and more communication. To improve, we use space-aware decompositions.

9.13.2 Space-Filling Curves

Space-filling curves (SFCs) provide one approach:

- Map higher-dimensional positions (2D/3D) to a 1D ordering that preserves locality.
- Examples: Hilbert curves, Z-order (Morton) curves.

Procedure:

1. Compute the 1D SFC index for each particle.
2. Sort particles by this index.

3. Partition the resulting 1D array into contiguous blocks of size $\sim n/p$.

This tends to:

- Group particles that are close in space onto the same or nearby processes.
- Reduce long-distance interactions compared to naive 1D splitting.

9.13.3 Orthogonal Recursive Bisection (ORB)

Orthogonal Recursive Bisection recursively partitions the simulation domain along coordinate axes:

- At each step, choose a coordinate axis (e.g., x, then y, then z).
- Split the domain into two regions so that each half has equal particle count (or equal estimated cost).
- Continue recursively until there are p partitions with roughly n/p particles each.

ORB naturally corresponds to a tree structure:

- Each node in the tree represents a subdomain.
- Leaf nodes represent final partitions mapped to processes.

9.13.4 Quadtrees and Octrees

Quadtrees (2D) and **octrees** (3D) generalize recursive spatial subdivision:

- In 2D, each node splits into 4 children (quadrants).
- In 3D, each node splits into 8 children (octants).

Some child regions may be empty; these are often represented as dummy nodes or omitted, depending on the code. A stopping condition is chosen so that:

- Each leaf contains roughly n/p particles (not necessarily one per cell).

Once the tree is built:

- Each process is assigned one or more leaf regions.
- Each process owns all particles in its assigned leaves.

Tree Metadata and Replication

Processes often need a shared view of the upper levels of the tree (including some internal nodes and possibly empty/dummy nodes) to:

- Understand the global partitioning.
- Decide which remote processes to contact for data or interactions.

Thus:

- Some levels of the tree may be replicated across all processes.
- Only deeper leaf data is stored in distributed fashion.

9.14 Reducing Computation: Barnes–Hut and Beyond

After decomposition, we still face potential $O(n^2)$ interactions if every particle interacts explicitly with all others. The lecture discusses methods to reduce this cost.

9.14.1 Barnes–Hut Algorithm

The **Barnes–Hut** algorithm is a classic spatial tree-based method. In 3D, it commonly uses an octree.

Key idea:

- Use the tree to separate *nearby* and *far-away* regions.
- For nearby particles, compute explicit pairwise interactions.
- For far-away clusters of particles, approximate their combined effect with a *single point mass* at the cluster's center of mass.

Implementation steps:

- Build an octree over the domain.
- Compute center-of-mass and total mass per internal node.
- For each particle, traverse the tree:
  - If a cell is sufficiently far away (angle criterion), treat it as a single aggregate.

  - Otherwise, open the cell and inspect its children.

This reduces the amount of work:

- Distant particle groups are aggregated.
- Complexity becomes roughly $O(n \log n)$ for many distributions, rather than $O(n^2)$.

9.14.2 Fast Multipole Method (FMM)

The lecture briefly mentions the **Fast Multipole Method (FMM)**:

- FMM uses multipole expansions to approximate distant interactions at multiple levels.
- It can achieve near-linear-time behavior $O(n)$ in favorable cases.
- It is more complex to implement correctly than Barnes–Hut.

9.14.3 Particle Mesh (PM) and Particle–Particle Particle Mesh (P3M)

The **Particle Mesh (PM)** and **Particle–Particle Particle Mesh (P3M)** families:

- Split work between short-range particle–particle computations and long-range computations on a mesh.
- Use grid-based methods and Fourier transforms for long-range interactions, conceptually similar to particle-mesh schemes discussed in MD.

9.14.4 Practical Takeaway and Exam Guidance

A pragmatic trade-off:

- Barnes–Hut is simpler and often sufficient for moderate scales (e.g., up to hundreds of processes).
- More advanced methods like FMM can be worthwhile at very large scales, but their complexity is higher.