Computação em Larga Escala

Introduction to High-Performance Computing

Eurico Pedrosa

António Rui Borges

Universidade de Aveiro - DETI

2025-02-09

Introduction to HPC

High-Performance Computing (HPC)

High-Performance Computing (HPC) has evolved significantly in recent years, adapting to new technologies and methodologies to address computational challenges. **HPC** encompasses not just the hardware architecture but also the associated software tools, programming platforms, and parallel programming paradigms. It focuses on leveraging multiple tightly coupled processors or computer clusters to execute computationally intensive tasks with high throughput and efficiency.

High-Performance Computing (HPC)

The field has seen a paradigm shift with the rise of **heterogeneous computing** architectures, such as those combining *CPUs* and *GPUs*. *GPUs*, with their inherent support for data parallelism, have proven highly effective for certain types of computations, particularly those with high levels of data parallelism.

These architectures necessitate **new programming approaches**, such as *CUDA* for *GPUs*, enabling **fine-grained parallelism**. This shift has also influenced the design of parallel programming paradigms, emphasizing the need for explicit management of parallelism to fully exploit these advanced systems.

What is HPC?

- **Definition of HPC:**
  - Use of powerful computing resources for solving complex problems.
  - Involves tightly coupled processors or clusters running tasks concurrently.
- **Core Components:**
  - *Hardware Systems*: Multi-core CPUs, GPUs, and clusters.
  - *Software Tools*: Compilers, libraries, debuggers.
  - *Programming Paradigms*: Parallel programming frameworks (MPI, OpenMP, CUDA, etc.).
- **Goal:**
  - Deliver high throughput and efficiency for computationally intensive tasks.

Why is HPC Important?

- Solves **large-scale computational problems** in:
  - Cosmology, astrophysics and astronomy.
  - Computational chemistry, biology and engineering.
  - Computer science, AI/ML.
  - Earth sciences and materials.
  - Weather forecasting.
  - Geographic information science and technology
  - Global security
  - Nuclear fusion
- Enables **innovation** in science, technology, and engineering.
- Drives **advancements** in hardware and software.

Evolution of HPC

- **Early HPC:**
  - Based on single-processor systems and vector processors.
- **The Paradigm Shift:**
  - Emergence of **CPU-GPU Heterogeneous Architectures**:
    - GPUs excel at data parallel tasks.
    - Examples: NVIDIA CUDA, AMD ROCm.
  - Development of fine-grained parallelism and hybrid computing models.
- **Impact on Programming:**
  - New paradigms to manage parallelism effectively (explicit data transfers, task distribution).

HPC Today

- **HPC systems now feature:**
  - Exascale computing (systems capable of 10^{18} calculations per second).
  - AI/ML integration with HPC.
  - Cloud-based HPC for accessibility.
- **Programming Frameworks:**
  - MPI, OpenMP, CUDA, and hybrid models.
- **Trends:**
  - Emphasis on energy efficiency and scalability.

HPC Today

Top 10 - November 2024

Rank	Name	Cores	Rmax TFlops/s	Rpeak TFlops/s	Power (kW)	Country
1	El Capitan	11,039,616	1,742,000.00	2,746,376.09	29,580.98	United States
2	Frontier	9,066,176	1,353,000.00	2,055,716.99	24,607.00	United States
3	Aurora	9,264,128	1,012,000.00	1,980,006.00	38,698.36	United States
4	Eagle	2,073,600	561,200.00	846,835.20		United States
5	HPC6	3,143,520	477,900.00	606,965.76	8,460.90	Italy
6	Supercomputer Fugaku	7,630,848	442,010.00	537,212.00	29,899.23	Japan
7	Alps	2,121,600	434,900.00	574,841.28	7,124.00	Switzerland
8	LUMI	2,752,704	379,700.00	531,505.15	7,106.82	Finland
9	Leonardo	1,824,768	241,200.00	306,311.16	7,493.74	Italy
10	Tuolumne	1,161,216	208,100.00	288,881.05	3,386.62	United States

https://top500.org/lists/top500/2024/11/

Fugaku Supercomputer

Architecture of a Parallel Machine

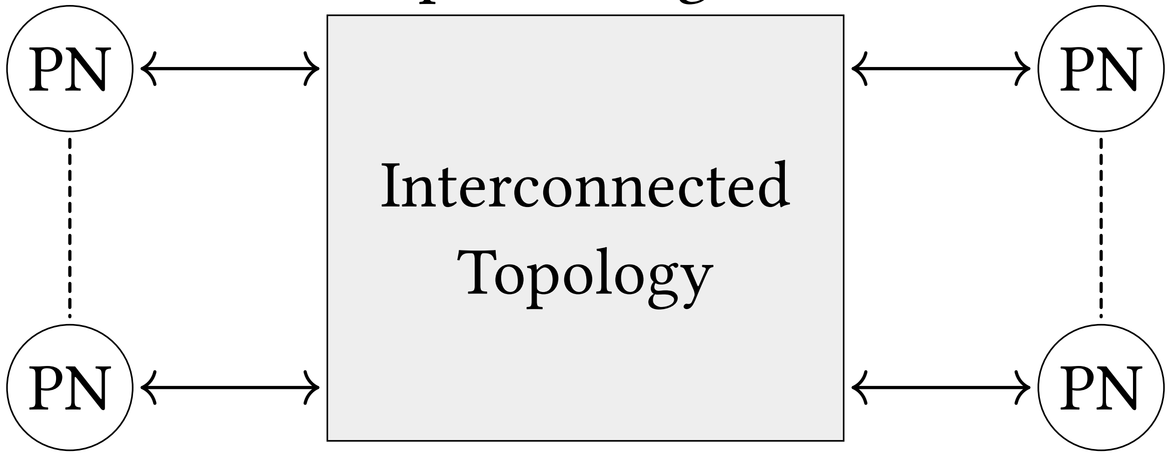
Scalability

HPC systems today primarily operate as *distributed memory parallel architectures*, often organized into extensive clusters of processing nodes (**PN**) interconnected via sophisticated **interconnected (network) topologies**. These configurations are designed to prioritize **scalability**, enabling system performance to scale effectively with the addition of new nodes. This architecture facilitates distributed computation while addressing communication and data synchronization challenges inherent in such systems

Interconnection Topologies

The choice of interconnection topology depends on the application requirements, scalability needs, and the physical constraints of the **HPC** system. Interconnection topology design focuses on two primary concerns:

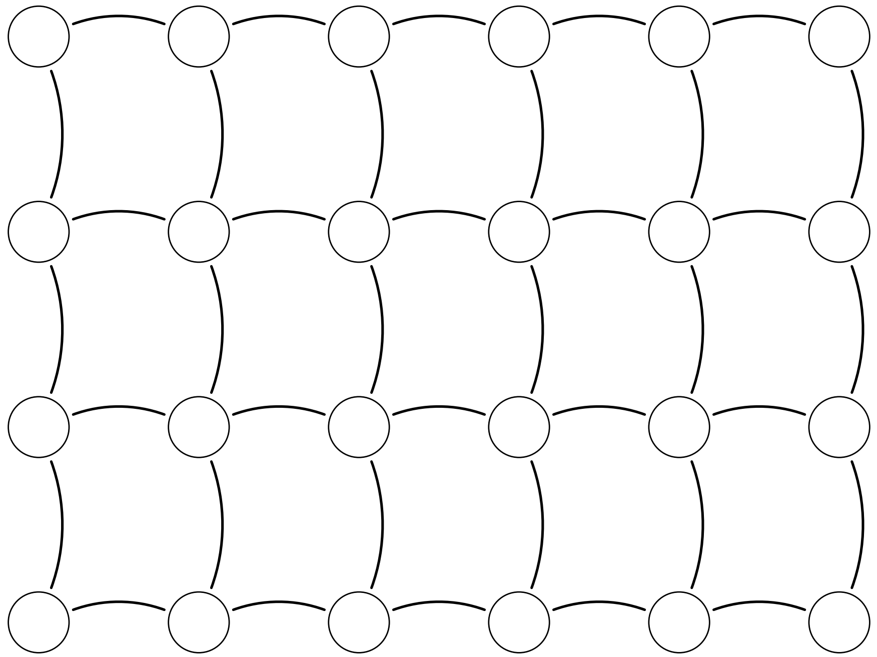
- Minimizing the number of connections per node as the cluster scales.
- Ensuring that communication time and bandwidth remain constant regardless of the number of processing nodes.



Interconnection Topologies

Mesh Topology

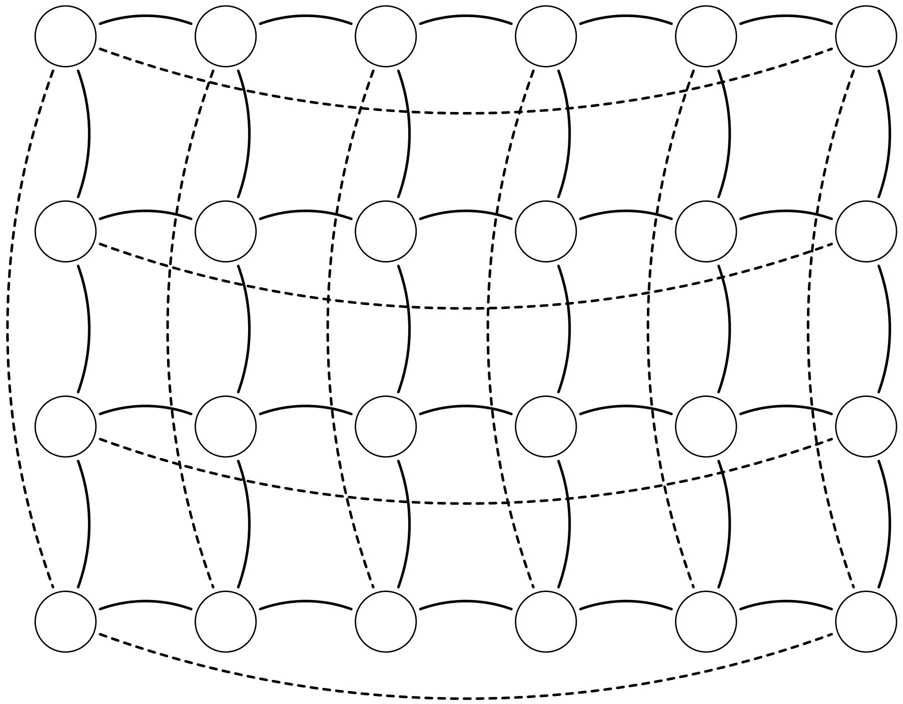
- Nodes are arranged in a grid-like structure where each node connects directly to its neighbors.
- Simple design, efficient for localized communication.



Interconnection Topologies

Torus Topology

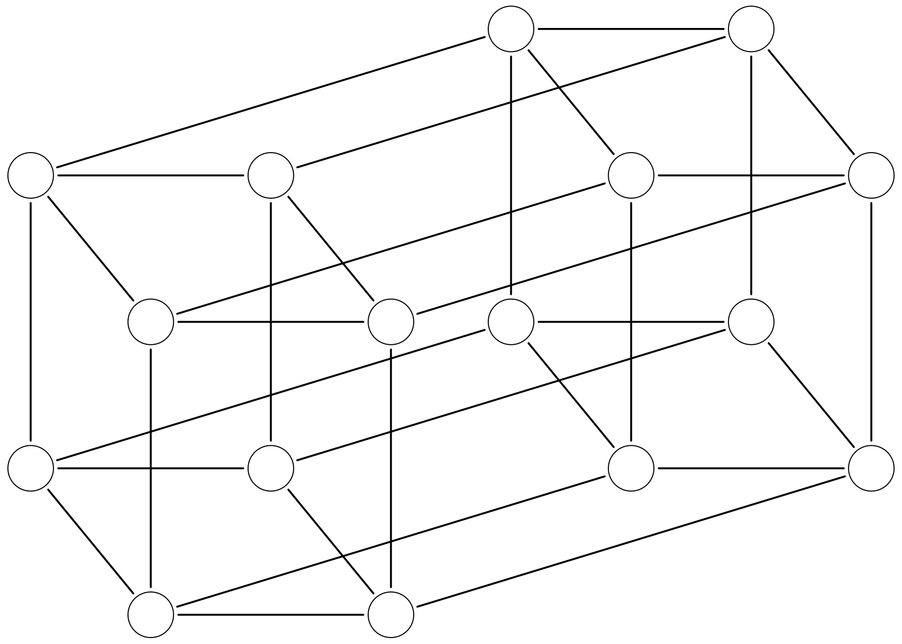
- Extends the mesh by connecting the edges of the grid, creating a wrap-around effect.
- Reduces communication distance, improving performance for large-scale problems.



Interconnection Topologies

Hypercube Topology

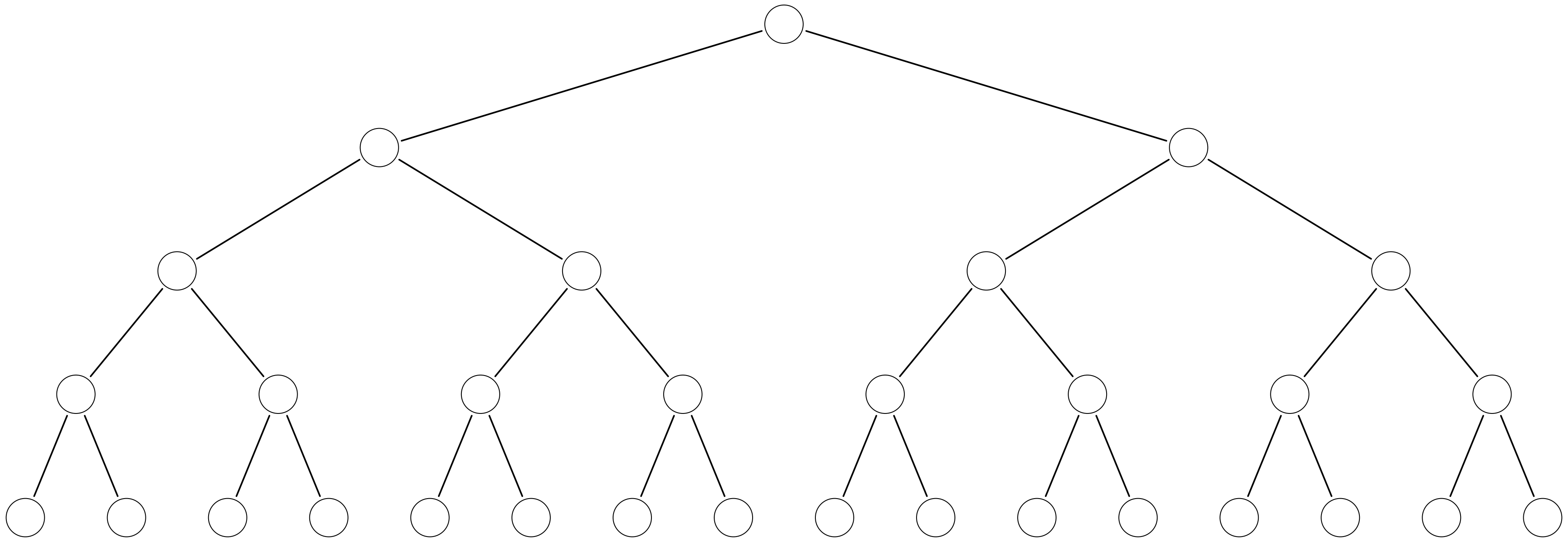
- Nodes are arranged in a binary cube structure, with each node connected to others differing by one binary digit.
- Highly scalable and efficient for data-intensive applications.



Interconnection Topologies

Tree Topology

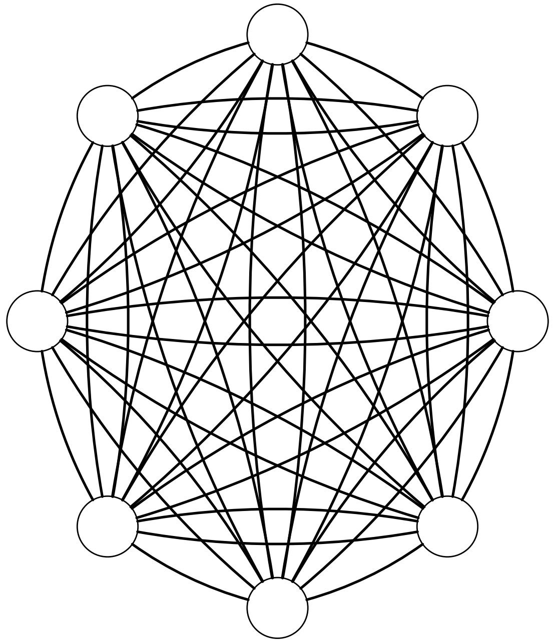
- Nodes are connected hierarchically, forming a tree structure.
- Simplifies routing and is efficient for collective communication.



Interconnection Topologies

Fully Connected Topology

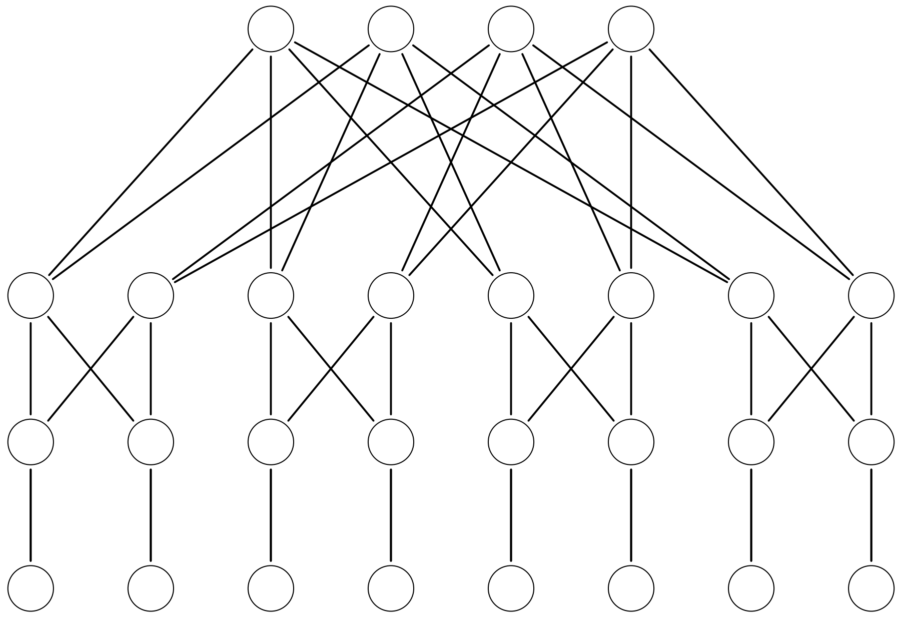
- Every node is directly connected to every other node.
- Maximum throughput and minimal latency for small systems.
- Impractical for large-scale implementations due to the hardware cost.



Interconnection Topologies

Fat-Tree Topology

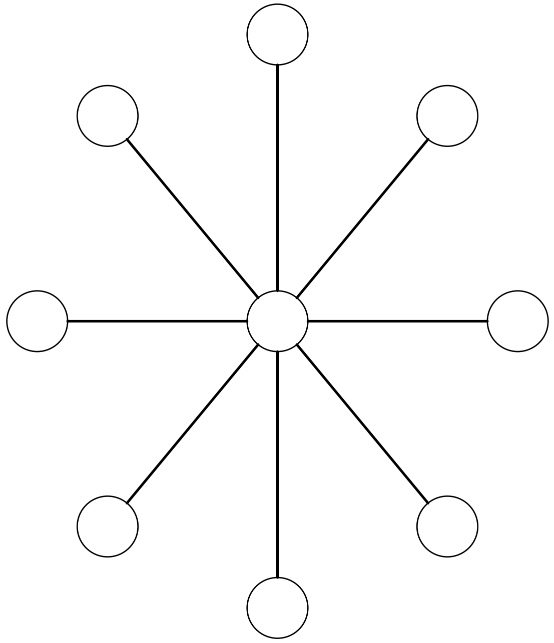
- A variation of the tree topology with additional bandwidth in higher levels of the hierarchy to prevent bottlenecks.
- Balances scalability and bandwidth.



Interconnection Topologies

Star Topology

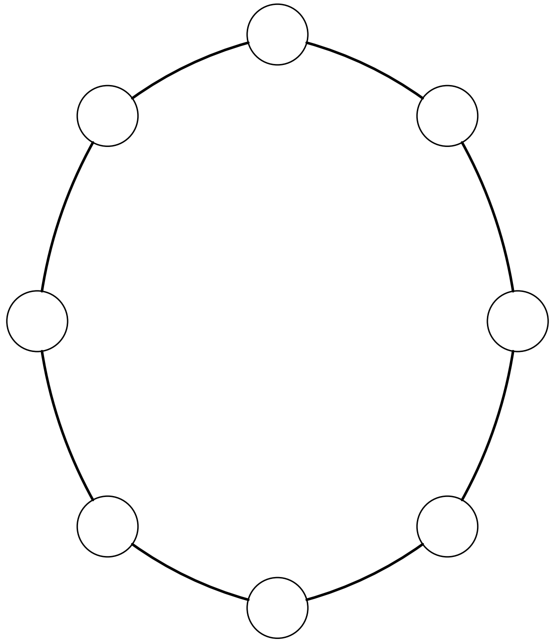
- All nodes are connected to a central hub.
- Simple implementation for small-scale systems.
- Rare but useful in small-scale clusters where central management is key.



Interconnection Topologies

Ring Topology

- Nodes are connected in a circular fashion, with each node connected to its immediate neighbors.
- Simple and cost-effective for moderate-sized systems.



Interconnection Topologies

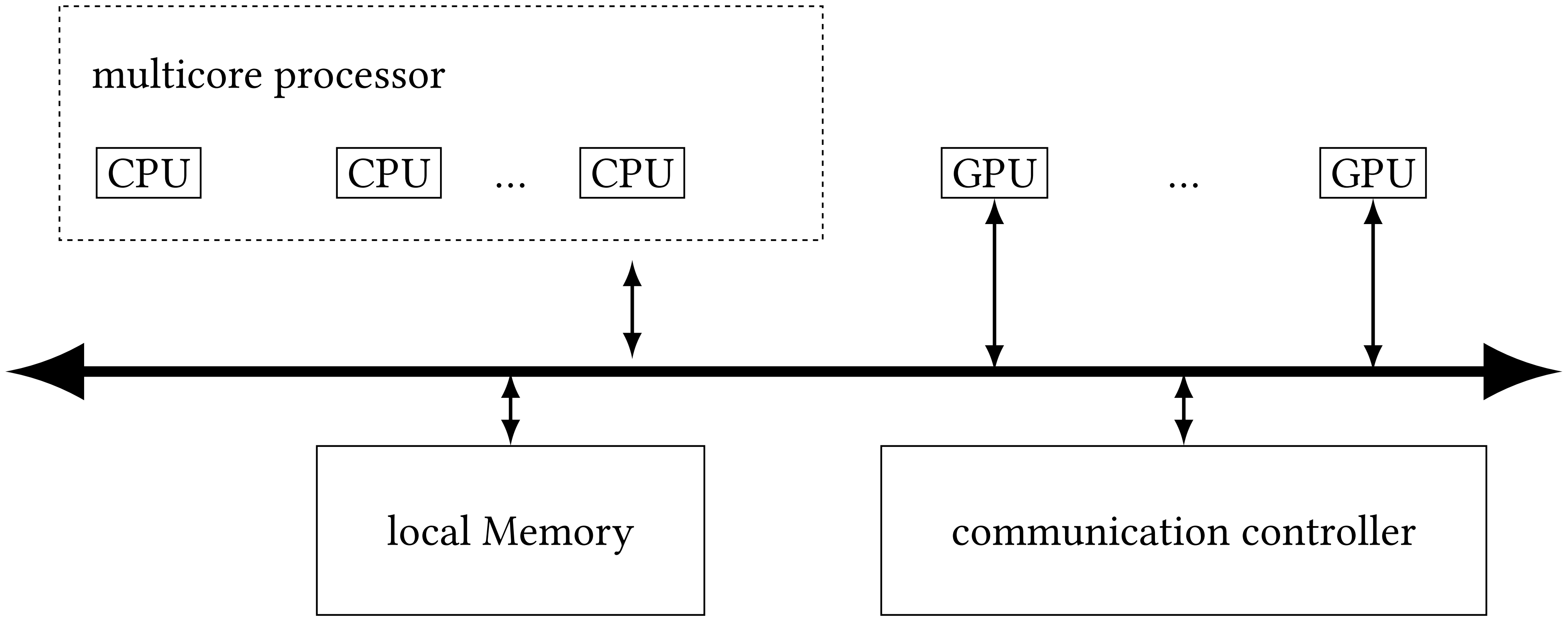
Mesh, *torus*, and *fat-tree* topologies are particularly prevalent in modern supercomputers due to their balance of simplicity and performance scalability.

In a torus mesh topology, each node has four fixed connections, with communication time increasing up to $\sqrt{n}$, where n is the total number of nodes. On the other hand, the hypercube topology requires $\log_2(n)$ connections per node, with communication times scaling up to ($\log_2(n)$), where ($n = 2^k$). Both designs utilize point-to-point connections, maintaining fixed bandwidth, although communication times depend on the locations of the communicating nodes.

Interconnection Topologies

The fat tree topology takes a different approach by using a hierarchical structure designed to maintain consistent bandwidth across all bisections. This topology ensures that all processing nodes can transmit at line speed, provided packets are distributed uniformly across available paths. With k-port switches, the fat tree allows for excellent scalability, connecting $\left(\frac{k^3}{4}\right)$ processing nodes using just one connection per node. This hierarchical design makes the fat tree particularly suitable for large-scale systems, balancing bandwidth and scalability effectively.

Processing Node



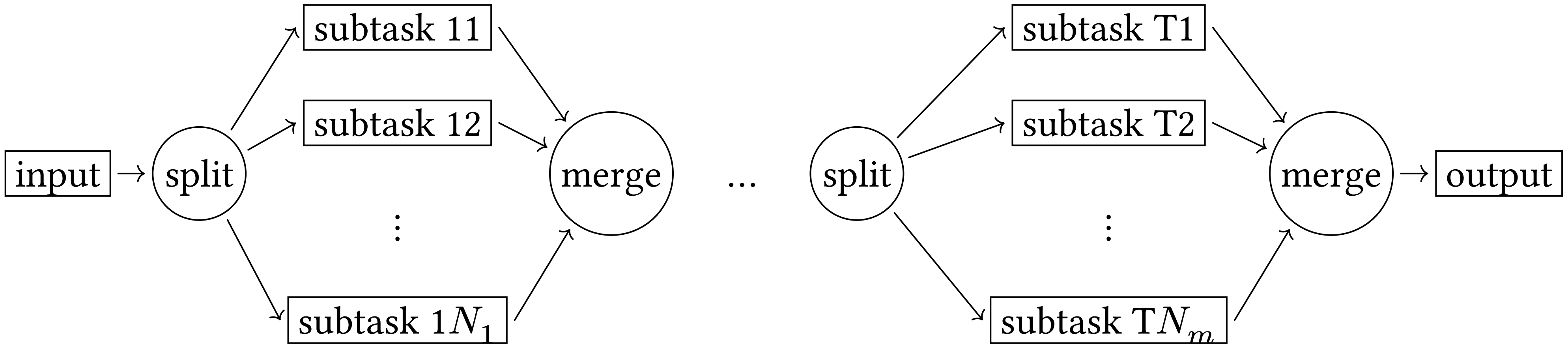
Processing Node

- A processing node has multi-core CPUs and many-core GPUs, enabling **heterogeneous computing** for better performance.
- **CPU manages** the environment, code, and data before offloading computation-heavy tasks to the **GPU**.
- **GPU computing** complements, not replaces, **CPU computing**—CPUs handle dynamic tasks, while GPUs excel at **highly parallel workloads** with simple flow control.
- **Heterogeneous computing** evolved as CPUs and GPUs complement each other, maximizing application performance.

Parallel Decomposition

Parallel Decomposition

- Typically, **parallel decomposition is data-driven**!
- Data chunks pass through a **T-stage pipeline**, splitting further at each stage for **independent processing**, with possible **reshuffling** between stages.



Parallel Decomposition

- **Granularity** in parallel algorithms depends on the hardware and is categorized into:
  - **Fine-grained**: SIMD – operations on multiple data at the **variable level**.
  - **Medium-grained**: MIMD (shared memory) – parallelism at the **thread level**.
  - **Coarse-grained**: MIMD (distributed memory) – parallelism at the **process level**.
- **HPC algorithms** often blend all three levels for optimal performance.

- SIMD (Single Instruction -- Multiple Data)
- MIMD (Multiple Instruction -- Multiple Data

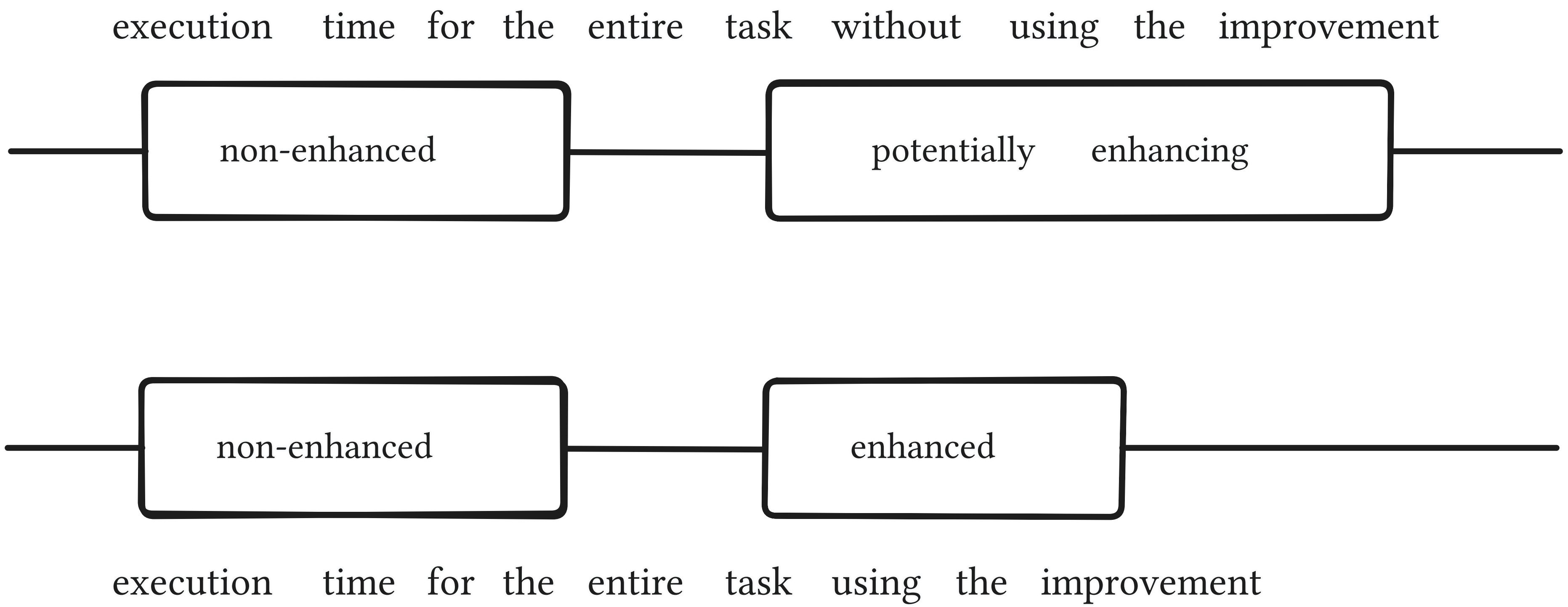
Amdahl's Law

- **Amdahl's Law (1967)** estimates **performance gain** by showing that **speedup** is limited by the fraction of execution time affected by the **faster mode**.
- The overall speedup S_{overall} is calculated by the formula

$$S_{\text{overall}} = \frac{1}{(1 - P) + \frac{P}{N}}$$

where **P** is the parallelizable fraction and **N** is the speedup factor.

Amdahl's Law


execution time for the entire task without using the improvement
non-enhanced
potentially enhancing
non-enhanced
enhanced
execution time for the entire task using the improvement

Amdahl's Law

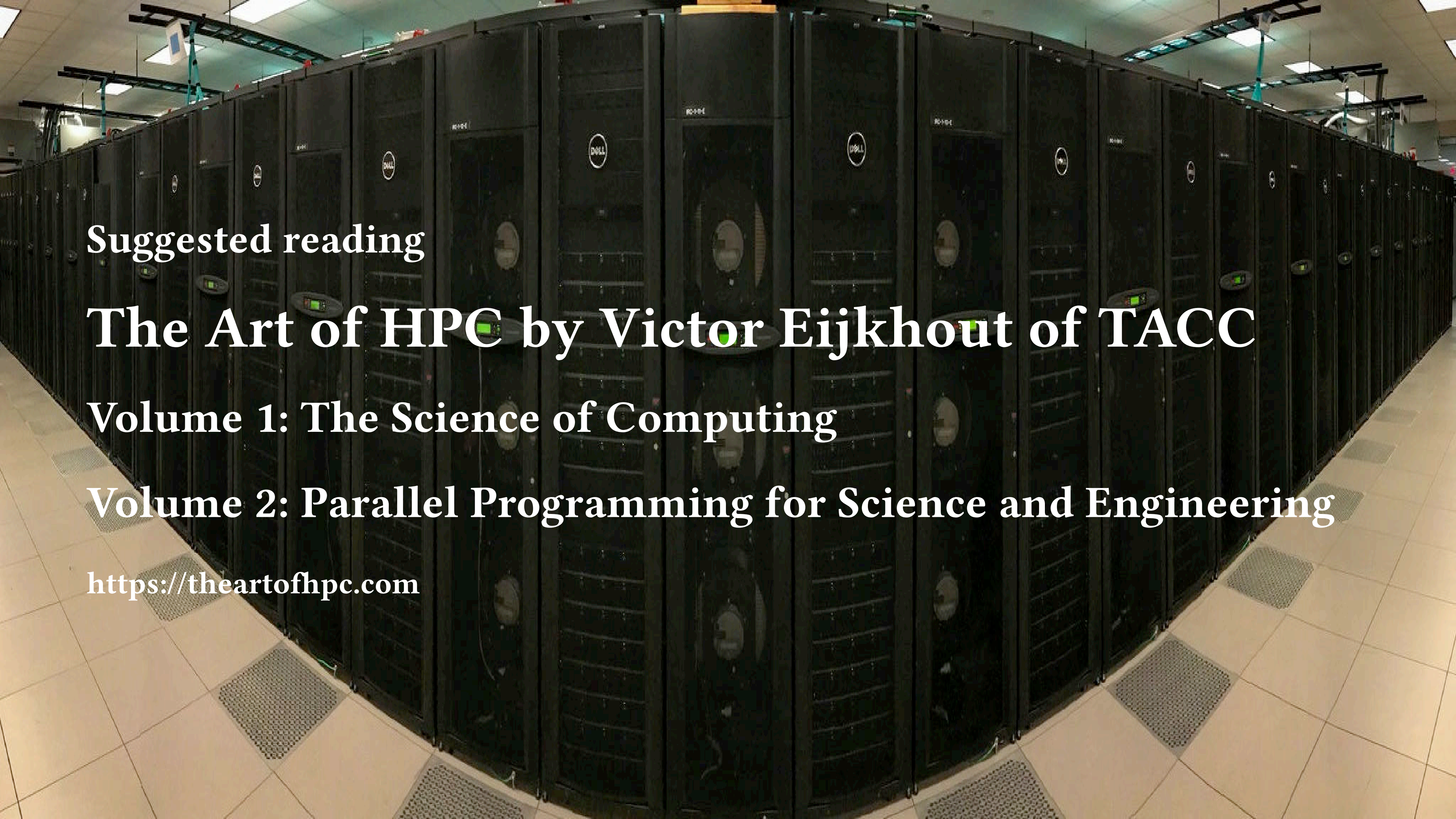
- **Amdahl's Law** sets a **limit** on speedup—no matter how fast a section runs, the **non-parallel fraction** constrains performance.
- **Parallel decomposition** is often **non-trivial** and requires a **different approach** than the original sequential design.
- In **distributed memory systems**, **communication overhead** further limits speedup, eventually leading to **diminishing returns**.

$$S_{\text{overall}} = \frac{1}{(1-P) + C + \frac{P}{N}}$$

where **C** is the communications overhead.

Tools for Parallel Programming

- **Parallel applications** will be written in **C/C++** using:
    - **std::thread**: for **multithreading** in shared memory (**medium-grained parallelism**).
    - **MPI**: for **multiprocessing** in distributed memory (**coarse-grained parallelism**).
    - **CUDA C/C++**: for **fine-grained parallelism** in **CPU-GPU architectures**.



Suggested reading

The Art of HPC by Victor Eijkhout of TACC

Volume 1: The Science of Computing

Volume 2: Parallel Programming for Science and Engineering

https://theartofhpc.com