Parallel Algorithms

Professor

Dr. Masoumeh Damroudi

Author

Mohammad Khorshidi

Course Overview

In this lecture note, I have documented my personal interpretations and understanding of the material. Please note that there may be inaccuracies, and it is recommended to consult primary and authoritative sources for complete and accurate information. Additionally, I welcome any contributions to improve and refine this note. If you are reading this and would like to offer suggestions, feel free to do so via pull requests on my **GitHub** or by emailing me at **mohammad_khorshidi@outlook.fr**.

0.1 Course Structure

0.2 Instructor's Approach

0.3 Grading and Evaluation

The grading structure for this course is as follows, with students expected to earn their grades based on the criteria outlined below:

- **Presentation:** 4 points will be awarded for presenting and explaining a paper related to one aspect of parallelization, accompanied by its relevant implementation.
- **Midterm Exam:** 4 points will be given for the midterm exam.
- **Final Exam:** 12 points will be awarded for the final exam.

Additionally, students are expected to attend all classes and give due attention to the assignments that will be provided throughout the semester.

0.4 Course Materials and Resources

Contents

Chapter 1

Introduction

In today's world, with the increasing volume of data and the need for faster computations, parallel algorithms have emerged as an effective method to enhance the performance of computing systems. Parallel algorithms allow us to divide a problem into smaller sub-problems, each of which can be processed simultaneously by multiple processors. This not only speeds up processing but also optimizes the use of hardware resources.

In this lecture note, we will explore fundamental concepts of parallel processing, computational models such as the PRAM model, and interconnection structures like network links. We will also delve into the principles of designing and analyzing parallel algorithms, covering key metrics such as performance, speedup, and scalability. Our goal is to provide a comprehensive understanding of the challenges and opportunities in the field of parallel algorithms.

Parallel algorithms play a crucial role in improving the performance of complex and resource-intensive computations. The primary goal of parallel algorithms is to divide a large problem into smaller, independent sub-problems, which can be solved simultaneously by multiple processors. This technique is essential in multi-core and distributed computing environments such as supercomputers and large computing clusters.

The need for parallel computation arises from the requirement to process large amounts of data or perform complex computations in a reasonable amount of time. A parallel computer, which consists of multiple processing units or processors, enables this by dividing a large problem into smaller sub-problems. Each of these sub-problems is solved independently and simultaneously by separate processors. The results are then combined to solve the original problem more efficiently.

This approach, known as parallel computing, leverages the power of multiple processors to work on different parts of a problem at the same time. By doing so, it accelerates the overall computation and makes optimal use of hardware resources. Algorithms that are specifically designed to utilize parallel computing, called parallel algorithms, are key to solving complex problems faster and more efficiently by running them on parallel computer architectures.

1.1 Computational Models

In any computing system, whether sequential or parallel, the instructions are executed based on the flow of data. The execution flow dictates the sequence in which the tasks are carried out by the computer and what needs to be done at each stage of execution.

The flow of data through the instructions influences how they operate. Understanding the correct computational model is crucial for designing parallel algorithms. These models determine the architecture and behavior of the system, impacting the performance and efficiency of parallel computation.

Flynn, in 1966, introduced a taxonomy of computer architectures based on instruction and data streams. The four major classes of this classification are:

- **SISD (Single Instruction, Single Data):** A single instruction is executed on a single data stream.

- **MISD (Multiple Instruction, Single Data):** Multiple instructions are executed on a single data stream.
- **SIMD (Single Instruction, Multiple Data):** A single instruction is executed on multiple data streams simultaneously.
- **MIMD (Multiple Instruction, Multiple Data):** Multiple instructions are executed on multiple data streams simultaneously.

		Instruction Streams	
		one	many
Data Streams	one	**SISD** traditional von Neumann single CPU computer	**MISD** May be pipelined Computers
	many	**SIMD** Vector processors fine grained data Parallel computers	**MIMD** Multi computers Multiprocessors

Figure 1.1: A comparison between SISD, MISD, SIMD, and MIMD architectures

1.1.1 SISD Architecture

The Single Instruction Single Data (SISD) architecture, introduced by John von Neumann in the late 1940s, processes one instruction at a time over a single data stream. This sequential nature means it doesn't support parallel processing, requiring only one processor to execute the tasks.

An SISD computing system is a uniprocessor machine which is capable of executing a single instruction, operating on a single data stream. In SISD, machine instructions are processed in a sequential manner and computers adopting this model are popularly called sequential computers. Most conventional computers have SISD architecture. All the instructions and data to be processed have to be stored in primary memory.

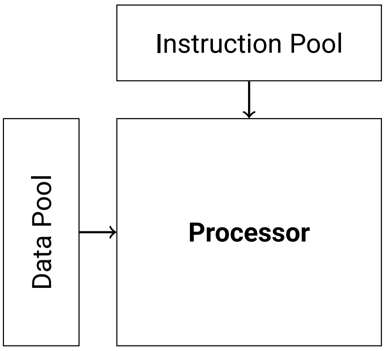


Figure 1.2: A conceptual diagram of SISD architecture.

In the SISD model, the algorithm works in a serial fashion. For example, to sum a sequence of n numbers, the algorithm accesses the memory n times and performs n-1 additions. The time complexity for this approach can be expressed as:

$$O(n)$$

Where:

- n: The number of elements to process.
- $O(n)$: Reflects that the algorithm's running time scales linearly with the size of the input.

In this architecture, there is no room for parallelism, and each operation occurs sequentially, requiring every data element to be processed one at a time. For a sum operation on n numbers:

- Memory accesses: n memory fetches.
- Additions: n-1 addition operations.

Thus, the total computational complexity is linear, i.e., $O(n)$, representing the number of accesses and operations required to complete the task. For this type of architecture, the performance is highly dependent on the efficiency of the processor and memory access speed, as there is no way to split the workload across multiple processors.

1.1.2 MISD Architecture

The Multiple Instruction Single Data (MISD) is a less common architecture where multiple processors execute different instructions but operate on the same data stream simultaneously. This approach is primarily used in specialized applications like fault-tolerant systems or systems requiring redundancy. The MISD model is structured to handle cases where having multiple processors work on the same data but with different operations is crucial.

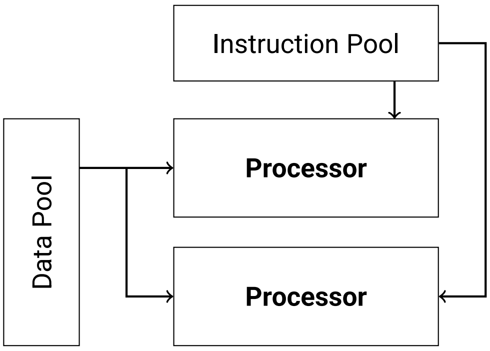


Figure 1.3: A conceptual diagram of MISD architecture.

In MISD architecture, each processor has its own control unit that dictates the operations performed. Unlike SIMD, which focuses on parallel data processing, MISD involves parallel instruction execution on a single data stream. To illustrate, imagine a real-time safety system where different sensors provide data that multiple processors analyze for distinct purposes:

- **Processor 1** checks if the temperature is within a safe range.
- **Processor 2** monitors the pressure of the system.
- **Processor 3** assesses voltage levels.

All processors work on the same data stream from the sensors but perform different tasks to ensure system reliability and fault tolerance. If any processor detects a problem, the system can react accordingly to prevent failure.

In a typical MISD system:

- Multiple processors (N) receive the same input data simultaneously.
- Each processor is responsible for executing a unique instruction on that shared data stream.
- The processors operate in parallel, but the parallelism applies to the instruction execution, not to the data.

This kind of architecture is rare due to its niche applicability, often used in scenarios where redundancy is crucial, such as avionics or control systems, where the same data needs to be processed in multiple ways to ensure accuracy and safety.

The time complexity of an MISD system depends heavily on the operations performed by each processor. Each processor runs independently, and the overall time to process the data is determined by the slowest processor, similar to SISD's sequential operation but distributed across different instruction sets.

For N processors with execution times $T_1, T_2, \ldots, T_N$, the overall time complexity is governed by the maximum time:

$$T_{total} = max(T_1, T_2, \ldots, T_N)$$

This means the overall system performance is constrained by the processor taking the longest to complete its task.

1.1.3 SIMD Architecture

The Single Instruction Multiple Data (SIMD) architecture enables a single instruction to operate simultaneously across multiple data streams. In this model, one instruction, such as an addition or comparison, is applied to different sets of data in parallel, which significantly enhances processing efficiency.

In an SIMD system, there are N identical processors, all controlled by a single stream of instructions from a central control unit. Each processor executes the same instruction but operates on its own separate data stream. Therefore, N data streams exist, with one data stream assigned to each processor.

Communication between processors for data transfer in SIMD computers can occur in two main ways:

1. **Shared memory:** All processors access a common memory space to fetch or store data.
2. **Interconnection networks:** A specialized network connects the processors to facilitate data transfer between them.

SIMD architecture is particularly well-suited for tasks requiring parallel data processing, such as graphical computations or heavy workloads where operations need to be performed on multiple data elements simultaneously. For example, in image or video processing, SIMD can process multiple pixels or image sections in parallel, leading to a significant speedup in computation.

> Shared Memory (SM) SIMD Computers

In Shared-Memory (SM) SIMD computers, the communication between processors occurs through a shared memory space. This model is often referred to as the Parallel Random-Access Machine (PRAM) model, where multiple processors work in parallel, and data exchange happens via common memory access.

When a processor (say **processor i**) wants to send data to another processor (**processor j**), the process follows two key steps:

1. **Processor i** writes the data into a known location in the shared memory, which **processor j** can access.
2. **Processor j** then reads the data from that specific location.

SIMD shared-memory computers are categorized based on how processors can concurrently access the same memory location. These categories determine how simultaneous read and write operations are handled, as follows:

- **Exclusive-Read, Exclusive-Write (EREW):** No two processors can read from or write to the same memory location at the same time.
- **Concurrent-Read, Exclusive-Write (CREW):** Multiple processors can read from the same memory location simultaneously, but only one can write to a memory location at any given time.
- **Exclusive-Read, Concurrent-Write (ERCW):** Only one processor can read from a memory location at a time, but multiple processors can write to the same location simultaneously.
- **Concurrent-Read, Concurrent-Write (CRCW):** Both read and write operations can be performed by multiple processors on the same memory location simultaneously.

These categories define the degree of parallelism and the restrictions on data sharing among processors in a shared-memory SIMD system.

Example

Searching for element x in an unordered file with n distinct entries using different SIMD models.

Using the EREW Model with N processors where $N \geq n$

Phase 1:
Broadcasting the value of x to all processors in $O(N \log N)$ time

- Processor P1 sends x to P3 and P2 sends x to P4, and so on.
- This process continues until all processors have the value x.

Phase 2:
Simultaneous search in sub-files by processors in $O(\frac{N}{n})$ time.

- Processor P1 searches the first $\frac{n}{N}$ data.
- Processor P2 searches the second $\frac{n}{N}$ data.
- And so on for all N processors.

Initially, F is set to false. The first processor to find x sets F to true. All processors check F at each step, and if F becomes true, the search ends. The broadcast of F takes $O(\log N)$ time per step. In the worst case, the EREW model requires $+O(N \log N(N \log(\frac{N}{n})))$ time.

Using the ERCW Model with N processors where $N \geq n$

Phase 1:
Broadcasting x to all processors in $O(\log N)$ time.

- Similar to CREW, all processors can read x from a shared location.

Phase 2:
Simultaneous search in sub-files by processors in $O(\frac{N}{n})$ time.

- Again, processors search different parts of the data in parallel.
- However, here multiple processors can concurrently write to the same memory location. If one processor finds x, it writes to the shared F location. Other processors can also attempt to write, but only one write will succeed.

With concurrent writes, the time complexity remains $+O(\log N(N \log(\frac{N}{n})))$, though write conflicts may slightly impact performance.

Using the CREW Model with N processors where $N \geq n$

Phase 1:
Broadcasting the value of x to all processors in parallel using a single write per memory location.

- Since concurrent reads are allowed, all processors can read the value of x from a shared memory location at the same time, significantly speeding up the process.
- This reduces the broadcast time to $O(\log N)$ since only one write operation is needed.

Phase 2:
Simultaneous search in sub-files by processors in $O(\frac{N}{n})$ time.

- Similar to the EREW model, P1 searches the first $\frac{n}{N}$ data, P2 searches the second, and so on.
- As soon as one processor finds x, it updates F to true, and the search ends.

With concurrent reads, the worst-case time complexity is reduced to $N \log(\frac{N}{n}) + O(\log N)$.

Using the CRCW Model with N processors where $N \geq n$
Phase 1:
Broadcasting x to all processors in constant time $O(1)$.

- In this model, both reads and writes are performed concurrently, so all processors can obtain the value of x simultaneously from the shared memory.

Phase 2:
Simultaneous search in sub-files by processors in parallel.

- Each processor searches its assigned portion of the data. Once a processor finds x, it writes true to F, and the search concludes. Since multiple writes are allowed at the same memory location, any processor finding x can update F without delay.

The worst-case time complexity for CRCW is minimized to $O(\frac{N}{n})$ because concurrent reading and writing reduce the overhead.

When designing shared-memory SIMD computers, one of the main challenges is how to efficiently manage access to memory locations shared by multiple processors. The key concern is the cost of routing data between processors and memory, which scales with the number of processors N and memory locations M. Now the problem overview raised is below:

- Memory circuits need to be established to create pathways between the processors and the shared memory locations.
- If each processor has its own dedicated memory, the cost for this approach is proportional to $f(M)$.

- Sharing memory among processors reduces the total memory cost, but for large N and M, the circuit cost can still become significant, growing proportionally to $N \times f(M)$.

To address this problem, several strategies have been proposed:

1. **Approach 1:** Dividing shared memory into R blocks of equal size, where each processor can access a block of memory at a time ($\frac{M}{R}$). This reduces the required memory per processor.
2. **Approach 2:** Distributing N memory locations equally among N processors, with each processor handling a specific portion of the memory ($\frac{M}{N}$). This helps balance the workload and reduce memory contention.

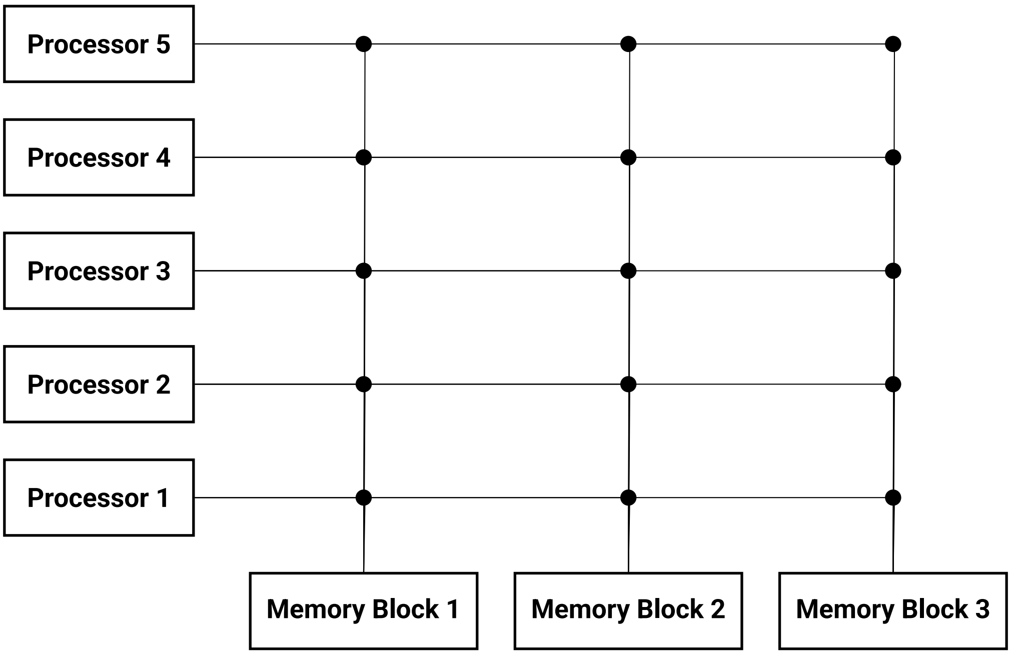
Example: Memory Partitioning and Access (SM SIMD)

In more detail, consider the scenario where memory is shared between N processors by dividing the memory into R blocks of equal size, where each processor can access one block at a time. For example, if $R = 3$ and $N = 5$, the processors share memory with each accessing its respective block.

Key Points:

- In each time step, a processor can only access one block of shared memory ($\frac{M}{R}$).
- The $N + R$ bidirectional links provide access between each processor and its designated block of memory.
- The overall circuit cost grows according to $N \times R \times f(\frac{M}{R})$, with an additional switch needed for managing simultaneous access.

In practice, this model ensures efficient memory access without overwhelming the system with excessive memory demands, but it requires careful management of access timing and switching to avoid conflicts.



> **Interconnection Network (IN) SIMD Computers**

In SIMD computers, interconnection networks play a crucial role in efficiently distributing and processing data across multiple processors. The memory is distributed evenly between N processors, and each processor is connected to others, facilitating data exchange. In fully connected networks, every processor is directly connected to every other processor, ensuring fast communication and reducing latency. The communication cost, however, increases significantly as the number of processors grows, following the relationship $O(N^2)$ for connection setup.

Interconnection networks can be classified into two main types:

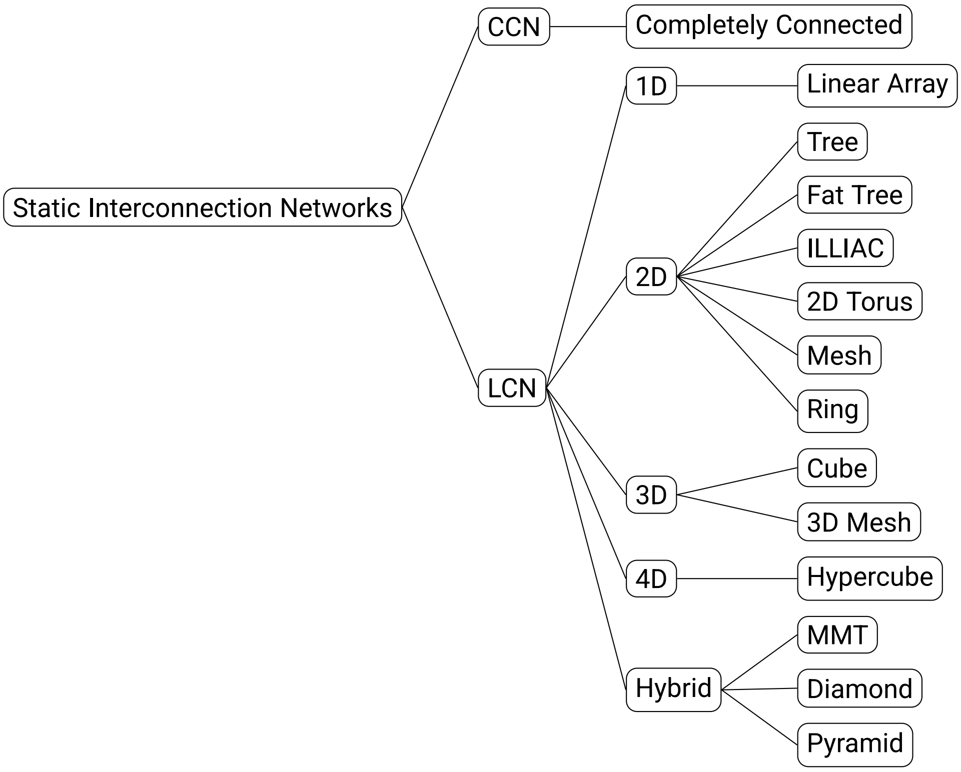
- **Dynamic Networks:** The connections are configured dynamically, adjusting based on the specific task or algorithm.
- **Static Networks:** These networks have fixed, unchanging connections between processors.

Dynamic and static networks can further be differentiated based on their topologies. Topologies are the physical or logical arrangement of the network, defining how processors are interconnected. For SIMD computers, static networks are more common due to their simplicity and ease of design, and they include structures like linear arrays, trees, rings, and hypercubes. Each topology offers a different balance between complexity, cost, and communication speed, impacting the overall performance of the system. Compared to models like MISD, static interconnection networks in SIMD computers are easier to design, and the underlying algorithms are simpler. These systems are most suited for problems where data is evenly distributed among processors and need to be processed concurrently.

When designing interconnection networks, key factors include:

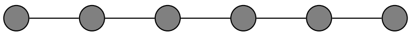
- **Degree:** The maximum number of direct connections a processor can have.
- **Diameter:** The longest distance between any two processors in the network.
- **Bisection Width:** The minimum number of connections that must be removed to split the network into two equal halves.

Different topologies offer varying levels of performance based on these parameters. For instance, a completely connected network provides the best communication speed, but at the cost of increased complexity in wiring and connection management.

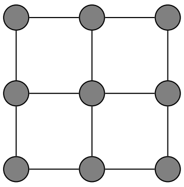


Here's a detailed overview of the various static interconnection network topologies depicted in the diagram. These topologies represent different ways to connect processors and memory blocks in parallel computing systems, ensuring efficient communication between different processing elements (PEs). Each topology has its own unique structure, offering distinct advantages in terms of latency, scalability, and fault tolerance. The choice of an appropriate interconnection network plays a crucial role in optimizing the performance of SIMD or MIMD systems, depending on the specific requirements of the task at hand.

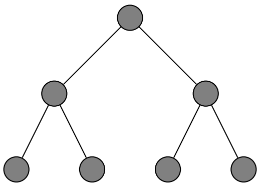
Linear Array
Processors are connected in a straight line, with each processor linked to its immediate neighbor.



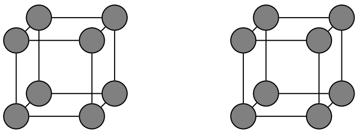
Mesh
Processors are arranged in a 2D grid, with each processor connected to its four adjacent neighbors.



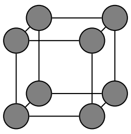
Tree
A hierarchical structure where each node (processor) has a parent-child relationship, used for efficient data propagation.



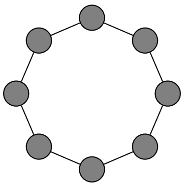
Hypercube
An extension of the cube structure with more dimensions, providing even faster communication and better fault tolerance.



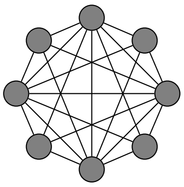
Cube
Processors are connected in the shape of a 3D cube, offering fast communication between nearby nodes.



Ring
Processors form a circular connection where each one is linked to two neighbors, forming a loop.



Completely Connected
Every processor is directly connected to every other processor, allowing the fastest possible communication.



Fat Tree
A tree structure with more bandwidth at higher levels, commonly used for network switches to balance loads.

ILLIAC
A specific grid-like topology designed for parallel processing, offering uniform communication.

Chordal Ring
A ring with additional links between non-adjacent nodes to reduce communication delays.

2D Torus
A mesh where the edges wrap around to form a torus shape, reducing edge communication costs.

3D Mesh
A three-dimensional extension of the mesh topology, useful for large systems with many processors.

2D Tour (variation 2)
A variation of the 2D torus, possibly with different connectivity patterns for specialized tasks.