# 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 | Instruction Streams: many |
|---|---|---|---|
| Data Streams | one | **SISD** traditional von Neumann single CPU computer | **MISD** May be pipelined Computers |
| Data Streams | 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: Time Complexity and Characteristics

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.

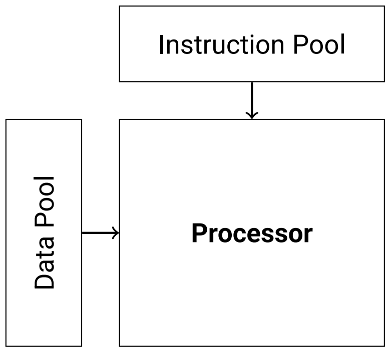

Figure 1.2: A conceptual diagram of SISD architecture.

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.