Parallel Computing - Notes - v0.3.0-dev

260236

October 2024

Preface

Every theory section in these notes has been taken from the sources:

- Course slides. [2]

About:

GitHub repository

These notes are an unofficial resource and shouldn't replace the course material or any other book on parallel computing. It is not made for commercial purposes. I've made the following notes to help me improve my knowledge and maybe it can be helpful for everyone.

As I have highlighted, a student should choose the teacher's material or a book on the topic. These notes can only be a helpful material.

Contents

1 PRAM

1.1 Prerequisites

Before we introduce the PRAM model, we need to cover some useful topics.

- A **Machine Model** describes a "machine". It gives a value to the operations on the machine. It is necessary because: it makes it easy to deal with algorithms; it achieves complexity bounds; it analyses maximum parallelism.
- A **Random Access Machine (RAM)** is a model of computation that describes an abstract machine in the general class of register machines. Some features are:
  - **Unbounded** number of local memory cells;
  - Each memory cell can hold an integer of **unbounded** size;
  - Instruction set includes simple operations, data operations, comparator, branches;
  - All operations take **unit time**;
  - The definition of **time complexity** is the number of instructions executed;
  - The definition of **space complexity** is the number of memory cells used.

1.2 Definition

Definition 1: PRAM

A **parallel random-access machine (parallel RAM or PRAM)**s a **shared-memory abstract machine**. As its name indicates, the PRAM is intended as the parallel-computing analogy to the random-access machine (RAM) (not to be confused with random-access memory). In the same way that the RAM is used by sequential-algorithm designers to model algorithmic performance (such as time complexity), the **PRAM is used by parallel-algorithm designers to model parallel algorithmic performance** (such as time complexity, where the number of processors assumed is typically also stated).

The PRAM model has many interesting features:

- **Unbounded collection of RAM processors** (P_0, P_1, and so on);
- Processors don't have tape;
- Each processor has **unbounded registers**;
- **Unbounded** collection of **share memory cells**;
- All **processors** can **access all memory cells in unit time**;
- All **communication via shared memory**.

1.3 How it works

1.3.1 Computation

A single **processor** of the PRAM, at each computation, is **composed of 5 phases** (carried out in parallel by all the processors):

1. **Reads a value from one of the cells** $X(1), \ldots, X(N)$
2. Reads one of the shared memory cells $A(1), A(2), \ldots$
3. Performs some internal computation
4. **May write into one of the output cells** $Y(1), Y(2), \ldots$
5. May write into one of the shared memory cells $A(1), A(2), \ldots$

1.3.2 PRAM Classificiation

During execution, a subset of processors may remain idle. Also, some processors can read from the same cell at the same time (not really a problem), but they could also try to write to the same cell at the same time (**write conflict**). For these reasons, PRAMs are classified according to their read/write capabilities (realistic and useful):

- **Exclusive Read (ER)**. All processors can simultaneously read from distinct memory locations.
- **Exclusive Write (EW)**. All processors can simultaneously write to distinct memory locations.
- **Concurrent Read (CR)**. All processors can simultaneously read from any memory location.
- **Concurrent Write (CW)**. All processors can write to any memory location.

  But what value is ultimately written?

  It depends on the mode we choose:

  - **Priority Concurrent Write**. Processors have priority based on which value is decided, the **highest priority is allowed to complete write**.
  - **Common Concurrent Write**. All processors are allowed to complete write **if and only if all the value to be written are equal**. Any **algorithm** for this model has to **make sure that this condition is satisfied**. **Otherwise**, the **algorithm is illegal** and the **machine state will be undefined**.
  - **Arbitrary/Random Concurrent Write**. One **randomly** chosen **processor is allowed to complete write**.

1.3.3 Strengths of PRAM

PRAM is attractive and important model for designers of parallel algorithms because:

- It is **natural**. The number of operations executed per one cycle on P processors is at most P (equal to P is the ideal case).
- It is **strong**. Any processor can read/write any shared memory cell in unit time.
- It is **simple**. It abstracts from any communication or synchronization overhead, which makes the complexity and correctness of PRAM algorithm easier.
- It can be used as a **benchmark**. If a problem has no feasible/efficient solution on PRAM, it has no feasible/efficient solution for any parallel machine.

1.3.4 How to compare PRAM models

Consider two generic PRAMs, models A and B. Model A is **computationally stronger** than model B ($A \geq B$) **if and only if any algorithm** written for model B will **run unchanged** on model A in the **same parallel time** and with the **same basic properties**.

However, there are some useful metrics that can be used to compare models:

- **Time to solve problem** of **input size** n on **one processor**, using **best sequential algorithm**:
$$T^*(n) \tag{1}$$
- **Time** to solve problem of input size n on **p processors**:
$$T_p(n) \tag{2}$$
- **Speedup on p processors**:
$$\mathrm{SU}_p(n) = \frac{T^*(n)}{T_p(n)} \tag{3}$$
- **Efficiency**, which is the work done by a processor to solve a problem of input size n divided by the work done by p processors:
$$E_p(n) = \frac{T_1(n)}{pT_p(n)} \tag{4}$$
- **Shortest run time** on any process p:
$$T_\infty(n) \tag{5}$$

- **Cost**, equal to processors and time:

$$C(n) = P(n) \cdot T(n) \tag{6}$$

- **Work**, equal to the total **number of operations**:

$$W(n) \tag{7}$$

Some properties on the metrics:

- The time to solve a problem of input n on a single processor using the best sequential algorithm *is not equal to* the time to solve a problem of input n in parallel using one of the p processors available. In other words, **the problem should not be solvable on a single processor on a parallel machine** (otherwise, what would be the point of using a parallel model?)

$$T^* \neq T_1$$

- $\mathrm{SU}_P \leq P$
- $\mathrm{SU}_P \leq \dfrac{T_1}{T_\infty}$
- $E_p \leq 1$
- $T_1 \geq T^* \geq T_p \geq T_\infty$
- $T^* \approx T_1 \Rightarrow E_p \approx \dfrac{T^*}{pT_p} = \dfrac{\mathrm{SU}_p}{p}$
- $E_p = \dfrac{T_1}{pT_p} \leq \dfrac{T_1}{pT_\infty}$
- $T_1 \in O(C), T_p \in O\left(\dfrac{C}{p}\right)$
- $W \leq C$
- $p \approx \text{AREA} \quad W \approx \text{ENERGY} \quad \dfrac{W}{T_p} \approx \text{POWER}$

1.4 MVM algorithm

The **Matrix-Vector Multiply (MVM) algorithm** consists of four steps:

1. **Concurrent read of vector** $X\,(1:n)$ (transfer N elements);
2. **Simultaneous reads of different sections of the general matrix** A (transfer $\frac{n^2}{p}$ elements to each processor);
3. **Compute** $\frac{n^2}{p}$ operations per processor;
4. **Simultaneous writes** (transfer $\frac{n}{p}$ elements from each processor).

Let i be the processor index, so the MVM algorithm is simply written as:

```
1 GLOBAL READ (Z ← X)
2 GLOBAL READ (B ← A_i)
3 COMPUTE (W := BZ)
4 GLOBAL WRITE (W → y_i)
```

Algorithm 1: Matrix-Vector Multiply (MVM)

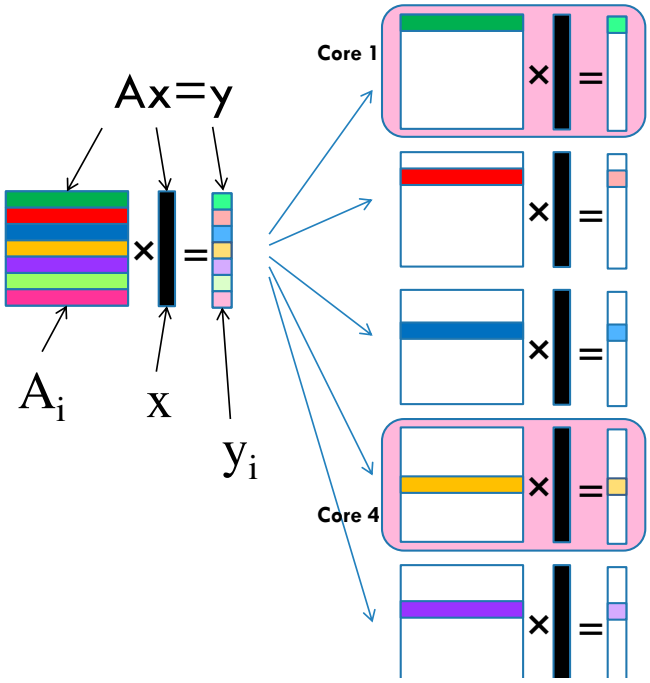


Figure 1: Example of MVM algorithm.

The performance of the MVM algorithm is as follows:

- The **time to solve** a problem of size n^2 is equal to the big O of the squared size of the problem as input divided by the number of processors available:
$$T_p\left(n^2\right) = O\left(\frac{n^2}{p}\right)$$
- The **cost** is equal to the number of processors and the time it takes to solve the problem. So it is quite trivial:
$$C = O\left(\cancel{P} \cdot \frac{n^2}{\cancel{p}}\right) = O\left(n^2\right)$$
- The **work** is equal to the cost, and the **linear power** P is equal to the ratio of work and time to solve the problem on p processors:
$$W = C \qquad \frac{W}{T_p} = P$$
- The **perfect efficiency** is equal to:
$$E_p = \frac{T_1}{pT_p} = \frac{n^2}{p\frac{n^2}{p}} = 1$$

1.5 SPMD sum

The **Single Program Multiple Data (SPMD)** is a term that has been used to **describe computational models** for exploiting parallelism, where **multiple processors work together to execute a program to get results faster**.

In this section, we will see an SPMD approach on a Parallel Random Access Machine (PRAM). We will introduce one of the most common and simple mathematical operations: the sum.

The following pseudocode takes as **input an array** of size $n = 2^k$. In this case, n is a power of 2 because it ensures that the array can be evenly divided at each step of the computation. The value k is the number of iterations or levels of the summation process.

```
1  BEGIN
2      GLOBAL READ (A ← A(I))
3      GLOBAL WRITE (A → B(I))
4      FOR H = 1 : K
5          IF i ≤ n ÷ 2^h THEN BEGIN
6              GLOBAL READ (X ← B(2i - 1))
7              GLOBAL READ (Y ← B(2i))
8              Z := X + Y
9              GLOBAL WRITE (Z → B(i))
10         END
11     IF I = 1 THEN
12         GLOBAL WRITE (Z → S)
13 END
```

Algorithm 2: Single Program Multiple Data (SPMD) sum

- First, read the entire input array `A` and copy the read data to another array `B`.
- Loop over `h` (`1 to k`). In each iteration, for each index i less than or equal to $n \div 2^h$, read values from array `B` at positions $2i - 1$ and $2i$; sum these values (and store the result in `Z`) and store the result (`Z`) back into `B(`i`)`.
- Once all iterations are complete, the final sum is stored in a variable `S`.

For example, if $n = 8$, then k would be 3, meaning that the algorithm will run for 3 iterations to sum all the elements in parallel.

h	i	adding
1	1	1,2
	2	3,4
	3	5,6
	4	7,8
2	1	1,2
	2	3,4
3	1	1,2

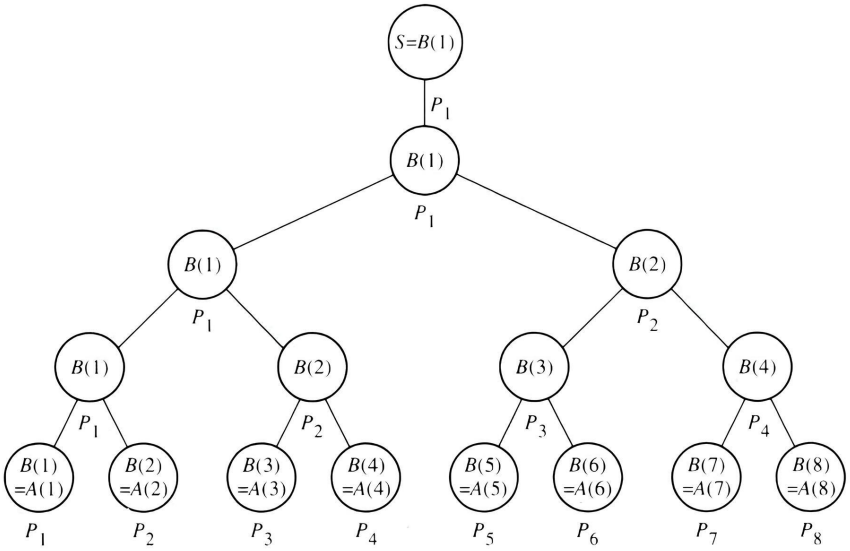
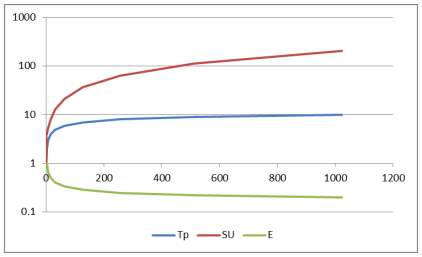


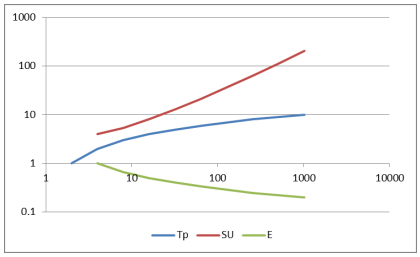
Figure 2: Computation of the sum of eight elements on a PRAM with eight processors. Each internal node represents a sum operation. The specific processor executing the operation is indicated below each node.

Performance of sum

When the size of the array is equal to the number of processors ($N = P$), the **speedup and efficiency decrease**:

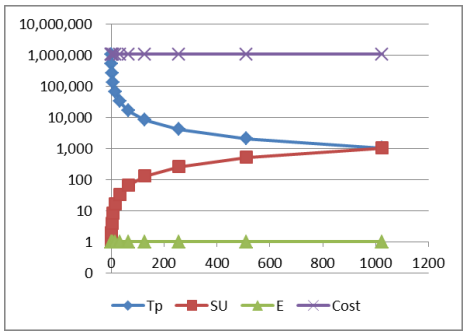
- $T^*(N) = T_1(N) = N$
- $T_{N=P}(N) = 2 + \log N$
- $\mathrm{SU}_P = \dfrac{N}{2 + \log N}$
- $T^*(N) = P \cdot (2 + \log N) \approx N \log N$
- $E_p = \dfrac{T_1}{pT_p} = \dfrac{N}{N \log N} = \dfrac{1}{\log N}$





If the size of the array is much larger than the number of processors ($N \gg P$), the **speedup and power are linear**, the **cost is fixed** and the **efficiency is maximum (equal to 1)**:

- $T^*(N) = T_1(N) = N$
- $T_p(N) = \frac{N}{p} + \log p$
- $\mathrm{SU}_P = \frac{N}{\frac{N}{p} + \log p} \approx P$
- $\mathrm{COST} = p\left(\frac{N}{p} + \log p\right) \approx N$
- $\mathrm{WORK} = N + P \approx N$
- $E_p = \frac{T_1}{pT_p} = \frac{N}{p\left(\frac{N}{p} + \log p\right)} \approx 1$



$n = 1'000'000$

Example 1

Refer to Figure 2 (page 11), the performance metrics are:

- $T_8 = 5$
- $C = 8 \cdot 5 = 40$ (could do 40 steps)
- $W = 2n = 16$ (16 on 40, wasted 24)
- $E_p = \frac{2}{\log n} = \frac{2}{3} = 0.67$
- $\frac{W}{C} = \frac{16}{40} = 0.4$

There is also the **Prefix Sum**, which takes **advantage of idle processors in the sum**. It computes all prefix sums:

$$S_i = \sum_{1}^{i} a_j \qquad a_1, \quad a_1 + a_2, \quad a_1 + a_2 + a_3$$

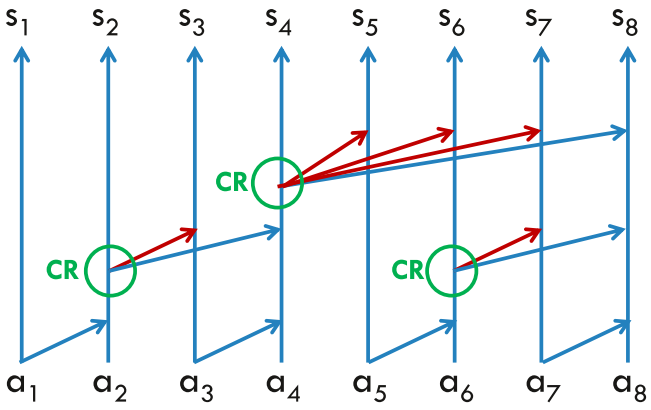


Figure 4: Prefix sum.

1.6 MM algorithm

The **Matrix Multiply (MM) algorithm** consists of three steps:

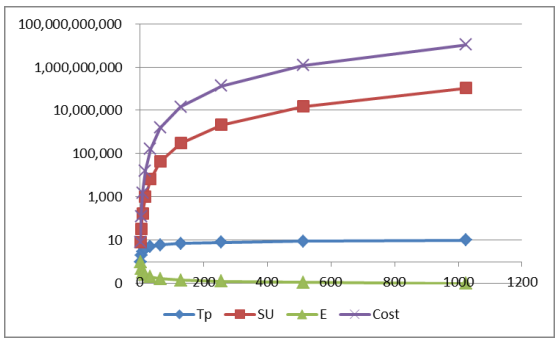
1. **Compute the two matrices** $A_{i,l}$ and $B_{l,j}$, so use the concurrent read.
2. Make the **sum**.
3. **Store** the result using exclusive write.

```
1 BEGIN
2     T_{i,j,l} = A_{i,l} B_{l,j}
3     FOR = H = 1 : K
4         IF l ≤ n ÷ 2^h THEN
5             T_{i,j,l} = T_{i,j,2l-1} + T_{i,j,2l}
6     IF l = 1 THEN
7         C_{i,j} = T_{i,j,1}
8 END
```

Algorithm 3: Matrix Multiply (MM)

Performance of MM

- $T_1 = n^3$
- $T_{p=n^3} = \log n$
- $\text{SU} = \dfrac{n^3}{\log n}$
- $\text{Cost} = n^3 \log n$
- $E_p = \dfrac{T_1}{pT_p} = \dfrac{1}{\log n}$



1.7 PRAM variants and Lemmas

The PRAM model presented here is one of the most commonly used. However, there are other important variants:

- PRAM model with a **limited number of shared memory cells** (small memory PRAM). If the input data set exceeds the capacity of the shared memory, the I/O values can be evenly distributed among the processors.
- PRAM model with **limited number of processors** (small PRAM). If the number of execution threads is higher, processors can interleave multiple threads.
- PRAM model with **limited size of one machine word**.
- PRAM model with **access conflicts handling**. These are restrictions on simultaneous access to shared memory cells.

Lemma 1. *Assume $P' < P$ and same size of shared memory. Any problem that can be solved for a P processor PRAM in T steps can be solved in a P' processor PRAM in:*

$$T' = O\left(\frac{TP}{P'}\right) \tag{8}$$

Proof. Partition P is simulated processors into P' groups of size $\frac{P}{P'}$ each. Associate each of the P' simulating processors with one of these groups. Each of the simulating processors simulates one step of its group of processors by:

- Executing all their read and local computation substeps first;
- Executing their write substeps then.

QED

Lemma 2. *Assume $M' < M$. Any problem that can be solved for a P processor and M-cell PRAM in T steps can be solved on a $\max(P, M')$-processors M'-cell PRAM in $O\left(\frac{TM}{M'}\right)$ steps.*

Proof. Partition M simulated shared memory cells into M' continuous segments S, of size $\frac{M}{M'}$ each. Each simulating processor P'_I $(1 \leq I \leq P)$, will simulate processor P_I of the original PRAM. Each simulating processor P'_I $(1 \leq I \leq M')$, stores the initial contents of S_I into its local memory and will use $M'[I]$ as an auxiliary memory cell for simulation of accesses to cell of S_I.
Simulation of one original read operation:

```
1 EACH P'_I (I = 1,...,max(P,M')) REPEATS FOR K = 1, ..., M/M'
2     WRITE THE VALUE OF THE K-TH CELL OF S_I INTO M'[I] (I = 1,...,M')
3     READ THE VALUE WHICH THE SIMULATED PROCESSOR P_I (I = 1,...,P)
      WOULD READ IN THIS SIMULATED SUBSTEP, IF IT APPEARED IN THE
      SHARED MEMORY
```

The local computation substep of P_I $(I = 1, \ldots, P)$ is simulated in one step by P'_I. SImulation of one original write operation is analogous to that of read.
QED

1.8 PRAM implementation

The PRAM is an ideal model for creating parallel algorithms. Now we look at "*is it really implementable?*" The short answer is yes.

The longest answer is the following. There are already some examples of PRAM being converted to real machine models, such as Explicit Multi-Threading (XMT), Rigel, Tilera, etc. If conversion is not easy or possible, the implementation can be "*direct*":

- The concurrent read is implemented as a detect-and-multicast technique.
- The concurrent write is implemented depending on the end result we want to achieve. Fetch-and-operate and prefix-sum are examples of serialized writing; otherwise, the CRCW technique is used:
  - Common CRCW: detect and merge
  - Priority CRCW: detect-and-priorities
  - Arbitrary CRCW: arbitrary

Example 2: Boolean DNF (sum of products) common CRCW

A logical formula is considered to be in DNF if it is a disjunction of one or more conjunctions of one or more literals.
Consider X as the sum of products of AND/OR operations:

$$X = a_1 b_1 + a_2 b_2 + \dots$$

The PRAM code, with X initialized to 0 and task index equal to \$, is:

```
if (a$b$) X = 1;
```

The common result is that not all processors write X and those that do write 1. The time complexity is $O(1)$. It works on common, priority and arbitrary CRCW.

Despite the previous example, exists also the PRAM SoP for the concurrent write. Let boolean X as:

$$X = a_1 b_1 + a_2 b_2 + \dots$$

The PRAM algorithm is:

```
if (aibi) X = 1;
```

Where all cores which write into X, **write the same value**.

✔ PRAM advantages

- Large body of algorithms.
- Easy to think about it.
- Sync version of shared memory. It eliminates sync and common issues, allows focus on algorithms, but allows adding these issues and allows conversion to async versions.
- Exists architectures for both synch (PRAM) and async (SM) model.
- PRAM algorithms can be mapped to other models.

1.9 Amdahl's and Gustafson's Laws

The **Amdahl's Law** is a formula which gives the **theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved**. The law can be stated as:

Definition 2: Amdahl's Law

The overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used.

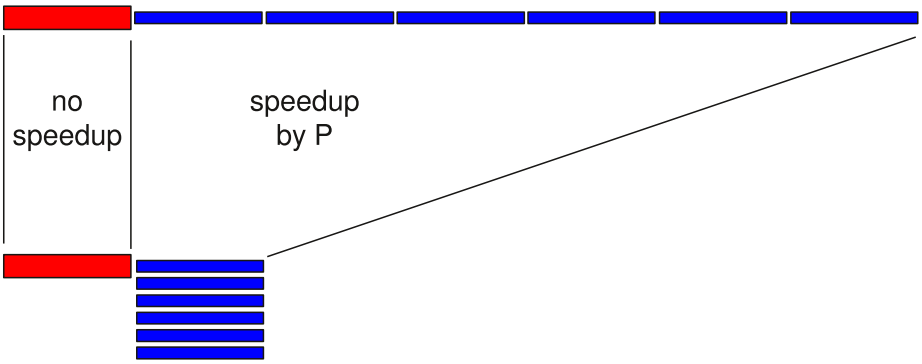
In practice, Amdahl's law says that the computation consists of interleaved segments of two types:

1. **Serial segments** (which cannot be parallelized);
2. **Parallelizable segments**.

Therefore, the metrics we can obtain are the time on P processors metric, that it is greater than the fraction of time on a processor divided by the processors P, and the speedup metric, that it is less than the number of processors P:

$$T_P > \frac{T_1}{P} \qquad SU < P$$

Graphically, we can see a fixed part of the line, which is the **serial segment** (no speedup), and a set of instructions that can be **parallelized** (the sum of these segments is equal to the unit time 1).



Furthermore, if we identify the parallelizable segment as f and the serial segment as $1 - f$, we obtain the following expressions:

$$\begin{aligned} SU\,(P, f) &= \frac{T_1}{T_P} = \frac{T_1}{T_1 \cdot (1-f) + \frac{T_1 \cdot f}{P}} = \frac{1}{(1-f) + \frac{f}{P}} \\ \lim_{P \to \infty} SU\,(P, f) &= \frac{1}{1-f} \end{aligned} \tag{9}$$

In the following figure we can see the speedup with parameter f. Note the pessimism: for a problem with inherent $f = 90\%$, there is no point in using more than 10 processors.

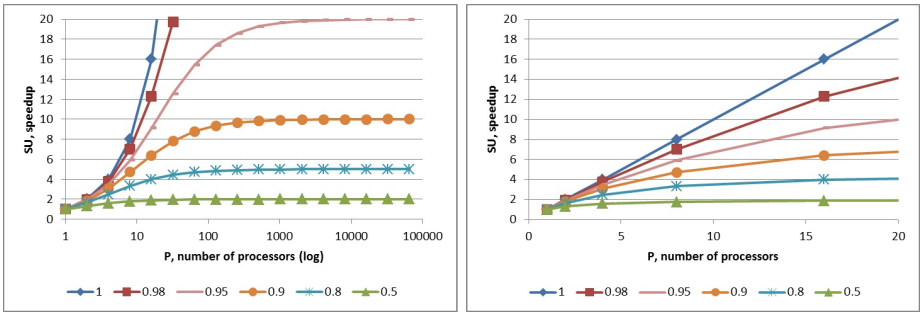


Figure 5: Amdahl's law, $SU(P)$, parameter f.

The original paper presenting Amdahl's Law [1] can be viewed by clicking on the link below or by scanning the QR code.

Amdahl's Law

Amdahl's law applies only to the cases where the problem size is fixed. In practice, as more computing resources become available, they tend to get used on larger problems (larger datasets), and the time spent in the parallelizable part often grows much faster than the inherently serial work. In this case, **Gustafson's law gives a less pessimistic and more realistic assessment of the parallel performance**. [4]

Gustafson's Law gives the speedup in the execution time of a task that theoretically gains from parallel computing, using a hypothetical run of the task on a single-core machine as the baseline. To put it another way, it is the **theoretical "slowdown" of an already parallelized task if running on a serial machine**.

Against Amdahl's law, Gustafson suggests the following ideas:

- Portion f is not fixed;
- The absolute serial time is fixed;
- Parallel problem size is increased to exploit more processors;
- Fixed serial time (s of total) and fixed parallel time ($1-s$ of total) are invariants;
- **Fixed time model** and not fixed size model (as Amdahl's law):

$$SU(P) = \frac{T_1}{T_P} = \frac{s + P \cdot (1-s)}{s + (1-s)} = s + P \cdot (1-s) \tag{10}$$

Gustafson's law suggests a **linear speedup** and is **empirically applicable to highly parallel algorithms**.

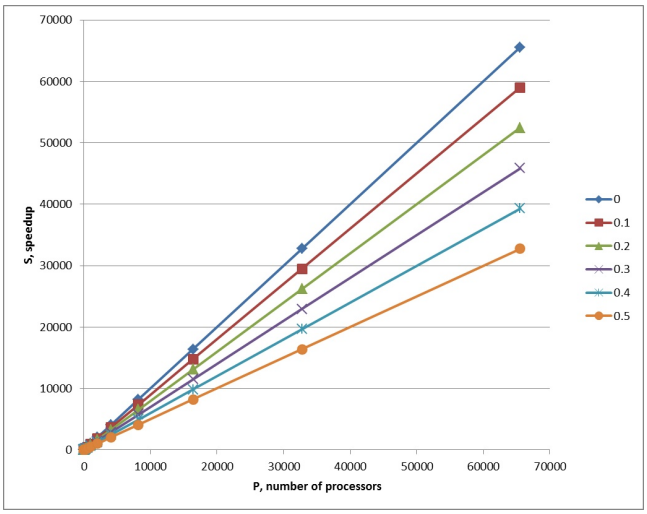


Figure 6: Gustafson's law.

Amdahl's Law states that as computing power increases, computational requirements remain the same. In other words, the **analysis of the same data will take less time with more computing power**.

Gustafson, on the other hand, argues that **more computing power leads to more careful and complete analysis of the data**. Where it would not have been possible or practical to simulate the impact of nuclear denotation on every building, car, and their contents (including furniture, structural strength, etc.) because such a calculation would have taken more time than was available to provide an answer, the increase in computing power will prompt researchers to add more data to more fully simulate more variables, giving a more accurate result.

The original paper presenting Gustafson's Law [3] can be viewed by clicking on the link below or by scanning the QR code.

Gustafson's Law

2 Fundamentals of architecture

2.1 Introduction

2.1.1 Simplest processor

Inside a computer, a processor executes instructions.

- **Fetch/Decode**: Determine which instruction to run next;
- **ALU** (execution unit): Performs the operation described by an instruction, which may change values in the processor's registers or the computer's memory;
- **Registers**: maintain program state, store values of variables used as inputs and outputs to operations.

The simplest and most basic processor executes **one instruction per clock cycle**.

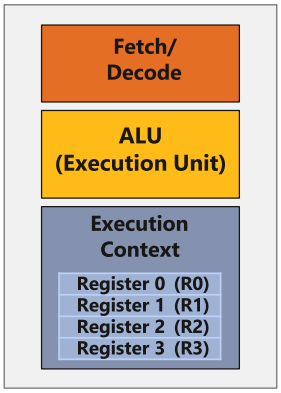


Figure 7: The simplest and most basic processor.

2.1.2 Superscalar processor

A more "complex" and realistic model is the **superscalar processor** This **processor can decode and execute up to two instructions per clock**. The execution is slightly different from the simplest processor. The **processor automatically finds independent instructions in an instruction sequence and can execute them in parallel on multiple execution units**.

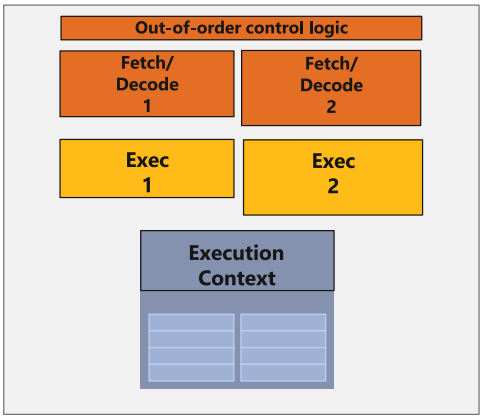


Figure 8: The superscalar processor.

The superscalar processor takes advantage of **Instruction-Level Parallelism (ILP)**[1] within an instruction stream.

- Processing **different instructions** from the same instruction stream **in parallel** (**within a core**).
- **Parallelism is automatically detected by the hardware during execution**.

[1] Instruction-level parallelism (ILP) is the parallel or simultaneous execution of a sequence of instructions in a computer program. More specifically ILP refers to the average number of instructions run per step of this parallel execution.

2.1.3 Single Instruction, Multiple Data (SIMD) processor

Adding execution units (ALUs) to the simplest processor can increase compute capability. Amortize the cost/complexity of managing an instruction stream across many ALUs using **Single Instruction, Multiple Data (SIMD)** processing. Therefore, the **same instruction is sent to all ALUs**. This **operation is performed in parallel on all ALUs**.

✔ **Advantages**

- **Efficient for data-parallel workloads**: amortize control costs over many ALUs.
- Vectorization done by:
  - Compiler (**explicit SIMD**): parallelism is explicitly requested by the programmer through intrinsics, conveyed through parallel language semantics, and inferred through loop dependency analysis by the "auto-vectorizing" compiler. In other words, the **SIMD parallelization is done at compile time, and when we inspect the program binary, we can see the SIMD instructions**.
  - At runtime by hardware (**implicit SIMD**): the **compiler generates a binary with scalar instructions**, but n instances of the program are always executed together on the processor. The **hardware** (not the compiler) is **responsible for the simultaneous execution of the same instruction by multiple program instances on different data on SIMD ALUs**.

2.1.4 Multi-Core Processor

A **Multi-Core Processor (MCP)** is a **microprocessor** on a single integrated circuit (IC) with **two or more separate central processing units (CPUs)**, called *cores* to emphasize their multiplicity (e.g., *dual-core* or *quad-core*). Each core reads and executes program instructions, specifically ordinary CPU instructions (such as `add`, `move data`, and `branch`). However, the MCP can **execute instructions on separate cores simultaneously**, **increasing overall speed for programs that support multithreading or other parallel computing techniques**.

✔ **Advantages**

- Provides **thread-level parallelism**: execute a completely different instruction stream on each core simultaneously.
- **Software creates threads to expose parallelism to hardware** (e.g., via threading API)

References

[1] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities, reprinted from the afips conference proceedings, vol. 30 (atlantic city, nj, apr. 18–20), afips press, reston, va., 1967, pp. 483–485, when dr. amdahl was at international business machines corporation, sunnyvale, california. *IEEE Solid-State Circuits Society Newsletter*, 12(3):19–20, 2007.

[2] Ferrandi Fabrizio. Parallel computing. Slides from the HPC-E master's degree course on Politecnico di Milano, 2024.

[3] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[4] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. ITPro collection. Elsevier Science, 2012.

Index