Parallel Computing - Notes - v0.5.0-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)** is a **shared-memory abstract machine**. As its name indicates, the PRAM is intended as the parallel-computing analogy to the random-access machine (RAM) (<u>not</u> 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 $X(1),\dots,X(N)$**

2. Reads one of the shared memory cells $A(1),A(2),\dots$

3. Performs some internal computation

4. **May write into one of the output cells $Y(1),Y(2),\dots$**

5. May write into one of the shared memory cells $A(1),A(2),\dots$

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.

    ❷ 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 <u>one</u> **processor**, using **best** <u>sequential</u> **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:
$$E_p (n) = \frac{T_1 (n)}{p T_p (n)} \tag{4}$$

- **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?)

$$T^* \neq T_1$$

- $\mathrm{SU}_P \leq P$

- $\mathrm{SU}_P \leq \dfrac{T_1}{T_\infty}$

- $E_p \leq 1$

- $T_1 \geq T^* \geq T_p \geq T_\infty$

- $T^* \approx T_1 \Rightarrow E_p \approx \dfrac{T^*}{pT_p} = \dfrac{\mathrm{SU}_p}{p}$

- $E_p = \dfrac{T_1}{pT_p} \leq \dfrac{T_1}{pT_\infty}$

- $T_1 \in O(C), T_p \in O\left(\dfrac{C}{p}\right)$

- $W \leq C$

- $p \approx \mathrm{AREA} \quad W \approx \mathrm{ENERGY} \quad \dfrac{W}{T_p} \approx \mathrm{POWER}$

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 $\dfrac{n^2}{p}$ elements to each processor);

3. **Compute** $\dfrac{n^2}{p}$ operations per processor;

4. **Simultaneous writes** (transfer $\dfrac{n}{p}$ 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)
```

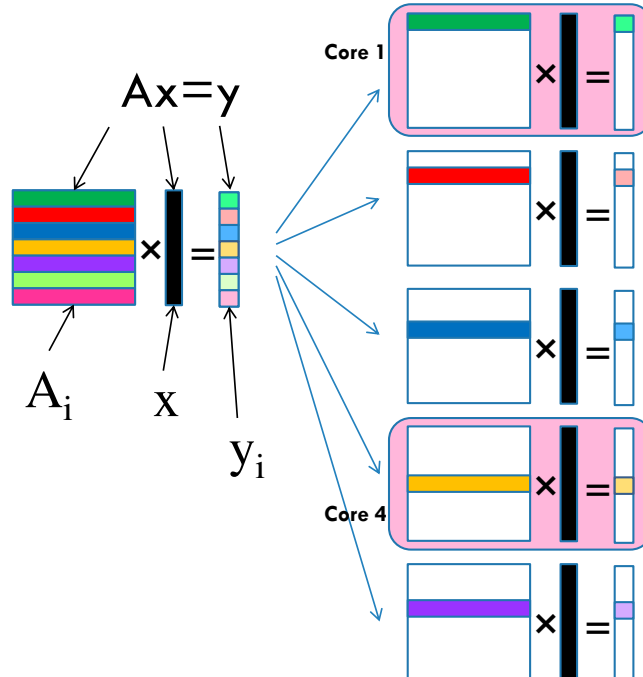Algorithm 1: Matrix-Vector Multiply (MVM)

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:
$$C = O\left(\not{p} \cdot \frac{n^2}{\not{p}}\right) = O\left(n^2\right)$$

- 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)**:

- $T^*\left(N\right) = T_1\left(N\right) = N$

- $T_p\left(N\right) = \dfrac{N}{p} + \log p$

- $\mathrm{SU}_P = \dfrac{N}{\frac{N}{p} + \log p} \approx P$

- $\mathrm{COST} = p\left(\dfrac{N}{p} + \log p\right) \approx N$

- $\mathrm{WORK} = N + P \approx N$

- $E_p = \dfrac{T_1}{pT_p} = \dfrac{N}{p\left(\frac{N}{p} + \log p\right)} \approx 1$
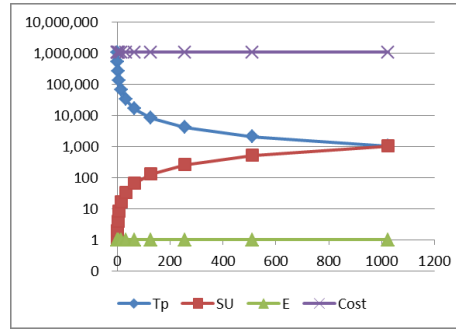
$$n = 1'000'000$$

<div style="border:1px solid green">

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)

- $E_p = \dfrac{2}{\log n} = \dfrac{2}{3} = 0.67$

- $\dfrac{W}{C} = \dfrac{16}{40} = 0.4$

</div>

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

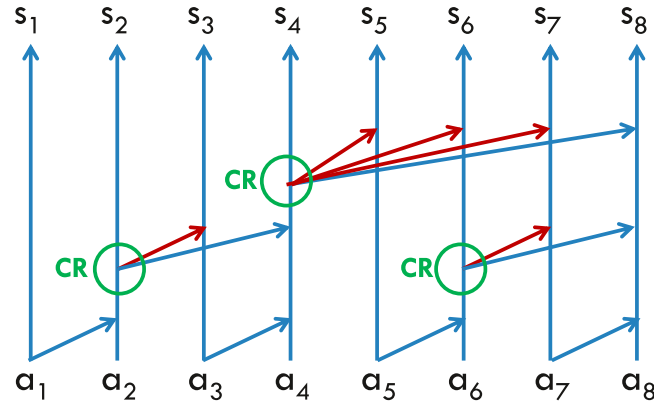$$S_i = \sum_1^i a_j \qquad a_1, \quad a_1 + a_2, \quad a_1 + a_2 + a_3$$

Figure 4: Prefix sum.

13

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**.
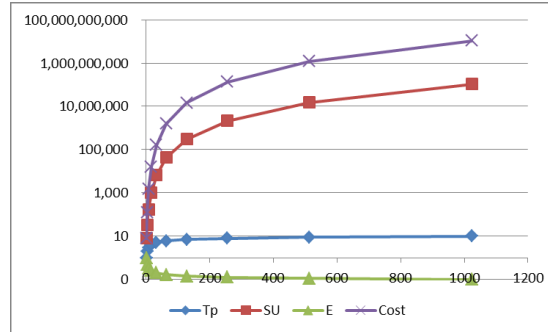
3. **Store** the result using exclusive write.

```
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

- $T_1 = n^3$

- $T_{p=n^3} = \log n$

- $\text{SU} = \dfrac{n^3}{\log n}$

- $\text{Cost} = n^3 \log n$

- $E_p = \dfrac{T_1}{pT_p} = \dfrac{1}{\log n}$

14

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.

<div align="right">QED</div>

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 ($1 \le I \le P$), will simulate processor P_I of the original PRAM. Each simulating processor P'_I ($1 \le I \le M'$), 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.
<div align="right">QED</div>

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:

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

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:

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

The PRAM algorithm is:

```
if (a_i b_i) 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).
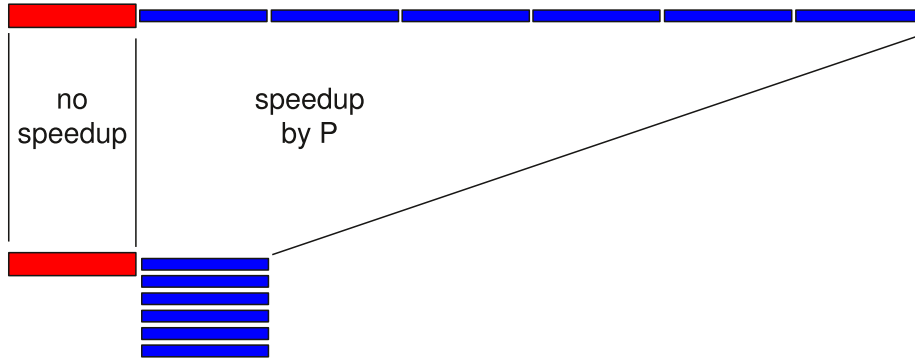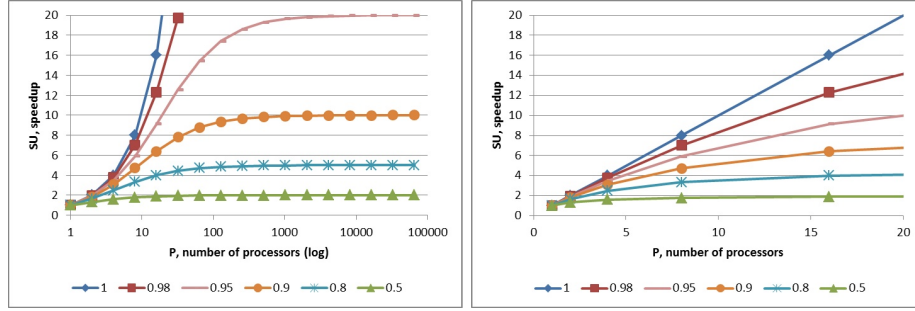

Furthermore, if we identify the parallelizable segment as f and the serial segment as $1 - f$, we obtain the following expressions:

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

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, $SU\left(P\right)$, 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**. [6]

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):

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

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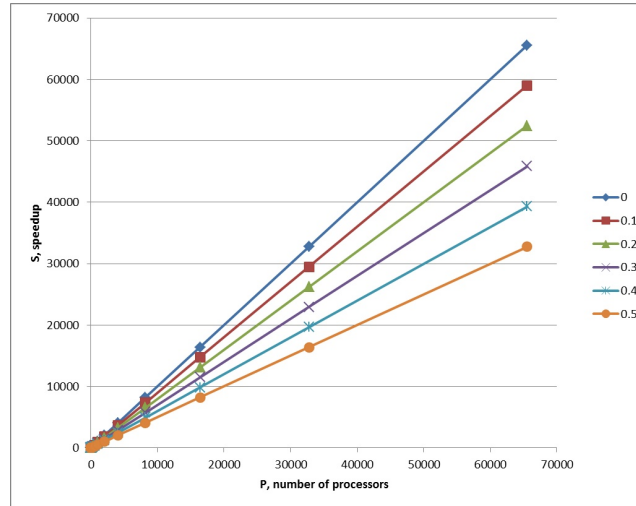

Figure 6: Gustafson's law.

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

The main purpose of this chapter is to introduce some basics of parallel computing theory. It will introduce the simplest and trivial processor and the more complex and efficient variants. The topics introduced are explained in a simple way and without any deepening, because it is only an introduction. For those who have studied computer science, this chapter might be a little boring and you might notice that some topics are explained in a simplistic way.

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**.
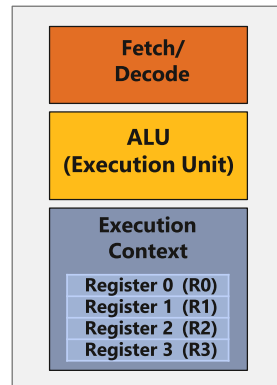

Figure 7: The simplest and most basic processor.

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**.
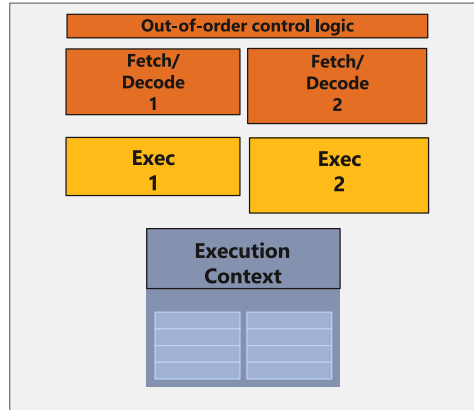

Figure 8: The superscalar processor.

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)

2.2 Accessing Memory

2.2.1 What is a memory?

A computer's memory is organized as an array of bytes. Each byte is identified by its address in memory (its position in that array).

Address	Value
0x0	16
0x1	255
0x2	14
0x3	0
0x4	0
0x5	0
0x6	6
0x7	0
0x8	32
0x9	48
0xA	255
0xB	255
0xC	255
0xD	0
0xE	0
0xF	0
0x10	128
⋮	⋮
0x1F	0

Table 1: Example illustration of the program's memory address space of 32 bytes, range from `0x0` to `0x1F`.

From the processor's point of view, loading an instruction to access the contents present in memory is done with the `ld` assembly instruction. For example, `ld R0 ← mem[R2]` means "take the value from register R2 and put that value into register R0".

Before we introduce new concepts, let us take a moment to explain some important **terminology**:

- **Memory Access Latency**, is the **time** it takes for the **memory system to deliver data to the processor**.

- **Processor Stall**. A **processor** stalls when it **cannot execute the next instruction in an instruction stream because of a dependency on a previous instruction that has not been completed**. Accessing memory is a major source of stalling, which is one of the main reasons why memory accesses should be limited.

For **example**, in the following three assembler instructions, the add has to wait for the loading of `R2` and `R3` values, making parallelization more complicated:

```
ld r0 mem[r2]
ld r1 mem[r3]
add r0, r0, r1
```

- **Memory Bandwidth**, is the **rate at which the memory system can provide data to a processor**.

  Bandwidth is the <u>critical</u> resource in modern computing.

  High-performance parallel programs will:

  1. **Organize computation to fetch data from memory less frequently**. For example, reuse data previously loaded by the same thread (temporal locality optimizations) or share data across threads (inter-thread cooperation);

  2. Prefer to **perform additional arithmetic to store/reload values**;

  3. **Programs need to access memory infrequently** to take advantage of modern processors.

2.2.2 How to reduce processor stalls

2.2.2.1 Cache

One of the most common solutions is caching.

A **cache** is a hardware or software **component that stores data so that future requests for that data can be served faster**; the data stored in a cache might be the result of an earlier computation or a copy of data stored elsewhere.

- A **Cache Hit** occurs when the **requested data is found** in a cache;

- A **Cache Miss** occurs when **it cannot**.

Cache hits are served by reading data from the cache, which is faster than recomputing a result or reading from a slower data store, so the **more requests that can be served from the cache, the faster the system performs**. [8]

Many modern CPUs have logic that predicts what data will be accessed in the future and "pre-fetches" that data into caches. **Prefetching** reduces stalls because the data is resident in the cache when it is accessed. But beware, the other side of the coin is that if the **guess is wrong**, the **performance is worse than the system without prefetching**!
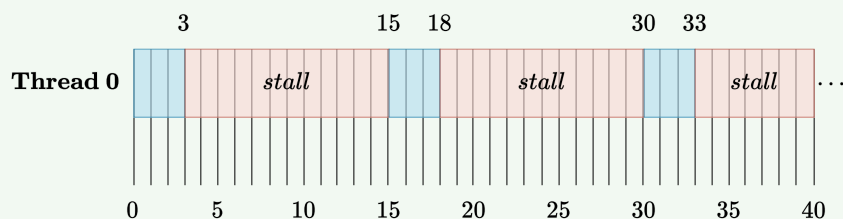

2.2.2.2 Multi-threading

A **Multithreaded Processor** is one that has the **ability to follow multiple instruction streams without software intervention**. In practice, then, this **includes any machine that stores multiple program counters (PCs) in hardware within the processor** (i.e., on chip, for microprocessor-era machines). [7]

The main idea in this architecture is to **interleave processing of multiple threads on the same core to hide stalls**. In other words, if we can't make progress on the current thread, we work on another one.

> **Example 1: core utilization**
>
> To better understand our explanation, suppose we are running a program where threads perform **three arithmetic instructions followed by a memory load** (with 12 cycle latency).
>
>
>
> From the figure, it is clear that if we consider an arithmetic instruction and a memory stall, the core is not fully optimized to work at 100%. In

practice, we see that a single arithmetic instruction takes 3 clock cycles and the memory stall takes 12 clock cycles. This means that from this situation we are using the CPU at only 20% (3 work clock cycles on 15)! Without suggesting the final solution, try to see what happens when we add another thread.
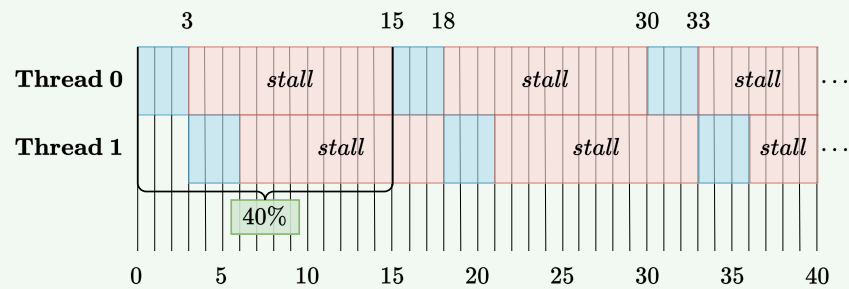
We have gained three clock cycles, and now we are taking advantage of the 40% of the core.

Now, how many threads do we need to achieve 100% utilization? The answer is simple: the number of clock cycles of the operations to be done before the stall plus the clock cycles of the stall divided by the working operations (operations that are not stalls). In our case: $15 \div 3 = 5$.
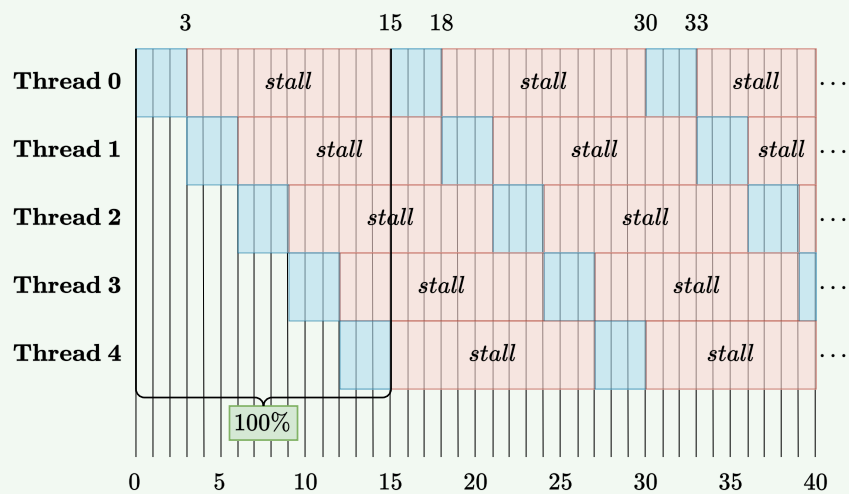
Note that if we add more threads, there will be no benefit because the CPU is already at 100%.

✔ **Multithreaded Processor benefits**

- A processor with multiple hardware threads has the **ability to avoid stalls** by executing instructions from other threads when one thread must wait for a long latency operation to complete. The latency of the memory operation is not changed by multithreading, it just no longer causes reduced processor utilization.

- A **multithreaded processor hides memory latency by performing**

arithmetic from other threads. Program that feature more arithmetic per memory access need fewer threads to hide memory stalls.

▮ Type of hardware-supported multithreading

- **Core manages execution contexts for multiple threads**. This type still has the same number of ALU resources: multi-threading only helps to use them more efficiently in the face of high latency operations such as memory access. The **processor decides which thread to run each clock cycle**.

- **Coarse-Grain Multithreading**, also called **Block Multithreading** or **Switch-On-Event Multithreading**, has multiple hardware contexts associated with each processor core. A hardware context is the program counter, register file, and other data required to enabled a software thread to execute on a core. However, only one hardware context has access to the pipeline at a time. [7]

- **Fine-Grain Multithreading (FGMT)**, also known as **Interleaved Multithreading** or **Temporal Multithreading**, is the type just described on the previous pages, as in the example on page 26.

- **Simultaneous Multithreading (SMT)** has multiple hardware contexts associated with each core. In a simultaneous multithreaded processor, instructions from multiple threads are available to be issued on any cycle. Therefore, all hardware contexts, and in particular all register files, must be accessible to the pipeline and its execution resources. [7]

  In other words, **each clock**, the **core selects instructions from multiple threads to execute on ALUs**.

3 Programming models

3.1 Implicit SPMD Program Compiler (ISPC)

Before introducing the ISPC compiler, we give the definition of SPMD.

Definition 1: Single Program, Multiple Data (SPMD)

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

The **difference** between *SPMD* and *SIMD* (page 23) is that in SPMD parallel execution, **multiple autonomous processors simultaneously execute the same program at independent points**, rather than in SIMD it is vectorization at the instruction level so that **each CPU instruction processes multiple data elements**.

In other words:

- SPMD: is the **programming abstraction**, because the programmer has to think; the program is written in terms of this abstraction.

- SIMD: in general, the compilers (ISPC) issue special vector instructions that execute the logic performed by each parallel instance created (ISPC gang spawned). In addition, the compilers handle the mapping of conditional control flow to vector instructions.

The difference and the terminology used by ISPC will become clearer in the following pages. We suggest that finish this section and come back here in a moment.

Definition 2: Implicit SPMD Program Compiler (ISPC)

Implicit SPMD Program Compiler (ISPC) is a **compiler for a variant of the C programming language**, with extensions for *Single Program, Multiple Data (SPMD)* programming. Under the SPMD model, the programmer writes a program that generally appears to be a regular serial program, though the execution model is actually that a number of program instances execute in parallel on the hardware. In other words, the **ISPC gives the programmer some API to do parallelization on the code; it also generates high quality SIMD code to increase performance**.

The definition, implementation, and other details are explained in the official Intel GitHub repository.

❷ How it works?

Let us take a general main program; when we call an `ispc` function, it causes a **spawn of gang of ISPC program instances upon return, all instances have completed**. These **instances execute the same ISPC code simultaneously**, and **each instance has its own copy of local variables**. Take the following ISPC code as an example:

```
export void ispc_sinx(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result
){
    // assume N % programCount = 0
    for (uniform int i=0; i<N; i+=programCount) {
        int idx = i + programIndex;
        float value = x[idx];
        float numer = x[idx] * x[idx] * x[idx];
        uniform int denom = 6; // 3!
        uniform int sign = -1;
        for (uniform int j=1; j<=terms; j++) {
            value += sign * numer / denom
            numer *= x[idx] * x[idx];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[idx] = value;
    }
}
```

In the example, the `programCount` (row 8) and `programIndex` (row 9) variables, `uniform` (row 2, and so on) data type tell us:

- `programIndex` gives the index of the SIMD-lane being used for running each program instance (in other words, it's a varying **integer value that has value zero for the first program instance, and so forth**).

- `programCount` gives the **total number of instances in the *gang***.

- A variable that is declared with the `uniform` qualifier represents a **single value that is shared across the entire *gang***.

Together, these can be used to uniquely map executing program instances to input data (programIndex and programCount, uniform data type).

With the ISPC analogy, the **SPMD programming model** should be clear:

1. **Single thread of control** (typically a main program);

2. **Invoke** the **SPMD function** (in the previous example, the `ispc_sinx` function);

3. **SPMD execution**, then **multiple instances of the function run in parallel** (multiple logical threads of control);

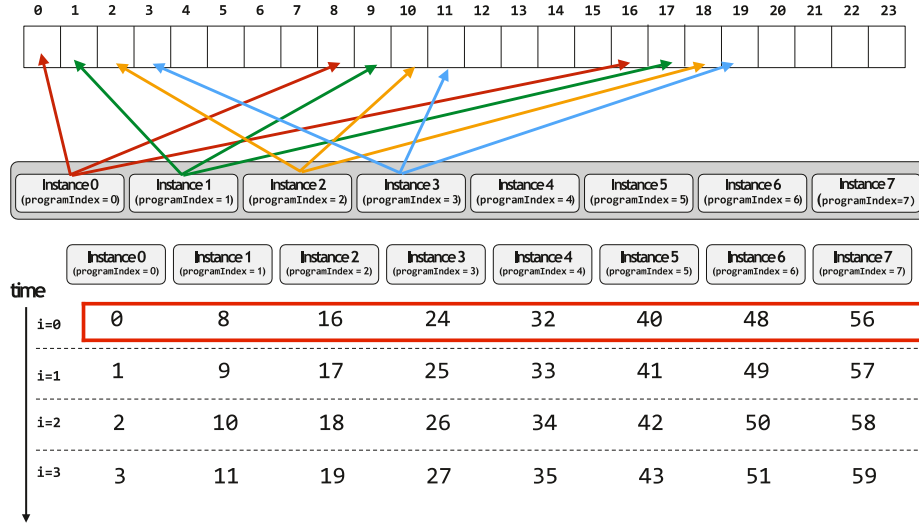4. **Returns** and **resumes a single thread** of control.

Figure 9: Example of execution with 8 instances (`programCount` equal to 8). For all program instances, there are eight non-contiguous values in memory. A special instruction called `gather` is needed to implement this, but unfortunately it is a more complex and expensive SIMD instruction rather than a contiguous implementation.

Figure 9 shows a possible execution of the ISPC function using 8 instances. The result is obtained and all is well. But there is one interesting observation. **Each ISPC instance writes each value in a non-contiguous way**. This can be done better:

```
export void ispc_sinx_v2(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result
){
    // assume N % programCount = 0
    uniform int count = N / programCount;
    int start = programIndex * count;
    for (uniform int i=0; i<count; i++) {
        int idx = start + i;
        float value = x[idx];
        float numer = x[idx] * x[idx] * x[idx];
        uniform int denom = 6; // 3!
        uniform int sign = -1;
        for (uniform int j=1; j<=terms; j++) {
            value += sign * numer / denom
            numer *= x[idx] * x[idx];
            denom *= (j+3) * (j+4);
            sign *= -1;
        }
        result[idx] = value;
    }
}
```
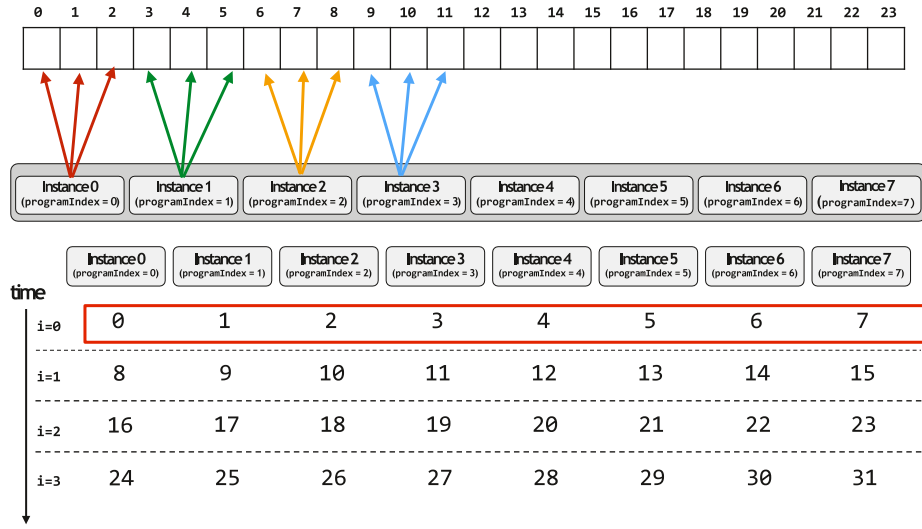
Figure 10: Example of execution with 8 instances (`programCount` equal to 8). A single "packed vector load" instruction efficiently implements this. For all program instances, since the eight values are contiguous in memory.

3.2   Shared Address Space Model

We give a general introduction to memory in the chapter 2.2. In parallel computing theory, **each thread communicates with other threads using read-/write operations**. These instructions operate on a special area called the **Shared Address Space** (also called **Shared Variables**).

Definition 3: Shared Address Space

The **Shared Address Space** view of a parallel platform supports a common data space that is accessible to all processors. Processors interact by modifying data objects stored in this shared-address-space. [5]

Now the first and trivial question should be: a powerful tool is the possibility to allow communication between threads, but *how can we guarantee that two or more threads accessing the same resource do not create well known problems, such as* race condition?

This property, commonly called **mutual exclusion** or **atomic operation**, can be guaranteed with some techniques:

- **Lock/Unlock mutex around a critical section**:

```
1  Lock lock_variable;
2
3  // some operations, such as spawn of threads
4
5  lock_variable.lock();
6  // critical section
7  lock_variable.unlock();
```

- Some languages have first-class **support for atomicity of code blocks**:

```
1  atomic {
2      // critical section
3  }
```

- Intrinsics for **hardware-supported atomic read-modify-write operations**:

```
1  atomicAdd(x, 10);
```

The shared address space **requires hardware support to be efficiently implemented**. The main idea is that **each processor can directly reference the contents of any memory location**. Some interesting examples that can be explored in depth are: SUN Niagara 2, Knights landing (KNL): 2nd Generation Intel Xeon Phi processor.

4   Parallel Programming Models and pthreads

4.1   How to create parallel algorithms and programs

Although *parallel algorithms* and *parallel programs* are in the same father set, the parallel computing topic, these two arguments are a little different.

> **Definition 1: Parallel Algorithms**
>
> A **parallel algorithm**, as opposed to a traditional serial algorithm, is an **algorithm which can do multiple operations in a given time**.

> **Definition 2: Parallel Programs**
>
> A **parallel program** is a **program that uses multiple CPU cores**, with **each core performing a task independently**.

However, designing *parallel algorithms* is not an easy task because there is no heuristic for designing *parallel algorithms*. There are some rules that help in the design. The same reasoning applies to *parallel programs*, because they depend on the chosen language and architecture.

Furthermore, there is no single correct solution, but several possible parallel solutions. A **good first approach is to start with machine-independent issues** (concurrency) and **delay target-specific issues as much as possible**.

Design a parallel algorithm	Design a parallel program
Understand the problem to be solved	Analyze the target architecture(s)
Analyze data dependencies	Choose the best parallel programming model and language
Partition the solution	Analyze the communications (cost, latency, bandwidth, visibility, synchronization, etc.)

Table 2: Design parallel algorithms and parallel programs.

The **PCAM (Partitioning, Communication, Agglomeration, Mapping)** methodology described by Argonne National Laboratory is intended to promote an exploratory **approach to design in which machine independent issues**, such as *concurrency*, are **considered early** and **machine specific aspects of design are deferred until late in the design process**. In other words, we immediately consider the machine-independent issues (e.g., concurrency) at the beginning of the design approach, and all machine-specific aspects are postponed to an advanced stage of the design process.

This methodology structures the design process into **four distinct stages**:

1. **Partitioning**. The **computation** that is to be performed and the **data** operated on by this computation are **decomposed into small tasks**. Practical issues such as the number of processors in the target computer are ignored, and attention is focused on recognizing opportunities for parallel execution.

2. **Communication**.   The communication required to **coordinate task execution** is determined, and appropriate **communication structures and algorithms are defined**.

3. **Agglomeration**. The task and **communication structures** defined in the first two stages of a design are **evaluated with respect to performance requirements and implementation costs**. If necessary, tasks are combined into larger tasks to improve performance or to reduce development costs.

4. **Mapping**. **Each task is assigned to a processor** in a manner that attempts to satisfy the competing goals of **maximizing processor utilization** and **minimizing communication costs**. Mapping can be specified statically or determined at runtime by load-balancing algorithms.

In the **first two stages**, we focus on **concurrency** and **scalability** and seek to **discover algorithms with these qualities**. In the **third and fourth stages**, attention shifts to **locality** and **other performance-related issues**.
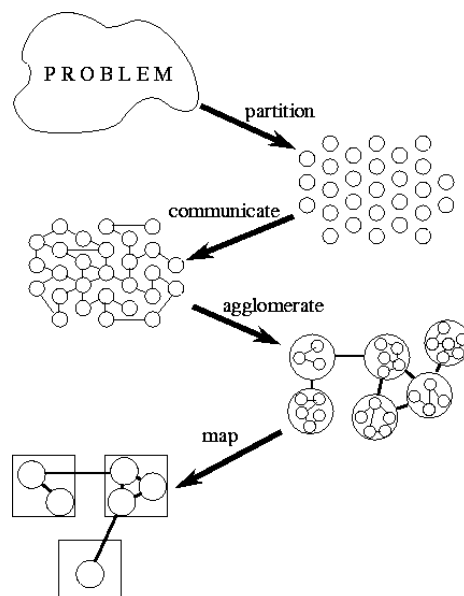

Figure 11: PCAM design methodology for parallel programs. Starting with a problem specification, develop a partition, determine communication requirements, agglomerate tasks, and finally map tasks to processors.

4.2 Analyze parallel algorithms

Whether we want to analyze our parallel algorithm created with the PCAM model or evaluate a general parallel algorithm, we need some metrics.

The classical **metrics** needed to evaluate a parallel algorithm are:

- **Time complexity**: quantifies the amount of **time required to produce a solution**.

- **Resource complexity**: quantifies how many **resources are needed to produce the solution in that time**.

In general, to analyze a **parallel algorithm**, we can consider its **structure as a directed acyclic graph (DAG)**[2], where the nodes are the task and the edges are the data dependencies.

🗒 Parallel Algorithm Terminology and Metrics

- **Concurrent tasks**, each task is executed *independently*.

- **Parallel tasks**, each task is executed at the *same time* (because multiple computing resources are available).

- **Work** W is the *number of operations executed*. It may be higher than the sequential version of the algorithm due to communication overhead, etc.

- **Span** S is the *longest chain of dependencies* (i.e., the critical path) that determines the *minimum time required to execute the algorithm*. This is a *lower bound* on the running time, regardless of the number of processors. The range indicates the ability of an algorithm to get better performance on more processors.

    ### ❓ How do we calculate the Span metric?

    1. As we just said, we **represent a parallel algorithm as a DAG** graph, where nodes represent tasks and edges represent dependencies between tasks;

    2. We **assign weights to each node** that represent the **time required to perform the corresponding task**;

    3. We try to **find the Critical Path**. In other words, we determine the path from the start node to the end node that has the *maximum cumulative weight*;

    4. Finally, the **sum** of the **weights of the nodes on the critical path** gives us the span value!

---

[2]A **Directed Acyclic Graph (DAG)** is a directed graph, i.e. with oriented edges, without cycles.

- **Parallelism** P is the ***measure of efficiency in the use of resources***. Trivially, it is the number of operations performed divided by the longest chain of dependencies:

$$P = \frac{W}{S} \tag{11}$$

It indicates **how many processors can be effectively used by the computation**. If the work is equal to the span, the parallelism is 1 and the computation is sequential. Ideally, but not necessarily, we win with polylogarithmic span, because if the work is $O(n \log n)$ and the span is $O(\log^2 n)$, then the parallelism is $O\left(\frac{n}{\log n}\right)$, which is actually quite high (and unlikely to be a bottleneck on most machines in the next 25 years). [4]

This measure is *one of the most important*. It indicates the **number of processors that are not idle**. It is obvious that a **good parallel algorithm is designed to have the lowest possible work** (less operation, then less resource usage, then less cost, and so on) **and the highest possible parallelism** (achievable by reducing the span, and this should be trivial, since the metric P is given by work divided by the span, so reducing the denominator, you can get a higher value).

As in all things, there is a **trade-off** between the lowest possible "work" and the highest possible "parallelism". Reducing the work too much could eliminate the possibility of parallelizing our algorithm, and on the other hand, reducing the span too aggressively could cause communication/synchronization overhead.
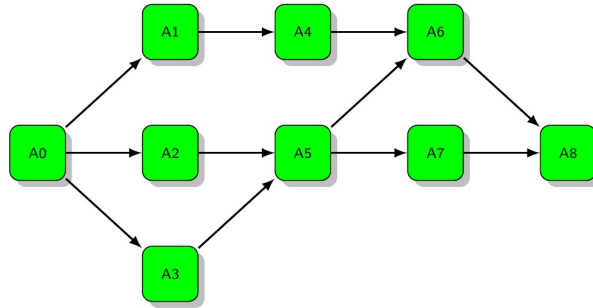

Figure 12: Example of DAG implementation with work equal to 9, span equal to 5, and parallelism equal to 1.8 ($9 \div 5$). The span calculus is not well known, has been calculated *a priori*.

Finally, we use a mathematical annotation and not only a graphical (DAG) annotation. The **Composition Rules** help determine how to combine smaller parallel tasks into a larger algorithm, while analyzing the Work and Span of the combined algorithm:

- **Single operation**. An operation takes 1 unit of work and 1 unit of span time.

$$W(op) = 1 \qquad S(op) = 1$$

- **Sequential Composition**.

  - The total work of executing e_1 and e_2 **sequentially** is the sum of their individual work.

  $$W(e_1, e_2) = W(e_1) + W(e_2)$$

  - The total span of executing e_1 and e_2 **sequentially** is the sum of their individual spans.

  $$S(e_1, e_2) = S(e_1) + S(e_2)$$

- **Parallel Composition**.

  - The total work of executing e_1 and e_2 in **parallel** is still the sum of their individual works.

  $$W(e_1 || e_2) = W(e_1) + W(e_2)$$

  - The total span of executing e_1 and e_2 in **parallel** is the ***maximum of their individual spans***, since they can be executed simultaneously.

  $$S(e_1 || e_2) = \max(W(e_1), W(e_2))$$

4.3   Technologies

Some famous architecture to work with parallel programming:

- **Verilog/VHDL** are *hardware description languages*. The target architectures are *ASIC and FPGA*. The **parallelism** and the **communication** are **explicit**.

  ✔ **Pros**

  - Complete control on computation and memory
  - No overhead introduced in the computation
  - Provides access to potentially large computational power

  ⚠ **Cons**

  - Requires specific hardware (e.g., ASIC or FPGA) to implement functionality
  - Difficult to learn: completely different programming language and programming paradigm
  - Depends on the chosen target architecture

- **MPI** is a *library*. The target architectures are *Multi CPUs*. The **parallelism** is **implicit** and the **communication** is **explicit**.

  ✔ **Pros**

  - Can be adopted on different types of architecture
  - Scalable solutions
  - Synchronization and data communication are explicitly managed

  ⚠ **Cons**

  - Communication can introduce significant overhead
  - Programming paradigm more difficult than shared memory-based ones
  - Standard does not reflect immediately advances in architecture characteristics

- **PThread** is a *library*. The target architectures are *Multi-core CPUs*. The **parallelism** is **explicit** and the **communication** is **implicit**.

  ✔ **Pros**

  - Can be adopted on different types of architecture
  - Explicit parallelism and full control over application

  ⚠ **Cons**

- – Task management overhead can be significant
- – Not easily scalable solutions
- – Low level API

- **OpenMP** is a *C/Fortran extensions*. The target architectures are *Multi-core CPUs*. The **parallelism** is **explicit** and the **communication** is **implicit**.

  ### ✔ Pros

  - – Easy to learn
  - – Scalable solution
  - – Parallel applications can also be executed sequentially

  ### ⚠ Cons

  - – Mainly focused on shared memory homogeneous systems
  - – Requires small interaction between tasks

- **CUDA** is a *C extensions*. The target architectures are *CPU plus GPU(s)*. The **parallelism** is **implicit/explicit** and the **communication** is **implicit/explicit**.

  ### ✔ Pros

  - – Provides access to the computational power of GPUs
  - – Writing a CUDA kernel is quite easy
  - – Already optimized libraries

  ### ⚠ Cons

  - – Targets only NVIDIA GPUs
  - – Difficult to extract massive parallelism from application
  - – Difficult to optimize CUDA kernel

- **OpenCL** is a *C/C++ extensions and API*. The target architectures are *heterogeneous architecture*. The **parallelism** is **implicit/explicit** and the **communication** is **implicit/explicit**.

  ### ✔ Pros

  - – Target-independent standard
  - – Hides architecture details
  - – Same programming infrastructure for every heterogeneous architecture: CPU + GPU (and FPGA)

  ### ⚠ Cons

- – Difficult programming paradigm for its heterogeneity
- – Hiding of architecture details makes difficult to obtain best performances
- – Gradually abandoned

- **Apache Spark** is an *API*. The target architectures are *multi CPUs*. The **parallelism** is **implicit** and the **communication** is **implicit**.

    ✔ **Pros**

    - – API for different languages
    - – Explicit parallelization and communication are not required
    - – Preinstalled on cloud provide VMs

    ⚠ **Cons**

    - – Suitable only for big data applications
    - – Does not (yet) fully support GPUs

Regardless of these technologies, it is quite common to mix some of them:

- **OpenMP + CUDA**: allows to exploit multi-core CPU and GPU. CUDA is used to parallelize GPU code and OpenMP is used to parallelize CPU code.

- **MPI + OpenMP**: the most common scenario are:

    1. MPI used to express coarser parallelism (multi CPU) and OpenMP used to express finer parallelism (multi core).
    2. MPI used to implement communication and OpenMP used to parallelize computation.

- **OpenCL + Verilog or VHDL**: in principle, hardware kernels (implemented for example on FPGA) can be used as accelerators; OpenMP used to describe parallelism among different processing elements; Verilog/VHDL used to describe hardware kernel. An example of target: Intel Xeon Scalable.

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] Laxman Dhulipala Guy E. Blelloch and Yihan Sun. Introduction to parallel algorithms. https://www.cs.cmu.edu/~guyb/paralg/paralg/parallel.pdf, 2024. [Accessed 22-10-2024].

[5] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing*, volume 110. Benjamin/Cummings Redwood City, CA, 1994.

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

[7] M. Nemirovsky and D. Tullsen. *Multithreading Architecture*. Synthesis Lectures on Computer Architecture. Springer International Publishing, 2022.

[8] Wikipedia. Cache (computing) - Wikipedia. https://en.wikipedia.org/wiki/Cache_(computing). [Accessed 20-10-2024].

Index