Parallel Computing - Notes - v0.2.0-dev

260236

October 2024

Preface

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

- Course slides. [1]

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** $X(1), \dots, X(N)$
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.

  ? 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:

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

Algorithm 1: Matrix-Vector Multiply (MVM)

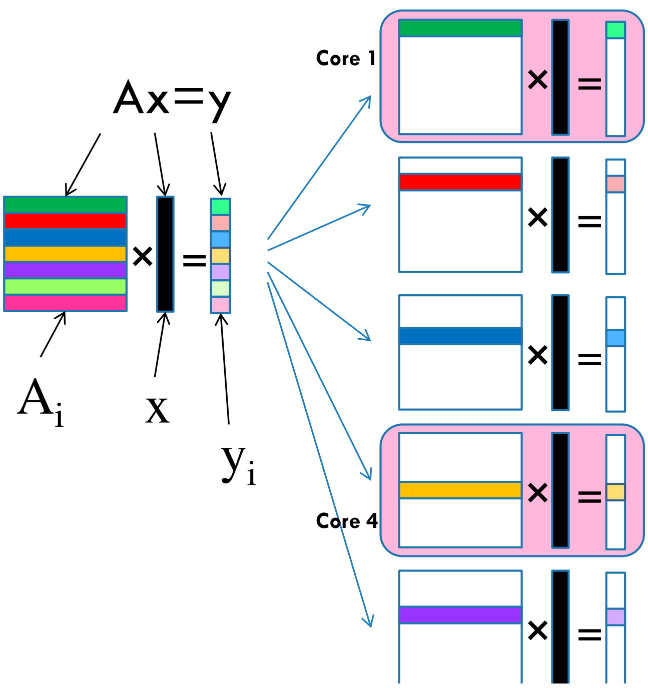


Figure 1: Example of MVM algorithm.

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.

```
BEGIN
    GLOBAL READ (A ← A(I))
    GLOBAL WRITE (A → B(I))
    FOR H = 1 : K
        IF i ≤ n ÷ 2^h THEN BEGIN
            GLOBAL READ (X ← B(2i - 1))
            GLOBAL READ (Y ← B(2i))
            Z := X + Y
            GLOBAL WRITE (Z → B(i))
        END
    IF I = 1 THEN
        GLOBAL WRITE (Z → S)
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

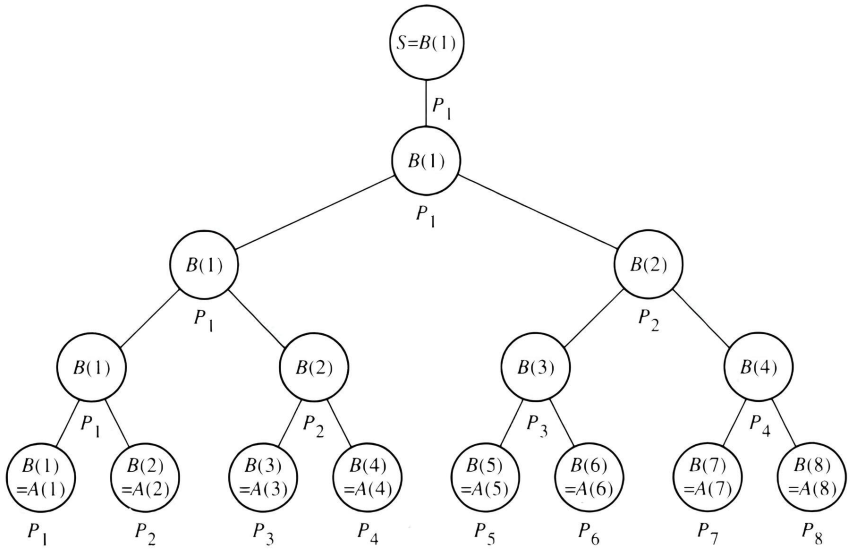
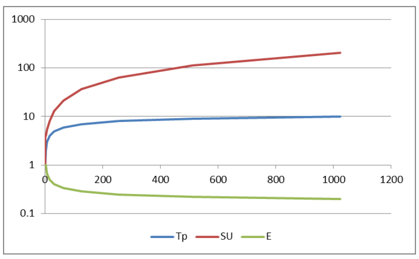


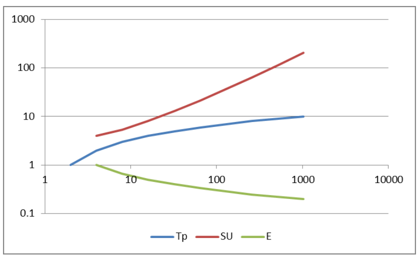
Figure 2: Computation of the sum of eight elements on a PRAM with eight processors. Each internal node represents a sum operation. The specific processor executing the operation is indicated below each node.

Performance of sum

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

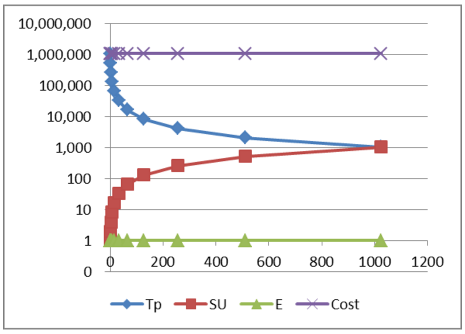
- $T^* (N) = T_1 (N) = N$
- $T_{N=P} (N) = 2 + \log N$
- $\mathrm{SU}_P = \dfrac{N}{2 + \log N}$
- $T^* (N) = P \cdot (2 + \log N) \approx N \log N$
- $E_p = \dfrac{T_1}{pT_p} = \dfrac{N}{N \log N} = \dfrac{1}{\log N}$





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)**:

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



$n = 1'000'000$

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)
- $E_p = \dfrac{2}{\log n} = \dfrac{2}{3} = 0.67$
- $\dfrac{W}{C} = \dfrac{16}{40} = 0.4$

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

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

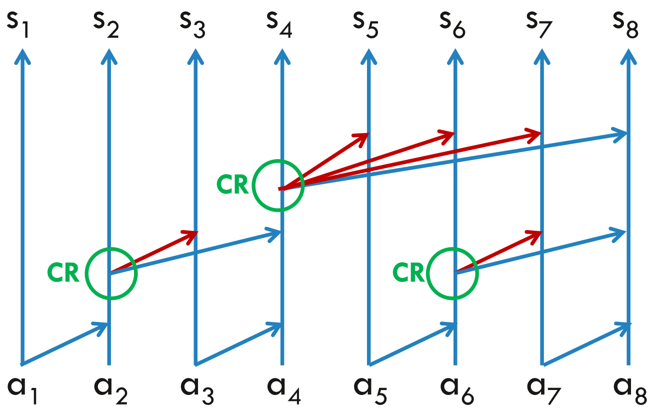


Figure 4: Prefix sum.

1.6 MM algorithm

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

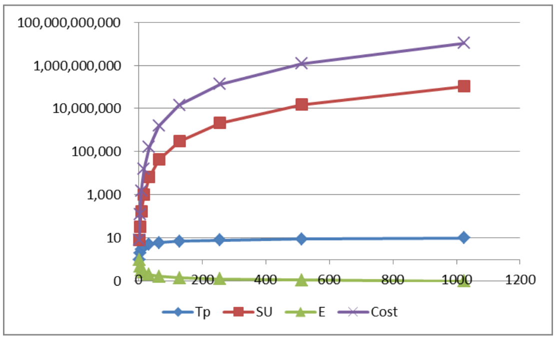
1. **Compute the two matrices** $A_{i,l}$ and $B_{l,j}$, so use the concurrent read.
2. Make the **sum**.
3. **Store** the result using exclusive write.

```
BEGIN
    T_{i,j,l} = A_{i,l}B_{l,j}
    FOR = H = 1 : K
        IF l ≤ n ÷ 2^h THEN
            T_{i,j,l} = T_{i,j,2l−1} + T_{i,j,2l}
    IF l = 1 THEN
        C_{i,j} = T_{i,j,1}
END
```

Algorithm 3: Matrix Multiply (MM)

Performance of MM

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



References

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

Index