Outline

Sequential Vs Parallel Computing

Sequential	Parallel
While Sequential programming involves a consecutive and ordered execution of processes one after another	Parallel programming involves the concurrent computation or simultaneous execution of processes or threads at the same time.
sequential programming, processes are run one after another in a succession fashion	while in parallel computing, you have multiple processes execute at the same time
Simple to implement	Complex to implement
Processor Utilization is poor	Processor utilization is efficient

Random Access Machine Model (RAM)

RAM model of serial computers:

- Memory is a sequence of words, each capable of containing an integer.
- Each memory access takes one unit of time
- Basic operations (add, multiply, compare) take one unit time.
- Instructions are not modifiable
- Read-only input tape, write-only output tape

RAM Sequential Algorithm Steps

A ***READ*** phase in which the processor reads datum from a memory location and copies it into a register.

A ***COMPUTE*** phase in which a processor performs a basic operation on data from one or two of its registers.

A ***WRITE*** phase in which the processor copies the contents of an internal register into a memory location.

Parallel Algorithm

In computer science, a **parallel algorithm**, as opposed to a traditional serial **algorithm**, is an **algorithm** which can be executed a piece at a time on many different processing devices, and then combined together again at the end to get the correct result.

Approaches to parallelism

shared-memory parallel processing

- A *thread* consists of a single flow of control, a program counter, a call stack, and a small amount of thread-specific data
- Threads share memory, and communicate by reading and writing to that memory

message-passing parallel processing

- A *process* is a thread that has its own private memory
- Threads send messages to one another.

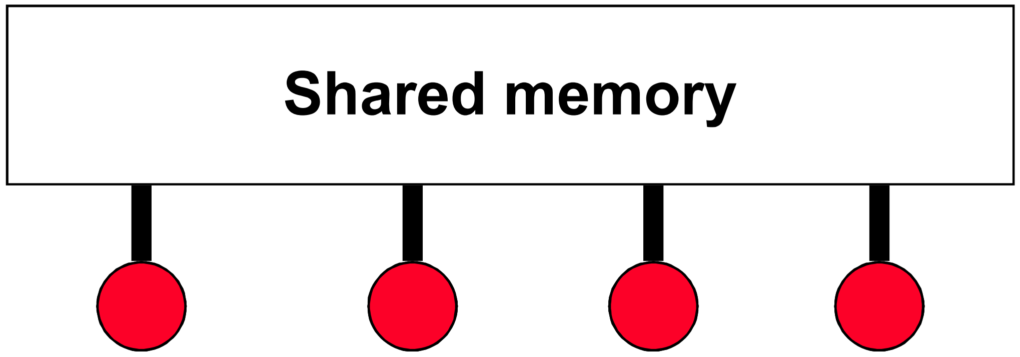
PRAM [Parallel Random Access Machine]

PRAM composed of:

- P processors, each with its own unmodifiable program.
- A single shared memory composed of a sequence of words, each capable of containing an arbitrary integer.
- a read-only input tape.
- a write-only output tape.

PRAM model is a synchronous, MIMD, shared address space parallel computer.

PRAM model of computation



PRAM model Characteristics

At each unit of time, a processor is either active or idle (depending on id)

All processors execute same program

At each time step, all processors execute same instruction on different data (“data-parallel”)

Focuses on concurrency only

Variants of PRAM model

	Exclusive Write	Concurrent Write
Exclusive Read	EREW	ERCW
Concurrent Read	CREW	CRCW

Concurrent (C) means, many processors can do the operation simultaneously in the same memory
Exclusive (E) not concurent

More PRAM taxonomy

EREW - exclusive read, exclusive write

- A program isn’t allowed to have two processors access the same memory location at the same time.

CREW - concurrent read, exclusive write

CRCW - concurrent read, concurrent write

- Needs protocol for arbitrating write conflicts

CROW – concurrent read, owner write

- Each memory location has an official “owner”

Sub-variants of CRCW

Common CRCW

- CW iff all processors writing same value

Arbitrary CRCW

- Arbitrary value of write set stored

Priority CRCW

- Value of min-index processor stored

Combining CRCW

Amdahl's law

Amdahl's law, also known as **Amdahl's argument**, is used to find the maximum expected improvement to an overall system when only part of the system is improved. It is often used in parallel computing to predict the theoretical maximum speedup using multiple processors

Amdahl's law

In the case of parallelization, Amdahl's law states that if *P* is the proportion of a program that can be made parallel and (1 − *P*) is the proportion that cannot be parallelized (serial), then the maximum speedup that can be achieved by using *N* processors is

$$S(N) = \frac{1}{(1 - P) + P/N}$$

As N goes to infinity,

the maximum speedup $S(N) = 1/(1 - P)$

Amdahl's law Example 1

- 95% of a program's execution time occurs inside a loop that can be executed in parallel. What is the maximum speedup we should expect from a parallel version of the program executing on 8 CPUs?

$$\psi \le \frac{1}{0.05 + (1 - 0.05)/8} \cong 5.9$$

Amdahl's law Example 2

- 5% of a parallel program's execution time is spent within inherently sequential code.The maximum speedup achievable by this program, regardless of how many PEs are used, is

$$\lim_{p \to \infty} \frac{1}{0.05 + (1 - 0.05)/p} = \frac{1}{0.05} = 20$$

Limitations of Amdahl's law

- Amdahl's law only applies to cases where the **problem size is fixed**.

- **Example**

- If 75% of a process can be parallelized, and there are four processors, then the possible speedup is

$$1 / ((1 - 0.75) + 0.75/4) = 2.286$$

- But with 40 processors the speedup is only

$$1 / ((1 - 0.75) + 0.75/40) = 3.721$$

- This has led to conclude that having lots of processors won’t help very much

Brent’s Theorem

Brent's theorem specifies that for a sequential algorithm with t time steps, and a total of m operations, that a run time T is definitely possible on a shared memory machine (PRAM) with p processors.

It may be possible to implement this algorithm faster (by scheduling instruction differently to minimise idle processors, for instance), but it is definitely possible to implement this algorithm in this time given p processors.

Brent’s Theorem : Given a parallel algorithm ***A*** that performs m computation steps in time ***t*** , then ***p*** processors can execute algorithm in A in time

$$T = t + (m-t)/p$$

Brent's Theorem Example

Say we are summing an array.

```
        for (i=0;i < length(a); i++)
            sum = sum + a(i);
```

Using this algorithm, each add operation depends on the result of the previous one, forming a chain of length n, thus **$t = n$.** There are n operations, so **$m = n$.**

$$T = n + 0/p.$$

so no matter how many processors are available, this algorithm will take **time n.**

Brent’s Theorem Example

Now consider the **adding the array Parallely**:

sum(a) = $((A_0 + A_1) + (A_2 + A_3)) + ((A_4 + A_5) + (A_6 + A_7))$

We can **add A_2 to A_3** without needing to know what **$A_0 + A_1$** is.

For an array of length n, the longest chain(s) will be of length log(n). t = log(n). m = n.

T = log(n) + (n - log(n))/p.

Brent’s Theorem
Analysis/Importance

This tells us many useful things about the algorithm:

If we have n processors, the algorithm can be implemented in log(n) time

If we have log(n) processors, the algorithm can be implemented in 2log(n) time (asymptotically this is the same as log(n) time)

If we have 1 processor, the algorithm can be implemented in n time.

Brent’s Theorem

Analysis/Importance

If we consider the amount of work done in each case, with one processor, we do n work, with log(n) processors we do n work, but with n processors we do nlog(n) work.

The implementations with 1 or log(n) processors, therefore are cost optimal, while the implementation with n processors is not.

It is important to remember that Brent's theorem does not tell us how to implement any of these algorithms in parallel; it merely tells us what is possible.

Efficient and optimal parallel algorithms Analysis

- **A parallel algorithm is efficient** iff
  - it is fast (e.g. polynomial time) and
  - the product of the parallel time and number of processors is close to the time of at the best know sequential algorithm

$$T^{\text{sequential}} \approx (T^{\text{parallel}} \cdot N^{\text{processors}})$$

- **A parallel algorithms is optimal** iff this product is of the same order as the best known sequential time

Metrics:
Speedup , Efficiency and Cost

- **Speedup:**

$$S = \frac{\text{Time(sequential algorithm-}T_s)}{\text{Time(parallel algorithm- }T_p)}$$

- **Efficiency:**

$$E = S / N$$

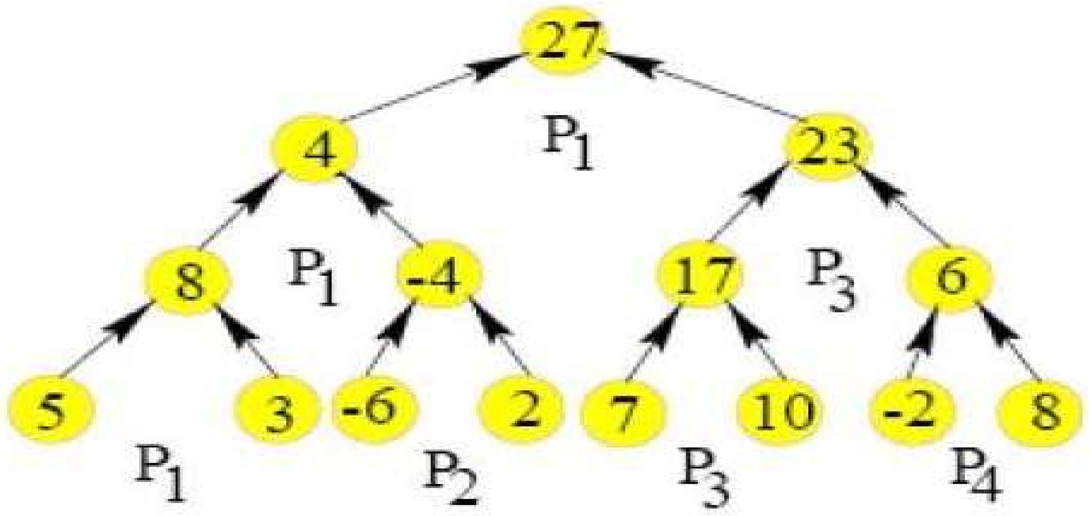
N is the number of processors

- **Cost :**

$$C = N * T_p$$

Analysis a parallel algorithm

Example



Adding n numbers in parallel

Analysis a parallel algorithm

Example:

Example for 8 numbers:

We start with 4 processors and each of them adds 2 items in the first step.

The number of items is halved at every subsequent step. Hence log *n* steps are required for adding *n* numbers. The processor requirement is $O(n)$.

A parallel algorithms is analyzed mainly in terms of **its time, processor and work complexities.**

Time complexity $T(n)$: How many time steps are needed? $T(n) = O(\log n)$

Processor complexity $P(n)$: How many processors are used? $P(n) = O(n)$

Work complexity $W(n)$: What is the total work done by all the processors? $W(n) = O(n \log n)$