Complexity Analysis of a Parallel Algorithm for Symbolic Controller Synthesis

Bachelor Thesis

Scientific work to obtain the degree
B.Sc. Electrical Engineering and Information Technology

Department of Electrical and Computer Engineering
Technische Universität München

Supervised by Prof. Dr. Majid Zamani & Mr Mahmoud Khaled
Assistant Professorship of Hybrid Control Systems
TUM Department of Electrical and Computer Engineering

Submitted by Guay Wei Jun Gordon

Filed On München, on 12 June 2018

Abstract

This Bachelor Thesis represents an already-established algorithm for the synthesis of symbolic controllers into a parallel algorithm for pFaces. This thesis begins with an overview of OpenCL, Big-O Notation, Search Algorithms, Parallel Algorithms and analysing parallel algorithms in order to provide a better understanding of the complexity and design of the algorithms in relation to the two sub-tasks in Fixed-point (FP) kernels under pFaces. The first of the two tasks is Abstract Algorithm. Second, Synthesis algorithm. For each individual FP kernels, a parallel algorithm is represented from an established C code by implementing a combination of C++ and OpenCL.The algorithms are mainly described for the PRAM(Parallel Random Access Machine). Following that, an evaluation of the application of parallel complexity is conducted. From these two tasks, they will be able to integrate into pFaces. In addition, it reviews and explains the theory of parallel computation. Finally, it presents an understanding to parallel algorithms, OpenCL and complexity theory.

Acknowledgement

For Mum and Dad, the bastions of support during my pursuit of this Bachelor's degree. You have provided me with unyielding love and encouragement over the years. Nothing can express my gratitude.

I wish to thank my supervisors Prof. Dr Majid Zamani and Mr Mahmoud Khaled for introducing me to parallel algorithms, OpenCL and the complexity theory for algorithms. I am grateful to their dedication, encouragement and patience with me during this Bachelor Thesis in Munich.Also, to the professionals of Department of Hybrid Control System and TUM Asia, without their invaluable assistance and support, the completion of this thesis would not have been possible.

Contents

1 OpenCL - Open Computing Language

1.1 What is OpenCL?

OpenCL (Open Computing Langauge) is a specification standard for the development of data parallel applications. Developed by Khronos Group, it is a programming framework which integrates parallel computing across multiple vendors and hardwares. It uses low-level language for high-performance, heterogenous, data parallel computation.

OpenCL takes advantage of all the computing power on Personal Computers, including processors, Graphic Processing Units (GPUs) and Central Processing Units (CPUs). It accelerates parallel computations and is most appropriate for the computing of devices with large amounts of data parallelism. Each Computation problem breaks down into smaller lots and is executed with the various processing unit. [1]

1.2 Basic Ideas of OpenCL Programs

OpenCL is a programming language for writing parallel programs to leverage on the advantages of the number of processors and multi-core architecture in a computing system. An OpenCL program consists of three basic components: Compute Devices - processors such as GPUs and CPUs; Data Buffers - a region of the physical memory storage used to temporarily store data and; Kernels - an OpenCL program that processes blocks of data stored in data buffers, where the data is usually organised in N dimensional arrays. [2]

1.3 OpenCL Terms

1. Platform - Vendor Specific OpenCL implementation
2. Context - Devices selected to work together
3. Device - Physical devices supporing OpenCL, CPU/GPU/Accelerator
4. Host - Client Side calling code, the code you write from your application
5. Kernel - Blueprint function which does the work
6. Work Item - A unit of work executing a kernel
7. Work Groups - A collection of work items

8. Command Queue - The only way to communicate with a device, send it commands
9. Memory - local / global / private / constant
10. Buffer - Area of memory on the GPU
11. Compute unit - Basically a work group and its local memory [3]

1.4 Why are there developments for OpenCL?

OpenCLs computational performance has shifted from clock speed to cores, having multiple CPUs and programmable GPUs. In addition, OpenCL supports parallel computations which means that it has the ability to execute the same code on various architectures. Further, OpenCL is device agnostic. In other words, it not only has the flexibility to move from device to device, but, it can be anything so long as it fulfils the requirement of executing the program.

1.5 Goals of OpenCL

The goals of OpenCL can be described as follows:

1. Single Computing Model for clean Application Programming Interface (API)
2. Easy to use and needs to be lightweight and efficient
3. Minimum errors for math functions
4. IEEE-754 compliant in round behaviour
5. Ability to write a code and run it on different accelerator devices

1.6 Uses of OpenCL

OpenCL is frequently used to leverage CPUs, GPUs and other processors to accelerate parallel computations. It is able to obtain significant speedups for computationally intensive applications and is applied in many industries such as medical and financial institutions. OpenCL is also used for image, video and audio processing. Additionally, OpenCL has the ability to write accelerated portable code across different devices and architectures. [4]

1.7 OpenCL Platform Model

The OpenCL Platform Model consists of one **Host** and one or more **OpenCL Devices**. Each of these devices is composed of one or more compute units. Each compute unit is divided into one or more processing elements as shown in Figure 1 below. The memory in the model is further divided into host memory and device memory.

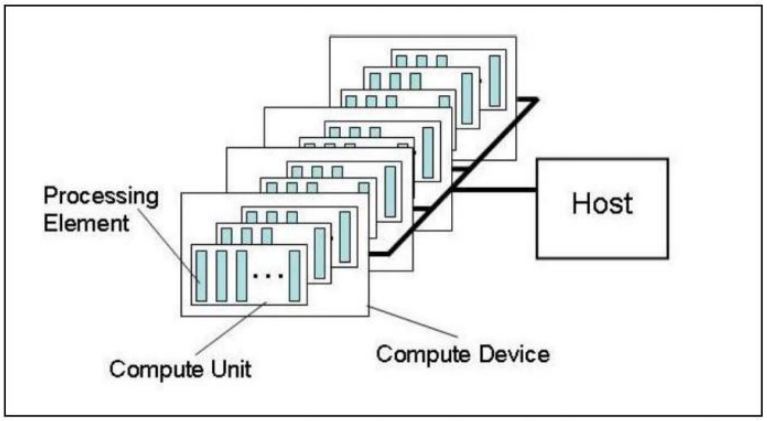


Figure 1: The OpenCL Platform Model [1]

1.8 OpenCL Memory Model

The Open CL Memory Model consists of four virtual memory regions.

1. Private Memory
2. Local Memory
3. Global Memory
4. Constant Memory

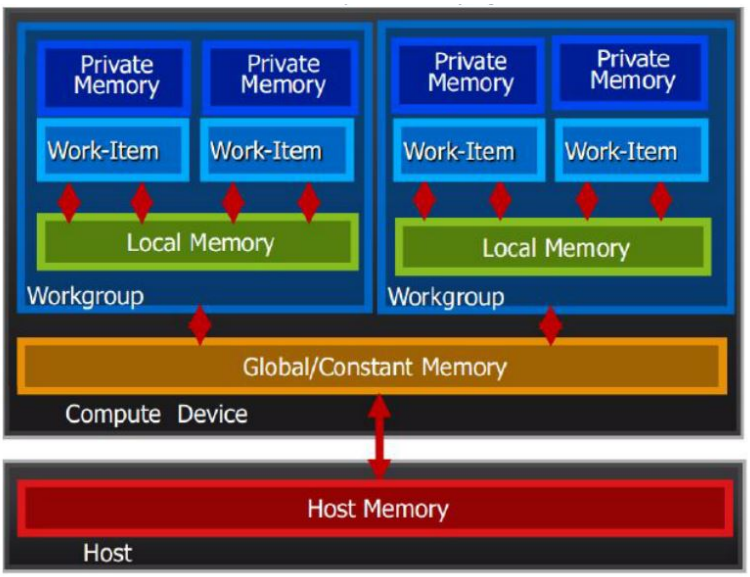


Figure 2: An OpenCL Memory Model [1]

1.8.1 Private Memory

In a work-items private memory, there is scarcity in resources and only a few tens of 32-bit words per work-item can be executed on a GPU. Private memory is created and managed by delcaring it statically inside a kernel. If too much memory is used, it either spills into the global memory or reduces the number of work-items which can be executed at the same time.This may result in a decrease in performance. [2]

1.8.2 Local Memory

A work-group's shared memory usually consists of tens of kilobytes per compute unit. The local memory is used to hold data which can be reused by all the work-items in a workgroup. Due to multiple work-groups running concurrenly on each compute unit, only a fraction of the total local memory size may be available to each workgroup. Local memory is created and managed by host with the following set of instruction: [2]

$$cl :: LocalSpaceArg\ localmem = cl::Local(sizeof(float)*N);$$

1.8.3 Global Memory

In the global memory, read or write access is available for all work-items and all work-groups.

1.8.4 Constant Memory

In the Constant Memory, there is only read access for all work-items and work-groups.[2]

1.8.5 Rules of OpenCL Memory Model

1. Threads can share memory with other threads in the same Work-Group
2. Threads can synchronize with other threads in the same Work-Group
3. Global and Constant Memory is accessible by all threads in all work-groups
4. Global and Constant Memory is often cached inside a Work-Group
5. Each thread has registers and private memory
6. Each Work-Group has a maximum number of registers it can use.

1.9 Setting up an OpenCL Program

1. Setup
    - Get the **devices**
    - Create a **context** (for sharing between devices)
    - Create command **queues** (for submitting work)
2. Compilation
    - Create a **program**
    - Build the program (**compile**)
    - Create **Kernels**
        - ***get_global_id()*** uniquely identifies each work item executing the kernel
        - ***get_local_id()*** uniquely identifies each work item in a work group

3. Create memory objects
4. **Enqueue writes** - to copy data to the GPU
5. Set the kernel arguments
6. **Enqueue** kernel executions
7. **Enqueue reads** to copy data back from the GPU
8. **Wait** for all commands to finish (Wait is needed that the enqueue actually executes the queue because OpenCL is asynchronous and runs at the same time)

1.10 Understanding the Host Program

1. The host program is the code that runs on the host to:
   - Setup the environment for the OpenCL program
   - Create and manage kernels
2. There are 5 simple steps in a basic host program:
   - Define the **platform**... platform = devices + context + queues
   - Create and Build the **program** (dynamic library for kernels)
   - Setup **memory** objects
   - Define the **kernel** (attach arguments to kernel function)
   - Submit **commands** .. transfer memory objects and executes the kernel

1.11 Global Dimensions

Parallelism is defined by the 1D,2D or 3D **global dimensions** for each kernel execution (Kernel is the item you are executing). A **work-item** (thread) is executed for every point in the global dimensions.

1.12 Local Dimensions

The global dimensions are broken down evenly into **local work-groups**. Each work-group is logically executed together on one **compute unit**. Synchronization is only allowed between **work-items** in the same **work-group**

1.13 Data Movement

Data movement involves getting the data to the device and back. Therefore, the user has to **allocate** global data, **write** it from the host, **allocate** local data and **copy** data from global to local memory. [4]

2 Big-O Notation

The Big-O notation, also known as Landau's symbol, is a symbolism used in complexity theory to describe the asymptotic behaviour of functions. It is used in Computer Science to illustrate the performance and complexity of an algorithm. The Big-O notation depicts the speed of function's growth or decline and specifically describes the worst-case scenario or execution time of the algorithm.Big-O describes an upper bound of the algorithm.With the help of the Big-O notation, we are able to make a statement on the efficiency of an algorithm.

Big-O analysis not only provides a good illustration to evaluate an algorithm's efficiency at a high level, but also demonstrates the expected behaviour in relation to input sizes. An understanding of the Big-O will allow us to further understand a function and determine the type of runtime expected and whether any improvements can be made.[5]

2.1 Understanding Big-O notation

How efficient is a particular algrothim? Efficiency covers a span of resources which includes memory usage, disk usage, and CPU (time usage). Performance describes how much time or memory is used when the program is running.
Complexity describes how the algorithm scales when the size of the problem gets arbitrarily large. When considering the efficiency of an algorithm, we always look at the worst case scenario. [6]

2.1.1 Upper and Lower bounds

A substantial part of computer science research consists of designing and analysing new algorithms which are more efficient. This would establish an 'upper' bound. Most of the upper bounds focuses on the execution time and describes the asymptotic behaviour and is expressed using the Big-O notation.
An **Upper Bound** is established by the algorithm using the least number of steps in the worst case scenario. Conversely, a **Lower Bound** indicates to us that no algorithm will be able to solve problem in lesser number of steps in the worst case. Thus, there will not be a reason for trying to design algorithms which would be more efficient.

2.2 Orders of Functions

Notation	Name	Example
O(*1*)	Constant	Calculating $(-1)^n$
O(n)	Linear	Searching for item in an array
O(n^2)	Quadratic	Selection Sort, Insertion Sort
O(n^3)	Quadratic	Multiplication of 2 $n \times n$ matrices
O(*log(n)*)	Logarithmic	Binary Search
O(n *log(n)*^{c})	Linearithmic	Heapsort, merge sort
O(c^n)	Exponential	Solving travelling salesmen problem by DP
O(n!)	Factorial	Solving travelling salesmen by brute force

2.3 Functions of Big-O notation

$O(1)$ describes an algorithm that will execute within the same time regardless of the size of the input data set. The input array could be 1 or many items, but $O(1)$ would only require one "step". The order of 1 also means that it is a constant.
$O(n)$ describes an algorithm which as a lineary growth in performance in direct proportion to the size of the input data. $O(n)$ is usually seen as a linear search finding all values one by one. It is executed by locating one match in the whole array. Each of the input accounts for a smaller share of processing.
O(n^2) describes an algorithm with a performance behaviour that is directly proportional to the square of the size of the input data. It is seen as a square running time and is typical of algorithms that all pairwise combinations off data item process. An example of this algorithm would be multiplying a matrix by a vector.
O(n^3) describes an algorithm that is usually executed with the multiplication of matrices
O(2^n) describes an algorithm whose growth doubles with each addition to the input data set. The curve of O(2^n) function is exponential. The curve starts off slow, but exponentiates quickly. An example of the O(2^n) function is the recursive calculation of Fibonacci nummbers.
O(*log n*) shows that the algorithm executes in proportion to the logarithm of the problem size. (i.e the algorithm increases with n at a slower rate). As the amount of data increases with an increase in n, the $log(n)$ result will be dramatically different. An example of the O($log(n)$) function algorithm would be the binary search algorithm.
O(n *log* n) describes a more advanced sorting algorithm, quicksort, mergesort. Mergesorts sorts the elements in the array recursively. This means that

the algorithms to the problem is solved by dividing it into sub-problems. This is much more efficient in looking for the particular element as values are onlly compared once.
O(c^n) describes an exponential time. An example would be a recursive Fibonacci implementation

2.4 Expression of Algorithms with Big O notation

The primary function of order notation in computer science is to compare the efficiency of the various algorithms. Big-O notation allows the programmer to analyse the efficiency of the algorithms and better evaluate the performance of an algorithm.

2.5 Growth Functions of Big-O notation

The Big-O notation enables us to represent the asymptotic growth of a function neglecting the constants and lower-order additive terms. The order of growth is determined by the fastest growing term.

2.5.1 How Big-O grows with respect to input

In the Big-O notation, the runtime is expressed by how quickly it grows relative to the input and when the input gets arbitarily large.

2.5.2 How fast runtime grows

The runtime of the algorithm is dependant on the speed of the processor and therefore, the Big-O notation represents how quickly the runtime grows

2.5.3 Relative to input

In the Big-O notation, the size of the input n affects the growth of the runtime. The runtime growth is evaluated by means of a variable. As such, the size of the input is used in terms of a variable.

2.6 Typical Growth Behaviour of Big-O Functions

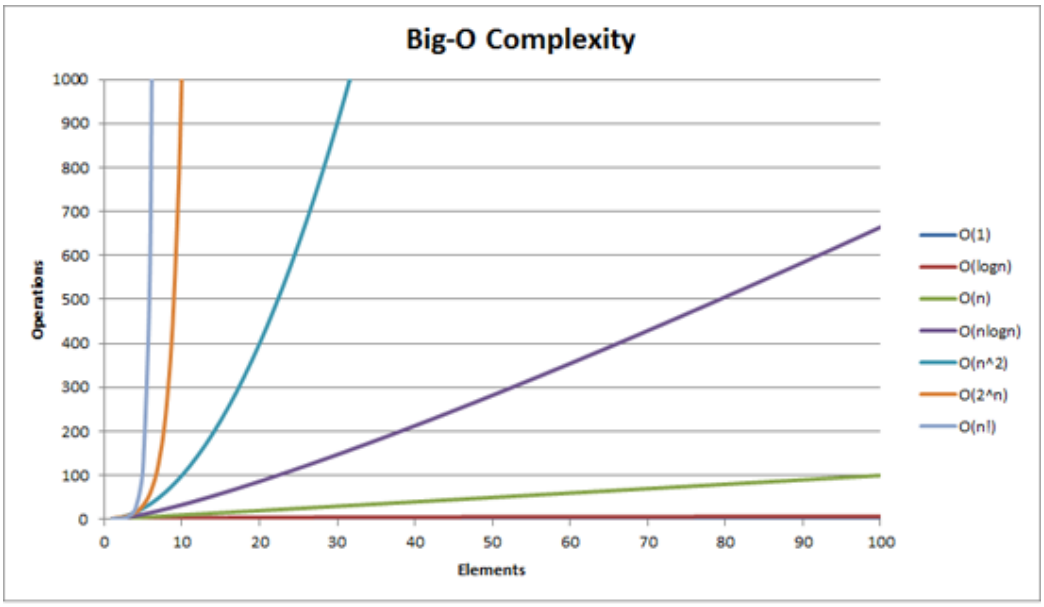


Figure 3: Growth Behaviour of Functions [7]

2.7 Properties of Calculations

The following shows the calculation rules of the Big-O notation:

1. $f(n) = O(f(n))$
2. Removing the multiples
   $c \times O(f(n)) = O(f(n))$, where c is a constant
3. Sum Rule:
   $O(f(n)) + O(f(n)) = O(f(n))$
4. Maximum values:
   $O(f(n)) + O(g(n)) = O(max(f(n), g(n)))$
5. $O(O(f(n))) = O(f(n))$
6. Multiplication of terms:
   $O(f(n)) \times O(g(n)) = O(max(f(n), g(n)))$
7. $O(f(n) \times g(n)) = f(n) \times O(g(n))$

2.8 Big-O - Exponential Complexity Example O(2^n)

If algorithm A's worst-case time complexity $t_A(n)$=(n^2), and algorithm B's worst-case time complexity $t_B(n)$=(2^n), O(2^n) will grow much faster than O(n^2) after input size n is larger than a certain value, as shown in Figure 4.

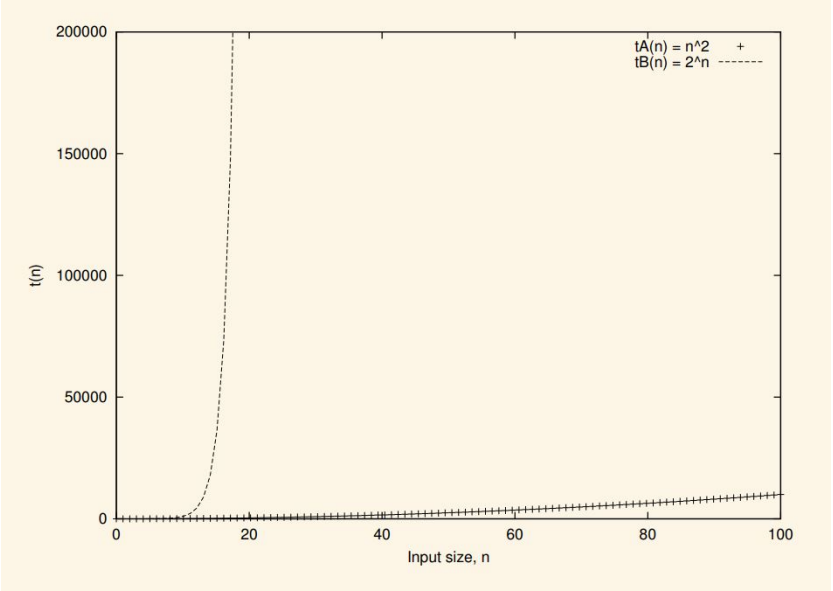


Figure 4: Comparison of curve between O(n^2) and (2^n)

Multiplying and adding constants and other terms shifts and stretches the graphs. This may result in both curves crossing paths. However the exponential function will still grow much faster than the polynomial curve.

The example in Figure 5 shows Algorithm A's worst-case time complexity $t_A(n)$=$10(n^2)$+1000 & Algorithm B's worst-case time complexity $t_B(n)$=$\frac{2^n}{10}$ For small inputs, Algorithm A, whose time complexity is quadratic, takes more time than algorithm B, whose time complexity is exponential. However, when the input size exceeds 15, Algorithm A becomes faster and, subsequently, the larger the input, the larger the advantage Algorithm A has over Algorithm B.[8]

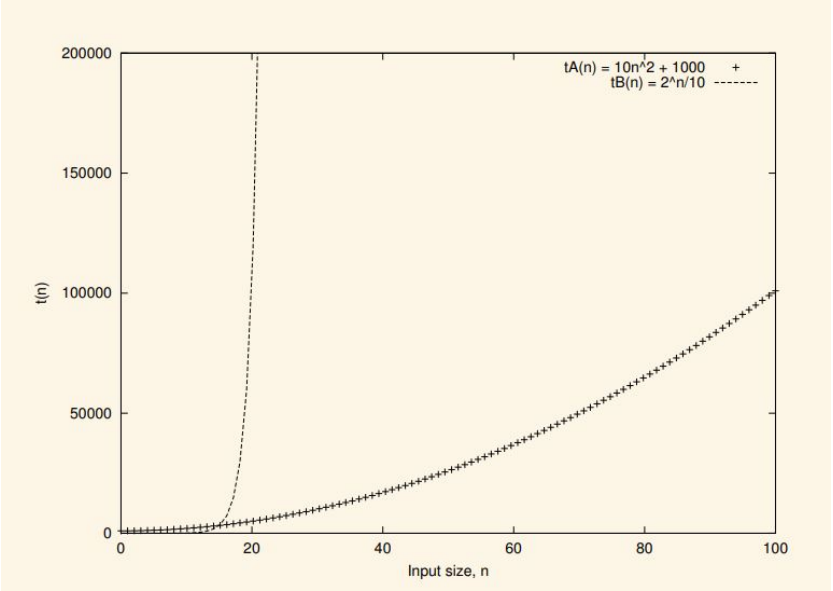


Figure 5: Comparison of curve between $t_A(n)$=$10(n^2)$+1000 and $t_B(n)$=$\frac{2^n}{10}$

3 Computation of Big-O Notation

3.1 Linear Search Algorithm (LSA)

LSA is a search which traverses a collection of elements until the desired element is found or until the entire collection of elements has been searched.

3.1.1 Execution

The linear search is also known as the sequential search where it begins with a first element and sequentially steps through an array, searching for the particular element until the particular element is located. The algorithm iterates across the array from left to right, searching for a particular element until it is found.

3.1.2 Run Time Analysis

In the best case scenario, the search algorithm will produce immediate results where $T(n) = O(1)$. In the worst case scenario, the search has to browse through all the elements in the array which produces $T(n) = O(n)$. In an average case, a successful search on the assumption that each arrangement of the elements is equally likely, therefore,

$$T(n) = \frac{1}{n} = \sum_{i=1}^{n} i = \frac{n+1}{2} = O(n)$$

The LSA is a very simple process and is well suited for single-linked list. This procedure is also suitable for unsortable array, however it is only practical for small values of n.

3.2 Binary Search Algorithm (BSA)

Binary Search Algorithm is an algorithm which searches a sorted array by reducing the search area by half each time. The search algorithm begins with an interval that covers the whole array. If the target number is less than the item in the middle interval, the search will then proceed only with the lower half of the array. This step is repeated until the value is located or if the interval is empty.

3.2.1 Execution

The BSA adopts the approach of divide and conquer. With each step, it halves the search area by looking for an element with the element at the

center position of the entire array. The goal is to locate the element in the middle. If the ideal number is equal to that, the search is completed. Otherwise, the the search area will be reduced by half with each step.

3.2.2 Runtime Analysis of BSA

The advantage of a binary search over a linear search is astounding for searching in an array for large numbers. In the worst case scenario, the search area must be halved until it just contains 1 unit, which has a complexity of $O(log(n))$. In the best case scenario, the algorithm obtains the immediate result which has a complexity of $O(1)$. For example, let input size $n = 2$ million, a linear search requires in the worst case of 2 million comparisons, whereas a Binary Search only requires $log\ (2 \times 10^6) \approx 20$ comparisons. The Binary Search Algorithm is not suited for data that changes frequently.

3.3 Breadth First Search Algorithm(BFS)

BFS is an approach most commonly used to traverse graphs.The process of Graph Traversal includes visiting all vertexes and edges only once in a well-defined order. In certain graph algorithms, the user has to ensure that each vertex of the graph is visited only once. The order of the vertices being visited is also important and it varies depending on the algorithm.

3.3.1 Execution

It begins with traversing from a starting node and subsequently traverses the graph with the surrounding neighbour nodes which are connected directly to the starting node. Subsequently, it proceeds on to the next-level neighbour nodes.

There are usually 2 types of graphs, directed and undirected. A directed graph has no particular set of directions, whereas an undirected graph has no particular set of directions.

Figure 7 shows a graph with weights, $G = (v, e)$ where $v =$ set of vertices. $e =$ set of edges. Edges may or may not have weights. We have to take note that an edge connects two vertices and can be denoted by it's 2 endpoints.

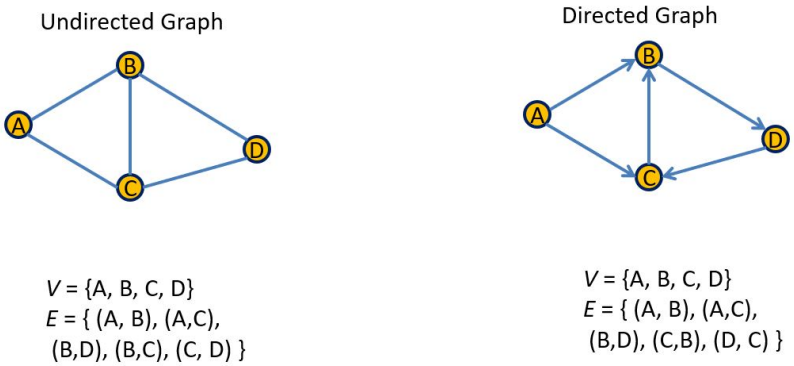


Figure 6: Example of Directed & Undirected Graph

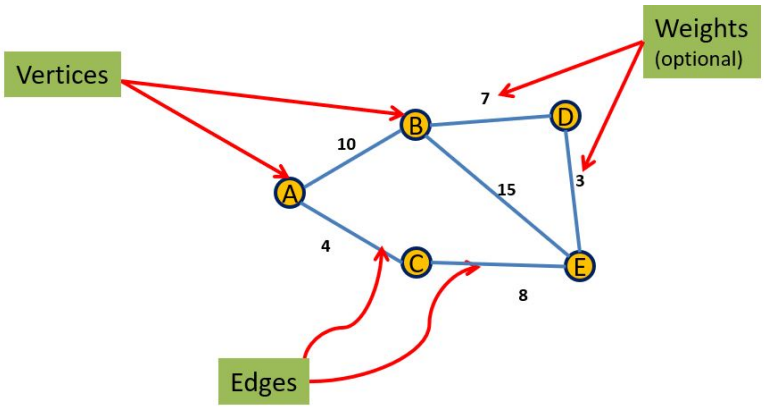


Figure 7: A graph 'G' with 5 vertexes & weights

3.3.2 Space Complexity for BFS

As all nodes of a level has to be saved until their child nodes in the next level has been visited, the space complexity is proportional to the number of nodes at the deepest level. For example, given a branching factor b and dept d, the asymptotic space complexity is the number of nodes at the deepest level $O(b^d)$. If the number of vertices in the graph is known ahead of time and additional data structures are used to determine which vertices have already been added to the queue, the space complexity can also be expressed as $O(|V|)$ where $|V|$ is the cardinality of the set of vertices.
In the worst case, the graph has a dept of 1 and all vertices must be stored. Since it is exponential in the dept of the graph, BFS is often impractical for large problems on systems with bounded space. [9]

3.3.3 Time Complexity for BFS

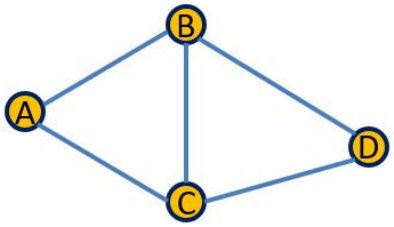
In the worst case scenario in BFS, the search has to consider all paths to all possible nodes. The time complexity of BFS would be $1+b^2+b^3+...b^d$ which is O(b^d).
The time complexity can also be expressed as O(|E|+|V|) since every vertex and every edge will be explored in the worst case. [9]

3.3.4 Adjacency Lists

The idea of adjacency lists is that you have an array (Adj) of size V. Each element of the array is a pointer to the linked list and the array is indexed by a vertex, $Adj[u]$.

3.3.5 Graph Terminologies

1. A path is a walk where no vertex is repeated
2. A walk from x to y is a sequence of vertices
3. A cycle is a walk where no intermediate vertex gets repeated and $y = x$.



Examples:
<A, B, C, D, B, A, C> is a walk.

<A, B, D> is a path.

<A, B, D, C, A> is a cycle.

Figure 8: Graph Terminologies

3.3.6 Properties of Breadth First Search

1. Breadth First Search will always reach its goal if edges are finite.
2. The queue always consists of ≥ 0 vertices of distance k from s, followed by ≥ 0 vertices of distance $k + 1$.
3. In any connected graph G, BFS computes the shortest paths from s to all other vertices in time proportional to $O(E + V)$.

3.3.7 Applications of Breadth First Search

1. Web Crawling - Google
2. Social Networking Facebook -locating friends near you or friends of friends
3. Checking mathematical conjectures
4. GPS Navigation Systems - BFS is used to find alll neighboring locations
5. Path Finding - We can use Breadth First Depth First Traversal to find if there is a path between two vertices.

4 Parallel Algorithms

4.1 Introduction

This chapter introduces several parallel models and how to specify a suitable framework for presenting and analysing parallel algorithms.
Algorithms which are executed step by step are known as sequential algorithms.Algorithms where several operations are executed simultaneously are referred to as parallel algorithms. A parallel algorithm is further defined as a process which simultaneously executes and communicates with each other to solve a given problem.
To design an efficient algorithm, the efficient use of available resources is crucial to maximise the speedup of the parallel algorithm. When the parallel algorithm has been developed, the efficiency of the algorithm has to be measured to evaluate its performance and efficiency on the parallel machine. A method used to measure the efficiency is *run time* of the algorithm. Run time can be referred to as the completion time for the algorithm to be executed completely. To be more precise, it can be described as the elapsed time between the start of the first processor until the termination of the final processor.
Directed acyclic graphs (DAGs) is used to represent certain parallel computations in a natural way.
Shared-memory model is where a number of processors communicate through a common global memory. It offers an attractive framework for the development of algorithmic techniques for parallel computations.[10]

In the discussion of this Bachelor Thesis, the **shared-memory model** will be the primary platform for designing and analyzing parallel algorithms.

4.2 Why is there development for parallel algorithms?

Over the years, the advancements in technology has doubled the regular CPU clock-speed, enabling algorithms to be computed faster. However the CPU has reached a maximum clock rate of 3Ghz in 2002. This was achieved by Intel with a Pentium 4 processor. Following then, manufacturers have turned to developing CPUs with more cores instead of faster cores. Because of the switch, computation has changed from sequential opreations to parallel operations. The parallelism in an algorithm has proved to be able to yield improved performance on various computers. An example would be on a parallel computer where operations in a parallel algorithm can be executed

simultaneously by different processors.[11]
Parallel algorithms opens up the possibility of assigning more resources to a task and shortening the process of completion. In return, this will save time and money. Parallel algorithms also have the ability to solve larger and more complex problems, of which is impractical and impossible to solve on a single computer due to the limited amount of memory available.

4.3 Parallel Computing

Hitherto, software has been written for serial computation all along. The program is executed on a single computer with a single Central Processing Unit (CPU), and is broken into a discrete series of instructions. Thereafter, instructions are executed one after another, however only one instruction may be executed at any point in time.
Parallel Computing is the simultaneous use of multiple compute resources to solve a computation problem. This allows the program to be executed using multiple CPUs. This is achieved by breaking the problem down into discrete parts which can be solved concurrently. Thus, allowing the instructions from each part to be simultaneously executed on different CPUs. [12]
Parallel computing is an evolution of serial computing that attempts to emmulate what has always been the state of affairs in the natural world: many complex, interrelated events happening at the same time, yet within a certain sequence. Some examples include: meteorological patterns, rush hour traffic in cities, automobile assembly line, and daily operations within a business. The goal of parallel processing it to perform computations faster than a single processor. This is achieved by implementing a number of processors concurrently to execute the task.[13]

4.3.1 Resources of Parallel Computing

The compute resources includes a single computer with multiple processors, or computers connected by a network. A ***parallel computer*** is simply a collection of processors, typically of the same type, interconnected in a certain fashion to allow the coordination of their activities and the exchange of data.[12]

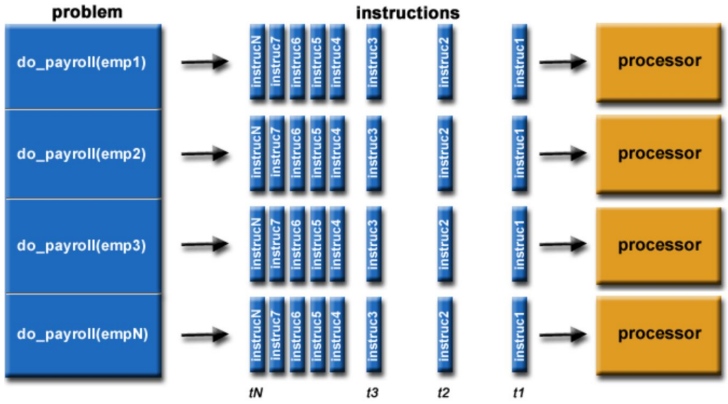


Figure 9: Parallel Computing Model

4.4 Measures used for evaluating the performance of a Parallel Algorithm

Given ***P*** to be a computational problem and ***n*** to be its input size. The sequential complexity of ***P*** is *T*(n)*. There is a sequential algorithm that solves ***P*** within this time bound, and also proves that no sequential algorithm can solve ***P*** faster. Let ***A*** be a parallel algorithm that solves ***P*** in time $T_p(n)$ on a parallel computer with p processors. Then the **Speedup** achieved by ***A*** is defined to be

$$S_p(n) = \frac{T^*(n)}{T_p(n)}$$

$S_p(n)$ represents the speedup factor obtained by algorithm A when p processors are available. Ideally since $S_p(n) \leq p$, we aim to design algorithms that achieve $S_p(n) \approx p$. However, certain factors attribute to a different calculation. This comprises of insufficient concurrency in the computation, delays introduced by communication, and **overhead** incurred in synchronizing the activities of various processors and in controlling the system. We have to take note that $T_1(n)$, the running time of the parallel algorithm A when the number p of processors is equal to 1, is not necessarily the same as *T*(n)*. Therefore the speedup is mesaured relative to the *best possible sequential algorithm*. It is common practice to replace *T*(n)* by the time bound of the best *known* sequential algorithm whenever the complexity of the problem is not known.

A different performance measure of the parallel algorithm A would be the

calcuation of its **efficiency**, defined by

$$E_p(n) = \frac{T_1(n)}{pT_p(n)}$$

.

This measure provides an indication of the effective utilization of the p processors relative to the given algorithm. A value of $E_p(n) \approx 1$, for some p, indicates that algorithm A runs approximately p times of the processors is doing "useful work" during each time step relative to theh total amount of work required by algorithm A.

There exists a limiting bound on the running time, denoted by T_∞(n), beyond which the algorithm cannot run any faster, no matter what the number of processors. Hence, T_p(n) $\geq T_\infty$(n) for any value of p, and therefore the efficiency E_p(n)$\leq \frac{T_1(n)}{pT_\infty\ (n)}$

Therefore, the efficiency of an algorithm degrades quickly as p grows beyond $\frac{T_1(n)}{T_\infty\ (n)}$. [10]

4.5 Background

Algorithms are often expressed in a high-level language and each algorithm begins with a description of its input and output. Subsequently a statement which consists of one or more sequences are added. The following statements below are most commonly used developing parallel algorithms. [10]

Algorithm 1: Assignment statement

```
1 begin
2 |  variable:= expression
3 end
```

The expression on the right of Algorithm 1 is evaluated and assigned to the variable on the left.

Algorithm 2: Begin / End Statement

```
1 begin
2 |  statement
3 |  statement
4 |  .
5 |  .
6 |  .
7 |  statement
8 end
```

Algorithm 2 defines a sequence of statements that must be executed in the order in which they appear.

Algorithm 3: Conditional Statement

```
1 begin
2 |  if (condition) then
3 |  |  statement
4 |  else
5 |  |  statement
6 |  end
7 end
```

In algorithm 3, the **if** condition is evaluated, and the statement following **then** is executed if the value of the condition is **true**. The **else** part is optional; it is executed if the condition is **false**. In the case of nested conditional statements, we use braces to indicate the **if** statement associated with each **else** statement.

Algorithm 4: Loops

```
1 begin
2 |  for (variable 1 to variable 2) do
3 |  |  statement
4 |  end
5 end
```

Algorithm 4 demonstrates a **for** loop being executed: If the initial value is less than or equal to the final value, the statement **do** will be executed, and the value of the variable is incremented by one. Otherwise, the execution of the loop terminates. The same process is repeated with the new value of the variable, until that value exceeds the final value, in which case the execution of the loop terminates.

Algorithm 5: Exit Statement

```
1 begin
2 end
```

Algorithm 5 explains an exit statement which causes the whole algorithm to terminate.

The time and space bounds on the resources required by a sequential algorithm are measured as a function of the **input size** which reflects the amount of data to be processed. In our context, we are most interested in the **worst-case** analysis of algorithms. Therefore, given an input size n, each resource bound will represent the maximum amount of the particular resource it required by any instance of size n. These bounds are expressed asymptotically using the following standard notation.

$T(n) = O(f(n))$ if there exists positive constants c and n_0 such that $T(n) \leq cf(n)$, for all $n \geq n_0$

$T(n) = \Omega(f(n))$ if there exists positive constants c and n_0 such that $T(n) \geq cf(n)$, for all $n \geq n_0$

$T(n) = O(f(n))$ if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

The **running time** of a sequential algorithm is calculated by estimating the number of *basic* operations required by the algorithm as a function of the input size. In general, a unit time is calculated for each operating reading from and writing into the memory, arithmetic operations and logic operations. The logic operations includes operations such as adding, subtracting, comparing and multiplying two numbers. The model most suitable for our purpose in this thesis will be the **Random Access Memory (RAM)** and **Parallel Random Access Memory (PRAM)**. These memories assumes the presence of a central processing unit with access to a random-access mmory. Subsequently, these memories handle the input and output operations. [10]

5 Models of Parallel Computation

The RAM model has been used successfully to predict the performance of sequential algorithms. Modeling parallel computaiton is much more challenging given the new dimension introduced by the presence of many interconnected processors. We are primarily interested in *algorithmic models that can be used as general frameworks describing and analyzing parallel algorithms.* Ideally, we would like our model to satisfy the following (conflicting) requirements:

1. **Simplicity:** The model shoud be simple enough to describe parallel algorithms easily, and to analyze mathematically important performances measures such as speed, communication, and memory utilization. In addition, the model should not be tied to any particular class of architectures, and hence should be as hard-ware independent as possible.

2. **Implementability:** The parallel algorithms developed for the model should be easily implementatble on parallel computers. In addition, the analysis performed should capture in a significant way the actual performance of these algorithms on parallel computers.

There are 3 commonly used parallel models. They are: **parallel comparison trees, sorting networks and Boolean Circuits.**

In sequential algorithms, the RAM model is used to successfully predict the performance. In order to model parallel computation, there is a new set of challenges with the introduction of many interconnected processors. To develop an ideal model, we aim to keep the model simple and great in implementation. [10]

5.1 Models of Computation

In both sequential and parallel computers, Parallel Computing operates by executing a set of instructions called algorithms. These set of instructions or algorithms instructs the computer about what it has to execute in each step. Depending on the instruction and data stream, computers can be classified into four categories:

1. Single Instruction Stream, Single Data stream (SISD)

2. Multiple Instruction Stream, Single Data stream (MISD)

3. Single Instruction Stream, Multiple Data stream (SIMD)

4. Multiple Instruction Stream, Multiple Data stream (MIMD)

5.1.1 SISD Computers

An SISD computer consists of a single processing unit receiving a single stream of instructions from the control unit and operates it on a single stream of data from the memory unit as shown in the figure below. At each step, the processor receives only one instruction from the control unit and operates on a single data from the memory unit. [14]

Figure 10: SISD

5.1.2 MISD Computers

In an MISD Computer, it contains multiple control units, multiple processing units, and one common memory unit. This would mean that there are n streams of instructions and one stream of data. Each of the processor here has its own control unit and share a common memory unit. All the processors obtain instructions individually from their own control unit and operate on a single stream of data as per the instructions they have received from their respective control unit. Therefore, parallelism is achieved by allowing the processors to do different things at the same time. [14]

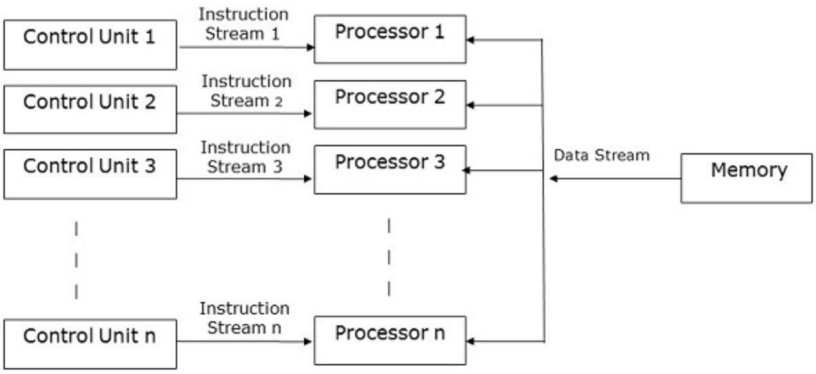


Figure 11: MISD

5.1.3 SIMD Computers

For the SIMD computer, it is a parallel computer which consists of one control unit, n identical multiple processing units, and shared memory. All the processors in this computer operate under the control of a single instruction stream issued by a central control unit. The processors in this computer operate synchronously. At each step, all the processors execute the same instruction, while the remaining processors wait for the next set of instructions. This class of computer can be further described as the Parallel Random-Access Machine (PRAM) which will be elaborated in further detail in the subsequent chapter. [14]

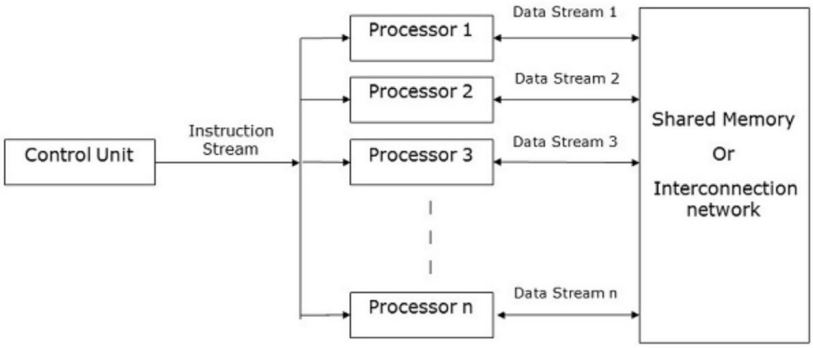


Figure 12: SIMD

5.1.4 MIMD Computers

The MIMD Computers have multiple control units, multiple processing units, a shared memory or interconnection network. In this class of computers, it has its own control unit, local memory unit, and arithmetic and logic unit. Each of these units receives a different set of instructions from their respective control units and operates on different sets of data.

5.2 The Shared - Memory Model

In a shared-memory model, many processors have access to a single shared memory unit. The shared memory model consists of a number of processors with individual local memory and it has the ability to execute its individual local program. All the processes in the shared memory model exchange data through communication with a shared memory unit. The individual processor has to be identified with a ***processor number*** or ***id***.

There are 2 basic modes of operation in a shared memory model.. Firstly, **synchronous** mode. In this mode, all the processors operate synchronously under the control of a common clock. This synchronous shared-memory model is known as **parallel random-access machine (PRAM)** model. Secondly, **asynchronous** mode. The processors operate under a separate clock in this mode. The programmer has to ensure the correct values are obtained, since the value of a shared variable is deteremined dynamicallly during the execution of the programs of the different processors.

In the following explanation, the shared memory model is a **multiple instruction multiple data (MIMD) type**. This signifies that the processor willl be able to execute instructions and data which are different from those being executed or operated by any other processor during any given time unit. For a given algorithm, the size of data transferred between the shared memory and local memories of the different processors represents the amount of **communication** required by the algorithm.[10]

The following algorithmic language is used for the 2 algorithms below.

$$\textbf{global read}(X, Y)$$
$$\textbf{global write}(U, V)$$

The effect of **global read** instruction is to move the block of data X stored in the shared memory into the local variable Y.
The effect of **global write** is to write the local data U into the shared variable V.

In the following example, we are demonstrating a Matrix Vector Multiplication on the Shared- Memory Model.

Let A be an $n \times n$ matrix, and let x be a vector of order n, both stored in the shared memory. Assume that we have $p \leq n$ processors such that $r{=}\frac{n}{p}$ is an integer and that mode of operation is *asynchronous*. Let A be partitioned as follows:

$$A = \begin{bmatrix} A_1 \\ A_2 \\ : \\ A_p \end{bmatrix}$$

where each block A_i is of size $r \times n$. The problem of computing the product $y{=}Ax$ can be solved in the following: Each processor P_i reads A_i

and x from the shared memory, and performs the computation $z=A_x$, and finally stores the r components of z in the appropriate components of the shared variable y

In algorithm 6, the notation *A(l:u, s:t* will be used to denote the submatrix of A consisting of rows *l, l+1, ...,t*. The same notation can be used to indicate a subvector of a given vector. Each processor executes the same algorithm. [10]

Input: An $n \times n$ *matrix A and a vector x of order n residing in the shared memory. The initialzed local variables are (1) the order n, (2) the processor number i, and (3) the number* $p \leq n$ *of processors such that* $r = n/p$ *is an integer.*

Output: The components $(i-1)r + 1$ ..... ir of the vector $y = Ax$ stored in the shared variable y.

Algorithm 6: Matrix Vector Multiplication on the Shared-Memory Model

```
1 begin
2 |  global read(x,z)
3 |  global read(A((i-1)r+1:ir,1:n),B)
4 |  Compute w = Bz
5 |  global write(w, y((i − 1)r + 1 : ir))
6 end
```

In step 2, a *concurrent read* of the same shared variable x is required by all the processors. At this instance, there are no more than 1 processor attempting to write into the same location of the shared memory.

Steps 3 & 4 transfers $O(\frac{n^2}{p})$ numbers from the shared memory into each processor.

Step 4 is the only computation step that requires $O(\frac{n^2}{p})$ arithmetic operations.

Step 5 stores $(\frac{n}{p})$ numbers from each local memory into the shared memory.

Algorithm 6 has an important feature whereby processors do not need to synchronise the activities due to the way the matrix vector product was partitioned. It is also possible to deisgn a parallel algorithm based on partitioning A and x into p blocks such that $A = (A_1, A_2, \ldots A_p)$ and $x = (x_1, x_2, \ldots x_p)$, where each A_i is of size $n \times r$ each x_i is of size r. The product $y = Ax$ is now given by $y=A_1x_1,A_2x_2, \ldots,A_rx_r$. Hence, the processor P_i can compute $z_i = A_i\ x_i$ after reading A_i and x_i from the shared memory, for $1 \leq i \leq p$. At this point, no processor should begin the computation of sum $z_1 + z_2 + \ldots +$

z_r before ensuring that all the processors have completed their matrix vector products. Therefore, an explicit synchronization primitive must be placed in each processor's program after the computation of $z_i = A_i x_i$ to force all the processors to synchronise before continuing the execution of their programs. [10]

6 The Parallel Random Access Machine (PRAM)

The PRAM is considered as one of the most straightforward method to model a parallel computer. The PRAM consists of a number of sequential processors which has the ability to access a global shared memory asynchronously. It is developed as an extension of the RAM model used in sequential algorithm analysis. The individual processors will be able to access its shared memory quickly and efficiently, similar to accessing its local memory.
The main advantages of the PRAM is its simplicity. It captures parallelism and abstracting away communication and synchronisation issues related to parallel computing.
The ability of abstraction offered by the PRAM is a fully synchronous collection of processors as well as a shared memory which makes it popular for parallel algorithm design. This would mean that the PRAM would store all data in the shared memory. However, the abstraction may be viewed as unrealistic from a practical point of view as full synchronisation using the PRAM icurs a very high cost in parallel machines currently in use.[15]

A PRAM consists of the following:

1. A set of similar type of processors
2. All the processors share a common memory unit. Processors has the ability to communicate themselves through a shared memory.
3. A memory access unit (MAU) connects the processor with the single shared memory.

During the execution of of a parallel algorithm, n number of processors gain access to the shared memory for reading input data, for reading or writing immediate results, and for writing the final results. The basic model of a PRAM allows all processors to gain access to the shared memory simultaneously if the memory locations they are trying to acccess or write is different. The SIMD can be further elaborated into 4 variants as described in the next section.

6.1 Variants of PRAM

There are 4 variants of PRAM.

1. **EREW** - exclusive read exclusive write does not allow concurrent access of the shared memory. If there are two or more processors attempting to read from or write into the same memory cell currently,the behaviour is undefined

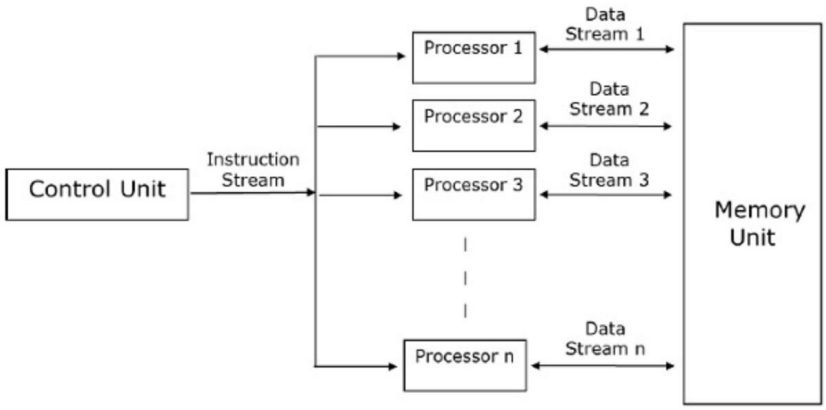


Figure 13: Basic Architecture of PRAM

2. **ERCW** - exclusive read concurrent write does not allow concurrent access of the reading the shared memory. However, it allows concurrent writing into the memory cell.

3. **CREW**- Concurrent Read Exclusive Write allows reading into the same memory cell but does not allow two concurrent write into the shared memory which will lead into an undefined behaviour

4. **CRCW**- Concurrent Read Concurrent Write allows both reading and writing into the memory cell concurrently. However, there are various rules that arbitrate how concurrent writes are handled.

    - in the *arbitary* PRAM, if there are multiple processors writing into a single shared memory cell, only one processor succeeds in writing into this cell.
    - in the *common* PRAM, all the processors must write the same value into a single shared memory cell, only then the arbitrary processor willl succed in writing into this cell.
    - in the *priority* PRAM, the processor with the highest priority will succeed in writing.

Among the PRAM variants, the weakest PRAM is the EREW. A CREW PRAM willl be able to simulate an EREW, while both the CREW and EREW can be simulated by the more powerful CRCW PRAM. Therefore, the algorithm can be deisgned for the common PRAM, executed on a priority or arbitary PRAM while exhbiting similar complexity. [15]

Algorithms developed for PRAM model have been of type **SIMD**. That is, all processors execute the same program such that, during each time unit,

alll the active processors are executing the same instruction, but with different data in general. However, we are still able to load different programs into the local memories of the processors, as long as the processors can operate synchronously; therefore, different type of instructions can be executed within the unit time allocated for a step.

6.2 Examples of Parallel Algorithms

6.2.1 Sum on the PRAM

The following example below gives an example of a sum of the PRAM model. Given an array A of $n = 2^k$ numbers, and a PRAM with n processors P_1, P_2, ..., P_n, we wish to compute the sum $S = A(1) + A(2) + ... + A(n)$. Each processor executes the same algorithm, given here for processor P_i. [10]

Input: *An array A of order $n = 2^k$ stored in the shared memory of a PRAM with n processors. The initialized local variables are n and the processor number i.*
Output: *The sum of the entries of A stored in the shared location S. The array A holds its initial value.*

Algorithm 7: Sum on the PRAM

```
1  begin
2  |  global read A(i),a
3  |  global write ((a,B(i))
4  |  for h = 1 to log(n) do
5  |  |  if (i ≤ n/2^h) then
6  |  |  |  begin
7  |  |  |  |  global read (B(2i − 1), x)
8  |  |  |  |  global read (B(2i), y)
9  |  |  |  |  Set z := x + y
10 |  |  |  |  global write (z, B(i))
11 |  |  |  end
12 |  |  end
13 |  end
14 |  if i = 1 then
15 |  |  global write(z, S)
16 |  end
17 end
```

To simplify the presentation of PRAM algorithms in Algorithm 7, we omit the details concerning the memory access-operations. An instruction

of the form $SetA := B + C$, where A, B, C are shared variables, should be interpreted as following sequence of instructions.

1. **global read** (B,x)
2. **global read** (C,y)
3. *Set* $z := x + y$
4. **global read** (z,A)

6.2.2 Matrix Multiplication of the PRAM

In the next example, an explanation of an algorithm for matrix multiplication on the PRAM will be introduced. [10]
Consider the problem of computing the product C of the two $n \times n$ matrices A and B, where $n = 2^k$, for some integer k. Suppose that we have n^3 processors available on our PRAM, denoted by $P_{i,j,l}$ where $1 \leq i,\ j,\ l \leq n$. Processor $P_{i,1,l}$ computes the product $A(i,l)B(l,j)$ in a single step. Then, for each pair (i,j), the n processors $P_{i,j,l}$, where $1 \leq l \leq n$, compute the sum $\sum_{l=1}^{\infty} A(i,l)B(l,j)$ as the example shown in Algorithm 7 earlier. [10]

Input: *Two $n \times n$ matrices A and B stored in the shared memory, where $n = 2^k$. The initialised local variables are n, and thhe triple of indices (i,j,l) identifying the processor.*
Output: *The product $C = AB$ stored in the shared memory.*

Algorithm 8: Matrix Multiplication on the PRAM

```
1  begin
2  |  Compute C' (i,j,l) = A(i,l)B(l,j)
3  |  for h = 1 to log(n) do
4  |  |  if (i ≤ n/2^h) then
5  |  |  |  set (C'(i,j,l) := C'(i,j,2l − 1) + C'(i,j,2l))
6  |  |  end
7  |  end
8  |  if l = 1 then
9  |  |  setC(i,j) := C'(i,j,1)
10 |  end
11 end
```

We have observed that Algorithm 8 requires concurrent read capabilities and that different processors may have access to the same data while executing step 2. For example, processors $P_{i,1,l}$, $P_{i,2,l}$, ... $P_{i,n,l}$, all require $A(i,l)$

from the shared memory when step 2 is executed. Thus, this algorithm runs on the CREW PRAM model. As for the running time, the algorithm take $O(log(n))$ time.

However, a modification of step 4 in Algorithm 8 can be executed by removing the **if** condition. This means (removing the whole statement $C'(i,j) := C'(i,j,1)$, then the corresponding algorithm requires a concurrent write of the same value capability. In fact, processors $P_{i,j,l}$, $P_{i,j,2}$, ... $P_{i,j,n}$ all attempt to write the value $C'(i,j,1)$ into location $C(i,j)$.

7 Analysing Parallel Algorithms

7.1 Parallel Complexity Theory

There are many appproaches to determine the running time and complexity of an algorithm. It depends on the type of statements used. [5]

7.1.1 Sequence of Statements

```
1 begin
2 |  statement 1
3 |  statement 2
4 |  .
5 |  .
6 |  statement k
7 end
```

The total time for executing the algorithm will be the addition of the time taken for all statements.

$$Total\,Time = (time\ for\ statement\ 1) + (time\ for\ statement\ 2) + (time\ for\ statement\ k)$$

If each of the statement only involves basic computations, then the time for each statement is constant and the time taken will be $O(1)$.

7.1.2 If - Then - Else

```
1 if (cond) then
2 |  block 1 (sequence of statements)
3 else
4 |  block 2 (sequence of statements)
5 end
```

In the above algorithm only either block 1 or block 2 will be executed. Thus, the worst-case time is the slower of the two possibilities.

$$max(time(block1, (time(block2))$$

If block 1 requires a time of $O(1)$ and block 2 requires a time of $O(N)$, therefore the if-then-else statement have a complexity of $O(N)$.

7.1.3 Loops

Algorithm 9: Single Loop Algorithm

```
1 for l in 1 .. N loop do
2 |  sequence of statements
3 end
```

The loop in the above algorithm executes **N** times, therefore the sequence of the statements also executes N times. If we assume the statements are $O(1)$, the total time for the loop is $N \times O(1)$, which is $O(N)$ overall.

7.1.4 Nested Loops

Algorithm 10: "For" Loop Algorithm

```
1 for l in 1 .. N loop do
2 |  for j in 1 .. M loop do
3 |  |  sequence of statements
4 |  end
5 end
```

In the above algorithm, the outer loop executes N number of times. Every time the outer loop executes, the inner loop willl execute M number of times. As a result, the statements in the inner loop execute a total of $N \times M$ times. Therefore, the complexity will be O($N \times M$). In the event of a special case, where the stopping condition of the inner loop is $J < N$ instead of $J < M$, the complexity for the two loops will be O(N^2).

7.2 Statements with function or procedure calls

In the event where a statement involves a function or procedure call, the complexity of the statement includes the complexity of the function of procedure. Assuming that the function or procedure f takes constant time, and the other function or procedure g takes a linear time proportional to a parameter k. Then the statements below will have the time complexities indicated

$$f(k) has O(1)$$

$$g(k) has O(k)$$

7.3 Amdahl's Law

7.3.1 History of Amdahl's law

Amdahl's Law is named after Gene Amdahl, the chief architect of IBM's first mainframe series and founder of Amdahl Corporation. Amdah discovered that there were some fairly stringent restrictions on how much of a speedup one could get for a given parallelized task. To state the fundamental limitation of parallel computing, Amdahl then formulated Amdahl's Law. Amdahl is most well known for formulating Amdahl's law uncovering the limits of parallel computing.

7.3.2 Introduction to Amdahl's law

Amdahl's law is a formula which gives the theoretical speed up of the program with some instructions so as to obtain the overall speed of the program Amdahl's law defines the *speedup* which can be gained by using a particular feature

$$Speedup = \frac{Performance\ for\ entire\ task\ using\ the\ enhancement\ when\ possible}{Performance\ for\ entire\ task\ without\ using\ the\ enhancement}$$

Speedup tells us how much faster a task can be executed using the computer with enhanced features as compared to the original computer.

Amdahl's Law allows us to predict and obtain the speedup from the enhancements which depends on two factors:

1. The fraction of the computation time in the original computer that can be converted to take advantage of the enhancement

2. The improvement

$$Speedup = \frac{1}{(1 - FRAC_{enhanced} + e)}$$

If some fraction $0 < f < 1$ of a computation **must be executed sequentially**, then the **speed up** which can be obtained when the rest of the computation is executed in parallel, is bounded above $\frac{1}{f}$ irrespective of p

7.3.3 Example of Amdahl's Law

Let T_1 denote the computation time on a sequential system, and we can split the total time as follows:

$$T_1 = t_s + t_p$$

where
t_s is the computation time needed for the sequential part
t_p is the computation time needed for the parallel part
Clearly, if we parallelize the problem, only t_p can be reduced. Assuming ideal parallelization, we get

$$T_p = t_s + \frac{t_s}{p}$$

where p is the number of processors Therefore, we are able to get the speedup of

$$S = \frac{T_1}{T_p} = \frac{t_s + t_p}{t_s + \frac{t_p}{p}}$$

Let f denote the sequential portion of the computation, i.e

$$f = \frac{t_s}{t_s + t_p}$$

Therefore, the speedup formula can be simplified into

$$S = \frac{1}{f + \frac{1-f}{p}}$$

less than $\frac{1}{f}$. [16]

We have to note that Amdahl's law can only be applied to cases where the problem size is fixed.

8 Representing the 2 Algorithms from pFaces

In this chapter, a representation of the parallel construction of Abstractions and Synthesis in *pFaces* will be presented. Firstly, *pFaces* flattens each space into a single-dimensional space, denoted by flattened space, where each element is in this flattened space can be mapped back to the higher-dimensional one. Next, the constroller synthesis sub-tasks follow the abstraction sub-tasks. The synthesis algorithm will then read the data already present in the abstraction memory and quantize the over approximation of the Hyper-rectangle in (x'_l and x'_h). The bounds are quantized with the same quantization parameters used for the state space.

8.1 Abstractions Algorithm

For the Abstractions Algorithm, the sub-task executes the processing element given a tuple (k, j). The actual vector in higher dimension will then be reconstructed, i.e (x_k, u_j). The sub-task willl then compute the reachable state from x_k with the input u_j and save only the higher and lower bounds (x'_l and x'_h),

In Algorithm 11, the simplified version and explanation of complexity for the Abstraction Algorithm is presented.

Algorithm 11: Abstractions Algorithm

Input: $flat\_xu, ssDim, isDim, f$
Output: XU_bags.

1 **begin**
2 **global_read***(flat_xu[i], (x_flat,u_flat))*
3 u = *flat_to_concrete(u_flat, isDim)*
4 x = *flat_to_concrete(x_flat,ssDim)*
5 (x_h, x_l)= $compute\_OARS(f, x, u)$
6 **global_write**(x_h,$XU\_bag[i].x\_h$)
7 **global_write**(x_l,$XU\_bag[i].x\_l$)
8 **global_write**($0, XU\_bag[i].flags$)
9 **end**

We are able to estimate the amount of computation and communication by the algorithm as follows:

In step 2, there is a concurrent read of the shared variable x_flat and u_flat, It transfers values from x_flat and u_flat into the shared memory *flat_xu*[i]. There is a transfer of O($\frac{|X||U|}{p}$) as $\mid flat\_xu \mid = \mid X \mid\mid U \mid$ in

the shared memory which represents the total size of x & u into the shared memory.

In relation to this algorithm, The input space is u, 2-dimension which consists of u_1 and u_2.The state space is x, 3-dimension, which consists of x, y and θ.

For steps 3 and 4, The complexity depends on the state space dimension and input space dimension. The operation of the conversion from the index to the value takes 3 loops for the input space u and 2 loops for the state space x. We now have 2 points, x which is 3 dimension and u in 2 dimension. Step 3 converts the index i to quantized vector x_k which compromises of 2 values, (u, $isDim$). As there is a for loop to calculate the values from flat to concrete. It obtains values from 0 to $isDim$, Thus, the complexity for step 3 will be $O(isDmin)$ and the complexity for step 4 is $O(ssDim)$.

Step 5 is the computation of overall approximation of reachable set using the ODE. This computation consists of 4 iterations in the Runge-Kutta method, with a complexity of O($4ssDim$), which is always dependant on the constant.

Step 6, 7 and 8 stores the values into XU_bags. which is similar to memory read as it is reading and writing the same data into the memory. Hence, the complexity isO($\frac{|X||U|}{p}$) due to the property: $O(f(n)) + O(g(n)) = O(f(n))$ if $O(f(n))$ & $O(g(n))$ has the same value.

Therefore the overall complexity for the abstract algorithm is O($\frac{|x\_h \times x\_l|}{p}$) assuming that ssDim & isDim is less than p where p is the number of processors.

8.2 Synthesis Algorithm

The synthesis algorithm follows the abstraction sub-task by reading (x'_l and x'_h) from the abstraction memory. It proceeds to quantize the over approximation of the hyper-rectangle in (x'_l and x'_h). Next, it checks if all post states have tuples (x, u) belong to the FP set. If it belongs to the FP set, the pair (x_k, u_j) will be considered as a good pair to be included in the next update of the FP set.

Algorithm 12: Synthesis Algorithm

Input: XU_bags , q ,x_h ,x_l, local $x\_flat = 0$, $localu\_flat = 0$,$local\_flags = 0$, $all\_post\_in\_z = true$, $my\_private\_xu\_behav\_post\_x\_flags = 0$

Output: XU_bags

```
1  begin
2  global_read(XU_bags[i].bag, bag)
3   flat_post_points = quantize_hyperrectangle (bag.x_h, bag.x_l , q)
4  for k ∈ flat_post_points do
5  |  global_read (XU_bags[k].flags,post_x_flags)
6  |  if (post_x_flags.isX_A) or !(post_x_flags.is_PiZ) then
7  |  |  all_posts_in_z = false
8  |  |  break
9  |  end
10 end
11 if (all_posts_in_Z) then
12 |  my_private_xu_behav_post_x_flags += isX_T
13 end
14 global_read (XU_bags.[xu_thread_idx].flags, xu_flags)
15 if (x_flags.is_PiZ(k)) and (!x_flags.isPiZ(k − 1)) or
     (!x_flags.isPiZ(k)) and (x_flags.isPiZ(k − 1)) then
16 |  if (xu_flags.is_Z(k)) then
17 |  |  xu_flags = isX_T
18 |  end
19 end
20 if (all_posts_in_Z) then
21 |  xu_flags+=is_Z(k)
22 else
23 |  xu_flags-=is_Z(k)
24 end
25 if !(xu_flags.is_Z(k − 1)) and (xu_flags.isZ(k)) and
     !(xu_flag.isX_A) then
26 |  my_private_xu_behav_flags+=is_Z(k)
27 else
28 |  my_private_xu_behav_flags-=is_Z(k)
29 end
```

```
30 if (my_private_xu_behav_flags! = ∅) then
31 |  atomic_or(x_flags[local_idx_x], my_private_xu_behav_flags)
32 end
33 if (!(xu_flags.isX_A)) then
34 |  global_write(XU_bag.[xu_thread_idx].flags, xu_flags)
35 end
36 [wait_for_all_other_processes][a]
37 if (local_idx_u == 0) then
38 |  global_read(x_flags[local_idx_x], my_private_xu_behav_flags)
39 |  if (isX_T ∈ my_private_xu_behav_flags) then
40 |  |  xu_flags+=is_PiZ
41 |  else
42 |  |  xu_flags-=is_PiZ
43 |  end
44 |  if (is_Z(k) ∈ my_private_xu_behav_flags) then
45 |  |  xu_flags+=isX_S
46 |  else
47 |  |  xu_flags-= isX_S
48 |  end
49 |  global write(XU_bag[x_thread_idx].flags, xu_flags)
50 end
```

[a] A processor i has number of subtasks $\frac{|X||U|}{p}$ and all sub-tasks need to be completed before continuing to step 37

For the Synthesis Algorithm, we are able to estimate the amount computation and communication as follows:

In Step 2, there is a concurrent read of *bag* from $XU\_bags[i].bag$. Therefore, the complexity for Step 2 is O($\frac{|X||U|}{p}$).

For steps 4 to 9, the complexity of the for loop for flat_post_points will be the number of posts, which can be bounded by O(| X |).
Step 6 is executed to check whether the is_Piflag has this post state of (x,:) in the Z set. The PiZ indicates that the x is in the projection of Z

In steps 11 and 12, if the condition ($all\_posts\_in\_Z$)is fulfilled, the complexity will be O(| X |), which is covered in Steps 4 to 9 .

In step 12, $isPiZ$ is declared and has to be set in $localx\_flags$.

In step 14, there is a concurrent read of the $XU\_bags.[xu\_thread\_idx]$ to xu_flags. Step 18 is setting our own (x, u) flags to indicate whether we are still in Z and reading our own xu_flags. Therefore, the complexity for this memory read would be $O(\frac{|X||U|}{p})$.

The if condition in steps 15 to 18 requires a complexity of $O(\frac{|X||U|}{p})$.

For steps 25 to 29, it has a if- then - $else$ condition, this means that either $xu\_flags{+}{=}is\_Z(k-1)$ or $xu\_flags{-}{=}is\_Z(k-1)$ is executed. Thus, the complexity willl be the worst-case time of the two possibilities, which is $O(\frac{|X||U|}{p})$.

In steps 33 and 34, if not $(xu\_flags.isX\_A)$, it will store values into XU_bag with a complexity of $O(\frac{|X||U|}{p})$.

In steps 36 to 50, the algorithm is waiting for the execution of all other processes. With a problem size being n and p being the number of processes and $p < n$, there are chunks of executions in this step, therefore it results in a complexity of $O(\frac{|X||U|}{p})$.

Summing up the complexity for the systhesis algorithms, the overall complexity is $O = \max\{\mid X \mid, \frac{|X||U|}{p}\}$.

References

[1] J. Tompson and K. Schlachter, "An introduction to the opencl programming model," *Person Education*, vol. 49, 2012.

[2] S. McIntosh-Smith. Opencl. [Online]. Available: https://www.nersc.gov/assets/pubs_presos/MattsonTutorialSC14.pdf

[3] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg, *OpenCL programming guide*. Pearson Education, 2011.

[4] L. Howes, "Opencl parallel computing for cpus and gpus," *Advanced Micro Devices (AMD) presentation, http://developer. amd. com/gpu_assets/OpenCL _Parallel_Computing_for_CPUs_and_GPUs_201003. pdf*, 2010.

[5] MIT, "Big o notation."

[6] P. Danziger, "Big o notation," *Source internet: http://www. scs. ryerson. ca/˜ mth110/Handouts/PD/bigO. pdf, Retrieve: April*, 2010.

[7] stack overflow. Big o complexity. [Online]. Available: https://stackoverflow.com/questions/487258/what-is-a-plain-english-explanation-of-big-o-notation

[8] E. Horowitz, S. Sahni, and S. Rajasekaran, *Computer algorithms C++: C++ and pseudocode versions*. Macmillan, 1997.

[9] Saylor.org. Breadth first search. [Online]. Available: https://www.saylor.org/site/wp-content/uploads/2012/06/CS408-2.3.2-BreadthFirstSearch.pdf

[10] J. JáJá, *An introduction to parallel algorithms*. Addison-Wesley Reading, 1992, vol. 17.

[11] P. P. Mattsson, "Why havent cpu clock speeds increased in the last few years?" *COMSOL blog*, 2014.

[12] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing: design and analysis of algorithms*. Benjamin/Cummings Redwood City, 1994, vol. 400.

[13] B. Barney *et al.*, "Introduction to parallel computing," *Lawrence Livermore National Laboratory*, vol. 6, no. 13, p. 10, 2010.

[14] T. Point. Parallel algorithm - introduction. [Online]. Available: https://www.tutorialspoint.com/parallel_algorithm/index.htm

[15] A. Gerbessiotis, "The pram model."

[16] J. Zapletal, "Amdahl's and gustafson's laws," *VSB-Technical University of Ostrava*, 2009.