IMPLEMENTATION AND PARALLELIZATION OF A SAT SOLVER ALGORITHM

BY

ABDULLAHI, MOHAMMED

DEPARTMENT OF MATHEMATICS AHMADU BELLO UNIVERSITY, ZARIA

NIGERIA

JUNE, 2011

IMPLEMENTATION AND PARALLELIZATION OF A SAT SOLVER ALGORITHM

BY

ABDULLAHI, MOHAMMED B.Tech(FUT MINNA 2006)

M.Sc/SCIE/00720/08-09

A THESIS SUBMITTED TO THE

POSTGRADUATE SCHOOL,

AHMADU BELLO UNIVERSITY, ZARIA

NIGERIA

IN PARTIAL FULFILMENT FOR THE AWARD OF MASTER OF SCIENCE IN COMPUTER SCIENCE

DEPARTMENT OF MATHEMATICS

AHMADU BELLO UNIVERSITY, ZARIA

NIGERIA

JUNE, 2011

DECLARATION

I declare that the work in the thesis entitled 'IMPLEMENTATION AND PARALLELIZATION OF A SAT SOLVER ALGORITHM' has been performed by me in the Department of Mathematics under the supervision of Prof. S. B. Junaidu. The information derived from the literature has been duly acknowledged in the text and a list of references provided. No part of this thesis was previously presented for another degree or diploma at any university.

ABDULLAHI, MOHAMMED

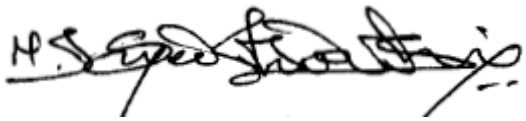
Name of student	Signature	Date

CERTIFICATION

This thesis entitled 'IMPLEMENTATION AND PARALLELIZATION OF A SAT SOLVER ALGORITHM' by ABDULLAHI, MOHAMMED meets the regulations governing the award of the degree of Master of Science of Ahmadu Bello University, Zaria, and is approved for its contribution to knowledge and literary presentation.

---------- Date----------

Prof. S. B. Junaidu
Major Supervisor



---------- Date----------

Dr. Seyed M. I. Buhari
Minor Supervisor

---------- Date----------

Dr. A. A. Tijjani
Head of Department

---------- Date----------

Prof. A. A. Joshua
Dean, Postgraduate School

DEDICATION

I would like to thank my parents, brothers and sisters for their constant love and support. It would be impossible for me to express my gratitude towards them in mere words. I dedicate this thesis to them.

ACKNOWLEDGEMENT

Most importantly, I owe all what I have been obtaining to Allah, who has always been shining my life. Now I can only reaffirm the following words – "*Allah, whatever grace we have, it is from you. No god but you"*.

I would like to thank my supervisor Prof. S. B. Junaidu for his guidance and insight he provided throughout this thesis work, and in teaching the course unit on High Performance Computing which formed the basis of this work. His dedication to research and patience for students and colleagues will always be an example for me to follow.

I would like to thank my minor supervisor, Dr. Seyed M. I. Buhari for his support and encouragement on this thesis.

I will also like to thank all the staff of Mathematics Department for their various contributions towards making this course a success.

To my fellow academics and friends, for helping me to make this dream come true.

Thanks also to my parents who have been extremely understanding and supportive of my studies. I feel very lucky to have parents that share my enthusiasm for academic pursuits. Thank you to my parents for bringing me up and supporting my decisions all the way along. Thanks to my brothers and sisters, who are my best friends.

Thanks to my wife and daughter, Amina and Asiyah for their unquantifiable encouragement over the years.

This dissertation has been completed with support from the ABU MacArthur Foundation Project.

ABSTRACT

There have been remarkable progresses in propositional satisfiability (SAT) in recent years, with significant theoretical and practical contributions. The problem of determining whether a propositional Boolean formula can be true is called the Boolean Satisfiability Problem or SAT. In practice, SAT is a core problem in many applications such as Electronic Design Automation (EDA) and Artificial Intelligence (AI). In this thesis, an implementation of a SAT solver algorithm proposed by Bagais (2008) using C programming language is presented. The performance of the sequential SAT solver is independent of the nature of Boolean formula for efficiency. The algorithm belongs to the class of complete search procedures which determine whether the given formula is satisfiable or not. It keeps all the satisfying assignments of previous clauses before moving forward thereby avoiding backtracking which consumes much time before a decision is reached and this is the case of other algorithms in the literature. A parallel version of the SAT solver algorithm which can be implemented on 2-Dimentional mesh architecture was proposed and a theoretical analysis of the proposed algorithm was carried out.

Table of Contents

DECLARATION iv
CERTIFICATION v
DEDICATION vi
ACKNOWLEDGEMENT vii
ABSTRACT vii
LIST OF TABLES xi
LIST OF FIGURES xii
CHAPTER ONE: INTRODUCTION 1
1.0 What is SAT? 1
1.1 Basics on SAT 1
1.2 Why Parallel Programming? 3
1.3 Complexity 3
1.4 Motivation 6
1.5 Thesis Objectives 7
1.6 Research Methodology 8
1.7 Contributions 8
1.8 Thesis Structure 8
CHAPTER TWO: LITERATURE REVIEW 10
2.0 Introduction 10
2.1 Overview of Parallel Programming 10
2.1.1 Parallel Approaches 12
2.1.1.0 Threads 12
2.1.1.1 OpenMP 14
2.1.1.2 Message Parsing Interface (MPI) 18
2.2 Bagais' Algorithm 21
2.2.0 Time Complexity 29
2.2.1 Implementation 30
CHAPTER THREE: PARALLELISATION 30
3.0 Introduction 30
3.1 Parallel Process 30
3.2 Distribution of Literals 31
3.3 Communication Steps 31
3.4 An Illustration 35

3.5 The Proposed Parallel Algorithm....47
CHAPTER FOUR: THEORETICAL ANALYSIS....49
4.0 Introduction....49
4.1 Communication and Computation Cost Model....49
4.1.0 Row Shift Communication Cost....50
4.1.1 Column Shift Communication Cost and Formation of Assignment....50
4.2 Memory Cost Model....57
CHAPTER FIVE....58
5.0 Summary and Conclusion....58
5.1 Future Work....58
References....59
APPENDIX A....61

LIST OF TABLES

2.0: Processes vs. Threads **13**

2.1: Do-it-yourself Threads vs. OpenMP **17**

2.2: OpenMP vs. MPI **21**

3.0a: Computed Assignments after Clock Tick 1 ..... **37**

3.0b: Satisfying Assignments after Clock Tick 1..... **37**

3.1a: Computed Assignments after Clock Tick 2 ..... **39**

3.1b: Satisfying Assignments after Clock Tick 2..... **39**

3.2a: Computed Assignments after Clock Tick 3 ..... **40**

3.2b: Satisfying Assignments after Clock Tick 3..... **40**

3.3a: Computed Assignments after Clock Tick 4 ..... **42**

3.3b: Satisfying Assignments after Clock Tick 4..... **42**

3.4a: Computed Assignments after Clock Tick 5 ..... **44**

3.4b: Satisfying Assignments after Clock Tick 5..... **45**

LIST OF FIGURES

2.0: Execution model in OpenMP 15

2.1: Hello World in OpenMP 16

2.2: Hello World in MPI 19

2.3: Satisfying Assignments for the Clause $(y_1 \vee y_3 \vee \bar{y}_4)$ 23

2.4: Computing Satisfying Assignments from Current Assignment and Clause $(\bar{y}_1 \vee y_2 \vee \bar{y}_5)$ 23

2.5: Satisfying Assignments for Clauses $(y_1 \vee y_3 \vee \bar{y}_4)(\bar{y}_1 \vee y_2 \vee \bar{y}_5)$ 24

2.6: Computing Satisfying Assignments from Current Assignment and Clause, $(y_2 \vee \bar{y}_3 \vee y_4)$ 24

2.7: Satisfying Assignments for Clauses $(\bar{y}_1 \vee y_2 \vee \bar{y}_5)(y_2 \vee \bar{y}_3 \vee y_4)(y_1 \vee y_3 \vee \bar{y}_4)$ 25

2.8: Computing Satisfying Assignments from Current Assignment and Clause $(y_3 \vee y_5 \vee y_1)$ 26

2.9: Satisfying Assignments for the Boolean Formula. 27

2.10: Bagais' Algorithm 28

3.0: 3x3 2-D Mesh of Process Grid 31

3.1: Data Mapping onto Process Grid and Computation of Satisfying Assignment and Shifting Process at Clock Tick 1. 32

3.2: Computation of Satisfying Assignment and Shifting Process at Clock Tick 2. 33

3.3: Computation of Satisfying Assignment and Shifting Process at Clock Tick 3. 33

3.4: Computation of Satisfying Assignment and Shifting Process at Clock Tick 4. 34

3.5: Computation of Satisfying Assignment and Shifting Process at Clock Tick 5. 34

3.6: 3x3 2-D Mesh of Process Grid. 35

3.7: Data Mapping onto Process Grid 36

3.8: Computation of Satisfying Assignment and Shifting Process at Clock Tick 1. ..... **37**

3.9: Computation of Satisfying Assignment and Shifting Process at Clock Tick 2. ..... **38**

3.10: Computation of Satisfying Assignment and Shifting Process at Clock Tick 3.. .... **39**

3.11: Computation of Satisfying Assignment and Shifting Process at Clock Tick 4.. .... **41**

3.12: Computation of Satisfying Assignment and Shifting Process at Clock Tick 5.. .... **43**

3.13: Proposed Parallel Version of Bagais' Algorithm. **47**

4.0: Data Mapping of 3 Clauses Boolean Formula onto Process Grid. **49**

4.1: Data Mapping of 4 Clauses Boolean Formula onto Process Grid.. ..... **50**

4.2: Data Mapping of 5 Clauses Boolean Formula onto Process Grid... ..... **43**

4.3: Data Mapping of 6 Clauses Boolean Formula onto Process Grid.. ..... **47**

4.4: Data Mapping of 7 Clauses Boolean Formula onto Process Grid.. ..... **54**

CHAPTER ONE: INTRODUCTION

1.0 What is SAT?

The **propositional satisfiability problem (SAT)** is a decision problem of determining whether a given propositional formula is satisfiable, that is, whether there exists an assignment of truth-values to the propositional variables of the formula under which the formula evaluates to true according to the standard two-valued semantics (TRUE and FALSE). SAT is considered important in many disciplines, such as mathematical logic, electrical engineering, computer science, operations research etc. It was the first problem shown to be NP–complete (Cook, 1971), which means that it is considered unlikely that an algorithm exists that can solve any instance of SAT in polynomial time. SAT is well-known for having many different applications in various areas of computer science, including theoretical computer science, algorithmic, artificial intelligence like planning (Kautz and Selman, 1999), hardware design and electronic design automation (Smith *et al.*, 2004; Goldberg *et al.*, 2001; Larrabee, 1992; Wood and Rutenbar,1998), and software verification (Shlyakhter *et al.*, 2003). Other applications of SAT include scheduling (Crawford and Baker, 1997; Memik and Fallah, 2002; Zhang *et al.*, 2004) and so on.

1.1 Theoretical Background of SAT

Propositional variables are denoted $y_1, y_2, y_3, \ldots, y_n$, and can be assigned truth values *false* (also, *F* or 0) or *true* (also,*T* or 1). The truth value assigned to a variable y is denoted by $v(x)$.

A **literal** l is either a variable y_i or its negation: $\bar{y}_i$.

A **clause** is a disjunction ($\vee$) of literals. For example, y_i is a positive literal and $\bar{y}_i$ is a negative literal. A clause is said to be satisfied if at least one of its literals assumes value 1, unsatisfied if all of its literals assume value 0.

Conjunctive Normal Form (CNF) formula F is a conjunction ($\wedge$) of clauses. A formula is said to be satisfied if all its clauses are satisfied, and is unsatisfied if at least one clause is unsatisfied. The SAT problem is to decide whether there exists a truth assignment to the variables such that the formula becomes satisfied. It will often be simpler to refer to clauses as sets of literals, and to the CNF formula as a set of clauses. Hence, the notation $l \epsilon c$ indicates that a literal l is one of the literals of clause c, whereas the notation $c \epsilon F$ indicates that clause c is one of the clauses of CNF formula F.given the instance of SAT $F = (y_1 \vee y_2 \vee y_3) \wedge (\bar{y}_2 \vee y_4) \wedge (y_6 \vee \bar{y}_2)$. Setting the variables y_2, y_4, and y_6 to true, the formula is satisfied. But how do we decide this in general? Of course, we can always search through all truth assignments, one by one, but for formulae with n variables, the number of possible assignments is exponential, 2^n. A CNF formula in which the maximum clause length is equal to ℓ is referred to as an ℓ-CNF formula. In particular, 2-CNF formulas are solvable in linear time (Papadimitriou, 1991), while 3-CNF formulas are in general NP-complete (Cook, 1971).

If all clauses contain at most one positive literal, then the Boolean formula is called a **Horn formula**.

Subsumption: a clause is said to subsume another if the literals in one clause are a subset of those in another clause. Simply speaking, the smaller clause is more specific than the larger one. Any assignment of values to variables that satisfies the smaller clause also satisfies the larger one. So, the larger clause can be dropped without changing the set of solutions. For instance, consider the clauses c1 = $(y_1 \vee \bar{y}_2 \vee y_3)$ and

($y_1 \vee y_3$). In this case, clause c2 subsumes clause c1, and therefore clause c1 can be eliminated. Subsumption is of particular importance in resolution algorithms because resolution tends to produce large clauses. The subsumption helps keep the clause database small, and consequently help keep the search space small. The SAT problem is the following: given a Boolean formula in conjunctive normal form, either find a satisfying truth assignment or else report that none exists

1.2 Why Parallel Programming?

Many applications, especially scientific ones, today require a vast amount of computational power, e.g. weather predictions and particle simulations. Due to the slow processing power of uniprocessors in solving these problems there is need to utilize a larger amount of processors that work together in order to solve these problems. The rate by which performance is increasing for uniprocessors is also on a decline, largely because of increasing leakage currents, and increased power consumption due to the higher clock frequencies. This makes it highly unlikely that uniprocessors ever will be able to reach such speeds that are demanded. This is realized by chip manufacturers like Intel and AMD and most new processors developed by them today use dual cores, and even more cores are being projected. That is, parallel computing is needed today and in future, there is a need for parallel algorithms to easily utilize the power of parallel computers.

1.3 Complexity

Many of the games and puzzles people play are interesting because of their difficulty: it requires cleverness to solve them. Often this difficulty can be measured mathematically, in the form of complexity. The complexity of a process or algorithm is a measure of how difficult it is to perform.

The study of the complexity of algorithms, also known as complexity theory, deals with the resources required during computation to solve a given problem. The most common resources are time (how many steps does it take to solve a problem) and space (how much memory does it take to solve a problem). Other resources can also be considered, such as how many parallel processors are needed to solve a problem in parallel. The time complexity of a problem is the number of steps that it takes to solve an instance of the problem, as a function of the size of the input, using the most efficient algorithm. To further understand time complexity intuitively, consider the example of an n size instance that can be solved in n^2 steps. For this example we say that the problem has a time complexity of n^2. Of course, the exact number of computer instructions will depend on what machine or language is being used. To avoid this dependency problem, we generally use the Big O notation. If a problem has time complexity $O(n^2)$ on one typical computer, then it will also have complexity $O(n^2)$ on most other computers, so this notation allows us a generalization away from the details of a particular computer. In this case, we say that this problem has polynomial time complexity. This is also true for all problems having $O(n^x)$ time complexity, with x being a real constant. The space complexity of a problem defines how much memory it takes to solve a problem. For example, consider an n size instance that can be solved using 2n memory units. Hence, this problem has O(n) space complexity and therefore linear complexity. Much of complexity theory deals with decision problems. A decision problem is a problem where the answer is always YES/NO. For example, the problem IS-PRIME is: given an integer, return whether it is a prime number or not. Decision problems are often considered because an arbitrary problem can always be reduced to a decision problem. Decision problems fall into sets of comparable complexity, called complexity classes. P and NP are the most well-known complexity classes, meaning Polynomial time and

Nondeterministic Polynomial time, respectively. The complexity class P is the set of decision problems that can be solved by a deterministic machine with a number of steps bounded by a power of the problem's size. This class corresponds to an intuitive idea of problems which can be effectively solved even in the worst cases. The complexity class NP is the set of decision problems with a nondeterministic solution and with the number of steps to verify the solution being bounded by a power of the problem's size. In other words, all the problems in this class have the property that their solutions can be checked effectively in polynomial time. The complexity class Co-NP is the set of decision problems where the NO instances can be checked effectively in polynomial time. The Co in the name stands for complement. The class of P-problems is a subset of the class of NP-problems. The question of whether P is the same set as NP is the most important open question in theoretical computer science.

Observe that if P and NP are not equivalent, then finding a solution for NP-problems requires an exhaustive search in the worst case. The question of whether P = NP motivates the concepts of hard and complete.

A set of problems X is hard for a set of problems Y if every problem instance in Y can be transformed easily (i.e. in polynomial time) into some problem instance in X with the same answer. The most important hard set is NP-hard. A problem is said to be NP-hard if an algorithm for solving it can be translated into one for solving any other NP-problem.

In general, is easier to show that a problem is NP than to show that it is NP-hard. Set X is complete for Y if it is hard for Y, and is also a subset of Y. The most important complete set is NP-complete. A NP-complete problem is both NP (verifiable in nondeterministic polynomial time) and NP-hard (any other NP-problem can be

translated into this problem). SAT is an example of an NP-complete problem (Cook, 1971).

1.4 Motivation

Because of the practical importance of Boolean Satisfiability, significant research effort has been spent on developing efficient SAT solving procedures for practical purposes. The research has resulted in the development of many SAT algorithms that have seen practical success. All the methods that address this typical NP-Complete problem may be divided in two categories: Complete ones, usually based on the Davis, Logemann and Loveland procedure (Davis et al., 1962; Moskewicz *et al*., 2001), can prove that a formula is satisfiable or not. Incomplete solvers (also called one-sided solvers) only give an answer if the instance has a given property (generally if it is satisfiable). Powerful heuristics to solve SAT instances have been proposed (Zhang, 1997; Goldberg & Novikov 2002; Moskewicz *et al.* 2001) and new upper bounds have been proved but most of them require some kind of structure or property that must exist in the SAT formula to be correct and effective. That is because SAT is a data-dependent problem and to state that an algorithm is superior, some assumptions need to be provided. Bagais' proposed an algorithm that does not rely on the structure of the formula in hand, which will be considered an advantage with respect to other algorithms.

For many decades, Moore's law has bestowed a wealth of transistors that hardware designers and compiler writers have converted to usable performance, without changing the sequential programming interface. The main techniques for these performance benefits increased clock frequency and smarter but increasingly complex architectures are now hitting the so-called power wall. The computer industry has accepted that future performance increases must largely come from increasing the number of

processors (or cores) on a die, rather than making a single core go faster. This historic shift to multicore processors changes the programming interface by exposing parallelism to the programmer, after decades of sequential computing. Parallelism has been successfully used in many domains such as high performance computing (HPC), servers, graphics accelerators, and many embedded systems. The multicore inflection point, however, affects the entire market, particularly the client space, where parallelism has not been previously widespread.

There were a number of motivations for developing and applying parallel SAT algorithms to SAT problems. One was the particularly challenging nature of the problem. The challenges include the NP-completeness of finding the satisfiable instances of a propositional formula, and the enormous size of problems of current industrial interest. Also, as parallel computers move into mainstream computing, the challenge to researchers in all areas is to develop algorithms that can exploit the potential of these powerful machines. The model of Bagais' algorithm was discussed in this thesis has great potential for scaling to take advantage of larger and larger numbers of processor. Since we believe the algorithm maps well to parallel computers, it motivates us to see whetter this can help us to solve hard problems of practical importance.

1.5 Thesis Objectives

The main objective of this thesis to implementing a SAT solver algorithm and proposes a parallel version of the algorithm. Also, a parallel version of his algorithm was proposed which is implementable on a mesh topology. Thus, in the context of this thesis, a few research goals will be considered as follows:

a. Implementing the SAT solver algorithm using C programming language.

b. Developing the parallel version of the SAT solver algorithm.

c. Carrying out the performance analysis of the parallel algorithm

1.6 Research Methodology

The research methodology is described as follows. We implemented the SAT solver algorithm using C programming language.

Second, we proposed the parallel version of the SAT solver algorithm. The performance model for interprocess communication and memory overhead were defined for the proposed parallel version of the SAT solver algorithm.

1.7 Contributions

The main contribution of this thesis is the implementation of Bagais' algorithm and development of the parallel version. The parallel version of the algorithm can be implemented on 2-Dimentional grid topology.

1.8 Thesis Structure

This thesis has a total of five chapters. The first chapter is an introduction, and the last chapter describes conclusions and future work. The remaining chapters contain an overview and the main contributions of the thesis. After the Introduction, the second chapter overviews Bagais' algorithm and parallel programming. We give a special emphasis to the concepts related to Bagais' search algorithms, since most of our contributions are related to his algorithm. Afterwards, a detailed description of the algorithm is given. Chapter three describes the parallel version of the Bagais' algorithm. We start with analysing the data partitioning for the parallel processes. Afterwards, we describe the data distribution on 2-Dimentional mesh topology. Then, we describe the communication steps for the parallel algorithm. Next we gave an example to test the

correctness of the proposed algorithm. We finally capture the parallel process in an algorithm. Chapter three presents the theoretical analysis of the parallel algorithm. Chapter five gives the summary of the work and presents conclusion and future research work.

CHAPTER TWO: LITERATURE REVIEW

2.0 Introduction

This chapter provides an overview of parallel programming and the proposed SAT solving algorithm by Bagais. Section 2.1 gives an overview of parallel programming and introduces parallel programming languages. In Section 2.2 the details of solving SAT using Bagais' algorithm are outlined. The topics discussed include the procedures of the algorithm as well as ideas such as an example for illustration of the proposed procedure and time complexity of the algorithm.

2.1 Overview of Parallel Programming

Parallel programming refers to numerous programming methods aiming to take advantage of parallel computing resources provided by computers. Parallel programming is based on the simple idea that having more than one program execution simultaneously for the same problem yields faster results.

Parallel programming is not a new concept as it has been used in various areas for many years, most notably in high performance computing. Despite this, parallel programming methods have seen a surge of interest in the recent years. This popularity is largely attributed to the rise of multi-core CPU technology and the popularity of cheaper cluster computing environments made totally of commodity hardware (desktop PCs). Moore's Law (first introduced in (Gordon, 1965), revised later) is defined as the observation and prediction that number of transistors (or components, as used in the original paper) in an integrated circuit doubles approximately every two years. This has resulted in exponential growth in computer performance. Traditionally, processor speed-ups in line with the Moore's Law had been mostly due to frequency gains by packing more and more transistors. While the Moore's Law is still valid, processor development has taken

a new spin in the last few years. The old technique of increasing CPU frequency has recently been challenged because of various issues such as physical constraints (including the problem of packing up smaller microelectronics and heat dissipation problems), increased power consumption and the problem that memory lags behind processing power and becomes the bottleneck (the memory wall problem). The new trend in increasing processor performance is using the multi-core processor model wherein several processing cores are utilized in a single processor. Multi-core CPU model is claimed to address many deficiencies existent in the old model, and thus it is becoming widely used. Consumer processors now include quad-core CPUs with 4 processing units, while 8 core CPUs are becoming cheaper.

Another trend adding to the interests in parallel programming has been the use of commodity hardware in constructing cluster environments. Institutes and organizations are increasingly deploying high performance computing systems by making use of cheap commodity hardware. This has enabled growing deployment of computing clusters, especially at institutions which could not afford custom high-end servers and also broadened their use at those which could. At the peak of this trend seems to be Google, where thousands of computers are used in a wide range of jobs. It is claimed that computer clusters composed of commodity hardware "achieves superior performance at a fraction of the cost of a system built from fewer, but more expensive, high-end servers" (Barroso *et al*, 2003).

How the multi-core CPU development and the growing use of cluster computers did helped parallel programming? To take advantage of the multi-core processor technology and shared nothing cluster architectures to enable faster running programs, one needs to make use of parallel programming methodologies. Several methods exist in achieving parallel programming, each with varying success and properties. Some of the more

popular methods include parallel languages, automatic code generation techniques, implicitly parallel executing operating systems, POSIX Threads, message passing libraries and APIs for flexible thread management using compiler directives.

Before moving on to discuss parallel programming methods, definition of two important terms useful in the measurement of parallel programming will now be given.

Definition 2.0

Speedup of a parallel program presents how much faster it is than the corresponding serial version. It is defined by the following formula:

$$S = \frac{Execution\ \ time\ \ of\ \ the\ \ serial\ \ version}{Execution\ \ time\ \ of\ \ the\ \ parallel\ \ version} \qquad (2.1)$$

Speedup is useful for determining how fast the new implementation is, but it is also used to find efficiency, which also takes the number of processors into account. Efficiency measures how well the processors added for computations are used.

$$E = \frac{Speedup\ \ shown\ \ by\ \ the\ \ program}{Number\ \ of\ \ processors} \qquad (2.2)$$

2.1.1 Parallel Approaches

2.1.1.0 Threads

Threads are defined as the concurrent executions of different tasks of a given program, and they are one of the most widely used methods in parallel programming. They are related with the “process” notion, but they offer flexibility and efficiency compared with processes. Process and thread implementations may differ from an operating system to another or even within programming languages, but in this thesis both terms will be used with their typical properties.

There are several important differences between processes and threads. These differences can be summarized in Table 2.0. One important distinction between a

process and a thread is that a process has its own address space while a thread shares it with other threads from the same process. This results in easy access of variables by threads while processes have to go through the tedious and slow interprocess communication mechanisms provided by the OS. Thanks to the sharing of the address space and state, threads are usually associated with less overhead, and thus are more efficient than processes. Nevertheless, there is a trade-off. Sharing the address space introduces new types of bugs through deadlocks, race conditions and other interesting behaviour when concurrent access to the same data is desired. Using semaphore-like constructs, one can ensure the proper concurrent access of variables by threads. Processes have their own address space and are thus protected from these errors.

Table 2.0: Processes vs. Threads

Processes	Threads
Processes are independent units of execution	Threads are tied to a process
Processes have their own address space and state information	Threads share the same address space and state within the parent process
Processes interact through interprocess communication (IPC) methods provided by the OS (harder to do, with security considerations)	Threads can use more flexible methods, such as seamlessly reading memory locations since the address space is shared
Switching between processes is costly	Switching between threads is relatively Cheaper

Threads are the most used method for achieving concurrent execution. They are generally provided in a variety of programming languages so that programs can work on different tasks at once, and still be able to run efficiently. One popular usage of threads is in the user interface design. For example, several different user interface elements

(eg. windows or tabs) can be controlled by different threads as it is mainly done in Java. Threads in user interface programming provide increased responsiveness and it avoids the hang-up or freezing of a window while another one is busy.

A disadvantage of the threads model is that it is not portable. There are various Application Package Interfaces (APIs) for different platforms if one wants to use threads in parallel programming. The most popular ones are POSIX threads (pthreads) for unix operating systems and WinAPI threads for Microsoft Windows operating systems. Furthermore, it can be a challenging task if one does all thread management by himself, without using any libraries or tools tailored for this specific job. It could be complicated to devise how and when new threads will be created, when they will be deleted or slept and how the jobs will be distributed. There have been various libraries put forward with the intend of automating these jobs. In the next section, OpenMP will be introduced which tries to address some of the problems.

2.1.1.1 OpenMP

OpenMP is a specification, a set of terms and their definitions, which enables parallel programming in shared memory environments. It has implementations supporting C/C++ and FORTRAN, working on many architectures such as Unix and clones (Linux) and Microsoft operating systems. OpenMP does not fit the definition of library, parallel programming is achieved by using compiler directives (or pragma definitions, as it is also widely used). OpenMP uses multithreading for running tasks concurrently. It uses the general concept of multithreading where a master thread forks other threads which all run concurrently. If the tasks are finished, a join operation is conducted so that all the threads merge and only the master thread remains, like in the beginning. If more parallel tasks are forthcoming, a new set of threads could be forked again and this execution cycle could continue.

The cycle of creating new threads, merging and starting all over again can be best seen in Figure 2.0 (image taken from Wikipedia). In the image, the top part shows a serial execution of various tasks, and the bottom part shows how it can be executed in parallel. We have to note here that parallel tasks 1 to 3 seen in the image all contain independent subtasks as shown. Had there been a dependency, parallel execution of the subtasks would not have been carried out. In this case, where tasks are dependent, serial execution must be preferred.

OpenMP uses preprocessor directives to achieve parallel execution. Also called pragma definitions, these directives are ignored by the compiler if OpenMP extensions are not installed and the compiler does not support it. Thus, serial versions of the program can be easily built, without any modifications (there are some caveats involved when more complex operations need to be done, and so serial versions need modifications, but advanced OpenMP is left to the reader).

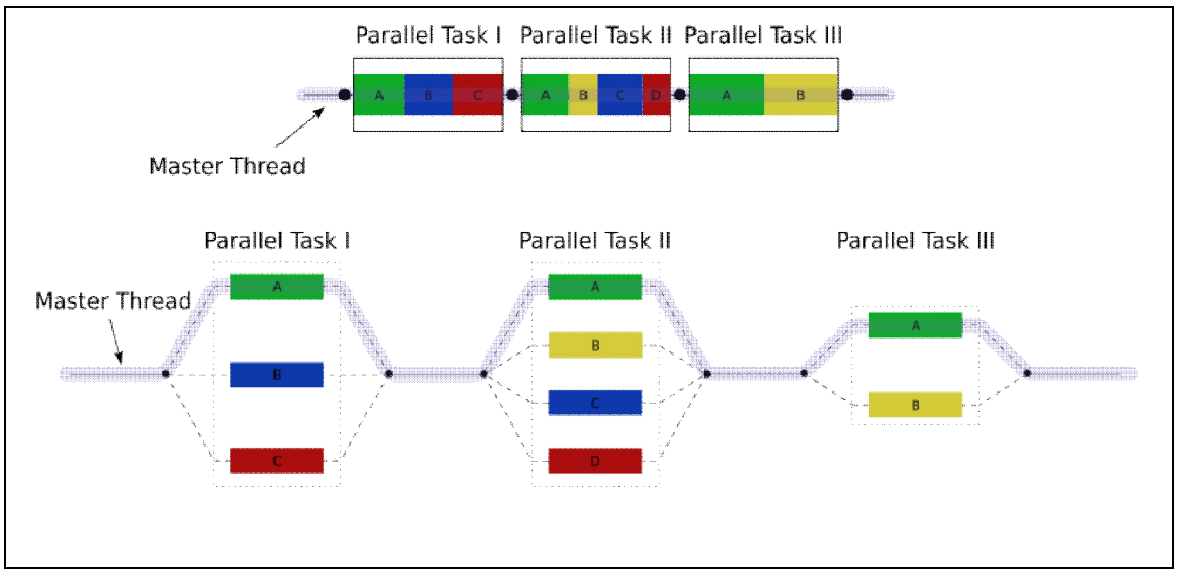


Figure 2.0: Execution model in OpenMP

A “hello world” program using OpenMP can be found in Figure 2.1. As can be seen, OpenMP pragma directives start with “#pragma.omp”. This simple example may give

rise to the following question: How do we specify the number of threads we want? This and many other configuration parameters can be set using environment variables; thread number is specified by setting “OMP_NUM_THREADS” variable. This flexibility of setting numerous configuration options by environment variables is given as an advantage of OpenMP.

```
#include "omp.h"
int main (int argc , char* argv[])
{
#pragma omp parallel
printf ("Hello, world . \ n" ) ;
return 0 ;
}
```

Figure 2.1: Hello World in OpenMP

It is noted that while multiprogramming is easy and flexible with OpenMP, avoiding race conditions and deadlocks is left to the programmer. If tasks have dependence on each other, synchronization constructs such as barriers, critical sections and others can be used to avoid erroneous execution. Private variables can also be used within OpenMP to avoid races between threads accessing non-local variables and global data.

Table 2.1: Do-it-yourself (DIY)Threads vs. OpenMP

DIY Threads	**OpenMP**
Threads need different libraries	OpenMP is implemented using compiler directives
Threads are not as portable, different libraries exist between different platforms (eg. Unix and Windows) and there are different libraries even in the same platform.	OpenMP is portable, implementations conform to the standard
Parallelism has to be specified explicitly, high workload on the programmer.	OpenMP takes care of most of the decomposition of work and data
Serial and parallel versions have many differences.	Serial and parallel versions can be generated in general without any modifications (except for some advanced constructs where limited modifications are needed)

OpenMP is a widely popular solution to parallel programming. It provides various advantages over custom threading implementations. Scalability, simplicity and flexibility concerns are addressed to some extent. In the shared memory architectures, OpenMP promises satisfactory performance, and the latest specification OpenMP 3.0 released on May 2008 is expected to improve some of the shortcomings (eg. tasks are added to OpenMP) as more and more of the compiler producers embrace it.

OpenMP is basically a multithreading solution saving the programmer the hassle of maintaining threads and parallelizing a work. In Table 2.1 differences between a custom multithreaded implementation and OpenMP is summarized. OpenMP provides

flexibility, scalability and simplicity on shared memory systems. In the next section we will be looking at message passing techniques, which provide parallelism in shared nothing architectures. By using OpenMP and MPI in a hybrid application, one can parallelize a program to run on multicore distributed clusters.

2.1.1.2 Message Passing Interface (MPI)

MPI is a specification describing an API for communication between different computers (and hence the name message passing interface). It has been used widely in the high performance computing community, starting much before multicore processors were common, to program distributed memory architectures. MPI has proven itself to be the de facto standard for many applications using distributed memory, but it can be used for shared memory programming as well. It uses the general concept of message passing, of which one of the earlier examples had been PVM (Parallel Virtual Machine) (Sunderam ,1990). PVM also happens to be one of the main motivations for a message passing standard.

MPI works with processes, as opposed to threads, and thus it directly inherits process properties. Refer to the earlier comparison in Table 3.1 for more information. MPI provides an API for communication between different program processes working on a problem. Normally each core is assigned a single process for maximum performance. MPI enables detailed configuration and more processes can be assigned at the cost of processor performance. If more than one process is assigned to a processor core, it is said to be oversubscribed and processes share the CPU time.

What happens when MPI is used in a shared memory environment? If a program can run in distributed memory architectures, then there should be no problem running in shared memory architectures as well. Even so, having a multithreaded approach would increase the performance of the application because threads are more suitable in shared

memory systems. MPI standard does not require implementations to be multithreaded and different implementations have taken different approaches. Multithreading is not supported at large by implementations. Still, there is some room for improvements in process handling when shared memory architectures are used, such as direct memory copying. This is used by the Open MPI implementation.

Similar to the example source code given in the OpenMP section, MPI equivalent is presented in Figure 3.3. From the “hello world” example, it can be seen that MPI is used by calling library functions, as opposed to compiler pragma directives used in OpenMP. Although both source codes are given in C, MPI can be used with C++ and FORTRAN too, as it is the case with OpenMP.

```
#include <stdio.h>
#include <mpi.h>
int main (int argc , char*  argv[]) {
int myrank , s i z e ;
MPI_Init(&argc , &argv ) ;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank ) ;
MPI_Comm_size(MPI_COMM_WORLD, &size ) ;
printf("Processor %d of %d says : Hello World!\n", myrank, size) ;
MPI_Finalize() ;
return 0 ;
}
```

Figure 2.2: Hello World in MPI

MPI provides many functions for communication between processes. These include point-to-point communication constructs where two processes are connected or collective operations including more than two processes. MPI also provides ways to group processes according to any way programmer likes. Furthermore, virtual topologies can be chosen so that MPI can simplify and increase the efficiency of communication between processes. There are also many variations in communication functions according to the communication format. It can be synchronous where the functions block until the message is safely shared between the parties, or it can be asynchronous where processes continue their work and check later whether the operation has completed successfully. Collective communication also has many forms including synchronization constructs (eg. barriers), scatter, and gather and reduction operations. These provide low level flexibility to the programmer, he is free to choose the best strategy to use.

MPI naturally supports the SPMD form of parallelism by spawning multiple copies of the same program and enabling communication in between them. Nevertheless, MPI implementations also allow different programs to be spawned as well, with communication between them intact. Thus, MPMD (Multiple Program Multiple Data) form can also be implemented easily.

You can see a comparison of OpenMP and MPI in Table 2.2. The most obvious difference to take into consideration is that OpenMP works on shared memory architectures while MPI works in both (Basumallik, 2007).

Table 2.2: OpenMP vs. MPI

OpenMP	MPI
OpenMP is implemented using compiler directives	MPI uses library function calls
OpenMP uses the thread model	MPI uses the process model
OpenMP is suited for shared memory programming	MPI is best suited for distributed memory programming, but shared memory programming can also be used
OpenMP has simple parallelization of loops and other tasks, demands less work on programmer's side	MPI has more flexibility and low level configuration options. There's more work for the programmer to do
Serial versions in OpenMP is often trivial	Serial version without the MPI overhead needs “ifdef” directives that clutter the code or an MPI stub library containing dummy code

Among the hottest topics in MPI development community include “fault tolerance”. This is important in grid computing systems where the computing elements making up the architecture could be physically distant and system wise heterogeneous. Fault tolerance is resuming operation as a whole if a failure occurs in some part of the system. There is continued development in this area in many MPI implementations.

2.2 The SAT solver Algorithm

The search process of Bagais’ algorithm keeps all possible assignments that satisfy a clause before moving forward. This is unlike the case of other complete and incomplete algorithms which take a decision on a variable in a formula. Such a decision may result in backtracking if the decision was wrong. The current assignments in Bagais’ algorithm can be used to determine the values of variables that satisfy the next clauses.

If none of the literals of the current clause agrees with all the assignments, then the formula is unsatisfiable.

The algorithm starts from the first clause c_1 and produces all assignments that satisfy c_1. The assignments that satisfy c_1 are its literals. If the clause has k literals, then k assignment can satisfy it. For instance if $c_1 = (\bar{y}_1 \vee y_2 \vee \bar{y}_5)$, then the assignments are $\bar{\mathbf{y}}_1$, $\mathbf{y}_2$, and $\bar{\mathbf{y}}_5$ respectively. Next each assignment is combined with all the literals of the second clause which results to one of the following cases.

1. The literal agrees with the assignment.
2. The literal disagrees with the assignment.
3. The literal is neutral.

The literal agrees with the assignment.

The assignment contains a literal that is identical to the literal in the current clause. For example, if the assignment is $\mathbf{y_3y_2}$ and a literal in the current clause is $\mathbf{y_3}$, then $\mathbf{y_3}$ is said to agree with $\mathbf{y_3y_2}$. Therefore, the assignment $\mathbf{y_3y_2}$ satisfies both the previous clauses and the current clause. There is no need duplicating the literal in an assignment.

The literal disagrees with the assignment.

The assignment contains a literal that is compliment of the literal in the current clause. For example, if the assignment is $\mathbf{y_3y_2}$ and a literal in the current clause $\bar{\mathbf{y}}_3$, then $\bar{\mathbf{y}}_3$ is said to disagree with $\mathbf{y_3y_2}$. This is a contradiction since a literal and its compliment cannot satisfy an assignment simultaneously as a result no assignment is produced.

The literal is neutral.

The assignment did not contain a literal that is identical to the literal in the current clause. For example, if the assignment is $\mathbf{y_3y_2}$ and a literal in the current clause is $\bar{\mathbf{y}}_4$, then $\bar{\mathbf{y}}_4$ is said to be neutral to $\mathbf{y_3y_2}$ and resulting assignment is $\mathbf{y_3y_2\bar{y}_4}$.

Example 2.1

Consider the following Boolean formula:

$$F=(y_1 \vee y_3 \vee \bar{y}_4) \wedge (\bar{y}_1 \vee y_2 \vee \bar{y}_5) \wedge (y_2 \vee \bar{y}_3 \vee y_4) \wedge (y_3 \vee y_5 \vee y_1) \quad (2.1)$$

The sequential determination of the satisfying assignments can be achieved as follows:

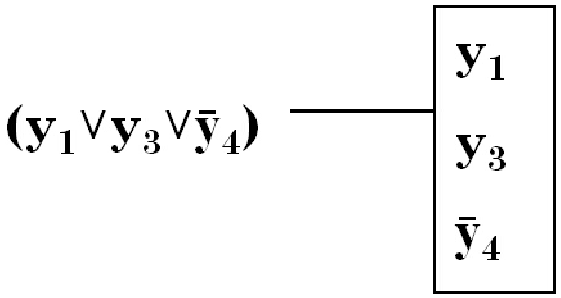


Figure 2.3: Satisfying Assignments for the Clause $(y_1 \vee y_3 \vee \bar{y}_4)$

The rectangle in Figure 2.3 contains the satisfying assignments for the first clause which are its literals.

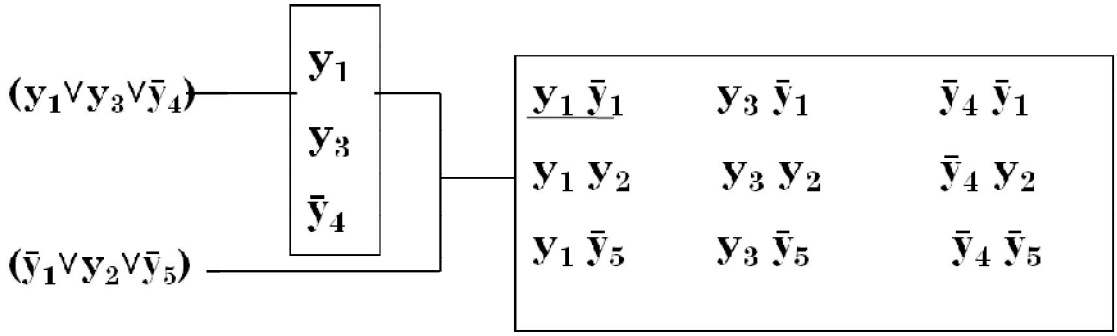


Figure 2.4: Computing Satisfying Assignments from Current Assignment and Clause $(\bar{y}_1 \vee y_2 \vee \bar{y}_5)$.

The first literal of the second clause, $\bar{y}_1$ is combined with the assignments in the first rectangle of Figure 2.4 to obtain the three assignments in the first line of the second rectangle, namely, $y_1\bar{y}_1$, $y_3\bar{y}_1$ and $\bar{y}_4\bar{y}_1$. The second and last literals of the second clause are combined with the assignments in the first rectangle to obtain the second and last line in the second rectangle respectively.

The underlined assignments in the second rectangle can be deleted since the previous assignment and the literal with which it was combined disagree. Figure 2.5 gives the satisfying assignments after deleting the conflicting assignment.

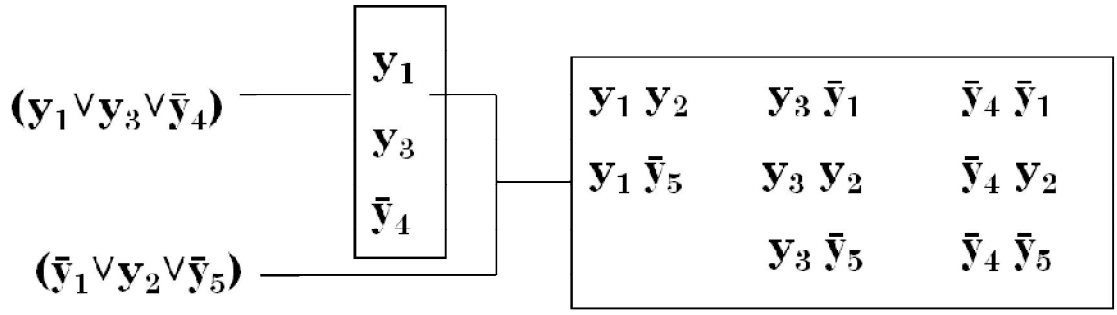


Figure 2.5: Satisfying Assignments for Clauses $(y_1 \vee y_3 \vee \bar{y}_4)$ $(\bar{y}_1 \vee y_2 \vee \bar{y}_5)$

The assignments in the second rectangle of Figure 2.5 are combined with the literals of the third clause, $(y_2 \vee \bar{y}_3 \vee y_4)$, which results in the assignments in the third rectangle of Figure 2.6.

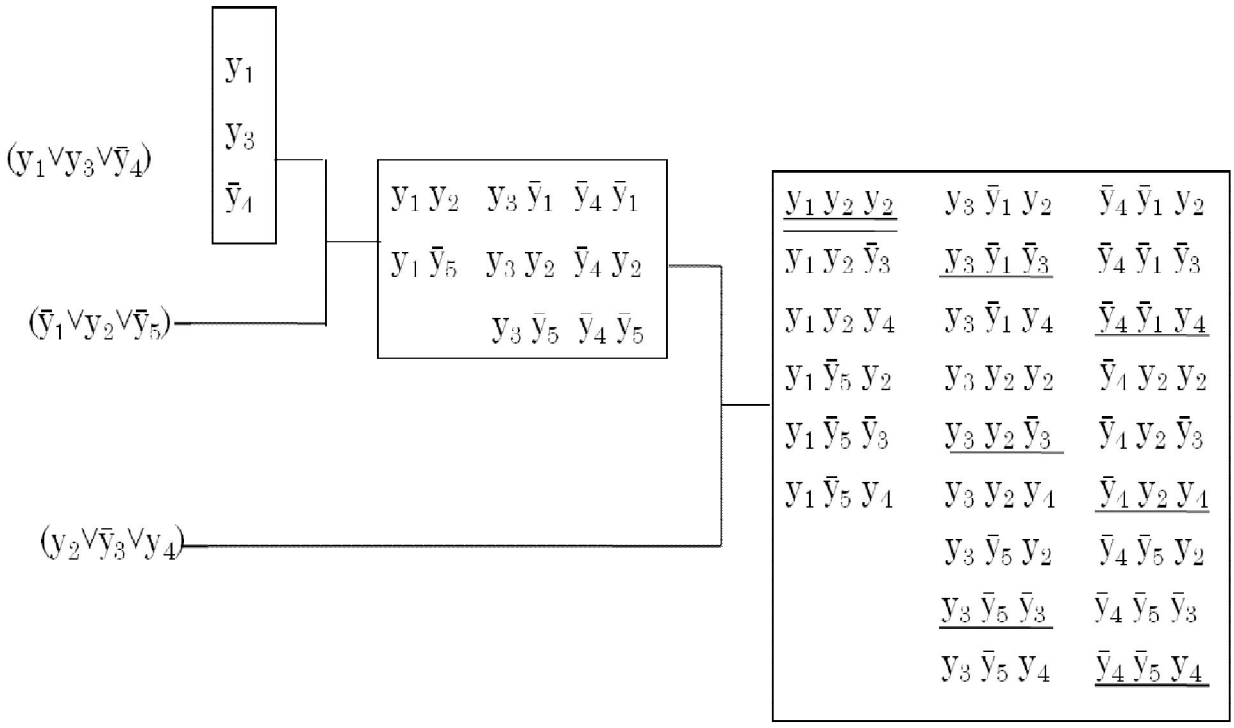


Figure 2.6: Computing Satisfying Assignments from Current Assignment and Clause, $(y_2 \vee \bar{y}_3 \vee y_4)$.

The singly underlined assignments in Figure 2.6 are the conflicting assignments which cannot participate in formation of new assignments; as a result they are deleted. The doubly underlined assignments contain literals neutral to the old assignments, therefore the old assignments also satisfy the new clause, and in that case the literal need not duplicated. The deletion of the conflicting assignments and simplification of assignments containing neutral literals results in assignments in the third rectangle of Figure 2.7.

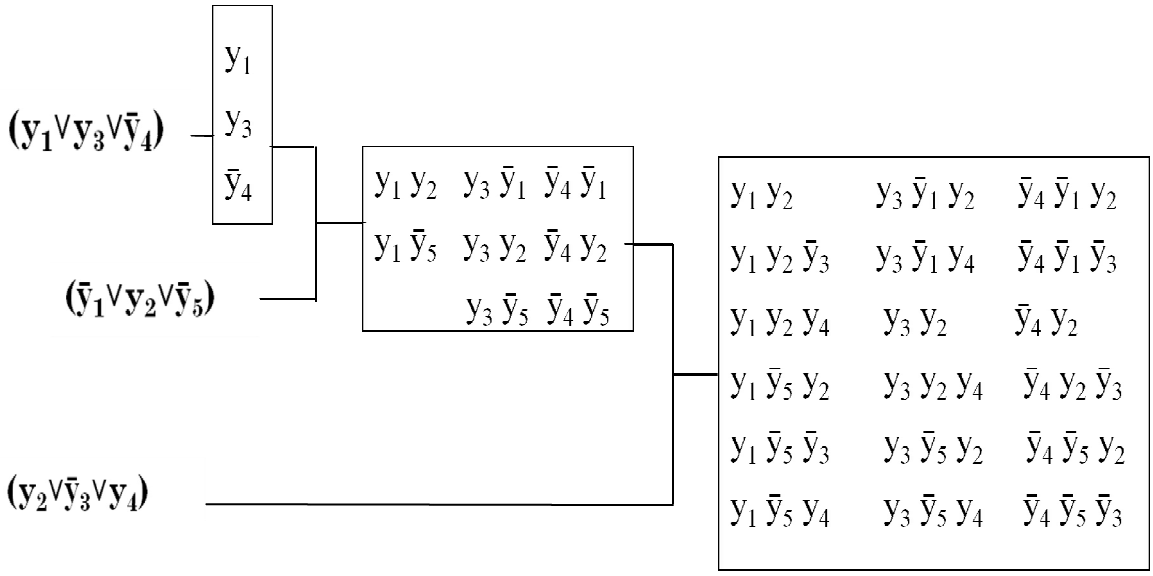


Figure 2.7: Satisfying Assignments for Clauses $(\bar{y}_1 \vee y_2 \vee \bar{y}_5)(y_2 \vee \bar{y}_3 \vee y_4)$ $(y_1 \vee y_3 \vee \bar{y}_4)$

The assignments in the third rectangle of Figure 2.7 are checked with the literals of the last clause which results in the assignments in second rectangle of Figure 2.8.

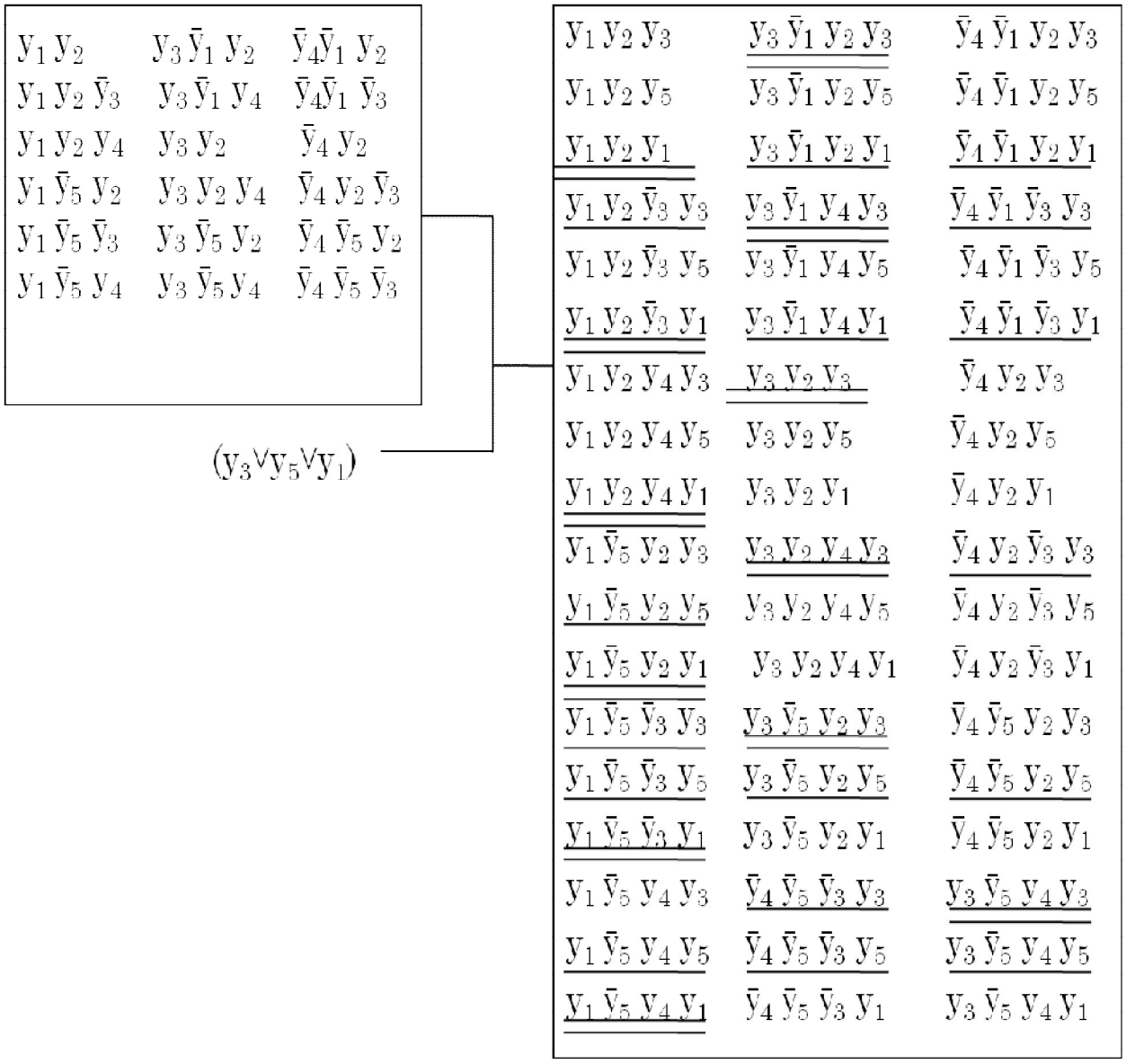


Figure 2.8: Computing Satisfying Assignments from Current Assignment and Clause $(y_3 \vee y_5 \vee y_1)$.

The assignments in the second rectangle of Figure 2.9 (which are the satisfying assignments for the formula) are obtained after deleting the neutral literals and assignments containing conflicting literals. All the remaining assignments cause the formula to evaluate to true if all its literals are assign true values. There are 38 satisfying substitution out of a potential of 81.

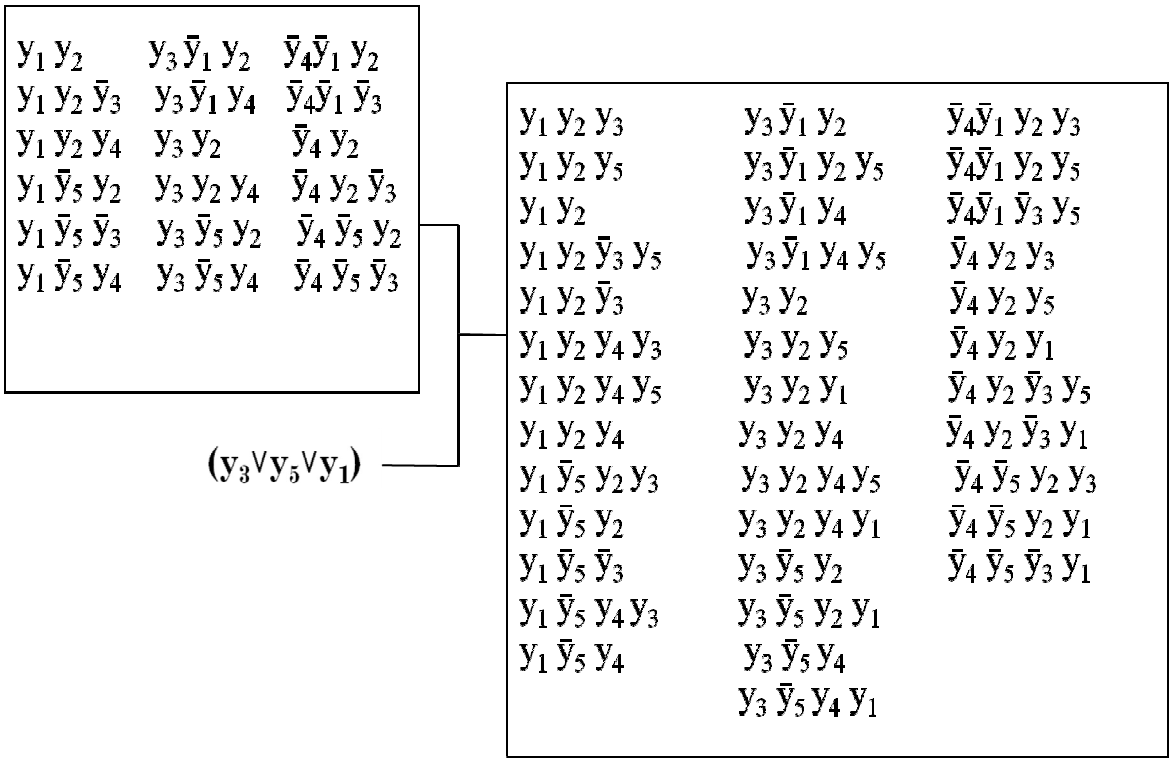


Figure 2.9: Satisfying Assignments for the Boolean Formula.

Figure 2.10 presents Bagais' algorithm.

The Algorithm

Input: satisfiability formula of m clauses

Output: possible assignments satisfying the m clauses stored in A

1. Extract clauses from the formula such that no two clauses share a literal and put them in array CLS.
2. Save the literals of clauses in CLS in array LIT
3. If there are missing literals not in LIT then

   Extract the clauses that contain them from the formula and add them in CLS
4. Add the literals of the first clause in CLS as assignments in a list A
5. For remaining clauses in CLS

   For all literals in CLS[i]

Generate-asn(literal, A, temp)

Endfor

6. For other clauses in formula not in CLS

For all literals in CLS[i]

Generate-asn(literal, A, temp)

Endfor

Remove subsumed assignments from temp

A:=A + temp

Endfor

7. If A is empty then the formula is unsatisfiable

Else the formula is satisfiable by assignments in A

Procedure Generate-asn(literal, list1, list2)

Input: a literal in a clause and a list of assignments in list1

Output: a list of assignments stored in list2 produced by combining literal with list1

For assignments in list1

If literal does not conflict with the assignment then

Combine the literal and the assignment

Add the combination in list2

Endif

Endfor

Return list2

Figure 2.10: The SAT solver Algorithm(Bagais, 2008)

2.2.0 Time Complexity

The first three steps of the algorithm take polynomial time of number of clauses, i.e. it costs O(m). Steps five and six are the main contribution to the time complexity of the whole algorithm. The two steps rely on the number of assignments generated in each iteration for the for-loop. For step five, that is determined by how many clauses in CLS and for step six, that number is determined by the end of step five.

For step 5: the number of clauses in CLS is either 2n/k clauses (worst case) or more than 2n/k clauses. Where n is the number of variables.

Upper bound:

1. In step 5 of the algorithm, generating assignments with the least number of clauses 2n/k that include 2n literals is the worst case.
2. For the worst case, the upper bound is $(k(k-1))^{n/k}$

The proof can be found in (Bagais, 2008).

CHAPTER THREE: IMPLEMENTATION AND OF THE SAT SOLVER PARALLELISATION

3.0 Introduction

This chapter presents the parallelisation of Bagais' algorithm. Section 3.1 identifies the parallel stages of the algorithm. In Section 3.2 the details of the data portioning for the parallel algorithm are presented. Section 3.3 presents the communication details among the processes. Section 3.4 gives an example to showcase the correctness of the parallel algorithm while Section 3.5 presents the Pseudocode of the proposed parallel algorithm.

3.1 Implementation

The implementation of the algorithm was accomplished using C programming language. The code for implementation of the algorithm is presented in Appendix A.

3.2 Parallel Process

The parallel process of the above algorithm can be achieved using a 2-D mesh of processing units. Since each previous assignment needs to be combined with all the literals of the next clause to form new assignments, concurrent formation of assignments can be achieved by placing different assignments and different literals on different processing units. The type of communication need is shift. These assignments and literals are systematically shifted between the neighbouring processing units after formation of new assignments. Each process stores and shifts the received literal to the process to its right. The literals are shifted to the right one step with wraparound and all the processing nodes get all the literals they need after L-1 steps. The right shift communication stops after L-1 times. The formed assignments are shifted downward one step so that they can be checked with literals of the next clause. The downward shift

communication is with no wrap around. The processing nodes on the last row of the mesh end up with the satisfying assignments for the formula.

3.3 Distribution of Literals

Figure 3.0 gives the mapping of processors onto a 2-D mesh grid. The processing nodes are labelled from $P_{0,0}$ to $P_{C-2,L-1}$. Initially processing nodes $P_{i,j}$ holds literals $Lit_{i+1,j.}$ A is the container for current satisfying substitution. In addition, process $P_{0,j}$ holds the literals A0j, which are the satisfying assignments for the first clause. Aij is the list of satisfying assignments for the first (i+1) clause(s). $L_{i,j}$ is the jth literal of the ith clause. L is the number of literals in a clause and C is the number of clauses in the formula.

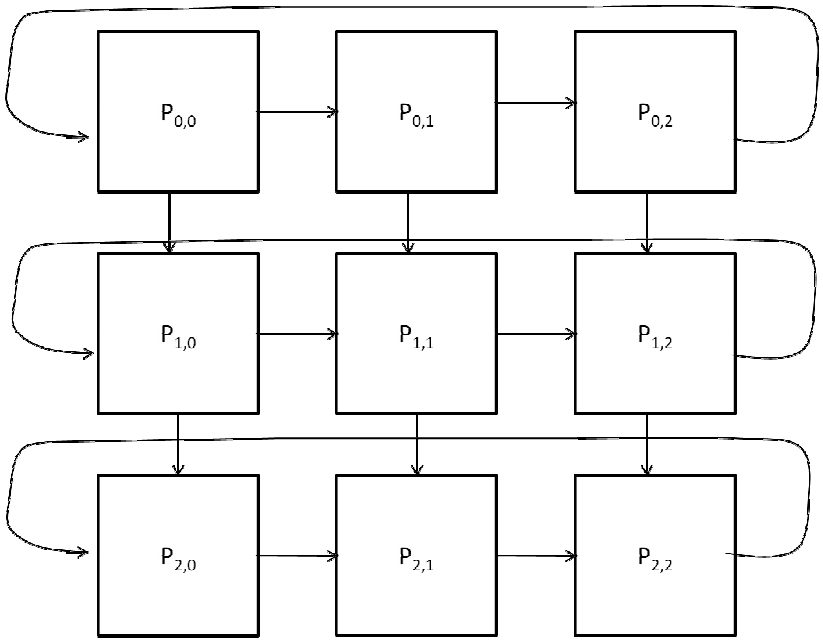


Figure 3.0: 3x3 2-D Mesh of Process Grid

3.4 Communication Steps

First, the distribution of literals and assignment can be achieved as shown in Figure 3.1; $P_{i,j}$ is assigned $L_{i+1,j}$ and $P_{0,j}$ is assigned A0j in addition. The assignments satisfying the

first two clauses are obtained by $P_{0,j}$ (which starts in step 1) by combining A0k with $L_{1,k}$ which are shifted down by one step and literals are shifted to the right by one step with wraparound as shown in Figure 3.1 . The assignments satisfying the first three clauses are obtained by $P_{1,j}$ (which starts in step 2) by combining assignments in A1j with the literals of the third clause. The formed assignments are shifted down one step while literals shifted to the right one step but with wrap around as shown in Figure 3.2. The assignments satisfying the first four clauses (which is the formula) are obtained by $P_{2,j}$ (which starts in step 3) by combining assignments in A2j with the literals of the fourth clause as shown in Figure 3.3. These assignments are collected in A3j and a gather operation can be employed to collate the satisfying assignments. The formation of assignments satisfying the first two clauses end at third step as shown in Figure 3.3. The formation of assignments satisfying the first three clauses end at the fourth step as shown in Figure 3.4. The formation of assignments satisfying the first four clauses (the formula) end at fifth step as shown in Figure 3.5.

In general, the formation of assignments satisfying the first i clauses end at i+1 step as shown in Figure 3.3. ($2 \leq i \leq (L+C-2)$) .

$$F = (L_{0,0} \vee L_{0,1} \vee L_{0,2}) \wedge (L_{1,0} \vee L_{1,1} \vee L_{1,2}) \wedge (L_{2,0} \vee L_{2,1} \vee L_{2,2}) \wedge (L_{3,0} \vee L_{3,1} \vee L_{3,2}) \quad (3.1)$$

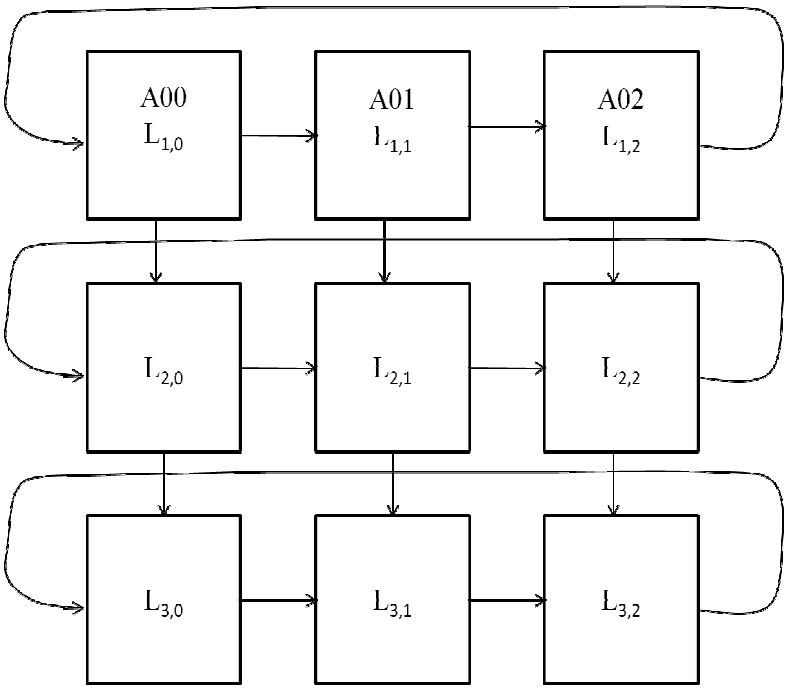


Figure 3.1: Data Mapping onto Process Grid and Computation of Satisfying Assignment and Shifting Process at Clock Tick 1.

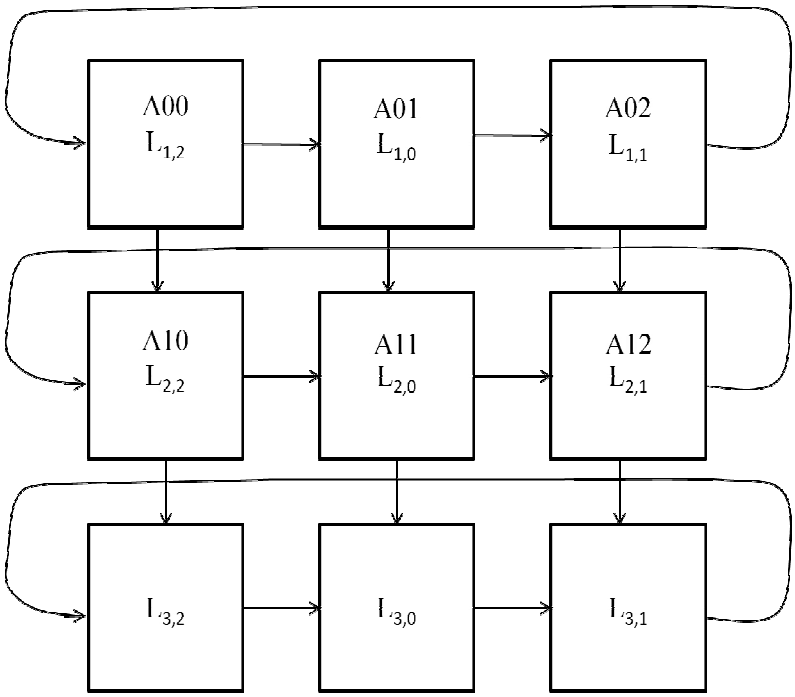


Figure 3.2: Computation of Satisfying Assignment and Shifting Process at Clock Tick 2.

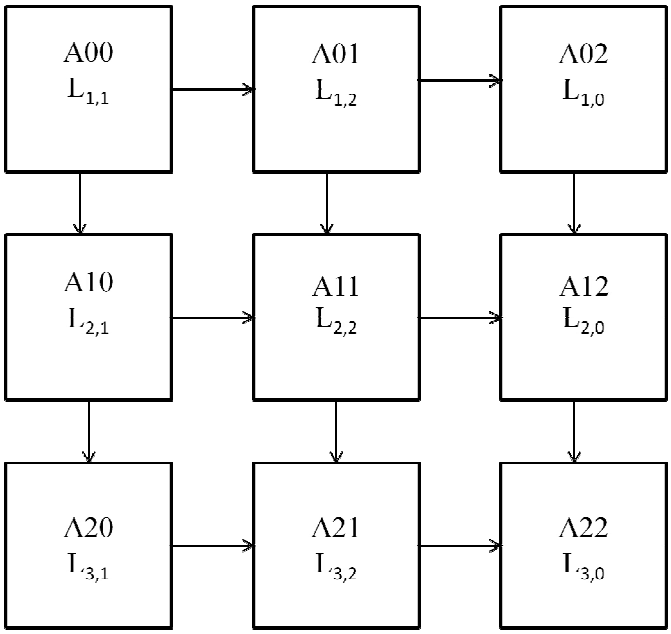


Figure 3.3: Computation of Satisfying Assignment and Shifting Process at Clock Tick 3.

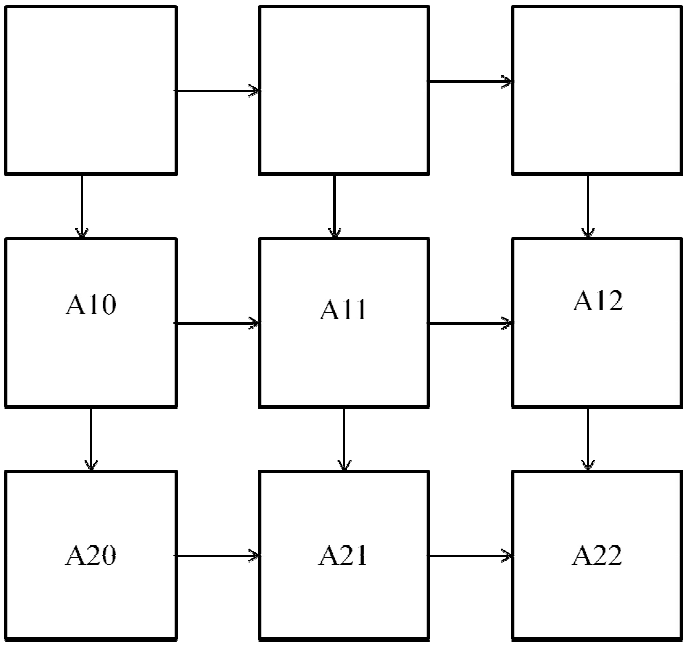


Figure 3.4: Computation of Satisfying Assignment and Shifting Process at Clock Tick 4.

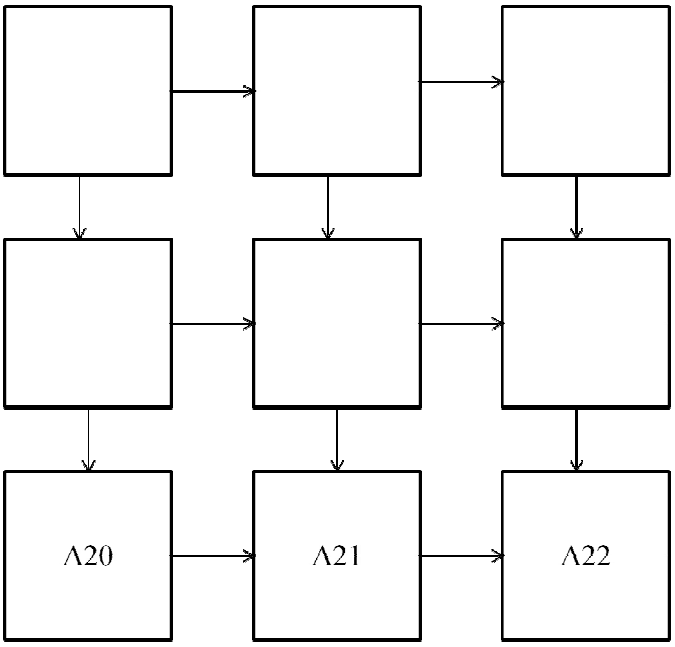


Figure 3.5: Computation of Satisfying Assignment and Shifting Process at Clock Tick 5.

3.5 An Illustration

Figure 3.6 is the mesh of processing nodes needed to determine the satisfiability of a formula with four clauses of three literals (3-CNF), for example,

$$F=(y_1 \vee y_3 \vee \bar{y}_4) \wedge (\bar{y}_1 \vee y_2 \vee \bar{y}_5) \wedge (y_2 \vee \bar{y}_3 \vee y_4) \wedge (y_3 \vee y_5 \vee y_1) \quad (3.2)$$

To run this in linear time we need L x C-1 processing units. In this case, 9 processing units are required as shown in Figure 3.6.

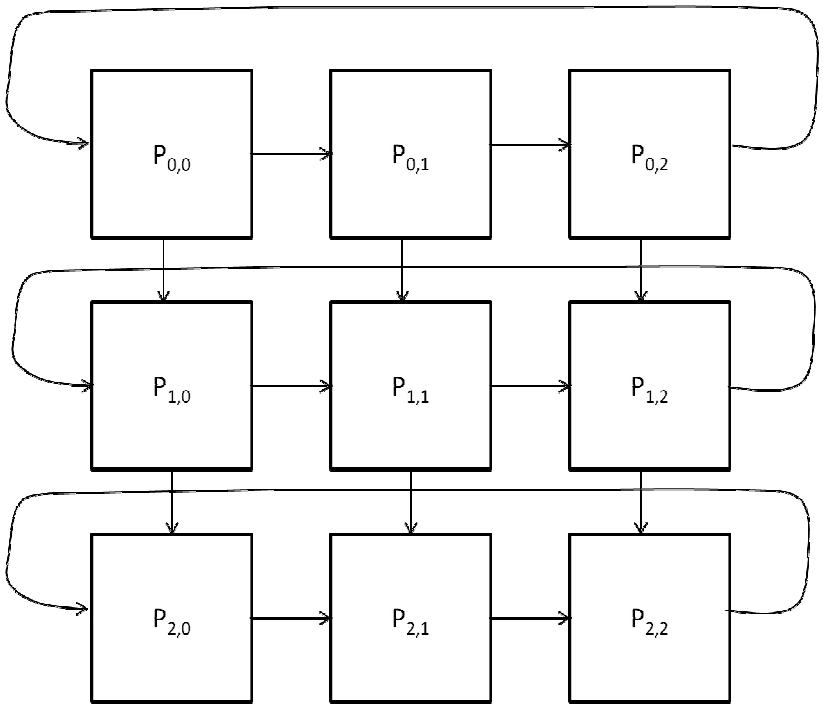


Figure 3.6: 3x3 2-D Mesh of Process Grid.

$$F=(y_1 \vee y_3 \vee \bar{y}_4) \wedge (\bar{y}_1 \vee y_2 \vee \bar{y}_5) \wedge (y_2 \vee \bar{y}_3 \vee y_4) \wedge (y_3 \vee y_5 \vee y_1) \quad (3.2)$$

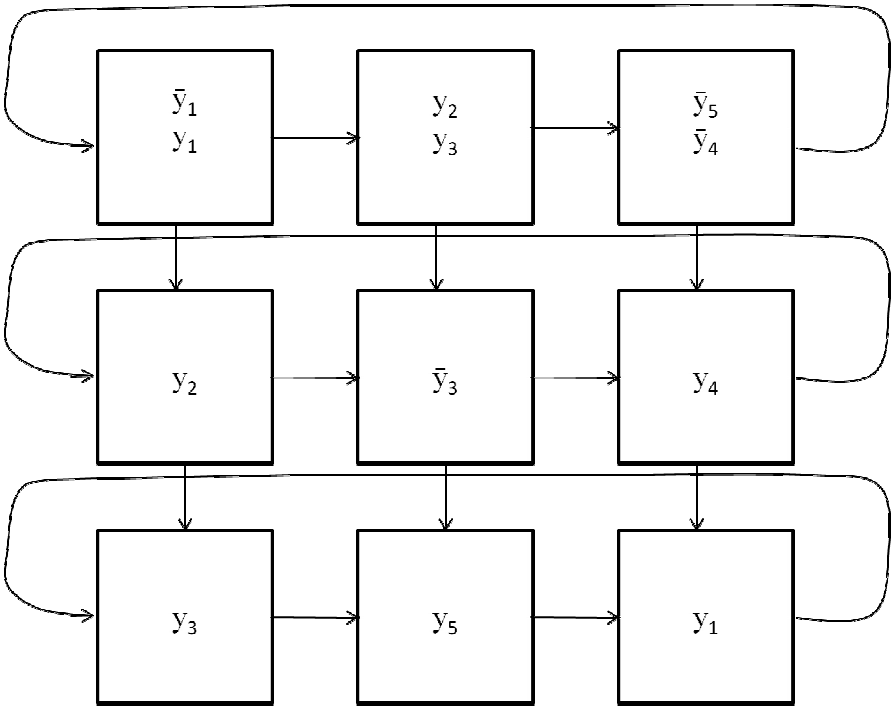


Figure 3.7: Data Mapping onto Process Grid.

Figure 3.7 gives initial data distribution. Formation of satisfying assignments for first and second clause is carried as depicted in figure 3.8. The literals are then shifted to the right one step with wraparound while the formed assignments are shifted down ward one step. The resulting assignments at this stage are shown in Table 3.0a and satisfying assignments after discarding the assignment with conflicting literals are given in Table 3.0b.

$$F=(y_1 \vee y_3 \vee \bar{y}_4) \wedge (\bar{y}_1 \vee y_2 \vee \bar{y}_5) \wedge (y_2 \vee \bar{y}_3 \vee y_4) \wedge (y_3 \vee y_5 \vee y_1) \quad (3.2)$$

Clock tick: 1

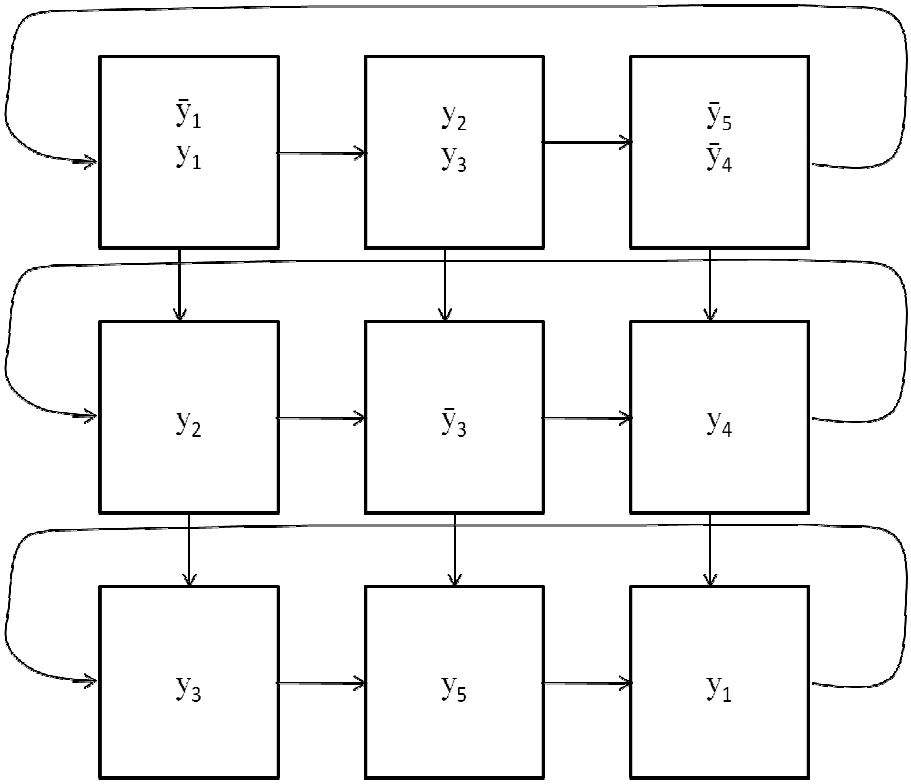


Figure 3.8: Computation of Satisfying Assignment and Shifting Process at Clock Tick 1.

Table 3.0a: Computed Assignments after Clock Tick 1

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{2,0}$	$P_{2,1}$	$P_{2,2}$
$\bar{y}_1 y_1$	$y_3\ y_2$	$\bar{y}_4 \bar{y}_5$						

Table 3.0b: Satisfying Assignments after Clock Tick 1

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{2,0}$	$P_{2,1}$	$P_{2,2}$
	$y_3\ y_2$	$\bar{y}_4 \bar{y}_5$						

$$F=(y_1 \vee y_3 \vee \bar{y}_4) \wedge (\bar{y}_1 \vee y_2 \vee \bar{y}_5) \wedge (y_2 \vee \bar{y}_3 \vee y_4) \wedge (y_3 \vee y_5 \vee y_1) \quad (3.2)$$

Clock tick: 2

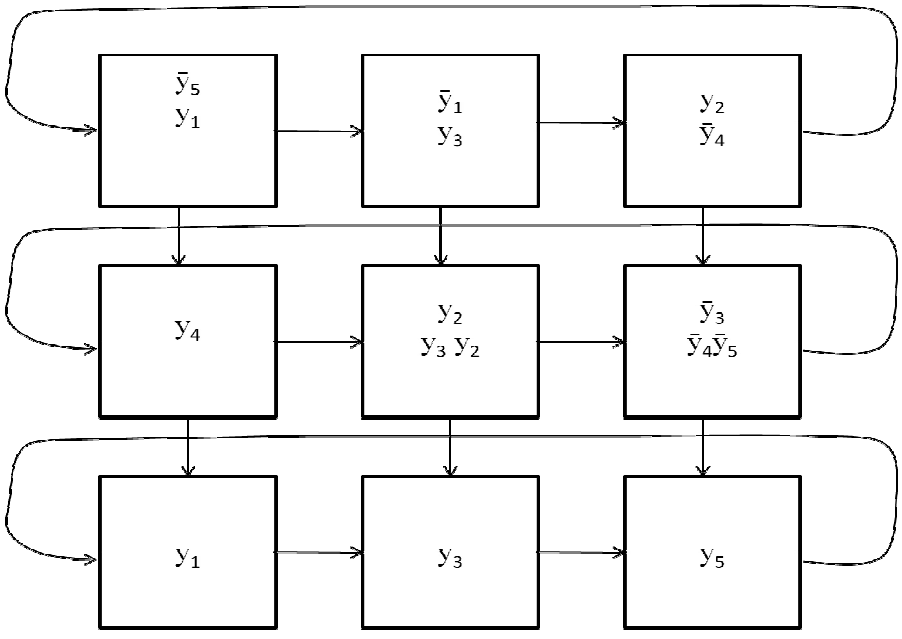


Figure 3.9: Computation of Satisfying Assignment and Shifting Process at Clock Tick 2.

right with wraparound while the formed assignments are shifted down without wrap around. The assignments formed results in the outputs on the Table 3.1a and Table 3.1b

contains some satisfying assignments for first, second and third clauses after removing the assignments with conflicting literals and the literals neutral to previous assignments.

Table 3.1a: Computed Assignments after Clock Tick 2

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{2,0}$	$P_{2,1}$	$P_{2,2}$
$y_1\bar{y}_5$	$y_3\bar{y}_1$	$\bar{y}_4y_2$		$y_3\ y_2\bar{y}_3$ $y_3\ y_2y_2$	$\bar{y}_4\bar{y}_5y_4$ $\bar{y}_4\bar{y}_5\bar{y}_3$			

Table 3.1b: Satisfying Assignments after Clock Tick 2

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{2,0}$	$P_{2,1}$	$P_{2,2}$
$y_1\bar{y}_5$	$y_3\bar{y}_1$	$\bar{y}_4y_2$		$y_3\ y_2$	$\bar{y}_4\bar{y}_5\bar{y}_3$			

$$F=(y_1 \vee y_3 \vee \bar{y}_4) \wedge (\bar{y}_1 \vee y_2 \vee \bar{y}_5) \wedge (y_2 \vee \bar{y}_3 \vee y_4) \wedge (y_3 \vee y_5 \vee y_1) \quad \textbf{(3.2)}$$

Clock tick: 3

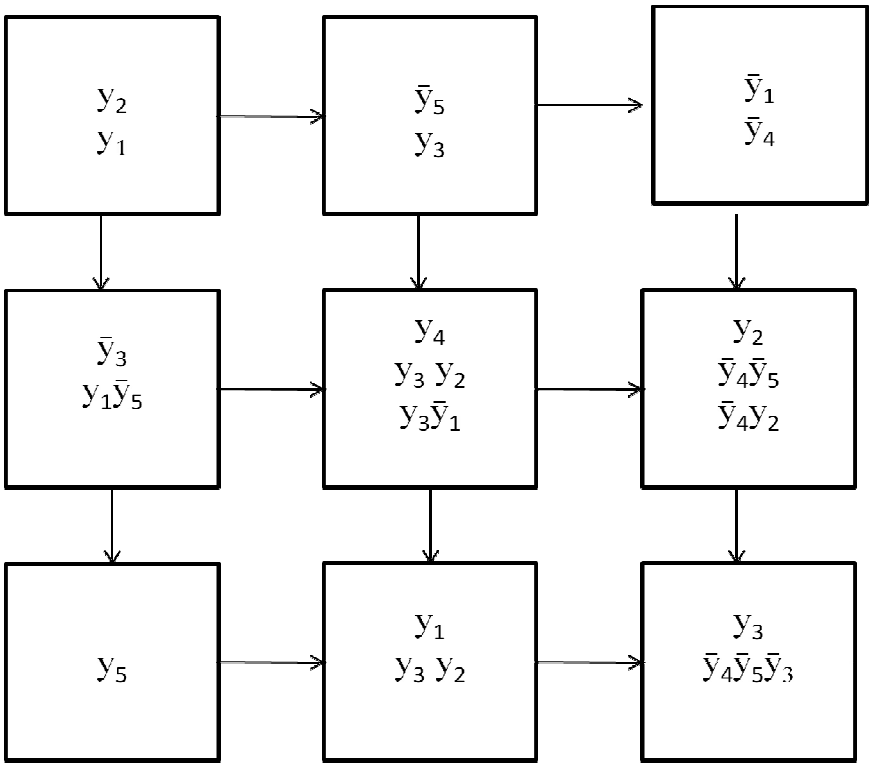


Figure 3.10: Computation of Satisfying Assignment and Shifting Process at Clock Tick 3.

The shifting of literals stops after the clock tick 2 and that is why the wraparound ring is discarded from Figure 3.10. In general, the shifting of literals stops after L-1 steps where L is the number of literals in the clause. The formed assignments are shifted down without wraparound. The assignments formed results in the outputs on the Table 3.2a while Table 3.2b has the satisfying assignments after removing the assignments with conflicting literals and the literals neutral to previous assignments.

Table 3.2a: Computed Assignments after Clock Tick 3

$\mathbf{P_{0,0}}$	$\mathbf{P_{0,1}}$	$\mathbf{P_{0,2}}$	$\mathbf{P_{1,0}}$	$\mathbf{P_{1,1}}$	$\mathbf{P_{1,2}}$	$\mathbf{P_{2,0}}$	$\mathbf{P_{2,1}}$	$\mathbf{P_{2,2}}$
y_1y_2	$y_3\bar{y}_5$	$\bar{y}_4\bar{y}_1$	$y_1\bar{y}_5y_2$ $y_1\bar{y}_5y_4$ $y_1\bar{y}_5\bar{y}_3$	$y_3\ y_2y_4$ $y_3\bar{y}_1\bar{y}_3$ $y_3\bar{y}_1y_2$ $y_3\bar{y}_1y_4$	$\bar{y}_4\bar{y}_5y_2$ $\bar{y}_4y_2y_4$ $\bar{y}_4y_2\bar{y}_3$ $\bar{y}_4y_2y_2$		$y_3\ y_2y_5$ $y_3\ y_2y_3$ $y_3\ y_2y_1$	$\bar{y}_4\bar{y}_5\bar{y}_3y_1$ $\bar{y}_4\bar{y}_5\bar{y}_3y_5$ $\bar{y}_4\bar{y}_5\bar{y}_3y_3$

Table 3.2b: Satisfying Assignments after Clock Tick 3

$\mathbf{P_{0,0}}$	$\mathbf{P_{0,1}}$	$\mathbf{P_{0,2}}$	$\mathbf{P_{1,0}}$	$\mathbf{P_{1,1}}$	$\mathbf{P_{1,2}}$	$\mathbf{P_{2,0}}$	$\mathbf{P_{2,1}}$	$\mathbf{P_{2,2}}$
y_1y_2	$y_3\bar{y}_5$	$\bar{y}_4\bar{y}_1$	$y_1\bar{y}_5y_2$ $y_1\bar{y}_5y_4$ $y_1\bar{y}_5\bar{y}_3$	$y_3\ y_2y_4$ $y_3\bar{y}_1y_2$ $y_3\bar{y}_1y_4$	$\bar{y}_4\bar{y}_5y_2$ $\bar{y}_4y_2$ $\bar{y}_4y_2\bar{y}_3$		$y_3\ y_2y_5$ $y_3\ y_2$ $y_3\ y_2y_1$	$\bar{y}_4\bar{y}_5\bar{y}_3y_1$

$$F=(y_1 \vee y_3 \vee \bar{y}_4) \wedge (\bar{y}_1 \vee y_2 \vee \bar{y}_5) \wedge (y_2 \vee \bar{y}_3 \vee y_4) \wedge (y_3 \vee y_5 \vee y_1) \quad (3.2)$$

Clock tick: 4

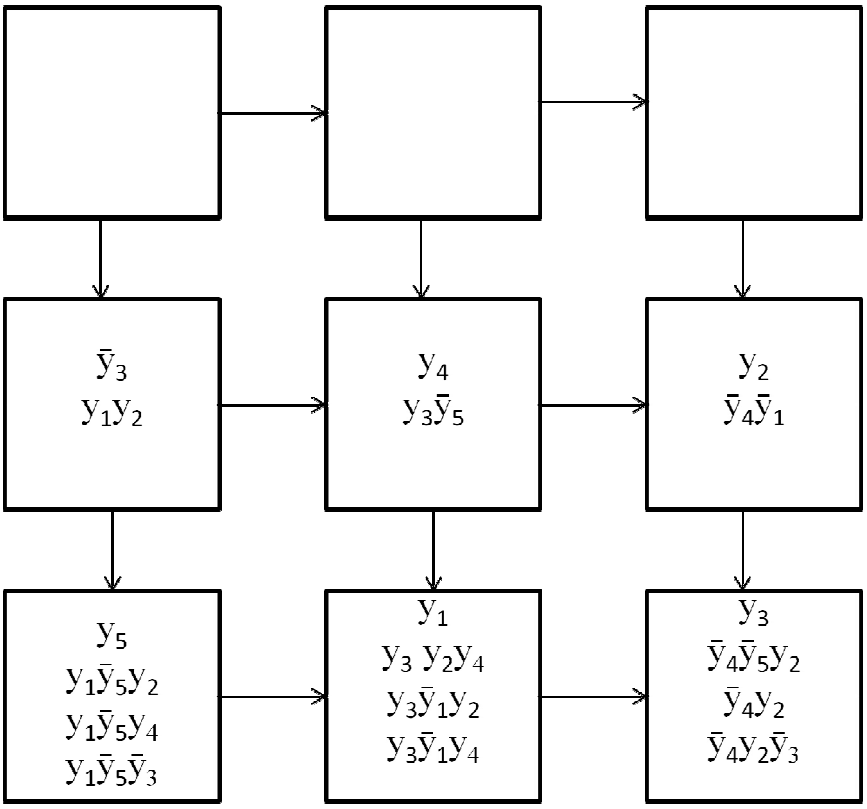


Figure 3.11: Computation of Satisfying Assignment and Shifting Process at Clock Tick 4.

After the Clock tick 4, the processes on the first row the grid have finished formation of assignments and the formed assignments by the process on other rows are shifted down without wrap around. The assignments formed results in the outputs on Table 3.3a and Table 3.3a contains the satisfying assignments after removing the conflicting and the literals neutral to previous assignments.

Table 3.3a: Computed Assignments after Clock Tick 4

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{2,0}$	$P_{2,1}$	$P_{2,2}$
			$y_1y_2y_2$ $y_1y_2y_4$ $y_1y_2\bar{y}_3$	$y_3\bar{y}_5\bar{y}_3$ $y_3\bar{y}_5y_2$ $y_3\bar{y}_5y_4$	$\bar{y}_4\bar{y}_1y_4$ $\bar{y}_4\bar{y}_1\bar{y}_3$ $\bar{y}_4\bar{y}_1y_2$	$y_1\bar{y}_5y_2y_3$ $y_1\bar{y}_5y_4y_3$ $y_1\bar{y}_5\bar{y}_3y_3$ $y_1\bar{y}_5y_2y_1$ $y_1\bar{y}_5y_4y_1$ $y_1\bar{y}_5\bar{y}_3y_1$ $y_1\bar{y}_5y_2y_5$ $y_1\bar{y}_5y_4y_5$ $y_1\bar{y}_5\bar{y}_3y_5$	$y_3\ y_2y_5$ $y_3\ y_2$ $y_3\ y_2y_1$ $y_3\ y_2y_4y_5$ $y_3\bar{y}_1y_2\ y_5$ $y_3\bar{y}_1y_4y_5$ $y_3\ y_2y_4y_3$ $y_3\bar{y}_1y_2y_3$ $y_3\bar{y}_1y_4y_3$ $y_3\ y_2y_4y_1$ $y_3\bar{y}_1y_2\ y_1$ $y_3\bar{y}_1y_4y_1$	$\bar{y}_4\bar{y}_5\bar{y}_3y_1$ $\bar{y}_4\bar{y}_5y_2y_1$ $\bar{y}_4y_2y_1$ $\bar{Y}_4y_2\bar{y}_3y_1$ $\bar{y}_4\bar{y}_5y_2y_5$ $\bar{y}_4y_2y_5$ $\bar{y}_4y_2\bar{y}_3y_5$ $\bar{y}_4\bar{y}_5y_2y_3$ $\bar{y}_4y_2y_3$ $\bar{y}_4y_2\bar{y}_3y_3$

Table 3.3b: Satisfying Assignments after Clock Tick 4

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{2,0}$	$P_{2,1}$	$P_{2,2}$
			y_1y_2 $y_1y_2y_4$ $y_1y_2\bar{y}_3$	$y_3\bar{y}_5y_2$ $y_3\bar{y}_5y_4$	$\bar{y}_4\bar{y}_1\bar{y}_3$ $\bar{y}_4\bar{y}_1y_2$	$y_1\bar{y}_5y_2y_3$ $y_1\bar{y}_5y_2$ $y_1\bar{y}_5y_4y_3$ $y_1\bar{y}_5y_4$ $y_1\bar{y}_5\bar{y}_3$	$y_3\ y_2y_5$ $y_3\ y_2$ $y_3\ y_2y_1$ $y_3\ y_2y_4$ $y_3\ y_2y_4y_5$ $y_3\ y_2y_4y_1$ $y_3\bar{y}_1y_2$ $y_3\bar{y}_1y_2\ y_5$ $y_3\bar{y}_1y_4$ $y_3\bar{y}_1y_4y_5$	$\bar{y}_4\bar{y}_5\bar{y}_3y_1$ $\bar{y}_4\bar{y}_5y_2y_1$ $\bar{y}_4\bar{y}_5y_2y_3$ $\bar{y}_4y_2y_1$ $\bar{y}_4y_2y_5$ $\bar{y}_4y_2y_3$ $\bar{y}_4y_2\bar{y}_3y_1$ $\bar{y}_4y_2\bar{y}_3y_5$

$$F=(y_1 \vee y_3 \vee \bar{y}_4) \wedge (\bar{y}_1 \vee y_2 \vee \bar{y}_5) \wedge (y_2 \vee \bar{y}_3 \vee y_4) \wedge (y_3 \vee y_5 \vee y_1) \quad \textbf{(3.2)}$$

Clock tick:5

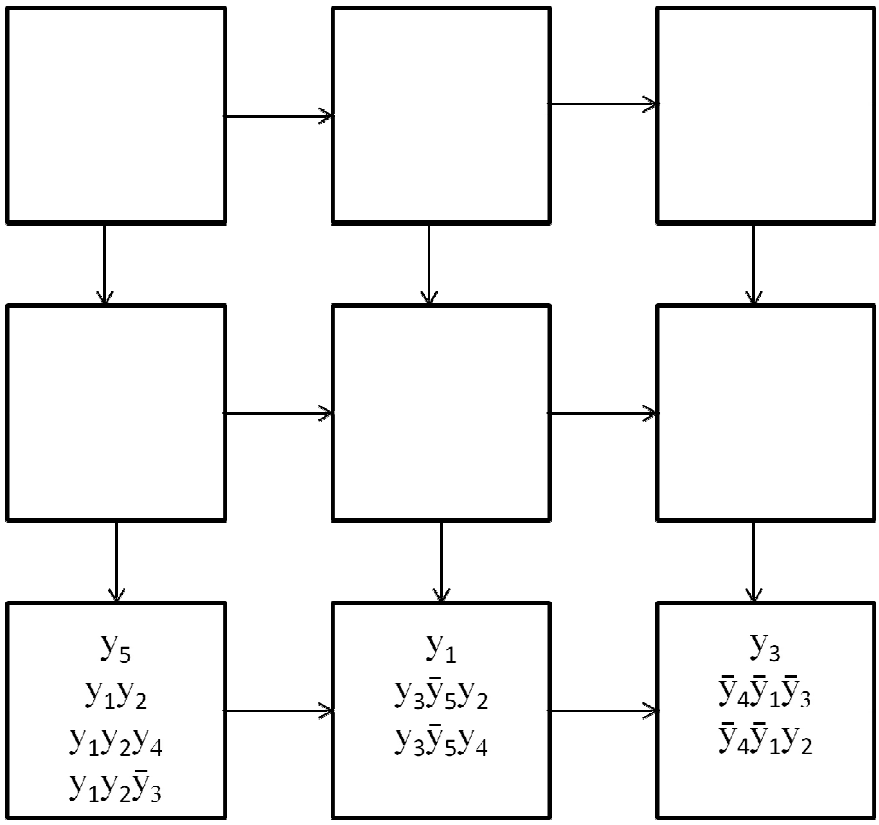


Figure 3.12: Computation of Satisfying Assignment and Shifting Process at Clock Tick 5.

The formed assignments after Clock tick 5 are not shifted since the shift are without wraparound. The satisfying assignments for the formula are collated on the processes on the last row of the mesh and Gather operation can be used to collect all the assignment onto one process. The assignments formed results in the outputs on Table 3.4a. Table 3.4b contains the satisfying assignments after removing the assignments conflicting literals and the literals neutral to previous assignments.

The assignments in Table3.4b are the satisfying assignments for the formula:

$$F=(y_1 \vee y_3 \vee \bar{y}_4) \wedge (\bar{y}_1 \vee y_2 \vee \bar{y}_5) \wedge (y_2 \vee \bar{y}_3 \vee y_4) \wedge (y_3 \vee y_5 \vee y_1) \quad (3.2)$$

Table 3.4a: Computed Assignments after Clock Tick 5

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{2,0}$	$P_{2,1}$	$P_{2,2}$
						$y_1\bar{y}_5y_2y_3$	$y_3\ y_2y_5$	$\bar{y}_4\bar{y}_5\bar{y}_3y_1$
						$y_1\bar{y}_5y_2$	$y_3\ y_2$	$\bar{y}_4\bar{y}_5y_2y_1$
						$y_1\bar{y}_5y_4y_3$	$y_3\ y_2y_1$	$\bar{y}_4\bar{y}_5y_2y_3$
						$y_1\bar{y}_5y_4$	$y_3\ y_2y_4$	$\bar{y}_4y_2y_1$
						$y_1\bar{y}_5\bar{y}_3$	$y_3\ y_2y_4y_5$	$\bar{y}_4y_2y_5$
						$y_1y_2y_3$	$y_3\ y_2y_4y_1$	$\bar{y}_4y_2y_3$
						$y_1y_2y_4y_3$	$y_3\bar{y}_1y_2$	$\bar{y}_4y_2\bar{y}_3y_1$
						$y_1y_2\bar{y}_3y_3$	$y_3\bar{y}_1y_2\ y_5$	$\bar{y}_4y_2\bar{y}_3y_5$
						$y_1y_2y_1$	$y_3\bar{y}_1y_4$	$\bar{y}_4\bar{y}_1\bar{y}_3y_1$
						$y_1y_2y_4y_1$	$y_3\bar{y}_1y_4y_5$	$\bar{y}_4\bar{y}_1y_2y_1$
						$y_1y_2\bar{y}_3y_1$	$y_3\bar{y}_5y_2y_5$	$\bar{y}_4\bar{y}_1\bar{y}_3y_5$
						$y_1y_2y_5$	$y_3\bar{y}_5y_4y_5$	$\bar{y}_4\bar{y}_1y_2y_5$
						$y_1y_2y_4y_5$	$y_3\bar{y}_5y_2y_3$	$\bar{y}_4\bar{y}_1\bar{y}_3y_3$
						$y_1y_2\bar{y}_3y_5$	$y_3\bar{y}_5y_4y_3$	$\bar{y}_4\bar{y}_1y_2y_3$
							$y_3\bar{y}_5y_2y_1$	
							$y_3\bar{y}_5y_4y_1$	

Table 3.4b: Satisfying Assignments after Clock Tick 5

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{2,0}$	$P_{2,1}$	$P_{2,2}$
						$y_1\bar{y}_5y_2y_3$	$y_3\ y_2y_5$	$\bar{y}_4\bar{y}_5\bar{y}_3y_1$
						$y_1\bar{y}_5y_2$	$y_3\ y_2$	$\bar{y}_4\bar{y}_5y_2y_1$
						$y_1\bar{y}_5y_4y_3$	$y_3\ y_2y_1$	$\bar{y}_4\bar{y}_5y_2y_3$
						$y_1\bar{y}_5y_4$	$y_3\ y_2y_4$	$\bar{y}_4y_2y_1$
						$y_1\bar{y}_5\bar{y}_3$	$y_3\ y_2y_4y_5$	$\bar{y}_4y_2y_5$
						$y_1y_2y_3$	$y_3\ y_2y_4y_1$	$\bar{y}_4y_2y_3$
						y_1y_2	$y_3\bar{y}_1y_2$	$\bar{y}_4y_2\bar{y}_3y_1$
						$y_1y_2y_5$	$y_3\bar{y}_1y_2\ y_5$	$\bar{y}_4y_2\bar{y}_3y_5$
						$y_1y_2y_4y_3$	$y_3\bar{y}_1y_4$	$\bar{y}_4\bar{y}_1\bar{y}_3y_5$
						$y_1y_2y_4$	$y_3\bar{y}_1y_4y_5$	$\bar{y}_4\bar{y}_1y_2y_3$
						$y_1y_2y_4y_5$	$y_3\bar{y}_5y_2$	$\bar{y}_4\bar{y}_1y_2y_5$
						$y_1y_2\bar{y}_3$	$y_3\bar{y}_5y_2y_1$	
						$y_1y_2\bar{y}_3y_5$	$y_3\bar{y}_5y_4$	
							$y_3\bar{y}_5y_4y_1$	

We arrived at 38 satisfying assignments as in the sequential case. The satisfying assignments are obtainable in L+C-2 steps which is linear.

3.5 The Proposed Parallel Algorithm

Figure 3.13 presents the proposed parallel version of Bagais' algorithm.

```
Parallel Algorithm
RAij[];          //list of  assignments satisfying (i+1)th clause on Pij process.
RA0j[]:=L0,j;              //assignments satisfying the first clause
P0,j  ◄——RA0j[];       //distribution of  assignments satisfying the first clause
Pi,j  ◄——Li+1,j;     //distribution of literals from remaining C-1 clauses(C1 to Cm-1)
PAij[];      //list of assignments on Pij expecting more literals to be formed with.
NAij[];                        // list of new formed assignments
litShift  :=  1;                // shift counter.
RLij[];                        //list of received literals
CLij;                          //current received literal
L;                             // number of literals in the caluse
C;                             // number of clauses in the formula
a: = L+i;                      // number of iterations by each processing unit.
while (a>0){
        If  (RAij.length !=0)
                If (PAij.length !=0)
                        For k := 1 to PAij.length
                                If PAij[k]*CL did not conflict
                                        If PAij[k]*CL  is neutral
                                                NAij[] := PAij[k]*CL;
                                        Else
                                                NAij[] := PAij[k];
                                        End if
                                End if
                        End for
                End if
                If  RLij.length<L
                        For x := 1 to RAij.length
                                For y := 1 to RLij.length
                                  If RAij[x]*RLij[y] did not conflict
                                        If RAij[x]*RLij[y] agrees
                                                NAij[]:= RAij[x]*RLij[y];
                                        Else
                                                NAij[] := RAij[x];
                                        End if
                                   End if
                                     PAij[]:= RAij[x];
                           End for
                   End for
          Else
```

```
                For x := 1 to RAij.length
                        For y := 1 to L
                                If RAij[x]*RLij[y] did not conflict
                                        If RAij[x]*RLij[y] is neutral
                                                NAij[] := RAij[x]*RLij[y];
                                        Else
                                                NAij[] := RAij[x];
                                        End if
                                End if
                        End for
                End for
                PA.delete;
        End if
        If litShift < L
                rowShift(Lij);
                litShift := litShift+1;
        End  if
        columnShift(NAij[]);
        a := a - 1;
  End if
End while
```

Figure 3.13: Proposed Parallel Version Of Bagais' Algorithm

CHAPTER FOUR: THEORETICAL ANALYSIS

4.0 Introduction

In this chapter, we will derive the models for the expected execution time and incremental memory overhead for the proposed parallel algorithm. First we will analyze the communication and computation costs of the algorithm. Then, we present the expected execution time of the algorithm. Furthermore, we give the analysis of the memory overhead of the algorithm.

The communication and computation cost analysis is based on two-dimensional logical processors (Skjellum 1990). The purpose of the analysis is to provide a relatively simple performance model so that we can use it to determine the characteristics of our parallel algorithm and to guide the experimental test design.

4.1 Communication and Computation Cost Model

Here is the message-passing model. Let α be the startup cost for a single message transmission, which is the cost of initiating a message to be transmitted. Let β (bandwidth) represent the number of data items that can be sent in either row dimension or column dimension of a logical process grid in a unit time so that the cost of sending an assignment containing k literals will be $\frac{k}{\beta}$ and λ be the cost of combining an assignment with a literal. Again the process grid G_{PxQ} has P rows and Q columns. The satisfiability formula is of dimension CxL where C is the number of clauses in the formula and L is the number of literals in a clause. We assume L to be 3 since every k-CNF formula can be converted to 3-CNF formula (Roberto S. 2002).

We discussed the cost of communication for shift operation along row and column dimension and cost of computation along the column dimension. The literals are shifted along the row and formed assignments are shifted down the column of the mesh.

4.1.0 Row Shift Communication Cost

During each shift operation along the row dimension, each process communicates it literal to the next process on the ring. Since we assume the number of literals in each clause to be 3 which is the number of columns of the mesh, the number of shift will be 2 for each process in a row to get all the literals and, in general there will be L-1 shifts. Assignments are not formed along the row dimension. Therefore, the time to send all the literals will be:

$$\alpha + \frac{1}{\beta} + \frac{1}{\beta} = \alpha + \frac{2}{\beta} \qquad (4.1)$$

4.1.1 Column Shift Communication Cost and Formation of Assignment

During each shift operation along the column, each process communicates the formed assignments to the process downward.

For a propositional formula with three clauses, the literals can be mapped on to process grid as shown in Figure 4.0

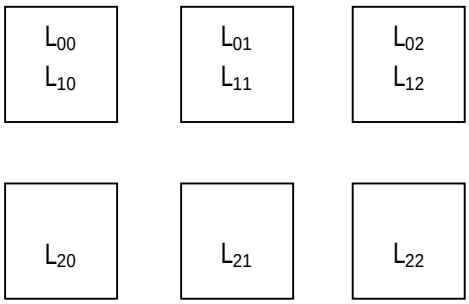


Figure 4.0: Data Mapping of 3 Clauses Boolean Formula onto Process Grid.

The total number of shifts along the column is C where C is the number of clauses in the formula.

During the first shift, the maximum number of assignments communicated is 1, while for the second shift is 1 and for the third shift is also 1. Hence, the time to send all the assignments will be: $\alpha + \frac{2\times1}{\beta} + \frac{2\times1}{\beta} + \frac{2\times1}{\beta} = \alpha + \frac{2}{\beta} + \frac{2}{\beta} + \frac{2}{\beta}$ (4.2)

During the first shift, the maximum number of assignments formed is 1, while for the second shift is 2, for the third shift is 4 and for the fourth it is 3. Hence, the time to form all the assignments will be:

$$\lambda + 2\lambda + 4\lambda + 3\lambda \equiv \lambda + 2\lambda + 3\lambda + 4\lambda \equiv$$
$$\lambda + 3^0 \times 2\lambda + 3^0 \times 3\lambda + 3^0 \times 4\lambda \quad (4.3)$$

For a propositional formula with four clauses, the literals can be mapped on to process grid as shown in Figure 4.1.

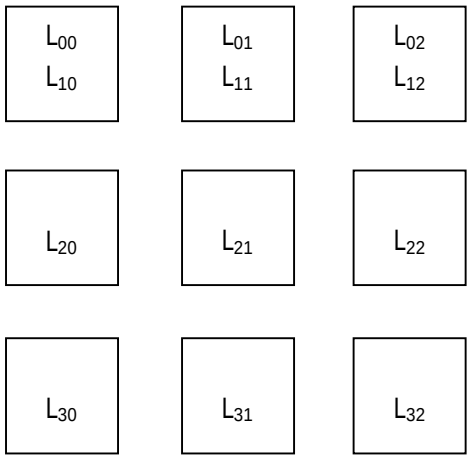


Figure 4.1: Data Mapping of 4 Clauses Boolean Formula onto Process Grid.

During the first shift, the maximum number of assignments communicated is 1, while for the second shift is 2, for the third shift is 4 and for the fourth it is 3. Hence, the time to send all the assignments will be:

$$\alpha+\frac{2}{\beta}+\frac{3\times 2}{\beta}+\frac{3\times 4}{\beta}+\frac{3\times 3}{\beta}\equiv\alpha+\frac{2}{\beta}+\frac{3\times 2}{\beta}+\frac{3\times 3}{\beta}+\frac{3\times 4}{\beta}\equiv$$
$$\alpha+\frac{2}{\beta}+\frac{3^0\times 3\times 2}{\beta}+\frac{3^0\times 3\times 3}{\beta}+\frac{3^0\times 3\times 4}{\beta} \qquad (4.4)$$

During the first shift, the maximum number of assignments formed is 1, while for the second shift is 2, for the third shift is 6, for the fourth it is 12 and for the fifth it is 9. Hence, the time to form all the assignments will be:

$$\lambda+2\lambda+6\lambda+12\lambda+9\lambda\equiv\lambda+2\lambda+6\lambda+9\lambda+12\lambda\equiv$$
$$\lambda+3^0\times 2\lambda+3^1\times 2\lambda+3^1\times 3\lambda+3^1\times 4\lambda \qquad (4.5)$$

For a propositional formula with five clauses, the literals can be mapped on to process grid as shown in Figure 4.2.

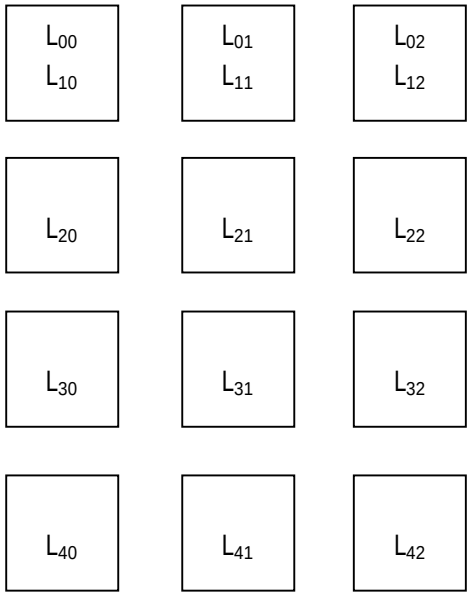


Figure 4.2: Data Mapping of 5 Clauses Boolean Formula onto Process Grid.

During the first shift, the maximum number of assignments communicated is 1, while for the second shift is 2 , for the third shift is 6, for the fourth it is 12 and for the fifth it is 9. Hence, the time to send all the assignments will be:

$$\alpha+\frac{2}{\beta}+\frac{3\times 2}{\beta}+\frac{4\times 6}{\beta}+\frac{4\times 12}{\beta}+\frac{4\times 9}{\beta}\equiv\alpha+\frac{2}{\beta}+\frac{3\times 2}{\beta}+\frac{4\times 6}{\beta}+\frac{4\times 9}{\beta}+\frac{4\times 12}{\beta}\equiv$$
$$\alpha+\frac{2}{\beta}+\frac{3\times 3^0\times 2}{\beta}+\frac{4\times 3^1\times 2}{\beta}+\frac{4\times 3^1\times 3}{\beta}+\frac{4\times 3^1\times 2}{\beta} \qquad (4.6)$$

During the first shift, the maximum number of assignments formed is 1, while for the second shift is 2 , for the third shift is 6, for the fourth it is 18, for the fourth it is 36 and for the fifth it is 27. Hence, the time to form all the assignments will be:

$$\lambda + 2\lambda + 6\lambda + 18\lambda + 36\lambda + 27\lambda \equiv \lambda + 2\lambda + 6\lambda + 18\lambda + 27\lambda + 36\lambda \equiv$$
$$\lambda + 3^0 \times 2\lambda + 3^1 \times 2\lambda + 3^2 \times 2\lambda + 3^2 \times 3\lambda + 3^2 \times 4\lambda \qquad (4.7)$$

For a propositional formula with six clauses, the literals can be mapped on to process grid as follows.

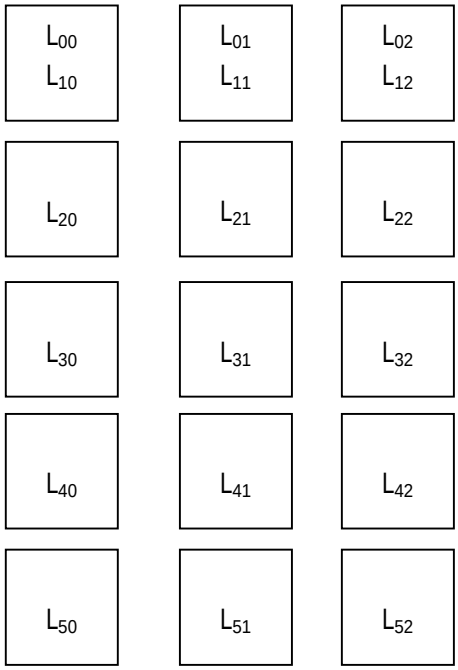


Figure 4.3: Data Mapping of 6 Clauses Boolean Formula onto Process Grid.

During the first shift, the maximum number of assignments communicated is 1, while for the second shift is 2, for the third shift is 6, for the fourth it is 18, for the fifth it is 36 and for the sixth it is 27. Hence, the time to send all the assignments will be:

$$\alpha + \frac{2}{\beta} + \frac{3\times 2}{\beta} + \frac{4\times 6}{\beta} + \frac{5\times 18}{\beta} + \frac{5\times 36}{\beta} + \frac{5\times 27}{\beta} \equiv \alpha + \frac{2}{\beta} + \frac{3\times 2}{\beta} + \frac{4\times 6}{\beta} + \frac{5\times 18}{\beta} + \frac{5\times 27}{\beta} + \frac{5\times 36}{\beta} \equiv$$
$$\alpha + \frac{2}{\beta} + \frac{3\times 3^0\times 2}{\beta} + \frac{4\times 3^1\times 2}{\beta} + \frac{5\times 3^2\times 2}{\beta} + \frac{5\times 3^2\times 3}{\beta} + \frac{5\times 3^2\times 4}{\beta} \qquad (4.8)$$

During the first shift, the maximum number of assignments formed is 1, while for the second shift is 2 , for the third shift is 6, for the fourth it is 18, for the fifth it is 54, for the sixth it is 108 and for the seventh it is 81. Hence, the time to form all the assignments will be:

$$\lambda + 2\lambda + 6\lambda + 18\lambda + 54\lambda + 108\lambda + 81\lambda \equiv$$
$$\lambda + 2\lambda + 6\lambda + 18\lambda + 54\lambda + 81\lambda + 108\lambda \equiv$$
$$\lambda + 3^0 \times 2\lambda + 3^1 \times 2\lambda + 3^2 \times 2\lambda + 3^3 \times 2\lambda + 3^3 \times 3\lambda + 3^3 \times 4\lambda \qquad (4.9)$$

For a propositional formula with seven clauses, the literals can be mapped on to process grid as Shown in Figure 4.4.

During the first shift, the maximum number of assignments communicated is 1, while for the second shift is 2, for the third shift is 6, for the fourth it is 18, for the fourth it is 54, for the fourth it is 108 and for the fifth it is 74. Hence, the time to send all the assignments will be:

$$\alpha + \frac{2}{\beta} + \frac{3\times 2}{\beta} + \frac{4\times 6}{\beta} + \frac{5\times 18}{\beta} + \frac{6\times 54}{\beta} + \frac{6\times 108}{\beta} + \frac{6\times 74}{\beta} \equiv$$
$$\alpha + \frac{2}{\beta} + \frac{3\times 2}{\beta} + \frac{4\times 6}{\beta} + \frac{5\times 18}{\beta} + \frac{6\times 54}{\beta} + \frac{6\times 108}{\beta} + \frac{6\times 74}{\beta} \equiv$$
$$\alpha + \frac{2}{\beta} + \frac{3\times 3^0\times 2}{\beta} + \frac{4\times 3^1\times 2}{\beta} + \frac{5\times 3^2\times 2}{\beta} + \frac{6\times 3^3\times 2}{\beta} + \frac{6\times 3^3\times 3}{\beta} + \frac{6\times 3^3\times 4}{\beta} \qquad (4.10)$$

From equations (4.2) to (4.10), a general expression for communication and computation cost along the column dimension respectively, given a formula with C clauses can be obtained as follows:

$$\alpha+\frac{2}{\beta}+\frac{3\times3^{0}\times2}{\beta}+\frac{4\times3^{1}\times2}{\beta}+\frac{5\times3^{2}\times2}{\beta}+\frac{6\times3^{3}\times2}{\beta}+\frac{7\times3^{4}\times2}{\beta}+\frac{8\times3^{5}\times2}{\beta}+\ldots+$$
$$\frac{(C-2)\times3^{C-5}\times2}{\beta}+\frac{(C-1)\times3^{C-4}\times2}{\beta}+\frac{(C-1)\times3^{C-4}\times3}{\beta}+\frac{(C-1)\times3^{C-4}\times4}{\beta} \quad (4.11)$$

$$\lambda+3^{0}\times2\lambda+3^{1}\times2\lambda+3^{2}\times2\lambda+3^{3}\times2\lambda+3^{4}\times2\lambda+3^{5}\times2\lambda+\ldots+3^{C-4}\times2\lambda+$$
$$3^{C-3}\times2\lambda+3^{C-3}\times3\lambda+3^{C-3}\times4\lambda \quad (4.12)$$

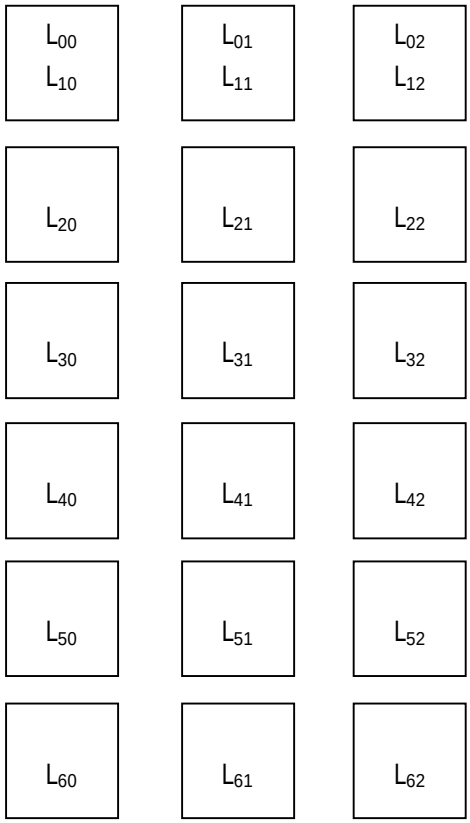


Figure 4.4: Data Mapping of 7 Clauses Boolean Formula onto Process Grid.

Equation (4.11) can be further simplified as follows:

$$\alpha+\frac{2}{\beta}+\frac{3\times3^{0}\times2}{\beta}+\frac{4\times3^{1}\times2}{\beta}+\frac{5\times3^{2}\times2}{\beta}+\frac{6\times3^{3}\times2}{\beta}+\frac{7\times3^{4}\times2}{\beta}+\frac{8\times3^{5}\times2}{\beta}+\ldots+$$
$$\frac{(C-2)\times3^{C-5}\times2}{\beta}+\frac{(C-1)\times3^{C-4}\times2}{\beta}+\frac{(C-1)\times3^{C-4}\times3}{\beta}+\frac{(C-1)\times3^{C-4}\times4}{\beta}$$
$$=\alpha+\frac{2}{\beta}+\frac{2}{\beta}\left[3\times3^{0}+4\times3^{1}+5\times3^{2}+6\times3^{3}+7\times3^{4}+8\times3^{5}+\ldots+(C-2)\times3^{C-5}+(C-1)\times3^{C-4}\right]$$
$$+\frac{(C-1)\times3^{C-4}\times3}{\beta}+\frac{(C-1)\times3^{C-4}\times4}{\beta}$$
$$=\alpha+\frac{2}{\beta}+\frac{2}{\beta}\sum_{i=0}^{C-4}3^{i}\left(3+i\right)+\frac{(C-1)\times3^{C-4}\times3}{\beta}+\frac{(C-1)\times3^{C-4}\times4}{\beta}$$
$$=\alpha+\frac{2}{\beta}+\frac{2}{\beta}\sum_{i=0}^{C-4}3^{i}\left(3+i\right)+\frac{(C-1)\times3^{C-4}\times7}{\beta}$$
$$=\alpha+\frac{1}{\beta}\left[2+(C-1)\times3^{C-4}\times7+2\sum_{i=0}^{C-4}3^{i}\left(3+i\right)\right] \qquad (4.13)$$

The communication model is obtained by adding equations (4.1) and (4.13).

$$T_{communication}=\left[\alpha+2/\beta\right]+\alpha+\frac{1}{\beta}\left[2+(C-1)\times3^{C-4}\times7+2\sum_{i=0}^{C-4}3^{i}\left(3+i\right)\right]$$
$$=2\alpha+\frac{1}{\beta}\left[4+(C-1)\times3^{C-4}\times7+2\sum_{i=0}^{C-4}3^{i}\left(3+i\right)\right];C\geq4$$
$$T_{communication}=\left[\alpha+\frac{2}{\beta}\right]+\left[\alpha+\frac{6}{\beta}\right]$$
$$=2\alpha+\frac{8}{\beta}$$
$$=2\left[\alpha+\frac{4}{\beta}\right];C=3$$
$$T_{communication}=\begin{cases}2\left[\alpha+\frac{4}{\beta}\right];C=3\\ 2\alpha+\frac{1}{\beta}\left[4+(C-1)\times3^{C-4}\times7+2\sum_{i=0}^{C-4}3^{i}\left(3+i\right)\right];C\geq4\end{cases} \qquad (4.14)$$

The computation model is obtained by equation (4.12)

$$
\begin{aligned}
T_{computation} &= \lambda + 3^0 \times 2\lambda + 3^1 \times 2\lambda + 3^2 \times 2\lambda + 3^3 \times 2\lambda + 3^4 \times 2\lambda + 3^5 \times 2\lambda + \ldots + 3^{C-4} \times 2\lambda + \\
&3^{C-3} \times 2\lambda + 3^{C-3} \times 3\lambda + 3^{C-3} \times 4\lambda \\
&= \lambda + 2\lambda \left[3^0 + 3^1 + 3^2 + 3^3 + 3^4 + 3^5 + \ldots + 3^{C-4} + 3^{C-3} \right] + 3^{C-3} \times 7\lambda \\
&= \lambda + 2\lambda \left[\frac{3^0 \left(3^{C-2} - 1 \right)}{3-1} \right] + 3^{C-3} \times 7\lambda \\
&= \lambda + \lambda \left[3^{C-2} - 1 \right] + 3^{C-3} \times 7\lambda \\
&= \lambda \left[1 + 3^{C-2} - 1 \right] + 3^{C-3} \times 7\lambda \\
&= 3^{C-2} \times \lambda + 3^{C-3} \times 7\lambda \\
&= 3^{C-3} \times 10\lambda
\end{aligned}
\tag{4.15}
$$

The time execution cost of the parallel algorithm is the sum of the communication and computation cost.

$$
\begin{aligned}
T_{execution} &= T_{communication} + T_{computation} \\
&= \begin{cases} 2\left[\alpha + \frac{4}{\beta} \right] + 3^{C-3} \times 10\lambda ; C = 3 \\ 2\alpha + \frac{1}{\beta} \left[4 + (C-1) \times 3^{C-4} \times 7 + 2 \sum_{i=0}^{C-4} 3^i \left(3 + i \right) \right] + 3^{C-3} \times 10\lambda ; C \geq 4 \end{cases}
\end{aligned}
\tag{4.16}
$$

4.2 Memory Cost Model

Each process requires $\Theta(C \times L^{C-1})$ memory. The process on the earlier rows of the mesh requires less $\Theta(C \times L^{C-1})$ memory. The total memory requirement increase down the row of the process mesh.

CHAPTER FIVE: SUMMARY, CONCLUSION AND RECOMMENDATION FOR FUTURE WORK

5.1 Summary

In this work, we implemented Bagais' Algorithm which is independent of the nature of Boolean formula for efficiency. The algorithm belongs to the class of complete search procedures which determine whether the given formula is satisfiable or not. The algorithm keeps all the satisfying assignments of previous clauses before moving forward thereby avoiding backtracking which is the case of other algorithms in the literature.

We also proposed a parallel version of the algorithm which can be implemented on two-dimensional process grid topology. We discussed data partitioning and communication steps of the algorithm. Furthermore, we illustrated the correctness of the algorithm by trying an example manually which gave the same result as the sequential version.

5.2 Conclusion

In order to understand the characteristics of the proposed algorithm, interprocess communication, computation and memory overhead cost models were developed. The proposed algorithm can make decision in a linear time.

5.3 Recommendation for Future Work

There are several aspects of this research to be continued. The implementation of the proposed parallel algorithm is one. The possibility of blocking the clauses can be investigated. Also, implementation of the proposed algorithm on other parallel architectures such as tree, hypercube, butterfly etc. is also a prospect for future research.

REFERENCES

Bagais, A. (2008). History of Satisfiability Problem and a Proposed Algorithm. Unpublished M.Sc. Thesis. King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia.

Barroso, L.A. Dean, J. and Holzle, U. Web search for a planet: The google cluster architecture. *Micro, IEEE*, 23(2):22-28, 2003.

Basumallik, A. Min, S.J. and Eigenmann, R. (2007). Programming Distributed Memory Systems Using OpenMP. In: *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1-8.

Cook, S.A. (1971). The complexity of theorem-proving procedures. In: *Proceedings of the Third IEEE Symposium on the Foundations of Computer Science*, East Lansing pp. 151-158.

Crawford, J.M. and Baker, A. B. (1997). Experimental results on the application of satisfiability algorithms to scheduling problems. In: *Proceedings of AAAI-94*, Seattle, Washington pp. 1092–1097.

Davis, M. Logemann, G. and Loveland, D. (1962). A machine program for theorem proving. *Communications of the Association for Computing Machinery* 5:394-397.

Goldberg, E. and Novikov, Y. (2002). BerkMin: a fast and robust sat-solver. In: *Proceedings of the Design and Test in Europe Conference* Rotterdam, Netherlands, pp. 142-149.

Goldberg, E.I. Prasad, M.R. and Brayton, R. K. (2001). Using SAT for combinational equivalence checking. In: *Proceedings of the IEEE/ACM Design, Automation and Test in Europe*.

Gordon, Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

http://en.wikipedia.org/wiki/File:Fork_join.svg. Retrieved on 21/12/2010.

Kautz, H.A. and Selman B. (1999). Unifying SAT-based and graph-based planning. In: *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99)*, Stockholm, Sweden.

Larrabee, T. (1992). Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):6-22.

Memik, S.O. and Fallah, F. (2002). Accelerated SAT-based scheduling of control/data flow graphs. In: *Proceedings of 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)* Freiburg (Germany).

Moskewicz, M. Madigan, C. Zhao, Y. Zhang, L. and Malik, S. (2001). Engineering an efficient SAT solver. In: *Proceedings of the Design Automation Conference*, Las Vegas, Nevada pp. 530-535.

Papadimitriou, C. H. (1991). On selecting a satisfying truth assignment. In: *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)* Princeton Univ., NJ, USA., pp. 163-169.

Roberto, S. (2002). SAT: Propositional Satisfiability and Beyond. Dept. of Information and Communication Technologies University of Trento, Italy. [Online]:http://www.inf.ed.ac.uk/teaching/courses/propm/papers/main_lan1.pdf accessed on 21/09/2010.

Shlyakhter, I. Seater, R. Jackson, D. Sridharan, M. and Taghdiri, M. (2003). Debugging overconstrained declarative models using unsatisfiable cores. In: *Proceedings of 18th IEEE International Conference on Automated Software Engineering*, Montreal, Canada. pp. 94–105.

Skjellum, A. (1990). Concurrent dynamic simulation: Multicomputer algorithms research applied to ordinary differential-algebraic process system in chemical engineering. Ph. D, dissertation, California Institute of Technology, Pasadena, CA.

Smith, A. Veneris, A. and Viglas, A. (2004). Design diagnosis using Boolean satisfiability. In: *Proceedings of IEEE Asian-South Pacific Design Automation Conference* IEEE Press Piscataway, NJ, USA.

Sunderam, V.S. (1990). PVM: a framework for parallel distributed computing. Concurrency: Pract. Exper., 2(4):315-339.

Wood, R.G. and Rutenbar, R.A. (1998). FPGA routing and routability estimation via boolean satisfiability. IEEE Transactions on VLSI Systems, 6(2):222–231.

Zhang, H. (1997). SATO: an efficient propositional prover. In: *Proceedings of the International Conference on Automated Deduction (CADE'97)*, Townsville, North Queensland, Australia pp. 272-275.

Zhang, H. Li, D. and Shen, H. (*2004*). A SAT based scheduler for tournament schedules.In: *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004),* Vancouver (BC), Canada pp. *191–196.*

.

APPENDIX A

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAXclause 10000

int get_clause(char *, char *clause[]);

int get_literal(char *literal[][100], int n);

void sort(char *p[], int n);

void print_strings(char *, char *p[], int n);

void print_formula(char *, char *p[], int n);

void print_literals(char *p[][100], int n, int m);

void generate_assign(char *lit, char *array1[], char
*array2[],

char *arrSubsumed[], int size );

int conflictCheck(char assign[], char *lit);

int neutralCheck(char assign[], char *lit);

int subsumptionCheck(char arrSA[], char arrSub[]);

int removeSubsumedAssign(int numSub, int temp);

void print_info(char *);


char *clause[MAXclause];

char *SA[MAXclause];

char *literal[MAXclause][100];

char *array1[MAXclause];

char *array2[MAXclause];

char *sub[MAXclause];

char *arrSubsumed[MAXclause];

char str[10000];

int arrNum, numSub;

FILE *outputFile;
```

```
int main(int argc, char **argv)

{

     int number_of_clause, number_of_literal,
     number_of_assign, i, j, k, l, flag,temp, temp2, t,
     lenAssign;

     char *lit;

     if(argc != 3){

          print_info(argv[0]);

          exit(1);

     }

     number_of_clause = get_clause(argv[1], clause);

     if (number_of_clause < 0)

     {

          puts(" Memory allocation error");

          exit(-1);

     }

     print_formula(argv[2], clause, number_of_clause);

     //sort(clause, number_of_clause);

     //printf("\n%d\n", number_of_clause);


     number_of_literal =
get_literal(literal,number_of_clause);


     //print_literals(literal,number_of_clause,number_of_li
teral);

     temp = number_of_literal+1;

     for (i=0; i<temp; ++i)

     {


     if((SA[i](char*)malloc(strlen(literal[0][i])+1))==NULL
     )

          exit(0);

     strcpy(SA[i],literal[0][i]);
```

```
 }
arrNum =0;
numSub = 0;

for(i=1; i< number_of_clause; ++i)
{
     for(j=0;j<=number_of_literal;++j)
     {
          lit = literal[i][j];
     generate_assign(lit, SA, array2, arrSubsumed,
temp);
     }
     temp2 = numSub;
     numSub = 0;
     for (k=0; k<arrNum; ++k)
     {
     SA[k] = calloc(strlen(array2[k])+1,
sizeof(char));
          strcpy(SA[k],array2[k]);
     }
     temp=arrNum;
     arrNum=0;
}
if (temp > 0)
{
     printf("The output had been written to %s",
argv[2]);
     print_strings(argv[2], SA, temp);
}
else
{
```

```
          printf("The output had been written to %s",
argv[2]);
          fputs("The formula is unsatisfiable",
outputFile);
     }
    fclose(outputFile);
     return 0;
}

int get_clause(char *ifp, char *clause[])
{
     int n=0;
     char *p;
   FILE *inputFile;
   if((inputFile = fopen(ifp, "r")) == NULL)
   {
fprintf(stderr, " Opening of %s for reading failed ", ifp);
perror("because");
     exit(1);
   }

  fgets(str, sizeof(str), inputFile);
  fclose(inputFile);
  p = strtok(str, ",");
  if ((clause[n]=(char *)malloc(strlen(p)+1))==NULL)
     exit(0);
  strcpy(clause[n++], p);

   while (1)
   {
```

```
         /* extract string from string sequence */
          p = strtok(NULL, ",");
          /* check if there is nothing else to extract */
           if (p == NULL)
           {
               break;
            }
          if ((clause[n]=(char
*)malloc(strlen(p)+1))==NULL)
               exit(0);
          strcpy(clause[n++], p);
        }
     return n;
}


int get_literal(char *literal[][100], int n)
{
     char *q;
      int i,j;
     for(i=0;i<n;++i)
     {
      j=0;
     q = strtok(clause[i], " ");
     if ((literal[i][j] = (char
*)malloc(strlen(q)+1))==NULL)
          exit(0);
     strcpy(literal[i][j], q);


          while(1)
          {
```

```
                /* extract string from string sequence */
                    q = strtok(NULL, " ");
          /* check if there is nothing else to extract */
                    if (q == NULL)
                    {
                    break;
                    }
               ++j;
     if ((literal[i][j] =(char*)malloc(strlen(q)+1))==NULL)
            exit(0);
          strcpy(literal[i][j], q);
          }
     }
     return j;
}


void generate_assign(char *lit, char *array1[], char
*array2[], char *arrSubsumed[], int size)
{
     int i, conflictFlag, neutralFlag;
     char temp[100];


     for (i=0; i <size; ++i)
     {
          strcpy(temp, array1[i]);
          conflictFlag=conflictCheck(temp, lit);
          neutralFlag=neutralCheck(temp, lit);
          if (conflictFlag==0)
          {
               if (neutralFlag==1)
```

```
                {
if((array2[arrNum] = (char *)malloc(strlen(temp)+1))==NULL)
     exit(0);
strcpy(array2[arrNum],temp);
if((arrSubsumed[numSub] = (char
*)malloc(strlen(temp)+1))==NULL)
     exit(0);
strcpy(arrSubsumed[numSub],temp);
//printf("%s\n" ,arrSubsumed[numSub]);
++numSub;
}
else
{
if((array2[arrNum] = (char *)malloc(strlen(strcat(temp,
lit))+1))==NULL)
     exit(0);
strcpy(array2[arrNum],temp);
}
++arrNum;
          }
     }
}


void print_formula(char *ofp, char *p[], int n)
{
     int count, temp;
     if((outputFile = fopen(ofp, "w")) == NULL)
     {
     fprintf(stderr, " Opening of %s for writng failed ",
ofp);
          perror("because");
```

```
     exit(1);
     }
fputs("The satisfying assignments for the Boolean Formula F
= ", outputFile);
     for (count =0; count < n; count++)
     {
          temp = count +1;
          fprintf(outputFile,"%s,", p[count]);
     }
     fputs("are:\n\n",outputFile);
 fclose(outputFile);
}


void print_strings(char *ofp, char *p[], int n)
{
     int count, temp;
     if((outputFile = fopen(ofp, "a")) == NULL)
     {
fprintf(stderr, " Opening of %s for reading failed ", ofp);
perror("because");
exit(1);
     }
//fputs("The satisfying assignments for the Boolean Formula
F = \n", outputFile);
     for (count =0; count < n; count++)
     {
          temp = count +1;
          fprintf(outputFile,"%d. %s\t\t", count+1,
p[count]);
          if(temp%3==0)
          fprintf(outputFile,"\n");
```

```
     }
     fprintf(outputFile,"\n");
}

void print_literals(char *p[][100], int n, int m)
{
     int i, j;
     for (i =0; i < n; i++)
          for (j =0; j <= m; j++)
               printf("%s\n", p[i][j]);
}

int conflictCheck(char assign[], char *lit)
{
char *p;
char temp[100];
int i,j, conflictFlag;

conflictFlag = 0;
p = strstr(assign, lit);
if ((p==NULL)&&(*(lit)=='!')){
     i=0;
     while((temp[i]=*(++lit))!='\0')
     {
          i++;
     }
     p = strstr(assign, temp);
     if (p!=NULL)
          conflictFlag =1;
}
```

```
else if((p!=NULL)&&((*(--p)) == '!'))
          conflictFlag=1;
return conflictFlag;
}

int neutralCheck(char assign[], char *lit)
{
     char *p;
     int neutralFlag=0;
     p = strstr(assign, lit);
     if((p!=NULL)&&((*(--p)) != '!'))
          neutralFlag = 1;
        return neutralFlag;
}

int subsumptionCheck(char arrSA[], char arrSub[])
{
  char *p;
  char str1[1000];
  int i, j;
  char *subsumedLit[1000];
  int n, counter, subsumedFlag;
  j=0;
  n=0;
  counter = 0;
  subsumedFlag = 0;
  p = strstr (arrSA, arrSub);
  if((p!=NULL)&&((*(--p)) != '!'))
  {
     subsumedFlag = 1;
```

```
      //puts("yes1\n");
   }
   else
   {
      for(i=0; arrSub[i]!='\0'; i++)
      {
           str1[j++]=arrSub[i];
           if
(isdigit(arrSub[i])&&isalpha(arrSub[i+1])||(arrSub[i+1]=='\
0'))
           {
                str1[j]='\0';
                subsumedLit[n] = (char
*)malloc(strlen(str1)+1);
                strcpy(subsumedLit[n], str1);
                ++n;
                j=0;
           }
      }

      for (i=0; i<n; ++i)
      {
           p = strstr (arrSA, subsumedLit[i]);
           if((p!=NULL)&&((*(--p)) != '!'))
                ++counter;
           else
           {
                break;
           }
      }
        if (counter == n)
```

```
     {
       subsumedFlag = 1;
       //puts("yes2\n");
     }
  }
  return subsumedFlag;
}

int removeSubsumedAssign(int numSub, int temp)
{
int n, flag, i, j;
n =0;
for (i=0;i<numSub;++i)
     {
          for (j=0;j<temp;++j)
          {
          flag = subsumptionCheck(SA[j], arrSubsumed[i]);
               //printf("%s\t%s\n", SA[j], arrSubsumed[i]);
          if((strcmp(SA[j],arrSubsumed[i]) == 0)||(flag ==
0))
               {
                    strcpy(SA[n], SA[j]);
                    //printf("%s\n", SA[n]);
                    ++n;
               }
          }
          temp = n;
          n = 0;
     }
return temp;
```

```
}


void print_info(char *prog_name)

{

     printf("\n%s%s%s\n\n%s\n%s\n\n", "Usage:  ",
prog_name,

     " infile  outfile", "The Boolean Formula will be
obtained from infile ",

     "and satisfying assignments written to outfile.");

}
```