Sheffield Hallam University

SHEFFIELD HALLAM UNIVERSITY

SCHOOL OF ENGINEERING

A GENETIC APPROACH TO THE

BOOLEAN SATISFIABILITY PROBLEM

FOR A GRID COMPUTING ENVIRONMENT

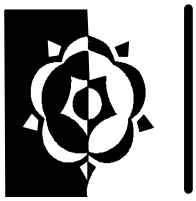
BY

MOHAMMAD SHAHIDUL HASAN

MSc in COMPUTER AND NETWORK ENGINEERING

FULL TIME

2003-2004



Sheffield Hallam University

PREFACE

This report describes project work carried out in the School of Engineering at Sheffield Hallam University between June 2004 and September 2004.

The submission of the report is in accordance with the requirements for the award of the degree of **MSc in Computer and Network Engineering**, under the auspices of the University.

ACKNOWLEDGMENT

It is a matter of great honour for me to complete this thesis successfully as a student of MSc in Computer and Network Engineering of the School of Engineering, Sheffield Hallam University, Sheffield, UK. I express the deepest gratitude to the school and to all my teachers for providing me the required resources and support for this dissertation work.

I like to acknowledge my debt to my supervisor Dr Bala Amavasai for his special attention and inspiration to me. I appreciate his patience for extracting my problems and showing the light of solutions for these obstacles. I gratefully acknowledge the enthusiasm and encouragement of his invaluable advice, support, questions, experience that have been most helpful in ensuring the quality of this dissertation's contents and presentation. I wish to extend a very special thanks to Dr Abusaleh Jabir, Senior Lecturer, Oxford Brookes University, Oxford, UK, for providing guidelines and materials for this project.

Next my appreciation goes to all of my mates and well-wishers for their comments and suggestions. I like to thank specially Asif Rehman, Wasantha for providing all sorts of support towards completion of the dissertation. I express my heartiest apologies to those persons that I may have forgotten to mention.

Finally, I wish to express the deepest respect and appreciation to my parents for their unlimited love, affection, help without which everything would have been impossible.

ABSTRACT

The term “Grid computing” is used to describe an advanced distributed computing environment where the resources and tasks are dynamically assigned to the computing nodes depending on the current load/demand of the entire system. Numerically intensive tasks can be executed faster using low-cost general purpose computers that are converted to run on a grid. This project attempts to improve the solution of the NP complete Boolean Satisfiability (BSAT) problem by partitioning the task into 3 sub-tasks and distributing them over 3 grid nodes for parallel execution. The BSAT problem is of crucial importance in the fields of artificial intelligence, hardware design *etc*. and a faster solution will greatly aid the verification and testing of digital circuits. Two strategies are considered as solutions to the BSAT problem: the brute-force/exhaustive approach and an artificial genetic algorithm (GA) based approach. GAs have been used to consider multiple feasible solutions for the BSAT problem that are consequently refined towards a desired solution, if any exists. Both the algorithms are applied to the standard Boolean satisfiability benchmarks on a single computer configuration and on grid computers using non-optimised and optimised executables. The task is partitioned (coarse grain) and distributed over the grid using Simple Object Access Protocol (SOAP) technology. The results reveal that the grid enabled solution exhibits better performance than single computer for exhaustive search (non-optimised and optimised code) and for non-optimised GA search code. However, no clear correlation could be identified between the single computer and the grid in case of the optimised code GA search. The main contribution of this thesis is the design of a GA based solution to the BSAT problem for a grid computing environment.

NOMENCLATURE

BCP	:	Boolean Constraint Propagation
BDD	:	Binary Decision Diagram
BSAT	:	Boolean Satisfiabilty
CNF	:	Conjunctive Normal Form
CORBA	:	Common Object Request Brokerage Architecture
DCOM	:	Distributed Component Object Modelling
DP method	:	Davis and Putnam's method
DPLL method	:	Davis-Putnam-Logemann-Loveland method
GA	:	Genetic Algorithm
MIMD	:	Multiple-Instruction-Multiple-Data
RMI	:	Remote Method Invocation
SOAP	:	Simple Object Access Protocol
VO	:	Virtual Organisation
XML	:	Extensible Markup Language

TABLE OF CONTENTS

List of figures

Chapter 1:

Chapter 2:

Chapter 3:

Chapter 4:

Chapter 5:

Chapter 1

Introduction

1.1 Introduction

This project explores solutions of the Boolean Satisfiability (BSAT) problem [1] using two methods: the exhaustive search algorithm and an artificial genetic algorithm (GA) [2]. Each approach is implemented in grid computing [3] environment.

1.2 Background

Verification and testing is one of the most important tasks in the production of new VLSI digital circuits. Testing ensures fault free circuits and verification succeeds if the circuits conform to the design specification [4]. Incomplete testing might result in faulty hardware with bugs that can cause serious damage when applied to critical applications. Testing requires a lot of effort and time since verifying a circuit of V inputs involves testing 2^V input combinations which is an NP complete problem [1]. The Boolean Satisfiability (BSAT) [1] problem is one of the most studied NP-complete problems because of its importance in both theoretical research and practical applications [5], especially in the formal verification [6, 7] of hardware design. A distributed solution of the BSAT problem has been explored in this project using a grid of general purpose computers [3].

1.3 Motivation

In a grid computing environment a numerically intensive task is partitioned dynamically among a number of heterogeneous computers that can run in parallel to

obtain better performance [8]. Instead of using special purpose computer and software, a grid can be implemented by combining inexpensive computers and software protocols like Remote Procedural Call (RPC) running under the LINUX operating system. In contrast, a huge amount of research effort has been concentrated on highly expensive specialised hardware for efficient solution of the BSAT problem. Therefore, designing a cost effective solution based on a grid should aid verification and testing of new generation VLSI circuits massively.

1.4 Project description

A genetic algorithm (GA) [2] based BSAT solution has been proposed and implemented on single computer and grid computing environment. The project investigates the performance of the two algorithms: exhaustive/brute force search and genetic algorithm based solution (GA BSAT) on a grid computing environment running under the LINUX Mandrake 10.0 operating system.

1.5 Aims and objectives

The objectives of the project are to

- investigate existing BSAT solutions and grid implementation mechanisms.
- design and implementing a GA based solution to the BSAT problem.
- develop an inexpensive grid system solution for BSAT problem using the LINUX operating system.
- compare the performance between a single computer and grid computing environment.
- investigate execution speed and file size of non-optimised and O3 optimised executables generated by *gcc* compiler.

1.6 Methodology

The grid system consists of a client and multiple servers. In this case, three servers are employed. The client partitions the parent task and distributes the sub-tasks using Remote Procedural Call (RPC) mechanism among the servers running on the grid. This mechanism is depicted in figure 1.1.

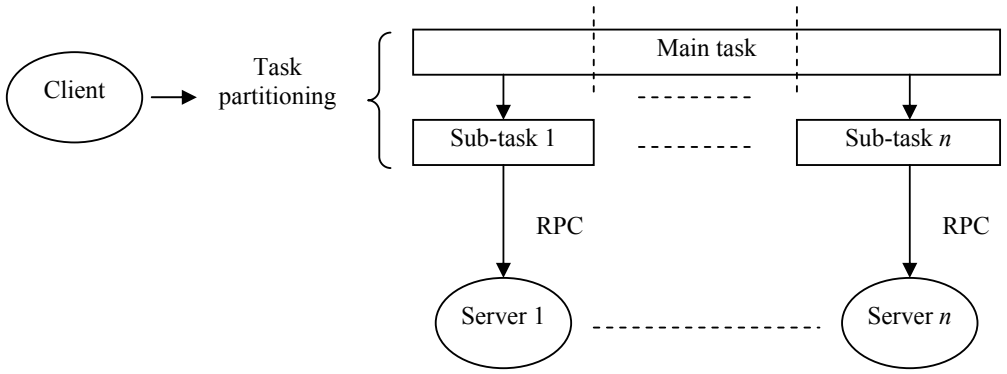


Figure 1.1: Task partitioning and distribution on the grid of computers using RPC.

1.7 Deliverables

If the system is designed correctly, the client should be able to split the parent problem into sub-problems and assign them to servers automatically. The deliverable of the project will be an inexpensive general purpose computer based grid capable of solving BSAT problem in a distributed manner.

1.8 Project formulation

This section discusses constraints such as time, technical limitations *etc*. and develops a work plan for the entire project.

1.8.1 Time/schedule

Table 1.1 shows the definitions of the tasks and timescale of the project.

Task	Time period
Literature survey and investigations of existing techniques	June and July 2004
Design and implementation of algorithms and the overall grid environment	July and August 2004
Report writing	July and August 2004

Table 1.1: Tasks to be completed for the project and their schedule

1.8.2 Technical limitations

This section focuses on various limitations of the project and they are listed below

- SOAP technology has been used to invoke RPC that generates time overheads for marshalling/un-marshalling data into XML format.
- Static task partitioning is used.
- The client follows a coarse grained partitioning approach.

1.8.3 Potential hazards

The precautions that were strictly followed during the entire project period are given below

- Health and safety problems caused by working for long hours on computers without any break.
- Accidental electric shock while connecting computers, switches *etc*. to power supply.

1.9 Report guideline

Chapter 2 explores the concept of grid computing techniques, the BSAT problem, genetic algorithms and remote procedural call technique. The exhaustive search algorithm for BSAT problem is described and a GA based BSAT algorithm is proposed and explained in chapter 3. Coarse grained partitioning of both the exhaustive and the GA BSAT algorithm for grid computing is also discussed in this chapter. Chapter 4 presents the specification and design of the grid system. Implementation details and results of application of the algorithms (the exhaustive and the GA BSAT) on single computer and on grid are discussed in chapter 5. Finally, chapter 6 draws some conclusion.

1.10 Summary

A brief overview of the entire project work has been presented in this chapter. A preview of the mechanisms of implementing the grid has also been discussed.

The next chapter brings all the terminologies, relevant theories, analysis and research works done previously on BSAT, grid, GA *etc*.

Chapter 2

Relevant Theory and Analysis

2.1 Introduction

This chapter explores the basic concepts of grid computing, SOAP technology, the Boolean satisfiability problem and artificial genetic algorithms. Existing grid computing infrastructures, previous works on Boolean satisfiability and artificial genetic algorithm are also discussed in this chapter.

2.2 Grid computing

Grid computing made its appearance as an advanced distributed computing technology in the mid-1990s. It focused on large scale resource sharing for processing intensive applications and encompasses everything from sophisticated networking to artificial intelligence [9]. Nowadays grid computing carries unlimited opportunities in the fields of business and technology and therefore, more and more organisations are moving towards it for solving real-world problems that involve massive computation.

Grid computing can be categorised as a parallel and distributed computing environment that allows dynamic sharing, balancing of resources based on availability, performance, cost and quality-of-service (QoS) of connected autonomous systems [10]. In other words, it is a network of computing resources, tools and protocols with a high degree of coordination in order to share processing-intensive tasks among pooled assets (resources) so as to use them efficiently. These resources can be connected with high-speed LAN, MAN, WAN and distributed

across the continents. They can be heterogeneous *i.e.*, consisting of workstations, servers, mainframes and even supercomputers [9]. These pooled assets are known as Virtual Organisations (VO). Figure 2.1 depicts the concept of grid computing where the VOs are scattered all over the globe [11].

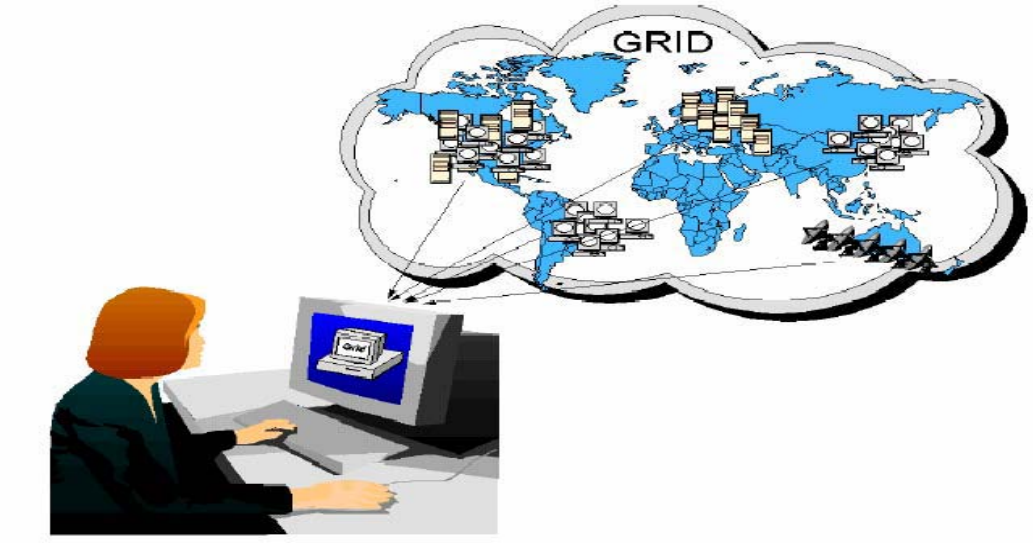


Figure 2.1: The overall structure of grid computing with heterogeneous and geographically disperse virtual organisations.

Grids can deliver scalable, high-performance computing facility equivalent to supercomputing capacity that can be allocated to authenticate users and applications in real time [12]. The grid computing environment can extend from local Ethernet to the Internet and can scale from tens to thousands of server nodes. These nodes must be interconnected with a scalable high-performance network to have the best performance [12]. Figure 2.2 depicts the topology of grids on the Internet and WAN.

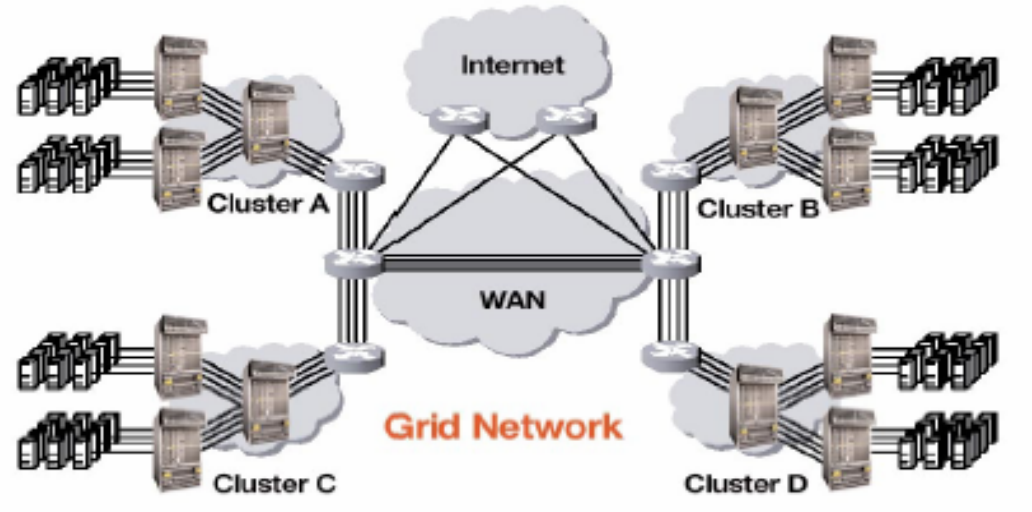


Figure 2.2: Topology of large scale grids

2.2.1 Advantages of grid computing

This section focuses on the special services that a grid can provide over an expensive high-performance computer.

High utilisation of idle resource

Processes can migrate and run on an idle remote machine on the grid when the originating machine becomes overloaded or busy. However, two criteria must be met to allow remote execution [11]

- the process must be executed with the least migration overhead.
- the remote machine must satisfy all types of hardware and software requirements of the migrating process.

Parallel execution on multiple CPUs

By designing applications to support parallelism, a process can be partitioned into independent parts (sub-tasks) and these parts can be executed on different CPUs of

the grid. In an ideal situation, a process can finish *N* times faster if it is distributed among *N* processors as shown in figure 2.3. Nevertheless, the following two points restricts the degree of parallelism

- the algorithm can split a task into a maximum number of subtasks that can create a barrier on the scalability of the grid.
- intercommunication among the subtasks also limits overall performance of parallel execution of the subtasks. For instance, execution efficiency degrades when subtasks access a common database or file.

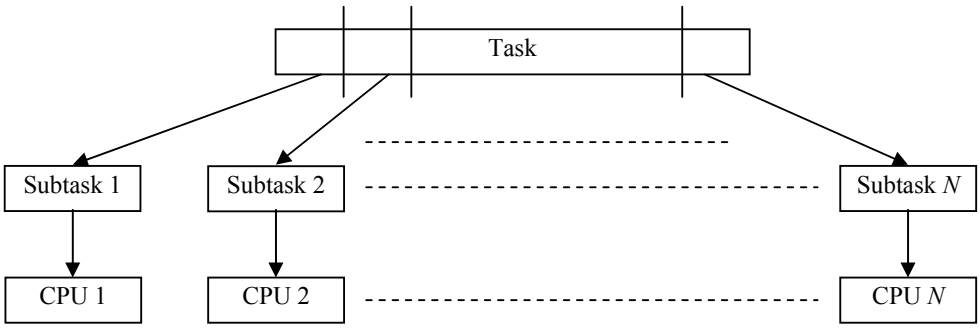


Figure 2.3: Task splitting and parallel execution of subtasks on multiple CPUs.

Collaboration among Virtual Organisations (VOs)

A grid presents a more versatile form of distributed computing that allows heterogeneous systems to work together and produce the image of a large virtual computing system with different types of resources to the user [11]. The users can be associated dynamically to virtual organisations (VOs) depending on the criteria and policy requirements. These VOs on the same grid can share resources in many ways, for instance

- Data sharing: data can be distributed among several systems in form of files and databases to provide more capacity than a single system. Such data distribution technique improves transfer rate by locating the closest data source on the grid.

Furthermore, redundant copies of the same data on different systems ensure reliable data retrieval in case of system failure and supports fault tolerance [11].

- Hardware sharing: A process requiring special type of device (for example, laser colour printer, bar code reader *etc.*) can use the hardware attached to a remote computer. Organisations participating in the grid build up the grid resources and can use other organisation's special resources when they need additional resources.
- Software service, licence sharing: Expensive licensed software service can be installed on some machines of the grid and requests can be sent to these machines to utilise software licenses [11].
- High bandwidth for the Internet: In case of high bandwidth requirement, the load can be split among several grid machines that have independent connections to the Internet [11].
- Security: The grid can enforce security rules/policies in a distributed fashion to protect unauthorised access to the grid. It eliminates single point failure problem.

Resource/load balancing

The grid can be configured to balance resource/load by scheduling grid enabled tasks on machines with low utilisations as depicted in figure 2.4. This technique is very useful to handle occasional peak loads in a larger organisation. However, overload situation can be dealt in the following ways

- a sudden overload of processes can be transferred to relatively low utilised machines in the grid.
- low priority tasks can be suspended temporarily and restarted later if the grid is busy with high priority tasks.

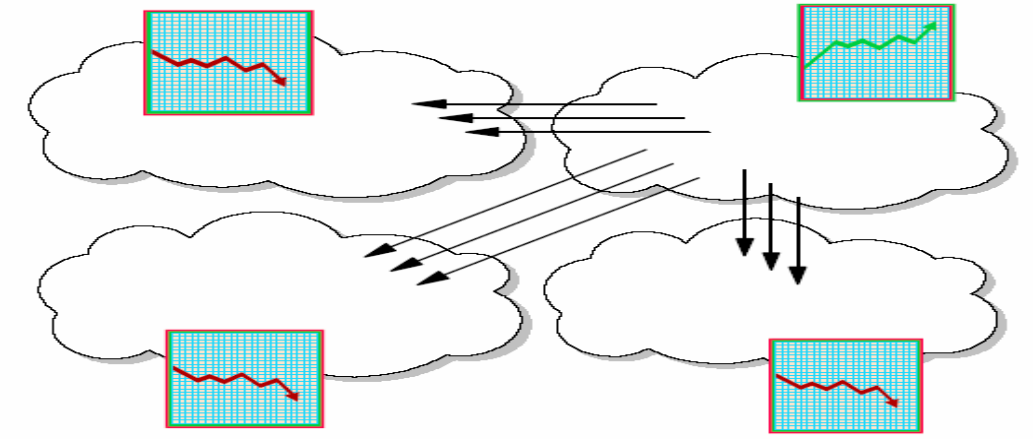


Figure 2.4: Tasks are migrated to less busy parts of the grid to balance resource/loads and improve overall performance

A grid is also suitable for real time tasks with specific deadlines. The task can be split into subtasks and executed on several processors simultaneously if the size and type of the task is known in advance. However, processes running on different processors might need to communicate with each other through the Internet or storage media. Nevertheless, communication traffic/overhead can be minimised by using an advanced scheduler.

Reliability

In case of failure at one site of the grid, the other parts can be designed to continue functioning. Therefore, grid management system can automatically resubmit tasks to other machines on the grid. Furthermore, in critical real-time applications, multiple copies of the same task can be run on different machines throughout the grid, as illustrated in figure 2.5 and the results can be checked for any kind of inconsistency [11].

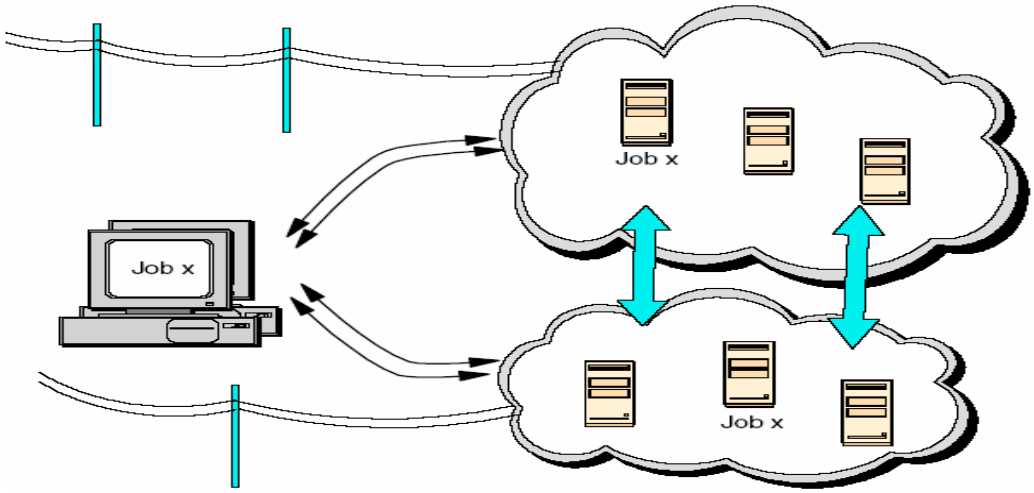


Figure 2.5: Real time critical task/job x is executed on two sites to provide high degree of reliability.

Management

The grid can be viewed as a combined and shared computing environment of several organisations. Administrators can change the policies that affect how the different organisations might share or compete for resources of the grid [11].

In summary, all these features make the grid look like a large virtual machine with a collection of virtual resources of different types.

2.2.2 Classification of grid applications

Grid applications can be categorised into four broad classes based on computational intensity, memory demands, data-locality and inter-task communications requirements [13].

- Loosely Coupled: This class exhibits low memory requirements, small amounts of data and little inter task communication. These are suitable for execution on wide-area clusters connected via low bandwidth networks [13].

- Pipelined: Applications in this class deal with real-time data and the algorithms are often very memory and data intensive. They display coarse-grained inter-task communication. Typical examples in this class are the real-time signal processing and subsequent storage of data captured from satellites, remote sensors including microscopes *etc* [13].
- Tightly Synchronised: This class of applications require frequent inter-task synchronisation and therefore, demands strong communication infrastructure. However, these may have significant data intensive computation. Examples of applications in this class are climate, physics, and molecular models employing explicit iterative methods [13].
- Widely Distributed: Applications in this class search, update, and/or merge distributed databases. Typically these have small computation, data, and memory requirements, but access databases owned by different organisations across the grid environment. [13].

2.2.3 Open source code for grid computing

The open source code technologies that are available for the grid computing are discussed in this section.

N1 Grid Engine open source code

The *N1 Grid Engine* (former *Sun Grid Engine*) is a piece of software for managing distributed computing resources. It dynamically satisfies user's computing resource requirements to the computing resources available [14]. The *N1 Grid Engine* is suitable for cluster grids — one-project, one-department grids. The *N1 Grid Engine* has some functionality of global grids that can span multiple enterprises. However,

the Sun Grid Engine Enterprise Edition from *Sun Microsystems* is suitable for enterprise level grids — multi-project, multi-department grids in a single organisation and has basic functionality for global grids [14]. It runs on Solaris 9, 8, 7, and 2.6 SPARC® operating environment variants and on Sun x86 Linux and Linux x86 [14]. The computing grid master daemon may run on some arbitrary node.

The Globus Project

The *Globus ALLiance* project is a research and development project focused on enabling the application of grid concepts to scientific and engineering computing. It is developing an integrated open-architecture, open-source, grid services implementations called the *Globus Toolkit*. It provides a range of basic services and software libraries to support grids and grid applications. The toolkit includes software for security, information infrastructure, resource management, data management, communication, fault detection and portability [15]. The *Globus toolkit* includes components designed to integrate the distributed hardware of the grid [16]

- Globus Resource Allocation Manager (GRAM): Library service to handle job submission.
- Grid Information Service (GIS): Directory service to locate grid resources. Also known as the metacomputing directory service.
- Grid Security Infrastructure (GSI): A library providing security services.
- GridFTP: File transfer protocol for high bandwidth wide area networks based on FTP. It includes GSI security, multiple data channels, partial file transfers, authenticated data channels and reusable data channels.
- Globus Access to Secondary Sources (GASS): Remote data access component.

Globus also provides a layer above these services to give a simple user interface.

2.2.4 Existing grid infrastructures and previous works on grid

The *North Carolina Research and Education Network (NCREN)* in the US, established in 1985 forms the backbone infrastructure for the state-wide grid that interconnects universities of the state. Through this grid, the University of North Carolina 16-campus system and other *NCREN* customers will offer research and development resources beyond the major metropolitan areas of North Carolina where many of the advanced computing resources already exist [17]. The structure of the grid is shown in figure 2.6.

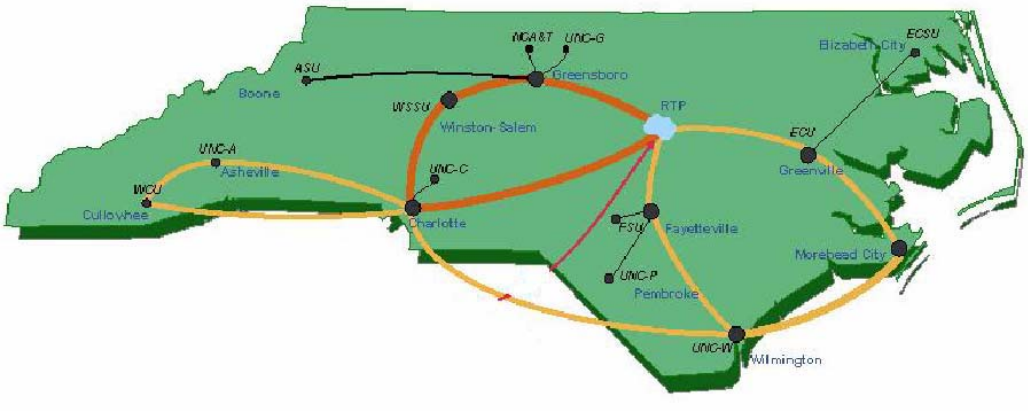


Figure 2.6: Structure of *North Carolina Research and Education Network (NCREN)* grid

Sun Microsystems has taken control of several grid computing projects. Their grid strategy scales from cluster grid to global grid to expand existing technologies and integrate new ones [18], [19]. Figure 2.7 depicts the three levels of grid as planned by Sun.

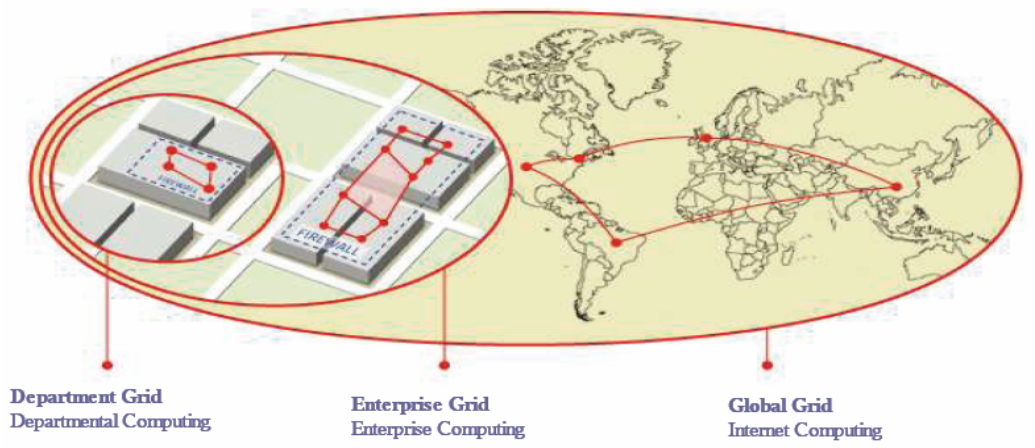


Figure 2.7: Three levels of grid computing: Department grid, Enterprise grid and Global grid.

A grid based on the Ethernet standard has been proposed in [20] to share resources. It is an effective and reliable technique for exploiting coarse-grained parallelism when failures are common. This approach places several simple but important responsibilities on client software to back off during periods of failure and to inform the competing clients in case of resources contention. It is employed to perform several grid computing tasks such as job submission, disk allocation and data replication [20].

2.3 Distributed communication techniques for RPC: SOAP

XML based web technology provides more flexibility and simplicity than the previous distributed computing technologies such as DCOM, CORBA, and RMI. Microsoft, IBM and Sun are already promoting XML web services way to improve business systems through the use of industry standard XML protocols like SOAP, WSDL, UDDI *etc* [21],[22]. Web services can be utilised to

- call an application specific computation on a remote machine using simple web methods.
- perform parallel computing behind the scenes on several computers to have better execution speed than on a single machine.

Therefore, by the careful design of the appropriate back-ends, they can hide any high-performance processing resource – whether it is a supercomputer, a cluster, or even the grid as a whole as shown in Figure 2.8 [22]. By using simple interfaces, computational web services allow users to get computing power as easily as one can get electrical power through a wall socket [22].

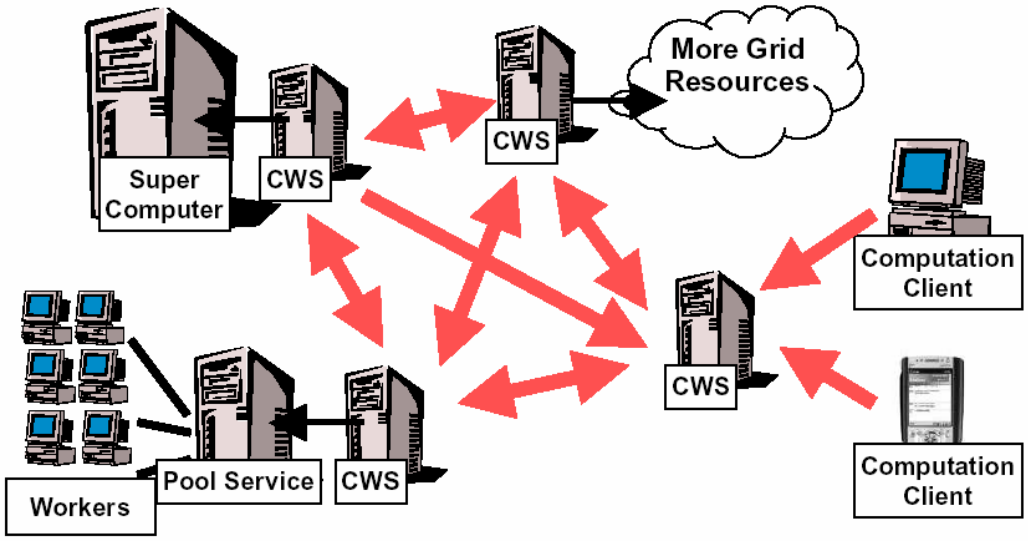


Figure 2.8: A grid based on computational web services (CWS).

The Simple Object Access protocol (SOAP) provides a way to create widely distributed, complex computing environments that can run over the Internet using existing Internet infrastructure [23]. It overcomes the limitation where most firewalls are configured to block non-HTTP request to remote objects [24].

The SOAP technology is language and platform independent through the use of an Extensible Markup Language (XML) [25] scheme. The two participating applications can be written in different languages and can run on different operating systems [26]. The SOAP defines encoding techniques of a set of built-in and users defined data types. It allows the passing of almost any type of data between two applications. However, when it is used to make remote procedural calls (RPCs), it behaves as a request/response protocol. [26]. Figure 2.9 explains the request/response message formats of SOAP. The client wraps a method call into SOAP/XML packet, which is then sent over HTTP to the server. The XML request is parsed to read the method name and parameters passed and delegated for processing at the server side. The XML response is then sent back to the client, containing the result or fault data of the method call. Finally, the client may parse the response XML to make use of the return value [27].

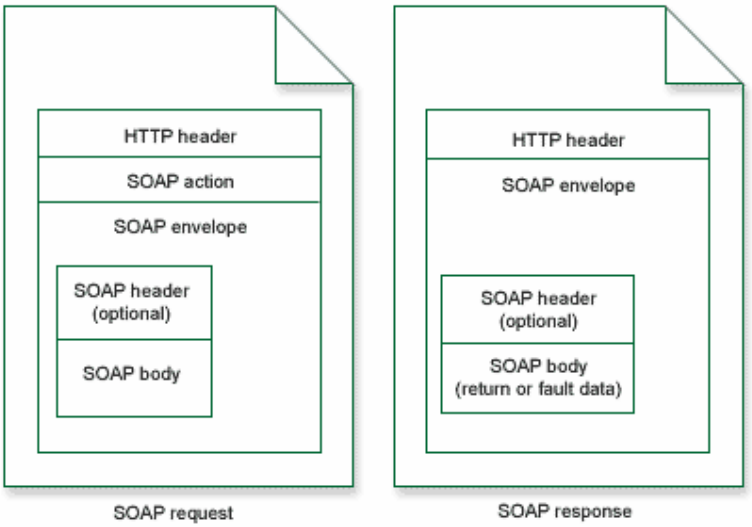


Figure 2.9: SOAP uses the standard HTTP request/response model

Figure 2.10 shows the data flow between client and server. The server runs in listening mode to process SOAP requests. The listener is simply the server code at

the specified URL for parsing the XML request, making the procedure call, and wrapping the result in XML to send as the response to the client [27].

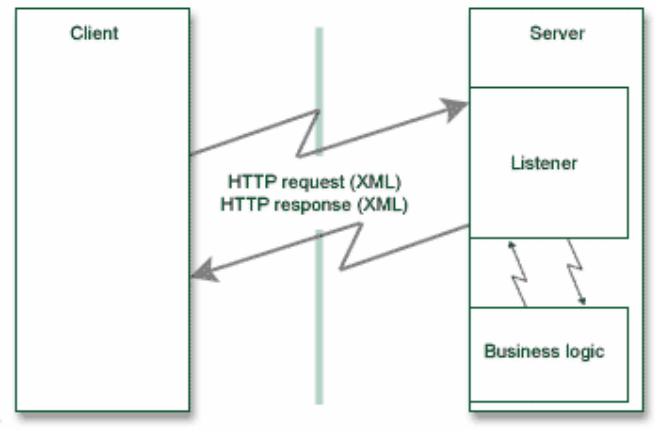


Figure 2.10: Data flow between client and the server of a SOAP call

SOAP specifications define how HTTP is employed to send and receive SOAP messages. Though SOAP messages can be sent over any protocol, HTTP is used in most applications. However, the message format can be extended to support custom applications [26]. Figure 2.11 explains the entire procedure of making a RPC by SOAP using HTTP [26].

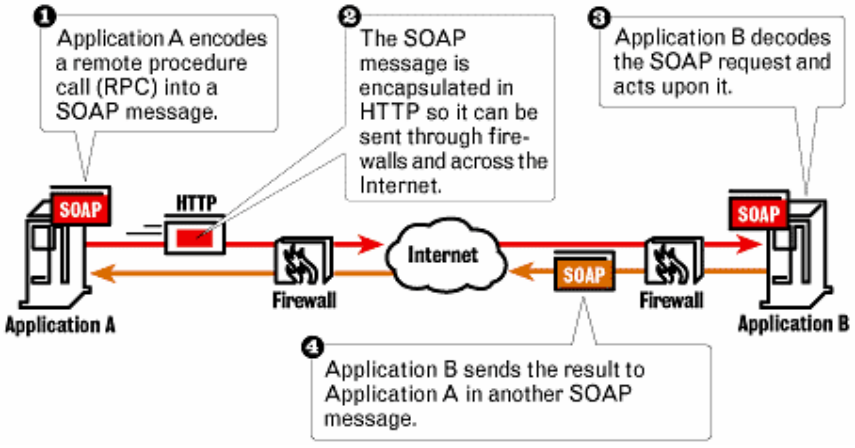


Figure 2.11: Activation of RPC by Application A and return of the result by Application B over the Internet.

2.4 The Boolean satisfiability (BSAT) problem

The BSAT problem [1] is a decision problem where the answer is either TRUE or FALSE. An "instance" is satisfied when the Boolean expression is TRUE for some assignment to the variables [28]. Otherwise it is said to be "unsatisfiable". For a Boolean function of N variables, there exists a total number of 2^N enumerations. So, in the worst case, a brute force/exhaustive method performing linear search will have to consider all 2^N combinations to generate the decision. Due to the complexity of the problem, it will take several years to obtain a solution using the current fastest computer, even for $N = 50$ [29]. The Boolean SAT has a large number of applications in automatic test pattern generation (ATPG) to test digital systems [29], computer architecture, computer aided design, reasoning [1], [30], logic verification, equivalence checking, timing analysis [31] *etc*. Therefore, new techniques are constantly being proposed, either in software or in hardware, to accelerate the solution of SAT [28]. Relevant terminologies and methods are discussed in the following sections.

2.4.1 Definition and Terminology

Any Boolean expression/formula can be transformed to an equivalent satisfiable formula in Conjunctive Normal Form (CNF) in polynomial time [32]. CNF is the most frequently used format for the Boolean satisfiability problem [33].

In CNF, the variables of the formula appear in literals – it can be either a single variable (x) or the negation/complemented form of a single variable (~x). Literals are grouped into clauses, which represent a disjunction (logical OR) of the literals they contain. A single literal can appear in any number of clauses. The conjunction

(logical AND) of all clauses represents the whole formula [33]. For example, the CNF formula

F = (x1 OR ~x2) AND (x3) AND (x2 OR ~x3)

has the following properties-

- 3 variables namely x1, x3 and x3
- 3 clauses that are (x1 OR ~x2), (x3) and (x2 OR ~x3)
- 3 literals are in positive/original form: x1, x3 and x2
- 2 literals are in negative/complemented form: ~x2 and ~x3

It can be noted that a variable assignment that satisfies all the clauses will satisfy this CNF formula. In this case, an assignment x1=1, x2=1 and x3=1 satisfies the formula and hence it is satisfiable.

2.4.2 DP and DPLL methods

The Davis and Putnam's (DP) method can be applied to test if a Boolean formula *F* is satisfiable and it was first described in the paper [34]. Modern general purpose SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) [35], [36] backtracking search approach that eliminate variables by case analysis rather than ordered resolution [37]. These apply a learning mechanism to derive new clauses for representing an abstraction of unsatisfiable parts and this learning mechanism effectively uses a heuristic to implement the backtrack search [38]. However, every DPLL solver exhibits an exponential runtime [38]. The Davis-Putnam procedure [31] is described below.

```
while (true)
{
    if (decide())  {                                    // branching
        while (deduce ()==conflict) {                   //BCP
            backtrack_level = analyse_conflicts();      // conflict analysis
            if (backtrack_level==0)
                return UNSATISFIABLE
            else
                back_track (backtrack_level);           // backtrack
        }
    }
    else                                                // no unassigned variables
        return SATISFIABLE
}
```

The decide() procedure chooses a variable that has not been assigned yet. Decisions are mostly based on heuristics and it can affect the performance up to some extent [39]. After each decide() call, the decision level is increased by one.

The deduce() is the inference process also known as Boolean Constraint Propagation (BCP) [38]. It extends the current assignment by following the logic consequence of the assignments made so far. If all literals in a clause are false then a conflict is reached. If all but one literal in a clause are false, then the clause is called a *unit clause*. Similarly the remaining literal is called a unit literal. Undoubtedly, in order to make the clause true, the unit literal must be true. This is called an *implication*. Thus deduce() is to identify unit clauses and get the corresponding implications or find a conflict [31].

The analyse_conflicts() function is used to detect decisions which lead to the conflict and avoid making a wrong decision again. This is called *conflict-based learning* – obtaining knowledge about the decisions that will lead to immediate conflict. [31].

The back_track() function is used to undo the latest assignment that caused the conflict. All the implications of these assignments are also invalid because of the conflict. However, it is apparent that not all the assignments made so far are responsible for the conflict and some decision levels can be skipped during backtracking. This is called *non-chronological backtracking* [31].

The following points must be considered carefully while implementing DP or DPLL methods

Boolean Constraint Propagation

A large amount of execution time is spent on the BCP. So, an efficient implementation of the BCP is vital to the performance. BCP basically does two things: first, identify unit clauses, hence unit literal; second, learning the implication or report a conflict. The straightforward strategy is checking each clause to identify if the clause is unit clause or not for any assignments. But this is a very inefficient method since most SAT problem database involves memory and accessing large memory will slow down execution speed. Execution speed can be improved by avoiding the clauses with two or more literals not false that means these literals are either true or currently unknown [31].

Decision Heuristic

A SAT solver has two major concerns in the decision heuristic. The first one is which variable to choose, *i.e.*, variable ordering and the second one is what value to assign the first [31].

- Variable ordering: This problem is resolved by a greedy approach based on the frequency of the variables. A counter is associated with each variable to record the number of times that the variable appears in the current clause and the first variable with the maximum counter value is chosen [31]
- Choosing value(s): Since the clauses are in OR-ed form of variables, values should be assigned in such a way that at least one literal is true in each clause [31].

Conflict Analysis & Non-Chronological Backtracking

The most recent relevant decision level is the proper backtrack level and it indicates that all decisions below that level will lead to conflicts regardless of the decisions made. This can greatly reduce the search space and consequently improve the performance [31].

Binary Decision Diagrams (BDD)

Binary decision diagrams (BDDs) [40] have also been widely used in Computer Aided Design (CAD) applications, for instance, logic synthesis, testing and formal verification [41]. This strategy transforms a circuit into a canonical form (CF), depending on an ordering of the Boolean variables. Two circuits are considered to be equivalent if and only if they have the same canonical form. For many kinds of circuits, BDDs work very well, especially when a good ordering of the variables can be found [37]. Equivalence checking of two circuits [42], [43] is of significant importance so that new or optimised circuit can be verified by showing that it is equivalent to an old and tested circuit [37]. However, satisfiability solvers based on Davis and Putnam method are more efficient than BDDs when there is limited

backtracking [37]. BDDs make use of an ordering of the variables which breaks the processing down into smaller steps that are easier to perform and thus can handle large formulae.

2.4.3 Previous works

An efficient implementation of DPLL can be found in [44]. [45] proposed a solution to verification problems by combining BDDs and satisfiability testers [37]. A method discussed in [46] tries all possible truth assignments to small subsets of the variables of a formula using breadth first search. Then the information, obtained about dependencies among the variables, from these assignments can be utilised to determine satisfiability.

Chaff's [47] algorithm based on DPLL follows a depth-first traversal through the decision tree where each node is a value assignment for a particular decision variable. The decision level of an assignment is the length of the path from the root to that assignment [38]. Chaff's algorithm demonstrates 10-100x speed up compared to all previous software solutions and solutions can be found in reasonable computing time using SAT software packages running on general purpose processors [48].

A satisfiability procedure Q_{SAT} is discussed in [37] that replaces sub-formulae by simpler equivalent sub-formulae repeatedly. It tests satisfiability of a formula by successively eliminating variables from it, producing an equivalent formula, until all variables have been eliminated [37].

A new parallel algorithm MP_SAT has been proposed in [48] that uses the fine grain parallelisms in the clause and variable operations. It speeds up SAT solver performance by exploiting the following points —

- efficient single processor SAT algorithms like Chaff.
- configurable processor cores that provide a practical low-cost alternative for custom processor design.
- integrated processor and DRAM chip
- Multiple-Instruction-Multiple-Data (MIMD) stream architecture

MP_SAT uses a decomposition strategy for both data and function. Computationally expensive functions in all SAT algorithms repeatedly perform the same operations on a large set of data. Furthermore, there is no strong correlation among the data. Therefore, each processor can be assigned a subset of the clauses, variables and runs the functions on its own data subset in parallel as shown in figure 2.12 [48].

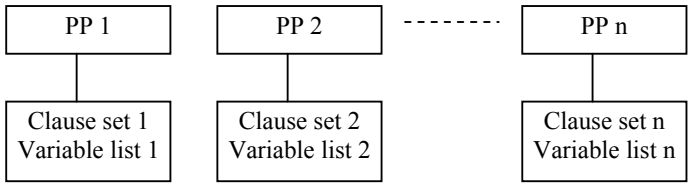


Figure 2.12: Task partitioning among several processors.

Figure 2.13 shows the overall architecture with processing nodes arranged in a two-dimensional mesh. Each node contains processor and communication hardware. Processors have in-built floating point cache and DRAM. The communication part (com) performs message routing and buffering. However, global synchronisation is necessary before MP_SAT makes a new decision. It must make sure that all the PPs

have completed with the BCP at the current level. Therefore, the MP needs to detect whether this condition has satisfied from time to time [48].

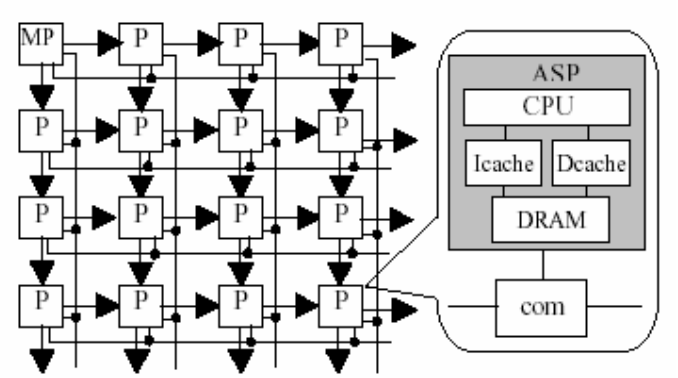


Figure 2.13: System architecture with embedded DRAM in processor chip.

The sequential DPLL algorithm is applied in parallel fashion by partitioning the entire search space into several disjoint parts and treating these in parallel [33]. However, for a SAT search space, it is difficult to predict the required time to explore a particular branch of the search space and therefore, it is not possible to partition the search space at the beginning statically. In [33], the problem is resolved by dynamic partitioning and assigning work load to the available threads at run-time. The partitioning is performed by the concept of *guiding path*. It associates a Boolean value to the variables and flag to indicate either both the values or one value has been assigned. Variables that have been assigned both values are called "*closed*" and those for which one value assignment has been performed is said to be "*open*". These open variables represent junctions in the guiding path for unexplored search path. Therefore, another thread (called *child thread*) can start execution by flipping the value of the open variable and marking it as closed to stop another thread to start from this variable [33]. The thread that follows the main search path is the parent

thread and each thread searches only one path. The whole concept is depicted in figure 2.14.

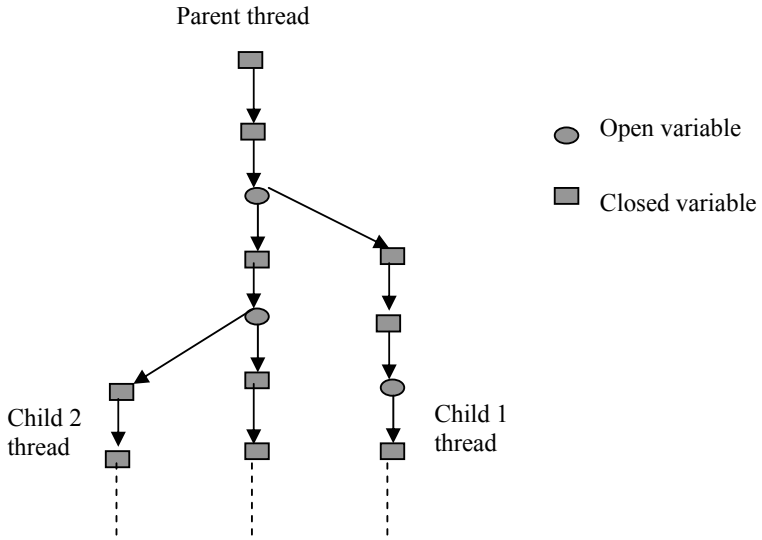


Figure 2.14: Parallel threaded Implementation of DPLL algorithm.

[49] has proposed an algorithm derived from Chaff [47] for grid application. The computational grid provided by the Grid Application Development Software (GrADS) project [50] has been used to apply it on the SAT2002 benchmark [51], [52]. The algorithm is able to solve previously unsolved problems of the benchmark suit using the machines of GrADS located at various institutions of the United States [49]. The SAT problem is split into independent sub-problems that can be investigated for satisfiability. These sub-problems can be partitioned further in the similar fashion to form a recursive tree of problems. A new sub-problem consists of a set of variable assignments and a set of clauses. Variable assignments include all the assignments of the first level and complement of the first assignment of the second decision level and so on. Learned clauses from one machine are shared by the others so that single learning can propagate through all over the grid [49]. However, a learned clause can result in one of four cases:

- If the clause has only one unknown literal then it results in an implication.
- If the clause has more than one unknown literal then the clause is simply added to the set of learned clauses.
- If the clause has all literals false then there is a conflict and the sub-problem is unsatisfiable.
- If the clause evaluates to true then the clause is discarded since it does not prune any part of the search space.

2.5 Genetic algorithm

Genetic algorithms (GA) attempt to solve complex problems by modelling Darwin's theory of evolution where solutions of a particular problem are allowed to evolve over time. GAs are widely used for optimised searching [53]. A fitness function is applied to judge the eligibility of the probable solutions. Various aspects of GAs and suitablity of fitness functions can be found in [54].

2.5.1 Definition and terminologies

- Chromosome: GAs consider simultaneous multiple solutions and each solution is called a chromosome. The target of a GA is to produce new chromosomes (solutions) that are better than the parent chromosomes.
- Gene: Each chromosome contains a number of genes and each gene carries one or a number of properties. However, genes are generally represented by a bit.
- Population: The nunber of solutions in a generation is called the population of the generation.

The GA algorithm is shown in the figure 2.15.

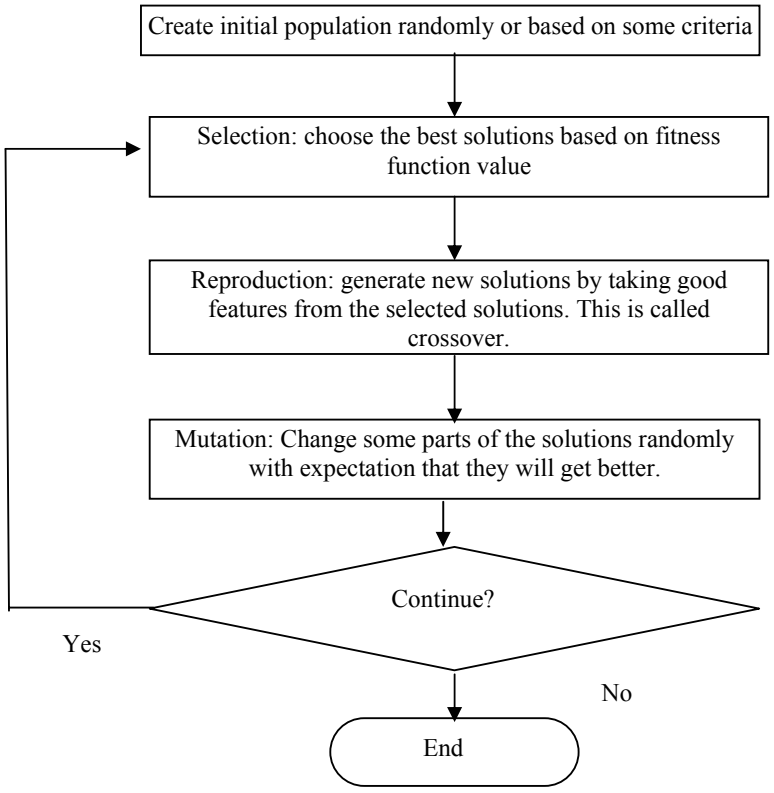


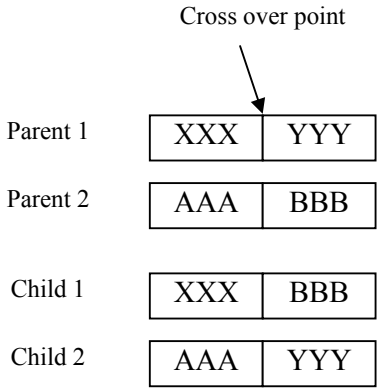
Figure 2.15: Flowchart of generic Genetic Algorithm

2.5.2 Selection

The selection mechanism chooses the best parents from the current generation to generate children for the next generation. Suitable parents are chosen based on the fitness value of the parents. The simplest form of selection is the roulette wheel selection where each solution is allocated a section of roulette wheel proportioned to its fitness. The wheel is spun a number of times and the solution landed is picked to form part of a new generation. This solution has survived to reproduce.

2.5.3 Reproduction

As in nature, reproduction is a mechanism that generates a child that carries properties of its parents. In case of GAs, the new child solution inherits the properties/data of the parent solutions. The process of generating new chromosome from parent chromosomes is called cross over. Figure 2.16 explains single point and multipoint cross over process. Unfortunately, it only works if the parents (patterns) are sufficiently differently.



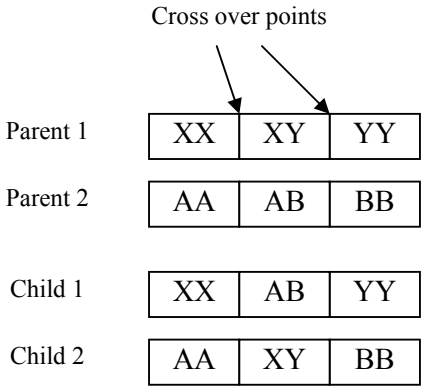


Figure 2.16a: Single point cross over. Figure 2.16b: Multi-point (two) cross over.

2.5.4 Mutation

To incorporate new properties in new solution, chromosomes have a small probability of changing. Since chromosomes are represented as bit strings, mutation simply means flipping a bit, *i.e.*, changing a bit from 0 to 1 and vice versa at random location. Figure 2.17 shows mutation in 4^{th} and 8^{th} bit places from left.

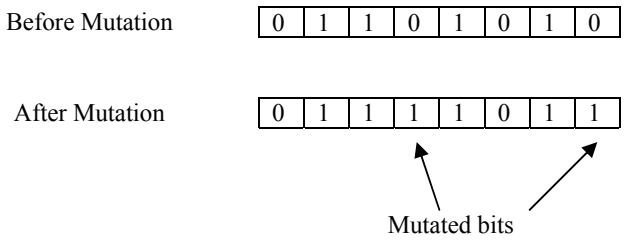


Figure 2.17: Mutation of a chromosome.

2.5.5 Previous GA works on BSAT

A fuzzy-genetic approach to BSAT problem is presented in [29] that uses fuzzy logic [55] , [56] to assign fitness to chromosomes/feasible solutions of the search space. The original binary domain {0, 1} is mapped into continuous fitness domain [0, 1] by fuzzy logic. GA is utilised to optimise the solution in the continuous domain and finally the derived solution is converted back (decoded) to the Boolean format. The entire process is depicted in the figure 2.18. However, it can be noted that the binary world limits the vertices to a unit hypercube. On the other hand, fuzzy domain involves any interior point of the unit hypercube [29].

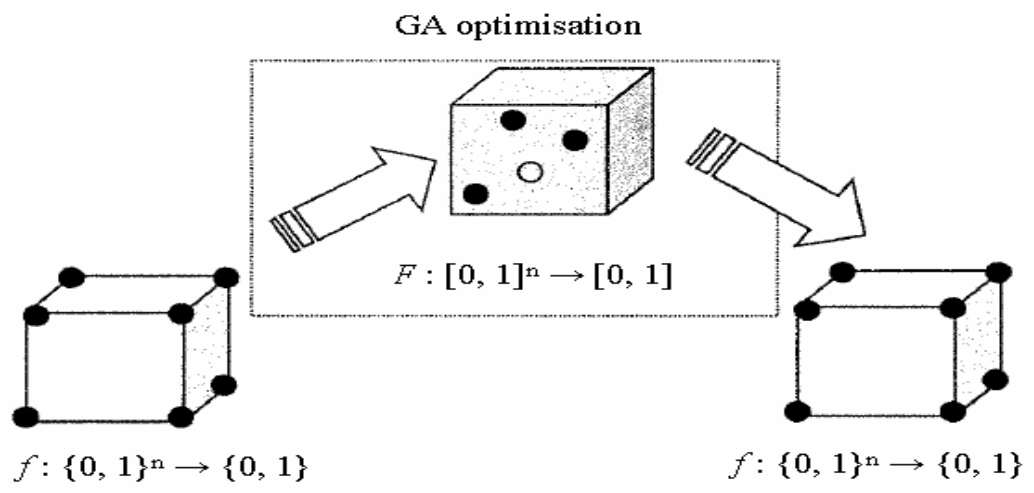


Figure 2.18: Application of fuzzy logic for fitness value and GA optimisation

The fitness function for a GA based BSAT solution can be made intelligent by incorporating some sort of knowledge and can be allowed to be heuristic routine [57]. Some of the proposals for fitness function are listed below.

- Fitness is 1 (true) if the Boolean expression is true otherwise false.
- Convert Boolean expression into Conjunctive Normal Form (CNF) and fitness function returns the total number of top level conjuncts that evaluate to true.
- Fitness is associated to sub-expressions of the main Boolean expression. Then the final fitness is computed from intermediate fitness values.

2.6 Summary

The overviews of grid, SOAP, BSAT problem and GAs have been presented in this chapter. Previous works performed on BSAT using GA and distributed computing have also been discussed.

Chapter 3

Design and partitioning of the algorithms

3.1 Introduction

Two algorithms: (1) brute force/exhaustive search algorithm and (2) a genetic optimisation algorithm have been developed for the Boolean satisfiability benchmark suit uf20-91 [58] and are executed on a single computer configuration and grid computing environment as mentioned before. The benchmark consists of 1000 satisfiable Conjunctive Normal Form (CNF) instances where each instance has 91 clauses in 20 variables and each clause has exactly 3 variables. This chapter describes both of the algorithms developed and coarse grained partitioning for parallel execution. All the programs can be found in the Appendices.

3.2 A Brute Force/Exhaustive search algorithm

The brute force/exhaustive search algorithm performs a linear search through the binary numbers/sequences starting from all zeros (00...0) to all ones (11…1) and returns as soon as the first solution is found. Since each instance of the benchmark [58] has 20 variables, the binary sequence has length 20 and each bit represents one variable. For 20 variables there will be 2^{20} = 1048576 binary numbers (corresponding to decimal 0 to 1048575) and in the worst case, all these numbers will be checked for satisfiability. The search algorithm is explained with the help of flowchart in figure 3.1. A detailed pseudo-code representation of the functions is given in the following sections.

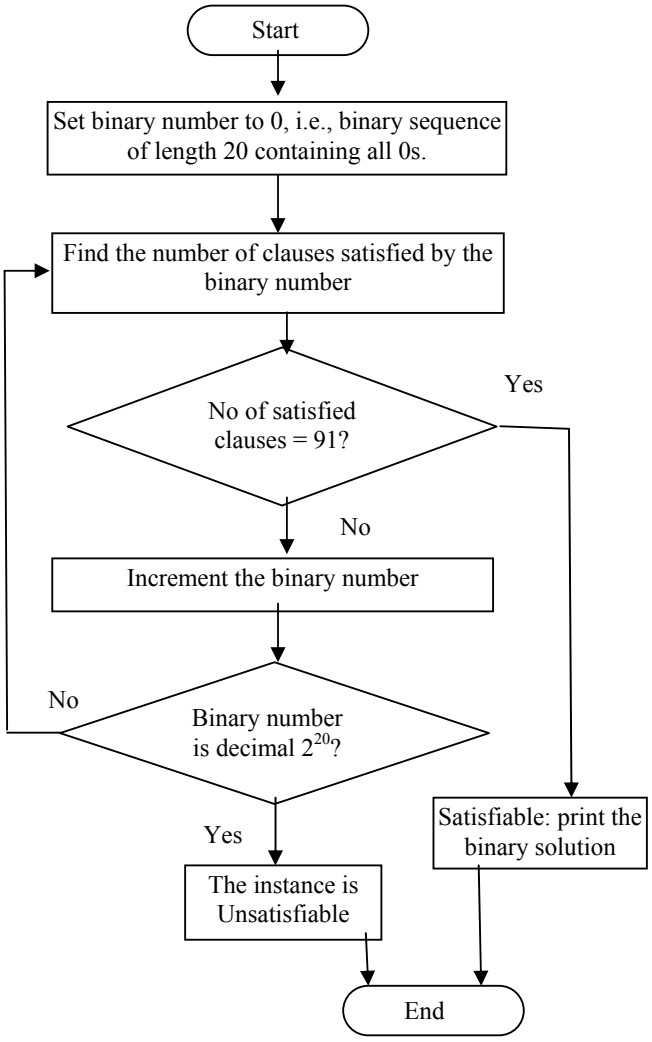


Figure 3.1: Flowchart representation of the Exhaustive search.

3.2.1 Function EXHAUSTIVE_SEARCH()

The function checks the satisfiability of a Boolean expression. The linear search method is shown below.

```
1.  Load the Boolean expression into EXPRESSION
2.  For VALUE = 0 to 1048575 do
3.      CONVERT (VALUE, SOLUTION)
4.      FITNESS = FIND_FITNESS (SOLUTION)
5.      If FITNESS = C then
6.          Print "Satisfiable" and SOLUTION
7.          Return
8.      End of if
9.  End of for
10. Print "Unsatisfiable"
```

EXPRESSION is a 2 dimensional array that stores the Boolean expression and each row stores one clause of the expression. *VALUE* is a long integer that generates the binary number for sequencing. *SOLUTION* is a 1-dimensional array and each cell stores TRUE or FALSE based on the corresponding bit value of the binary number stored in *VALUE*. *C* is the number of clauses in an instance.

3.2.2 Function CONVERT(VALUE, SOLUTION)

The function that converts the binary number into a sequence of TRUE / FALSE in an array is shown below.

```
1.  For J=1 to V do
2.      If bit J of VALUE is 1 then
3.          SOLUTION[J] = TRUE
4.      Else
5.          SOLUTION[J] = FALSE
6.      End of if
7.  End of for
```

Variable *V* contains the number of variables (20) in the expression. The *SOLUTION* array has 20 cells and cell *k* is set to TRUE if bit *k* of the binary number in *VALUE* is 1, otherwise it is set to FALSE.

3.2.3 Function FIND_FITNESS(SOLUTION)

The FIND_FITNESS function returns the number of satisfied clauses by *SOLUTION*. *FIT* is a local variable that keeps track of the number of satisfied clause so far and at the end of the function this value is returned. It should be noted here that, row *J* of *EXPRESSION* stores clause *J*. *C* is the number of clauses in the instance.

```
1. FIT = 0
2. For J=1 to C do
3.       If SOLUTION makes the J th clause TRUE then
4.            FIT = FIT + 1
5.       End of if
6. End of for
7. Return FIT
```

3.3 Partitioning of the exhaustive search algorithm

The entire binary sequence space (from decimal 0 to 1048575) is partitioned into 3 (three) disjoint or non-overlapping sub-sequences. On each computer, one sub-sequence is executed independently. For the Boolean satisfiability problem, as soon as a solution is found on any computer, *i.e.*, in a sub-sequence, the expression is Satisfiable and other sub-sequences can be aborted. Therefore, exhaustive BSAT search is highly suitable for grid computing where sub-tasks can execute without depending on each other. However, in this investigation, the minimum required time is considered in case of multiple solutions. The partitioning concept is depicted in figure 3.2.

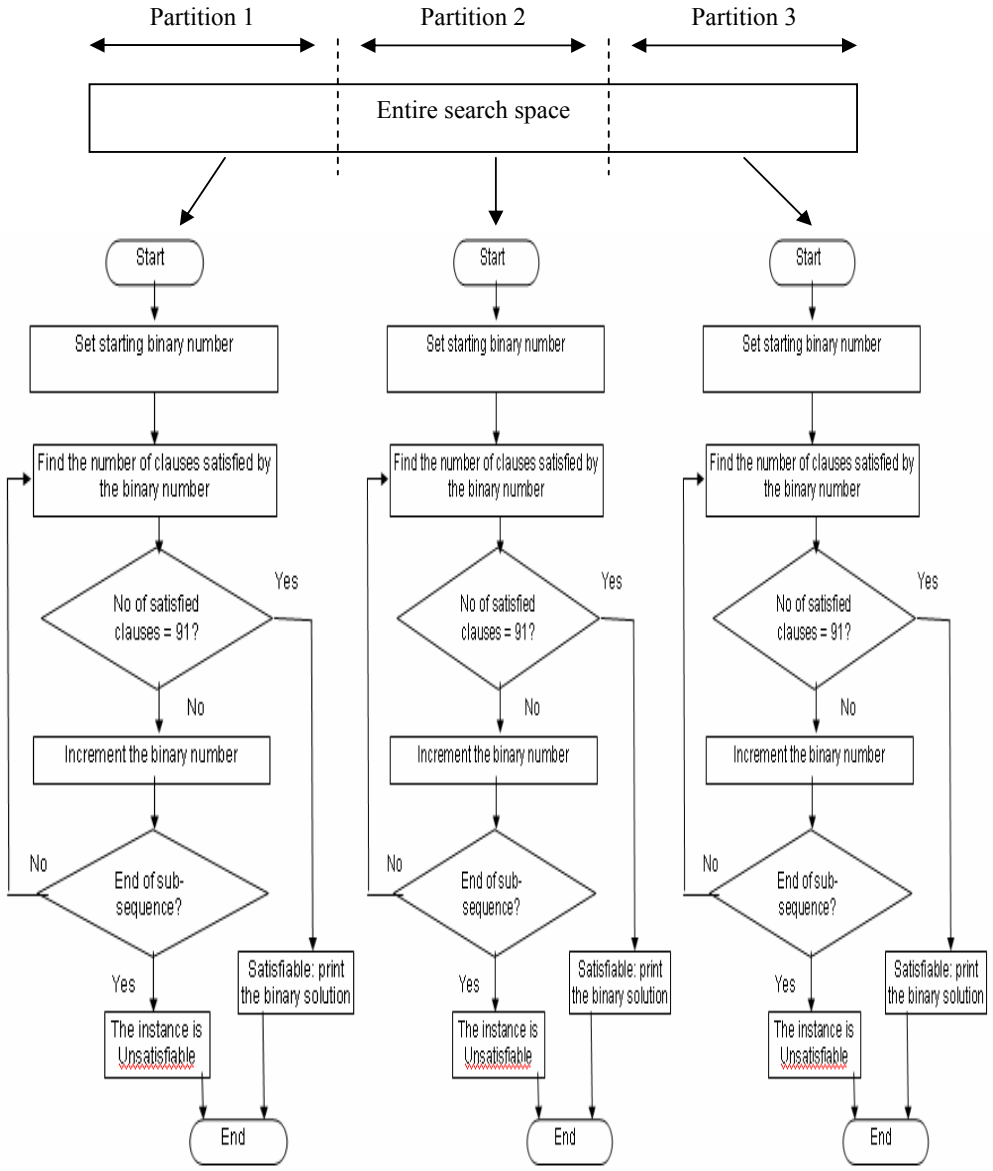


Figure 3.2: Parallelisation of exhaustive algorithm among 3 grid computers for the BSAT problem.

3.4 Genetic algorithm search

In this investigation, genetic algorithm (GA) is used to consider multiple probable solutions simultaneously. Crossover and mutation operations are applied to the solutions to improve them and to generate a solution eventually. For simplicity and speed, the fitness function returns an integer that is the number of clauses in the Boolean expression satisfied by a solution. The algorithm stops when a solution with fitness equal to the number of clauses (91) is found or a predefined number of generations (iterations) are observed. However, because of the nature of the algorithm, it does not guarantee that it will generate a solution for a Satisfiable expression.

3.4.1 Data structures

For the next sections, it is assumed that-

- *V*: Number of variables in the Boolean expression/function
- C: Number of clauses in the Boolean expression/function
- *P*: Size of /number of solutions in current generation
- *Q*: Size of /number of solutions in next generation
- CURRENT_GENERATION: a *P*x*V* matrix that stores the current *P* probable solutions where *P* is the size of population. CURRENT_GENERATION[*k*] is the k-th solution.
- NEW_GENERATION: a *Q*x*V* matrix that stores the new probable solutions after cross over and *Q* <= *P*.
- GENERATION: Number of generations the algorithm is applied to the benchmark.

- CURRENT_FITNESS: a *P*x1 matrix to store fitness of CURRENT_GENERATION solutions.
- NEW_FITNESS: a *Q*x1 matrix to store fitness of NEW_GENERATION solutions.
- EXPRESSION: A matrix that store the Boolean instance. Row *k* stores the *k*-th clause.

Matrices for CURRENT_GENERATION and NEW_GENERATIONS

Each solution/chromosome of CURRENT_GENERATION (CG) and NEW_GENERATION (NG) is a row vector of *V* components where component *k* represents k-th variable in that solution. Each component is a Boolean variable that stores a 0 or 1. Here, CG_{ij} or NG_{ij} represents the j-th variable of i th solution. Figure 3.3 shows the scenario when *P* and *Q* simultaneous soutions are considered and hence these two matrices store a generation.

$$\begin{bmatrix} CG_{11} & CG_{12} & \ldots\ldots & CG_{1V} \\ CG_{21} & CG_{22} & \ldots\ldots & CG_{2V} \\ \ldots\ldots & \ldots\ldots & \ldots\ldots & \ldots\ldots \\ CG_{P1} & CG_{P2} & \ldots\ldots & CG_{PV} \end{bmatrix} \begin{bmatrix} NG_{11} & NG_{12} & \ldots\ldots & NG_{1V} \\ NG_{21} & NG_{22} & \ldots\ldots & NG_{2V} \\ \ldots\ldots & \ldots\ldots & \ldots\ldots & \ldots\ldots \\ NG_{Q1} & NG_{Q2} & \ldots\ldots & NG_{QV} \end{bmatrix}$$

Figure 3.3: Data structures of CURRENT_GENERATION (CG) and NEXT_GENERATION (NG) of multiple solutions (a generation).

Matrices for CURRENT_FITNESS and NEXT_FITNESS

Two column vectors of size *P*x1 and *Q*x1 are maintained to store the fitness values of the solutions of current generation (CG) and next generation (NG), respectively. $Fitness_i$ stores the fitness of i th solution. Since fitness of a solution is defined as the

number of clauses satisfied by that solution, it is an integer value that can range from 0 to *C* inclusive. Figure 3.4 depicts fitness matrices.

Figure 3.4: Data structures of CURRENT_FITNESS and NEXT_FITNESS of multiple solutions.

Matrix for EXPRESSION

The entire expresssion/instance is stored in a matrix. Each clause is expressed by a row vector of length *V*. Here component *k* represents the k-th Boolean variable. Figure 3.5 depicts the data structure of storing a Boolean expression of *C* clauses. If the k-th variable appears in original form then it is a 1 else it is a 0 (complemented form). However, if the k-th variable is missing in the clause then k-th component is a

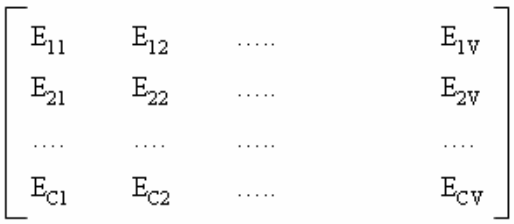


Figure 3.5: Data structure of EXPRESSION.

3.4.2 The proposed GA algorithm

This section proposes a serial GA based Boolean satisfiability algorithm. Figure 3.6 depicts the flowchart of the GA algorithm followed by pseudo-code representation of the search. The initial *P* chromosomes/solutions are generated with random values. Mutation is applied after every 100 (hundred) generations.

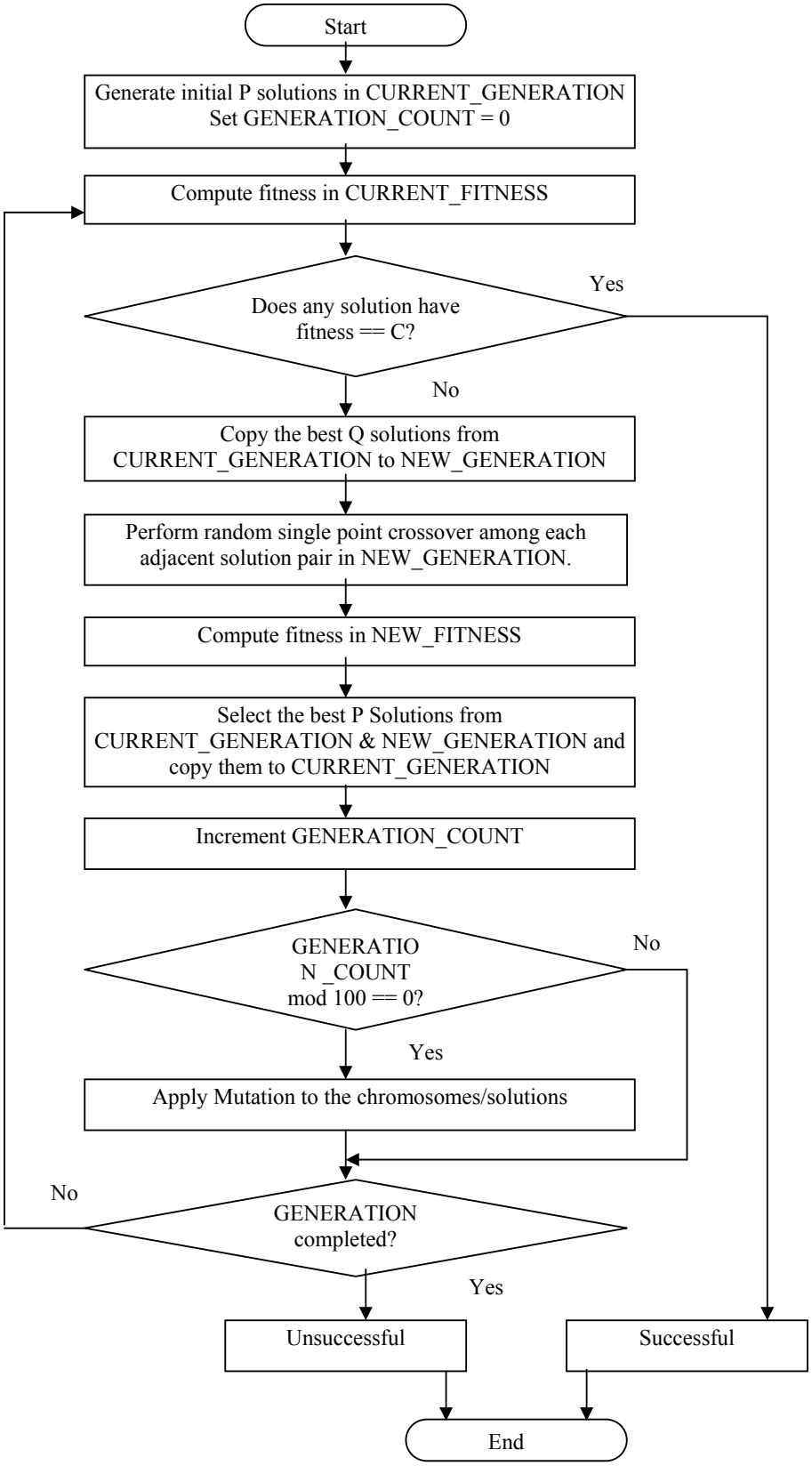


Figure 3.6: Flowchart representation of the GA BSAT algorithm.

A detailed pseudo-code representation of the functions is given in the following sections.

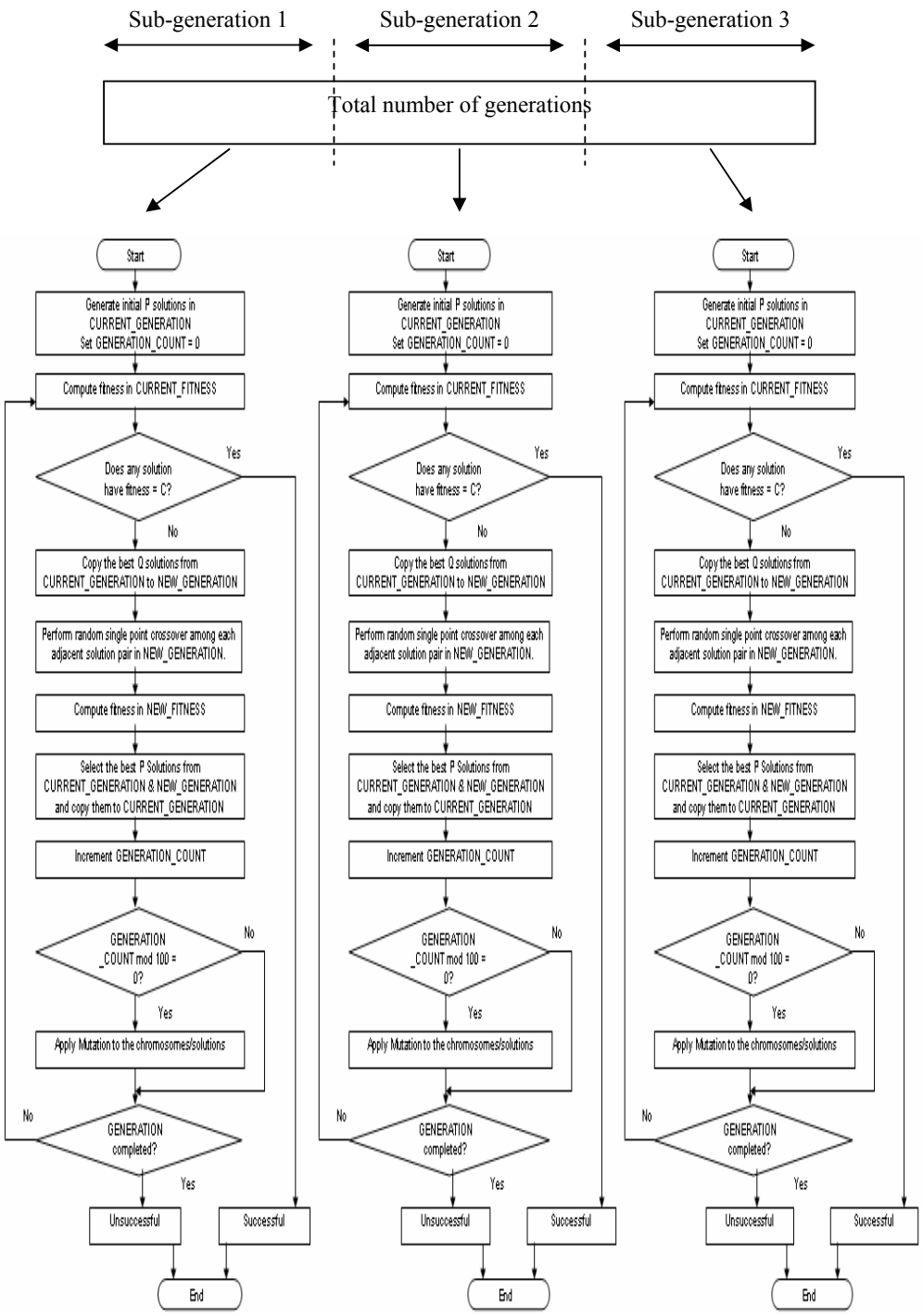
Function GA_BSAT_SEARCH()

Function to check the satisfiability of a Boolean expression/instance using genetic algorithm search method. The Boolean expression is stored in 2-dimensional array EXPRESSION.

```
1.  CURRENT_GENERATION = Generate initial P solutions randomly and
    Set GENERATION_COUNT = 0
2.  For J = 1 to GENERATION do
3.      For each solution in CURRENT_GENERATION do
4.          Compute CURRENT_FITNESS[k]
5.      End of for loop
6.      If there exists a solution k for which fitness is C then
7.          Print "Successful" and CURRENT_GENERATION[k]
8.          Return
9.      End of if
10.     Copy the best Q solutions from CURRENT_GENERATION to
        NEW_GENERATION based on fitness value
11.     For each solution pair in NEW_GENERATION do
12.         Apply cross over NEW_GENERATION[k] with
            NEW_GENERATION[k+1]
13.     End of for loop
14.     For each solution in NEW_GENERATION do
15.         Compute NEW_FITNESS[k]
16.     End of for loop
17.     Select the best P solutions from CURRENT_GENERATION and
        NEW_GENERATION
18.     Copy them to CURRENT_GENERATION
19.     GENERATION_COUNT = GENERATION_COUNT + 1
20.     If GENERATION_COUNT mod 100 = 0 then
21.         Apply Mutation to all chromosomes/solutions
22.     End of if
23.     If GENERATION_COUNT = GENERATION then
24.         Print "Unsuccessful"
25.     End of if
26. End of for loop
```

3.5 Partitioning of GA based BSAT search algorithm

The number of generations to execute is simply partitioned into 3 (three) sub-generations and each of the 3 (three) computers of the grid computing system executes one sub-generation. For instance, the first computer takes care of the first 3 generations, the second one handles the next 3 generations and the third computer runs the last 4 generations until the total number of generations is 10. Therefore, each computer can execute its own sub-generations without depending on the other. Like parallelised exhaustive search algorithm, GA based BSAT aborts/discards the other two sub-generations whenever one computer obtains a solution. For multiple solutions, only the solution with the least time is considered. The partitioning concept is depicted in figure 3.7.



Sub-generation A	Sub-generation B	Sub-generation C
GENERATION/3	GENERATION/3	GENERATION/3
Computer A	Computer B	Computer C

Figure 3.7: Parallelisation of GA based BSAT algorithm among 3 grid computers.

3.6 Summary

The same FIND_FITNESS (SOLUTION) function is used for both Exhaustive and GA BSAT algorithm. However, unlike [29] where the fitness function returns a real/floating point value based on fuzzy logic as discussed in chapter 2, FIND_FITNESS function simply returns the number of satisfied clauses that must be an integer. This makes it simple and allows executing faster.

An "instance" might have more than one solution and any one solution makes BSAT decision TRUE. Each sub-sequence in case of exhaustive search is independent and can result in a solution. Similarly, all three sub-generations can be executed without depending on the other sub-generation. These allow to search solution in 3 parts in a parallel fashion. Therefore, this type of application is very much suitable for grid computing where sub-tasks are completely independent.

The design of the overall grid computing environment consisting of 3 computers is discussed in the next chapter.

Chapter 4

Design of the Grid computing system

4.1 Introduction

The entire grid consists of three computers connected via a local area network (LAN). The client splits the satisfiability task and sends them as Remote Procedural Call (RPC) using SOAP to 3 servers run on three different machines. This chapter explores various aspects of the system.

4.2 Specification

The grid is implemented with the following system specification

Hardware:

Each computer is equipped with the following hardware

- Processor: Pentium 4 computers with 2.66 MHz clock
- 512 MB main memory and 60 GB hard disk
- 100 Mbps LAN card, CD ROM *etc*.
- Switch: NETGEAR 8 port 10/100/1000 Mbps Gigabit switch, Model GS 108

Software:

Each computer is installed with the following software

- Mandrake Linux 10.0 is used as the Operating System.
- Simple Object Access Protocol (SOAP) used for Remote Procedural Call (RPC).

4.3 LAN topology

All three computers are connected to a high speed switch and are assigned static IP addresses. The switch allows parallel communication among the computers. Both the

client and servers are installed with Mandrake Linux 10.0. The client and one of the servers are run on the same machine. When RPCs are sent by the client machine, it will then wait for results to arrive. Hence, the client machine can also be used to execute a server. Moreover, this avoids network overhead of sending RPC to another computer. The network topology is shown is figure 4.1.

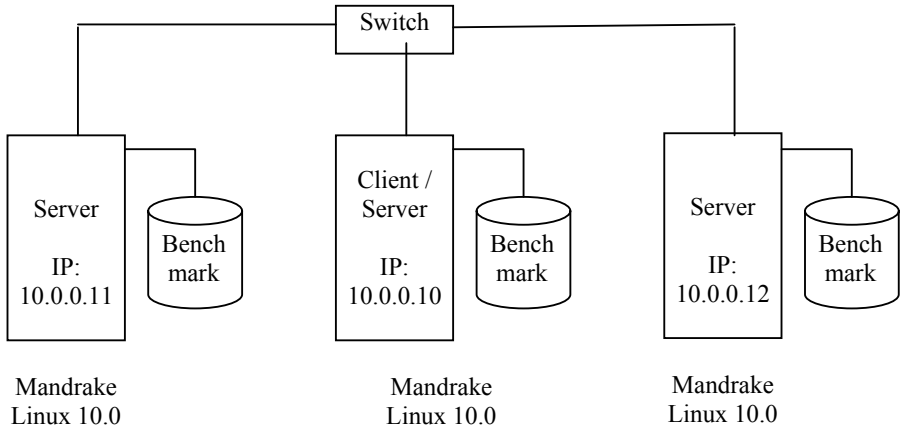


Figure 4.1: LAN topology of the grid System.

4.4 Client design

The client is designed as a Multi-threaded application using the C language. It launches 3 threads and each of these invokes an RPC SOAP call independently. Each thread waits for a corresponding server to finish. In this strategy the client does not need to wait for a server to finish current RPC before making another one. Two threads send RPC request to other machines and the other one sends to the server running on the same machine. No synchronisation is required among the threads or servers since each sub-task is completely disjoint and can be run independently.

An alternative implementation follows a multi-process model using the fork call in the C language. However, this puts a heavier burden of creating new processes on the

client machine. On the other hand, threads are light weight processes that share data with the parent thread very easily. Moreover, they execute faster and avoid overheads associating with creating new processes. The client structure is depicted in figure 4.2. High port numbers are chosen to ensure that these user defined ports will not conflict with the operating system ports. The numbers 25000, 30000, 35000 are chosen arbitrarily.

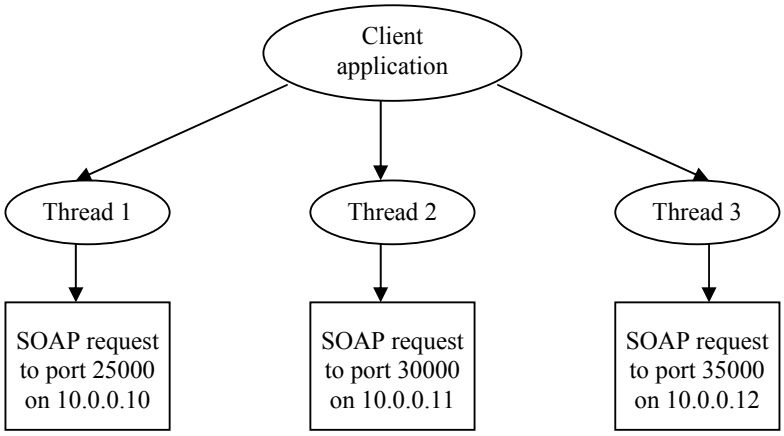


Figure 4.2: Multi-threaded architecture of the client application.

4.5 Server design

The server is written in the C language. It initiates the SOAP system and listens to a particular predefined port for SOAP request. Whenever a call arrives it reads the benchmark file into the main memory and then executes the call. After finishing the call it starts listening again. The behaviour of the server is shown in figure 4.3 with the aid of a flowchart.

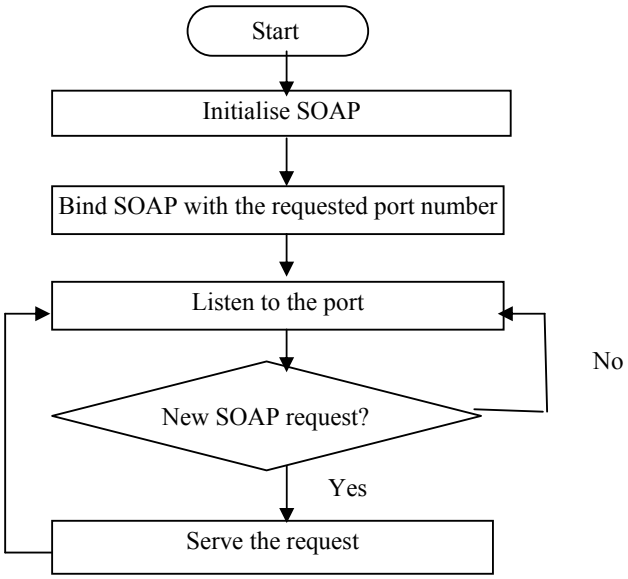


Figure 4.3: Server execution on grid computers.

4.6 Argument passing

For both exhaustive / GA BSAT search the client passes arguments to the servers and in reply the server returns some values to the client. These are discussed in this section.

4.6.1 Exhaustive BSAT search

The client passes two arguments to each server. The first and second values are the start and end values of the binary sub-sequence to be executed by the server. For example, the first server runs the sub-sequence from 0 to 349524. Therefore, 0 and 349524 are passed as first and second arguments, respectively.

In reply, the server returns the time required to read the benchmark file and status of execution: successful (satisfiable) or unsuccessful (unsatisfiable). If the instance is satisfied, the result is also returned. The mechanism is depicted in figure 4.4.

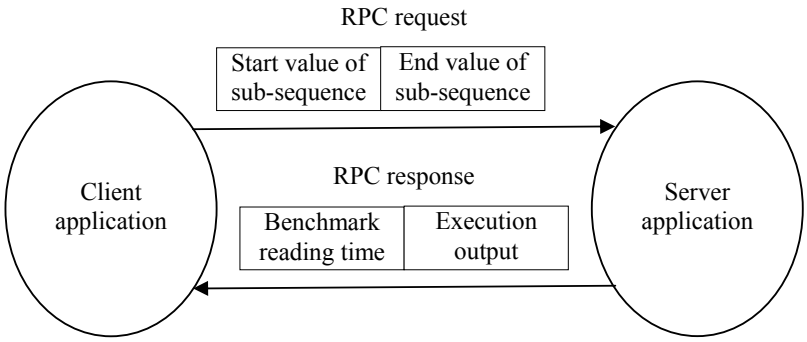


Figure 4.4: Argument passing between client and server for Exhaustive search.

4.6.2 GA BSAT search

In the GA case, the client passes two arguments that specify the number of iterations in sub-generation and size of population to each server. For example, for total 100,000 generations and population size 20, 33,333 (100,000/3) and 20 are passed to each server.

Each server responds by returning benchmark file reading time and satisfiability status. In case of satisfiable instance, the solution is also passed back to the client. Figure 4.5 shows the technique.

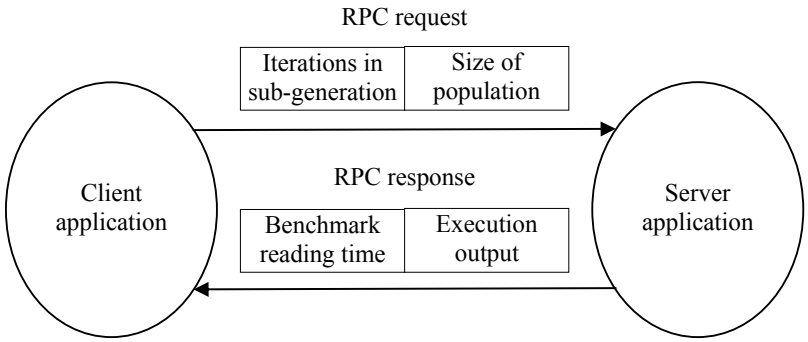


Figure 4.5: Argument passing between client and server for Genetic BSAT search.

4.7 Time calculation

Both the server and the client performs computation of elapsed time independently. This section explains how the client and server keep track of time.

Client timing

The client starts the timer just before initialising SOAP mechanism. Then it performs the SOAP call and stops the timer just after the soap call has returned. The elapsed time to complete each SOAP call is recorded by the client using the function *gettimeofday()*. It can be noted here that, *gettimeofday()* can be used to get the time elapsed. For example, an exhaustive search in the 3-computer grid, SOAP took 321.13 ms to return. But, on a single server it took only 310.0 ms to execute.

Server timing

Each server performs two time computations. These are

- Benchmark File reading time: The clock starts just before reading the benchmark file and after finishing it are recorded using *gettimeofday()* function. Then from these times the time needed to read the benchmark file is computed and returned to the client.
- Execution/processing time: the clock() function is used to compute the time required by execution of the algorithm. It should be mentioned that the clock() function returns the number of CPU clocks for a program to execute. Then the execution/processing time can be obtained by dividing the number of clocks by CLOCKS_PER_SEC.

Time computation on Client

The client records the total elapsed time to complete a SOAP call. This time has 3 (three) components: SOAP overhead time (this includes transmission, marshalling into XML *etc*.), benchmark file reading time and execution time on the server. In other words,

Elapsed time = SOAP overhead + Benchmark reading time + Execution time

However, since the server returns the time for reading benchmark file, the client can compute the time for a SOAP call on a remote machine using the formula

Total time = Elapsed time – Benchmark file reading time

4.8 Benchmark File replication

The benchmark file is copied / replicated to all the server machines so that each server can read the file in parallel as shown in figure 4.1. However, file reading time can vary depending on hard disk properties, like latency time, seek time *etc*.

4.9 Summary

The parallel algorithm proposed in [48] uses special tightly coupled hardware that supports Multiple-Instruction-Multiple-Data (MIMD) stream architecture and fine grained parallelism. The processing nodes are assigned a subset of variables and clauses and are organised in 2-dimensional mesh as discussed in chapter 2.

In contrast, this investigation uses loosely coupled grid system that supports coarse grained parallelism. Each processor is assigned a sub-sequence (Exhaustive search) or sub-generation (GA BSAT). Processing nodes are connected to a LAN via a high speed switch.

When a task is split into 3 parts and executed in parallel, theoretically it should run 3 times faster than on a single computer. However, some time is wasted because of the overhead of SOAP and of transmission over the local network.

The next chapter presents and analyses the results for the exhaustive search and genetic algorithm on a single computer and in the grid computing environment.

Chapter 5

Implementation and Results

5.1 Introduction

This chapter provides the implementation details and results of execution times for four cases: the Exhaustive BSAT search on (1) a single computer and (2) on grid, GA BSAT search on (3) a single computer and (4) on grid. For each of these cases, two types of executable codes are generated: non-optimised and *gcc* compiler O3 optimised [59], [60], [61] codes. Both positive and negative error bars for the readings are investigated and file sizes for non-optimised and O3 optimised codes are compared.

5.2 Number of instances considered

The benchmark suit uf20-91 [58] contains 1000 Satisfiable instances. Among these, 50 instances are picked randomly for this investigation. Both the exhaustive and GA BSAT search are applied to these 50 (fifty) instances/expressions and the results are observed.

5.3 Number of readings taken

5.3.1 Exhaustive BSAT search on single computer

The exhaustive BSAT algorithm performs a linear search in the search space. It finds the same solution in every execution instance and requires the same amount of time no matter how many times it is run on a single computer. So, in case of observation only one reading is required to be taken for the Exhaustive BSAT search on a single computer.

5.3.2 Exhaustive BSAT search on grid

Linear search in sub-sequences will result in the same execution time on grid computers for all executions. But, SOAP calls take different times to complete from one run to another. So, for each instance 30 readings are taken to obtain a stable and statistically interpretable result in the case of exhaustive grid based BSAT search.

5.3.3 GA BSAT search on single computer

As mentioned earlier, GA BSAT search does not guarantee a result even if the instance has one or more solution. Moreover, this algorithm might produce inconsistent execution time on the same computer. Because of these reasons, 30 readings are considered for each expression/instance.

5.3.4 GA BSAT search on grid

When the GA based BSAT search algorithm is executed on the grid platform, completion time can vary because of the non-deterministic properties of GA as well as SOAP overhead. In the worst case, all 3 sub-generations might fail to find a solution. Like before, 30 readings are taken into account for each of the 50 instances considered.

5.4 Non-optimised vs O3 Optimised code

The gcc compiler has options to generate various optimised executable codes. Without any type of optimisation, the compiler will try to reduce the cost of compilation. Statements become independent and breakpoints can be set between two statements to change values of variables. It allocates registers to only those

variables that have been declared as *register*. The compiler's goal is to reduce code size and execution time [59], [60].

However, the *gcc* compiler supports several types of code optimisation options and turning on optimisation options will instruct the compiler to attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program [59], [60]. These options are discussed below.

5.4.1 O1 Optimisation

The O1 optimising option takes longer time to compile and more memory for a large function. It turns on several options. Some of these include [61]

- -fthread-jumps: Optimises the cases when a jump branches to a location where another comparison subsumed by the first. If so, the first branch is redirected to either the destination of the second branch or a point immediately following it, depending on whether the condition is known to be true or false.
- -fdelayed-branch: This attempts to reorder instructions to exploit instruction slots available after delayed branch instructions
- -fomit-frame-pointer: This does not reserve a register for frame pointer for functions that do not need one. This avoids the instructions to save, set up and restore frame pointers. Therefore, it also makes an extra register available in many functions

5.4.2 O2 Optimisation

The O2 optimisation option performs almost all supported optimisations that do not involve a space-speed tradeoff. However, it does not do loop unrolling, function in-lining and register renaming [59], [60]. Some of the options are explained below.

- -fforce-mem: It is turned on all machines and forces operands to be copied into registers before doing arithmetic operations on them. This produces better code by making all memory references potential common subexpressions. When they are not common subexpressions, instruction combination should eliminate the separate register-load [61].
- Frame pointer elimination: This is turned on those machines where doing so does not interfere with debugging [59], [60].

5.4.3 O3 Optimisation

The O3 optimisation option turns on all optimisations specified by O2 and also turns on the following features-

- -finline-functions: This integrates all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way. If all calls to a given function are integrated and the function is declared static, then the function is normally not output as assembler code in its own right [61].
- -frename-registers: This attempts to avoid false dependencies in scheduled code by making use of registers left over after register allocation. This optimisation benefits processors that supports lots of registers [61].

5.5 File size for Non-optimised and O3 optimised code

For the Exhaustive and GA BSAT search on single computer there is only one application file. On the other hand, in case of the grid environment, there are two applications: the client and the server. The following sections discuss the results obtained.

5.6 Exhaustive search on a single computer

In Exhaustive search on a single computer, the execution time is dependent on the location of the first solution in the entire search space. In other words, if the first solution is located towards the beginning of the search space, it takes much less time to find it.

5.6.1 Non-optimised vs. O3 optimised machine code

It is apparent from figure 5.1 that O3 optimised code takes less time to execute than non-optimised code for all the instances. But due to the linear search method, search both the graphs have the same shape. The difference in execution time is significant when the solution is near the end of the search space. O3 optimised code executes at least twice faster than the non-optimised code and therefore it is proposed that

Execution time of non-optimised Exhaustive BSAT *search on single computer* ≥

*2 * Execution time of O3 optimised Exhaustive* BSAT *search on single computer*

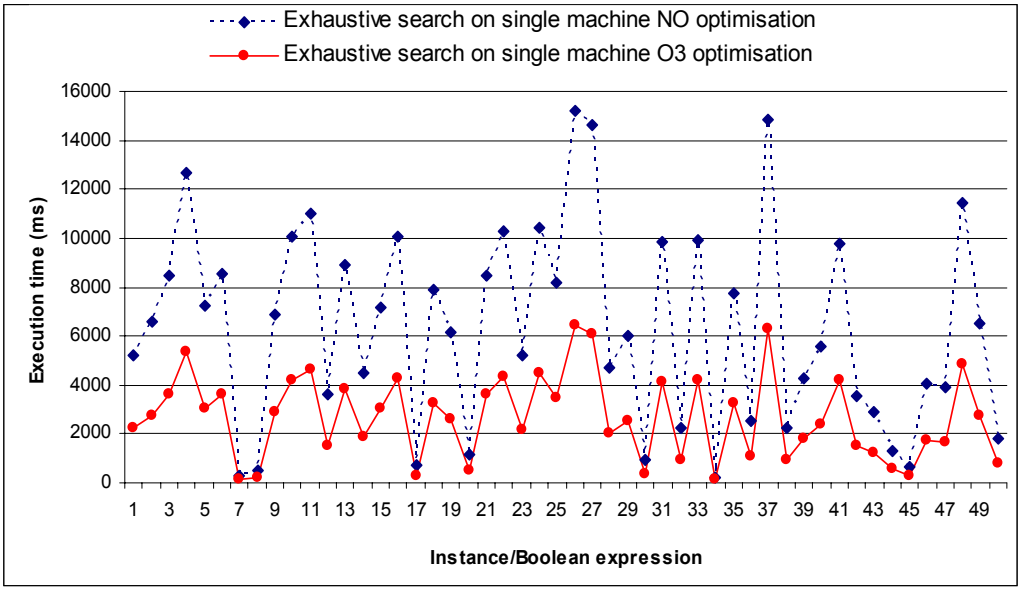


Figure 5.1: Execution time of Exhaustive BSAT search on single computer for non-optimised and O3 optimised machine code.

5.7 Exhaustive search on grid

The grid implementation shows better performance when the instance has more than one solution. In this case, more than one computer generates results and the best one is picked. Furthermore, the exhaustive BSAT search on the grid implementation executes faster if the solutions are located near the start of the sub-sequences.

5.7.1 Non-optimised vs. O3 optimised machine code

Figure 5.2 depicts the comparison between non-optimised and O3 optimised code. Each point of the figure represents the average of 30 readings. Error bars are computed as standard deviation both in positive and negative directions.

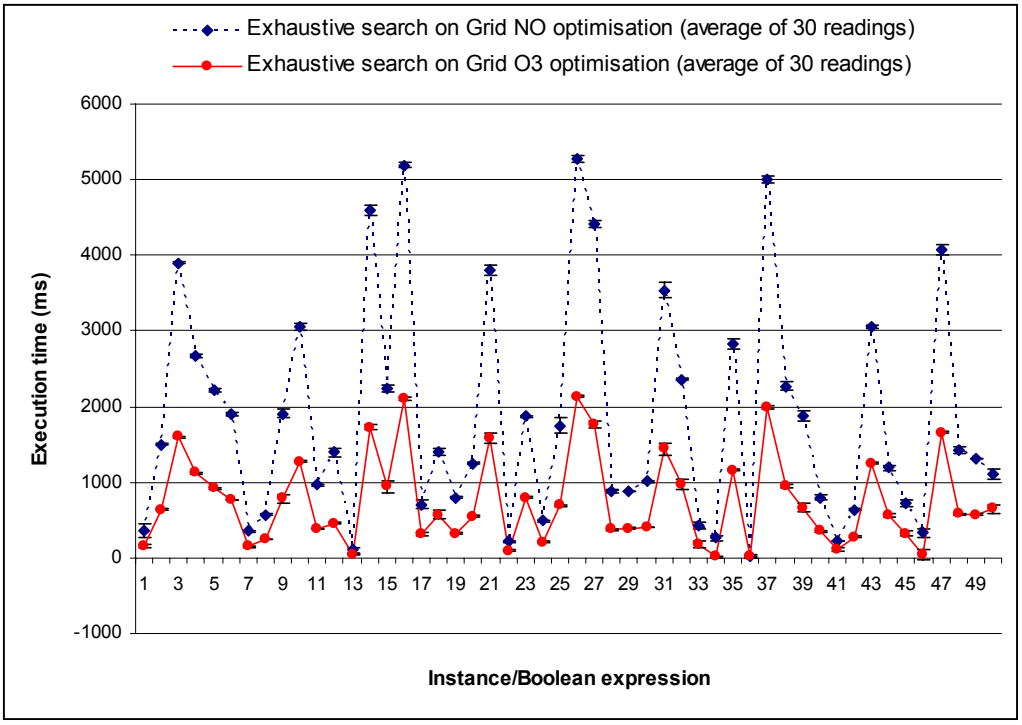


Figure 5.2: Execution time of Exhaustive BSAT search on grid for non-optimised and optimised machine code with error bars.

It is clear from figure 5.2 that in general, both the curves show the same behaviour (shape of curve) for all the instances. However, it can be established that

Execution time of non-optimised Exhaustive BSAT *search on grid* ≥

*2 * Execution time of O3 optimised Exhaustive* BSAT *search on grid*

5.8 Exhaustive search on single computer vs. on grid

The grid approach incurs some SOAP overhead time for data type marshalling and network transmission. Since each sub-sequence is $\frac{1}{3}$rd of the entire search space, searching a sub-sequence should take $\frac{1}{3}$rd time. In general, it is obvious from figure 5.3 and 5.4 that the grid shows better execution time than single computer for most of the instances. It can be proposed that the non-optimised and O3 optimised code support the following relationship

Execution time on single computer ≈ *3 * Maximum execution time on grid*

Theoretically, the exhaustive BSAT search on a single computer exhibits better performance than on the grid enabled one if the first sub-sequence contains a solution. The time to find the solution will be the same for the both cases but the grid will experience more overhead time for SOAP mechanism. This is apparent from figure 5.3 and 5.4 that show the comparison between non-optimised and O3 optimised code, respectively. For some instances, the execution time of the exhaustive search on single computer is lower than that on the grid.

5.8.1 Non-optimised machine code

Figure 5.3 shows that maximum execution time for non-optimised and O3 optimised machine codes are approximately 15000 and 5200 ms, respectively.

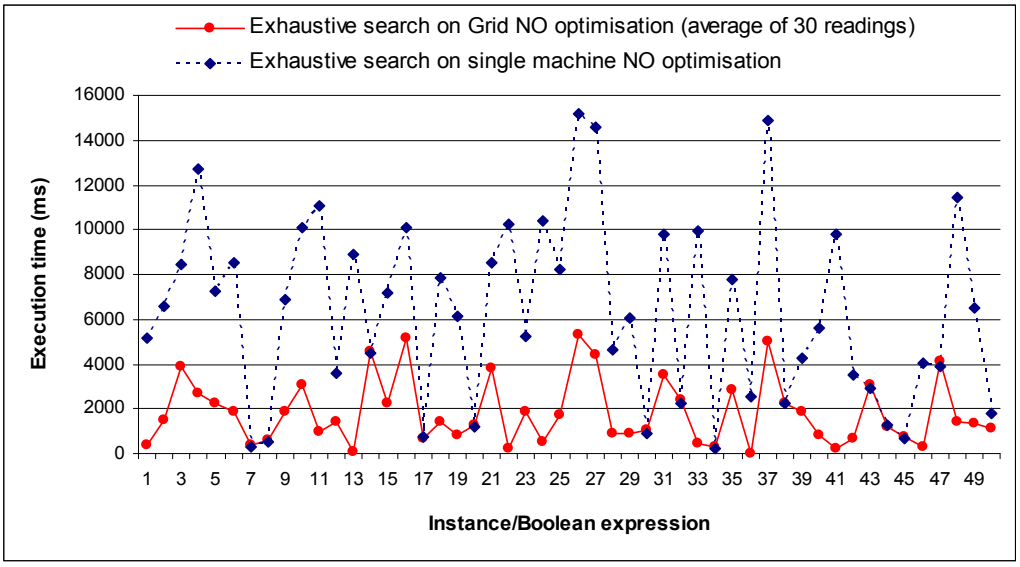


Figure 5.3: Execution time of Exhaustive BSAT search on single computer and grid for non-optimised machine code.

5.8.2 O3 optimised machine code

According to the figure 5.4, highest execution times are around 6500 (non-optimised) and 2200 ms (O3 optimised).

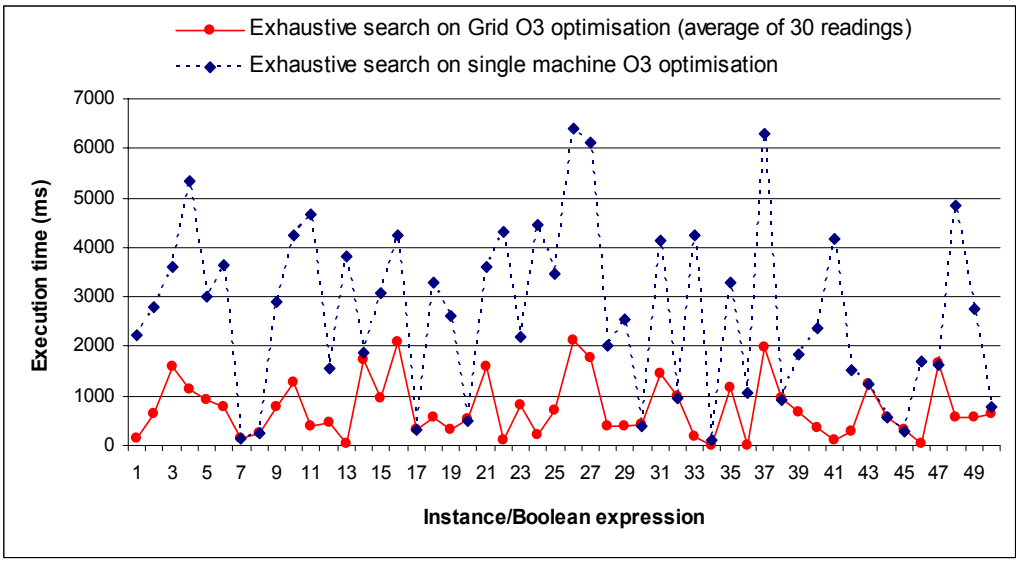


Figure 5.4: Execution time of Exhaustive BSAT search on single computer and grid for O3 optimised machine code.

5.8.3 File size

Figure 5.5 depicts the comparison for the executable files for the exhaustive BSAT search. It is evident that optimisation has almost no effect on file size for the BSAT search on single computer. But for the grid enabled O3 optimised files are larger than non-optimised files and it can be inferred that for both server and client

*O3 Optimised grid file size ≈ 1.8 * non-optimised grid file size*

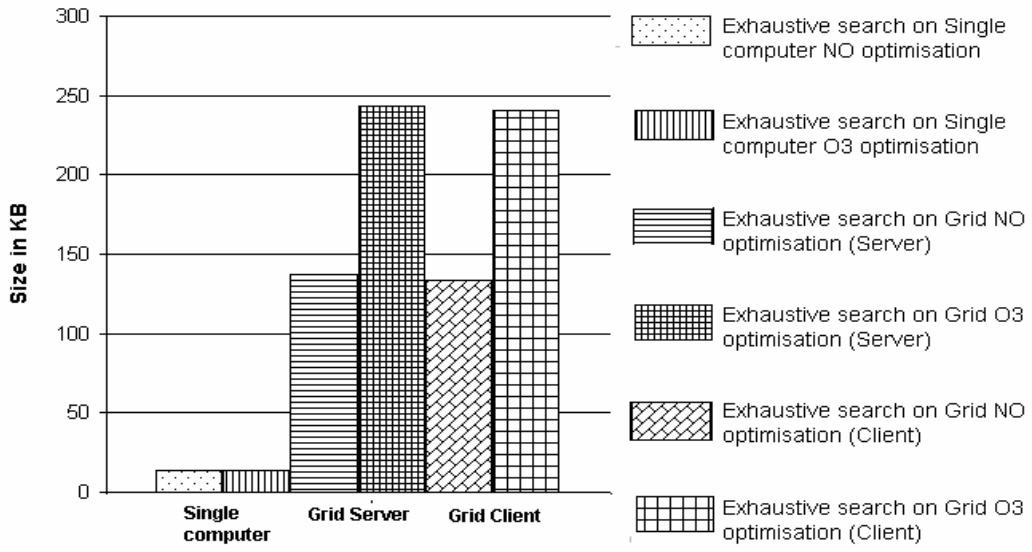


Figure 5.5: Comparison of executable file size on single computer and on grid (Server and Client)

5.9 GA BSAT search on single computer

The GA BSAT search method is applied for population size of 20 and 10,000 generations. Since each expression has 20 variables, 20 random chromosomes / solutions are considered. It has been observed that population size larger than 20, for instance, 40 or 60 and generations higher than 10,000 do not have significant improvement in results. However, though every instance is satisfiable, a number of executions did not find any solution since this algorithm does not guarantee that it will find a solution if it exists.

5.9.1 Number of successful search

The GA BSAT algorithm found a solution for all 30 executions in case of 24 (non-optimised) and 22 (O3 optimised) instances as depicted in figure 5.6a and 5.6b. For both of the cases, the lowest probability to find a solution for an instance is around 0.33.

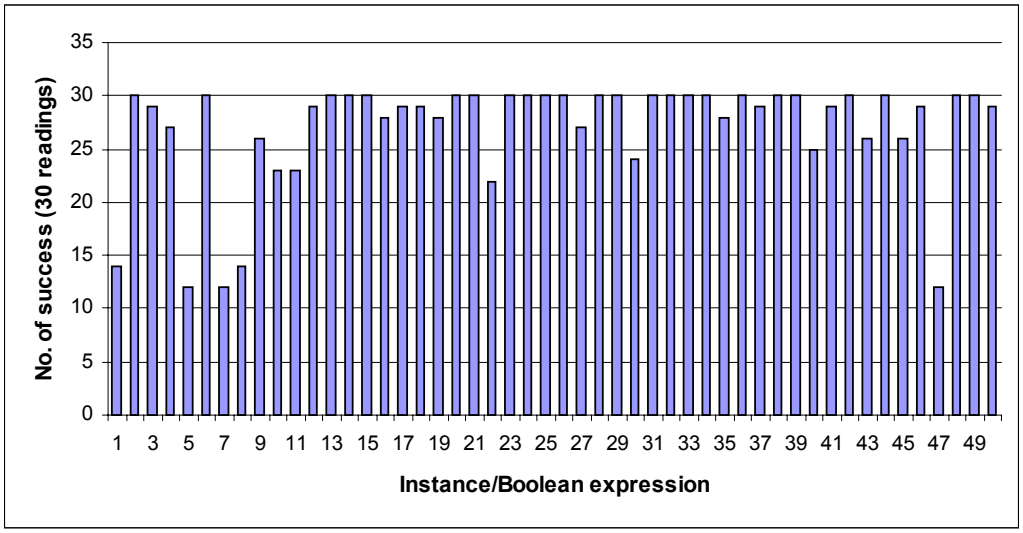


Figure 5.6a: No. of successful searches in 30 executions for GA BSAT on single computer for non-optimised machine code.

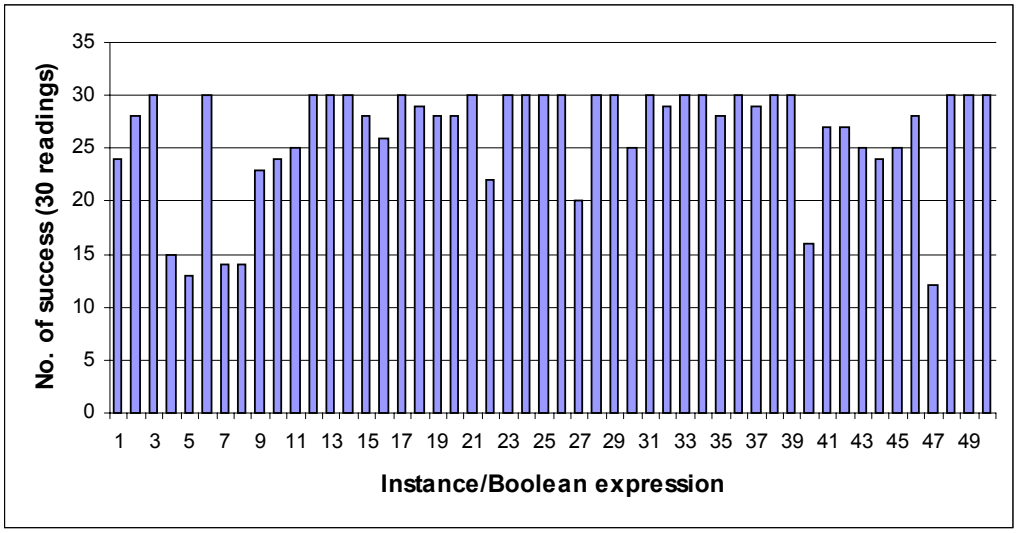


Figure 5.6b: No. of successful searches in 30 executions for GA BSAT on single computer for O3 optimised machine code.

5.9.2 Non-optimised vs. O3 optimised code for successful search

For each instance, the executions that found a solution are considered. The average of these successful search times are plotted in figure 5.7. Due to the nature of genetic algorithms, there is no clear relationship between the execution time of non-optimised and O3 optimised machine code. Even for some instances, non-optimised code shows better performance. However, for most of the instances, O3 optimised code exhibits less execution time and stays within 250 ms.

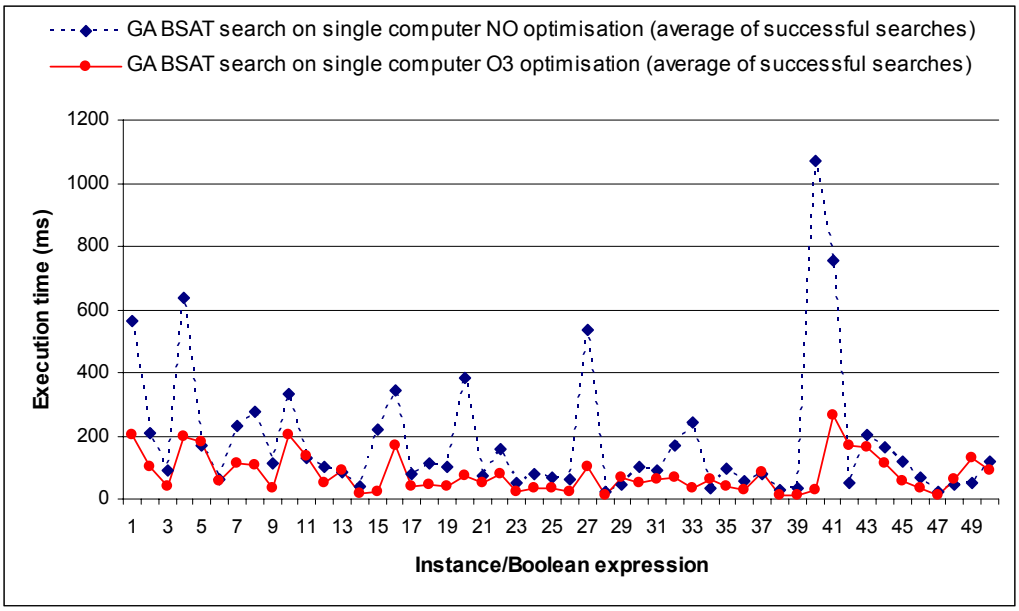


Figure 5.7: Execution time of GA BSAT search on single computer for non-optimised and O3 optimised machine code.

5.9.3 Error bars of non-optimised and O3 optimised machine codes

Figure 5.8 and 5.9 depict the error bars for non-optimised and O3 optimised code, respectively. Each point of the figures represents the average of successful readings and error bars represent the standard deviation both in positive and negative directions. For higher execution time (1000 ms for non-optimised code and 200 ms for O3 optimised code), the error bars are comparatively larger. In some executions,

the feasible solutions become similar after a good number of generations and further crossover can not improve their fitness significantly. In this case, the algorithm is trapped at a local minima.

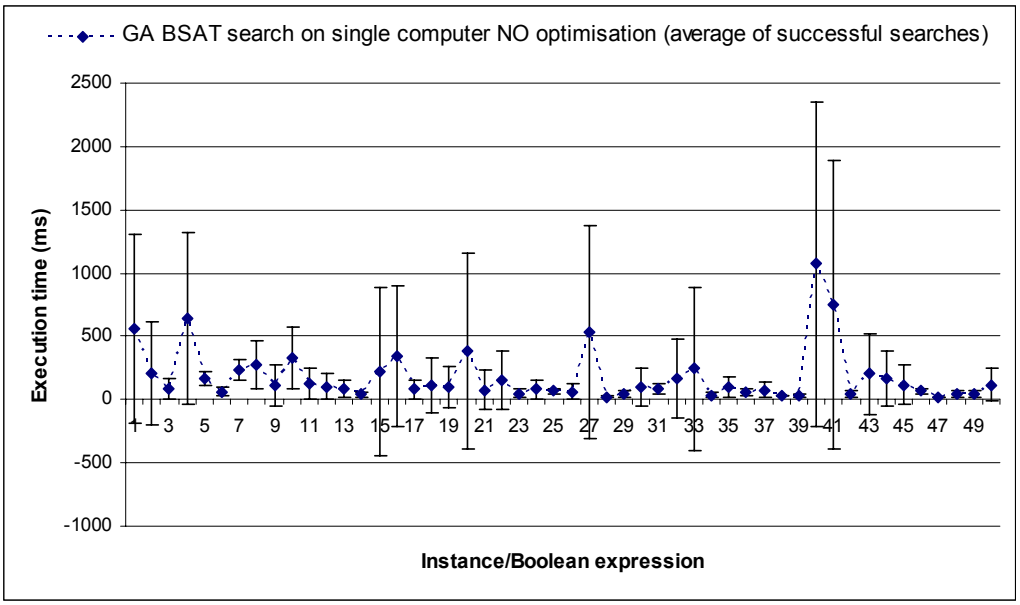


Figure 5.8: Graph with error bars of execution time of GA BSAT search on single computer for non-optimised machine code.

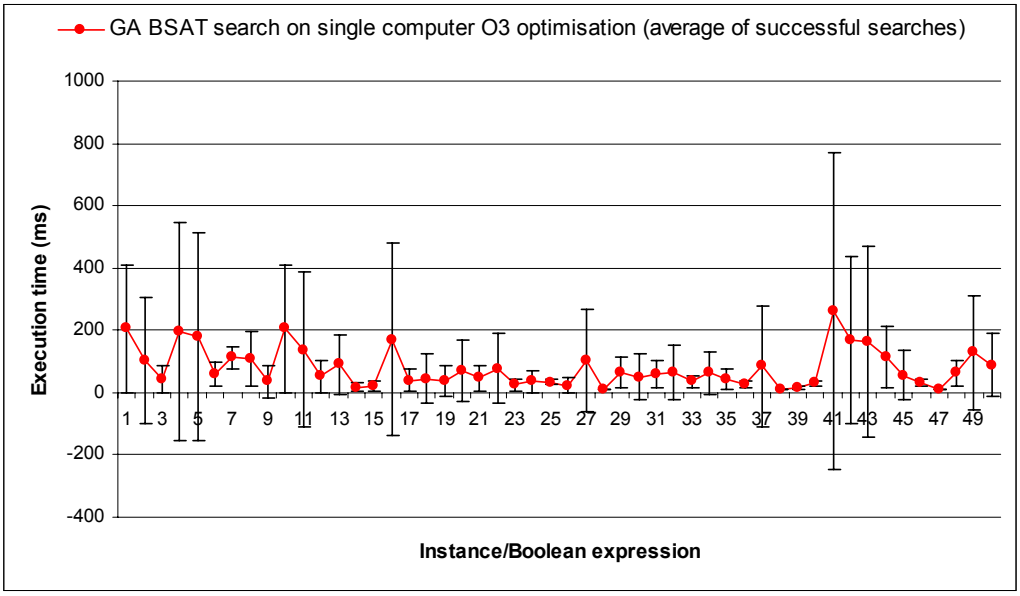


Figure 5.9: Graph with error bars of execution time of GA BSAT search on single computer for O3 optimised machine code.

5.10 GA BSAT search on the grid configuration

The GA BSAT search was implemented on the grid configuration for population size 20 and 3,333 (=10,000/3) generations on each of the grid computers. A search is classified to be successful if any of the computers finds a solution. However, in some cases, none of the computers was able to find a solution though every instance of the benchmark is Satisfiable.

5.10.1 Number of successful search

Figure 5.10a and 5.10b show that the GA BSAT is successful in all 30 executions for 30 (non-optimised code) and 31 instances (O3 optimised code). It indicates that the O3 optimised executable code is able to produce solutions for more instances than the non-optimised code. It is apparent from these figures that the smallest probability for successful search is 0.03 (non-optimised code) and 0.33 (optimised code).

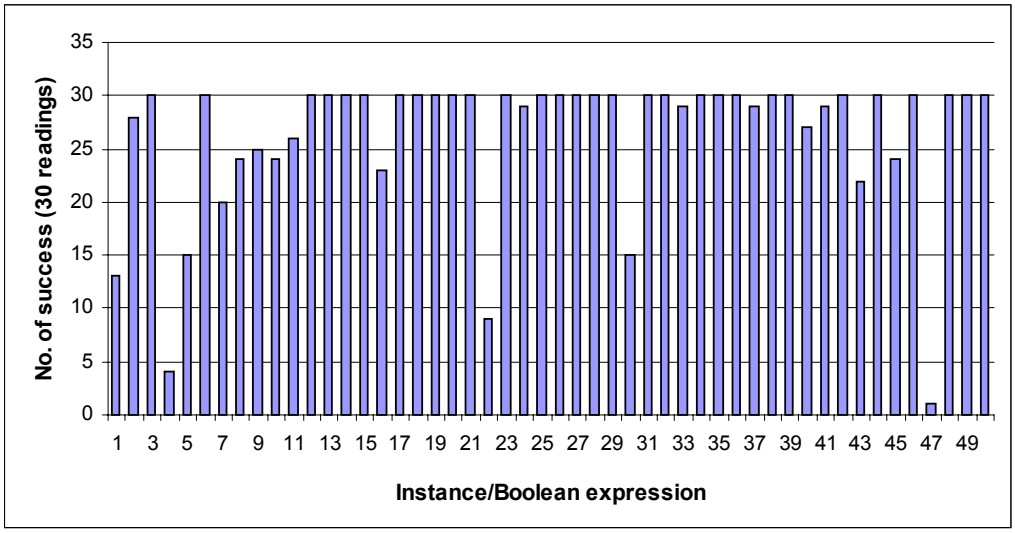


Figure 5.10a: No. of successful searches in 30 executions for GA BSAT on grid for non-optimised machine code.

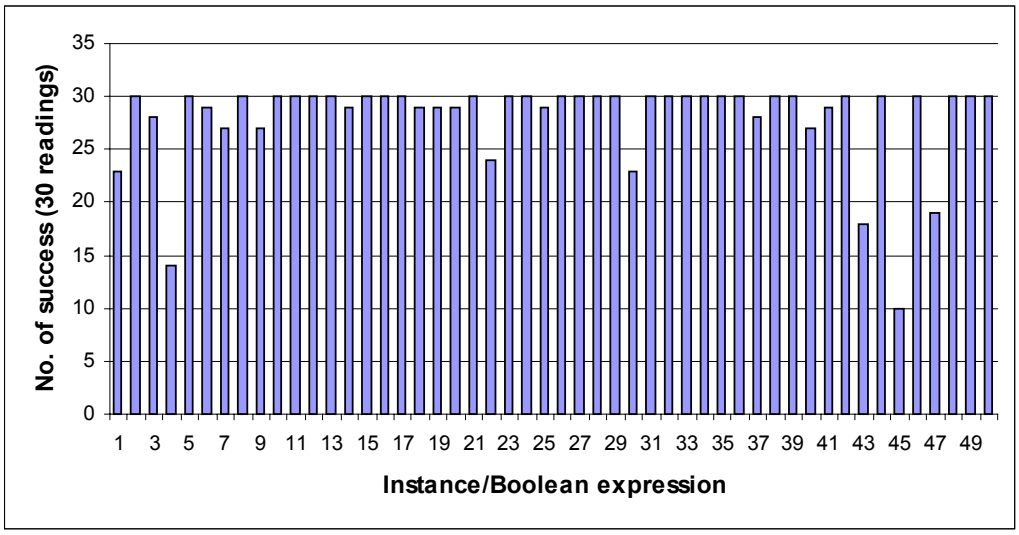


Figure 5.10b: No. of successful searches in 30 executions for GA BSAT on grid for O3 optimised machine code.

5.10.2 Non-optimised vs. O3 optimised code for successful search

The average time of the successful searches is plotted in figure 5.11 for comparison.

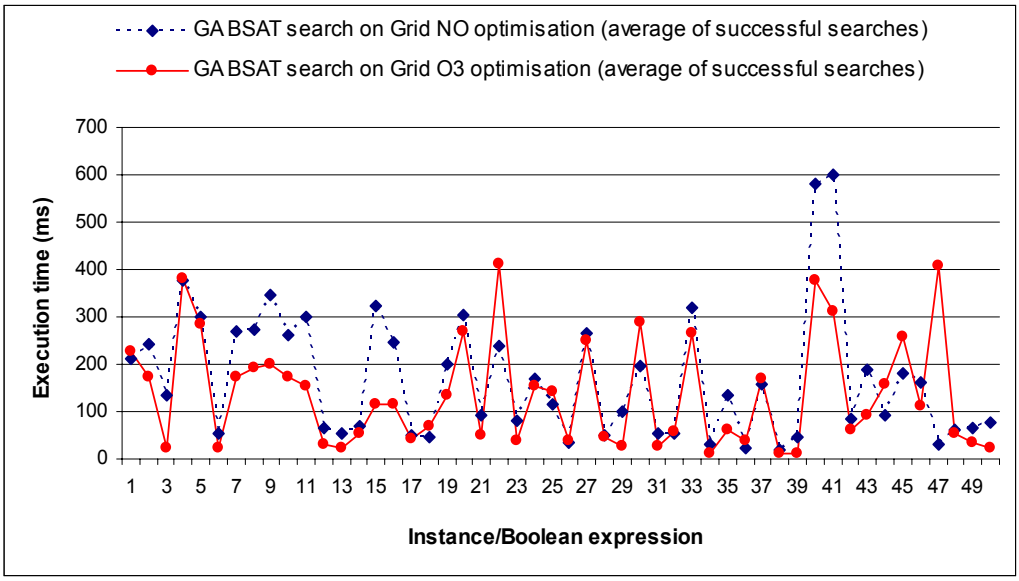


Figure 5.11: Execution time of GA BSAT search on grid for non-optimised and O3 optimised machine code.

There is no straightforward relationship between non-optimised and O3 optimised code. It is obvious that O3 optimised code exhibits lower execution time for more that 35 instances and maximum execution time stays below 400 ms.

5.10.3 Error bars of non-optimised and O3 optimised machine codes

The error bars for non-optimised and O3 optimised code are represented by figure 5.12 and 5.13, respectively. In both of the cases, standard deviation (error bar) is low for execution time less than 100 ms.

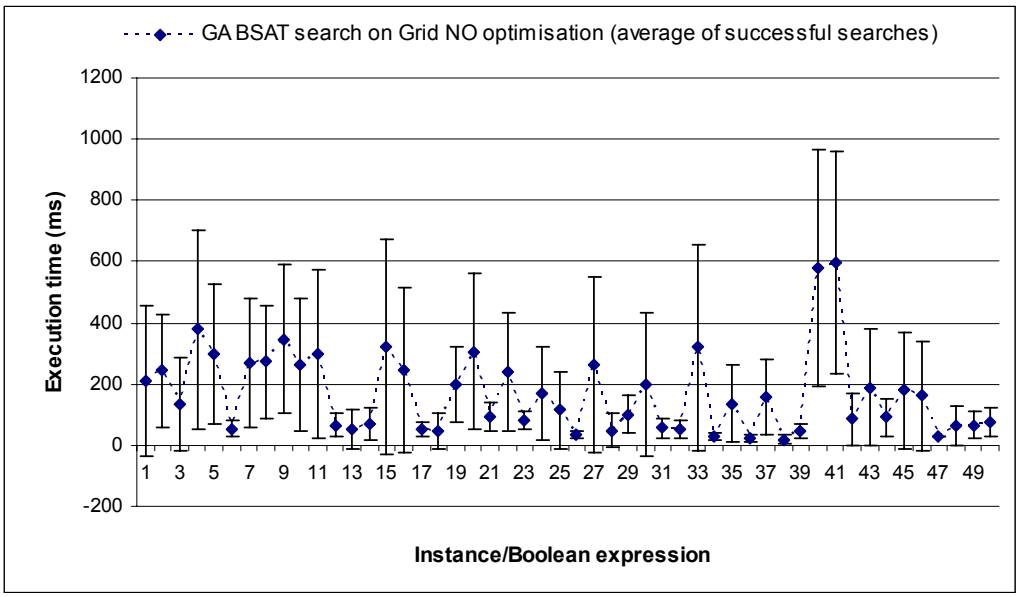


Figure 5.12: Error bars of execution time of GA BSAT search on grid for non-optimised machine code.

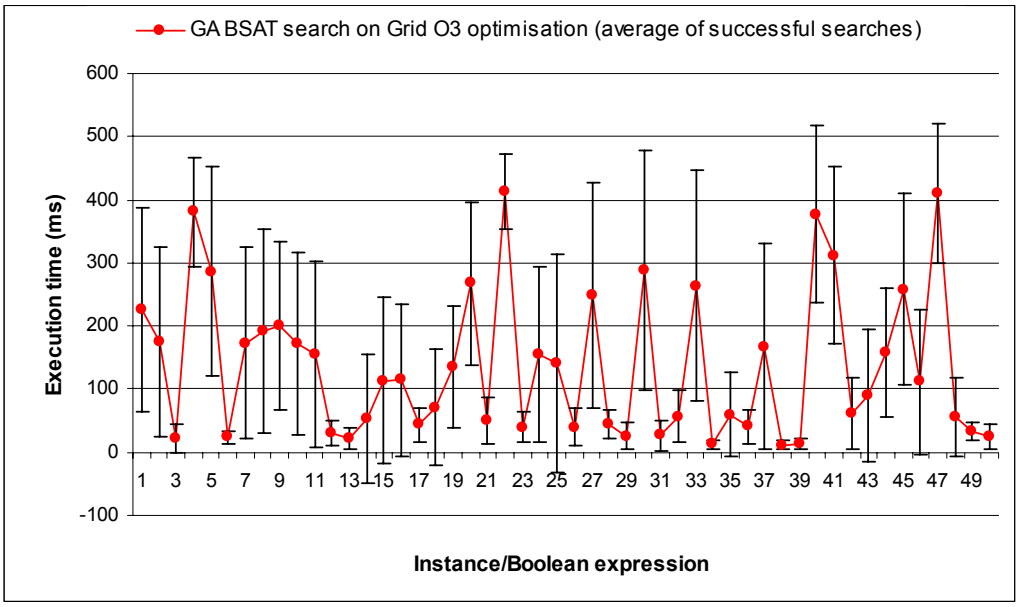


Figure 5.13: Error bars of execution time of GA BSAT search on grid for O3 optimised machine code.

5.11 GA BSAT search on single computer vs. on grid

It is clearly understood that if the GA BSAT algorithm finds a solution on single computer and on a computer on the grid with the same time *T*, the grid will take more time (SOAP overhead) to return the solution to the client. Therefore, overall completion time will be higher than a single computer. Furthermore, because of random characteristics of GA BSAT search, successful searching time can not be predicted. Figure 5.14 and 5.15 show that the maximum execution time for successful search is higher for non-optimised code (≈1100ms) than that (≈600ms) of O3 optimised code. But, no clear correlation can be identified between the single computer and the grid approach.

5.11.1 Non-optimised code

In general, the GA BSAT algorithm showed superior performance on the grid implementation. Figure 5.14 sketches the scenario that in 400 ms time, the grid found solutions for 48 instances, whereas single computer found 45 instance solutions.

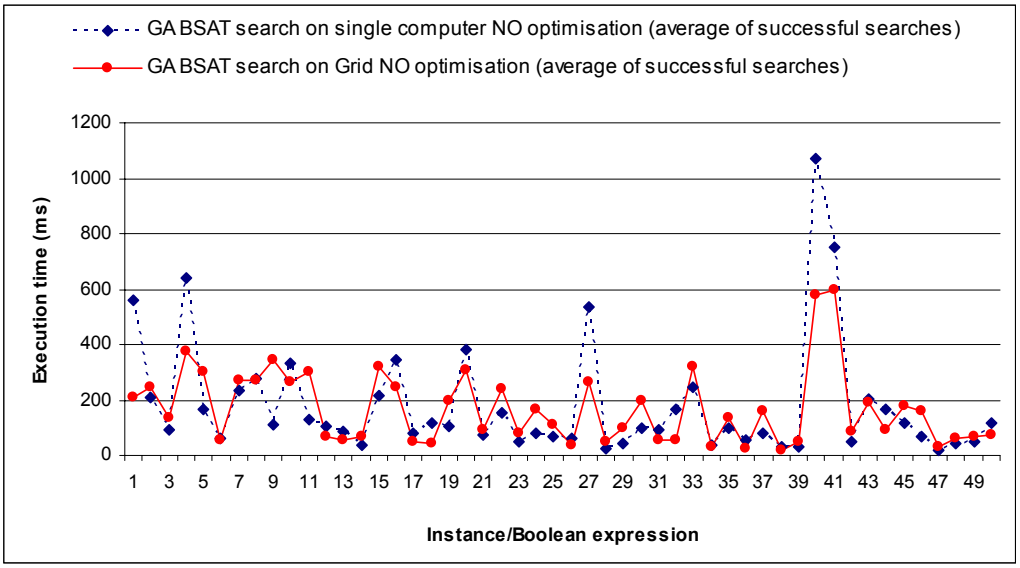


Figure 5.14: Execution time of GA BSAT search on single computer and grid for non-optimised machine code.

5.11.2 O3 optmised code

For the O3 optimised code, the single computer approach exhibits better execution time than the grid implementation. Figure 5.15 shows that for most of the instances, successful search execution times lie within 200 ms for GA BSAT on single computer.

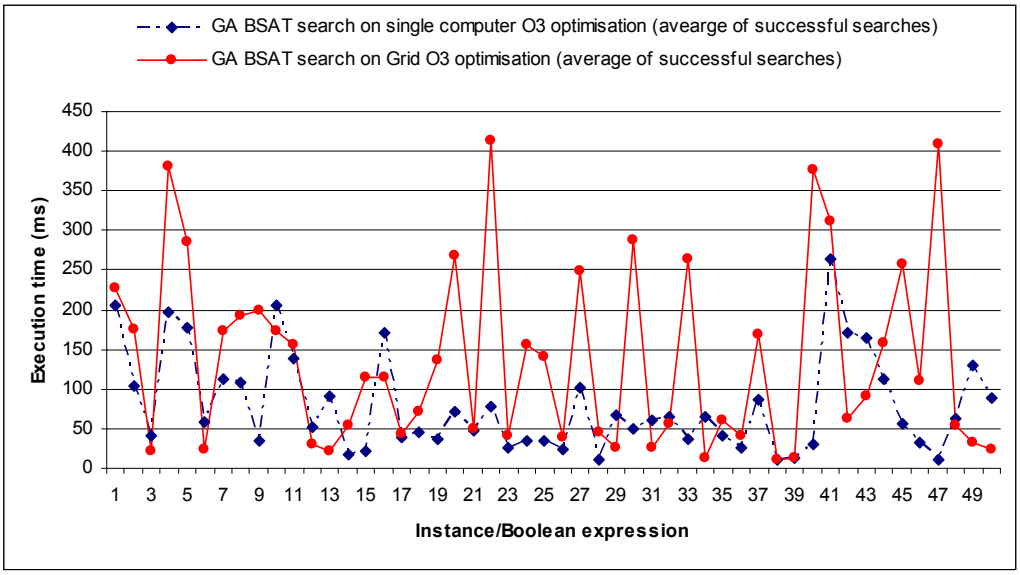


Figure 5.15: Execution time of GA BSAT search on single computer and grid for O3 optimised machine code.

5.11.3 Maximum execution time for un-successful search

Figure 5.16 and 5.17 show the comparison of maximum execution time (unsuccessful search) for non-optimised and O3 optimised machine code, respectively. The sub-generations should take approximately $\frac{1}{3}$rd time of the entire generations to execute in case of unsuccessful search. Figure 5.16 and 5.17 both support this statement.

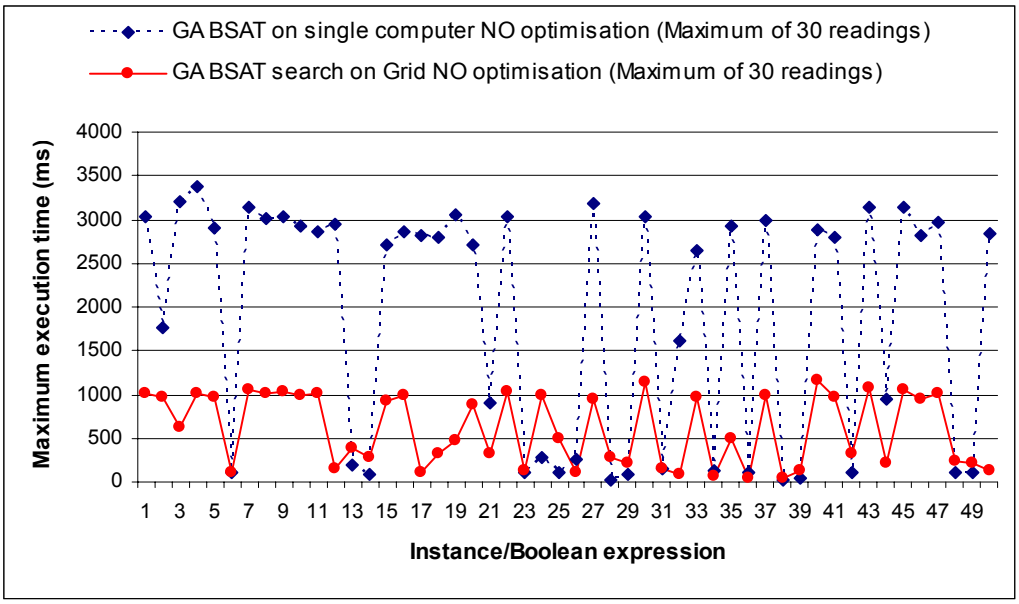


Figure 5.16: Maximum execution time of GA BSAT search on single computer and grid for non-optimised machine code.

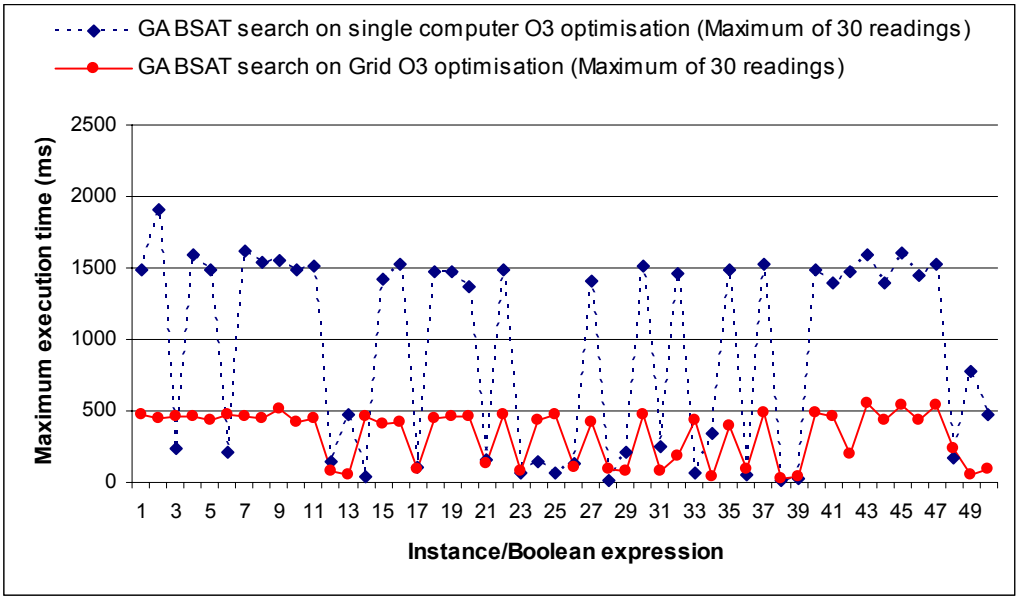


Figure 5.17: Maximum execution time of GA BSAT search on single computer and grid for O3 optimised machine code.

5.11.4 File size

Figure 5.18 represents the comparison among all the executable files for GA BSAT search. For GA BSAT on single computer, O3 optimisation has almost no effect on file size. On the other hand, O3 optimised files are much larger than the non-optimised versions. In general for both server and client, the following relation can be identified

GA BSAT *search O3 optimised grid file size* ≈ *1.8* * GA BSAT *search non-optimised grid file size*

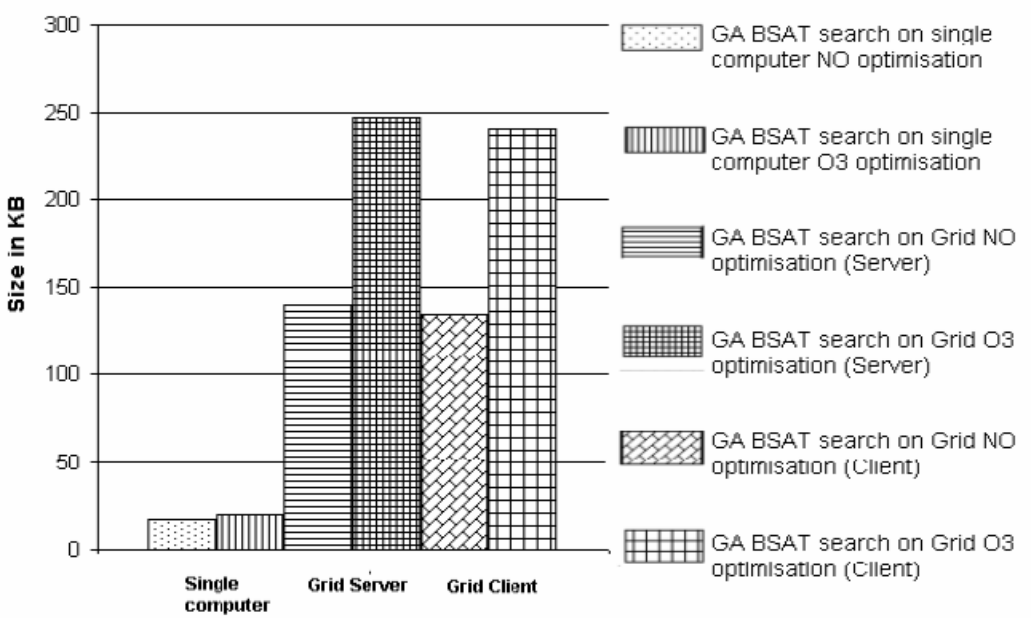


Figure 5.18: Comparison of GA BSAT search executable file size on single computer and on grid (Server and Client)

5.12 Summary

This chapter has presented all the results of execution of the exhaustive and GA BSAT search on single computer and on grid in terms of number of successful search, execution time, error bars, maximum execution time, file size *etc*. The following chapter contains the conclusion of the report and future work plan.

Chapter 6

Conclusion and Further work

6.1 Discussion

This thesis has developed and analysed various methods and technologies of grid computing which is of importance to the BSAT problem.

The objective of the project was to partition a processing intensive task on the computers of a grid to take advantage of distributed execution. BSAT is one of the most studied NP complete problems for verification and testing. Therefore, it was chosen as the computation intensive task for this investigation.

The same algorithm is executed on a single computer and on a grid containing three similarly configured computers. In case of the grid, each of the 3 sub-tasks is independent and deals with $\frac{1}{3}$rd of the entire task. However, executing on the grid incurs some time overhead for marshalling and transmission over a Local Area Network (LAN).

The exhaustive BSAT algorithm explores the entire search space in a linear fashion and does not exhibit any type of intelligence. On the other hand, an artificial GA has been used for optimisation and searching using the theory of evolution. An algorithm for BSAT problem based on GAs has been proposed and implemented both on a single computer and on a grid. The GA BSAT algorithm refines the feasible solutions by fitness function to construct a desired solution.

The execution times for the non-optimised and O3 optimised machine codes generated by the *gcc* compiler are observed for all cases. Besides, the executable file sizes are also compared.

The exhaustive BSAT search supports the following relationship on single computer and on the grid

Execution time of non-optimised Exhaustive BSAT *search* ≥

*2 * Execution time of O3 optimised Exhaustive* BSAT *search*

The BSAT problem is highly suitable as a grid application since each sub-task can be executed independently. The grid implementation shows much better performance than a single computer where the instance has multiple solutions. For non-optimised and optimised machine code it was found that

*Execution time on single computer ≈ 3 * Maximum execution time on grid*

The GA BSAT search algorithm demonstrated diverse results for different cases. These are listed below.

- Single computer - non-optimised vs. optimised machine code: No straightforward relation was identified between non-optimised and O3 optimised machine code. But, for most of the instances, O3 optimised code takes less time to execute.
- Grid - non-optimised vs. optimised machine code: No clear correlation was established between non-optimised and O3 optimised code. It is obvious that O3 optimised code exhibits lower execution time for more that 35 instances.

- Non-optimised machine code - single computer vs. grid: In general, the GA BSAT algorithm showed superior performance on the grid by finding solutions for more instances than single computer.
- O3 optimised machine code - single computer vs. grid: The grid demonstrates worse execution time than single computer.

6.2 Further work

The GA BSAT search algorithm uses a simple fitness function that returns the number of satisfied clauses. The following researches can be carried out to observe the results in future

- Many applications describe the problem as a multi-valued SAT problem. For example, in logic verification it is often desirable to describe *don't care* as a third value other than 0 and 1. By introducing the third value, the problem can be very efficiently formulated. [31].
- Other technologies, for instance, XML-RPC, CORBA, DCOM, can be compared with SOAP in distributing the sub-tasks over the grid computers.
- Development of deterministic GA BSAT search algorithm with intelligent fitness function that will ensure to find the solution if there is any.
- Generalised version of the client application where the main task can be partitioned in *N* sub-tasks depending on the number of available nodes of the grid at that time.

6.3 Conclusion

The exhaustive BSAT search is slower on single computer than on the grid for both non-optimised and optimised machine code. In case of the GA BSAT search, the non-optimised code performs well on the grid than on single computer. But, the grid implementation was unable to show any convincing improvements for O3 optimised GA BSAT search algorithm. However, it should be noted that GA does not ensure that it will find a solution if there is any. The GA BSAT search supports this fact since it could not find any solution during executions.

Reference

[1] Michael R. Garey, David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman & Company, San Francisco, CA: Freeman, 1979, ISBN: 0716710455.

[2] Bill P. Buckles and Frederick E. Petry, *Genetic algorithms*, IEEE Computer Society Press, CA, USA, 1992, ISBN 0818629355.

[3] I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations", International J. Supercomputer Applications, 15(3), 2001.

[4] Thomas Kropf, Hans-Joachim Wunderlich, "A Common Approach to Test Generation and Hardware Verification Based on Temporal Logic", Proceedings of IEEE International Test Conference 1991, USA, October 26-30, 1991, IEEE Computer Society 1991, pp: 57-66.

[5] L. Zhang, C. Madigan, M. Moskewicz and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver", Proceedings of International Conference on Computer Aided Design (ICCAD2001), San Jose, CA, Nov. 2001.

[6] M. Velev, and R. Bryant, “Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors,” Proceedings of the Design Automation Conference, July 2001.

[7] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. “Symbolic Model Checking without BDDs,” Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99), number 1579 in LNCS. Springer-Verlag, 1999.

[8] I. Foster, C. Kesselman, J. Nick, S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration", Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.

[9] http://www-106.ibm.com/developerworks/grid/library/gr-fly.html - Last accessed in Aug 2004.

[10] http://www.gridcomputing.com/gridfaq.html - Last accessed in Aug 2004.

[11] http://www.redbooks.ibm.com/redpapers/pdfs/redp3613.pdf - Last accessed in Aug 2004.

[12] http://www.intel.com/update/departments/servers/sv12031.pdf - Last accessed in Aug 2004.

[13] Allan Snavely, Greg Chun, Henri Casanova, Rob F. Van der Wijngaart, Michael A. Frumkin, "Benchmarks for Grid Computing: A Review of Ongoing Efforts and Future Directions", ACM SIGMETRICS Performance Evaluation Review, March 2003, Volume 30 Issue 4, Page 27 - 32.

[14] http://www.sun.com/products-n-solutions/edu/whitepapers/pdf/gridcomputing_architecture.pdf - Last accessed in Aug 2004.

[15] http://globus.org/about/faq/general.html - Last accessed in Aug 2004.

[16] http://www.tessella.com/Literature/Supplements/PDF/gridcomputing.pdf - Last accessed in Aug 2004.

[17] http://www.mcnc.org/rdi/files/GridComputingOverview.pdf - Last accessed in Aug 2004.

[18] http://www.sun.com/products-n-solutions/edu/events/archive/hpc/2003presentations/heidelberg/S10_Wolfgang_Gentch.pdf - Last accessed in Aug 2004.

[19] http://www.seeren.org/content/news/docs/GridComputing-SunGridEngine-N1-AndySchwierskott.pdf - Last accessed in Aug 2004.

[20] Douglas Thain and Miron Livny, "The Ethernet Approach to Grid Computing", Proceedings of the 12th IEEE Symposium on High Performance Distributed Computing, Seattle, WA, June, 2003.

[21] http://www.microsoft.com/serviceproviders/whitepapers/xml.asp - Last accessed in Aug 2004.

[22] Luis F. G. Sarmenta, Sandra Jean V. Chua, Paul Echevarria, Jose Mari Mendoza, Rene-Russelle Santos, Stanley Tan, and Richard Lozada, "Bayanihan Computing .NET: Grid Computing with XML Web Services", Global and Peer-to-Peer Computing on Large Scale Distributed Systems Workshop (GP2PC), 2nd ACM/IEEE International Symposium on Cluster Computing and the Grid (CCGrid '02), Berlin, Germany, May 2002.

[23] http://xml.coverpages.org/soap.html - Last accessed in Aug 2004.

[24] http://www.microsoft.com/mind/0100/soap/soap.asp - Last accessed in Aug 2004.

[25] http://www.nwfusion.com/details/526.html - Last accessed in Aug 2004.

[26] http://www.nwfusion.com/details/531.html - Last accessed in Aug 2004.

[27] http://techrepublic.com.com/5100-6296-1042699.html - Last accessed in Aug 2004.

[28] Iouliia Skliarova, António B. Ferrari, "A SAT Solver Using Software and Reconfigurable Hardware", Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition (DATE 2002).

[29] Witold Pedrycz, Giancarlo Succi, and Ofer Shai, “Genetic–Fuzzy Approach to the Boolean Satisfiability Problem”, IEEE transactions on evolutionary computation, vol. 6, no. 5, October 2002.

[30] J. Hartmanis, “On computational complexity and the nature of computer science,” Communications of the ACM*,* vol. 37, pp. 37–43, 1994.

[31] Cong Liu, Matthew W. Moskewicz, and Andreas Kuehlmann, “A Search Algorithm for Multi-Valued Satisfiability solver”, ECS219B Logic Synthesis Class Project Report.

[32] D.A. Plaisted and S. Greenbaum. “A structure preserving clause form translation”, Journal of Symbolic Computation, vol. 2, issue 3, 1986, pp. 293-304.

[33] Yulik Feldman, Nachum Dershowitz, Ziyad Hanna, "Parallel Multithreaded Satisfiability Solver: Design and Implementation", Proceedings of the 3rd International Workshop on Parallel and Distributed Methods in verification PDMC 2004, Sep 2004, London, UK.

[34] M. Davis and H. Putnam, “A computing procedure for quantification theory,” Journal of the ACM, vol. 7, pp. 102–215, 1960.

[35] M. Davis, G. Logemann, and D. Loveland, “A machine program for theorem proving,” Communications of the ACM, vol. 5, pp. 394–397, July 1962.

[36] L. Zhang and S. Malik, "The Quest for Efficient Boolean Satisfiability Solvers", Proceedings of 14th Conference on Computer Aided Verification (CAV2002), Copenhagen, Denmark, July 2002.

[37] David A. Plaisted, Armin Biere, Yunshan Zhu, “A Satisfiability Procedure for Quantified Boolean Formulae”, Journal of Discrete Applied Mathematics, Vol. 130, No. 2, August 2003.

[38] Donald Chai, Andreas Kuehlmann, "A Fast pseudoBoolean Constraint Solver", **Proceedings of** 40th Design Automation Conference (DAC'03), Anaheim, CA, USA, June 2003, pp 830.

[39] J. P. Marques-Silva, "The Impact of Branching Heuristics in Propositional Satisfiability Algorithms", Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA), September, 1999.

[40] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation", IEEE Transactions on Computers 35 (8), 1986, pp. 677-691.

[41] Abusaleh M. Jabir, Dhiraj K. Pradhan, "MODD: A New Decision Diagram and Representation for Multiple Output Binary Functions", Design, Automation and Test in Europe Conference and Exhibition Volume II (DATE 2004), France, Paris, Feb 2004, pp. 1388-1389.

[42] C. Berman, L. Trevillyan, "Functional comparison of logic designs for VLSI circuits", Proceedings of International Conference on Computer Aided Design, 1989, pp. 456-459.

[43] A. Kuehlmann, A. Srinivasan, D. LaPotin, "Verity - a formal verification program for custom CMOS circuits", IBM Journal of Research and Development 39 (1995), pp. 149-165.

[44] H. Zhang, "SATO: An efficient propositional prover", International Conference on Automated Deduction (CADE'97), no. 1249 in LNAI, Springer-Verlag, 1997, pp. 272-275.

[45] A. Gupta, P. Ashar, "Integrating a Boolean Satisfiability checker and BDDs for combinational verification", Proceedings of VLSI Design 98, 1998, pp. 222-225.

[46] G. Stalmarck, M. Saflund, "Modeling and verifying systems and software in propositional logic", in: B. K. Daniels (Ed.), Safety of Computer Control Systems (SAFECOMP'90), Pergamon Press, Oxford, 1990, pp. 31-36.

[47] M.W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in Proceedings of the 38th ACM/IEEE Design Automation Conference, Las Vegas, Nevada, June 2001, pp. 530–535.

[48] Ying Zhao, Sharad Malik, Matthew W. Moskewicz, Conor F. Madigan, "Accelerating Boolean Satisfiability through Application Specific Processing", Proceedings of International Symposium on Systems Synthesis (ISSS2001), Montréal, Québec, Canada, Sep, 2001, pp. 244-249.

[49] Wahid Chrabakh and Rich Wolski, "GrADSAT: A Parallel SAT Solver for the Grid", University of California, Santa Barbara (UCSB) Computer Science Technical Report, Feb 2003, No. 5.

[50] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski, "The GrADS Project: Software Support for High-Level Grid Application Development", International Journal of High Performance Applications and Supercomputing, 15(4):327–344, 2001.

[51] http://www.satlive.org/satcompetition/2002/submittedbenchs.html - Last accessed in Aug 2004.

[52] http://www.ececs.uc.edu/sat2002/sat2002-challenges.tar.gz - Last accessed in Aug 2004.

[53] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning,* Reading, MA: Addison-Wesley, 1989.

[54] K. A. De Jong and W. M. Spears, “Using genetic algorithms to solve NP-complete problems,” Proceedings of 3rd International Conference on Genetic Algorithms, San Mateo, CA: Morgan Kaufmann, 1989, pp. 124–132.

[55] Zadeh, L.A., "Fuzzy Sets", Information and Control", Vol. 8, No. 3, June 1965, pp. 338-353.

[56] Bellman, R.E., and Zadeh L.A., “Decision making in a fuzzy environment”, Management Science, Vol. 17B, No. 4, 1970, pp. 141-64.

[57] Kenneth A. De Jong and William M. Spears, "Using Genetic Algorithms to Solve NP-Complete Problems", Proceedings of the 3rd international Conference on Genetic algorithms, USA, 1989, pp. 124-132.

[58] http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html - Last accessed in Aug 2004.

[59] http://stephane.carrez.free.fr/doc/gcc_3.html#SEC13 - Last accessed in Aug 2004.

[60] http://computing.ee.ethz.ch/sepp/gcc-3.3.1-mo/gcc_13.html#SEC13 - Last accessed in Aug 2004.

[61] http://developer.apple.com/documentation/DeveloperTools/gcc-3.3/gcc/Option-Index.html#Option%20Index - Last accessed in Aug 2004.

Appendix A:

Header files for exhaustive search

Contents of the header file: bsat.h

```
// converts a 32 bit number into an array of 0 and 1
void convert(unsigned long v, unsigned int sol[])
{
   unsigned long mask = 1;
   int i;

   for (i=0; i<20; ++i)
   {
      if ((v & mask) == 0)
         sol[19-i] = 0;
      else
         sol[19-i] = 1;
      mask = mask << 1;
   }
}



// returns clause value for a particular solution
int find_clause_value (int sol[], int ter, char expr[][100])
{
   unsigned int tv = 0;
   int v;

   for (v=0; v<VARIABLE; ++v)
   {
      if (expr[ter][v]==EMPTY)
         continue;
      else           if           (islower(expr[ter][v])&&!sol[v]           ||
!islower(expr[ter][v])&&sol[v])
         return 1;
   }
   return 0;
}



// computes objective values of a generation
int find_fitness (int sol[], char expr[][100], int cla)
{
   int fit=0, t;


   fit = 0;
   for (t=0; t<cla; ++t)
   {
      fit += find_clause_value (sol, t, expr);
   }
   return fit;
}
```

Contents of the header file: sat_read.h

```
/* This function reads the benchmark from file */

int load_expr (char e[][100], int original[], int complemented[])
{
   FILE *in;
   int count=0, v, i, j;
   char ch;

   in = fopen ("/home/hasan/v20_c91.txt", "r");
   if (!in)
   {
      printf ("Unable to open the file. Exiting...\n");
      return -1;
   }

   for (i=0; i<100; ++i)
   {
      original[i] = complemented[i] = 0;
      for (j=0; j<25; ++j)
         e[i][j] = '-';
   }

   while (!feof(in))       // remove c and p lines
   {
      ch = fgetc (in);
      if (ch=='c'||ch=='p')
      {
         while (ch!='\n')
            ch = fgetc(in);
      }
      else
      {
         ungetc (ch, in);
         break;
      }
   }

   while (!feof(in))       // now read the clauses
   {
      ch = fgetc (in);
      if (ch=='%')    // end marker
      {
         printf ("End of file reached & count = %d", count);
         break;
      }
      ungetc (ch, in);
      ++count;     // new clause
      for (i=0; i<3; ++i)
      {
         fscanf (in, "%d", &v);
         if (v>0)
         {
            e[count-1][v-1]='A';
            original[v-1]++;
         }
         else if (v<0)
         {
            v = -v;
            e[count-1][v-1]='a';
            complemented[v-1]++;
         }
      }
      fscanf (in, "%d", &v);  // remove last 0
   }
   fclose (in);

   printf ("Displaying %d clauses in each line: \n", count);
   for (i=0; i<count; ++i)
   {
      for (j=0; j<20;++j)
         printf ("%c ", e[i][j]);
      printf ("\n");
   }
   return count;
}
```

Appendix B:

Implementation code for exhaustive search on single computer

Main program: bsat_exhaustive.c

```
#define VARIABLE   20
#define EMPTY   '-'

#include <string.h>
#include <time.h>
#include <ctype.h>
#include <stdlib.h>
#include <stdio.h>

#include "sat_read.h"
#include "bsat.h"


int main ()
{
   char expression[100][100];
   int clause, original[100], complemented[100], solution[100];
       int fitness, i;
   unsigned long value=0;
   clock_t t1, t2;

   // loads clauses from benchmark file
   clause = load_expr (expression, original, complemented);
   printf ("No of Clauses: %d\n", clause);
   printf ("Exhaustive BSAT is running. Please wait...\n");

   t1 = clock ();
   for (value=0; value < 1048576; ++value)
   {
      convert (value, solution);
      fitness = find_fitness (solution, expression, clause);

      if (fitness == clause)
      {
         printf ("\nA solution is found: \n");
         for (i=0; i<20; ++i)
            printf ("%d ", solution[i]);
         t2 = clock ();
         printf     ("\nTime     required:     %f     ms\n",     (float)(t2-
t1)/(float)CLOCKS_PER_SEC*1000.0);
         return;
      }
   }
   printf ("The expression is not Satisfiable");
}
```

Appendix C:

Implementation code for exhaustive search on grid environment

Client program: calcclientc

```
#include "soapH.h"
#include "calc.nsmap"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <pthread.h>
#include <time.h>
#include <sys/times.h>


const char server_1[] = "http://10.0.0.10:25000";
const char server_2[] = "http://10.0.0.11:30000";
const char server_3[] = "http://10.0.0.12:35000";

struct argument
{
   struct soap mysoap;
   char server[100];
   char unknown[100];
   char filename[100];
   double op1, op2, result;
};


struct argument soap_1, soap_2, soap_3;


void* ex_bsat(void* what)
{
   struct argument *ptr = (struct argument*)what;
   struct timeval t1, t2;
   double soap_exe_time;

   gettimeofday (&t1, NULL);
   soap_init(&ptr->mysoap);
   // file reading time by server is returned in result
   soap_call_ns__msh(&ptr->mysoap, ptr->server, ptr->unknown, ptr->filename,
ptr->op1, ptr->op2, &ptr->result);
   gettimeofday (&t2, NULL);

   soap_exe_time = (double)(t2.tv_sec-t1.tv_sec)*1000.0+(double)(t2.tv_usec-
t1.tv_usec)/1000.0;  // elapsed time
   soap_exe_time -= ptr->result;   // elapsed time - file reading time
   ptr->result = soap_exe_time; // soap overhead + execution time
}


int main(int argc, char *argv[])
{
   pthread_t  p_thread_1, p_thread_2, p_thread_3;
   int thr_id_1, thr_id_2, thr_id_3;
   void *retval_1, *retval_2, *retval_3;
   float tmp;

   strcpy (soap_1.server, server_1);
   strcpy (soap_1.unknown, "");
   strcpy (soap_1.filename, argv[1]);
```

```
   soap_1.op1 = 0.0;
   soap_1.op2 = 349525.0;
   soap_1.result = 0.0;

   strcpy (soap_2.server, server_2);
   strcpy (soap_2.unknown, "");
   strcpy (soap_2.filename, argv[1]);
   soap_2.op1 = 349525.0;
   soap_2.op2 = 699050.0;
   soap_2.result = 0.0;

   strcpy (soap_3.server, server_3);
   strcpy (soap_3.unknown, "");
   strcpy (soap_3.filename, argv[1]);
   soap_3.op1 = 699050.0;
   soap_3.op2 = 1048576.0;
   soap_3.result = 0.0;

// fprintf (stderr,"All servers are running concurrently\n");

   thr_id_1 = pthread_create(&p_thread_1, NULL, ex_bsat, (void*)&soap_1);
   thr_id_2 = pthread_create(&p_thread_2, NULL, ex_bsat, (void*)&soap_2);
   thr_id_3 = pthread_create(&p_thread_3, NULL, ex_bsat, (void*)&soap_3);

   pthread_join(p_thread_1, &retval_1);
   pthread_join(p_thread_2, &retval_2);
   pthread_join(p_thread_3, &retval_3);

   // compute the minimum time among three servers and print
   if (soap_1.result < soap_2.result)
      tmp = soap_1.result;
   else
      tmp = soap_2.result;

   if (tmp < soap_3.result)
      printf ("%0.2f ", tmp);
   else
      printf ("%0.2f ", soap_3.result);

   return 0;
}
```

Server program: calcserver.c

```
#include "soapH.h"
#include "calc.nsmap"

#define VARIABLE   20
#define EMPTY   '-'

#include <string.h>
#include <time.h>
#include <ctype.h>
#include <stdlib.h>
#include <stdio.h>

#include "sat_read.h"
#include "bsat.h"


int main(int argc, char **argv)
{
   int m, s; /* master and slave sockets */
   struct soap soap;

   soap_init(&soap);
   if (argc < 2)
   {
      fprintf (stderr, "No port number, serving as CGI application\n");
      soap_serve(&soap); /* serve as CGI application */
   }
   else
   {
      m = soap_bind(&soap, NULL, atoi(argv[1]), 100);
      if (m < 0)
      {
```

```
         soap_print_fault(&soap, stderr);
         exit(-1);
      }
      fprintf(stderr, "Socket connection successful: master socket = %d\n",
m);
      for ( ; ; )
      {
         s = soap_accept(&soap);
         fprintf(stderr, "Socket connection successful: slave socket = %d\n",
s);
         if (s < 0)
         {
            soap_print_fault(&soap, stderr);
            exit(-1);
         }
         soap_serve(&soap);
         soap_end(&soap);
      }
   }
   return 0;
}


int ns__msh(struct soap *soap, char filename[], double a, double b, double
*result)
{
   char expression[100][100];
   int clause, original[100], complemented[100], solution[100], fitness, i;
   double value, file_time, exe_time;
   struct timeval t1, t2;
   clock_t clk_t1, clk_t2;

   // loads clauses from benchmark file
   gettimeofday (&t1, NULL);
   clause = load_expr (filename, expression, original, complemented);
   printf ("Benchmark instance: %s\n", filename);
   printf ("No of Clauses: %d\n", clause);
   printf ("Exhaustive BSAT is running. Please wait...\n");
   printf ("Start value %0.0f\n", a);
   printf ("Finish value %0.0f\n", b);
   gettimeofday (&t2, NULL);
   file_time = (double)(t2.tv_sec-t1.tv_sec)*1000.0 + (double)(t2.tv_usec-
t1.tv_usec)/1000.0;
   *result = file_time;

   clk_t1 = clock();
   for (value=a; value < b; ++value)
   {
      //printf ("Current value %0.0f\n", value);
      convert ((long int) value, solution);
      fitness = find_fitness (solution, expression, clause);

      if (fitness == clause)
      {
         printf ("A solution is found: \n");
         for (i=0; i<VARIABLE; ++i)
            printf ("%d ", solution[i]);
         clk_t2 = clock ();
         exe_time = (double)(clk_t2-clk_t1)/(double)CLOCKS_PER_SEC*1000.0;
         printf ("\nExecution time: %f ms\n", exe_time);
         printf ("File time: %f\n", file_time);
         break;
      }
   }
   if (value==b)
      printf ("The expression is not Satisfiable\n");
   else
      printf ("Satisfiable\n");
   return SOAP_OK;
}
```

Appendix D:

Header files for GA BSAT search

Contents of the header file: ga_bsat.h

```
// initialiser
ga_bsat  (char  expr[][MAX_VAR],  int  var,  int  ter,  int  original[],  int
complemented[], int population)
{
   int s, v;

   variable = var;
   term = ter;
   solution = population;
   new_solution = population;

   for (s=0; s<ter; ++s)
      strcpy (expression[s], expr[s]);

   // assign the first solution based on original and complemented
   for (v=0; v<variable; ++v)
   {
      if (complemented[v]>original[v])
         current_generation[0][v] = 0;
      else
         current_generation[0][v] = 1;
   }
   // assign other n-1 solutions with random values
   for (s=1; s<solution; ++s)
   {
      for (v=0; v<variable; ++v)
         current_generation[s][v] = rand()%2;
   }
}


unsigned int find_clause_value (unsigned int generation[][MAX_VAR], int sol,
int ter)
{
   unsigned int tv = 0;
   int v;

   for (v=0; v<variable; ++v)
   {
      if (expression[ter][v]==EMPTY)
         continue;
      else     if     (islower(expression[ter][v])&&!generation[sol][v]     ||
!islower(expression[ter][v])&&generation[sol][v])
         return 1;
   }
   return 0;
}


// computes objective values of a generation
void find_fitness (unsigned int generation[][MAX_VAR], int sol, unsigned int
fitness[])
{
   unsigned int fit;
   int s, t;

   for (s=0; s<sol; ++s)
   {
      fit = 0;
      for (t=0; t<term; ++t)
      {
         fit += find_clause_value (generation, s, t);
      }
```

```
      fitness[s] = fit;
   }
}


// copies best new_sol parents to new_generation
void choose_parents ()
{
   int considered[MAX_SOL];
   unsigned int fit_value;
   int take_it, s, p, k;

   for (s=0; s<solution; ++s)
      considered[s] = FALSE;
   for (s=0; s<new_solution && s<solution; ++s)
   {
      fit_value = 0;
      take_it = NONE;
      for (p=0; p<solution; ++p)
      {
         if (considered[p])
            continue;
         else if (current_fitness[p]>fit_value)
         {
            fit_value = current_fitness[p];
            take_it = p;
         }
      }
      if (take_it==NONE)
      {
         printf ("Error in computing new generation and exiting...");
         exit (1);
      }
      else
      {
         for (k=0; k<variable; ++k)
            new_generation[s][k] = current_generation[take_it][k];
         considered[take_it] = TRUE;
      }
   }
}



// crosses sol and sol+1 at point p
void cross_parent (int p, int sol)
{
   unsigned int tmp;
   int v;

   for (v=0; v<p; ++v)
   {
      tmp = new_generation[sol][v];
      new_generation[sol][v] = new_generation[sol+1][v];
      new_generation[sol+1][v] = tmp;
   }
}

// multipoint cross over
void cross_over()
{
   int point=1, i;

   for (i=0; i<new_solution-1; i=i+2)// cross between i and i+1
   {
      point = rand ()%variable;
      cross_parent (point, i);
      //++point;
   }
}



void next_generation()
{
   int taken_current[MAX_SOL], taken_new[MAX_SOL];
   unsigned int tmp_generation[MAX_SOL][MAX_VAR], tmp_fitness[MAX_SOL];
```

```
   int from_current, from_new;
   unsigned int fit_value;
   int i, j, k;

   for (i=0; i<solution; ++i)
      taken_current[i] = FALSE;
   for (i=0; i<new_solution; ++i)
      taken_new[i] = FALSE;

   for (i=0; i<solution; ++i)
   {
      fit_value = 0;
      from_current = from_new = NONE;
      for (j=0; j<solution; ++j) // current generation search
      {
         if (taken_current[j])
            continue;
         if (current_fitness[j] > fit_value)
         {
            fit_value = current_fitness[j];
            from_current = j;
         }
      }
      for (j=0; j<new_solution; ++j)  // new generation search
      {
         if (taken_new[j])
            continue;
         if (new_fitness[j] > fit_value)
         {
            fit_value = new_fitness[j];
            from_new = j;
            from_current = NONE;
         }
      }
      if (from_current==NONE && from_new==NONE)
      {
         printf ("Error in generating next generation and exiting...");
         exit(1);
      }
      if (from_current!=NONE)    // taken from current generation
      {
         for (k=0; k<variable; ++k)
            tmp_generation[i][k] = current_generation[from_current][k];
         tmp_fitness[i] = current_fitness[from_current];
         taken_current[from_current] = TRUE;
      }
      else // from new generation
      {
         for (k=0; k<variable; ++k)
            tmp_generation[i][k] = new_generation[from_new][k];
         tmp_fitness[i] = new_fitness[from_new];
         taken_new[from_new] = TRUE;
      }
   }
   for (i=0; i<solution; ++i)
   {
      for (j=0; j<variable; ++j)
         current_generation[i][j] = tmp_generation[i][j];
      current_fitness[i] = tmp_fitness[i];
   }
}


int finished (int* index)
{
   int i;

   for (i=0; i<solution; ++i)
   {
      if (term-current_fitness[i]  == 0)
      {
         *index = i;
         return TRUE;
      }
   }
   return FALSE;
}
```

```
void mutate_all ()
{
   int v, s;

   for (s=0; s<solution; ++s)
   {
      v = rand()%variable;
      if (current_generation[s][v])
         current_generation[s][v] = 0;
      else
         current_generation[s][v] = 1;
   }
}


void   take_snapshot   (unsigned   int   past[][MAX_VAR],   unsigned   int
current[][MAX_VAR])
{
   int s, v;

   for (s=0; s<solution; ++s)
      for (v=0; v<variable; ++v)
         past[s][v] = current[s][v];
}



void run_ga_bsat (unsigned long int iteration)
{
   int sol_index=NONE, count=0;
   int done, v;
   unsigned int i;


   find_fitness (current_generation, solution, current_fitness);

   for (i=0; i<iteration; ++i)
   {
      done = finished (&sol_index);   //  sol_index  =  solution  index  when
finished
      if (done)
      {
         printf ("Desired solution found: \n");
         for (v=0; v<variable; ++v)
            printf ("%u ", current_generation[sol_index][v]);
         printf ("\n");
         return;
      }

      choose_parents();
      cross_over();
      find_fitness (new_generation, new_solution, new_fitness);

      next_generation();

      ++count;
      if (count==100)    // mutate after 100 generations
      {
         mutate_all ();
         find_fitness (current_generation, solution, current_fitness);
         count = 0;
      }
   }
   printf ("Solution not found in %u generations\n", iteration);
}
```

Contents of the header file: sat_read.h

```
/* This function reads the benchmark instance from file v20_c91.txt */

int load_expr (char fname[], char e[][100], int original[], int
complemented[])
{
   FILE *in;
   int count=0, v, i, j;
   char ch;

   in = fopen (fname, "r");
   if (!in)
   {
      printf ("Unable to open the file. Exiting...\n");
      return -1;
   }

   for (i=0; i<100; ++i)
   {
      original[i] = complemented[i] = 0;
      for (j=0; j<25; ++j)
         e[i][j] = '-';
   }

   while (!feof(in))    // remove c and p lines
   {
      ch = fgetc (in);
      if (ch=='c'||ch=='p')
      {
         while (ch!='\n')
            ch = fgetc(in);
      }
      else
      {
         ungetc (ch, in);
         break;
      }
   }

   while (!feof(in))       // now read the clauses
   {
      ch = fgetc (in);
      if (ch=='%')    // end marker
      {
         printf ("End of file reached & count = %d", count);
         break;
      }
      ungetc (ch, in);
      ++count;     // new clause
      for (i=0; i<3; ++i)
      {
         fscanf (in, "%d", &v);
         if (v>0)
         {
            e[count-1][v-1]='A';
            original[v-1]++;
         }
         else if (v<0)
         {
            v = -v;
            e[count-1][v-1]='a';
            complemented[v-1]++;
         }
      }
      fscanf (in, "%d", &v);  // remove last 0
   }
   fclose (in);
   printf ("Displaying %d clauses in each line: \n", count);
   for (i=0; i<count; ++i)
   {
      for (j=0; j<20;++j)
         printf ("%c ", e[i][j]);
      printf ("\n");
   }
   return count;
}
```

Appendix E:

Implementation code for GA BSAT search on single computer

Main program: ga_bsat.c

```
#define MAX_CLA    100
#define MAX_VAR    100
#define MAX_SOL    100
#define EMPTY   '-'
#define NONE    -1
#define TRUE    1
#define FALSE   0
#define POPULATION 20
#define GENERATION 10000

   // global variables
   int variable, term, solution, new_solution;
   char expression[MAX_CLA][MAX_VAR];
   // current generation data struct
   unsigned int current_generation[MAX_SOL][MAX_VAR];
   unsigned int current_fitness[MAX_SOL];
   // new generation data struct
   unsigned int new_generation[MAX_SOL][MAX_VAR];
   unsigned int new_fitness[MAX_SOL];

#include <string.h>
#include <time.h>
#include <ctype.h>
#include <stdlib.h>
#include <stdio.h>

#include "sat_read.h"
#include "ga_bsat.h"

int main ()
{
   char expr[100][100];
   int clause, original[100], complemented[100], found;
   double file_time, exe_time;
   struct timeval t1, t2;
   clock_t clk_t1, clk_t2;
   time_t moment;

   // loads clauses from benchmark file
   gettimeofday (&t1, NULL);
   clause = load_expr (expr, original, complemented);
   gettimeofday (&t2, NULL);
   file_time  =  (double)(t2.tv_sec-t1.tv_sec)*1000.0  +  (double)(t2.tv_usec-
t1.tv_usec)/1000.0;

   time (&moment);
   srand (moment);
   clk_t1 = clock();
   ga_bsat (expr, 20, clause, original, complemented, POPULATION);
   found = run_ga_bsat(GENERATION);
   clk_t2 = clock ();

   exe_time = (double)(clk_t2-clk_t1)/(double)CLOCKS_PER_SEC*1000.0;
   if (found)
      printf ("%0.2f %0.2f ", 1.0, exe_time);
   else
      printf ("%0.2f %0.2f ", 0.0, exe_time);
   return 0;
}
```

Appendix F:

Implementation code for GA BSAT search on grid environment

Client program: calcclientc

```
#include "soapH.h"
#include "calc.nsmap"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <pthread.h>
#include <time.h>
#include <sys/times.h>

#define ITERATION  3333
#define POPULATION 20

const char server_1[] = "http://10.0.0.10:25000";
const char server_2[] = "http://10.0.0.11:30000";
const char server_3[] = "http://10.0.0.12:35000";

struct argument
{
   struct soap mysoap;
   char server[100]; // segmentation fault if char* is used
   char unknown[100]; // segmentation fault if char* is used
   char filename[100];
   double op1, op2, result;
};


struct argument soap_1, soap_2, soap_3;


void* ga_bsat(void* what)
{
   struct argument *ptr = (struct argument*)what;
   struct timeval t1, t2;
   double soap_exe_time;

   gettimeofday (&t1, NULL);
   soap_init(&ptr->mysoap);
   // file reading time by server is returned in result
   soap_call_ns__msh(&ptr->mysoap, ptr->server, ptr->unknown, ptr->filename,
ptr->op1, ptr->op2, &ptr->result);
   gettimeofday (&t2, NULL);

   soap_exe_time = (double)(t2.tv_sec-t1.tv_sec)*1000.0+(double)(t2.tv_usec-
t1.tv_usec)/1000.0;  // elapesd time
   soap_exe_time -= ptr->result;   // elapsed time - file reading time
   ptr->result = soap_exe_time; // soap overhead + execution time
}


int main(int argc, char *argv[])
{
   pthread_t  p_thread_1, p_thread_2, p_thread_3;
   int thr_id_1, thr_id_2, thr_id_3;
   void *retval_1, *retval_2, *retval_3;
   float tmp;

   strcpy (soap_1.server, server_1);
```

```
   strcpy (soap_1.unknown, "");
   strcpy (soap_1.filename, argv[1]);
   soap_1.op1 = (double)ITERATION;   // generations
   soap_1.op2 = (double)POPULATION;  // population
   soap_1.result = 0.0;

   strcpy (soap_2.server, server_2);
   strcpy (soap_2.unknown, "");
   strcpy (soap_2.filename, argv[1]);
   soap_2.op1 = (double)ITERATION;   // generations
   soap_2.op2 = (double)POPULATION;  // population
   soap_2.result = 0.0;

   strcpy (soap_3.server, server_3);
   strcpy (soap_3.unknown, "");
   strcpy (soap_3.filename, argv[1]);
   soap_3.op1 = (double)ITERATION;   // generations
   soap_3.op2 = (double)POPULATION;  // population
   soap_3.result = 0.0;

// fprintf (stderr,"All GA BSAT servers are running concurrently\n");

   thr_id_1 = pthread_create(&p_thread_1, NULL, ga_bsat, (void*)&soap_1);
   thr_id_2 = pthread_create(&p_thread_2, NULL, ga_bsat, (void*)&soap_2);
   thr_id_3 = pthread_create(&p_thread_3, NULL, ga_bsat, (void*)&soap_3);

   pthread_join(p_thread_1, &retval_1);
   pthread_join(p_thread_2, &retval_2);
   pthread_join(p_thread_3, &retval_3);

   // compute the minimum time among three servers and print
   if (soap_1.result < soap_2.result)
      tmp = soap_1.result;
   else
      tmp = soap_2.result;

   if (tmp < soap_3.result)
      printf ("%0.2f ", tmp);
   else
      printf ("%0.2f ", soap_3.result);

   return 0;
}
```

Server program: calcserver.c

```
#include "soapH.h"
#include "calc.nsmap"
#include <string.h>
#include <time.h>
#include <ctype.h>
#include <stdlib.h>
#include <stdio.h>

#define MAX_CLA    100
#define MAX_VAR    100
#define MAX_SOL    100
#define EMPTY   '-'
#define NONE    -1
#define TRUE    1
#define FALSE   0

   // global variables
   int variable, term, solution, new_solution;
   char expression[MAX_CLA][MAX_VAR];
   // current generation data struct
   unsigned int current_generation[MAX_SOL][MAX_VAR];
   unsigned int current_fitness[MAX_SOL];
   // new generation data struct
   unsigned int new_generation[MAX_SOL][MAX_VAR];
   unsigned int new_fitness[MAX_SOL];

#include "sat_read.h"
#include "ga_bsat.h"
```

```
int main(int argc, char **argv)
{
   int m, s; /* master and slave sockets */
   struct soap soap;

   soap_init(&soap);
   if (argc < 2)
   {
      fprintf (stderr, "No port number, serving as CGI application\n");
      soap_serve(&soap); /* serve as CGI application */
   }
   else
   {
      m = soap_bind(&soap, NULL, atoi(argv[1]), 100);
      if (m < 0)
      {
         soap_print_fault(&soap, stderr);
         exit(-1);
      }
      fprintf(stderr, "Socket connection successful: master socket = %d\n",
m);
      for ( ; ; )
      {
         s = soap_accept(&soap);
         fprintf(stderr, "Socket connection successful: slave socket = %d\n",
s);
         if (s < 0)
         {
            soap_print_fault(&soap, stderr);
            exit(-1);
         }
         soap_serve(&soap);
         soap_end(&soap);
      }
   }
   return 0;
}


int ns__msh(struct soap *soap, char filename[], double a, double b, double
*result)
{
   char expr[100][100];
   int clause, original[100], complemented[100];
   double file_time, exe_time;
   struct timeval t1, t2;
   clock_t clk_t1, clk_t2;
   time_t moment;

   // loads clauses from benchmark file
   gettimeofday (&t1, NULL);
   clause = load_expr (filename, expr, original, complemented);
   printf ("Benchmark instance: %s\n", filename);
   printf ("No of Clauses: %d\n", clause);
   printf ("No of generations: %f\n", a);
   printf ("No of population: %f\n", b);
   printf ("Genetic BSAT is running. Please wait...\n");
   gettimeofday (&t2, NULL);
   file_time = (double)(t2.tv_sec-t1.tv_sec)*1000.0 + (double)(t2.tv_usec-
t1.tv_usec)/1000.0;
   *result = file_time;

   time (&moment);
   srand (moment);
   clk_t1 = clock();
   ga_bsat (expr, 20, clause, original, complemented, b);
   run_ga_bsat(a);
   clk_t2 = clock();

   printf ("\n");
   exe_time = (double)(clk_t2-clk_t1)/(double)CLOCKS_PER_SEC*1000.0;
   printf ("\nExecution time: %f ms\n", exe_time);
   printf ("File time: %f\n", file_time);

   return SOAP_OK;
}
```