A COMPARISON OF THREADED VERSUS NON-THREADED COMPUTING ON THE SIEVE OF ERATOSTHENES ALGORITHM

A MINI-THESIS SUBMITTED IN PARTIAL FULFILMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE (INFORMATION TECHNOLOGY)

OF

THE UNIVERSITY OF NAMIBIA

BY

SUAMA UUSHONA

200925121

APRIL 2023

SUPERVISOR: PROF. WILLIAM SVERDLIK (SCHOOL OF COMPUTING,

UNIVERSITY OF NAMIBIA)

Abstract

Performance enhancement in computers is a constant challenge for computer engineers, emanating from the constantly changing needs of consumers. Around the 1970s and 80s, computer engineers started designing computer architectures with multiple processors onboard, in order to address the performance challenge. These newer architectures provided a platform that enabled multiprocessing at affordable retail prices. This in turn created an opportunity for software developers to enhance program performance by creating software that could leverage on the underlying architecture through parallelisation to provide a performance advantage over sequential programs. This was not always achieved, however, since parallel programs present more opportunities to generate overhead, which can limit, or even deteriorate a program's overall performance. In this study, an experimental analysis of two Sieve of Eratosthenes programs, one sequential and one parallel, was conducted in order to verify that parallelisation provided a computational advantage in the program under investigation, to establish whether a variation in the number of available processors had an effect on the overhead incurred, and lastly to investigate the mathematical nature of the overhead incurred. The study concluded that the parallel program provided a computational advantage over the sequential program for all threads computing prime numbers in the ranges greater than 10,000. The results of the study also determined that there existed a positive statistically significant relationship between the number of threads employed and the overhead incurred. In addition to this, the study also determined that overhead was mathematically quantifiable, but not in relation to the number of threads employed. Instead, it was learned that overhead is a product of the parallel program's execution time and the fraction of efficiency lost. All in all, the study certainly highlighted one benefit of parallelisation, namely performance enhancement.

TABLE OF CONTENTS

Abstract i

List of Tables v

List of Figures vi

List of Equations vii

List of Abbreviations viii

Acknowledgements ix

Dedications x

Declaration xi

1 Introduction 12

1.1 Background of the Study 12

1.2 Statement of the Problem 15

1.3 Objectives of the Study 16

1.4 Significance of the Study 16

1.5 Limitation of the Study 17

1.6 Delimitation of the Study 17

1.7 Definition of Terms 17

1.8 Outline of the Thesis 21

2 Literature Review 23

2.1 Parallel Computer Architectures 23

2.2 Types of Parallelism 26

2.3 Overhead in Computer Programs 29
2.4 The Sieve of Eratosthenes 32
2.4.1 Algorithm Pseudocode 35
2.4.2 Optimisations 36
2.4.3 Sources of Parallelism 40
2.4.4 Related Work 41
2.5 Evaluation of Parallel Algorithms 42
3 Research Methods 48
3.1 Research Design 48
3.2 Procedure 49
3.2.1 Program Development 49
3.2.2 Data Collection 50
3.3 Data Analysis 51
3.3.1 Performance Evaluation 51
3.3.2 Relationship between Processors and Overhead 52
3.3.3 Investigation of Overhead 53
4 Results & Discussion 54
4.1 Program Development 54
4.2 Program Performance 60
4.3 Overhead Function 66
4.4 Cost of Performance 68
4.5 Discussion 69

5 Conclusion ... 73

6 Recommendations ... 76

References ... 77

Appendix A – SoE Source Code ... 84

Appendix B – Program Execution Times ... 91

Appendix C – Results of Correlation Analyses ... 95

Appendix D – Cost of Performance ... 96

List of Tables

Table 1 Program execution time at 2^{30} 61

Table 2 mark() method execution time at 2^{30} 62

Table 3 Execution time, speedup, and efficiency at 2^{30} 64

Table 4 Average program execution time from 100 to 2,000,000,000 64

Table 5 Average mark() method execution time from 100 to 2,000,000,000 64

Table 6 Speedup for all ranges. 66

Table 7 Efficiency for all ranges. 66

List of Figures

Figure 1 Finding prime numbers between 2 and 100 using the SoE 34

Figure 2 SoE algorithm pseudocode 35

Figure 3 Optimised SoE algorithm pseudocode 40

Figure 4 Performance of the SoE program using the Boolean array 55

Figure 5 Performance of the SoE program using the Character array 55

Figure 6 IDD thread creation and termination 58

Figure 7 BDD thread creation and termination 59

Figure 8 BDD and IDD program performance 60

Figure 9 Overhead at 2^{30} 67

Figure 10 Overhead ratio versus efficiency 68

Figure 11 Cost of performance 69

List of Equations

Equation 1: Execution time as a function of the processors, algorithm, and problem size.......... 29

Equation 2 Speedup.......... 51

Equation 3 Efficiency.......... 52

Equation 4 Overhead.......... 67

Equation 5 Overhead as a function of the execution time and efficiency.......... 72

List of Abbreviations

ALU	–	Arithmetic Logic Unit
BDD	–	Block Data Decomposition
BLP	–	Bit Level Parallelism
CPU	–	Central Processing Unit
DLP	–	Data Level Parallelism
GPU	–	Graphics Processing Unit
I/O devices	–	Input/output devices
IDD	–	Interleaved Data Decomposition
ILP	–	Instruction Level Parallelism
MISD	–	Multiple Instruction Single Data
MIMD	–	Multiple Instruction Multiple Data
PCI	–	Peripheral Component Interconnect
PE	–	Processing Element
PLP	–	Process Level Parallelism
SISD	–	Single Instruction Single Data
SIMD	–	Single Instruction Multiple Data
SoE	–	Sieve of Eratosthenes
TLP	–	Thread-Level Parallelism

Acknowledgements

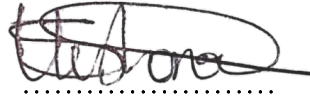
I would like to express my gratitude to the Lord without whom I would have never had the courage to complete my research. I would also like to thank my supervisor, Professor William Sverdlik for his tireless support during my research, and Professor Nicola Bidwell and Dr Suresh for their contribution to my studies, as well as my research. Finally, I would like to express my gratitude to the entire School of Computing and the Centre for Postgraduate studies for their forbearance and support throughout my studies.

Dedications

This work is dedicated to my mother, Meme Aino Nditodhino Angula, who is, and ever will be my source of inspiration.

Declaration

I, Suama N. N. Uushona, hereby declare that this study is my own work and is a true reflection of my research, and that this work, or any part thereof has not been submitted for a degree at any other institution. No part of this thesis may be reproduced, stored in any retrieval system, or transmitted in any form, or by means (e.g. electronic, mechanical, photocopying, recording, or otherwise) without the prior permission of the author, or The University of Namibia in that behalf. I, Suama N. N. Uushona, grant The University of Namibia the right to reproduce this thesis in whole or in part, in any manner or format, which The University of Namibia may deem fit.

Suama N. N. Uushona		April 2023
Name of Student	Signature	Date

1 Introduction

This chapter provides a brief background on parallel computing and introduces the problem statement, objectives, and scope of the study, along with a listing of key terms used in the study. The chapter is concluded with an outline of the thesis.

1.1 Background of the Study

As technology continues to evolve and become more complex and advanced, so do consumer requirements. Consumers now have a demand for higher performing general-purpose computers. As a result, engineers are constantly faced with the challenge of developing computers that meet consumer requirements at affordable retail prices. One of these requirements is performance.

In the 1960s and 1970s, advancements in performance surfaced in the form of supercomputers with multiple processors working together on shared data. Eventually, an idea emerged that performance could be further improved at a low cost by introducing multiple off-the-shelf microprocessors into the architecture. This theory was proven by the Caltech Concurrent Computation project in the mid-1980s, when it designed its first supercomputer which consisted of 64 off-the-shelf microprocessors (Intel, 2008). This design came to be known as Massively Parallel Processors (MPPs) and came to dominate the computing arena of its time, and only later yielding to cluster computers. The idea of designing computer architectures consisting of multiple processors eventually cascaded down to the design of general-purpose computers, where it was noted that increasing the number of processors in the architecture provided better performance, as opposed to increasing the processor clock frequency (Narang & Kothari, 2012). This led to the birth of multiprocessor, and later multicore

architectures. For the purpose of this study, both physical processors and processing cores are collectively referred to as processing elements (PEs), and an architecture consisting of multiple processing elements is henceforth referred to as a multi-PE architecture.

Having addressed the performance challenge at hardware level, software engineers were now presented with an opportunity to improve user experience by creating software that leveraged on the underlying architecture to provide improved program performance. Parallelisation, which is the design of computer programs and systems that utilize multiple processors simultaneously to solve a problem (Pacheco, 2011), is an emerging technique for improving performance in computer programs by allowing multiple parts of a program, or multiple programs to be executed simultaneously. This is not to be confused with concurrency, which although used synonymously to the term parallelism in some texts, is in fact distinguished by the fact that it is the interleaved execution of multiple tasks in the same timeframe. Unlike parallelisation, concurrency allows multiple tasks to start, execute, and complete in the same time interval, but not at the exact same time (Hughes & Hughes, 2008). For example, two tasks may execute within the same second, but at different fractions of the second, with the tasks swapping processor time until they complete their execution. Concurrency gives the illusion that more than one task is being executed at a given time, which is why it is possible to open more than one program on a uniprocessor computer. More importantly, concurrency does not necessarily imply the use of multiple PEs, as is the case in parallelism. Lastly, parallelism and concurrency are neither mutually exclusive, nor mutually inclusive, and can both be present in the same architecture at the same time (Jenkov, 2020).

One method of implementing parallelism at software level is through Thread Level Parallelism (TLP), which is also known as multithreading. Threads are the basic units by which processes can utilize the processor (El-Rewini & Abd-El-Barr, 2005). Each thread belongs to exactly one process; however, a process can have more than one thread. Different processes have their own blocks of memory, but threads belonging to the same process can share most of the process' resources, including memory and I/O devices (Padua, 2011). This allows for each thread to execute its own instructions independently of other threads and yet in parallel to those threads, which in turn allows for many flows of control, thereby achieving parallelism (Chhibber & Garg, 2014; El-Rewini & Abd-El-Barr, 2005; Johnson & Dinyo, 2015). In multicore computers, multiple threads can be executed on different processing cores (Chhibber & Garg, 2014; El-Rewini & Abd-El-Barr, 2005; Johnson & Dinyo, 2015), giving rise to terms such as "4 Core(s), 8 Logical Processor(s)". Such terms suggest that while the architecture contains four physical processing cores, the architecture has the ability to operate as though it consisted of eight processing cores (Intel, n.d.). Hyper-threading allows users to exploit these logical processors through the use of threads. To date, parallelisation of computationally intensive programs has become a popular technique for improving program performance.

While parallelisation certainly has the potential to improve program performance however, other factors such as overhead, the program's serial components, programming paradigm and programming style can still affect the programs overall performance (El-Nashar, 2011; Molia, 2014), sometimes so much so that the parallel program performs even worse than its sequential counterpart. In this study, one of the oldest know primality test algorithms, the Sieve of Eratosthenes (henceforth SoE), was

used in an experiment to determine whether the parallel program provided a computational advantage over its sequential counterpart, and to determine the cost of overhead associated with the performance enhancement.

1.2 Statement of the Problem

The main purpose of parallelism is to provide improved performance (Pasquali, 2013), meaning more work done in significantly less time than it would take a sequential program addressing the same problem. Eijkhout et al. (2014) state that two cores at a lower frequency can collectively provide the same throughput as a single processor operating at a higher frequency and are thus more efficient. This is because although individual cores in a multicore architecture do not run as fast as the highest performing single processor, they are able to execute independent tasks on separate cores simultaneously, thereby improving the system's overall performance by handling more workloads (Johnson & Dinyo, 2015).

Oaks and Wong (2004) concur and further state that threads can improve program performance since theoretically, a program that executes for one hour on one processor could require 30 minutes on 2 processors, and 15 minutes on 4 processors. Dar et al. (2018) and Silberschatz et al. (2013) argue that while an increase in the number of processors could reduce the program runtime, it would not necessarily satisfy the equation $\textit{parallel program execution time } (T_p) = \frac{\textit{sequential program execution time } (Ts)}{\textit{number of processors } (p)}$, due to the serial components within the program, as well as the overhead that is likely to be incurred. El-Nashar (2011) agrees with this observation and cautions that although it is generally understood that parallelising a sequential program would naturally result in a shorter execution time, this is not always the case. He further

cautions that it is just as likely that a parallel program running on a multi-PE architecture performs worse than its sequential counterpart.

As such, researchers are nonetheless faced with the challenge of determining whether or not it is worth parallelising certain programs. The aim of this study is therefore to investigate the effects of parallelism and the available processors on the Sieve of Eratosthenes, as well as to establish whether the overhead incurred is quantifiable.

1.3 Objectives of the Study

The objectives of this study were as follows:

i. To verify whether parallelisation provides a computational advantage over sequential processing in the SoE.
ii. To investigate how a change in the number of processors affects the overhead incurred.
iii. To investigate the mathematical nature of the overhead incurred, specifically whether overhead can be expressed as a function of the number of processors.

1.4 Significance of the Study

The aim of this study was to determine whether parallelising the SoE would improve the program's speedup, thus indicating improved program performance and a computational advantage over sequential processing. The study further sought to investigate the effect of the number of processors on the overhead incurred by the parallel program. Lastly, the study also sought to fill a gap in knowledge with regard to the mathematical nature of overhead incurred by the SoE as this has not been established.

1.5 Limitation of the Study

The study was limited to parallelisation on one MIMD computer of the type multicore. As such, the results may not scale to other parallel systems such as SIMD or MISD architectures. The multicore architecture was preferred over the multiprocessor architecture since the requirement of the study was for a multithreaded program. Multicore systems are able to execute multiple parts of a program, unlike multiprocessing systems, which are better suited to processing multiple programs simultaneously (Tabassum et al., 2016).

1.6 Delimitation of the Study

Since the purpose of this study was to observe the performance of a parallel program, in comparison to its sequential counterpart, one algorithm was considered sufficient for the experiment. The SoE was selected due to its simplicity in implementation, as well as the complexity that was once the reason for its popularity in the benchmarking of microcomputers (Peng, 1985). The SoE algorithm determines prime numbers in a range from 2 to a large integer *n*. For this study, we determined *n* to be 2^{30}. This includes, but does not restrict the study to the 2^{28} number range which Costa et al. (2014) say provided best performance for most of their parallel algorithms.

1.7 Definition of Terms

The following is a list of definitions for key terms used in the study.

Algorithm An algorithm is a finite sequence of instructions for performing for solving a computation problem (Rosen, 2012).

Cluster Computers Are homogeneous groups of computers which are interconnected by high-speed networks and are accessible as a configured pool of computing resources (Sadashiv & Kumar, 2011).

Collection A collection, which is also known as a container, is an object such as Lists and Arrays, which groups multiple related elements of the same data type into a single unit (Pohl & McDowell, 2006).

Composite Number A positive integer that is greater than one and is not prime (Rosen, 2012).

Concurrency Refers to the execution of multiple tasks in the same timeframe, though not necessarily at the same time. The term is used in some texts to include parallelism, however in this study, concurrency refers strictly to the interleaved execution of multiple tasks in a given timeframe (Jenkov, 2020).

Critical Section A section of program source code that accesses shared resources and must only be executed by one thread at any given time (Pacheco, 2011).

Factor A number that divides into another number without a remainder (Ore, 2017).

Grid Computers Are a type of parallel and distributed computing system wherein heterogeneous geographically distributed autonomous computing resources are coordinated dynamically at runtime, based on various factors including

availability, to perform computationally intensive tasks (Sadashiv & Kumar, 2011).

Hyper-threading An Intel technology hardware innovation that allows multiple thread to execute simultaneously on each core (Intel, n.d.).

Multicore A computer architecture consisting of multiple processors known as cores, which are placed on a single computer chip (Chhibber & Garg, 2014).

Multi-PE Architecture The term is used as a hyponym for the many computer architectures consisting of more than one physical processor or processing core.

Multiple A product resulting from multiplying a given integer *a* by another integer *b* (Ore, 2017).

Multiprocessing A technique for utilising multiple processing resources to collaborate on a large or complex problem at the same time (Hughes & Hughes, 2008).

Multiprocessor A computer system that contains more than one physical processor in its architecture and allows these processors to executed multiple tasks in parallel (Tabassum et al., 2016).

Multiprogramming A scheduling technique that allows for multiple computer programs to be executed at the same time (Hughes & Hughes, 2008).

Multithreading Implementation of parallelism by executing multiple parts of a program at the same time using multiple threads (Johnson & Dinyo, 2015).

Overhead The time processors spend on additional activities during a program's execution that are not relevant to accomplishing the actual task (Molia, 2014).

Parallel computing A form of computation whereby large or complex tasks are broken down into smaller tasks that can be executed simultaneously on multiple processors (Abdallah, 2013) .

Parallelisation The design of computer programs and systems that are able to utilize multiple processors simultaneously to solve a problem (Pacheco, 2011).

Prime Number An integer greater than 1 whose only positive factors are the numbers 1 and itself (Rosen, 2012).

Problem Size A problem size, which is also known as an input size is the number of characters an algorithm takes to write an input (Neapolitan, 2015). For an algorithm where the input is a collection as was the case in this study, the problem size can be the length of the collection used.

Processing Element The term is used to collectively refer to both processing cores and physical processors and is synonymous to the term processor in this study.

Program A program, which is also known as a computer program, is piece of software that can be executed on a computer to perform a specific task or accomplish a certain outcome.

Race condition A situation where the simultaneous access of a shared resource, usually memory, by multiple threads can result in

unexpected behaviour, including data inconsistencies (Gebali, 2011).

Seed The term is used in this study to refer to a prime number *k*, which is used to mark its multiples.

Seeding The term is used to refer to the process of marking composite numbers using a seed.

1.8 Outline of the Thesis

This thesis is outlined as follows.

Chapter 1 provides a brief background on parallel computing and presents the statement of the problem, the research objectives, the significance of the study, as well as the limitations and delimitations of the study. The chapter also provides a brief consideration on key terms used in the study.

Chapter 2 provides a review of literature relating to architectures that support parallel computing, types of parallelisms that can be implemented, factors that impact the performance of parallel programs, the algorithm under investigation, as well as the commonly used metrics for evaluating parallel program performance.

Chapter 3 details the manner in which the research was conducted. It further outlines the research approach, research instruments, and procedures used to conduct the study.

Chapter 4 presents the findings of the study. The chapter also analyses and discusses the data and explains how the findings relate to the research objectives.

Chapter 5 concludes the study with a summary of the key accomplishments of the study, and briefly discusses the lessons learnt, as well as gaps in knowledge identified.

Chapter 6 presents the recommendations for further research based on the research findings, as well as the gaps in knowledge identified.

2 Literature Review

This chapter briefly describes the architectures that support parallelism, the types of parallelism that can be implemented, as well as factors that affect the performance of parallel programs. The chapter also introduces the Sieve of Eratosthenes, as well as the techniques for efficiently implementing the algorithm, both as a sequential program and as a parallel program. The chapter is then concluded with a discussion on the evaluation of parallel algorithms.

2.1 Parallel Computer Architectures

Traditionally, software was written for sequential processing, meaning only one line of code could be executed at any given time. Parallelisation allows for multiple lines of code to be executed at the same time. Generally, it can be said that two prerequisites are to be met, in order for a program to be parallelised. The first is that the problem should be decomposable into tasks that can be executed independently, and the other is that the underlying architecture should support the required parallelism (Karimi, 2014).

In order to understand the role of the architecture in parallelisation, it is useful to understand the different classes of computer architectures. The most widely used system for classification of computer architectures is the Flynn's Taxonomy which was proposed by Michael Flynn in 1966 (El-Rewini & Abd-El-Barr, 2005). In his taxonomy, Flynn categorises the different computers based on the number of concurrent instructions and data streams that are supported by the architecture. These are the Single Instruction Single Data (SISD), Single Instruction Multiple Data

(SIMD), Multiple Instruction Single Data (MISD) and Multiple Instruction Multiple Data (MIMD) architectures.

SISD computers are the standard uniprocessor computers such as the Von Neuman machine. They consist of one uni-core processor and execute instructions one line of code after the other on a single set of data. SISD architectures are commonly found in personal computers and old mainframe computers (Rauf & Majeed, 2017).

SIMD computers consist of multiple processors all executing the same instruction on different data simultaneously. A SIMD architecture contains one control unit, and several Arithmetic Logical Units (ALUs). The control unit broadcasts instructions to the ALUs and the ALUs can either execute the instruction or become idle (Wilamowski & Irwin, 2016). SIMD architectures are based around the use of regular data structures such as vectors and matrices (Padua, 2011) and are therefore well suited for matrix-oriented computations such as the dot product of two vectors. Examples of SIMD architectures include high-end graphics cards like the GeForce FX series PCI Graphics cards which contain a Graphics Processing Unit (GPU) (Hennessy & Patterson, 2012).

MISD computers consist of multiple processors that are each able to apply different instructions to the same data. These are largely considered theoretical, however Roy et al. (2016) cite the Micron Automata processor as an example of the MISD class of computers, while Rauf and Majeed (2017) cite the Carnegie Mellon computer. MISD computers employ a pipelined technique to data processing, whereby instructions are processed in an assembly line fashion with processor A executing instruction A on the

data, and then sending it on to processor B to execute instruction B (Wilamowski & Irwin, 2016).

MIMD computers are the most flexible class of computers and exhibit the qualities of all the other architectures on the Flynn's Taxonomy. These computers are comprised of multiple autonomous PEs, meaning each processor is a fully-fledged Central Processing Unit (CPU) consisting of its own control unit and Arithmetic Logic Unit (ALU) (Pacheco, 2011). Each processor is therefore able to execute its own instructions on its own data simultaneous, independently, and asynchronously. This is referred to as the parallel approach to data processing. MIMD architectures have a broad application ranging from personal computers to networked computer clusters and grids (Irabashetti & Assistant, 2014). By definition of the term, SIMD, MISD and MIMD computers are all multi-PE architectures.

Multicore and multiprocessor architectures form part of the MIMD class of computers. The distinction between multiprocessor and multicore architectures lies in that multiprocessor architectures contain multiple physical processors, while multicore architectures consist of multiple processing cores placed on a single computer chip (Hughes & Hughes, 2008). Multiprocessor computers are therefore able to process multiple programs simultaneously, while multicore computers are able to process multiple parts of a program simultaneously (Tabassum et al., 2016). It can therefore be said that multiprocessor architectures are designed for multiprogramming, while multicore architectures are designed for multithreading. Both terms refer to multiprocessing (Wang & Ledley, 2016).

To date, programmers have devised methods of capitalising on multi-PE architectures by creating applications that leverage on the available PEs through parallelism to provide benefits such as:

1. Speedup, which is the notion that a program will run faster if multiple parts of it are executed at the same time, thereby reducing the overall execution time of the program.
2. Scalability in programs, as many threads can be employed to work on smaller parts of a larger task simultaneously and on as many processors as required.
3. Improved responsiveness in applications as part of a program can still continue to execute even though some parts of it are blocked (Silberschatz et al., 2013).

2.2 Types of Parallelism

Parallelism can be classified as either hardware-level, or software-level (Gebali, 2011). Hardware-level parallelism refers to parallelism which is provided by the computer's architecture such as in the case of multicore and multiprocessor architectures, or through hardware multiplicity as in the case of clusters and grids. Software-level parallelism on the other hand, refers to the control and data dependence of programs which is implemented during the development of the software program. In some cases, both hardware and software-level parallelism can be implemented for even better results (Wu, 1999).

There are many types of parallelism listed in various texts, however the most common forms are Bit Level Parallelism (BLP), Data Level Parallelism (DLP), Instruction Level Parallelism (ILP), and Thread Level Parallelism (TLP) which in some texts is also referred to as Task Level Parallelism (Wang & Ledley, 2016). Another form of

parallelism which is less popular is Process Level Parallelism (PLP) (Gebali, 2011). Implementing parallelism in software programs is considered a tedious task since programmers are required to explicitly define the parts of the program that should be executed in parallel, as well as those that should be executed sequentially. As such, processor architects first try to exploit parallelism that is available at hardware level (Padua, 2011).

BLP is a type of hardware level parallelism whereby the processor manipulates a group of bits per processor cycle. This is achieved through what is known as a word size. A processor's word size corresponds to the size of its registers and controls the amount of memory that the processor can access. The word size is what is referred to when it is said that a given processor has an n-bit processing capacity, where n historically referred to 8 or 16 bits, and more recently 32 or even 64-bits in personal computers. This means that the given processor can perform operations on n bits simultaneously, as opposed to bit-by-bit operations. In general, a processor with a larger word size has access to more memory and is therefore also able to process more data in each processing cycle. This then means that a processor with a larger word size is able to perform a given amount of work faster than a processor with a smaller word size, and thus provides better performance (Burd, 2016; Parsons, 2018; Wang & Ledley, 2016).

DLP is a type of parallelism usually associated with the SIMD architecture since it involves the execution of the same instruction on different data (Padua, 2011; Wang & Ledley, 2016). DLP can also be implemented on MIMD architectures by dividing data amongst the available PEs, with each PE executing the same instruction on its own portion of the data. DLP is thought to improve performance by ensuring that

multiple individual PEs work on smaller portions of the data set, which would be faster than if only one PE were to work on the entire data set (Abdallah, 2013). Click or tap here to enter text.In the context of a SIMD architecture, DLP is implemented at hardware level, however, in a MIMD architecture, it is usually implemented at software level.

ILP, which is commonly associated with the MISD architecture, is a form of parallelism where the execution of multiple instructions is overlapped. In some cases, the order in which the instructions are executed is also changed. A typical example would be a task consisting of three instructions to be carried out in three steps. If two of the three instructions are independent of the other instructions, then the overall task would then be executed in two steps, instead of the initial three, since two of the instructions would be executed simultaneously (Abdallah, 2013; Chhibber & Garg, 2014; Wang & Ledley, 2016). ILP can be implemented dynamically, whereby the hardware discovers and exploits the parallelism, or statically, whereby a software technology is used to detect parallelism in the instructions at compile time (Hennessy & Patterson, 2012).

TLP is another form of parallelism which in some texts is considered synonymous with the MIMD architecture. It involves the simultaneous execution of different instructions on different data. In this form of parallelism, a task is divided into sub-tasks which are then assigned to different threads and are executed by the processor(s) as independent tasks. TLP is implemented during the development of the program and is therefore a form of software level parallelism (Abdallah, 2013; Chhibber & Garg, 2014; Wang & Ledley, 2016). Although Thread Level Parallelism is largely

considered synonymous to Task Level Parallelism, some texts distinguish the two forms of parallelism by stating that Task Level Parallelism focuses on the distribution of threads across available PEs operating on the same, or different data, while Thread Level Parallelism is a capability available to software, which allows them to work with multiple threads at the same time (Abdallah, 2013).

Lastly, parallelism can also be implemented through PLP. PLP refers to the form of parallelism in which several processes or programs execute simultaneously on one or more machines (Gebali, 2011). Processes are able to reserve their own computer resources such as memory and registers, which enables multitasking and in other cases multiprogramming. This form of parallelism is also achievable at software level.

2.3 Overhead in Computer Programs

When parallelising a program, the expectation is that if *Tp* represents the time a program executes on *p* processors, then the ideal parallel program would satisfy the equation $T_p = \frac{Ts}{p}$, where T_s represents the execution time of the sequential program on one processor. However, this concept can be more accurately captured by representing the execution time as a function of the processors, algorithm, and problem size, as indicated in the equation below.

$$T(p, A, n) = \frac{T(1,\ A,\ n)}{p}$$

Equation 1: Execution time as a function of the processors, algorithm, and problem size

This is to say that the execution time *T* of a parallel program using *p* processors to execute an algorithm A for a problem size of *n* is equivalent to the execution time *T* of

a program using 1 processor to execute the same algorithm A on the same problem size n, all divided by the number of processors p used by the parallel program. This emphasises that the algorithm and the problem size have an effect on the performance of the parallel program and should therefore remain the same for both programs.

Evidently, the performance described here is idealistic and is not always achieved as there are many factors that can degrade the performance of a parallel program. Amongst these are the program's serial components, the programming paradigm, the programming style, overhead, and even the underlying architecture (El-Nashar, 2011). Of the aforementioned factors, overhead is the common denominator. There exist many kinds of overhead such as inter-process communication, idling, and excess computation. Grama et al. (2003) collectively refer to all forms of overhead as an overhead function and define an overhead function as the difference between the time collectively spent by all processors executing a parallel program and the time required by the fastest known sequential algorithm for solving the same problem on a single processor.

In parallel programs, a problem is divided into tasks which are then assigned to the available processors. Sometimes these processors require data from one another in order to execute their tasks. Once the tasks are executed, the partial results are again communicated and combined to produce a final result. This manner of communication between processors in referred to as inter-process communication and can be achieved either through message passing or through the shared-memory model. Inter-process communication impacts program performance by introducing delays in program execution in the form of the time required to distribute the workloads amongst the

processors, as well as through the transferring of data between processors (Molia, 2014). Communication overhead is considered the most significant source of overhead in parallel programs (Grama et al., 2003), although it is significantly less in multicore architectures than it is in multiprocessor architectures. This is because multicore processors are either on the same chip or on the same chip module (Gebali, 2011).

Oft times, some processors in parallel programs become inactive while other processors are busy. This is referred to as idling and it is caused by load imbalance, synchronization, or serial components within the algorithm. Load imbalance refers to the uneven distribution of labour whereby some processors are assigned larger workloads than others. This means that processors with smaller workloads may complete execution of their tasks before those with larger workloads, and become idle while waiting on the processors with larger workloads to complete execution of their tasks.

Parallelisation can also introduce race conditions in applications with shared memory, particularly in cases where threads are used. Threads that share the same parent process have access to the same data. This can introduce circumstances whereby the critical section of a program is executed at the same time by multiple threads. This can result in mutations to shared data, or an incorrect order of execution of tasks within a given program. Both situations can result in incorrect behaviour and are therefore addressed through synchronization. Synchronization ensures that shared resources are protected from concurrent access and that tasks within a program are executed in a certain order. In so doing, synchronization introduces overhead in two forms. The first is that while a certain resource is accessible to a given thread, other threads that require the same

resource at that point will become idle while waiting for the resource to become available again. The other is that some threads may become idle while waiting for other threads to complete execution of their tasks, in order for them to commence with execution of their own tasks (Pacheco, 2011).

Serial components, which Tokhi, Hossain and Shaheed (2003) refer to as inherent parallelism, refer to the parts of a parallel algorithm that cannot be parallelised. This means that only one thread may execute these components and while this is taking place, all other threads become idle.

Excess computation results from the additional operations that do not form part of the main execution but are nonetheless required by the parallel algorithm in order for it to accomplish the same objectives as a sequential algorithm addressing the same problem. These additional operations are not required in sequential programs since only one thread is required to execute all the tasks, but are required by parallel programs for synchronization and coordination of tasks amongst other tasks. Molia (2014) simplifies this definition by stating that excess computation is the difference in the computation required by a sequential approach and a parallel approach. The difference between excess computation and overhead is that excess computation refers to additional operations performed by the program, while overhead is the actual time spent on these additional operations.

2.4 The Sieve of Eratosthenes

In number theory, the prime factorization theorem states that every composite number can be represented as a product of its prime factors raised to a unique exponent. For

example, the integer 11,250 can be expressed as a factor of its prime factors as follows: 11,250 = 2 x 3 x 3 x 5 x 5 x 5 x 5, which is equivalent to $2 \times 3^2 \times 5^4$. This led to the understanding that prime numbers are the building blocks from which other numbers can be created mathematically (Ore, 2017). It is also noted that if *n* is a composite integer, then the prime numbers between 2 and *n* can be computed using the prime factors of *n* which are less than or equal to $\sqrt{n}$ (Rosen, 2012). The SoE, which is one of the oldest known prime sieve algorithms, applies this principle to the computation of prime numbers within a given range, starting with the number 2, until a given integer *n*. The sieve works by gradually striking out the multiples of 2, 3, 5 and the successive prime numbers up to $\sqrt{n}$. For example, given that *n* is 100, it is known that the prime factors of *n* must be less than $\sqrt{100}$, which is 10. Thus, only the numbers 2, 3, 5, and 7 are required to compute the prime numbers between 2 and 100. All multiples of these numbers are crossed out, leaving only the prime numbers unmarked. Figure 1 below illustrates this process graphically.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

(a) Multiples of 2 removed

	2	*3*	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

(b) Multiples of 3 removed

	2	*3*	4	*5*	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

(c) Multiples of 5 removed

	2	*3*	4	*5*	6	*7*	8	9	10
11	12	***13***	14	15	16	***17***	18	***19***	20
21	22	***23***	24	25	26	27	28	***29***	30
31	32	33	34	35	36	***37***	38	39	40
41	42	***43***	44	45	46	***47***	48	49	50
51	52	***53***	54	55	56	57	*58*	***59***	60
61	62	63	64	65	66	***67***	68	69	70
71	72	***73***	74	75	76	77	78	***79***	80
81	82	***83***	84	85	86	87	88	***89***	90
91	92	93	94	95	96	***97***	98	99	100

(d) Multiples of 7 removed

Figure 1 Finding prime numbers between 2 and 100 using the SoE

The SoE is of special interest in the computing fields due to the significance of prime numbers, in cryptographic and hash algorithms. The SoE is considered a simple algorithm to implement, albeit an inefficient one, since it requires O($n$loglog$n$) operations to compute the prime numbers, and n is said to increase exponentially as the number of digits increase. It is therefore considered an impractical algorithm for generating large prime numbers with hundreds of digits (Wirian, 2009). More efficient prime sieves such as the Sieve of Atkins have since been derived from this algorithm (Wirian, 2009), though the modified version of the SoE is still considered the most efficient prime sieve algorithm in recent years (Costa et al., 2014), and a modified version of the sieve is still relevant in modern day number theory (Wirian, 2009).

2.4.1 Algorithm Pseudocode

The most widely used SoE pseudocode is that which was proposed by Quinn (2004). For this study however, the researcher derived a pseudocode for the original SoE algorithm, as described in number theory (Rosen, 2012).

1. Create an unmarked list of natural numbers from 2 up to the limit (*n*).
2. Set *k* to 2, the first prime number on the list.
3. Repeat until $k \leq \sqrt{n}$
    a. Mark all multiples of *k* between *k* and *n*.
    b. Set *k* to the subsequent unmarked integer.
4. The unmarked numbers are prime numbers.

Figure 2 SoE algorithm pseudocode

O'Neill (2009) argues that this not the genuine SoE and explains that the SoE algorithm consists of only two steps as described below.

Step 1: Declare k to be prime, and cross off all the multiples of that number in the table, starting from k^2.

Step 2: Find the next number in the table after *k* that is not yet crossed off and set *k* to that number; and then repeat from Step 1.

O'Neill (2009) acknowledges that the concept of sieving from k^2 is also an optimisation, albeit a minor one. Bokhari (1986) adds that sieving up to $\sqrt{n}$ was introduced by Leonardo of Pissa over 1400 years later, and not by Eratosthenes. Nevertheless, the algorithm presented in Figure 2 is now widely adopted and is thus implemented in this study.

This pseudocode is herein referred to as the base algorithm. The base algorithm implements the SoE with minimal consideration for efficiency. Quinn (2004) provides a comprehensive analysis of the algorithm and proposed optimisations which Click or tap here to enter text. implemented in his sequential algorithm. Since parallel algorithms derive from sequential algorithms, it is safe to assume that an efficient parallel algorithm would also derive from an efficient sequential algorithm. As such, the best optimisations, as proposed by Quinn (2004) and Cordeiro (2012), are discussed here.

2.4.2 Optimisations

a. Odds Only

It is worth noting that all even numbers, with the exception of the number two, are composite. Excluding these numbers from the collection not only reduces the number of operations performed, thereby saving time, but it also saves on memory. Pande (2013) conducted a study between four variations of the prime sieve algorithm to determine how much more efficient the algorithms would become when not performing the primality test on even numbers. Sieve 1 sieved from the range of 2 to $n - 1$; Sieve 2.1.1 only performed the primality test for odd numbers using odd divisors, following the logic that if a given integer i is not divisible by 2, then i is also not divisible by $2j$; Sieve 2.1.2 reduced the range of prime divisors (k) from n to $n/2$ which means that if $n = 100$, then the largest $k = 50$; and lastly Sieve 2.1.3 limits the range of k to $\sqrt{n}$, meaning if $n = 100$ then the largest $k = 25$. Pande (2013) noted that Sieve 2.1.1, which did not include the even numbers, accurately completed its execution in nearly half the time required by Sieve 1 which included the even numbers. He further noted that the performance of the sieve was further improved when the value of k was

limited according to Sieves 2.1.2 and Sieve 2.1.3, with Sieve 2.1.3 providing the best performance.

b. Fast Marking

O'Neill (2009) argues that the genuine SoE algorithm uses the Trial Division method to compute the prime numbers. Trial division checks the primality of a given integer i using modulus operations of the form $i \bmod k$, where k represents a given prime number. This technique is considered to be inefficient however, since division operations are generally considered to be more time consuming in comparison to multiplicative operations (Levitin, 2012). The more efficient technique is Fast marking whereby multiples of k such as $2k$, $3k$, $4k$ and so forth, are computed and marked off. This eliminates the need to check if sums of the form $i + k$ are prime as in the Additive method. This technique was implemented by both Cordeiro (2012) and Costa et al. (2014).

c. Segmented Sieve

One of the challenges faced in generating prime numbers is that the prime numbers needed in security are significantly large (Borg & Dackebro, 2017). Helfgott (2019) states that at the time of his study, a standard computer could store approximately 10^{12} integers in the hard drive, 10^9 in the RAM, and 10^6 in the cache. This limits the number of prime numbers that can be generated to 10^9. Shawon and Pervez (2016) and Helfgott (2019) implemented what is known as a segmented sieve in their sequential algorithms. The segmented sieve addresses the memory limitation by allowing the range up to n to be manipulated in segments, block after block. This technique works by dividing n into segments not larger than $\sqrt{n}$, after which a collection smaller or

equal to the value of $\sqrt{n}$ is then created and populated with the integers for that specific segment. The sieve then finds the prime numbers up to $\sqrt{n}$, which corresponds to the first segment using the traditional SoE algorithm. These primes numbers are then stored in a collection of prime numbers which is herein referred to as the Primes Collection. The prime numbers in the Primes Collection are then used to compute the prime numbers in the remaining segments. In the study of Shawon and Pervez (2016), it was noted that this technique did not reduce the program execution time, however it enabled the program to generate prime numbers up to 1,000,500,000 within 24.94 seconds. The use of smaller blocks is said to improve the processor's access to memory, and can further improve cache hit rate, if the loops are exchanged, as explained in optimisation *d* below.

d. Reorganising of loops

The SoE consists of two loops. In the base algorithm, the outer loop selects the next k, while the inner loop iterates over the array to mark off the multiples of k. Reorganising the loops is said to improve the cache hit rate by ensuring that the multiples of all primes less than $\sqrt{n}$ are marked off in the current block before the next block is considered.

e. Minimising Duplications

An aspect that (Quinn, 2004a) considered, but did not explain in his study relates to the commutative law of multiplication which states that $i \times k = k \times i$. Möhring and Oellrich (2011) observed that when computing the multiples of a prime number, the result of $i \times k$ is only needed once, even though circumstances arise where i and k exchange values, for example $i = 3$ and $k = 5$ becoming $i = 5$ and $k = 3$. In this case,

the product of both computations is identical thus the second computation is unnecessary. They therefore proposed limiting these redundant computations by restricting $k \geq i$. (Costa et al., 2014) put forth that this can be achieved by starting the sieve at k^2 instead of $2k$. This method does not, however, cater for situations where the product of two different i and k combinations result in the same product, such as $i = 3$ and $k = 15$, and $i = 9$ and $k = 5$, whose product is 45 thus resulting in a redundant seeding operation.

f. Reducing Consumed Memory

Lastly, another aspect not mentioned by Quinn (2004) and Cordeiro (2012) is the data type. A data type specifies the kind of data that a variable can stored. More importantly, it occupies a fixed and predefined number of bits in memory. This applies to primitive data types such as int, char and boolean, as well as non-primitive data types such as collections. A light-weight data type occupies less memory, thereby enabling us to create a larger array, and ultimately increase the value of n. As such, one method of optimising the SoE would be to utilise the least memory intensive data type. The most commonly used data types for the SoE are a collection of type char or bool. The character data type requires two (2) bytes to represent the primality of a given number, while a boolean requires one bit of storage (Jana, 2005; Keogh, 2004). Some sources such as the (Oracle Java Documentation, 2021) state that a boolean represents one bit of information, but occupies an unspecified amount of memory. Nevertheless, the boolean is considered the less memory intensive of the two data types and therefore the more suitable data type for the collection in the optimised SoE, even though in the case of (Brumme, 2014), the C/C++ boolean array was slower than the character array. Various scholars have also investigated implementation of the SoE through bit-level

compression using a collection of type BitSet. A BitSet can represent a boolean value using one bit and is thus thought to be more memory efficient than a boolean array. BitSets can, however, have a cost in terms of lengthened execution time (E-maxx-eng, n.d.) and caution should therefore be applied when using them.

By applying optimisations a, b, and e, we derive the below pseudocode:

1. Create an unmarked list of 2 and the odd natural numbers up to the limit (*n*).
2. Set *k* to 3, the first odd prime number on the list.
3. Repeat until $k \leq \sqrt{n}$
    a. Mark all multiples of *k* of the form 3k, 5k, 7k etc. between k^2 and *n*.
    b. Set *k* to the next prime number.
4. The unmarked numbers are all prime numbers.

Figure 3 Optimised SoE algorithm pseudocode

2.4.3 Sources of Parallelism

Once an efficient sequential algorithm is developed, the next step is to derive a parallel algorithm. The main source of parallelism in the SoE stems from the marking of elements. Each thread can either be allocated a seed (*k*) which will be used to mark all the multiples of *k* in the array, or each thread can mark all the multiples of all *k*s within its allocated portion of the array. These techniques are referred to respectively as Interleaved Data Decomposition (IDD) and Block Data Decomposition (BDD) of the array (Quinn, 2004).

IDD makes it easy to determine which process controls a given index, however it can also lead to significant load imbalances as threads closer to will have fewer numbers to mark, as these would have been crossed out by previous threads. This can ultimately

lengthen program performance unnecessarily and is thus considered less efficient. In BDD, the array is divided into *p* contiguous blocks of roughly equal size where each thread finds the prime numbers in its own block. If the array of length *n* is not divisible by the number of processors *p*, some processors are allocated a few more elements than others. Large blocks can be distributed amongst the first or last neighbouring processors, or alternatively scattered amongst the processors. As such, this method provides better balanced loads and is considered a better alternative to IDD.

Other sources of parallelism include allowing each thread to populate its own portion of the array and print out the prime numbers it finds. While these techniques increase parallelism in the program, they also increase overhead in the form of communication overhead when the main thread broadcasts the next *k*; idling, when some processes wait for the main thread to communicate the next *k*; and also excess computation as a result of this implementation. (Wirian, 2009) proposes that the parallel algorithm be optimised to enable each process to compute its own *k*. This eliminates the need for broadcasting and further reduces idling.

2.4.4 Related Work

Other studies have also been conducted in the area of prime sieves, especially with respect to the implementation of prime sieve algorithms on GPU architectures. One of these is the study by Borg and Dackebro (2017) which was intended to investigate the difference in the performance of two parallel algorithms namely the SoE and Trial Division. The hypothesis was that any performance gained on the CPU architecture could possibly be improved on the GPU architecture since GPUs are specialised for massive parallelism on repetitive calculations. The conclusion of the study was that

the sequential programs which were executed on the CPU architecture, generally performed better than the parallel algorithms executing on the GPU architecture. Furthermore, it was noted that that Algorithm E, which sieved on the CPU while performing the Trial Division on the GPU, performed better than the algorithms executing solely on the GPU architecture.

Similarly, Månsson (2021) conducted a comparative study to investigate the performance of the SoE, Sieve of Sundaram, and Sieve of Atkin on CPU and GPGPU architectures. The findings of Månsson (2021) were in agreement with those of Borg and Dackebro (2017) in that the SoE algorithm performed better on the CPU architecture. In his study, Månsson (2021) observed that both the parallel and the sequential SoE algorithms performed best on a CPU architecture and worst on the GPGPU architecture in comparison to the Sieve of Sundaram and the Sieve of Atkin. Månsson (2021) speculates that this is because the SoE requires a master process to direct the activities in the algorithm. This is not possible on a GPU architecture since the architecture does not allow for direct control of the threads and thus renders the number of threads available irrelevant, as they cannot be put to effective use.

2.5 Evaluation of Parallel Algorithms

The purpose of parallelising programs is to improve performance, which in most cases is synonymous to doing more work in less time. Evaluating programs is important as it aids in determining whether it was worth it to parallelise the program and whether we have achieved any kind of performance enhancement. As part of the discussion on

program evaluation, three studies conducted on parallel computing are presented and the results of the evaluations are discussed.

The first is a performance analysis conducted by Fashanu et al. (2012), wherein the performance of a parallel program designed to simulate space weather was evaluated in terms of execution time, optimisation of results, and efficiency. The second is the study by Cordeiro (2012) on the SoE, wherein he proposes various techniques for optimising sequential and parallel SoE algorithms. He then proceeds to implement and evaluate the performance of the various optimisations. Finally, an expansion on Cordeiro's study is presented, wherein Costa et al. (2014) present more techniques for optimising the SoE and also present some optimisations for distributed SoE algorithms for heterogeneous architectures.

Sequential and parallel programs are evaluated differently because of the way instructions are processed, as well as the way resources are utilised. Sequential programs are evaluated based on the program's execution time as a function of the problem size, while parallel programs are evaluated based on the execution time as a function of the problem size, number of processors employed, as well as the overhead (Molia, 2014). The most widely used metrics for evaluating parallel programs are execution time, speedup, and efficiency.

Execution time is the time elapsed from when the first processor starts the program execution to when the last processor completes it. Execution time does not consider the number of processors employed nor the problem size, and is therefore not

considered a sufficient metric to determine how the number of processors and the problem size affect a program's performance (El-Nashar, 2011).

Since the main goal of parallelism is to obtain faster runtime, El-Nashar (2011) argues that the main criterion for consideration ought to be the speedup gained. Speedup is the ratio between the execution time of the best sequential program and that of a parallel program addressing the same problem. It is used to express how many times a parallel program works faster than its sequential counterpart. Speedup considers the number of processors, as well as the problem size. For a more accurate evaluation, it is recommended that a sequential program be used to evaluate speedup, and not a parallel program executed on one thread or process. This is because parallel algorithms contain excess computations, which can distort the speedup recorded. According to Post and Goosen (2001), a parallel algorithm executed serially is likely to perform worse than its sequential counterpart.

Speedup is also architecturally and algorithmically bound, which means that a program might display varying levels of speedup on different architectures, as well as based on the algorithm used (Molia, 2014). This is evidenced in the study of Costa et al. (2014) wherein the speedup of the recommended distributed algorithm was not as high as might be expected. This was attributed to the fact that one computer only had half the processing ability of the other and thus seemingly also contributed less to the overall speedup gained. Cordeiro (2012) and Costa et al. (2014) also observed that different SoE programs also attained varying speedup. For this reason, it is recommended that speedup be measured on the same computer for a sequential algorithm and its parallel counterpart (El-Nashar, 2011). Speedup can be sub-linear, linear, or super-linear. Sub-

linear speedup means that while the parallel program may still outperform its sequential counterpart, the speedup might be significantly less than what would be expected, considering the resources invested. Linear speedup is directly proportional to the resources invested, which is ultimately the speedup that one would expect when parallelising a program, while the rarer super-linear speedup occurs when the speedup achieved is higher than the resources invested (El-Nashar, 2011; Molia, 2014). The last metric is efficiency. Efficiency is the ratio between speedup and the number of processors, and it is used to measure the fraction of time which a processor spends on useful employment (Grama et al., 2003), or in other terms, how efficiently a program utilizes system resources to solve a problem (Molia, 2014).

Four noteworthy observations were made relating to the effect of processors and processes on the performance of parallel algorithms. These are discussed in the study of El-Nashar and Aljahdali (2013). The first observation is that while increasing the number of processors reduces the computation time, it also increases the communication time. In some cases, it may be that the communication time will be high such that it negates the decrease in computation time, which ultimately results in poor performance. This was the case with the distributed algorithms of Costa et al. (2014) which performed best at larger number ranges and not necessarily as the processors increased. They explain that in addition to network latency, the distributed algorithms experienced overhead resulting from initialization of processes and threads, which was higher for relatively small ranges than it was for larger ranges. In contrast, the parallel algorithms, which were executed on a shared memory architecture, were more efficient than the distributed memory algorithms, which amongst other factors was due to the lack of communication related overhead.

The second observation is that ideally, speedup should increase linearly while efficiency should decrease linearly in relation to the number of available processors. Gustafson (1988) explains that this is unfortunately not always the case as speedup tends to saturate, which in turn causes efficiency to drop as the number of processors increase. As a result, it is more common to observe sub-linear speedup in a program than linear or super-linear speedup. This is evident in the study of Fashanu et al. (2012) where speedup saturated at 4 processors, and reduced to 54% on eight (8) processors.

Thirdly, El-Nashar and Aljahdali (2013) observed that speedup and efficiency increase as the problem size increases on the same number of processors. This can be observed in the study of Costa et al. (2014) where all the parallel and distributed algorithms generally performed better as the range was increased. Costa et al. (2014) observed that most of the parallel algorithms reached their maximum performance at 2^{28}, while the distributed algorithms reached maximum performance at ranges larger than 2^{32} and anticipate that the most efficient distributed algorithm would perform best at 2^{64}.

Finally, Squyres (2004) also states that generally, maximum performance is attained when each process has its own processor. He further states that an application will run at its peak when the number of threads is less than or equal to the number of processors. This was noted by Costa et al. (2014) in their study, where the best parallel algorithm performed best when using 8 threads on a quad-core hyper-threading computer. They however also noted that the performance was not proportional to the number of threads, but rather to the number of real available cores.

From the three studies, it is evident that all the parallel and distributed algorithms performed better than the sequential algorithms, although in the studies of Cordeiro (2012) and Costa et al. (2014), some algorithms were better recommended than others. Another issue of note is that in the studies of Cordeiro (2012) and Costa et al. (2014), the parallel algorithms executed on one thread or one process still performed better than the sequential algorithm, but not by a large margin. This would then highlight the issue expressed by Post and Goosen (2001), that a parallel algorithm executed on one thread is not a sequential algorithm.

3 Research Methods

This chapter describes the research process used in this study, the technique used to develop the algorithms, as well as the methods and tools used to generate, collect, and analyse the data.

3.1 Research Design

This study was a quantitative research which employed the experimental research approach to compare the performance of a sequential algorithm and its parallel equivalent, and to investigate the nature of the overhead incurred by the parallel algorithm. Only one algorithm was used for this study, namely the Sieve of Eratosthenes (SoE). The SoE algorithm was selected due to its simplicity in implementation, as well as the complexity that was once the reason for its popularity in the benchmarking of microcomputers (Peng, 1985). Bokhari (1986) explains that the slightest overhead appears prominently in the performance data hence the algorithm also serves as a severe test of the capabilities of a parallel processor. In this study, the algorithm is used to investigate the effect of parallelism on an old algorithm, which as Borg and Dackebro (2017) emphasised, was designed for sequential processing.

The convenience sampling technique was used to select one quad-core computer with eight logical processors that is readily available to the researcher. The computer used for this study was an HP ProBook 470 G5 using a hyperthreaded Intel® Core™ i7-8550U processor at 1.80GHz with 12GB RAM. The researcher did not interact with the program during its execution; thus, the study was limited to the observation research instrument.

3.2 Procedure

3.2.1 Program Development

The programs used in this study were developed using the NetBeans 8.2 Integrated Development Environment (IDE) using the Java programming language. The sequential algorithm was based on the pseudocode provided in Figure 3. The best sequential algorithm was then used to develop the parallel algorithm. Parallelism was only implemented in the method which performed the seeding operations, herein referred to as the mark() method. The parallel program implemented IDD by extending the Thread class.

In the sequential program, the program starts by creating one array of type char called *list*, of the specified size *n*. The array is then initialised to '*0*', which indicates that the integer at the associated position has not been marked. Marking commences at position 3 of the array, however, the primality check is only performed for every second integer. This ensures that the primality test is only performed for odd numbers. All prime numbers are marked as '*p*', while the composite numbers are marked as '*c*'. Once the primality test is completed, all integers less than or equal to *n* which are marked as '*p*' are known to be prime numbers, as well as all unmarked integers greater than *n*. This is because prime numbers beyond *n* will not have had an opportunity to be used as a seed, therefore their status will remain unchanged.

The parallel algorithm operates in the same manner as the sequential algorithm; however, parallelism was applied in the seeding operations. As such, the *list* array was only created and initialised by the main thread, and not by individual threads. Once the *list* array was created, the required number of threads were also created. The

threads were assigned a seed as soon as a new prime number indicated by the '*p*' state was available. At that point, the thread commenced with marking all the factors of the seed assigned to it. This eliminated the need for synchronisation, and further ensured that there was no chance of two threads marking using the same prime number. Furthermore, it minimised the chance of two threads marking the same integer, since only unmarked integers, with a status of '*0*'can be actioned upon. Lastly, the process ensured 100% correctness in the integers identified as prime. On the other hand, this caused overhead in the form of delays associated with waiting for a seed to become available, as well as in the form of the status checks to determine whether to skip an integer. This overhead was considered to be acceptable. Once the seeding operations are concluded, the main thread prints all the prime numbers found.

3.2.2 Data Collection

The aim of the study was to evaluate the performance of the programs at the 2^{30} range. The programs were executed thirty (30) times each for this range so as to obtain a reasonable average program execution time for data analysis. In the case of the parallel program, the program was executed 30 times for each of the threads, ranging from 2 to 8. The execution time, as provided by the NetBeans IDE, was recorded in seconds. This time unit did could not capture the difference in performance between some ranges, and sometimes between the different threads. As such, the execution time was captured in the program source code in microseconds (μs). The result was that sometimes a difference of a few seconds was noted between the execution time recorded by the IDE as well as that recorded by the source code. This was expected and was considered an acceptable margin of error.

In addition to the 2^{30} range, the sequential and parallel programs were also executed for the ranges of 100 to 2,000,000,000 in order to observe the program's performance at a small, as well as a large problem size. The maximum array size of 2,147,483,647 (2^{31}-1) (Jana, 2005) was not considered due to limited available memory on the target computer. The programs were executed 10 times for threads varying from 1 and 9. The 9-thread program was executed with the intention of determining whether or not threads exceeding the available logical processors would enhance program performance. The NetBeans IDE was used to automatically provide a count of the number of prime numbers found, as well as the mark() method and overall program execution times. Where the performance of the 2^{30} range was compared to the performance of other ranges, only the average of the first 10 trials was considered.

3.3 Data Analysis

3.3.1 Performance Evaluation

The program's performance was analysed in terms of execution time, speedup, and efficiency. The execution times of the mark() method and the overall program runtime were captured, and presented in a table and then compared, in order to determine the behaviour of the mark() method, as well as the effect of the applied parallelism on the overall program performance. Speedup and efficiency were calculated based on the formulae provided in Equations 2 and 3 below, as provided by (Molia, 2014a).

$$Speedup = \frac{Sequential\ Execution\ Time\ (Ts)}{Parallel\ Execution\ Time\ (Tp)}$$

Equation 2 Speedup

$$Efficiency = \frac{Speedup}{Number\ \ of\ Processors\ \ (P)}$$

Equation 3 Efficiency

3.3.2 Relationship between Processors and Overhead

The allocation of threads to processing cores was automatically done by the underlying operating system and is therefore not known to the researcher. As such, the investigation of the relationship between the processors and the overhead incurred was done in terms of available logical processors (8), and not in terms of available physical processing cores (4).

A line graph was created in Excel, in order to visualise the data distribution of the variables under investigation (overhead incurred, and the number of threads employed). Using the Statistical Package for Social Sciences (SPSS), correlation analyses were applied to the data, in order to determine whether there existed a statistically significant relationship between the variables, and then to investigate the strength and direction of such a relationship. Both the Spearman, and Pearson Correlation techniques were used for comparison purposes, although (Hoskin, n.d.) emphasised that non-parametric techniques, such as the Spearman Correlation technique, are better suited for correlation analyses of data with a non-Gaussian distribution, as was the case in this study.

The level of significance (α-value) was defined as 0.05 and it was determined that the hypothesis (H_0) that there does not exist a relationship between the number of threads and the overhead incurred would be rejected if the probability value (p-value) is less

than or equal to the defined α-value (Daniel, 2021). This outcome would also then suggest that for this study, there exists a relationship between the number of processors and the overhead incurred.

3.3.3 Investigation of Overhead

The third objective of this study was to investigate the mathematical nature of the overhead incurred. In this regard, it was determined that the mathematical nature of the overhead function in relation to the number of processors employed would be derived from the graph of function represented by the data. Different graphs of functions exhibit different properties and abide by formulae specific to the type of graph that they represent, such as $f(x) = a\sqrt{x-h}+k$ for square root functions, and $f(x) = \log_b x$ for logarithmic functions. The graph of function represented by the overhead was therefore deemed sufficient to derive the mathematical nature of the overhead.

In this regard, the overhead function was computed from the execution times of the sequential and parallel programs, and then plotted on a line graph to provide a visual representation of the data. The graph was then used to determine the function represented by the data.

4 Results & Discussion

This chapter presents the findings of the study. The data was collected through observations on data generated by the programs and presented using charts, graphs, and tables. The data was analysed using statistical correlation tests. The results of the study are also discussed, as part of this chapter.

4.1 Program Development

Although the goal of the study was to compare the performance of one sequential algorithm to its parallel equivalent, development of the algorithm was such that various major revisions needed to be made, in order to arrive at the final algorithm. Development was also such that while the sequential algorithm was supposed to inform the parallel algorithm, ofttimes the sequential algorithm did not translate well into the parallel version. As such, the sequential algorithm was adjusted to the version that provided the required functionality most efficiently when parallelised. This section discusses the journey through the various revisions leading up to the final programs.

4.1.1 Collection

The program was implemented using both an ArrayList and an Array, however, no significant performance difference was noted between the collections. Since both collections provided the required functionalities, the researcher opted for the Array solely due to preference.

4.1.2 The Sequential Algorithm

Initially, two sequential SoE programs were developed based on the pseudocode provided in Figure 3. One program used a boolean Array, while the other used a character Array. It was found that generally, the boolean Array performed better than the character Array, therefore further optimisations were made to the algorithm using the boolean Array.

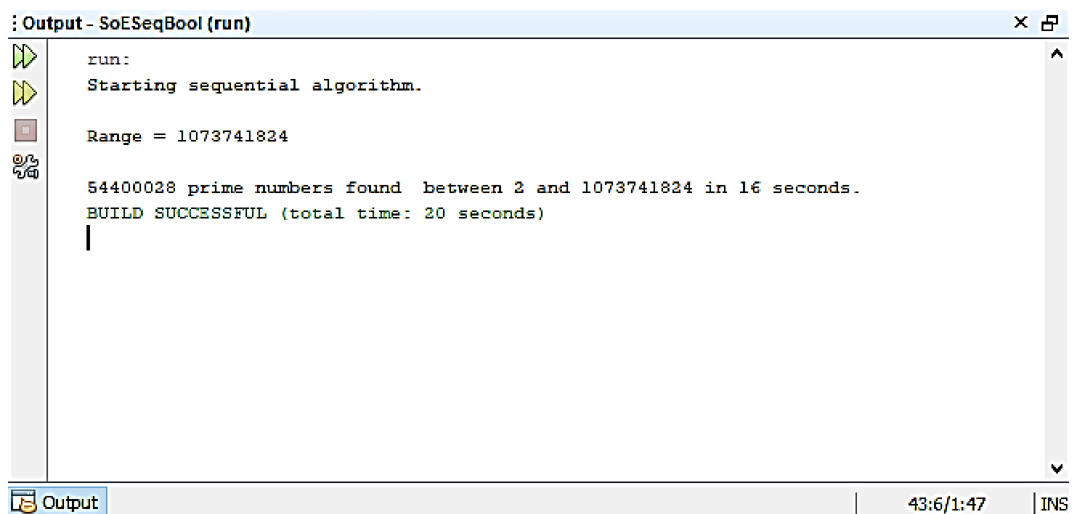


Figure 4 Performance of the SoE program using the Boolean array

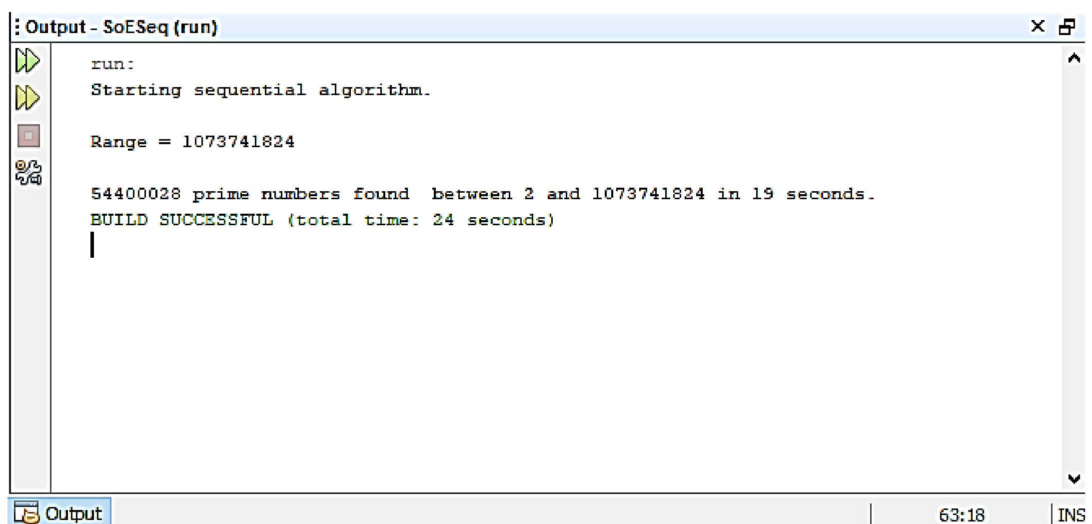


Figure 5 Performance of the SoE program using the Character array

In the sequential program, only one thread performed the seeding operations, therefore no synchronization was required. In the parallel implementation however, it was found that in order to minimise repetition of work, states needed to be assigned in order to differentiate between the integers that had been marked, as well as those that still needed to be marked. Three states were then identified, namely the unmarked state, as well as the marked state which consisted of the composite and prime states. The Array was initialised with a zero '0', which indicated that the integer at the specific index was not marked. Once the integer was identified as a prime number, the character 'p' was assigned as the new value. The multiples of this prime number would then be assigned the character 'c' to indicate that they are composite. All indices allocated a 'p' or a 'c' would then be ignored by other threads when the seeding operations were conducted.

A boolean data type can only represent two states, however. As such, the program using the boolean Array required additional lines of code to implement a functionality that was generic to a character Array. The result was that the boolean Array performed marginally worse than the character array in its parallel form. This was attributed to the excess computation introduced. The boolean Array was thereafter abandoned in favour of the character Array.

4.1.3 Odds Only Optimisation

Synchronization between threads in the SoE program is not a requirement. For the most part, threads are able to perform seeding operations without causing inconsistencies in the data. As such, the program used in this study consisted of only one array called *list*, from which all threads could read and write simultaneously. One

of the optimisations proposed by Quinn (2004) was to conduct the primality checks on odd numbers only. In the sequential algorithm, an array half the size of n could be created, with the operations performed on the odd numbers. This evidently provided a performance boost, however, the optimisation caused unexpected behaviour in the parallel algorithm whereby the program could not consistently find the correct number of prime numbers within a given range. As a result, an Array of size n was created. This meant that instead of excluing the even numbers from the Array, the even numbers were ignored by ensuring that only every second integer was eligible for the primality test. As such, while memory was not used efficiently, the seeding operations were still conducted efficiently.

4.1.4 Parallel Framework

There are many methods of implementing parallelism in Java programs such as through the manual creation of threads, through the use of executors like the Executor Interface, Thread Pools, and the Fork/Join framework, and finally through the use of Concurrent Collections. For this study, the manual creation of threads was preferred due to its simplicity in implementation, and also due to its flexibility with respect to defining the required number of threads. The manually created threads performed significantly better than the Thread Pool. According to the Oracle Java Documentation (2021), Thread Pools are best suited to large-scale applications, where they are known to minimise overhead resulting from the creation and management of threads. It would therefore appear that in this case, the problem size was not large enough to efficiently utilize the threads in the Thread Pool. As such, the approach of manually creating the threads was adopted.

4.1.5 Data Decomposition

Both IDD and BDD were implemented, and it was observed that the IDD program provided both better performance and better parallelism in comparison to the BDD program. More often than not, threads in the IDD commenced work out of the order in which they were created. For example, thread 5 would commence first, then thread 1, then thread 7 etc. Thread completion was also out of the order in which the threads commenced their computations. For example, thread 7 would complete first, then thread 5 and then thread 1. This behaviour suggested a high level of independence between the threads, which also suggests a high level of parallelism in the mark() method. This phenomenon is shown in Figures 6 and 7 below.

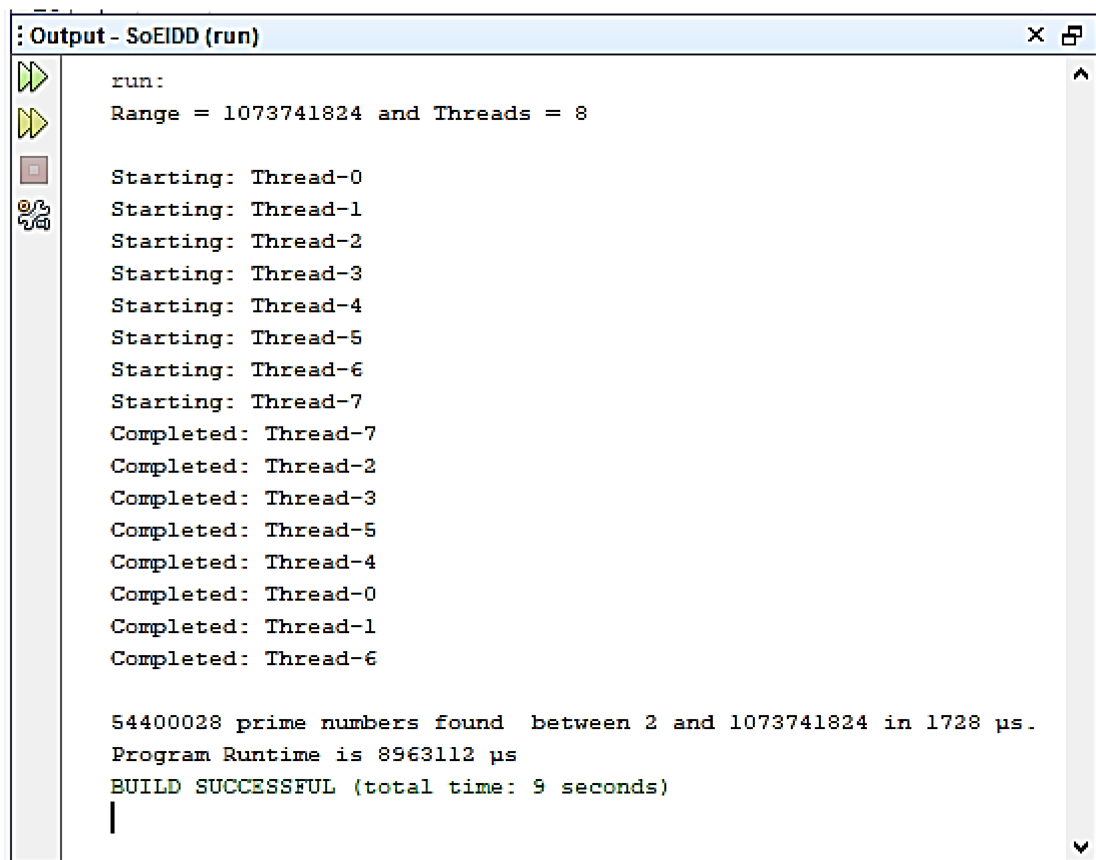


Figure 6 IDD thread creation and termination

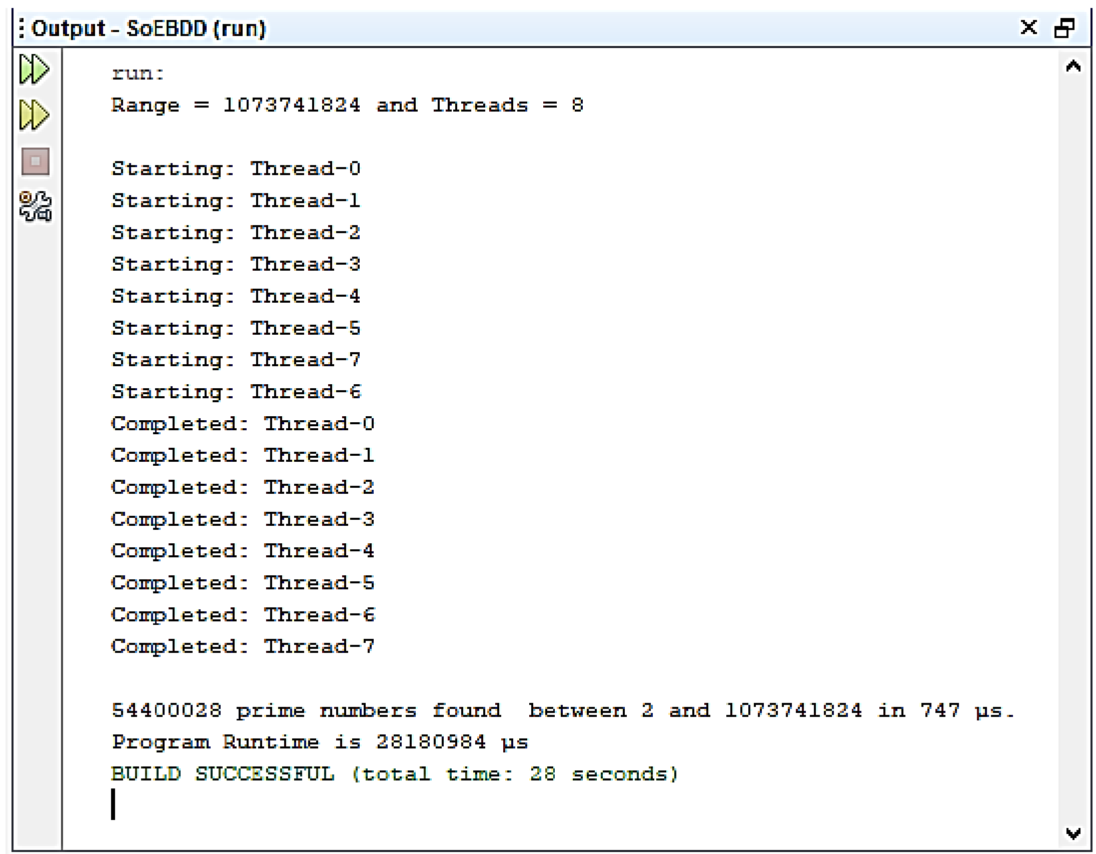


Figure 7 BDD thread creation and termination

As evidenced in Figure 7 above, the BDD program was unable to exploit the inherent benefits of parallelism. As such, the IDD algorithm was selected for implementation due to its performance, while the BDD algorithm was considered for future work. Figure 8 below plots the execution time of the BDD and IDD programs.

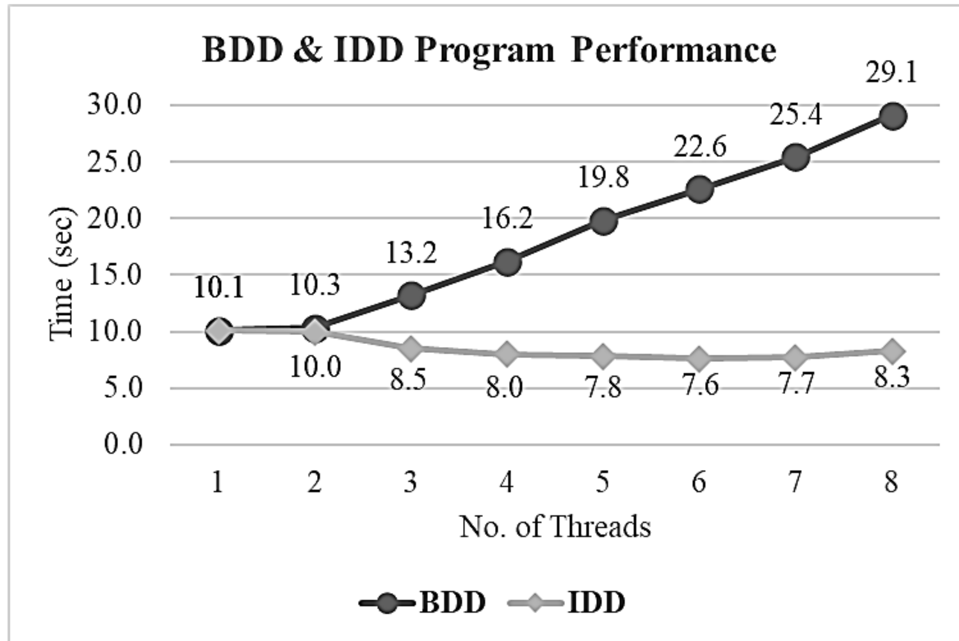


Figure 8 BDD and IDD program performance

4.2 Program Performance

Performance was the key requirement in this study. In context, performance refers to a shorter execution time. Improved performance comes with the execution of as few low-cost operations as possible, as well as minimal memory utilisation (GeeksforGeeks, 2021). The programs in this study were designed with the intention of reducing computation time, and not necessarily with the intention of utilising less memory. The programs were evaluated, and the results are as follows.

4.2.1 Execution Time

The program was evaluated at the 2^{30} range, and the program execution time in microseconds (μs) is as illustrated in Table 1 below.

	Seq	Parallel						
Trial	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	11010755	10487353	8913872	7635771	7976730	7363858	7886463	7656945
2	9906090	10373666	8360571	7981558	8171216	7874636	7484427	8696261
3	9919529	10213938	8749177	8529011	7782602	7358929	8061661	8026209
4	9922432	10096130	8364567	7758479	7537896	7519442	7624119	8349499
5	9879818	10389523	9060314	8270198	7742475	7665151	7401356	8432180
6	9889756	9980601	8632743	7944491	7878711	7408939	7433024	8319015
7	9889717	10307353	8167947	7964103	7916350	7772120	7784864	8165744
8	9836323	10082744	8299139	7947134	7519202	7097869	7880149	8396324
9	9892922	10285731	8303612	8064606	7590346	7805610	7859876	8542163
10	9855293	10160795	8711634	7994948	8275706	7742060	7815086	8277455
11	9917172	10375237	8635453	7943006	7951005	7421692	7790114	8402736
12	10092403	10101350	8727349	7710788	7727070	7400383	7772436	7885072
13	9952724	10044874	8780279	7837051	8084579	8022877	7504093	8912822
14	9949157	10082149	8780463	7733661	8084641	7300740	7777792	7918616
15	10078411	9905717	9005717	7748055	7515255	7341550	7506897	8225140
16	9983557	10016737	8804161	7917866	7423938	8166513	7924509	8311158
17	9960446	9891057	8369826	7874249	8479571	7877008	7725704	8437202
18	9991661	9761082	8876234	7731541	8020716	7568396	8030686	8429340
19	10645108	9780504	8178529	8164724	7838462	7240919	7389946	7995044
20	9951372	9756388	8387337	8008039	7877199	7450163	7617324	8364888
21	10011675	9870861	8040720	8162782	7481618	7837608	7850621	8200116
22	9974146	9676884	9350177	7918568	7366795	7965489	7818220	8229529
23	9993840	9740895	8224359	9006975	7657467	8132447	7808967	8294731
24	9965734	9698645	8197145	8409809	8035953	7313292	8051031	8202382
25	9984250	9685491	8275937	7892036	7552621	7687469	7737914	8476808
26	10126898	9694405	8610521	7750528	7364209	7712169	7607107	8151752
27	9999465	9720331	8422099	7828082	7603061	7539862	7596015	8311244
28	9995560	9697863	8172294	7917943	8038628	7625650	7405407	8160259
29	10473639	9710580	8289829	8185407	7881022	7370397	7747092	7850349
30	10585264	9693820	8246155	7804164	7787243	7592231	7702765	8399026
Average	**10,054,504**	**9,976,090**	**8,531,272**	**7,987,852**	**7,805,410**	**7,605,849**	**7,719,856**	**8,267,334**

Table 1 Program execution time at 2^{30}

The results indicate that the parallel program performed better than the sequential program for all threads, with the program performing best at 6 threads in terms of average program execution time. After 6 threads, the execution time increased, although the parallel program still performed better than the sequential program. The results suggest that program execution time tended to decrease as the number of threads were increased.

The mark() method execution time was also evaluated, and the execution time in microseconds (μs) is as illustrated in Table 2 below.

	Seq	Parallel						
Trial	1	2	3	4	5	6	7	8
1	9113207	237	247	289	374	512	1424	657
2	8167159	128	660	316	411	509	586	623
3	8175785	124	207	276	447	484	567	651
4	8165818	456	217	334	370	465	615	620
5	8137969	139	241	303	434	412	530	674
6	8147231	144	226	314	418	484	1428	758
7	8158652	271	243	342	397	467	514	639
8	8101508	215	246	294	375	500	563	772
9	8150020	122	276	410	441	517	497	1767
10	8122286	153	215	332	350	482	1789	653
11	8186064	154	577	313	403	415	553	697
12	8340548	138	750	312	382	1801	519	715
13	8214206	136	212	280	380	484	534	692
14	8205751	126	221	280	372	417	1234	599
15	8317502	142	278	271	388	438	541	772
16	8244419	122	221	315	400	439	431	682
17	8220017	167	255	282	450	399	609	866
18	8246717	124	229	292	321	481	560	628
19	8877233	141	225	333	376	463	510	663
20	8210476	125	220	348	400	545	477	659
21	8273265	200	222	287	444	507	625	691
22	8234949	121	229	307	388	425	476	660
23	8251024	131	235	307	425	559	694	706
24	8227900	475	217	266	395	474	479	1913
25	8248197	123	213	329	385	526	522	659
26	8380645	144	217	336	360	456	467	719
27	8257774	191	246	333	435	473	495	667
28	8248507	123	235	392	396	534	484	699
29	8677929	142	210	264	394	466	481	2584
30	8694357	146	217	304	381	430	605	653
Average	8,299,904	172	274	312	396	519	660	825

Table 2 mark() method execution time at 2^{30}

The results show that the parallel mark() method was on average 10,000 times faster than the sequential mark() method at its worst average execution time (8 threads). It is also evident that unlike the average program execution time which was lowest at 6 threads, the mark() method execution time was lowest at 2 threads and continued to increase marginally thereafter, although still performing better than the sequential mark() method. This is unlike the average program execution time which decreased and was lowest at 6 threads but increased marginally thereafter.

4.2.2 Speedup and Efficiency

Speedup is said to be the better metric for evaluating program performance since it considers the number of processors, as well as the problem size (El-Nashar, 2011). In this study, it was observed that the program experienced speedup, which increased as the number of threads were increased, and eventually saturated at 6 threads (1.32) for the 2^{30} range. This in accordance with the findings of Bokhari (1986) whose study indicated that no speedup could be achieved beyond 6 processors since the first few seeds performed the most work. The findings of this study were however contrary to those of Costa et al. (2014) who found that their recommended parallel program performed best while using 8 threads, with the results being proportional to the number of available cores and not the available threads. In this study, the speedup was neither proportional to the number of available threads which were 8 and thus should have yielded a speedup closer to 8; nor the number of available cores, which were 4 and thus should have yielded a speedup closer to 4.

The most efficiency was experienced at 2 threads but decreased as the threads increased, with the least efficiency experienced at 8 threads. The best speedup suggests that the program only performed 32% faster than the sequential program. Table 3 below shows the speedup and efficiency experienced against the number of threads employed. The speedup tended to increase slightly, while the execution time and efficiency both decreased.

Threads	Execution Time (μs)	Speedup	Efficiency
1	10,054,504	1.00	1.00
2	9,976,090	1.01	0.50
3	8,531,272	1.18	0.39
4	7,987,852	1.26	0.31
5	7,805,410	1.29	0.26
6	7,605,849	1.32	0.22
7	7,719,856	1.30	0.19
8	8,267,334	1.22	0.15

Table 3 Execution time, speedup, and efficiency at 2^{30}

4.2.3 Program Performance at Low and High Workloads

The program was also evaluated at the ranges of 100 to 2,000,000,000. The results in microseconds (μs) are as shown in Tables 4 and 5 below.

N	Primes Found	Seq	Parallel							
		1	2	3	4	5	6	7	8	9
100	25	585	2,276	2,792	2,987	3,071	3,152	3,802	3,554	3,938
1,000	168	562	2,448	2,834	2,770	2,898	1,659	1,769	1,806	1,966
10,000	1,229	1,781	1,830	1,901	1,887	1,966	2,016	2,062	2,291	2,388
100,000	9,592	9,834	5,770	5,622	5,191	5,702	5,217	5,892	5,418	5,897
1,000,000	78,498	26,359	33,269	38,692	34,169	18,776	20,379	21,471	20,760	19,863
10,000,000	664,579	108,211	73,328	70,970	67,471	66,652	66,311	65,943	65,582	66,133
100,000,000	5,761,455	1,077,460	872,247	713,883	642,358	621,965	600,215	608,542	613,394	633,092
1,000,000,000	50,847,534	10,123,746	9,872,137	9,196,038	9,008,936	8,983,890	8,801,670	9,280,267	9,288,841	10,066,495
1,073,741,824	54,400,028	10,000,264	10,237,783	8,556,358	8,009,030	7,839,123	7,560,861	7,723,103	8,286,180	9,790,779
2,000,000,000	98,222,287	20,019,178	20,253,709	17,685,664	16,325,599	16,333,882	15,679,458	16,026,037	16,348,040	17,934,377

Table 4 Average program execution time from 100 to 2,000,000,000

N	Primes Found	Seq	Parallel							
		1	2	3	4	5	6	7	8	9
100	25	36	251	520	684	867	1,009	1,362	1,376	1,616
1,000	168	49	298	515	642	809	542	622	692	800
10,000	1,229	423	176	304	362	454	519	608	704	811
100,000	9,592	3,422	129	227	306	409	448	525	641	795
1,000,000	78,498	8,759	285	441	531	380	455	567	627	773
10,000,000	664,579	61,204	128	214	285	381	459	541	647	714
100,000,000	5,761,455	751,385	160	221	292	414	472	538	645	763
1,000,000,000	50,847,534	8,470,978	141	219	295	380	461	519	630	686
1,073,741,824	54,400,028	8,243,964	199	278	321	402	483	851	781	1,383
2,000,000,000	98,222,287	16,802,409	136	235	309	425	469	549	640	770

Table 5 Average mark() method execution time from 100 to 2,000,000,000

The results indicate that for the ranges of 100 to 10,000, the sequential program performed better than the parallel program in terms of average program execution time. This is evident in the execution time of the parallel mark() method at the 100 range, which performed up to 38 times slower than the sequential mark() method when using 8 threads. As the range increased however, the execution time decreased. At the 10,000 range, the parallel mark() method only performed 2 times slower than the sequential program when using 8 threads. It is however evident that from the 10,000 range, the parallel mark() method performed best using 2 threads, for all ranges. From the range of 100,000, all the parallel programs performed better than the sequential program in terms of average program execution time, and ultimately performed best at 10,000,000 with 8 threads. For the ranges of 100,000,000 to 2,000,000,000 the parallel program performed best while using 6 threads. The parallel program was also executed with 9 threads, and it was noted that although the program provided a computational advantage over the sequential algorithm, it did not perform best at any range. It is also evident that the mark() method of the 9-thread program experienced the longest execution time, which suggests that the method's execution would not be decreased at any range, using any number of threads.

The ranges of 100 to 10,000 experienced the lowest speedup and efficiency. As the range was increased however, it was noted that speedup and efficiency also increased. The ranges of 100,000 to 100,000,000 experienced the best speedup and efficiency, with the exception of the 1,000,000 range. Beyond the 100,000,000 range, both speedup and efficiency decreased. Tables 6 and 7 below show the speedup and efficiency gained at the various ranges.

N	Speedup							
	1	2	3	4	5	6	7	8
100	1.00	0.26	0.21	0.20	0.19	0.19	0.15	0.16
1,000	1.00	0.23	0.20	0.20	0.19	0.34	0.32	0.31
10,000	1.00	0.97	0.94	0.94	0.91	0.88	0.86	0.78
100,000	1.00	1.70	1.75	1.89	1.72	1.89	1.67	1.82
1,000,000	1.00	0.79	0.68	0.77	1.40	1.29	1.23	1.27
10,000,000	1.00	1.48	1.52	1.60	1.62	1.63	1.64	1.65
100,000,000	1.00	1.24	1.51	1.68	1.73	1.80	1.77	1.76
1,000,000,000	1.00	1.03	1.10	1.12	1.13	1.15	1.09	1.09
1,073,741,824	1.00	0.98	1.17	1.25	1.28	1.32	1.29	1.21
2,000,000,000	1.00	0.99	1.13	1.23	1.23	1.28	1.25	1.22

Table 6 Speedup for all ranges.

N	Efficiency							
	1	2	3	4	5	6	7	8
100	1.00	0.13	0.07	0.05	0.04	0.03	0.02	0.02
1,000	1.00	0.11	0.07	0.05	0.04	0.06	0.05	0.04
10,000	1.00	0.49	0.31	0.24	0.18	0.15	0.12	0.10
100,000	1.00	0.85	0.58	0.47	0.34	0.31	0.24	0.23
1,000,000	1.00	0.40	0.23	0.19	0.28	0.22	0.18	0.16
10,000,000	1.00	0.74	0.51	0.40	0.32	0.27	0.23	0.21
100,000,000	1.00	0.62	0.50	0.42	0.35	0.30	0.25	0.22
1,000,000,000	1.00	0.51	0.37	0.28	0.23	0.19	0.16	0.14
1,073,741,824	1.00	0.49	0.39	0.31	0.26	0.22	0.18	0.15
2,000,000,000	1.00	0.49	0.38	0.31	0.25	0.21	0.18	0.15

Table 7 Efficiency for all ranges.

4.3 Overhead Function

One of the observations made with respect to the execution time of the mark() method and the overall program execution time is that the former tended to increase, while the latter tended to decrease as the number of processors were increased. This is due to the presence of overhead. The overhead was computed according to Equation 4 below (Grama et al., 2003a) and plotted on a line graph which provided a visual illustration of the function that the data represented. Figure 9 below plots the overhead function of the program at the 2^{30} range.

$$\text{Overhead} = \text{Parallel Execution Time} - \frac{Sequential\ \ Execution\ \ Time}{Number\ \ of\ \ Processors}$$

Equation 4 Overhead

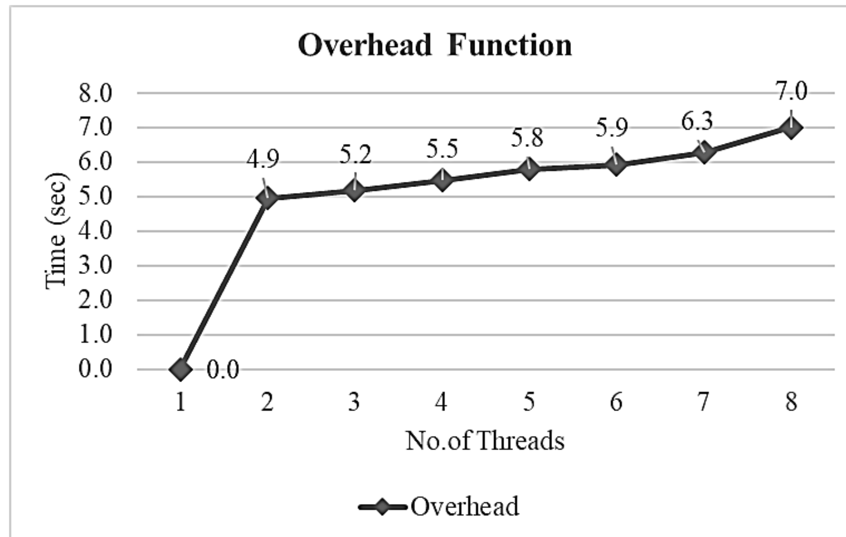


Figure 9 Overhead at 2^{30}

The graph shows that the 2-thread program experienced the lowest overhead, while the 8-thread program experienced the largest amount of overhead. It can be deduced from the graph that overhead tended to increase as the number of processors were increased. This suggests that there exists a positive relationship between the overhead and the processors employed. Two correlation tests were performed, and both tests yielded a p-value of 0.01, which confirms a positive statistically significant relationship between the variables under consideration. This suggests that for this study, an increase in the number of processors resulted in an increase in the overhead function.

The data also forms an angular graph with a slope extending towards the right. This graph does not resemble any particular graph of function. Since sequential algorithms

do not incur overhead however, it can be said that the graph starts at point (2, 4.9) and ends at point (8, 7). This graph would then be a line graph without an x nor y intercept, and therefore not conforming to the properties of a linear graph of the form $y = mx + c$. As such, the function of the overhead could not be derived from this graph.

Nonethless, it was noted that overhead is also related to efficiency, which is defined in this study as the measure of how efficiently a program utilizes system resources, including processors, to solve a problem. When the overhead was represented as a ratio of the execution time, it was not only evident that the overhead and efficiency were inversly proportional, but it was also evident that the two graphs were inverses of one another about point (0, 0.5) and (8, 0.5), as shown in Figure 10 below.

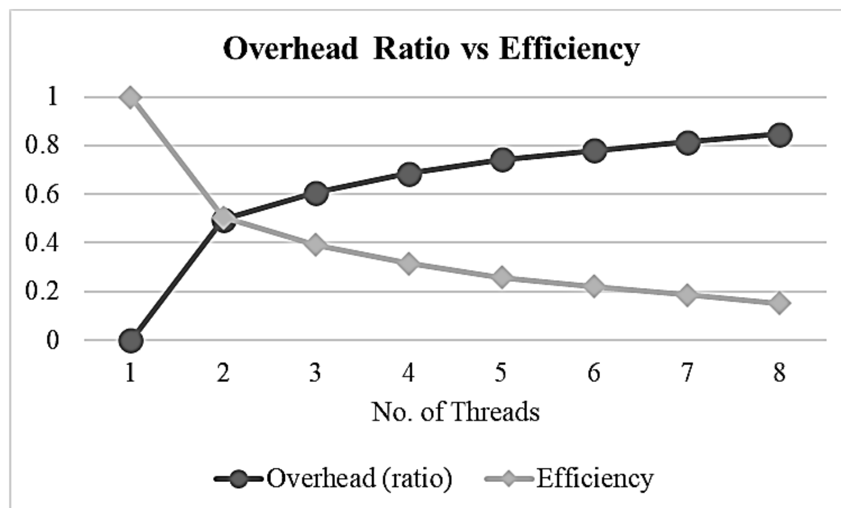


Figure 10 Overhead ratio versus efficiency

4.4 Cost of Performance

Figure 11 below plots the overhead, efficiency, and speedup gained by the program at the 2^{30} range in percentages and provides a visual representation of the cost of the performance gained. The lowest overhead incurred by the program is 49.61% at 2

threads, and the highest overhead recorded was 84.80% when using 8 threads. The results also show that at the program's worst execution time, which is while executing 2 threads, the program experienced a speedup of 1%, incurred nearly 50% overhead, and experienced 50% efficiency. This suggests that the program experience virtually no speedup at a cost of 50% loss of efficiency and 50% overhead. At the program's seemingly best performance in terms of execution time, which is while using 6 threads, the program experienced a speedup of 32%, incurred 77.97% overhead, and experienced only 22% efficiency. Evidently, the performance cost in terms of overhead far outweighed the performance gained when using any number of threads.

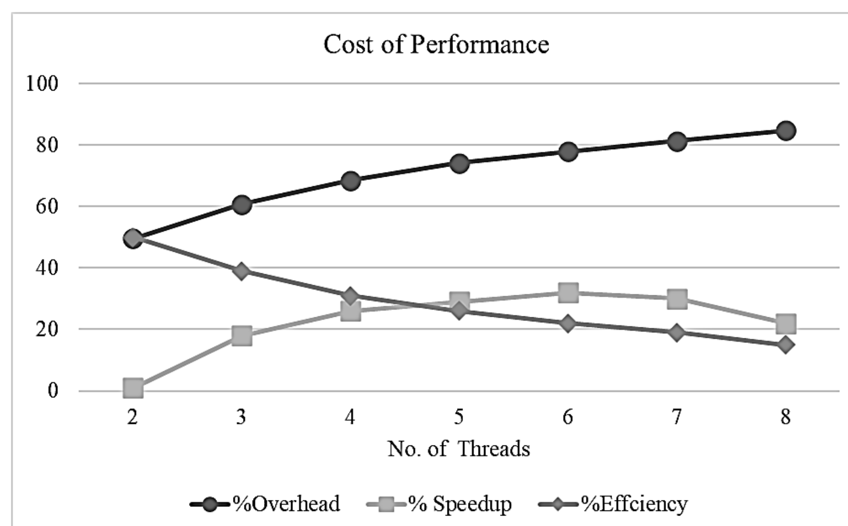


Figure 11 Cost of performance

4.5 Discussion

The first objective of the study was to observe the program's performance at the 2^{30} range, and to determine whether parallelisation provided a computational advantage. It is evident in the results that the parallel program provided a computational advantage not only for the 2^{30} range, but for all ranges greater than 10,000. For the ranges of 100 to 10,000, all the parallel programs performed worse than the sequential program. This

is consistent with the observation of El-Nashar and Aljahdali (2013) that increasing the number of processors can result in increased communication time, which can negate the decrease in computation time. Evidently this was the case for the smaller ranges. The execution time of the mark() method tended to increase as the threads were increased. This is reflective of the overhead that the method is generating and suggests that overhead tended to increase as the threads were increased.

It was also observed that while the 9-thread program performed better than the sequential algorithm, it did not perform better than the other parallel programs at any range. At 100, the 9-thread program performed worst. At 100,000, it performed as well as the 7-thread program, and at 100,000,000, it performed as well as the 4-thread and 5-thread programs. By 2,000,000,000, the 9-thread program performed as well as the 3-thread program. This suggests that increasing the problem size caused the 9-thread program's performance to degrade, rather than improve. This is because when threads exceed the number of available processors, cache thrashing occurs (El-Nashar & Aljahdali, 2013) which ultimately degrades program performance. Thus, it can be said that for this study, it was also observed that there was no performance to be gained from threads that exceeded the number of available processors.

It is said that ideally, speedup should increase linearly, while efficiency decreases linearly in relation to the number of processors. In this study, speedup increased, and efficiency decreased as the processors were increased, but not linearly. Both speedup gained and efficiency remained below 50%, which was neither in proportion to the number of available physical cores (4) as was the case for (Costa et al., 2014), nor the number of logical processors (8). Since speedup is also dependent upon the degree of

parallelism in the program, it was deduced that the program contained too many serial components to provide a significant performance boost, and thus achieved the sublinear performance noted. This was expected since only one method implemented parallelism.

Increasing the problem size improved speedup and efficiency up to the range of 100,000,000, after which it saturated, and ultimately caused efficiency to decrease. The best speedup was experienced at the 100,000 range when using a minimum of 3 threads, while the best efficiency was experienced at 100,000, when using a maximum of 3 threads. It was also noted that while the parallel program certainly experienced a shorter execution time in comparison to the sequential algorithm, the cost of parallelism was considered to be high since very little speedup and efficiency were achieved, in comparison to the high cost of overhead. This certainly does not suggest that it was not worth it to parallelise the program, but rather that parallelism could be implemented better, by reducing the serial components. Future work would therefore suggest attempting to optimise the source code, to minimise the amount of serial components.

The second objective of the study was to determine how a change in the number of processors affects the overhead function. Two two-tailed correlation analyses were conducted, and both confirmed that there exists a relationship between the number of processors and the overhead incurred, and that the relationship was significant at 0.01. The significance value also informs us that the relationship is positive, which means that an increase in the number of processors results in an increase in the overhead function.

The last objective of the study was to quantify the overhead incurred as a function of the processors employed. The mathematical nature of the overhead could not be derived from the graph in Figure 9, however; it was evident in Figure 10 that efficiency and the overhead ratio were inversely proportional to one another. As such, when the efficiency increased, the overhead ratio decreased. The graph suggests that in order to achieve 100% efficiency, one must incur 0% overhead. It can therefore be said that the overhead function represented a amount of the efficiency lost. As such, it was determined that overhead could be represented according to the Equation 5 below.

$$Overhead = execution\ time * (x - efficiency)$$

Equation 5 Overhead as a function of the execution time and efficiency

In Equation 5, x represents the maximum amount of efficiency attainable. In this study, x was equal to 1, which is equivalent to 100%. This objective was therefore satisfied since although the overhead could not be quantified as a function of the number of processors employed, it was nonetheless determined that the overhead was mathematically quantifiable as product of the parallel program's execution time and the fraction of efficiency lost. It was further determined that for any given thread, the overhead can be computed from the thread's execution time and efficiency.

5 Conclusion

This chapter concludes the study and provides a summary of the key accomplishments of the study, the lessons learned, and gaps in knowledge identified. The chapter also provides a summary of the conclusions drawn from the results, as well as how these relate to the research objectives.

The objectives of the study were to determine whether there existed a relationship between the overhead incurred in the program under investigation, and the number of processors employed; and to further determine whether the overhead function was mathematically quantifiable as a function of the processors employed. It was determined that there existed a statistically significant relationship between the variables under investigation; however, the study was unable to provide a closed mathematical formulation of the overhead function in relation to the processors employed. Instead, the empirical results confirmed that overhead also bore a relation to efficiency, and that overhead could be quantified as a product of the program's execution time and the fraction of efficiency lost, thus the objectives of the research were satisfied.

The study had also intended to observe the performance of the parallel programs so as to determine whether they provided a computational advantage over the sequential program, and to further determine whether the parallel programs conformed to known behaviours of parallel programs. Based on the quantitative analysis of the program execution time, it can be concluded that the parallel programs provided a computational advantage over the sequential program, but that the performance was not proportional to the number of processors employed and was further achieved at a

high cost of overhead. The study also confirms that as observed by other researchers, program performance was affected by the problem size and processors employed. It is further speculated that the minimal parallelism in the program also had a negative effect on the program's performance, although this was not confirmed.

The generalizability of the results is limited by the fact that the algorithm was executed on only one computer. This approach was favoured because it allowed for the program to be evaluated on an architecture which is known to best support the type of parallelism implemented. The approach clearly illustrates the effect of multithreading on a multicore architecture; however, it also raises the question of how the program would perform on other architectures.

In the study, it was learnt that parallelisation is a powerful tool that software developers need to embrace, in order to provide the best program performance, and to utilise computer architectures optimally. As it was discovered in the various revisions of the SoE however, parallelisation is just as likely to improve program performance, as it is to degrade it. This is due to a multitude of factors involved. Some of these factors such as overhead, serial components, programming paradigm, and the programming style are not beyond the software developer's influence in terms of tuning and optimisation so as to achieve the best performance. While it is certainly not guaranteed nor expected that all parallel programs will achieve linear or super-linear speedup when these factors are aligned, it can certainly be aspired that such an alignment will at least yield near linear performance. It is also evident that much research still needs to be conducted before we arrive at a point where such performance is achievable as a norm for all parallel programs.

The SoE is one of the oldest algorithms in existence and is not known to be in use today in its unadulterated form. Yet many of the algorithms in use today derive from the SoE. Evidently, these ancient algorithms remain relevant in the area of academic research, as does this study, which is intended to appeal to scholars and to be applied in the area of academic research.

Finally, this study proved that although optimal speedup and efficiency could not be achieved, the parallel SoE program evidently reduced the program execution time for ranges between 100,000 and 2,000,000. This suggests that nowadays the public has access to rather powerful off-the-shelf computers. Since technology is only expected to continue to advance, Namibian companies and organisations are therefore encouraged to strengthen their cybersecurity programmes and initiatives, since it is evident that computers that are available to the general population are now sophisticated enough to be able to launch successful cyberattacks within a reasonable amount of time.

6 Recommendations

This chapter presents suggestions for future research, based on the findings of the study, the literature reviewed, the results, and gaps in knowledge identified in the study.

Recommendation 1

Architecture plays an important role in the performance of parallel programs. In this study, the programs were only evaluated on one quad-core computer. To better understand the implications of these results, future studies might address the effect of other architectures such as an additional multi-core architecture, a multiprocessor architecture, as well as a uniprocessor architecture\, on the program's performance.

Recommendation 2

In this study, parallelism was only implemented in one method. Further research could improve on the study by determining the effect of additional parallelism on the program's performance. The additional parallelism could be applied when populating the Array, as well as when printing the prime numbers found.

Recommendation 3

In this study, only one method of data decomposition was implemented. For future work, both methods of data decomposition can be implemented for the same algorithm, in order to compare the performance of the programs. This can further be extended to other algorithms, to determine whether the performance of one method of data decomposition provided a computational advantage for all algorithms under investigation, or only some.

References

Abdallah, A. Ben. (2013). *Multicore Systems On-Chip : Practical Software / Hardware Design* (2nd ed.). Atlantis Press.

Bokhari, S. (1986). Multiprocessing the sieve of Eratosthenes. *Computer*, *20*(4), 50–58. https://ntrs.nasa.gov/api/citations/19870002076/downloads/19870002076.pdf

Borg, C. W., & Dackebro, E. (2017). A Comparison of Performance Between a CPU and a GPU on Prime Factorization Using Eratosthene's Sieve and Trial Division. In *DEGREE PROJECT TECHNOLOGY*. https://www.diva-portal.org/smash/get/diva2:1107835/FULLTEXT01.pdf

Brumme, S. (2014). *Parallel Sieve of Eratosthenes [updated]*. https://create.stephan-brumme.com/eratosthenes/

Burd, S. (2016). *Systems Architecture* (7th ed.). Cengage Learning.

Chhibber, R., & Garg, R. (2014). Multicore Processor, Parallelism and their Performance Analysis. *International Journal of Advanced Research in Computer Science and Technology*, *2*(3), 31–37.

Cordeiro, M. (2012). *Parallelization of the Sieve of Eratosthenes* [Doctoral Program in Informatics Engineering]. University of Porto.

Costa, C., Sampaio, A., & Barbosa, J. (2014). Distributed Prime Sieve in Heterogeneous Computer Clusters. *Lecture Notes in Computer Science*, *8582 LNCS*(PART 4), 592–606. https://doi.org/10.1007/978-3-319-09147-1_43

Daniel, T. (2021). *Research By Design*.

Dar, T. A., Fayaz, S., & Khan, A. A. (2018). Performance Evaluation of Parallel Algorithm on Multi Core System Using Open MP. *International Journal of Advanced Research in Science and Engineering*, *7*(4), 2371–2380.

Eijkhout, V., Chow, E., & van de Geijn, R. (2014). Introduction to High Performance Scientific Computing. In *HPC* (2nd ed.). https://doi.org/10.5281/zenodo.49897

El-Nashar, A. (2011). To Parallelize or Not to Parallelize, Speed Up Issue. *International Journal of Distributed and Parallel Systems*, *2*(2), 14–28.

El-Nashar, A., & Aljahdali, S. (2013). *Experimental and Theoretical Speedup Prediction of MPI-Based Applications*. https://doi.org/10.2298/CSIS120529047E

El-Rewini, H., & Abd-El-Barr, M. (2005). *Advanced Computer Architecture* (A. Zomaya, Ed.). John Wiley & Sons.

E-maxx-eng. (n.d.). *Sieve of Eratosthenes*. https://cp-algorithms.com/algebra/sieve-of-eratosthenes.html#toc-tgt-5

Fashanu, T. A., Ale, F., Agboola, O. A., & Ibidapo-Obe, O. (2012). Performance Analysis of a Parallel Computing Algorithm Developed for Space Weather Simulation. *International Journal of Research and Technology*, *1*(7), 46–55.

Gebali, F. (2011). *Algorithms and Parallel Computing* (A. Zomaya, Ed.). John Wiley & Sons.

GeeksforGeeks. (2021). *12 Tips to Optimize Java Code Performance*. https://www.geeksforgeeks.org/12-tips-to-optimize-java-code-performance/

Grama, A., Anshul, G., Karypis, G., & Kumar, V. (2003). *Introduction to Parallel Computing* (2nd ed.). Addison Wesley.

Gustafson, J. (1988). Reevaluating Amdahl's Law. *Communications of the ACM*, *31*(5), 532–533.

Helfgott, H. A. (2019). An Improved Sieve of Eratosthenes. *Mathematics of Computation*, *89*(321), 333–350. https://doi.org/10.1090/mcom/3438

Hennessy, J., & Patterson, D. (2012). *Computer Architecture: A Quantitative Approach Fifth Edition* (5th ed.). Morgan Kauffmann Publishers.

Hoskin, T. (n.d.). *Parametric and Nonparametric : Demystifying the Terms*.

Hughes, C., & Hughes, T. (2008). *Professional Multicore Programming: Design and Implementation for C++ Developers*. Wiley Publishing.

Intel. (n.d.). *What Is Hyper-Threading?* Retrieved August 17, 2021, from https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html

Intel. (2008). *Parallel Computing: Background*. Universal Parallel Computing Research. www.intel.com/pressroom/kits/upcrc/%0A

Irabashetti, P. S., & Assistant. (2014). Parallel Processing in Processor Organization. *International Journal of Advanced Research in Computer and Communication Engineering*, *3*(1), 5150–5153.

Jana, D. (2005). *Java and Object-Oriented Programming Paradigm*.

Jenkov, J. (2020). *Concurrency vs. Parallelism*. http://tutorials.jenkov.com/java-concurrency/concurrency-vs-parallelism.html

Johnson, O., & Dinyo, O. (2015). Comparative Analysis of Single-Core and Multi-Core Systems. *International Journal of Computer Science and Information Technology*, *7*(6), 117–130. https://doi.org/10.5121/ijcsit.2015.7610

Karimi, H. A. (2014). Big Data: Techniques and technologies in geoinformatics. In Hassan. A. Karimi (Ed.), *CRC Press*. CRC Press. https://doi.org/10.1201/b16524

Keogh, J. (2004). *Java Demystified*. McGraw-Hill.

Levitin, A. (2012). *Intrroduction to The Design and Analysis of Algorithms* (3rd ed.). Pearson Education Inc.

Månsson, J. (2021). *Comparative Study of CPU and GPGPU Implementations of the Sieves of Eratosthenes, Sundaram and Atkin*. Blekinge Institute of Technology.

Möhring, R., & Oellrich, M. (2011). Algorithms unplugged. *Algorithms Unplugged*, *January*, 119–130. https://doi.org/10.1007/978-3-642-15328-0

Molia, H. K. (2014). Analytical Modeling of Parallel Programs. *International Journal of Engineering Development and Research*, *2*(1), 164–171.

Narang, N., & Kothari, R. (2012). Assigning Threads and Data of Computer Program Within Processor Having Hardware Locality Groups. *United States Patent*, *2*(12).

Neapolitan, R. (2015). *Foundations of Algorithms Using Java Pseudocode* (5th ed.). Jones & Bartlett Learning.

Oaks, S., & Wong, H. (2004). *Java Threads* (3rd ed.). Oreilly.

O'Neill, M. E. (2009). The Genuine Sieve of Eratosthenes. In *Journal of Functional Programming* (Vol. 19, Issue 1). https://doi.org/10.1017/S0956796808007004

Oracle Java Documentation. (2021). *Concurrency*. https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Ore, O. (2017). Invitation To Number Theory. In *Invitation to Number Theory* (2nd ed.). The Mathematical Association of America. https://doi.org/10.5948/upo9780883859605.001

Pacheco, P. (2011). An Introduction to Parallel Programming. In *An Introduction to Parallel Programming* (p. 9). Morgan Kauffmann Publishers. https://doi.org/10.1016/C2009-0-18471-4

Padua, D. (2011). *Encyclopedia of Parallel Computing*. Springer Science+Business Media. https://doi.org/10.1007/978-0-387-09766-4

Pande, N. A. (2013). Algorithms of Three Prime Generating Sieves Improvised by Skipping Even Divisors (Except 2). *American International Journal of Research in Formal, Applied \& Natural Sciences*, *1*(4), 22–27. http://www.iasir.net

Parsons, J. (2018). *New Perspectives on Computer Concepts 2018: Introductory*. Cengage Learning. https://doi.org/10.5005/jp/books/11470_4

Pasquali, S. (2013). *Mastering Node.js*.

Peng, T. A. (1985). One Million Primes Through the Sieve. *Byte Magazine*, *10*(11), 243–244.

Pohl, I., & McDowell, C. (2006). *Java by Dissection* (2nd ed.). University of California,.

Post, E., & Goosen, H. (2001). Evaluating the Parallel Performance of a Heterogeneous System. *5th International Conference on High Performance Computing*, 310–323.

Quinn, M. (2004). *Parallel Programming in C with MPI and OpenMP*. McGraw Hill.

Rauf, I., & Majeed, A. (2017). Parallel-Processing: A Comprehensive Overview of Modern Parallel Processing Architectures. *International Journal of Computer Engineering and Information Technology*, *9*(8), 181–185.

Rosen, K. H. (2012). *Discrete Mathematics and Its Applications* (7th ed.). McGraw-Hill.

Roy, I., Srivastava, A., & Aluru, S. (2016). Programming Techniques for the Automata Processor. *45th International Conference on Parallel Processing (ICPP)*, 205–210. https://doi.org/10.1109/ICPP.2016.30

Sadashiv, N., & Kumar, D. (2011). Cluster, grid and cloud computing: A detailed comparison. *2011 6th International Conference on Computer Science & Education (ICCSE)*, 477–482. https://doi.org/10.1109/ICCSE.2011.6028683

Shawon, A., & Pervez, M. (2016). *Comparison Among Different Prime Generator Algorithms*. https://doi.org/10.13140/RG.2.2.36323.60968

Silberschatz, A., Galvin, P., & Gagne, G. (2013). *Operating System Concepts* (9th ed.). John Wiley & Sons.

Squyres, J. (2004). Processes, Processors, and MPI Cluster World. *MPI Mechanic*, *1*(2), 8–11.

Tabassum, T., Charles, A., & Patil, A. V. (2016). Multicore versus Multiprocessor: A Review. *International Journal of Innovative Research in Computer and Communication Engineering*, *4*(1), 227–232. https://doi.org/10.15680/IJIRCCE.2016

Wang, S., & Ledley, R. (2016). Computer Architecture and Security. In *Fundamentals of Designing Secure Computer Systems: Vol. Chapter 7* (pp. 224–273).

Wilamowski, B. M., & Irwin, J. D. (2016). Fundamentals of Industrial Electronics. In *Fundamentals of Industrial Electronics*. CRC Press. https://doi.org/10.1201/9781315218441

Wirian, D. J. (2009). Parallel Prime Sieve: Finding Prime Numbers. *Institute of Information & Mathematical Sciences Massey University at Albany, Auckland, New Zealand*.

Wu, X. (1999). *Performance Evaluation , Prediction and Visualization of Parallel Systems. The Kluwer International Series on Asian Studies in Computer Series* (K.-Y. Cai, Ed.). Springer Science+Business Media.

Appendix A – SoE Source Code

Sequential Source Code

```
1 /*
2  * Sieve of Eratosthenes Sequential Algorithm
3  * Suama Uushona, 200925121
4  */
5
6 package soeseq;
7
8 import java.util.Arrays;
9 import java.util.Scanner;
10
11 public class SoESeq {
12     private static int n, totalPrimes;
13     private static Scanner scanner = new Scanner(System.in);
14     public static long start, stop, begin, end, runtime;
15     public static char list[];
16
17     public static void main(String[] args) {
18         begin = System.nanoTime();
19         n = 1073741824;
20         System.out.print("Range = " + n + "\n");
21
22         list = new char[n];
23         Arrays.fill(list, '0');
24
25         start = System.nanoTime();
26         mark();
27         stop = System.nanoTime();
28         display();
```

```
29          end = System.nanoTime();
30          System.out.println("Program Runtime is " + (end - begin)
31                  /1000 + " µs");
32      }
33
34      public static void mark() {
35      // Mark all multiples of k as composite (i.e. false)
36          // Set k to 2, the first unmarked number in the list.
37          int root = (int) Math.sqrt(n);
38
39          for (int k = 3; k <= root; k += 2) {
40              // If i is prime, mark all its
41              // multiples as composite
42              if (list[k] == '0') {
43                  for (int i = k * k; i < n; i += k * 2) {
44                      list[i] = '1';
45                  }
46              }
47          }
48      }
49
50      public static void display() {
51          // convert nanosecond to microseconds
52          runtime = (stop - start) / 1000;
53          //System.out.println("\nThe prime numbers between 1 and "
54          // + range + " are: ");
55
56          // writing 2 separately
57          // System.out.print("2 ");
58
```

```
59          // Printing other primes
60          for (int i = 3; i < n; i += 2) {
61              if ((list[i] == 'p') | (list[i] == '0')) {
62                  //System.out.print(i + " ");
63                  totalPrimes++;
64              }
65          }
66
67          System.out.println("\n\n" + (totalPrimes + 1)
68          + " prime numbers found  between 2 and " + n + " in "
69                 + runtime + " µs.");
70
71      }
72 }
```

Parallel Source Code

```
1 /*
2  * Sieve of Eratosthenes Parallel Algorithm
3  * Suama Uushona, 200925121
4  * SoE Algorithm implemented using
5  * Interleaved Data Decomposition
6  * With code from Noushi Tutorial Java
7  * https://www.youtube.com/watch?v=y3nGmZTq_RM&t=13s
8  */
9
10 package soeidd;
11
12 import java.util.Arrays;
13
14 public class SoEIDD {
15
16     public static int n, p, memory, totalPrimes, len;
17     public static long start, stop, begin, end, runtime;
18     public static char list[];
19     public static String name;
20     public static ParallelWorker[] marker;
21
22     public static void main(String[] args) throws Throwable {
23         begin = System.nanoTime();
24         n = 1073741824;
25         p = 8;
26
27         System.out.print("Range = " + n + " and " + "Threads = "
28                + p + "\n\n");
29
```

```
30          list = new char[n];
31          Arrays.fill(list, '0');
32
33          marker = new ParallelWorker[p];
34
35          start = System.nanoTime();
36          mark(p);
37          stop = System.nanoTime();
38
39          for (ParallelWorker worker : marker) {
40              worker.join();
41          }
42
43          display();
44          end = System.nanoTime();
45          System.out.println("Program Runtime is " + (end - begin)
46                  / 1000 + " µs");
47      }
48
49 // Method to compute segments of array and assign to markers.
50      public static void mark(int p) {
51          for (int i = 0; i < p; i++) {
52              marker[i] = new ParallelWorker();
53
54                  //marker[i].setName("Thread " + i);
55              marker[i].start();
56          }
57      }
58
59 // Display primes found
```

```
60     public static void display() {
61                 // convert nanosecond to microseconds
62         runtime = (stop - start) / 1000;
63
64 // System.out.println("\nThe prime numbers between 1 and "
65 // + range + " are: ");
66
67         //System.out.print("2 ");
68
69         // Printing other primes
70         for (int i = 3; i < n; i += 2) {
71             if ((list[i] == 'p') | (list[i] == '0')) {
72                 //System.out.print(i + " ");
73                 totalPrimes++;
74             }
75         }
76
77     System.out.println("\n" + (totalPrimes + 1)
78     + " prime numbers found  between 2 and " + n + " in "
79         + runtime + " µs.");
80     }
81 }
82
83 class ParallelWorker extends Thread {
84
85     public void mark() {
86         int root = (int) Math.sqrt(SoEIDD.n);
87
88         for (int k = 3; k <= root; k += 2) {
89
```

```
90              if (SoEIDD.list[k] == '0') {
91                  SoEIDD.list[k] = 'p';
92
93                  for (int i = k * k; i < SoEIDD.n; i += k * 2) {
94                      SoEIDD.list[i] = 'c';
95                  }
96 //System.out.println(Thread.currentThread().getName()
97 // + " got " + k);
98              }
99          }
100     }
101
102     public void run() {
103 System.out.println("Starting: "
104 + Thread.currentThread().getName());
105     mark();
106 System.out.println("Completed: "
107 + Thread.currentThread().getName());
108     }
109 }
```

Appendix B – Program Execution Times

All execution times are recorded in microseconds. *n* represents the mark() method execution time, while *n** represents the overall program execution time.

100 Range

Trial	Seq		Parallel															
	1	1*	2	2*	3	3*	4	4*	5	5*	6	6*	7	7*	8	8*	9	9*
1	39	407	290	2549	454	2685	713	3338	604	3130	920	2961	1106	3505	1054	3051	1591	4013
2	60	532	185	1535	943	5167	664	2708	1027	3982	1172	3070	958	3074	1213	3054	1208	3537
3	21	431	208	2272	482	2467	657	2752	824	2601	1049	3584	1257	3637	1173	2893	1711	4069
4	23	530	272	2392	454	2707	669	2903	840	2313	685	2717	1088	3516	1378	3633	2501	5537
5	34	560	279	2363	423	2061	676	3082	762	2980	998	3325	1585	4242	1344	3175	1382	3619
6	23	388	288	2427	457	2455	596	2841	931	3094	918	3304	1407	3905	1522	3901	1240	3347
7	42	699	283	2271	416	2046	812	3324	1559	5258	1014	3026	1043	2779	1604	3790	1791	3803
8	23	804	203	2529	364	2124	567	2412	632	2507	1100	3181	1379	3740	1886	4687	1299	3130
9	33	549	321	2682	649	2877	867	3656	689	2383	1075	2883	2602	6673	1204	3717	1967	4587
10	59	949	185	1740	560	3327	614	2854	805	2461	1156	3468	1193	2949	1386	3635	1473	3741
Average	36	585	251	2276	520	2792	684	2987	867	3071	1009	3152	1362	3802	1376	3554	1616	3938

1,000 Range

Trial	Seq		Parallel															
	1	1*	2	2*	3	3*	4	4*	5	5*	6	6*	7	7*	8	8*	9	9*
1	41	505	448	3464	479	2645	661	3027	732	2864	448	1539	615	1866	815	2032	708	1881
2	68	801	270	2401	489	2522	569	2122	944	3197	553	1680	594	1680	640	1703	882	2022
3	72	655	195	1764	569	3455	581	2990	865	3151	586	1769	622	1875	723	1743	778	1886
4	34	462	332	1996	375	2096	505	2435	823	2666	493	1588	632	1845	648	1693	776	2039
5	53	479	346	2773	556	2889	742	2888	908	3108	573	1705	636	1721	781	2018	820	2096
6	54	661	357	3018	464	2689	502	2307	875	3048	527	1601	583	1719	688	1868	760	1961
7	37	445	230	2436	351	2021	889	3356	678	2582	522	1627	582	1651	651	1795	841	1978
8	53	688	309	2421	965	5644	797	3579	764	3587	544	1721	593	1806	619	1625	784	1823
9	39	463	215	1991	437	2105	601	2790	624	2214	673	1706	684	1728	652	1698	856	1978
10	40	459	276	2218	460	2272	568	2202	873	2561	498	1655	683	1794	702	1883	793	1992
Average	49	562	298	2448	515	2834	642	2770	809	2898	542	1659	622	1769	692	1806	800	1966

10,000 Range

Trial	Seq		Parallel															
	1	1*	2	2*	3	3*	4	4*	5	5*	6	6*	7	7*	8	8*	9	9*
1	292	1604	158	1635	307	1815	385	2121	496	2020	487	1956	672	2237	729	2614	769	2238
2	272	1349	183	2039	315	2109	294	1829	454	2106	408	1729	625	2244	683	2356	811	2303
3	782	2129	183	2062	324	1827	316	1932	475	1935	510	2073	531	1947	647	2007	778	2423
4	773	2399	202	1854	216	1684	377	1627	389	1832	530	1995	746	2200	768	2220	827	2463
5	306	2154	146	1757	312	1760	430	1926	425	1721	572	2193	596	1985	725	2007	928	2619
6	249	1582	146	1495	400	2129	374	1689	454	1899	506	2141	502	1928	723	2403	771	2398
7	383	1869	188	1832	329	2170	322	1852	417	1774	459	1844	546	1993	654	2293	793	2522
8	235	1114	186	1803	247	1702	400	1837	446	2400	548	1995	641	1876	619	2181	822	2439
9	238	1305	176	1825	293	2144	355	1861	481	1916	504	2006	605	2149	813	2263	817	2314
10	701	2306	187	2001	301	1672	364	2193	499	2056	667	2230	620	2065	680	2563	797	2159
Average	423	1781	176	1830	304	1901	362	1887	454	1966	519	2016	608	2062	704	2291	811	2388

100,000 Range

Trial	Seq		Parallel															
	1	1*	2	2*	3	3*	4	4*	5	5*	6	6*	7	7*	8	8*	9	9*
1	2517	7562	133	5134	242	5054	313	5027	395	5466	506	5354	599	5851	682	5607	770	5043
2	2562	7804	123	5574	274	4713	277	5051	471	7533	477	5405	480	7651	544	5548	796	5545
3	5470	17058	109	6901	210	5592	342	5199	466	6378	443	5105	446	5141	648	5531	951	7386
4	2228	7205	157	5261	221	5880	332	5455	419	6559	469	5174	539	5524	596	5213	629	5315
5	3402	11049	116	5266	210	5326	288	5049	373	5158	430	4845	564	5335	645	5278	661	5454
6	6315	13075	142	5096	227	5323	269	4864	512	5469	430	5231	433	5609	594	5538	855	5995
7	2578	8131	116	5222	232	5492	359	5652	345	5503	450	5040	552	5847	609	5486	759	5454
8	2904	7996	152	8970	222	5668	286	4871	351	4846	434	5489	515	7190	743	5464	1061	6463
9	3315	10306	115	5116	214	6382	301	5488	411	5201	406	5476	574	5350	613	5026	782	6050
10	2927	8157	127	5163	215	6788	297	5249	347	4902	437	5050	543	5418	737	5484	682	6261
Average	3422	9834	129	5770	227	5622	306	5191	409	5702	448	5217	525	5892	641	5418	795	5897

1,000,000 Range

Trial	Seq		Parallel															
	1	1*	2	2*	3	3*	4	4*	5	5*	6	6*	7	7*	8	8*	9	9*
1	8170	24947	348	29381	443	40050	626	29139	380	19240	426	21047	556	21612	610	20447	815	19567
2	8113	23872	277	33018	474	39619	669	46722	369	19545	392	21783	667	20346	658	20683	771	19636
3	6853	21761	518	59964	394	26253	787	42042	426	18549	425	18826	600	25167	551	20617	762	19798
4	11162	32804	194	25712	514	41197	522	40854	375	19491	462	20940	480	21549	685	20977	770	19386
5	11270	33203	218	38557	339	25727	603	26812	384	17945	501	18398	616	20692	590	20406	744	19829
6	7803	24983	296	26997	471	52365	730	54022	374	17954	523	19819	550	20575	661	20987	783	19441
7	6472	20820	221	26133	446	48449	601	48886	393	18518	479	21813	567	20938	625	21316	781	19228
8	9708	25491	277	31326	391	46741	250	16158	333	18863	470	20747	568	22357	619	20528	1039	23795
9	9518	29797	290	38495	462	33429	263	18663	386	18310	438	20852	493	19953	671	20956	641	19220
10	8523	25910	212	23103	478	33088	259	18390	381	19345	430	19562	573	21518	595	20683	620	18730
Average	8759	26359	285	33269	441	38692	531	34169	380	18776	455	20379	567	21471	627	20760	773	19863

10,000,000 Range

Trial	Seq		Parallel															
	1	1*	2	2*	3	3*	4	4*	5	5*	6	6*	7	7*	8	8*	9	9*
1	57190	97622	122	73336	227	72211	296	66966	395	68287	463	67348	544	66089	661	64421	688	64594
2	67647	127895	125	73520	196	69723	310	66474	390	69785	489	69212	547	64927	646	65863	753	66822
3	54641	94579	121	72581	222	73259	370	65229	340	67810	438	66612	510	65291	678	63890	762	67177
4	54731	93579	120	73088	203	71492	253	65081	426	68989	437	62996	580	66979	654	66359	706	64134
5	53445	93946	115	72155	212	72649	256	67640	416	70200	387	68165	568	67750	606	65425	684	67210
6	92273	162421	119	74311	261	70782	262	67374	346	64124	507	67955	520	65372	678	66882	587	66086
7	53072	94146	123	73047	206	69641	304	65907	355	64730	426	65922	537	65963	678	64114	768	63922
8	55111	100508	200	74028	204	70853	264	74554	361	62726	443	63212	535	63900	643	66578	713	65489
9	57161	103125	118	73886	198	69149	272	67901	417	63792	476	63716	528	64135	608	65201	741	67877
10	66769	114292	118	73327	207	69944	267	67579	361	66072	520	67971	538	69022	615	67084	735	68018
Average	61204	108211	128	73328	214	70970	285	67471	381	66652	459	66311	541	65943	647	65582	714	66133

100,000,000 Range

Trial	Seq		Parallel															
	1	1*	2	2*	3	3*	4	4*	5	5*	6	6*	7	7*	8	8*	9	9*
1	690949	998605	132	802706	215	725436	317	683245	466	670400	490	584612	449	599666	655	619632	724	602185
2	745489	1068809	235	1052394	228	699885	279	682335	398	593276	427	610244	542	565315	643	612583	763	607084
3	762096	1135815	248	1063669	226	785211	257	616335	455	645291	529	572576	540	612544	646	602253	712	659064
4	794222	1148277	146	816859	217	723675	315	623198	372	620233	484	592155	534	576871	640	595417	653	647989
5	727440	1044654	128	841183	239	709934	311	638012	453	592902	463	616575	480	628381	632	632312	743	658872
6	859120	1179702	125	837619	212	660587	334	635237	365	608934	390	615780	507	608752	668	611430	780	612783
7	706437	1015123	125	837704	220	706881	298	631368	458	635116	517	611427	589	655723	648	621588	704	596816
8	713617	1038734	124	806247	220	703093	259	639335	427	603193	486	593132	509	639496	643	610129	694	659374
9	756111	1078538	130	833398	218	702729	272	630674	378	605314	468	581278	692	587859	579	607073	1087	643398
10	758372	1066340	209	830695	213	721397	280	643838	372	644993	463	624371	538	610814	697	621524	772	643353
Average	751385	1077460	160	872247	221	713883	292	642358	414	621965	472	600215	538	608542	645	613394	763	633092

1,000,000,000 Range

Trial	Seq		Parallel															
	1	1*	2	2*	3	3*	4	4*	5	5*	6	6*	7	7*	8	8*	9	9*
1	8544509	10202433	162	9849441	216	8944717	284	8172619	349	8613519	417	8594141	467	9064089	542	9517985	740	10571358
2	8560744	10228404	134	9840774	240	9145750	284	8586097	398	8740661	428	8520274	525	8927880	656	8511431	703	10222215
3	8539804	10209748	137	9881004	213	9156805	303	9207783	367	9132026	394	9060828	615	9392519	573	9096321	615	9712326
4	8387764	10038311	153	9858051	219	8889214	284	9372244	384	9146311	459	8939653	569	9426887	703	9481907	712	10593563
5	8426396	10076722	135	9878180	218	9322812	289	8935002	388	9093062	478	8894468	425	9282320	641	9362003	658	11072266
6	8457118	10096674	146	9901229	225	9215859	315	9222187	391	9656383	485	8765240	505	9269678	643	9123817	766	9985590
7	8441913	10092915	123	9863602	218	9863333	268	9203522	463	9060739	431	8705190	490	9281947	616	9952851	702	9749879
8	8445390	10086502	157	9867424	222	8587423	329	9118875	363	8676372	495	8493123	547	9088658	586	9121723	615	9196273
9	8435164	10082008	131	9890332	211	9900715	285	9292405	349	9166266	507	9226251	585	9614681	685	9083620	625	9162312
10	8397099	10042703	135	9891337	207	8933752	312	8978623	345	8553562	519	8817533	461	9454007	654	9636752	722	10399165
Average	8470978	10123746	141	9872137	219	9196038	295	9008936	380	8983890	461	8801670	519	9280267	630	9288841	686	10066495

1,073,741,824 Range

Trial	Seq		Parallel															
	1	1*	2	2*	3	3*	4	4*	5	5*	6	6*	7	7*	8	8*	9	9*
1	9113207	11010755	237	10487353	247	8913872	289	7635771	374	7976730	512	7363858	1424	7886463	657	7656945	679	8846346
2	8167159	9906090	128	10373666	660	8360571	316	7981558	411	8171216	509	7874636	586	7484427	623	8696261	834	8818163
3	8175785	9919529	124	10213938	207	8749177	276	8529011	447	7782602	484	7358929	567	8061661	651	8026209	700	8897983
4	8165818	9922432	456	10096130	217	8364567	334	7758479	370	7537896	465	7519442	615	7624119	620	8349499	773	8809142
5	8137969	9879818	139	10389523	241	9060314	303	8270198	434	7742475	412	7665151	530	7401356	674	8432180	2301	10621400
6	8147231	9889756	144	9980601	226	8632743	314	7944491	418	7878711	484	7408939	1428	7433024	758	8319015	1359	9489706
7	8158652	9889717	271	10307353	243	8167947	342	7964103	397	7916350	467	7772120	514	7784864	639	8165744	1566	10501049
8	8101508	9836323	215	10082744	246	8299139	294	7947134	375	7519202	500	7097869	563	7880149	772	8396324	2224	10828999
9	8150020	9892922	122	10285731	276	8303612	410	8064606	441	7590346	517	7805610	497	7859876	1767	8542163	1352	10187167
10	8122286	9855293	153	10160795	215	8711634	332	7994948	350	8275706	482	7742060	1789	7815086	653	8277455	2037	10907832
Average	8243964	10000264	199	10237783	278	8556358	321	8009030	402	7839123	483	7560861	851	7723103	781	8286180	1383	9790779

2,000,000,000 Range

Trial	Seq		Parallel															
	1	1*	2	2*	3	3*	4	4*	5	5*	6	6*	7	7*	8	8*	9	9*
1	16785036	20026154	137	19970864	231	17818702	291	16154224	486	16091186	541	16461568	554	16101172	502	16285273	970	17192995
2	16930977	20128710	139	20005082	222	17740649	343	16053740	448	17199173	538	15426995	568	16231848	650	16128832	739	18479669
3	16693117	19933896	136	20699534	233	17790295	307	16914685	422	16631227	516	15846032	529	16226975	650	16415046	685	18283554
4	16899744	20123805	137	20888844	289	18021491	303	16026007	429	17035406	405	15351325	577	16294669	602	16524167	781	17534876
5	16680600	19866871	133	20360196	233	17672495	292	16519045	495	15699828	409	16174127	599	15966883	680	16602744	831	18202074
6	16750346	19952798	138	20339183	211	17518472	302	15971765	423	15670953	490	15278439	532	15412761	710	16056539	741	18098330
7	16740559	19933644	125	20669637	232	17610255	321	17084854	369	16044842	449	15424026	534	15799281	667	17129456	837	17870120
8	16722214	20000693	144	19880957	246	17296107	338	15927209	428	16474484	448	15699053	574	15684693	597	16353118	733	18283928
9	16736102	19936648	137	19920924	240	18065419	309	15960666	388	15766599	480	16026594	506	16342274	748	16142653	693	17881419
10	17085390	20288561	136	19801872	212	17322759	283	16643791	364	16725123	418	15106420	512	16199816	596	15842569	686	17516801
Average	16802409	20019178	136	20253709	235	17685664	309	16325599	425	16333882	469	15679458	549	16026037	640	16348040	770	17934377

Appendix C – Results of Correlation Analyses

Result of the Parametric Test

		No. of Threads	Overhead Function
No. of Threads	Pearson Correlation	1	.840**
	Sig. (2-tailed)		.009
	N	8	8
Overhead Function	Pearson Correlation	.840**	1
	Sig. (2-tailed)	.009	
	N	8	8

**. Correlation is significant at the 0.01 level (2-tailed).

Result of the Non-Parametric Test

			No. of Threads	Overhead Function
Kendall's tau_b	No. of Threads	Correlation Coefficient	1.000	1.000**
		Sig. (2-tailed)	.	.
		N	8	8
	Overhead Function	Correlation Coefficient	1.000**	1.000
		Sig. (2-tailed)	.	.
		N	8	8
Spearman's rho	No. of Threads	Correlation Coefficient	1.000	1.000**
		Sig. (2-tailed)	.	.
		N	8	8
	Overhead Function	Correlation Coefficient	1.000**	1.000
		Sig. (2-tailed)	.	.
		N	8	8

**. Correlation is significant at the 0.01 level (2-tailed).

Appendix D – Cost of Performance

Threads	% Overhead	% Speedup	% Effciency
1	0.00	0.00	0.00
2	49.61	1.00	50.00
3	60.72	18.00	39.00
4	68.53	26.00	31.00
5	74.24	29.00	26.00
6	77.97	32.00	22.00
7	81.39	30.00	19.00
8	84.80	22.00	15.00