AN APPROACH TO MODELLING AND SIMULATING MULTITHREADED SCHEDULERS FOR DIVIDE AND CONQUER PROBLEMS ON MULTICORE ARCHITECTURE

ALAA MOHAMMED ALI WADI AL-OBAIDI

FACULTY OF COMPUTER SCIENCE AND INFORMATION
TECHNOLOGY
UNIVERSITY OF MALAYA
KUALA LUMPUR

2016

AN APPROACH TO MODELLING AND SIMULATING MULTITHREADED SCHEDULERS FOR DIVIDE AND CONQUER PROBLEMS ON MULTICORE ARCHITECTURE

ALAA MOHAMMED ALI WADI AL-OBAIDI

THESIS SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
UNIVERSITY OF MALAYA
KUALA LUMPUR

2016

UNIVERSITI MALAYA

ORIGINAL LITERARY WORK DECLARATION

Name of Candidate: **Alaa Mohammed Ali Wadi Al-Obaidi**

Passport No:

Registration/Matric No: **WHA080001**

Name of Degree: **PhD in Computer Science**

Title of Project Paper/Research Report/Dissertation/Thesis ("this Work"):

AN APPROACH TO MODELLING AND SIMULATING MULTITHREADED SCHEDULERS FOR DIVIDE AND CONQUER PROBLEMS ON MULTICORE ARCHITECTURE

Field of Study: **Concurrency Modelling**

I do solemnly and sincerely declare that:

(1) I am the sole author/writer of this Work;

(2) This Work is original;

(3) Any use of any work in which copyright exists was done by way of fair dealing and for permitted purposes and any excerpt or extract from, or reference to or production of any copyright work has been disclosed expressly and sufficiently and the title of the Work and its authorship have been acknowledged in this Work;

(4) I do not have any actual knowledge nor do I ought reasonably to know that the making of this work constitutes an infringement of any copyright work;

(5) I hereby assign all and every rights in the copyright to this Work to the University of Malaya ("UM"), who henceforth shall be owner of the copyright in this Work and that any reproduction or use in any form or by any means whatsoever is prohibited without the written consent of UM having been first had and obtained;

(6) I am fully aware that if in the course of making this Work I have infringed any copyright whether intentionally or otherwise, I may be subject to legal action or any other action as may be determined by UM.

Candidate's Signature Date / /2016

Subscribed and solemnly declared before,

Witness's Signature Date / /2016

Name:

Designation:

ABSTRACT

The continuous increase in the number of cores and software size causes a distinct problem in the software world that utilizes multicore architecture. This problem is represented by the optimal use of the new technology and how this is reflected in software development. In the core of the problem, there are two main issues that must be considered. First, the partitioning of the workload of a problem at runtime so that the resultant workload partitions can be processed concurrently. Second, the dynamic balance of the workload that is generated by these partitions to be distributed among the cores. This matter is highly important because it addresses the problem of idle cores. In order to handle the problem of idle cores, this thesis adopts the work-stealing technique which has been successfully applied in multiprocessor systems to provide a workload balance between the multiprocessor systems by allowing the idle processors to work individually to steal part of the workload of the non-idle processors at run time so that the system can be balanced. However, as the number of cores increases, which may reach several hundred in the near future; it will be time consuming to allow each core to individually search for a non-idle core to steal part of its workload since the searching process in the existing work-stealing techniques is done randomly. This causes frequent failure especially when the workload is low and many cores are in an idle situation.

This thesis proposes an approach to partition Divide and Conquer algorithms into workload partitions at run time so that they can be executed concurrently on a scaled multicore architecture. Therefore, the researcher proposes several problem oriented mechanisms to partition the workload. In addition, the researcher proposes a modification to the work-stealing technique by imposing a centralized control over the stealing process rather than allowing each core to work individually. Several rebalancing strategies are proposed to suit the conditions of the cores. To achieve these goals, the researcher designs scaled concurrent models that work under the principle of multithreaded scheduling. Two types of schedulers are proposed. The first type is responsible for creating, dividing, and manipulating the threads of the Divide and Conquer algorithms. The second type of schedulers is for balancing the threads using different rebalancing strategies. The researcher uses Colored Petri Nets as language of modelling and Colored Petri Nets Tool as the software that creates, simulates, and validates the models.

The results of simulation models show a high efficiency in dealing with Divide and Conquer algorithms. The proposed concurrent models are scalable in terms of number of cores and problem size. The models can be easily expanded by adding more cores which influence effectively on the models' performance. In other words, the results indicate that adding more cores minimizes the number of steps required to complete the simulation process of the models. In addition, the models show a high flexibility in dealing with various problem sizes, and maintain the integrity of results even when problem size is highly increased.

ABSTRAK

Peningkatan yang berterusan dalam bilangan teras dan saiz perisian menyebabkan satu masalah yang nyata telah muncul dalam dunia perisian yang menggunakan senibina berbilang teras. Masalah ini diwakili oleh pengunaan teknologi baru yang optimum dan bagaimana ia dapat digambarkan dalam pembangunan perisian. Terdapat tiga isu utama yang perlu dipertimbangkan dalam masalah utama tersebut. Pertama, pembahagian beban kerja satu masalah pada masa larian supaya partisi beban kerja yang dihasilkan boleh diproses serentak. Kedua, keseimbangan dinamik bagi beban kerja yang dijanakan oleh partisi-partisi ini antara teras tersebut. Perkara ini adalah sangat penting kerana ia perlu menangani masalah teras terbiar, atau tidak bekerja. Dalam usaha untuk menangani masalah teras terbiar, tesis ini menerima pakai teknik pencurian kerja yang telah berjaya diaplikasikan dalam sistem berbilang pemproses untuk mencapai keseimbangan beban kerja antara sistem berbilang pemproses dengan membolehkan pemproses terbiar untuk bekerja secara berindividu bagi mencuri sebahagian daripada beban kerja pemproses sibuk pada masa larian supaya sistem boleh diseimbangkan. Walau bagaimanapun, apabila bilangan teras semakin meningkat di mana ia mungkin boleh mencapai beberapa ratus dalam masa terdekat, ia akan memakan banyak masa untuk membenarkan setiap teras untuk mencari teras sibuk secara berindividu bagi mencuri sebahagian daripada beban kerjanya. Ini adalah disebabkan oleh proses pencarian dalam teknik pencurian kerja yang sedia ada dilakukan secara rawak. Ini menyebabkan kegagalan yang kerap terutama apabila beban kerja adalah rendah dan banyak teras berada dalam keadaan yang terbiar. Tesis ini mencadangkan satu pendekatan untuk membahagikan algoritma Membahagi dan Menakluk ke dalam partisi-partisi beban kerja pada masa larian supaya mereka boleh dikasanakan serentak pada senibina berbilang teras yang diskalakan. Oleh itu, penyelidik mencadangkan beberapa mekanisme berorientasikan masalah untuk mebahagikan beban kerja. Di samping itu, penyelidik mencadangkan pengubahsuaian kepada teknik pencurian kerja dengan megenakan kawalan berpusat ke atas proses pencurian tersebut dan bukannya membenarkan setiap teras untuk bekerja secara individu. Beberapa strategi pengimbangan semula dicadangkan untuk disesuaikan dengan keadaan teras. Untuk mencapai matlamat ini, penyelidik merekakan model serentak yang diskalakan di mana mereka bekerja di bawah prinsip penjadualan berbilang bebenang. Dua jenis penjadual dicadangkan. Jenis pertama adalah bertanggungjawab untuk menciptakan, membahagikan, dan memanipulasikan benang algoritma Membahagi dan Menakluk. Jenis kedua adalah penjadual untuk mengimbangi benang yang menggunakan strategi pengimbangan semula yang berbeza. Penyelidik menggunakan Jaring Petri Berwarna sebagai bahasa pemodelan dan alatan Jaring Petri Berwarna sebagai perisian yang mencipta, simulasi, dan mengesahkan model tersebut. Keputusan model simulasi menunjukkan kecekapan yang tinggi dalam menangani algoritma Membahagi dan Menakluk. Model serentak yang dicadangkan adalah berskala dari segi bilangan teras dan saiz masalah. Model tersebut boleh diperluaskan secara mudah dengan menambahkan lebih banyak teras di mana ia mempengaruhi prestasi model tersebut secara berkesan. Dalam erti kata lain, keputusan menunjukkan bahawa penambahan lebih banyak teras boleh mengurangkan bilangan langkah yang diperlukan untuk menyelesaikan proses simulasi untuk model tersebut. Di samping itu, model tersebut menunjukkan fleksibiliti yang tinggi dalam menangani masalah yang berbagai saiz, dan mampu mengekalkan integriti keputusan walaupun saiz masalah meningkat secara tinggi.

ACKNOWLEDGEMENTS

First and foremost, praises and thanks to the God, the Almighty, for the good health and tranquillity that were indispensable to me since the first day in this study. I am grateful to my supervisor, Professor Dr. Lee Sai Peck, for accepting me as her student. Her personality, follow up, and patience were essential to complete this thesis. I am also grateful to the Faculty of Computer Science and Information Technology for providing me all the requirements to conduct this research.

I am very grateful to my wife and sons for their patience; they bore a great burden during the period of my studies. I also thank my mother and sisters for their support. Special thanks to my faithful friend Alaa Al-Janabi. I would also like to express my gratitude to Dr. Michael Westergaard and Professor John Reppy for clarifying some issues related with the modelling tool and the programming language.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AES	Average of Execution Steps
AMD	Advanced Micro Devices
BS	Binary Search
BSLLS	Binary Search Low-Level Scheduler
CAS	Compare and Swap
CL	Cores Load
CLIn	Cores Load In
CLOut	Cores Load Out
CMSS	Complete Multi Stealing Strategy
CPN	Colored Petri Nets
CPN-ML	Colored Petri Nets' Meta Language
CPN-Tool	Colored Petri Nets' Tools
CPU	Central Processing Unit
D&C	Divide and Conquer
Des	Destination
DistGuard	Distributor's Guard
DMMLLS	Direct Multi Stealing Low-Level Scheduler
DNo	Disk's Number
EC	End Column
EI	End Index
Ele	Element
ER	End Row
FatherId	Father Identity
FLLS	Fibonacci Low-Level Scheduler
GUI	Graphical User Interface
HLS	High-Level Scheduler
IBM	International Business Machines
IOMSS	In Order Multi Stealing Strategy
IOSSS	In Order Single Stealing Strategy
JAWS	Java Work Stealer
LLS	Low-Level Scheduler
LM	List of Moves
LN	List of Numbers
LT	List of Threads

MIT	Massachusetts Institute of Technology
MM	Matrix Multiplication
MMLLS	Matrix Multiplication Low-Level Scheduler
NumList	Numbers List
Ord	Order
PMSS	Partial Multi Stealing Strategy
PN	Petri Nets
POSIX	Portable Operating System Interface
Res	Result
ResIn	Result In
ResOut	Result Out
RFMSS	Richest First Multi Stealing Strategy
RFSSS	Richest First Single Stealing Strategy
SC	Start Column
SI	Start Index
SML	Standard Meta Language
SMP	Symmetric Multi-Processor
Sou	Source
SR	Start Row
TBB	Threading Building Blocks
TH	Towers of Hanoi
ThL	Threads List
THLLS	Towers of Hanoi Low-Level Scheduler
Thr	Through
ThreadId	Thread's Identity
TPL	Task Parallel Library
ZAPP	Zero Assignment Parallel Processor

LIST OF APPENDICES

CHAPTER 1: INTRODUCTION

1.1 Background

In 1965, a physical chemist named Gordon E. Moore predicted that the number of electronic components placed on an integrated circuit is going to be doubled every year. In other words, according to the Moore’s prediction, the computing power is going to be doubled every year (Mollick, 2006). After ten years, this prediction was revised by the same scientist to be two years instead of one year. David House, an Intel executive at that time, modified Moore’s prediction to be eighteen months rather than two years (Nambiar & Poess, 2011). This prediction became well known as Moore’s Law. The law continued its impact on the hardware industry for nearly four decades. During these decades, all the manufacturing efforts were directed to improve the single-processor computers through increasing the clock speed of these processors (Herlihy, 2007). However, a barrier of physical limit stood against the manufacturers’ desire. The power consumption and the increase in the generated heat due to the continuous adding of more transistors were the main obstacles against the continuity of this law (Mack, 2011).

1.1.1 Multicore Technology and Software Industry

At the beginning of twenty-first century, the processors industry has witnessed a dramatic change in its production. Chip manufacturers stopped their race in manufacturing high-speed single-processor computers, as a result, ending the era of single-processor improvement (Breshears, 2009). What is more, they announced the starting of the multicore architecture era. The new architecture has been built on the basis of replicating the processing element (core) rather than focusing on the processor improvement. This replication becomes the key factor to measure the performance of any modern computer system (Herlihy, 2007). In 2001, IBM released the first commercial microprocessor that has built on the multicore technology. The POWER4 microprocessor has two cores built on a single chip (Davis & Burns, 2011). Since that date, the general trend adopted by the giant microprocessor manufacturers such as Intel, AMD, IBM, and Sun was represented by the continuous adding of more cores to their microprocessors (Geer, 2005). The race between the hardware manufacturers has continued to produce chips with a higher number of cores. However, the success in adding more cores is not the ultimate goal for these manufacturers. There are certain requirements related with the core technology that are needed to be achieved such as the execution optimization and cache size (Sutter, 2005). The significant change in the hardware side expanded to cover different types of personal computers such as desktop, laptop, notebook, ultra book and tablet, and it also covers communication devices such as mobile and any other computerised devices (Wikipedia, 2014a). In the software side, software designers had totally depended in the past on the clock speed in producing faster software. However, the improvement in the hardware side has made a real impact on the software industry as well. This is because the new hardware offers more than one processing element (core) that can operate at the same time (Sutter, 2005). Therefore, software designers have no choice but to update their products to deal with the new changes in the hardware side. This kind of update has to consider the continuous increase in the number of cores per chip. That is, modern software should not be restricted to deal with a fixed number of cores. It should be adapted to deal with any number of cores in addition to exploiting these cores as much as possible (Sutter, 2005). Although this is not an easy task since most of the algorithms used in the software industry are designed to serve serial computations. However, failure to develop algorithms that suit the new environment will, undoubtedly, stand as a stumbling block against the optimal utilization of the new architecture. As a result, a new trend has imposed in the software industry through improving the aspect of concurrency of software products in addition to making these products more scalable to deal with any number of cores (Ding, Wang, Gibbons, & Zhang, 2012; Sutter & Larus, 2005). Therefore, concurrency and scalability are the two main features that modern software should gain for the purpose of making the best use of the multicore architecture.

1.1.2 Concurrency, Parallelism, and Multithreading

Concurrency has a realistic application in the physical world in which we live. In the process of building a new house, certain actions can be done at the same time such as plumbing, electrical work, painting, tiling, etc. In computer science, the idea of concurrency was closely related to the time-sharing principle of work. Back to 1960s, mainframe computers shared their CPUs with several consoles and each CPU has its own running program (process) (Bryant & David Richard, 2003). To increase the overall utilization, a software model called "Scheduler" was designed to dedicate a CPU to one of the processes for a specific period of time. After that, the Scheduler redirects the CPU to another process. This gives the perception that all these processes are working at the same time (concurrent executions). Another issue has also emerged since the early eras of computers. The difference between the speed of CPUs and peripheral devices caused a real headache to the scientist (Tucker, Barlow, & Stuart, 2012). As computers have become faster and faster, the gap between the two speeds has been widened. Moreover, computer networks add more burden to the overall performance since the speed of network components definitely cannot be compared with the speed of CPUs. Although there is no magical solution for these problems since the gap between these speeds is still big, it is possible to reduce the severity of the problem by developing more advanced schedulers. Different schedulers have different ways to balance requests of processes.

For a long time, the concurrency and parallelism concepts have been used interchangeably (Bryant & David Richard, 2003). This came from the fact that both the concurrency and parallelism refer to the state where computers are able to deal with more than one program (process) at the same time. However, there is a major difference between these two concepts. In single-processor computers, a system is said to be concurrent if it is able to run more than one process at the same time. These processes cooperate with each other in exploiting the processor time through interleaving. However, in parallelism, we have more than one processor, each of which has a separate running process. Usually a process typically consists of one or more pieces of code called threads. In general, a thread is the smallest sequence of instructions that can be managed by a scheduler. The threads that belong to the same process share the same area of memory. Thus, we can write concurrent programs that consist of multiple threads; however, if the computer does not include multiple processors, then these threads will not be executed in parallel. Therefore, we can conclude that concurrency represents the general case while parallelism represents the sub case (Knuth, 1968).

Multithreading is a mechanism of creating threads of executions. It enables running more than one program concurrently. A good planning for multithreading can provide an excellent utilization of computer resources. The basic concept of multithreading has been around for some time, but gained wider attention as computers became more commonplace during the decade of the 1990s. Dynamically growing multithreaded computations are nowadays quite common for multicore systems. One should specify which core executes which threads and when each thread should be executed; obviously, there is an urgent need for efficient schedulers. The efficient execution of such schedulers depends heavily on the runtime system. A good scheduling technique must ensure that enough threads remain active to keep the cores busy, while at the same time, the concurrent active threads must be within their limit in order to control the memory needed. Moreover, in order to reduce communication among cores, one should try to maintain related threads on the same core. Providing a scheduling technique to achieve all of the above goals is not a trivial task (Fatourou & Spirakis, 2000).

1.1.3 Scheduling Algorithms in Multicore Systems

In a multicore system, there are basically two sets of cores, i.e. working (busy) and non-working (idle) cores, at any one time (L. M. Nogueira, Pinho, Fonseca, & Maia, 2013; Tchiboukdjian, Danjean, Gautier, Lementec, & Raffin, 2010; Y. Wang, Ji, Shi, & Zuo, 2013). Working cores are in action, that is, they have their own threads currently in execution. Non-working cores are idle (out of threads). The reason behind having a core in an idle situation is that either its threads have already completed their assignment tasks or the scheduler did not assign any threads to the core yet. Naturally, at any time, the status of a working core may change to non-working, and vice versa. The ultimate goal for any software company is to direct their products to make the most use of the available cores (concurrent working) as much as possible and never let any core in an idle situation (Sutter & Larus, 2005). Despite this seeming to be unrealistic for all kinds of problems due to the nature of a problem, however, in certain problems, good results could be achieved. Therefore, in a multicore system, in order to reach to a high level of concurrency, a scheduler plays a major role in achieving this important objective through adopting the best scheduling algorithm that balances the working load among the cores (Quintin & Wagner, 2010; Tchiboukdjian, et al., 2010).

During the past decades, there were many scheduling techniques that have been developed for multiprocessor systems. These scheduling techniques fall into two categories: static and dynamic scheduling. In static scheduling, all the information related with tasks’ time-slice, synchronization requests, communication and dependency with other tasks are known and planned for before starting the execution, that is, at the compilation time (Kwok & Ahmad, 1999). In general, static scheduling guarantees the execution of the tasks on-time, in addition to being low cost. However, static scheduling suffers from certain drawbacks such as wastage of the processor time, besides that, no task exceeds the time slice assigned to it; additionally, any changes in the program sequences need a rescheduling. Moreover, modern computer systems with shared caches added several new drawbacks for static scheduling such as the difficulty to provide an accurate information about the tasks, the limitation of portability since the result of scheduling is directed for a specific architecture (Mattheis, Schuele, Raabe, Henties, & Gleim, 2012). The changing of data and input during run time plays a main role in task scheduling which is totally incompatible with the static mechanism (Breshears, 2009). Finally, static scheduling is inconsistent with the multicore environment. In such environment, it is so common to execute more than one program at the same time where each program consists of a set of processes. The operating system in a multicore environment assigns these processes to the available cores at run time which results in a continuous change in the number of the utilized cores during the execution of these processes. This stands against the principle of static scheduling where the execution time of these processes should be known prior to the execution. However, providing such information about the execution time is particularly hard to obtain for a multicore environment (Mattheis, et al., 2012).

In contrast to static scheduling, dynamic scheduling is implemented at run time that is on-the-fly (Kwok & Ahmad, 1999). Dynamic scheduling succeeded in fulfilling the requirements of the multicore technology since a dynamic scheduler has the ability to create new tasks and assign them to the cores at the execution time, a condition that becomes a must for most applications that wish to exploit the multicore architecture to the full extent. In general, within dynamic scheduling, two main objectives have been achieved; first, it becomes possible to balance workload at run time which makes the process of task/core mapping much easier and it achieves great benefits. Second, it becomes much easier to deal with different hardware architectures. On the other hand, the cost of these benefits is expected to be higher run time scheduling overhead and additional application development complexity (Kwok & Ahmad, 1999; Mattheis, et al., 2012).

1.1.4 Divide and Conquer Problems on Multicore Environment

In this thesis, the researcher focuses on solving Divide and Conquer (D&C) problems on a multicore environment. The term Divide and Conquer is also used to describe the method of solving such kind of problems. In computer science, every D&C problem has its own method of solving. That is, for example, the binary search differs from the matrix multiplication, yet all D&C methods share some common steps. In general, every D&C method is built on breaking the main D&C problem into two sub problems that can be recursively broken into smaller sub problems, then solving these sub problems, and recursively combining the results of these sub problems to form the final result (Cormen, Leiserson, Rivest, & Stein, 2009; Miller & Vandome, 2010). The D&C methods fit in the multicore environment, that is, any D&C problem can be split and distributed to the cores that work in a concurrent way at run time to reach the final result with the shortest time possible (Neill & Wierman, 2009; Tardieu, Wang, & Lin, 2012). Taking into account the multicore development, these methods should be adapted to be more efficient in order to suit the multicore architecture. One of the main ideas is to provide a concurrent multithreaded scheduler model that can dynamically manage threads creation and load distribution among the used cores.

1.2 Motivation

The multicore technology provided the solution for the shortage that single-processor computers suffered from. A landmark achievement has been made through the success in replicating the number of cores per chip. However, software industry is still not consistent with development in the hardware side. The lack of providing suitable techniques that make software more consistent with the key technology represents the main factor that affects the software industry. This study motivates two needs that find them necessary to promote software industry.

1.2.1 The Need for Modelling Concurrent Systems

Concurrent systems are complex, difficult to design and error prone. All these come from a common characteristic that concurrent systems share, that is, non determinism. One of the main challenges facing concurrent systems' designers is the non deterministic behaviour of these systems (Jensen, Kristensen, & Wells, 2007). In non concurrent systems, regardless of the number of executions, the transition during the execution, say from state A to state B is predetermined (deterministic). Even with the decision (like if or if …else …) and selection statements (switch), there are few predetermined states that come after it in an execution order. However, the execution of a concurrent system may carry on in different ways each time the execution is resumed. This is due to the large number of possible paths that can be generated from each state; nevertheless, all these paths should lead to the same final result. Taking into account this fact, the process of building concurrent systems that control critical and/or dangerous projects such as atomic power plants, aircraft control systems, etc should be error-free and well designed, otherwise it will lead to disaster (Jensen & Kristensen, 2009). As a result, such systems must be tested and debugged thoroughly prior to any real implementation. Using the traditional methods in debugging such as the inclusion of breakpoints is no longer effective as in non-concurrent programming due to continuous change of execution behaviour from run to run caused by the astonishing number of intermediate states towards reaching the final state (Wells, 2002).

Modelling provides a solution for this problem. Through modelling, many errors and weak points in the concurrent software can be identified and corrected. Building a well-designed model has three main advantages:

(a) Insight Look: Modelling gives an insight look into the system. It gives a general description of the system architecture and its mechanism. That is, it shows how the concurrent system has been designed, the way of performing the system's actions, and data paths inside the system. The designer should utilize these details to improve the simplicity and the usability of the model's parts such as those related to processing and storing data, in addition to the linkage paths that connect the processing and storing parts. Moreover, the insight look improves the obviousness of the design and should remove repeated acts to the fullest extent possible. This will definitely benefit the designer since it gives a comprehensive understanding of the system (Jensen, 1998).

(b) Specifications Completeness: The process of simulating the model reveals a lot of gaps in the model's specifications that clearly show the model's real capabilities as well its shortages. In addition, the model's requirements can be judged accurately as fully achieved or partly lost (Jensen & Kristensen, 2009).

(c) Model's Correctness: Definitely any model cannot be accepted until it is simulated and it generates proper results. Through the simulation process, many faults can be diagnosed and corrected. In addition, a verification test is then needed to verify all the system states so that no state is unreachable nor there is a chance for deadlocks to occur (Kristensen, 2000).

The above three advantages that modelling offers will not be achieved unless there is a proper modelling language and modelling tool that assists the designer in planning, simulating, and verifying the design. In this study, the researcher uses Colored Petri Nets (CPN) as a graphical language for building and analyzing models of concurrent systems. CPN has been developed from Petri Nets (PN) as being the origin of CP (Murata, 1989; Peterson, 1977). There are two main differences between CPN and PN: CPN has included the idea of data types besides the use of expressions and functions written in Standard Meta Language (SML) (Gansner & Reppy, 2004; Ullman, 1998). As a software tool, the researchers uses CPN-Tool (Jensen, Christensen, Kristensen, & Westergaard) which is developed by Kurt Jensen (Jensen & Kristensen, 2009). CPN-Tool provides all the necessary facilities to create, simulate, and validate Colored Petri Nets. In addition, it provides interaction methods such as menus and toolbars besides giving feedback messages when errors are encountered during the process of performing code's syntax checking. CPN-Tool uses Colored Petri Nets' Meta Language (CPN-ML) (Jensen, et al.) as a language of writing declarations, expressions, and code inside the model. CPN-ML has been built based on SML (Gansner & Reppy, 2004; Ullman, 1998). Appendix I includes more details about CPN.

1.2.2 The Need for New Techniques in Partitioning and Balancing Workload for Solving D&C Problem

Processor manufacturers are continuing in developing multicore technology towards replicating the cores. Instead of working towards producing more efficient cores, the real trend of these manufacturers is toward assembling more cores in one processor. This matter puts software developers in facing a big challenge which is the ability to utilize this growing number of cores. On the other hand, the omission of exploiting these cores causes the failure to achieve the real benefit from the purpose for which multicore was developed: achieving high speed in execution time through replicating the cores. In other words, the lack of providing new software techniques that make use of these cores to the full extent will no doubt lead to an imbalance in the workload distribution. During the running of a multicore system, this imbalance happens in the form of having two sets of cores: working (active) and non-working (idle), where the number of idle cores increases far from the desired goal of the multicore technology. Therefore, it is necessary to develop new techniques for partitioning and balancing workload for the sake of achieving a high level of concurrency among the utilized cores in the multicore environment.

In general, given the increase in the number of cores, it becomes gradually more important to boost the concurrency level between the cores through making these cores busy as much as possible. This can be achieved through the development of new techniques to partition and balance the workload of problems at runtime. The researcher focuses on the D&C as an example for problems, working on making such problems more adaptable with multicore environment through adopting those techniques.

1.3 Problem Statement

Multicore technology has succeeded in solving the drawback in the single-processor environment, that is, to get more powerful computers, more cores have to be added. However, the evolution in the hardware creates a real challenge for software designers. This challenge is represented in the ability of software to deal with this growing numbers of cores. The researcher categorizes the challenge into three groups:

(a) Workload Partitioning

As stated before, the multicore technology has been built on the basis of replicating the processing units (cores). Having more than one core working at the same time triggers the need for mechanisms to partition the workload. The need for workload partitioning was not urgent prior to the multicore technology because there was only one processor.

However, the significance of workload partitioning begins growing and has become a necessity to be utilized in the new multicore architecture. In this thesis, the researcher highlights several issues related with this kind of processes:

I- Prior to any partitioning process, are the traditional ways in representing the workload in single-processor computers still appropriate and could be used effectively in the multicore technology?

II- How can we partition the workload in a way that fits in the multicore environment? What sort of mechanism can be adopted for the partition?

III- Does the partitioning process follow a unique mechanism that fits all the types of problems? For instance, considering the D&C problems that the researcher focused on in this thesis, is it possible to apply the same partitioning technique for all given D&C problems?

IV- Do the partitioning techniques proposed in this study have the properties that qualify them to work in a multicore environment? Specifically, are the partitioning techniques scalable to deal with a variable number of cores? Do the partitioning techniques support concurrent actions?

(b) Workload Balancing

The ultimate goal of the partitioning process is to employ the maximum number of available cores so that workload balancing can be achieved. However, there are certain issues related to this matter:

I- How to distribute the partitioning workload? What are the strategies that can control the distribution process?

II- Does the work-stealing technique still appropriate with the growing number of cores? How can we improve this technique to be more adaptable for the increasing number of cores?

(c) Core Computations

One of the most important objectives of the multicore technology is to make all cores involved in solving a problem. That is, each core will be responsible for part of the problem. This requires reconsideration to the core's computations compared to what was previously in the single-processor computers. In this context and in relation to the D&C problems:

I- Are the original methods for solving D&C problems still appropriate to apply on the multicore environments, and why?

II- On what basis, the researcher attempts to build the D&C solving method that suits a multicore environment? In other words, what are the improvements that this work will have to make on the D&C problems in order to fit in the multicore architecture?

III- How can we coordinate the work between the strategies and the methods?

(d) Correctness and Validation

The strategies and methods both suggested new techniques to deal with D&C problems on a multicore environment. The researcher highlights certain issues related with this issue:

I- How can we ensure that these strategies and methods work just fine and they are able to generate results in addition to being error free?

II- Knowing that there are no errors and correct results can be generated, how to ensure that neither the strategies nor the methods may cause deadlock or data race?

1.4 Research Objectives

In this study, the researcher proposes a concurrent multithreaded scalable model. The model is dedicated to solve the D&C problems (Fibonacci Series, Binary Search, Towers of Hanoi, and Matrices Multiplications) on a multicore environment. The objectives of this study can be identified as follows:

(a) To propose a workload distribution scheduler that is able to control the workload distribution of the modelled cores.

The proposed scheduler should have the following properties:

I- The distribution process of this scheduler is controlled by a set of strategies which control the distribution of the partitioned workload. The working principle of these strategies is based on balancing the threads among the cores.

II- The proposed scheduler is concurrent and scalable. The scheduler deals with all the cores concurrently. In addition, the scheduler can deal with an open number of partitioned threads.

(b) To propose a core scheduler that has the ability to partition the workload and find a solution for each D&C problem. The proposed scheduler resides in every modelled core of the multicore model and it has the following properties:

I- The proposed scheduler works under the principle of multithreading, that is, the D&C problems are represented as threads. The scheduler's task is to partition the threads residing in its core into two or more threads.

II- The partitioning technique of the proposed scheduler is not unique; it depends on the type of the problem.

III- The core scheduler provides a solution for the D&C problems.

(c) To build CPN models implementing the two proposed types of schedulers.

(d) To perform simulation and monitoring of the models targeting at the reduction of idleness of the modelled cores to the maximum extent.

1.5 Research Significance

The potential impacts of the proposed research are in two directions:

(a) The partitioned techniques that the researcher proposed in this study may represent a forward step towards improving the execution of the D&C problems on the multicore environment. The value of these techniques comes from being fully fit in this environment taking into consideration the continuous increase of the number of cores in different multicore platforms such as laptops, tablets, mobiles, etc.

(b) The proposed balancing strategies would provide a new perspective to achieve workload balance among the cores. As stated before, the general trend of the manufacturers of processors stresses on adding more cores instead of improving the speed of the core itself. This will definitely trigger the need for balancing techniques.

1.6 Thesis Scope and Assumption

The scope of this thesis covers D&C problems. Four D&C problems have been taken as examples, namely Fibonacci Series, the Towers of Hanoi, Binary Search and Matrix Multiplication. The rest of D&C problems are assumed to be able to follow the same procedure that the researcher proposed in this study. However, the partitioning techniques are problem oriented. In other words, solving another D&C problem may need a specific partitioned technique. On the other hand, the strategies of balancing can be applied to any D&C problem.

The models that this thesis presents suit multicore architectures and fit well with a group of cores that share a common memory. Due to the high level of threads' exchange among the cores, it would be costly in terms of communication when these models are applied on non-shared memory systems. In addition, there is a need to assign one of the cores to control the redistribution of threads among the cores. Although assigning one core may affect the efficiency of the work, however, with the diminishing growth in the number of cores, the allocation of a single core to manage the scheduling process will not have a significant impact on the overall performance.

1.7 Thesis Outline

This thesis consists of six chapters and two appendices.

(a) Chapter 2 reviews the contributions of the existing research in the field of load balancing algorithms. The researcher highlights the main classifications of these algorithms with a particular focus on the classification of multithreading scheduling. This chapter includes a description of the work-stealing evolution, its significant achievements, and its contribution in the software industry.

(b) Chapter 3 is dedicated for the research methodology. This chapter includes a description of the way in which the researcher conducted the study. At the beginning, the chapter starts with a description on how the research idea arose. Then, the researcher shows on what basis the literature review has been conducted. After that, depending on the research idea and the literature review, the researcher defines the shortages and gaps in the existing researches, and then, the researcher determines the problem statement and the objectives of the study. Following that, the researcher defines, without going into details, the techniques that the researcher suggested for dividing the workload, in addition to the strategies of balancing. After that comes into play the simulation and monitoring section. In this section, the researcher explains how the proposed techniques are transformed into the elements of CPN models. Finally, this chapter ends with a description of the results that gathered from the execution of the models, in addition to discussing these results.

(c) Chapter 4 is reserved for the design methodology. The partition techniques of the D&C problems are fully explained in this chapter. In addition, the chapter explains in detail the modifications the researcher suggests for work-stealing technique and how to apply these modifications through the proposed balancing strategies. The chapter is supported by flowcharts that show the mechanisms of the partitioned techniques and the balancing strategies. In addition, the chapter includes the designs of the CPN models. This chapter also explains how the researcher builds CPN models. This includes the elements of the models, representation of the threads, and the representation of the partitioned and balancing mechanisms inside the models. Moreover, the chapter shows how to simulate and validate the models. This chapter is supported by an appendix (Appendix I), placed at the end of this thesis. Appendix I give more details about CPN, CPN-Tool, and the simulation and validation processes.

(d) Chapter 5 is reserved for the results and discussion. This chapter includes all the results that have been obtained from the simulation and monitoring processes of the proposed CPN models. The results appear as graphs that show the relation between the number of cores and the execution steps. In addition, the chapter includes a detailed explanation of the results that have been obtained.

(e) Chapter six is dedicated for the conclusion and future work. This chapter summarizes the problem addressed in this study and the purpose of this research. Then, the chapter briefly explains how the research was conducted, the proposed schedulers' mechanisms and the CPN models. Finally, the chapter discusses some of the possible future studies. These studies comprise the development of the threads' structure and the mechanisms of the schedulers.

CHAPTER 2: LITERATURE REVIEW

2.1 Introduction

A new fact has imposed itself on the processor industry with the beginning of the present century, i.e. processor manufacturers are no longer able to achieve a remarkable development on the processors' speed as was the case during the 80s and the 90s of the past century. This fact has made multicore technology a suitable solution to meet the challenges faced by the processors industry. Ostensibly, the problem seemed to be resolved at least from the hardware point of view, however, adding more cores did not attain what was expected from the new architecture. Many studies (Chhabra, Singh, Waraich, Sidhu, & Kumar, 2006; Rudolph, Slivkin-Allalouf, & Upfal, 1991; Zamanifar, Nematbakhsh, & Sadjady, 2010) have shown that in a multicore based system, the probability of having one (or more) core being idle while other cores having a long line of waited threads is very high. This will definitely lead to load imbalance which ultimately causes poor efficiency. As a solution to the above problem, research studies have been directed to develop scheduling algorithms that aim to achieve workload balancing between the cores to reduce the chances of having some cores without working to the minimum as much as possible (Breshears, 2009; Ding, et al., 2012; Mattheis, et al., 2012).

There are several classifications of workload balancing algorithms; each one is built on a particular aspect. An early classification belongs to Casey (Casey, 1981). He gives the basis of a hierarchical classification of load balancing algorithms in distributed systems. Since then, many other algorithms have emerged with different features since the classification in (Casey, 1981) has been considered insufficient. (Wang & Morris, 1985) suggest a taxonomy of load balancing algorithms, yet they restrict their study with Load-sharing algorithms. In Load-sharing, processes never migrate when they are initiated (Cheung & Jacobsen, 2006). Therefore, load-sharing is considered a subset of load balancing. As a result, the classifications discussed by Wang et al. describe only a sub group within load balancing algorithms. (Casavant & Kuhl, 1988) focus on the scheduling problem in general-purpose concurrent systems. They built their taxonomy on the work of Casey and Wang. The taxonomy of Casavant gives more details which are essential as it allows comparisons to be made between different approaches. Although the work of Casavant has been considered a landmark in classifying scheduling problems, nevertheless, the researchers are particularly dependent on the management and allocation of system resources in building their hierarchical taxonomy. Classification by Diekmann et al. (Diekmann, Monien, & Preis, 1997) was dedicated for distributed and parallel systems. They depend on applications' characteristics in classifying different load balancing problems. In order to achieve their goals, they introduced a model that describes the relation between the application and the computer architecture. However, they restricted the method of exchanging data only with message passing based systems.

2.2 The Emergence of Work-Sharing and Work-Stealing Scheduling

Multithreading scheduling has been classified into two main categories: work-sharing and work-stealing. "In work-sharing, whenever a processor generates new threads, the scheduler attempts to migrate some of them to other processors in an attempt of distributing the work to under-utilized processors. In work-stealing, however, under-utilized processors take the initiative: they attempt to "steal" threads from other processors" (Hendler, Lev, Moir, & Shavit, 2005). Intuitively, the migration of threads occurs less frequently with work-stealing than with work-sharing, since when all processors have work to do, no threads are migrated by a work-stealing scheduler, but threads are always migrated by a work-sharing scheduler (R.D. Blumofe & Leiserson, 1999). Work-stealing has been proven to be a more effective means of balancing loads than work-sharing in sharing memory systems, especially in terms of communication efficiency: when all processors are busy, no attempts are made to migrate work across processors. Work-stealing has therefore been a popular strategy for multithreaded computations (U. A. Acar, Charguéraud, & Rainey, 2013; Belal; Berenbrink, Friedetzky, & Goldberg, 2001; Hendrickson & Devine, 2000; Osman & Ammar, 2002). Regardless of the type of category, all scheduling algorithms target at a common main goal: they aim to make all the processors busy as much as possible. In other words, the real goal behind these two multithreading scheduling methods is to raise the level of concurrency between the processors which ultimately leads to better achievement.

2.2.1 Work-Sharing Scheduling

During the 80s of the last century, the design of multiprocessor systems included a common global memory organized as a queue where every processor in the system deals with it. The mechanism was simple; each idle processor pops a thread from the global memory and executes it. If the processor could not complete its thread execution within a specific period of time, then the same processor pushes the thread back to the end of the global memory. This type of mechanism gained the name work-sharing since all processors share this global data structure which is employed to maintain system threads (R.D. Blumofe & Leiserson, 1999).

Raetz (Raetz, 1987) has shown that due to the simplicity in computer architecture at that time, the global memory scheme was sufficient to multiprocessor systems. However, as computers become more advanced, the global memory principle has become ineffective. Feitelson et al.(Feitelson & Rudolph, 1995) have mentioned several reasons for this problem. First, it is quite possible to have more than one processor in the idle situation trying concurrently to access the global memory. The system has no choice other than serializing the processors' requests. As a result, with the increase in the number of processors, bottleneck arises. Second, when an incomplete thread is sent back to the global memory, there will be a little chance to reschedule this thread to the original processor; consequently this will cause losing valuable data such as the thread's state, temporary values, etc. Third, dealing with a global memory will create a big problem with applications that have a high rate of interacting and synchronizing between their threads.

In 1991, Rudolph et al. proposed an alteration to the principle of work-sharing from a global memory sharing to a local memory sharing (Rudolph, et al., 1991). Their work is built on two bases; first, they emphasize the importance of processors' local memories rather than on a single global memory; second, they proposed a new load balance mechanism. Usually, each processor is accompanied with a local memory. The authors suggest that every processor's local memory should keep its threads even when a thread's execution time exceeds the time period assigned to it. New generated threads also should be kept inside their local memories. In other words, processors should not easily give up their threads. As a result, the migration of threads between processors decreases. However, this also leads to imbalance between workload among the processors since some processors may complete their jobs prior to other processors, accordingly they become idle while the rest of the processors are still busy. What makes matters worse is the possibility to repeat this in an ongoing basis. For the purpose of going out of this impasse, Rudolph et al.(Rudolph, et al., 1991) suggested that any processor periodically checks the number of its threads. Then, any processor may launch the load balance operation with other processors if the probability of this operation is directly proportional to the inverse number of threads in that processor, i.e. the probability is $1/Th_{i,}$ where Th_i represents the number of threads in the processor i. Therefore, a heavily loaded processor will rarely share its threads while the opposite is true with a light loaded processor. Next, the processor that is ready to share its threads randomly searches for another processor to migrate some of its threads.

The approach of Rudolph et al. is simple, distributed and also adaptive; however, it suffers from a major weakness, i.e. the lack of dynamicity in initiating a load balance process. This is because any processor p_i starts looking randomly for an idle processor p_j immediately after the number of threads in processor p_i exceeds a fixed threshold value. This means that the process of searching for idle processors starts even if the system is balanced; as a result, the system loses precious time and effort, though the impact of this problem can be alleviated through dynamically adjusting the threshold value.

2.2.2 Work-Stealing Scheduling

The early use of work-stealing principle goes back to the 80s of the 20th century. Burton et al.(Burton & Sleep, 1981) developed this principle of work to improve the speed of their parallel project which is dedicated for functional programs. The ZAPP (Zero Assignment Parallel Processor) project has been built on the idea of allowing adjacent processors to steal tasks from each other for the sake of providing better work diffusion. Few years later, Halstead et al. (Halstead Jr, 1984) implemented work-stealing on MultiLisp through using SMP computer. The work has been dedicated primarily to improve locality in multiprocessor systems. The authors claim that it would be better to steal oldest tasks rather than newest tasks since the latter may be loaded with heavy computations such as being a root of substantial tree of computations. This is because Halstead et al. followed Fork-Join technique. During the Fork operation, new recursively created tasks are added to the processor’s queue. Therefore, any added task is less attractive to theft by a thief processor since it will have less computation. In other words, the oldest task can achieve the highest probability to make the thief processor busy to the maximum extent possible (Rainey, 2010; Tzannes, 2012).

Squillante et al. (Squillante & Nelson, 1991) studied shared-memory in multiprocessor systems. The researchers have concluded that it is better to schedule a thread on the same processor rather than allowing threads to move around processors. In other words, the affinity of a thread for a particular processor can highly improve system performance. This matter gets more attention in their next contribution (Squillante & Lazowska, 1993). In addition to that, the authors in (Squillante & Nelson, 1991) deal with the problem of idle processors through what they call threshold scheduling policies. As in the work of Rudolph et al.(Rudolph, et al., 1991) , Squillante et al.(Squillante & Nelson, 1991) suggest that each idle processor should randomly check non-idle processors in order to pick one of them. If a certain picked processor has a threshold number of threads, then one of its threads could be migrated to the idle processor. The major difference between the work of Rudolph et al.(Rudolph, et al., 1991) and Squillante et al.(Squillante & Nelson, 1991) is that in the former, a non-idle processor donates some of its threads to be accessed by idle processors, while in the later, idle processors randomly search for non-idle processors to migrate some threads in order to process them.

Karp et al. (Karp & Zhang, 1993) developed several methods to process both back-track search and branch-and-bound computations in parallel on a message-passing multiprocessor system without using a global data structure. The authors applied work-stealing in a way that makes it a donation rather than a stealing. The idle processor randomly selects one of the busy processors and sends a request to it. Following that, the selected processor receives the request and sends some of its work to the idle processor. Blumofe (Robert D Blumofe, 1995) although argued about the effectiveness of these methods in solving searching problems, they lack space requirements and communication costs.

There are several general observations on the works (Burton & Sleep, 1981; Halstead Jr, 1984; Karp & Zhang, 1993; Squillante & Nelson, 1991). First, these works are basically not built on the basis of work-stealing. In fact, work-stealing has been used as a complementary part to their original works. Second, restricting the theft from only the neighbour processor may be a waste of time since there is a chance of being the neighbour itself in an idle situation. Third, even when selecting non-neighbour processors for stealing, random selection does not guarantee the best choice. After all, selecting a processor with few threads is considered a bad choice when, at the same time, we have several wealthy processors. Finally, stealing a single thread might be worth in certain applications but it certainly does not worth in all applications. In addition, there is no clear indication on how many threads should be stolen and why.

2.2.3 Work-Sharing versus Work-Stealing

There are several studies that inspect both work-sharing and work-stealing. These studies made a trade-off between the scheduling techniques and concluded that work-stealing is preferable to work-sharing. Blumofe et al.(R.D. Blumofe & Leiserson, 1999) stressed the importance of the number of threads migration between the two techniques. They preferred work-stealing since threads migration happens less compared with work-sharing. They argued that in work-sharing (Rudolph, et al., 1991), processors seek to balance the system even when the system is already balanced or semi-balanced; while in work-stealing, the balancing process is only initiated when there is a need. For the same reason, Dinan et al.(Dinan et al., 2008) followed Blumofe et al. (R.D. Blumofe & Leiserson, 1999) in their opinion. They relied on the stability measurement to differentiate between the two techniques. They found that work-sharing suffers from a high percentage of load balance messages being circulated among the processors even when the system is balanced, the thing that does not happen in work-stealing. As a result, Dinan et al. (Dinan, et al., 2008) concluded that work-sharing is unstable while work-stealing is a stable technique. Chen et al. (Chen, Guo, & Huang, 2012) discussed the subject of lock contention between the processors. They preferred work-stealing since there is a low percentage of lock contention even during the stealing process. On the other hand, work-sharing with a global memory suffers from a high percentage of lock contention because any processor needs to lock the global memory for both adding and removing threads. Guo et al.(Guo, Barik, Raman, & Sarkar, 2009) concluded that work-sharing with a single shared memory unavoidably faces a future problem represented by scalability bottleneck because the number of processors is on a continuous increase.

On the other hand, Eager et al. (Eager, Lazowska, & Zahorjan, 1986) preferred work-sharing rather than work-stealing in a distributed system environment. The simulation results have shown that work-sharing with system load ranging from low to moderate makes better progress. However, work-stealing progress is better when dealing with high loads, assuming that the cost of load transferring is similar between the two strategies. Despite the fact, in distributed systems, the cost will be higher in work-stealing rather than work-sharing. The reason for this is due to the policy of work-stealing in transferring loads that have already been started; but in the case of work-sharing, the opposite happens, loads are transferred before they are executed. This argument is true for distributed systems; however, it does not apply for sharing memory systems where there is no difficulty in transferring loads that have already been started execution.

2.3 Significant Achievements of Work-Stealing

Since the beginning of the use of work-stealing technique, a large number of research studies have been conducted. These researches vary in importance and influence in the development of the use of the technique. However, certain researches considered it a landmark in the world of work-stealing which requires full attention. In this section, some of these researches are discussed:

2.3.1 Scheduling Fully-Strict Multithreaded Computations

The work of Blumofe et al.(R.D. Blumofe & Leiserson, 1999) is considered one of the distinguished achievements in work-stealing scheduling. They presented the first work-stealing scheduling algorithm which features the ability to schedule fully-strict (well-structured) multithreaded computations. Their work includes detailed analysis of the time and space complexity for scheduling multithreaded computations. According to their analysis, a P-fold speedup can be achieved during the execution of the parallel part of an application running on a P-processor environment using at most P times more space than when running on a single processor. In their approach, each processor is accompanied with a memory organized as a deque. The processor uses the top of its deque for two purposes: First, it pops the threads from the top in order to process them. Second, the processor uses the top side to enqueue newly generated threads. The bottom of the deque is dedicated for stealing. An idle (thief) processor randomly searches for the first encountered non-idle processor (victim) to steal threads from its bottom. Therefore, any processor is either working on its queue of threads or attempts to steal threads from other processors' queues. For any processor says X , the algorithm of Blumofe et al.(R.D. Blumofe & Leiserson, 1999) can be listed as follows:

```
Loop:
 While processor X’s deque is not empty Do
   Thread A ← Processor X pops the bottommost thread from its deque
   Processor X executes Thread A until one of the following actions happens:
   If Thread A completed its mission (Dies) Then
       Processor X removes Thread A
   Else
       If Thread A stalls (for example waiting for other threads to be completed) Then
           Processor X removes Thread A
       Else
           If Thread A enables a stalled Thread say C Then
               Thread C is placed on the bottommost position of the deque
           Else
               Thread A generates another Thread say B
               Processor X pushes Thread A into the deque and starts executing Thread B
   End While
   If Processor X’s deque is empty and there are still victim processors Then
       Processor X becomes a thief and it randomly searches for a victim processor and
       steals a thread located at the topmost position of the victim’s deque. The stolen
       thread is pushed into Processor X’s deque. Go to Loop
   Else
       Stop
```

The results of Blumofe et al.(R.D. Blumofe & Leiserson, 1999) were good especially when dealing with areas that need static partition. However, the quality of the results is not the same when the algorithm is applied in the modern environments. In other words, the algorithm does not work in multi-programmed environments that are supported by modern shared-memory multiprocessors and operating systems. This is due to the algorithm’s designed mechanism that deals with a fixed set of processors with the assumption of the full availability of these processors. In addition, a lot of failed attempts of theft may happen because multiple thief processors try to steal from one victim core (Cao, Sun, Qian, & Wu, 2011). This brings us to the importance of predetermination of the victim cores instead of wasting time in useless attempts. The researcher has addressed this point in this study through locating the victim and thief cores prior to any stealing process. In this way, no failed attempts of theft have ever happened in this study.

2.3.2 Non-Block Work-Stealing Algorithm

Arora et al. (Arora, Blumofe, & Plaxton, 2001) improved the work of Blumofe et al. (R.D. Blumofe & Leiserson, 1999) to produce a non-blocking work-stealing algorithm. In a non-blocking system, any delay in any process will not hinder other processes from making progress. In other words, contention can be prevented during concurrent operations. The new algorithm soon became the favourite choice both in the academic and industrial fields (Hendler & Shavit, 2002). The improvement lies in two points: First, the algorithm can deal with arbitrary multithreaded computations instead of restricting the computations with the fully-strict type only. Second, their algorithm can manage a multi-programmed environment in contrast to (R.D. Blumofe & Leiserson, 1999) which was unable to deal with such environment. In other words, the algorithm of Arora et al. has the ability to deal with more than one program at the same time where each program may utilize a different number of processors. To achieve such goals, the authors did not map threads to processors directly as in (R.D. Blumofe & Leiserson, 1999), instead they planned for two schedulers. The first scheduler maps the threads into P processes, while the second (complementary) scheduler maps the processes into the processors. Therefore, the P-fold speedup cannot be achieved all the time because the second scheduler may manage its work with less than P processors. In addition, Arora et al. (Arora, et al., 2001) improved the behaviour of the deque which represents the backbone of the work-stealing algorithm as in (R.D. Blumofe & Leiserson, 1999). Here the process rather than the processor as in (R.D. Blumofe & Leiserson, 1999) is in charge of managing its threads. Additionally, deques have the advantage to become a non-blocking data structure that can handle concurrent operations (U. A. Acar, et al., 2013). A CAS (compare-and-swap) instruction ("Compare-and-swap,") that stands behind the success of the concurrent updating of the deques is an atomic instruction used in multithreading to achieve synchronization. The

CAS' operands are a given value and a given memory location. The instruction compares the given value with the content of memory address by the given location. If they are the same, the CAS modifies the contents of that memory location to a given new value. This is done as a single atomic operation. The atomicity guarantees that the new value is calculated based on up-to-date information; if the value had been updated by another thread in the meantime, the write would fail (Arora, et al., 2001).

However, the use of the CAS instruction is only needed when there is only one thread in the deque. The deques are much like the one described in the work of Knuth (Knuth, 1968). However, in a work-stealing mechanism, processes can access only one end of the deques that is the "Bottom" while other processes can access the other end, i.e. the "Top" during the stealing operation.

Unfortunately, the algorithm of Arora et al. encountered several problems. First, due to the use of fixed sized arrays, the algorithm can deal with only m/n threads inside a deque where m represents the total memory size and n represents the number of processes (Hendler, et al., 2005). The second problem is related to memory management. Overflows can easily occur due to the use of fixed-size array (array of pointers) in representing a deque. This drawback especially happens when running several programs for which the authors designed their algorithm. Arora et al. tried to reduce the effects of overflows by using cyclic array technique. However, they succeeded in reducing the chances of overflow but could not avoid overflow from happening. Consequently, the continuous adjustments of the deque sizes are necessary during runtime since it cannot predict the size of each thread. There is no simple operation to free memory locations and return them to the free space (Hendler, et al., 2005).

Using a parallel garbage collector will definitely require precious time to accomplish (Hendler, et al., 2005). The second problem is associated with the number of stolen threads. Berenbrink et al.(Berenbrink, et al., 2001) criticized work-stealing systems that steal a single item at a time. The researchers use Markov model to analyze work-stealing technique. The authors use Markov model in arguing that a work-stealing system that is built on the basis of single-stealing at a time could end up with an unstable state (overflow) which becomes difficult to recover. In addition, the authors stress that even when extra spaces are allocated, at some point, overflow may occur.

2.3.3 Improving the Non-Block Work-Stealing Algorithm

The work of Arora et al. (Arora, et al., 2001) represents a significant achievement in the work-stealing techniques, though the work suffers from several shortages. Several studies have been conducted to improve this work while retaining the essence of the work at the same time.

2.3.3.1 Stealing the Half

The mechanisms of work-stealing of Blumofe et al.(R.D. Blumofe & Leiserson, 1999) and Arora et al.(Arora, et al., 2001) have been designed to steal a single item at a time. Several researchers argue that stealing more than one item at a time increases the stability and achieves a better system load balance. Mitzenmacher (Mitzenmacher, 1998) analyzed work-stealing algorithms using differential equations. He came to the conclusion that multi-stealing can improve the performance of an algorithm. Berenbrink et al.(Berenbrink, et al., 2001) claimed that slipping into unstable state for Arora-like algorithms can be avoided when the algorithm is modified to steal half the deque content instead of stealing a single item. Hendler et al. (Hendler & Shavit, 2002) applied the idea of stealing the half of the victim’s deque. The authors followed the Arora algorithm’s features such as non-blocking and minimizing of using the CAS instruction. The main drawback in the approach of Hendler et al. is the dependency in using fixed-size deques. It was not clear whether it was possible to utilize resizable deques.

2.3.3.2 Data Locality

Acar et al.(U. Acar, Blelloch, & Blumofe, 2000) studied the data locality of work-stealing algorithms. The authors found that randomized stealing may lead to cache unfriendliness; therefore they suggested extending the work of Arora et al. in a way that makes stealing happen in a locality-guided way. Here, the process gives priority to the threads that have affinity to it. If no thread has affinity to the process, then the process follows the work-stealing mechanism in selecting another victim process randomly. The authors claimed that their modified algorithm outperforms the standard algorithm of work-stealing.

2.3.3.3 Dynamic Deques

The work of Arora et al. is based on using fixed-size arrays. Practical experiments proved that the use of such data structure leads inevitably to overflow. This means that the deques’ sizes must be continuously adjusted to accommodate the unpredictable number of threads which dynamically change during the execution. To address this problem, Hendler et al. (Hendler, et al., 2005) suggested using dynamic structure instead of fixed-size arrays. The authors’ main contribution lies in implementing a deque as a doubly linked list, where each list is a short array which is dynamically allocated and freed. The authors have succeeded in dealing with the overflow problem; on the other hand, there was an increase in the complexity of the algorithm. Due to the extra work needed to maintain the dynamic list, the new algorithm shows a trade-off between time and space complexity.

2.3.3.4 Dynamic Circular Deques

Chase et al.(Chase & Lev, 2005) introduced the idea of dynamic circular array in implementing deques. The authors managed in eliminating the overflow problem with a simple and efficient algorithm. In addition, space complexity is linear, that is no memory is wasted as in Hendler et al. (Hendler, et al., 2005) besides no garbage collector is needed. However, when the deque becomes full, a new array is created and the elements are copied to the new deque. Although the process of copying is linear, this can be the only factor of delay.

2.4 More Work-Stealing Contributions

Agrawal et al. (Agrawal, Leiserson, He, & Hsu, 2008) presented an adaptive thread scheduler, called A-STEAL. They argued that their scheduler performs better than (Arora, et al., 2001) when the machine has a large number of cores and many jobs running on it. Vrba et al. have analysed the performance of applications running under graph-partitioning and work-stealing schedulers (Vrba, Espeland, Halvorsen, & Griwodz, 2009). Work-stealing has been formally proven to be optimal only for the restricted class of fully-strict computations. Recently, Ding et al. presented in (Ding, et al., 2012) a work-stealing scheduler for time-sharing multicore systems. Their scheduler has been designed to deal with two important drawbacks in the work of Arora et al, significant unfairness and degraded throughput. The scheduler improves average system throughput and reduces average unfairness.

2.5 Work-Stealing in Software Industry

The success of work-stealing in the academic research field has motivated software companies to adopt this principle of work in their products. Several languages and libraries have been developed based on the idea of work-stealing. Examples of these products are Cilk, TBB, TPL and Java.

2.5.1 Cilk

Cilk (pronounced “silk”) is a C-based multithreaded language for parallel programming. It adds several constructs to the original C language in order to deal with parallel control (Frigo, Leiserson, & Randall, 1998). The *spawn* construct creates a new thread that may execute concurrently with the threads’ parent. The *sync* construct has the duty of synchronizing a thread with its children’s threads. In other words, the *sync* blocks the execution of a function until all its spawned children complete their actions. The theoretical work of the Cilk language can be traced back to the work of (Robert D Blumofe, 1995) . The first version of this language (Cilk-1) was born at MIT in 1994. Parallelism has been represented with the first version; the language introduces an efficient scheduler based on work-stealing technique. However, the first version was awkward since parallelism was exposed “by hand” using explicit continuation passing (Frigo, et al., 1998). Cilk Arts Inc developed the commercial version of Cilk called Cilk++, which supports both C and C++. In July 2009, Intel Corp acquired the complete software. Currently, Intel® Cilk™ Plus is working on version 1.2.

2.5.2 Threading Building Blocks (TBB)

Intel TBB is a C++ template library that is designed for desktop shared memory computers. Since it does not represent a new language or even a language extension, TBB has been included in the existing C++ compilers without doing any modification to these compilers. The components of TBB are built at various levels of abstraction. In TBB, we can distinguish three levels of abstractions. At the highest level of abstraction, we can find concurrent containers and parallel algorithms. Threads scheduler is based on work-stealing similar to Cilk and it is located at the middle level of abstraction, while timing facility, atomic operation and mutexes are positioned at the lowest level of abstraction (Robison, Voss, & Kukanov, 2008). The TBB has managed to spare the programmer from getting into the complexities of using native threads packages such as Windows and POSIX threads where the processes of threads creation, synchronization and termination are done manually. Intel, the owner of TBB, argues that their product has proved its efficiency in dealing with the multicore environment. Any variation in the number of cores can easily be detected, the TBB can easily do the necessary adjustment to deal with the new number of cores ("Threading Building Blocks,"). First version of TBB has been introduced by Intel in August 2006 while the latest version is 4.2 was introduced in September 2013 (Wikipedia, 2014b).

2.5.3 Task Parallel Library

Microsoft's Task Parallel Library (TPL) works under .NET programming framework. Microsoft aims behind designing this library to simplify parallel programming in.NET environment (Lu & Adviser-Gannon, 2009). The library is responsible for threads creation and termination in addition to fitting the number of threads with the number of available processors (Wikipedia, 2013). As is the case in Cilk and TBB, TPL adopted work-stealing mechanism as the principle of work in its scheduler. Although there are several similarities between TBB and TPL, however, programming using TPL is easier since it provides the usage of .NET language supports (Olivier & Adviser-Prins, 2012).

2.5.4 Implementing Work-Stealing in Java

The principle of work-stealing has also been implemented in Java programming language in different ways. JAWS (Java Work Stealer) has been presented in (Mao, So, & Woo, 1998). It allows programmers to write parallel programs in pure Java that can run on a network of workstations. JAWS has been implemented as a user-level Java library which schedules user threads using a work-stealing technique. JAWS is strongly influenced by Cilk, however, there are two major differences between the two software: First, JAWS has been designed to deal with a network of workstations where there is no shared memory, while Cilk designers planned to implement their software in a symmetric multiprocessor machine which has a shared memory. Second, Cilk adds extension to the C language and it has the ability to access the C language faculties such as the stack of running threads. On the other hand, JAWS depends on Java Virtual Machine, there is no way to deal with stack frame as Cilk did. JAWS has a main drawback in its performance especially when dealing with a large-scale cluster, this is because any node in the cluster may steal several times from other nodes. This will absolutely lead to an increase in nodes' idle time besides causing a heavy overhead in network traffic (B.-Y. Zhang, Mo, Yang, & Zheng, 2007). In general, JAWS could not achieve optimal performance. The stealing and synchronizations processes are the main overheads that face this software. Satin represents a system for running programs on grid platforms (Van Nieuwpoort, Kielmann, & Bal, 2000). The programming model of Satin has been inspired by Cilk. Satin extends Java with two simple primitives for D&C programming. Originally, Satin has been presented by Nieuwpoort et al. in (Van Nieuwpoort, Kielmann, & Bal, 2001). The authors claim that the software which has not yet been applied in real grid shows an efficient load balance implementation based on work-stealing. In (Nieuwpoort, Maassen, Kielmann, & Bal, 2001), Nieuwpoort et al. evaluated Satin on a real grid. The authors argued that an efficient utilization of the resources has been achieved. However, they did not depend on the original work-stealing algorithm in their work. They extended the original algorithm to a Cluster Random Stealing algorithm which outperforms the original one. This new extended algorithm is specially designed for cluster-based wide area computing. Jcluster (B.-Y. Zhang, et al., 2007; B. Y. Zhang, Yang, & Zheng, 2006) is another Java based system that provides a parallel environment which is suitable for a large-scale heterogeneous cluster. It implements a task scheduler based on a Transitive Random Stealing algorithm. The proposed scheduler can be seen as an improvement to work-stealing algorithm. The authors in (B.-Y. Zhang, et al., 2007) argued that the mentioned scheduler outperforms the work-stealing scheduler in reducing processors' idle time and network communication overheads. Java language (Lea, 2005) has developed its java.util.concurrent packages in its 7th release by adding a framework for fork-join style parallel decomposition. The new framework provides a natural means for partitioning many algorithms to efficiently make use of hardware parallelism. The use of work-stealing has reduced the contention for the working deques.

It is noted the growing importance of these languages and libraries. However, there is a common drawback which these software shared (Gautier, Lima, Maillard, & Raffin, 2013). This drawback is represented by the synchronization points of these software. These points force some tasks to be completed before allowing new tasks to be executed. For example, concurrent deques that are utilized in these languages and libraries required expensive memory-fences which can affect the overall performance. Frigo et al. have shown in (Frigo, et al., 1998) that half of Cilk work-stealing schedulers are spent in executing memory fence.

2.6 Drawbacks of Work-Stealing

Despite the great success of work-stealing that has been achieved, the scheduling algorithm suffers from several drawbacks. Nogueira et al. have shown in (L. Nogueira, Fonseca, Maia, & Pinho, 2012) that the scheduler mechanism of Blumofe et al. (R.D. Blumofe & Leiserson, 1999) is fast and easy, however, they argued that the random approach employed in choosing the victim core cannot always determine the best victim core. To make matters worse, the current trend and the future, at least for the foreseeable extent to processor manufacturers is the increase in the number of cores. This will certainly lead to an increase in the number of victim cores, as a result, the probability of choosing the best victim core will decrease.

Moreover, there will be a considerable waste of time. A thief core that fails in stealing from a victim core due to competition, causes the loss of system resources, in addition to repeating this action with other victim cores, for example, two or more cores trying to steal from the same victim core which itself is a thread-poor core. This leads the researcher to emphasize the importance of the right choice of victims. In fact, if the scheduler failed in tailoring unsuccessful stealing attempts, this without doubt leads to slowing down application execution as much as 15-35% as mentioned in (R.D. Blumofe & Leiserson, 1999).

Neill et al. discussed the cost of stealing in (Neill & Wierman, 2009). The authors argued that this cost comes from system bus contention and threads' transfer latency. Another source of cost the authors highlighted is related to queues' affinity. According to the work-stealing principle of work, any process has the right to steal from any other queue, that is, from any non-local memory. When we put this matter into consideration besides having the hardware fact which says it is much faster for a processor to access its local-memory than accessing other processors' local-memories. The authors concluded that stealing from other queues loses the advantage of local cached computations. However, without stealing, there will be load balance. Therefore, it is important to keep threads in their queues as much as possible for the sake of affinity. Only when there is a real need to distribute them, then threads can be taken away from their local-queues.

2.7 Summary

This chapter started by giving a quick review of the papers that highlight the importance of scheduling algorithms on multicore systems. Then, the researcher moved towards reviewing the papers that cover the classification of workload balancing algorithms. Next, the range of papers was narrowed down to be dedicated to the general review of Work-Sharing and Work-Stealing Scheduling algorithms. The reviewing process has been redirected to focus only on Work-Stealing Scheduling algorithms. The researcher started by reviewing the early contributions in work-stealing and how this technique evolves during the eighties and nineties of the twentieth century. Then, the researcher gave more attention to the contribution of Blumofe et al. and Arora et al. Following that, the researcher reviewed the papers that discuss weaknesses and gaps in these two papers and the proposed solutions that have been submitted by other researchers. Next, the researcher covered the papers that criticise the drawbacks in work-stealing principle of work. Then, the researcher reviewed the principle of work-stealing in software industry. Finally, the researcher reviewed some of the recent papers that highlight the drawback in work-stealing. Table 2.1 gives a summary of the most related work in this study.

Table 2.1: A Summary of the most related work in this study

Title	Authors - Date	Contribution
Executing functional programs on a virtual tree of processors	(Burton & Sleep, 1981)	This work represents the origin of work-stealing principle of work. The authors try in their project to allow processors to steal from each other for the sake of providing better work diffusion.
Implementation of Multilisp: Lisp on a multiprocessor	(Halstead Jr, 1984)	The authors implemented work-stealing on MultiLisp using SMP computer. Their work was primarily directed to improve locality in multiprocessor systems. The authors claim that it would be better to steal oldest tasks rather than newest tasks since the latter may be loaded with heavy computations such as being a root of substantial tree of computations.
Analysis of task migration in shared-memory multiprocessor scheduling	(Squillante & Nelson, 1991)	The authors studied shared-memory in multiprocessor systems. They suggested that each idle processor should randomly check non-idle processors in order to pick one of them. If a certain picked processor has a threshold number of threads, then one of its threads could be migrated to the idle processor.
Randomized parallel algorithms for backtrack search and branch-and-bound computation	(Karp & Zhang, 1993)	The authors applied work-stealing in their algorithm in a way that makes it a donation rather than a stealing. The idle processor randomly selects one of the busy processors and sends a request to it. Following that, the selected processor receives the request and sends some of its work to the idle processor.
Executing multithreaded programs efficiently	(Robert D Blumofe, 1995)	Argued that the work of (Burton & Sleep, 1981), (Halstead Jr, 1984), (Karp & Zhang, 1993),(Squillante & Nelson, 1991) lack space requirements and communication costs, in addition, their works did not build on the basis of work-stealing ; it is considered secondary to the importance of their work. Moreover, stealing from neighbouring processor happens which is considered waste of time in the case where the neighbour is idle as well.
Scheduling multithreaded computations by work stealing	(R.D. Blumofe & Leiserson, 1999)	A distinguished achievement in the work-stealing scheduling. They presented the first work-stealing scheduling algorithm which is able to schedule fully-strict (well-structured) multithreaded computations. However, they could not succeed in implementing their algorithm in shared-memory systems that apply multiprogramming. This is due to the assumption that a fixed set of processors are fully available to perform a given computation.
Thread Scheduling for Multi-programmed Multiprocessors	(Arora, et al., 2001)	Improving the work of (R.D. Blumofe & Leiserson, 1999) through proposing a work-stealing scheduling algorithm which is able to do two things: First, the algorithm is able to schedule arbitrary multithreaded computations as opposed to the special case of “fully strict” as in the work of Blumofe et al. Second, the

		algorithm can deal with multiprogramming algorithm while the algorithm of Blumofe et al. is designed for a dedicated environment.
Non-blocking Steal-Half Work Queues	(Hendler & Shavit, 2002)	Point out to two main achievements in the work of (Arora, et al., 2001) : First, the algorithm of Arora et al., can deal with arbitrary multithreaded computations instead of restricting the computations with the fully-strict type only. Second, their algorithm can manage a multi-programmed environment in contrast to (R.D. Blumofe & Leiserson, 1999). Apply the idea of stealing the half of the victim's deque. The authors followed Arora algorithm's features such as non-blocking and minimizing of using the CAS instruction. The main drawback in the approach of Hendler et al. is the dependency in using fixed-size deques.
Scheduling parallel programs by work stealing with private deques	(U. A. Acar, et al., 2013)	Study the influence of private deques and identify one of the main points in the work of (Arora, et al., 2001), i. e., the use of non-blocking data structure as a deque to handle concurrent operations.
A Dynamic-Sized Non-blocking Work Stealing Deque	(Hendler, et al., 2005)	Discuss the work of (Arora, et al., 2001) and determine two main problems: First, the algorithm can deal with a limited number of threads due to the use of fixed-sized arrays. Second, a fixed-size array can itself cause an overflow problem. The authors suggest implementing a deque as a doubly linked list instead of using fixed-size arrays. In this way, they solved the main drawback in the work of (Arora, et al., 2001), as a result, the overflow problem has been eliminated but it came at the expense of an increase in the complexity of the algorithm and memory wastage.
The natural work-stealing algorithm is stable	(Berenbrink, et al., 2001)	Criticize the work of (R.D. Blumofe & Leiserson, 1999) (Arora, et al., 2001) for the reason of stealing a single item. Berenbrink et al., argue that a system with single stealing could end up with an unstable state (overflow) which becomes difficult to recover.
Analyses of load stealing models based on differential equations	(Mitzenmacher, 1998)	Analyzed the work-stealing algorithms using differential equations. He came to the conclusion that multi-stealing can be improved when stealing multiple items instead of one.
The natural work-stealing algorithm is stable	(Berenbrink, et al., 2001)	Claim that the work of (Arora, et al., 2001) may slip into unstable state . However, this state can be avoided when the stealing algorithm is modified to steal half of the deque instead of a single item.
The Data Locality of Work Stealing	(U. Acar, et al., 2000)	Suggest a method to improve the locality of work stealing. The authors found that randomized stealing may lead to cache unfriendliness; therefore they suggested extending the work of Arora et al. in a way that makes stealing happens in a locality-guided way.

Dynamic circular work-stealing deque	(Chase & Lev, 2005)	The authors managed in eliminating the overflow problem in (Arora, et al., 2001) with a simple and more efficient algorithm than in (Hendler, et al., 2005). They introduced the use of dynamic circular array in implementing deques; as a result, a garbage collector is no more needed. However, when the deque becomes full, there will be a need for extra time to transfer items from the old deque to the new deque.
Adaptive work-stealing with parallelism feedback	(Agrawal, et al., 2008)	They presented an adaptive thread scheduler, called A-STEAL. They argued that their scheduler performs better than (Arora, et al., 2001). Agrawal, et al. point out to the fact that Arora, et al. did not provide parallelism feedback in their work which leads to waste in processors' cycles.
On the benefits of work stealing in shared-memory multiprocessors	(Neill & Wierman, 2009)	They discussed the cost of stealing generated from system bus contention and threads' transfer latency which comes from leaving the cores steal from each other freely. The authors stressed on queues affinity and stealing should be allowed only when there is a real need for stealing.
Dynamic Global Scheduling of Parallel Real-Time Tasks	(L. Nogueira, et al., 2012)	The authors criticised the work-stealing random approach in choosing the victim core. This is due to the difficulty that may face a thief core in choosing a victim core because the competition between thief cores. For example, two thief cores try to steal from the same victim core. This will undoubtedly waste system resource and time.
BWS: Balanced Work Stealing for Time-Sharing Multicores	(Ding, et al., 2012)	They presented a work-stealing scheduler for time-sharing multicore systems. Their scheduler has been designed to deal with two important drawbacks in the work of Arora et al, significant unfairness and degraded throughput. The scheduler improves average system throughput and reduces average unfairness.
A work-stealing scheduling framework supporting fault tolerance	(Y. Wang, et al., 2013)	The authors proposed a work stealing scheduling framework that supports hardware fault tolerance. The framework is able to detect and recover both transient and permanent faults.
Friendly barriers: efficient work-stealing with return barriers	(Strang, 2011)	The authors address dynamic overheads that occur when a steal is taking place. They succeeded in reducing the dynamic overhead to half which results in improving the total performance.

CHAPTER 3: RESEARCH METHODOLOGY

3.1 Introduction

The multicore technology has seized control of the processors industry and becomes, without any doubts, the main player in any computer-based device. Processor manufacturers continuously try to replicate the number of cores per chip in order to achieve better results, i.e. executing the programs in less time. For instance, in the laptops world, it becomes natural to see the increase of the number of cores in the laptops. For the time being, commercial laptops have 4 to 8 cores. This also can be said for the mobile technology where the number of cores is on the rise. All the expectations for this technology are directed to a continuous increase in the number of cores. On the other hand, software industry has not developed sufficiently in order to fit the on-going development taking place in multicore technology.

This study is about building concurrent multithreaded models that solve D&C problems (Fibonacci Series, Towers of Hanoi, Binary Search, and Matrix Multiplication) on a multicore environment. To achieve this goal, the researcher proposes two types of schedulers: A Low Level Scheduler (LLS) and a High Level Scheduler (HLS). As shown in Figure 3.1, the LLS is included in each modelled core while there is only one HLS in the entire model. The D&C workload problem is represented as threads proposed by the researcher. Initially, every problem starts by a single main thread located in one of the cores. The duty of each LLS is to partition and manipulate the threads inside its core. On the other hand, the HLS balances the workload (partitioned threads) among the entire cores through pulling (at run time) some threads from the victim cores and adding these threads to the thieves' cores. The entire LLSs work at the same time with the HLS for the purpose of making the whole cores busy as much as possible so that a high level of concurrency can be achieved. The HLS has a guard which enables/disables the activation of the HLS. On the other hand, each core has its own guard that enables/disables the activation of the core itself.

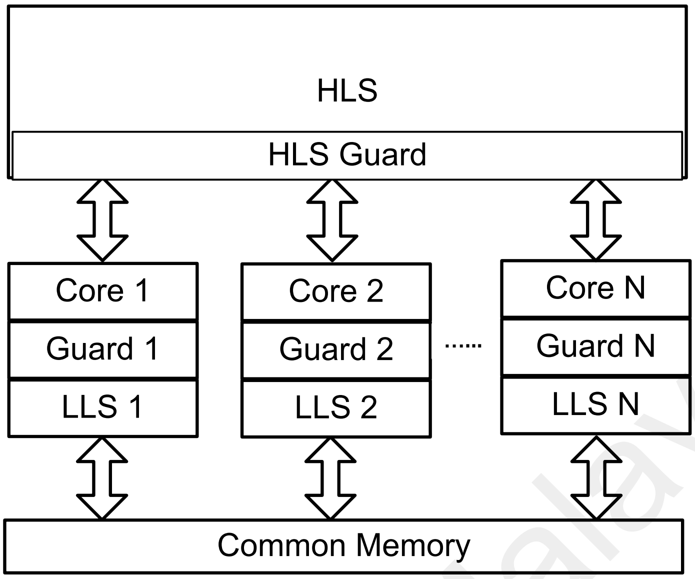


Figure 3.1: A multithreaded multicore model. The LLSs create the threads and manipulate them while the HLS redistributes the threads between the cores. The common memory is a shared area used to store data and temporary results

The methodology used in this study has been built on five main phases. Phase 1 is dedicated for the research idea. Phase 2 introduces research background. Phase 3 is related with literature review conducted by the researcher. Phase 4 is dedicated for identifying research problem and objectives. Phase 5 illustrates design methodology, and finally, Phase 6 is about the simulation and monitoring processes. Figure 3.2 shows the workflow of the study.

3.2 Phase 1: Building the Research Idea

In any computer system, there are two main components: Hardware and Software. This covers all types of computer systems starting from the supercomputers towards the smallest computers. Studies have proven that best results can be achieved when there is a great harmony between these two components. For instance, in computer systems that apply parallel processing, these systems cannot attain high achievement without providing a high compatibility between the parallel processors and the software that operate these processors, i.e. providing operating systems and other software that can deal with such an environment.

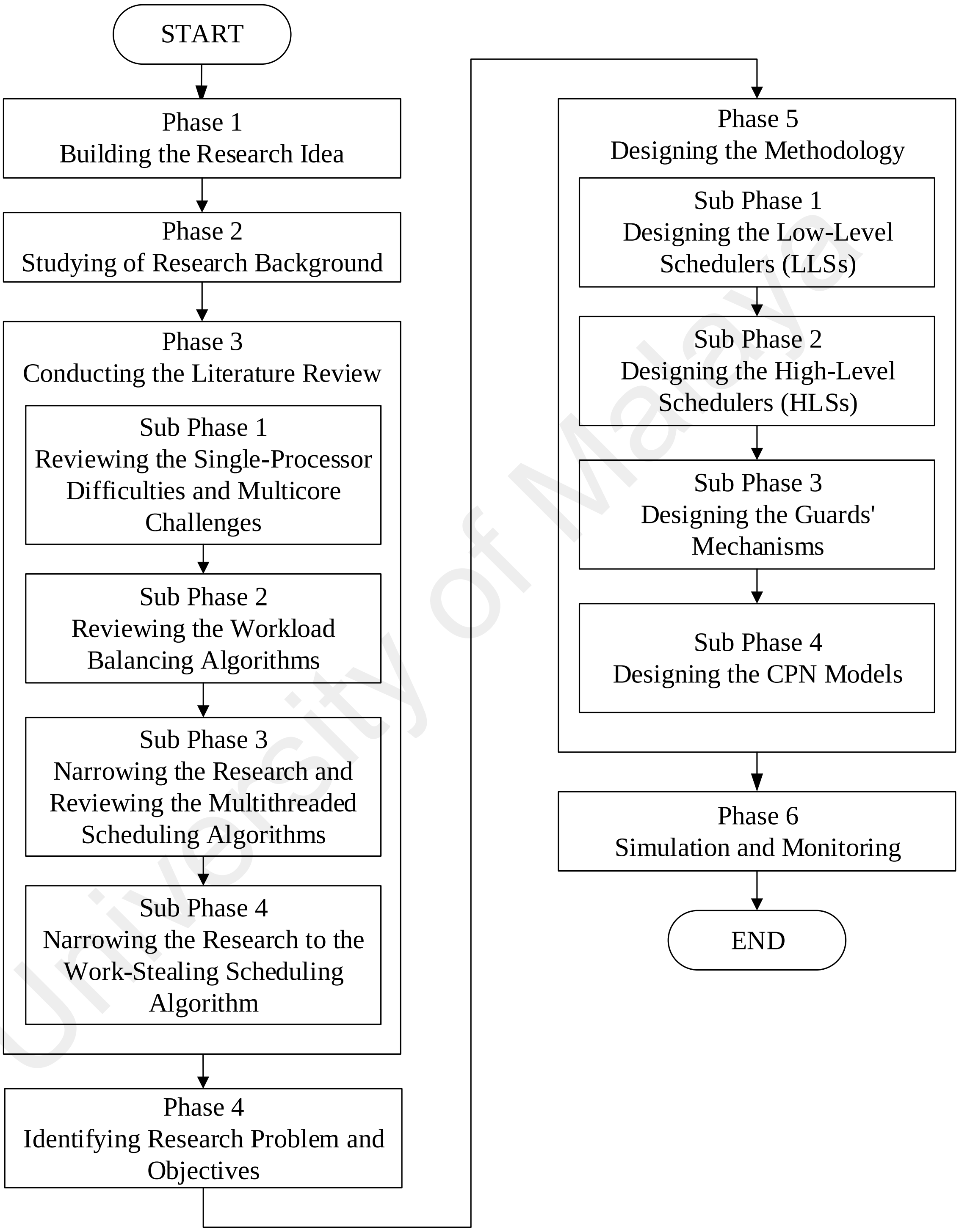

START
Phase 1
Building the Research Idea
Phase 2
Studying of Research Background
Phase 3
Conducting the Literature Review
Sub Phase 1
Reviewing the Single-Processor Difficulties and Multicore Challenges
Sub Phase 2
Reviewing the Workload Balancing Algorithms
Sub Phase 3
Narrowing the Research and Reviewing the Multithreaded Scheduling Algorithms
Sub Phase 4
Narrowing the Research to the Work-Stealing Scheduling Algorithm
Phase 4
Identifying Research Problem and Objectives
Phase 5
Designing the Methodology
Sub Phase 1
Designing the Low-Level Schedulers (LLSs)
Sub Phase 2
Designing the High-Level Schedulers (HLSs)
Sub Phase 3
Designing the Guards' Mechanisms
Sub Phase 4
Designing the CPN Models
Phase 6
Simulation and Monitoring
END

Figure 3.2 Workflow of the Study

In the case of personal computer systems, these systems became well known worldwide since 1981 when both IBM with Microsoft presented the first PC (Held, 1986). At that time, software were designed to deal with single-processor computers. However, with the beginning of the twenty first century, processor manufacturers adopted multicore technology instead of continuing to develop single-processor industry. Consequently, it was inevitable that software were influenced by this diversion in the hardware since most software were designed to deal with a single-processor environment. For that reason, the ability of software to deal with this expansion in the number of cores has become the focus of attention of researchers working in academic and industrial fields.

The researcher has noticed that the success that has been made in replicating the number of cores has not been matched with a similar success in the software side. The researcher touches on the difficulties faced by software developers in dealing with multicore technology. These difficulties are represented by adapting the software with the technologies of processors that have variable numbers of cores. From this point, the research idea has been generated. The research idea revolves around providing the mechanisms that help in making software more adaptable with the multicore technology. The researcher takes into consideration that these mechanisms should be adaptable to deal with variable number of cores; in addition, these mechanisms should operate the cores to the maximum extent, leaving a minimum number of idle cores. Given the large scope of research topic in this area, the researcher focuses on a class of problems, i.e. Divide and Conquer problems. Therefore, the idea of this research is built on designing mechanisms that manage Divide and Conquer workload problems on a multicore environment.

3.3 Phase 2: Studying of Research Background

In this sub phase, the researcher started by addressing the real reasons that prompted the development of multicore technology and how this is reflected on the processors manufactured. The influence of Moore’s Law and the improvement that is made on this law are discussed in this sub phase. Following that, the researcher gives some examples of the early processors that were built on the basis of multicore technology in the beginning of the twentieth century.

Then, the researcher redirects the attention to the software rather than the hardware. Here, the research touches on the difficulties faced by software developers in dealing with multicore technology. These difficulties are represented by adapting the software with the technologies of processors that have a variable numbers of cores. Subsequently, the researcher clarifies the terms: concurrency, scalability, parallelism, multithreading, working cores, and non-working cores. These terms are essential to the research and are important to be clearly explained. The researcher stresses on the concurrency and scalability that modern software should possess in order to exploit the new technology. The researcher also briefly highlights the similarity and difference between concurrency and parallelism because of their importance in the research. Then, the researcher clarifies the role of threads and multithreading in scheduling techniques. Later, two important terms, working (busy) core and non-working (idle) core, are defined. In addition, the researcher discussed the reason behind having these two types of cores.

Then, the research's background is redirected to give a glimpse of two main topics in this thesis: the scheduling techniques and Divide and Conquer problems. The researcher draws the attention to the types of scheduling (static and dynamic) and briefly mentions the differences between them. The researcher concludes the importance of the dynamic scheduling and its impacts on this study. Next, the researcher defines the problems of Divide and Conquer. In view of the fact that this thesis proposes CPN models that solve Divide and Conquer problems on multicore architecture, the researcher finds that it is necessary to give a quick look to this kind of problems in the background phase.

Finally, the researcher explains the importance of modelling in designing concurrent systems. The researcher draws the attention to the advantages of using modelling and how these advantages can be exploited in producing robust models. Then, the researcher introduces the modelling language (Colored Petri Nets) and the modelling tool (CPN-Tool).

3.4 Phase 3: Conducting the Literature Review

Based on the research background, the literature review went through three sub phases. Sub Phase 1 is related with the research background while Sub Phase 2 is to survey workload balancing algorithms. In Sub Phase 3, the researcher narrows down the research area and direct it to the multithreaded scheduling algorithms. Finally, in Sub Phase 4, the researcher focuses on the work-stealing scheduling algorithm.

3.4.1 Sub Phase 1: Reviewing the Single-Processor Difficulties and Multicore Challenges

The literature review chapter starts by discussing the difficulties faced by the processor manufacturers in developing faster single processors within the beginning of the twentieth century. The researcher clarified that for the purpose of continuous production of highly efficient processors, the processor manufacturers have developed the multicore technology with a purpose to try to get out of the single-processor predicament. Subsequently, the researcher indicates that, although the multicore technology is the savoir to the processor manufacturers, the new technology does not provide a perfect solution to the problem. The researcher examined several studies that addressed the problem of workload imbalance happening in multicore systems. The researcher concluded that the failure to achieve the desired results due to the lack of dynamic run-time scheduling algorithms that can balance the workload among the cores.

3.4.2 Sub Phase 2: Reviewing the Workload Balancing Algorithms

The research is then directed to workload balancing algorithms. The researcher starts by reviewing several classifications of these algorithms. The review includes a brief description of each classification which points out on what basis the classifier builds its taxonomy, in addition to mentioning the scope and limitation of the classification.

3.4.3 Sub Phase 3: Reviewing the Multithreaded Scheduling Algorithms

The researcher subsequently narrows the research scope and directs it towards multithreaded scheduling algorithms. The researcher reviews the studies on work-sharing algorithms, the algorithms' mechanisms, and then discusses the reasons behind the success that has been achieved in this type of algorithms during the 80s. Then, the researcher clarifies how this success abated when computers become more advanced and workload increases. To proceed, the researcher starts identifying the work-stealing scheduling algorithm, reviewing its early implementations and the advantages of this type of scheduling. Later, the researcher focuses on the defects in the work-sharing and how the work-stealing has succeeded in overcoming such defects. Finally, the researcher conducted a comparison between the two methods based on the studies in this area and concluded that work-sharing is preferred in distributed systems while work-steal are achieves better results in shared-memory systems.

3.4.4 Sub Phase 4: Reviewing the Work-Stealing Scheduling Algorithm

The researcher continues narrowing the scope of the thesis. The researcher reviews the major achievements in work-stealing scheduling techniques. In each achievement, the researcher tries to survey both the positive and negative aspects. After that, the researcher moves to review the most recent achievements in using work-stealing scheduling. Later, the researcher reviews the use of work-stealing in software industry. This includes adopting this technique in C, C++, and Java based software. Finally, the researcher discusses the drawbacks of work-stealing scheduling. Criticism was directed to the indiscriminate way in the selection of the victim core. This drawback has been addressed in this study and the researcher proposes a solution for it. Another drawback has been discussed which is related to the wasting in time when the thief core fails in stealing. This drawback has also been addressed in this thesis.

3.5 Phase 4: Identifying Research Problem and Objectives

Based on the research background and the literature review, in the next step, the researcher determines the research problem and the objectives of the study. Regarding the problem statement, the researcher initially defines the main problem; it is the problem that this study revolves around. Then, the researcher stated that this main problem actually can be divided into four sub problems. The researcher moved deep in explaining each sub problem. After that, the researcher shifted to the objectives of the study. Here, the researcher stated in one statement to provide a precise description of the actions to be taken in order to solve the research problem. In other words, the researcher tries to summarize what is to be done in this study. The researcher then described the objectives of the study one by one in a concise manner; in addition, the objectives were formulated based on the problem statement.

3.6 Phase 5: Designing the Methodology

This phase represents the contribution in this thesis. The researcher uses the research idea, research background and the literature review in carving the methodology of this study. The methodology phase consists of four sub phases. In the first and second sub phases, the researcher explains the mechanisms of the LLSs, and HLSs. The third sub phase is directed for explaining the Guards' mechanisms that control the activation/deactivation of the LLSs and HLSs. Finally, the fourth sub phase is about the CPN models that the researcher proposed in this thesis.

3.6.1 Sub Phase 1: Designing the Low-Level Schedulers (LLSs)

This sub phase is dedicated for explaining the proposed LLSs. The research describes the mechanisms of these schedulers one by one in detail. This includes the technique of partitioning and manipulating the threads. Starting with Fibonacci series (Section 4.1.1), the researcher gives an explanation supported with Figure 4.1 that shows the proposed Fibonacci thread's template. Then, the researcher demonstrates the Fibonacci LLS mechanism (Figure 4.2) as a flowchart. Following that, an example of threads' partitioning and manipulation (Figure 4.3) for computing Fibonacci (6) is given.

Then, the researcher moves to the Binary Search problem (Section 4.1.2). The researcher reviews the basis of the searching technique, the advantage and disadvantage of the technique. Following that, the researcher proposes a thread for the Binary Search problem (Figure 4.4). Next, a Binary Search LLS is illustrated as a flowchart which shows the mechanism of creating and evaluating the threads (Figure 4.5). Finally, an example to a tree of threads that shows the proposed partitioning technique is given (Figure 4.6).

Next, the researcher shifts to another D&C problem; it is the Towers of Hanoi problem (Section 4.1.3). The researcher reviews the history of this game. Then, the researcher explains how this game can be played, as well as the rules, and the objective of the game. After that, the proposed thread and move structures are explained (Figure 4.8). Later, the proposed Towers of Hanoi LLS is given in a flowchart that shows the partition of threads in addition to the creation of the game's moves (Figure 4.9). Lastly, an example for the game is given (Figure 4.10). The example includes the creation of the tree of threads beside the creation of the list of moves.

The Matrix Multiplication problem (Section 4.1.4) comes after the Towers of Hanoi problem. First, the researcher briefly surveys the importance of Matrix Multiplication in mathematics, and then moves to the proposed thread for this kind of problem (Figure 4.11). Then, a flowchart is given which shows the mechanism of the Matrix Multiplication LLS (Figure 4.12). Following that, a tree of threads' partitioning for a problem of multiplying two matrices is shown in detail (Figure 4.13). Next, the researcher provides another version of LLS (Figure 4.15). In this version, leaf-level threads are directly computed. Finally, another tree of threads' partitioning following the second version is shown in detail (Figure 4.14).

3.6.2 Sub Phase 2: Designing the High-Level Schedulers (HLSs)

After explaining the proposed threads and LLSs, the researcher moves to the proposed HLSs. In this sub phase, the researcher proposes five strategies that are built on the basis of Work-Stealing. The researcher explains the mechanisms of these strategies in redistributing the threads. All strategies are designed to move threads from the victim(s) core(s) to the thief cores. A set of flowcharts are given to explain the mechanisms of these strategies: The InOrderSingleStealing Strategy (IOSSS - Figure 4.16), the InOrderMultiStealing Strategy (IOMSS - Figure 4.17), the RichestFirstSingleStealing Strategy (RFSSS - Figure 4.18), the RichestFirstMultiStealing Strategy (RFMSS - Figure 4.19), the CompleteMultiStealing Strategy (CMSS - Figure 4.20), and the PartialMultiStealing Strategy (PMSS - Figure 4.24).

3.6.3 Sub Phase 3: Designing the Guards' Mechanisms

The researcher describes the Guards' mechanisms. The researcher designs two types of Guards that control the activation/deactivation of the LLSs and HLSs (Figure 4.27). The HLS Guard is activated, that is, locks the HLS when all the modelled cores are busy, and it unlocks the HLS when there is at least one idle core, at the same time, there is one or more victim core(s). On the other hand, the LLS's Guard locks the core for further processing when the core is considered a victim core; at the same time, there are one or more thief cores.

3.6.4 Sub Phase 4: Designing the CPN Models

In this sub phase, the researcher designs the proposed CPN models. The researcher starts by proposing the hierarchical designs of these models. After that, the researcher moves to explain the design of each model, starting with Fibonacci model and ending with Matrix Multiplication model. The researcher explains the elements of these models in detail and describes their functionality. Precisely, the researcher explains how the LLSs, HLSs, and the guards' mechanisms have been applied in these models. The sub phase (Section 4.4) is supported with many figures that show the CPN models as taken from inside the CPN-Tool: the CPN model for Fibonacci Problem (Figure 4.28, Figure 4.29, and Figure 4.30, and Figure 4.31), the CPN model for the Binary Search Problem (Figure 4.32, Figure 4.33, and Figure 4.34), the CPN model for the Towers of Hanoi Problem (Figure 4.35 and Figure 4.36), and finally, the CPN model for the Matrix Multiplication Problem (Figure 4.37, Figure 4.38, and Figure 4.39).

3.7 Phase 6: Simulation and Monitoring

This phase is dedicated for illustrating the results of the simulation and monitoring processes. This phase has been conducted as follows:

(a) At the beginning, the researcher explains the importance of the CoresLoad place (Section 5.1) in registering the sizes of the threads which are used to calculate the results. Following that, the researcher gives an example of the CoresLoad values (Table 5.1) by solving Fibonacci (10) problem three times: first using Eight-Thread Partitioning, then, using Four-Thread Partitioning, and finally, Two-Thread Partitioning.

(b) Then, the researcher gives an explanation to the criteria that is adopted by the thesis in the evaluation process. The Average Execution of Concurrent Steps (AES) is explained in detail (Section 5.1).

(c) After that, the researcher re-executes the Fibonacci (10) problem ranging from two until ten-core models. In each execution, the AES values (Table 5.2 and Table 5.3) are registered and saved inside a text file. Later, the researcher uses these values in drawing a histogram (Figure 5.2) that shows the relation between the AES values and the number of cores. This scenario is repeated for Fibonacci (13) (Table 5.4 and Figure 5.3) and Fibonacci (15) (Table 5.5 and Figure 5.4). The researcher solves these two problems two times; one using the three partition methods, and the second using only the two partition method. Finally, the researcher discusses the results.

(d) The researcher moves to the Binary Search Problems (Section 5.3). First, the problem is illustrated, then the researcher gives an example to a Binary Search Problem which includes the searching list, indices, and the searching item. After that, as in Fibonacci, the researcher solves the Binary Search Problem two times using nine models (Table 5.8 and Figure 5.7) and (Table 5.9 and Figure 5.8). The AES values are illustrated and histograms are drawn. Following that, the researcher discusses the results.

(e) Next problem is the Towers of Hanoi (Section 5.4). The researcher starts by giving a brief explanation on the parameters of this game which are represented by the number of disks and the pillar numbers. Then, the researcher explains also in brief the moves of the game. After that, the researcher solves two problems; one for seven disks and the other for nine disks. The results, the moves, are shown for the two examples (Table 5.10, Table 5.11, Table 5.12, and Figure 5.9) (Table 5.13 and Figure 5.10). Finally, a detailed discussion to the results is conducted.

(f) The final problem is the Matrix Multiplication (Section 5.5). As in the previous problems, the researcher explains the parameters of multiplying two matrices, the dimensions of the matrices. Following that, two examples are given: the first one is multiplying two matrices, each with dimension of 10 × 10 (Table 5.14), and the second is multiplying two matrices, each with the dimension of 20 × 20 (Table 5.15). The AES values are computed and listed inside the tables based on which the histograms are sketched (Table 5.16 and Figure 5.11) and (Table 5.17 and Figure 5.12). Finally, the researcher analyses the results.

3.8 Summary

This chapter describes the approach adopted by the researcher in conducting this study. The researcher conducted this research through six phases to highlight the main features of the study. This includes the purpose of each phase in this study and its relation with other phases. Some of these phases include sub phases, in such case, the researcher explains the purpose of each of these sub phases and its relation with other sub phases within the same phase. In conclusion, this chapter gives a general overview of the methodology adopted by the researcher in carrying out this study.

Starting with the first phase, Building the Research Idea, this phase represents the starting point of this research. It gives a clear image on how the idea originated in the researcher’s imagination. Then, in phase two, Studying of Research Background, the researcher explains the research background that has been established. The Literature

Review Phase comes next, the researcher shows how the literature review has been conducted in four sub phases. The first sub phase reviews the main problem, i.e. reviewing the existing research works involving the limitations of single-processors. Gradually moving to the second sub phase, the researcher narrows down the research area and focuses on the algorithms that partition the workload. The researcher continues in the next sub phases in narrowing down the search area through focusing on the multithreaded scheduling algorithms. Finally, the researcher settled down on existing studies covering work-stealing algorithms. These four sub phases assist the researcher in finding the gaps and issues in the other studies. This finding represents the keystone to formulate the problem statement and the objectives of this study in the next phase. Then, based on the above, the researcher proposes the design methodology through four sub phases. These sub phases clearly show the mechanisms of the low-level scheduler, the high-level scheduler, and the guards. Next, the last sub phase is dedicated for modelling the problem using Colored Petri Nets. The final phase is dedicated for conducting the simulation and monitoring processes. In this phase, the researcher gives examples of the problems that have been solved in this study and how the results have been collected in tables and sketched in histograms.

CHAPTER 4: DESIGN OF CONCURRENT MULTITHREADED MODELS FOR DIVIDE AND CONQUER PROBLEMS

In this thesis, the researcher focuses on solving D&C problems, namely, Fibonacci Series, Towers of Hanoi, Binary Search, and Matrix Multiplication on a modeled multicore environment. For this purpose, the researcher proposes two types of schedulers: The Low-Level Schedulers (LLSs) and the High-Level Schedulers (HLSs). The LLSs include workload partitioning and manipulating mechanisms while the HLSs include work-stealing mechanisms. The LLSs are responsible for dividing the main D&C problem (main thread) into smaller sub problems and then solve these sub problems towards reaching the final solution of the main D&C problem. On the other hand, the HLSs are in charge of balancing the workload (threads) among the modeled cores. The mechanisms of LLSs are presented in Section 4.1 while the mechanisms of HLSs are presented in Section 4.2. In Section 4.3, the researcher illustrates guards' mechanisms which control the activation of the LLSs and HLSs. Section 4.4 is dedicated for presenting the proposed CPN models, and finally, Section 4.5 is allocated for chapter summary.

4.1 The Low-Level Schedulers (LLSs)

In this section, the researcher proposes several workload partitioning and manipulating mechanisms that suit D&C problems. In general, any D&C problem is represented by a main thread and the duty of these schedulers is to partition the main thread into sub threads and manipulate them to get the results. The scheduler creates a tree of threads, the tree can be binary, or non-binary, it depends on the technique of partitioning. In this study, a thread is modeled as an n-tuple of parameters that represents the D&C problem's specification. The number and the type of the parameters depend on the type of the D&C problem itself. Different problems have different specifications; therefore they have different numbers and types of parameters. The importance of the threads comes from the fact that the division and the balancing operations are actually done on the thread's parameters not on the data. For this reason, it becomes crucial to decide the number of the parameters and the type of each parameter in the threads prior to any kind of processing. Finally, regardless of the type of D&C problem, every modeled thread is represented as a tuple that includes at least two distinguished parameters: the ThreadId parameter stands for thread's number and the FatherId stands for parent thread's number. Both these parameters are denoted as a positive integer number (Figure 4.1). The division process creates a tree of threads as shown in Figure 4.3. Starting with the main (root) thread, this root thread has ThreadId with value 1 and FatherId with value 0. On the other hand, the values of the ThreadId and FatherId in the ancestors' threads in the tree of threads are computed according to the division mechanism of the D&C problem. This is because the threads' tree is not always a binary tree as in Figure 4.3, the tree shape depends on the D&C problem's LLS.

4.1.1 The Fibonacci Low-Level Scheduler (FLLS)

Fibonacci series is an example of D&C problems (Cormen, et al., 2009; Miller & Vandome, 2010). The series is given as: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, etc. The first two terms are 0 and 1, the other terms can be calculated as: Tn = Tn-1 + Tn-2 , where n ≥ 2 and Tn is the n^{th} term. The proposed Fibonacci thread (Figure 4.1) is modeled as a 3-tuple: (ThreadId, FatherId, N), where the first two parameters represent the thread's number and the parent thread's number. The third parameter, N, holds the value "n", as Fibonacci (n) = Fibonacci (n-1) + Fibonacci (n-2).

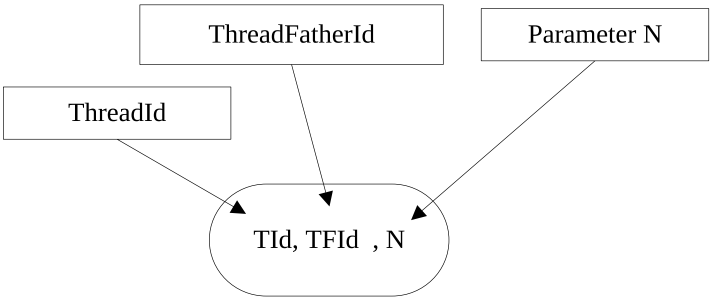


Figure 4.1: The Fibonacci thread where TId and TFId stand for ThreadId and FatherId respectively. The parameter N holds the value “n” as in Fibonacci (n) = Fibonacci (n-1) + Fibonacci (n-2).

The proposed workload partitioning mechanism (FLLS) is illustrated in Figure 4.2. The FLLS receives as input two parameters; a list of threads (LT) and the current result “Res”. The “Res” works as a global variable that can be updated by the FLLS. Initially, a new thread is extracted from the LT; then depending on the value of the parameter N, one of the following states is executed:

(a) State a: If the thread’s Parameter N is less than two, then Res is directly computed as:

Res ← Res + 0 ← 0 ← Fibonacci (0) ← Parameter N is 0
Res ← Res + 1 ← 1 ← Fibonacci (1) ← Parameter N is 1

(b) State b: If the Parameter N is less than or equals three, then the FLLS can create only two children threads at once and then connect those threads with the LT. The tree of threads is binary, i.e. a node in the tree can hold only two children threads. Therefore, in the case where N is equal to two, this leads to: Fibonacci (2) ← Fibonacci (1) + Fibonacci (0), while in the case where N is equal to three, this leads to: Fibonacci (3) ← Fibonacci (2) + Fibonacci (1). In both cases, two threads can be created at the same time; however, a value of N exceeding three can generate more than two threads as in the next state.

(c) State c: If the Parameter N is less than or equals five, then four children threads can be created by the FLLS and then connected them with the LT. The Fibonacci equation is computed as follows:

The original Fibonacci equation is given as:

Fibonacci (n) ← Fibonacci (n-1) + Fibonacci (n-2)   Equation 1

Since the FLLS is able to create four children threads, then there is a need to divide each child thread in the right hand side of Equation 1 into two children threads. Therefore we get:

Fibonacci (n-1) ← Fibonacci (n-1-1) + Fibonacci (n-2-1)
Fibonacci (n-2) ← Fibonacci (n-2-1) + Fibonacci (n-2-2)

After substituting Fibonacci (n-1) and Fibonacci (n-2) in Equation 1, we get:

Fibonacci (n) ← Fibonacci (n-2) + Fibonacci (n-3) + Fibonacci (n-3) +
Fibonacci (n-4)   Equation 2

The last equation is dedicated for values of N equal to four or five, four children threads can be created. Values of N less than four are already processed in the previous states while values of N above five are going to be processed in the next states, since a value of N, say six or above, can create eight children threads at the same time.

(d) State d: If the Parameter N is above five, then eight children threads can be created at the same time by the FLLS and then connected to them with the LT. The computation is as follows:

We already have in Equation 2:

Fibonacci (n) ← Fibonacci (n-2) + Fibonacci (n-3) + Fibonacci (n-3) + Fibonacci (n-4)

The FLLS extends each term in the right hand side into two children threads, so we get:

Fibonacci (n-2) ← Fibonacci (n-3) + Fibonacci (n-4)
Fibonacci (n-3) ← Fibonacci (n-4) + Fibonacci (n-5)

Fibonacci (n-4) ← Fibonacci (n-5) + Fibonacci (n-6)

Therefore, Fibonacci (n) can be computed as:

Fibonacci (n) ← Fibonacci (n-3) + Fibonacci (n-4) + Fibonacci (n-4) +
Fibonacci (n-5) + Fibonacci (n-4) + Fibonacci (n-5) + Fibonacci (n-5) + Fibonacci (n-6)
Equation 3

The last equation, Equation 3, indicates that for any value of N greater than five, eight children can be created at the same time.

The process is repeated until the LT becomes null. Each core executes its own FLLS on its threads. However, all the cores share the same value of Res. The value of the ThreadId is computed as $2^{\text{Tree's Level}}$, starting with zero as the first level number (main thread's level). On the other hand, the FatherId is computed by dividing ThreadId by two.

Figure 4.3 gives an example to the above mechanism. The figure shows a tree of partitioning the thread (1,0,6) which is dedicated for computing Fibonacci (6). In this example and according to the FLLS (Figure 4.2), eight children threads (those with bold font) are computed directly without passing through their ancestors. In other words, the FLLS jumps directly to those children threads. The FLLS can be extended to create sixteen children threads at once but in this case the value of N should be greater so it can afford creating sixteen threads at the same time.

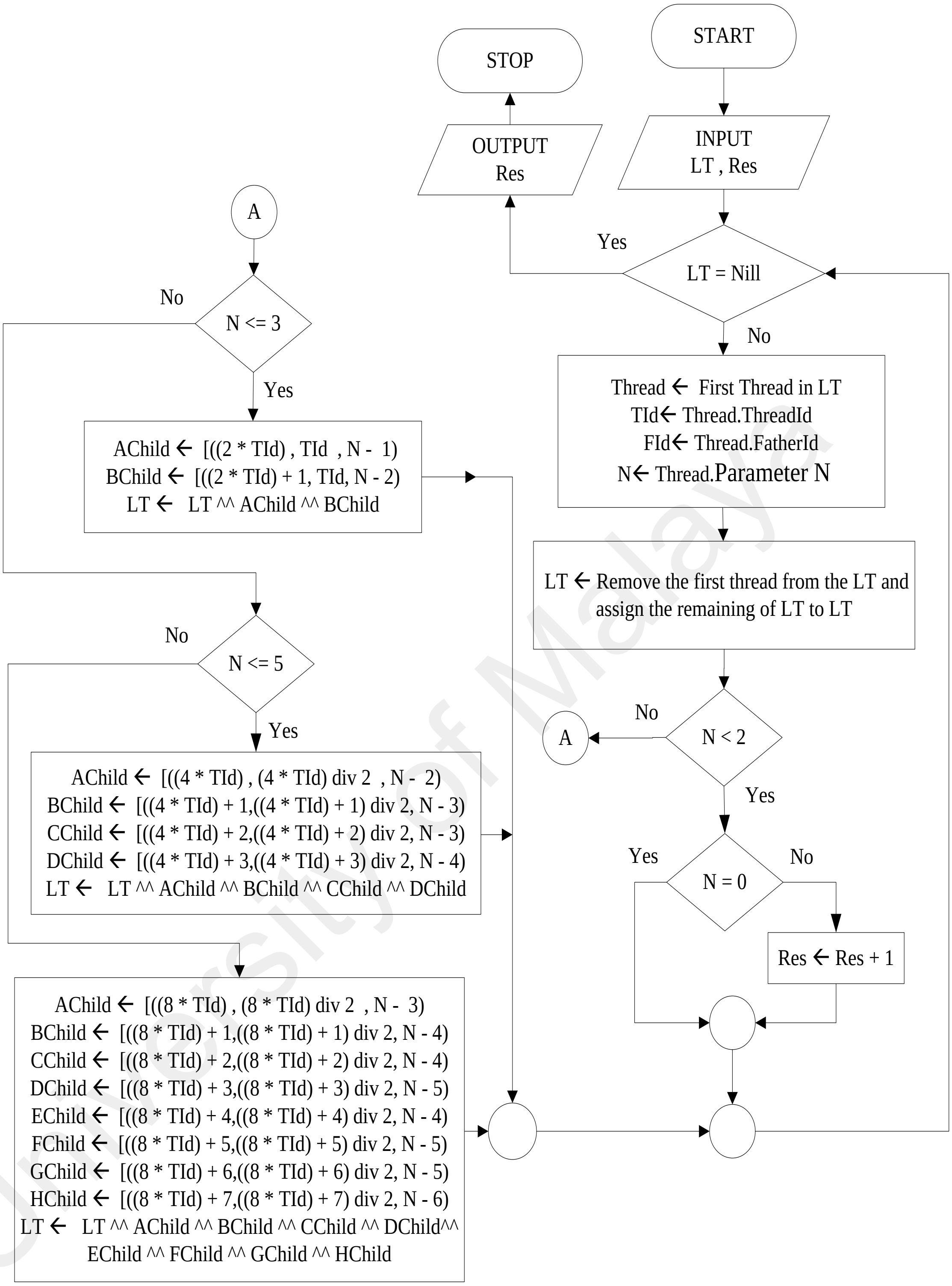

START
INPUT
LT , Res
LT = Nill
Yes
OUTPUT
Res
STOP
No
Thread ← First Thread in LT
TId← Thread.ThreadId
FId← Thread.FatherId
N← Thread.Parameter N
LT ← Remove the first thread from the LT and assign the remaining of LT to LT
N < 2
No
A
Yes
N = 0
Yes
No
Res ← Res + 1
A
N <= 3
No
Yes
AChild ← [((2 * TId) , TId , N - 1)
BChild ← [((2 * TId) + 1, TId, N - 2)
LT ← LT ^^ AChild ^^ BChild
N <= 5
No
Yes
AChild ← [((4 * TId) , (4 * TId) div 2 , N - 2)
BChild ← [((4 * TId) + 1,((4 * TId) + 1) div 2, N - 3)
CChild ← [((4 * TId) + 2,((4 * TId) + 2) div 2, N - 3)
DChild ← [((4 * TId) + 3,((4 * TId) + 3) div 2, N - 4)
LT ← LT ^^ AChild ^^ BChild ^^ CChild ^^ DChild
AChild ← [((8 * TId) , (8 * TId) div 2 , N - 3)
BChild ← [((8 * TId) + 1,((8 * TId) + 1) div 2, N - 4)
CChild ← [((8 * TId) + 2,((8 * TId) + 2) div 2, N - 4)
DChild ← [((8 * TId) + 3,((8 * TId) + 3) div 2, N - 5)
EChild ← [((8 * TId) + 4,((8 * TId) + 4) div 2, N - 4)
FChild ← [((8 * TId) + 5,((8 * TId) + 5) div 2, N - 5)
GChild ← [((8 * TId) + 6,((8 * TId) + 6) div 2, N - 5)
HChild ← [((8 * TId) + 7,((8 * TId) + 7) div 2, N - 6)
LT ← LT ^^ AChild ^^ BChild ^^ CChild ^^ DChild^^ EChild ^^ FChild ^^ GChild ^^ HChild

Figure 4.2: The Mechanism of FLLS

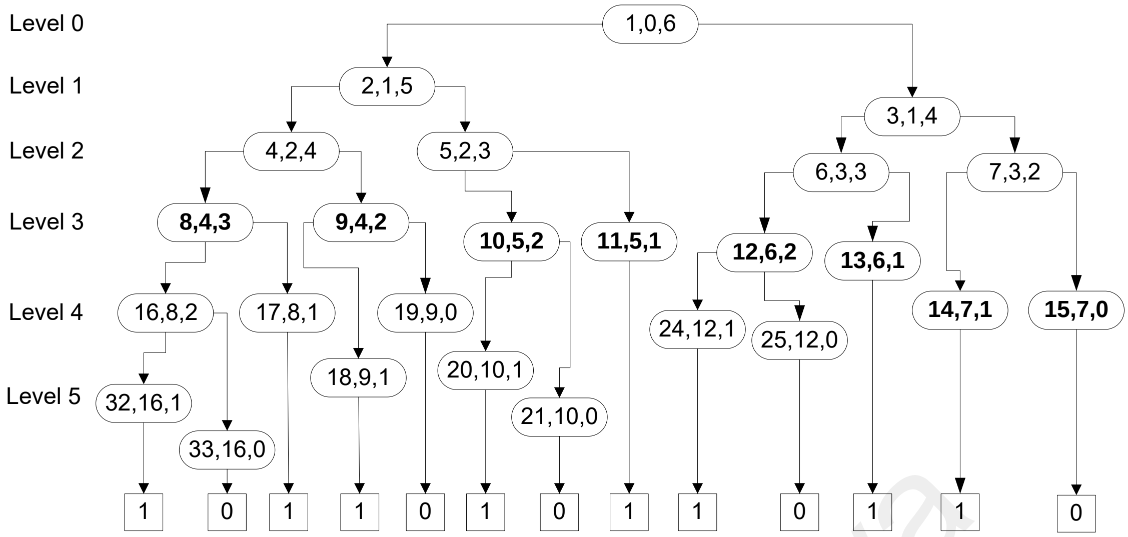


Figure 4.3: A binary tree for computing Fibonacci (6). The root thread is (1,0,6) which represents the main thread. The result is the summation of values inside the squares which is equal to 8.

4.1.2 The Binary Search Low-Level Scheduler (BSLLS)

Binary Search (BS) is another example of D&C problems (Cormen, et al., 2009; Miller & Vandome, 2010). It is a searching algorithm that is dedicated for finding a certain element in an ordered array (list). The algorithm's technique starts by comparing an input value with the element in the middle of the ordered list. If no matching exists then the result of comparison determines in which half of the list the process will be repeated. In case that the input value is less than the element in the middle; then the algorithm is repeated only on the elements that come before the element in the middle (left half). Otherwise, the searching will be focused only on the elements that come after the element in the middle (right half). The result of the algorithm which is considered as an algorithmic function has a running time of *O*(log n) and it can be applied iteratively or recursively. A major downside in this algorithm happens when new elements are added to the list. This will enforce to resort the list again prior to any new searching (Lea, 2005) (Mitzenmacher, 1998).

The BS thread (Figure 4.4) is modeled as 5-tuple: (ThreadId, FatherId, Element, StartIndex, EndIndex). The Element parameter represents the value that is to search for while the StartIndex and EndIndex stand for the starting and ending indices of the array or list.

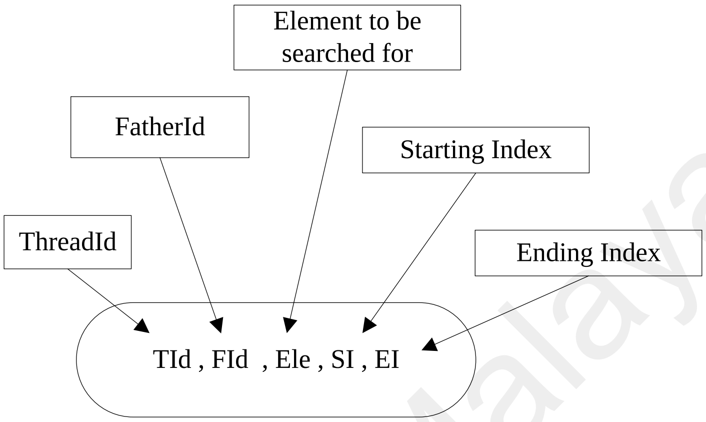


Figure 4.4: The Binary Search thread

The mechanism of the BSLLS (Figure 4.5) accepts three inputs: First the list of threads (LT) which initially holds a single (main) thread. This thread as shown in Figure 4.4 includes the element to be searched for (Ele) and the start and end indices (SI and EI) for the entire array or list. The second input is the list of numbers (LN) where the BSLLS is going to search inside it (the LN is kept inside the common area as shown in Figure 3.1). In this study, the researcher does not stress on the type of LN whether it is a list or an array. The only thing that matters is that LN can be indexed and this index is greater than or equal zero. For simplicity, the researcher averts using negative indices. The third input is Delta which holds the difference between the SI and EI in the thread. The value of Delta determines the number of threads that are going to be created.

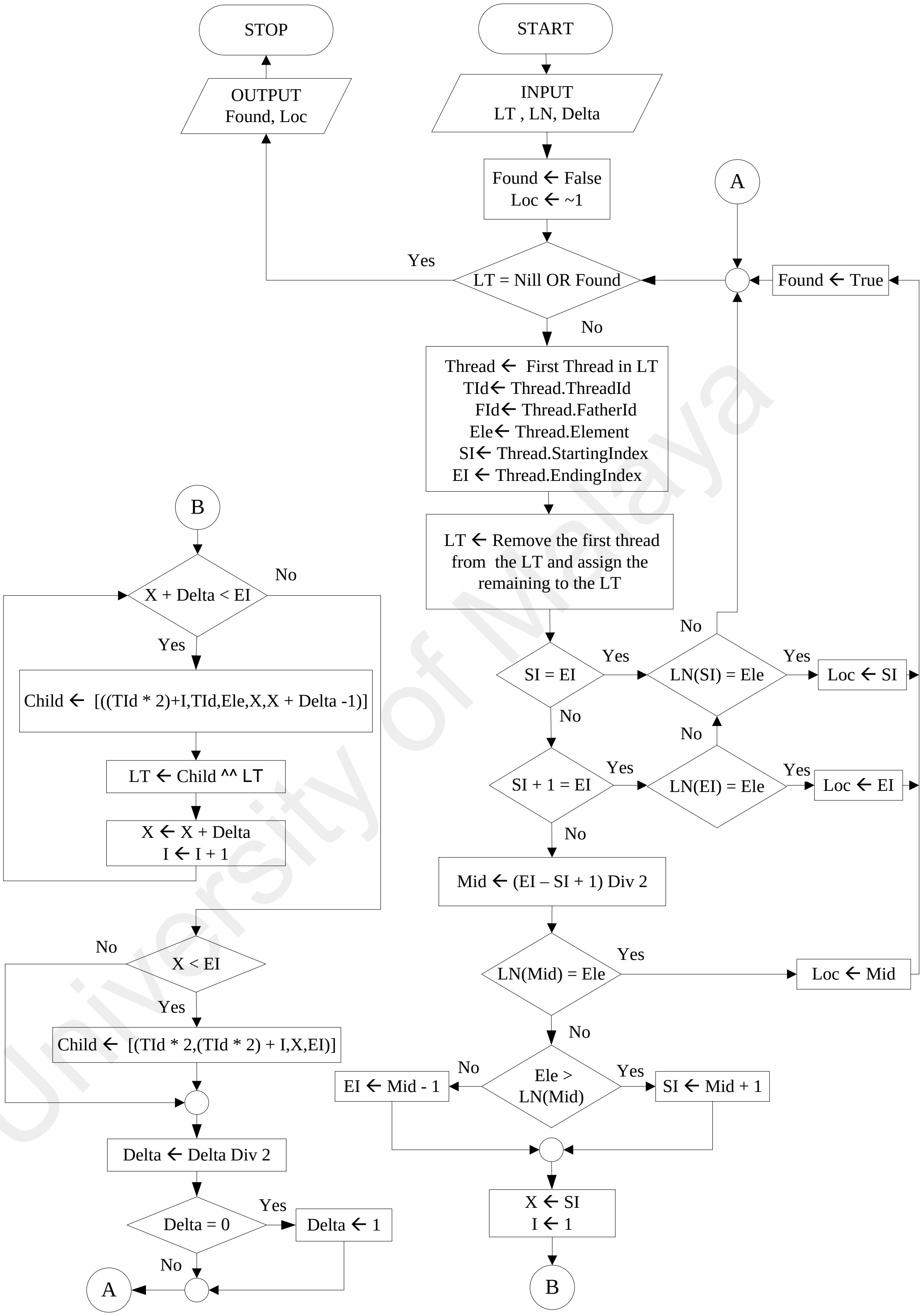


Figure 4.5: The Mechanism of BSLLS

Initially, the BSLLS initializes two variables: Found and Loc. In case the searching process succeeded in finding the element required searching for, then the Boolean variable Found and the integer variable Loc are adjusted to carry the values True and the element's location. Otherwise, Found gets False and Loc gets ~1 (the ~ symbol stands for minus in SML). Next, as in the FLLS, the BSLLS extracts a thread from the LT, takes out the parameters, and then adjusts the LT. After that, the scheduler checks whether the thread's range is equal to one (SI=EI) or two (EI=SI+1). In case there is a matching, the Found and Loc variables' values are adjusted and the searching terminates. Otherwise, the middle location is computed and the element in that location is tested against the searching element.

Then the process of threads' partitioning starts, here, the BSLLS creates a number of threads, each thread covers an area with length equal to Delta. The variables X (originally starts with SI) and Delta are used to compute the values of SI and EI for each thread while the variable I is used to create thread number. Because it is possible to repeat the partitioning process on the generated threads several times, it would be necessary to reduce the value of Delta. The searching process stops when the searching element is found or when no threads remain, which means the searching element is not included in LT. As an example to the above BSLSS, Figure 4.6 shows a tree of partitioning the thread (1,0,22,1,100000) which is dedicated for searching for the value 22 in an ordered list that consists of 100000 elements. The BSLSS partitions the main thread into 3334 threads; each is dedicated for searching for a specific range in the list. Delta is chosen to be 30 (this value can be increased or decreased as long as it won't exceed the range limits). First searching area is in the range 1-30, second area is in the range 31-60,…last one is in the range 99991-100000.

It is worth noting that the BSLLS tree of threads in Figure 4.6 differs from the FLLS tree (Figure 4.3). The former is a non-binary tree while the latter is a binary tree. This is because the BSLLS directly computes the leaf-level threads while the FLLS creates intermediate threads especially when dealing with high values of arguments. From the concurrency point of view, it is better to create leaf-level threads at once since these threads are distributed to all the cores and processed while creating intermediate threads may need to create more threads towards reaching leaf-level threads. In other words, it would be a waste of time. However, the full creation to the leaf-level threads in the BSLLS may also exhaust precious time. Therefore, there is a kind of trade-off between the full creation of leaf-level threads and the creation of the intermediate threads.

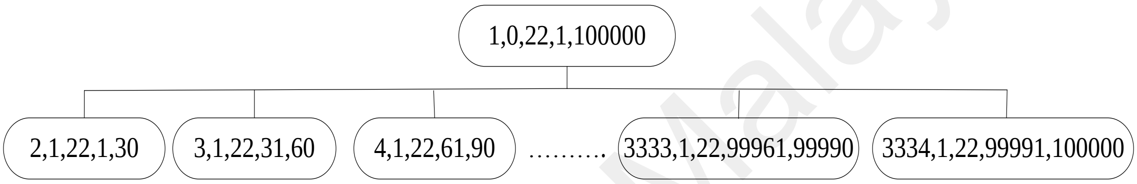


Figure 4.6: An example of partitioning a BS thread where list consists of 100000 elements.

4.1.3 The Towers of Hanoi Low-Level Scheduler (THLLS)

The Towers of Hanoi (TH) game is based on a puzzle that was first published by a French mathematician (François Éduoard Anatole Lucas) in 1883(Cormen, et al., 2009). The game (Figure 4.7) consists of three pillars and n disks. Initially, two of the pillars are empty. The first pillar contains n disks stacked with the largest disk at the bottom. Figure 4.7 shows an example of this game. In this example, three disks are located on the first pillar. The smallest disk has the number 1 while the largest disk has the number 3. The objective of this game is to move all the disks one by one from pillar 1 to pillar 3 under one condition: putting a large disk on top of a small one is not allowed. The output of this game is represented by a sequence of moves. Any single move consists of three parameters: disk number, source pillar, and destination pillar. The number of steps is equal to $(2^{n}) – 1$.

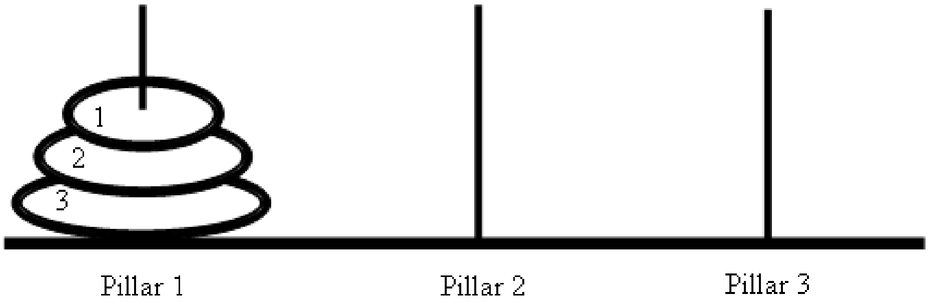


Figure 4.7: The Towers of Hanoi game with three disks

The proposed TH's thread (Figure 4.8) consists of seven parameters. The third parameter (Ord) stands for the move's Order. The fourth parameter (DNo) represents disk's number while the last three parameters represent source disk number (Sou), destination disk number (Des), and through disk number (Thr). In addition, the THLLS generates a list of moves (LM). A move (Figure 4.8) is an abbreviated description of a TH thread. The move is a 4-tuple: (Ord,DNo,Sou,Des), it has the information of moving sequence, disk number and the source and destination pillars.

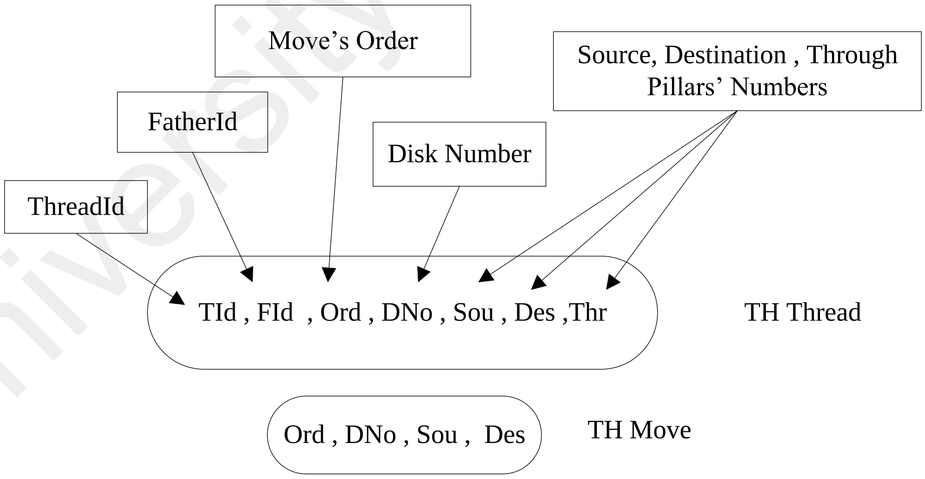

Move's Order
Source, Destination , Through Pillars' Numbers
FatherId
Disk Number
ThreadId
TId , FId , Ord , DNo , Sou , Des ,Thr
TH Thread
Ord , DNo , Sou , Des
TH Move

Figure 4.8: The Towers of Hanoi: thread and move

The mechanism of THLLS is given in Figure 4.9. The THLLS receives as input a list of threads (LT) and a list of moves (LM). The THLLS mechanism starts by decomposing the thread on the top of the LT into its components and then computes the new move.

Every new move consists of move's order (or sequence), the disk number, the pillar number that holds the disk, and the destination disk number. The moves are kept inside the LM. The AddNewMove routine (located to the left) is dedicated for adding new move to the LM. This routine works recursively in adding the moves to the LM. The routine uses the Ord parameter in arranging the moves in an ascending order. Following that, the THLLS checks the disk number whether it is greater than one; if it is so, the THLLS creates two sub threads (left and right) and adds them to the LT. Otherwise, the THLLS picks another thread from the LT. The process of computing the left and right sub threads (children threads) consists of computing sub threads' parameters one by one. The disk number for both sub threads is equal to the current thread's disk number minus one. However, the order of the left sub thread (left child) differs from its corresponding order in the right sub thread. Then, it is the turn of computing the pillars' numbers. Finally, the two sub threads are added to the LT. This process continues until the LT becomes Nil. The result of the mechanism is represented by the LM. An example to TH thread's partition is given in Figure 4.10. Threads are included inside oval shapes while the moves are included inside rectangle shapes.

A simple comparison between the binary tree of threads created by the THLLS (Figure 4.10) and those that belong to FLLS (Figure 4.3) and BSLLS (Figure 4.6) shows clearly the inability of the THLLS to create more than two threads at the same time. This is due to the fact that no more than one disk can be moved at the same time. In other words, the concurrent side in the Towers of Hanoi game is weak, while in the BSLLS, all the cores deal with various sections of the searching area at the same time. The same thing can be said regarding FLLS, where every core deals with a branch of the Fibonacci tree, at the same time, which ultimately leverages the concurrency level.

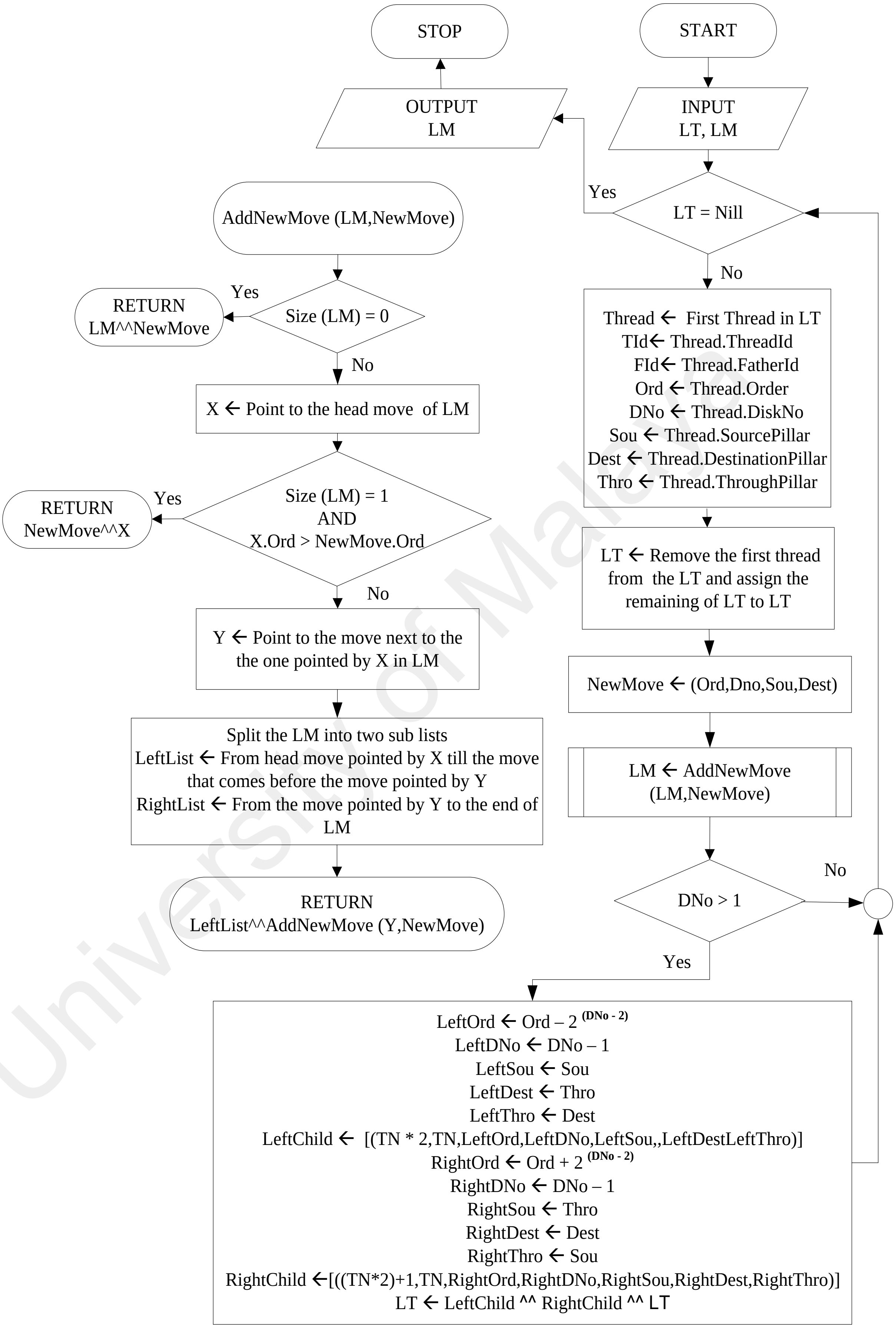

STOP
START
OUTPUT
LM
INPUT
LT, LM
Yes
LT = Nill
No
AddNewMove (LM,NewMove)
RETURN
LM^^NewMove
Yes
Size (LM) = 0
No
X ← Point to the head move of LM
Thread ← First Thread in LT
TId← Thread.ThreadId
FId← Thread.FatherId
Ord ← Thread.Order
DNo ← Thread.DiskNo
Sou ← Thread.SourcePillar
Dest ← Thread.DestinationPillar
Thro ← Thread.ThroughPillar
RETURN
NewMove^^X
Yes
Size (LM) = 1
AND
X.Ord > NewMove.Ord
No
LT ← Remove the first thread from the LT and assign the remaining of LT to LT
Y ← Point to the move next to the the one pointed by X in LM
NewMove ← (Ord,Dno,Sou,Dest)
Split the LM into two sub lists
LeftList ← From head move pointed by X till the move that comes before the move pointed by Y
RightList ← From the move pointed by Y to the end of LM
LM ← AddNewMove (LM,NewMove)
No
RETURN
LeftList^^AddNewMove (Y,NewMove)
DNo > 1
Yes
LeftOrd ← Ord – 2 (DNo - 2)
LeftDNo ← DNo – 1
LeftSou ← Sou
LeftDest ← Thro
LeftThro ← Dest
LeftChild ← [(TN * 2,TN,LeftOrd,LeftDNo,LeftSou,,LeftDestLeftThro)]
RightOrd ← Ord + 2 (DNo - 2)
RightDNo ← DNo – 1
RightSou ← Thro
RightDest ← Dest
RightThro ← Sou
RightChild ←[((TN*2)+1,TN,RightOrd,RightDNo,RightSou,RightDest,RightThro)]
LT ← LeftChild ^^ RightChild ^^ LT

Figure 4.9: The Mechanism of THLLS

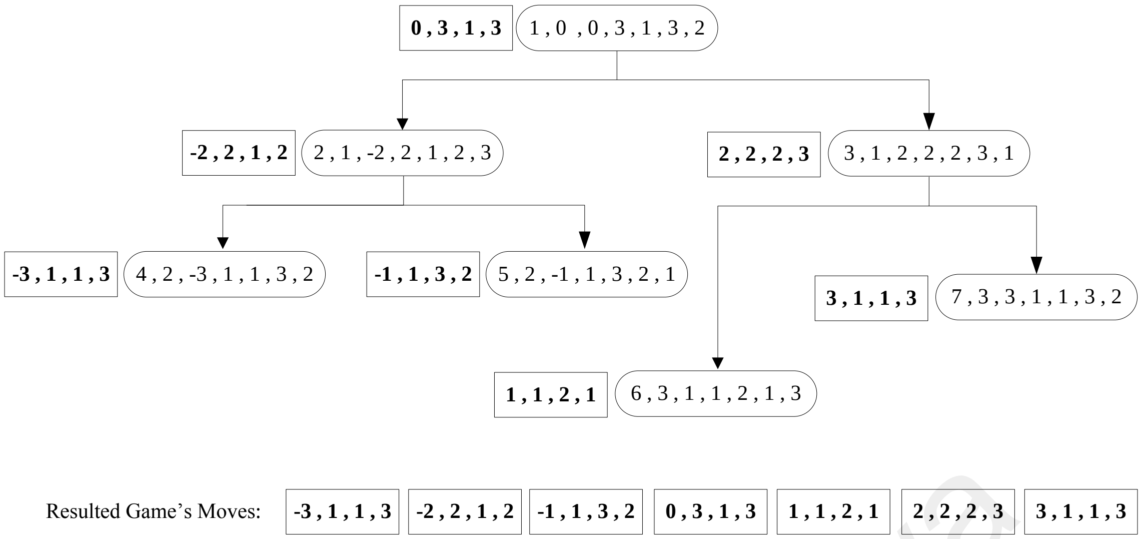


Figure 4.10: An example of partition a TH thread where the main thread (1,0,0,3,1,3,2) indicates that there are three disks.

4.1.4 The Matrix Multiplication Low-Level Scheduler (MMLLS)

Matrix Multiplication (MM) is one of the D&C problems that plays a main role in many scientific applications. It represents a keystone in a numerous number of problems such as transitive closure and reduction, solving linear systems of equations, matrix inversion, etc. The MMLLS schedules threads creation, partition, and managing to multiply two matrices. The MMLLS dynamically divides threads till reaching leaf-level threads (a leaf-level thread holds a row number of the first matrix and a column number of the second matrix). The researcher has modeled the thread (Figure 4.11) as a 7-tuple: (ThreadId, FatherId, StartRow, EndRow, N, StartColumn, EndColumn). In mathematics, multiplying two matrices, $A_{m,n} \times B_{n,p}$ generates a new matrix $C_{m,p}$, where m, n, p > 0. The parameter N stands for both the first matrix (A) column number and the second matrix (B) row number. The StartRow and EndRow parameters carry the starting and the ending numbers of the rows that belong to the first matrix (A). These two numbers are reduced through the process of division till they match. The resulting match number represents the required row number. The same thing can be said for

StartColumn and EndColumn. They carry the starting and ending column numbers in the second matrix (B).

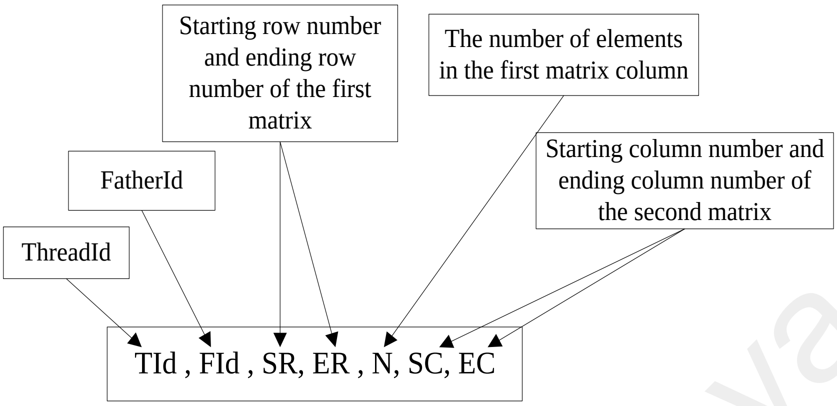

Starting row number and ending row number of the first matrix
The number of elements in the first matrix column
Starting column number and ending column number of the second matrix
FatherId
ThreadId
TId , FId , SR, ER , N, SC, EC

Figure 4.11: The Matrix Multiplication thread

The mechanism of the MMLLS is illustrated in Figure 4.12. The essence of the mechanism is in computing the values of the parameters: StartRow, EndRow, StartColumn, and EndColumn. The scheduler receives a list of threads and it (scheduler) continuously partitions these threads till reaching the level where a matching happens between the (StartRow ,EndRow) and (StartColumn,EndColumn). To explain the MMLLS in more detail, we can distinguish seven different statuses in calculating the above parameters.

(a) In Status 1, when both the StartRow and EndRow are matched, at the same time, StartColumn and EndColumn are matched also; here we have a leaf-level thread. The multiplier routine is called to compute multiplying a row numbered SR from the first matrix (MatA) by a column numbered SC from the second matrix (MatB). The parameter N stands for the number of elements in the SR row as well as for the SC column. The results of multiplication are assigned to the third matrix (MatC).

(b) Status 2 is reserved for threads that come one step before the leaf-level threads. Here, both the StartRow and EndRow are matched, at the same time, EndColumn exceeds StartColumn by one. The result is in generating two leaf-level threads, the first for multiplying row StartRow from the first matrix by column StartColumn from the second matrix. The second leaf-level thread is for multiplying row StartRow from the first matrix by column EndColumn from the second matrix.

(c) Status 3 is activated when both the StartRow and EndRow are matched, at the same time; StartColumn and EndColumn are not matched. However, EndColumn exceeds StartColumn by more than one. Here, the MMLLS divides the threads into two sets. The first set deals with threads that use row number StartRow from the first matrix with first half of the columns between StartColumn and EndColumn from the second matrix. The second set of threads uses the same row number, EndRow, from the first matrix with the second half of columns between StartColumn and EndColumn from the second matrix.

(d) All the mentioned statuses are dedicated for managing threads that have a matching in the row's parameters of the first matrix while statuses 4 and 5 are customized for threads that have mismatching in the row's parameters of the first matrix. In these statuses, the MMLSS mechanism divides the rows in the first matrix. When EndRow exceeds StartRow by one, Status 4 is called. Two normal threads are sent to the list; the first thread deals with row number StartRow from the first matrix with a set of columns (StartColumn, EndColumn) from the second matrix while the second thread deals with EndRow from the first matrix with the same set of columns from the second matrix. Status 5 manages the threads that have more than one value between StartRow and EndRow. The MMLSS mechanism divides the set of rows into two divisions. The ((EndRow – StartRow) / 2 + StartRow) has been taken as the point of threads' division. An example to MM threads' partitioning is given in Figure 4.13. The main thread contains the dimensions of the matrices: $A_{3,4}$× $B_{4,4}$. The resulting leaf-level threads are surrounded by bold borders. These threads are taken by the multiplier (Figure 4.12) to compute the values of the matrix $C_{3,4}$.

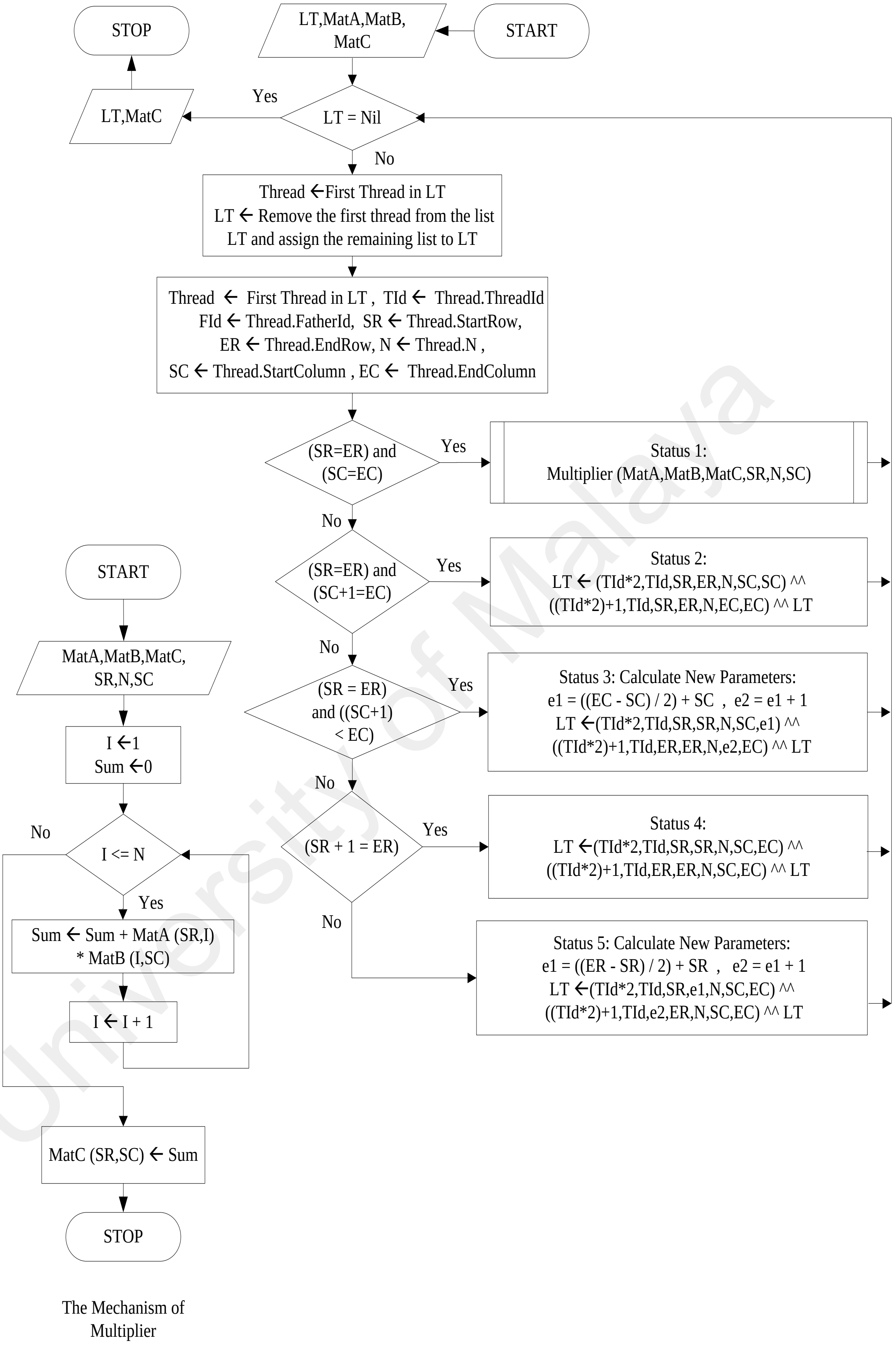

STOP
LT,MatA,MatB,
MatC
START
Yes
LT,MatC
LT = Nil
No
Thread ←First Thread in LT
LT ← Remove the first thread from the list
LT and assign the remaining list to LT
Thread ← First Thread in LT , TId ← Thread.ThreadId
FId ← Thread.FatherId, SR ← Thread.StartRow,
ER ← Thread.EndRow, N ← Thread.N ,
SC ← Thread.StartColumn , EC ← Thread.EndColumn
(SR=ER) and
(SC=EC)
Yes
Status 1:
Multiplier (MatA,MatB,MatC,SR,N,SC)
No
(SR=ER) and
(SC+1=EC)
Yes
Status 2:
LT ← (TId*2,TId,SR,ER,N,SC,SC) ^^
((TId*2)+1,TId,SR,ER,N,EC,EC) ^^ LT
No
(SR = ER)
and ((SC+1)
< EC)
Yes
Status 3: Calculate New Parameters:
e1 = ((EC - SC) / 2) + SC , e2 = e1 + 1
LT ←(TId*2,TId,SR,SR,N,SC,e1) ^^
((TId*2)+1,TId,ER,ER,N,e2,EC) ^^ LT
No
(SR + 1 = ER)
Yes
Status 4:
LT ←(TId*2,TId,SR,SR,N,SC,EC) ^^
((TId*2)+1,TId,ER,ER,N,SC,EC) ^^ LT
No
Status 5: Calculate New Parameters:
e1 = ((ER - SR) / 2) + SR , e2 = e1 + 1
LT ←(TId*2,TId,SR,e1,N,SC,EC) ^^
((TId*2)+1,TId,e2,ER,N,SC,EC) ^^ LT
START
MatA,MatB,MatC,
SR,N,SC
I ←1
Sum ←0
No
I <= N
Yes
Sum ← Sum + MatA (SR,I)
* MatB (I,SC)
I ← I + 1
MatC (SR,SC) ← Sum
STOP
The Mechanism of
Multiplier

Figure 4.12: The Mechanism of MMLLS

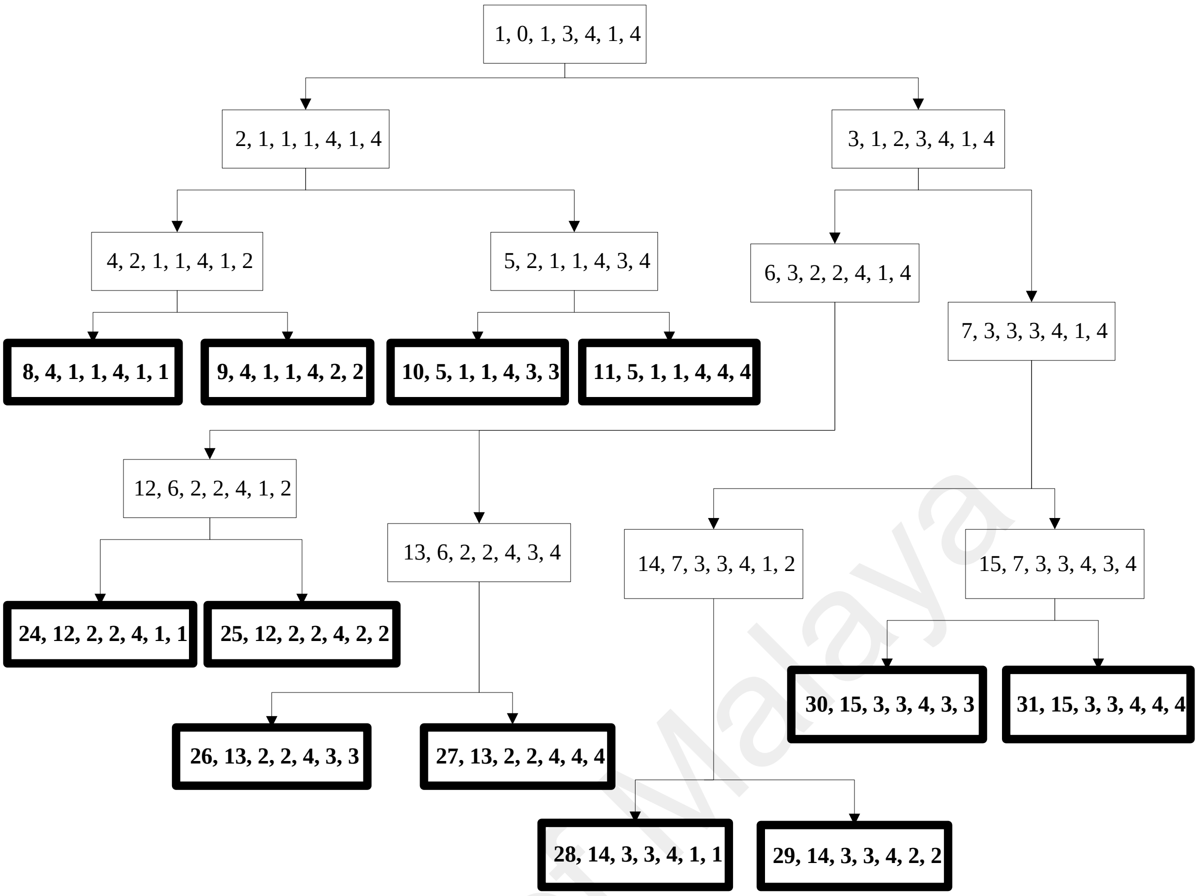


Figure 4.13: An example of partitioning a Matrix Multiplication thread where the main thread indicates $A_{3,4}$× $B_{4,4}$, and leaf-level threads are surrounded with bold borders

The ultimate goal of the MMLLS is to create a set of leaf-level threads (Figure 4.13) in order to permit the Multiplier (Figure 4.12) to compute the new matrix. However, it is possible to get rid the intermediate threads through creating the leaf-level threads directly from the main thread. In other words, it is possible to improve the MMLLS in a way similar to the BSLLS where leaf-level threads are generated directly from the main thread. The Direct Matrix Multiplication Low-Level Scheduler (DMMLLS) may be inconsistent with principle of D&C; nevertheless, it provides a fast decomposition to the workload. Figure 4.14 gives an example to the DMMLLS, while Figure 4.15 illustrates the mechanism of this scheduler. The x, y variables represent counters which generate the row and column numbers. Therefore, in DMMLLS, leaf-level threads are directly generated and joined to the LT.

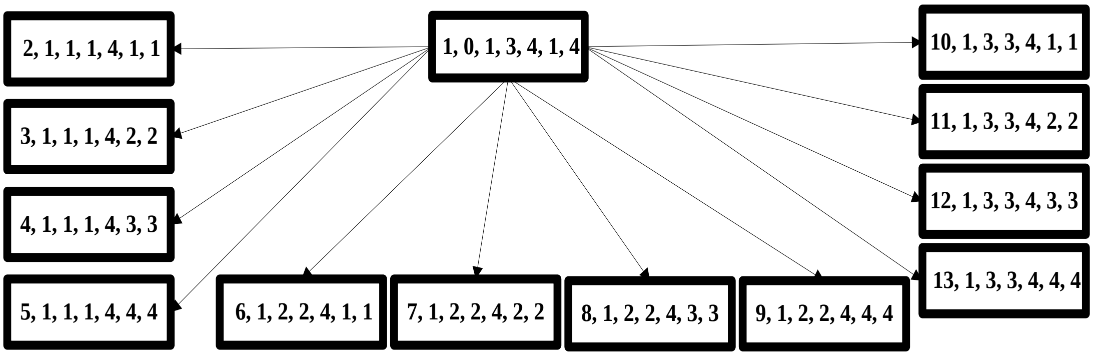


Figure 4.14: An example of partitioning a Matrix Multiplication thread where the leaf-level threads are computed directly

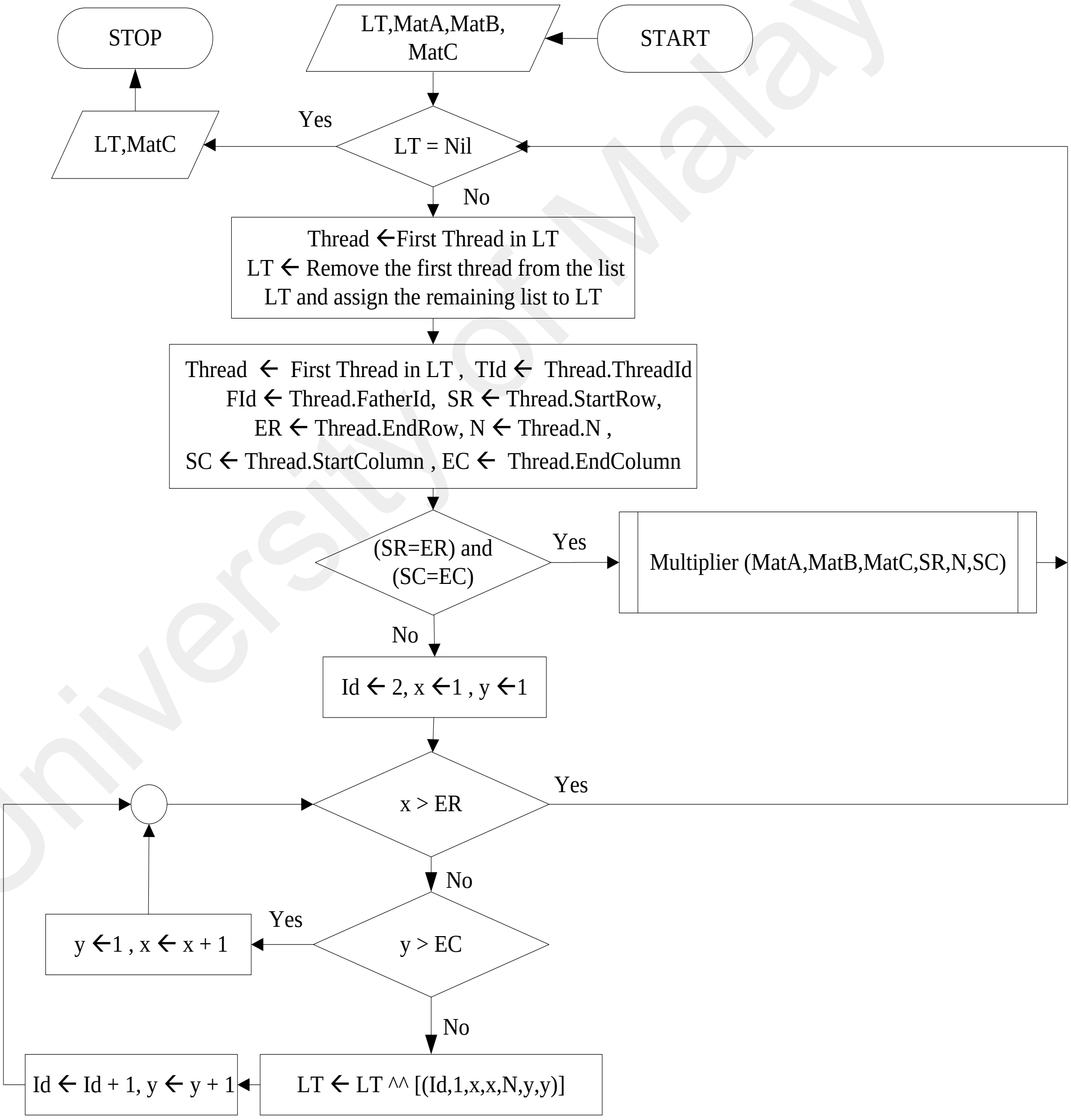

STOP
LT,MatA,MatB,
MatC
START
Yes
LT,MatC
LT = Nil
No
Thread ←First Thread in LT
LT ← Remove the first thread from the list
LT and assign the remaining list to LT
Thread ← First Thread in LT , TId ← Thread.ThreadId
FId ← Thread.FatherId, SR ← Thread.StartRow,
ER ← Thread.EndRow, N ← Thread.N ,
SC ← Thread.StartColumn , EC ← Thread.EndColumn
(SR=ER) and
(SC=EC)
Yes
Multiplier (MatA,MatB,MatC,SR,N,SC)
No
Id ← 2, x ←1 , y ←1
x > ER
Yes
No
Yes
y ←1 , x ← x + 1
y > EC
No
Id ← Id + 1, y ← y + 1
LT ← LT ^^ [(Id,1,x,x,N,y,y)]

Figure 4.15: The Mechanism of DMMLLS

4.2 The High-Level Schedulers (HLSs)

The function of the HLS is to balance the workload among the modelled cores. By workload, we mean the threads (created by LLSs) that are scattered in the cores. Although, the HLS has no rule in creating and manipulating these threads, its main purpose is to reallocate these threads to achieve a better performance. In other words, the HLS aims to make these cores busy as much as possible by moving threads from the non-idle (busy or victim) cores to the idle (thief) cores. In this study, the researcher didn't allow the cores to act alone in stealing the threads from other cores. It is the HLS's responsibility in reallocating the threads. For this purpose, the researcher develops several strategies (mechanisms) that control the redistribution process. The strategies are:

(a) The InOrderSingleStealing Strategy (IOSSS)
(b) The InOrderMultiStealing Strategy (IOMSS)
(c) The RichestFirstSingleStealing Strategy (RFSSS)
(d) The RichestFirstMultiStealing Strategy (RFMSS)
(e) The CompleteMultiStealing Strategy (CMSS)
(f) The PartialMultiStealing Strategy (PMSS)

Two important variables are common to all the strategies: *MainList* and *NumOfCores*. The variable *MainList* represents a list of sub lists of threads where each sub list is dedicated for a core. That is, *MainList (1)* is the list of threads that belongs to core No. 1, etc; therefore, *MainList* is a list of sub lists of threads. The other variable is *NumOfCores* which stands for the number of cores.

4.2.1 The InOrderSingleStealing Strategy (IOSSS)

This is the simplest strategy. The strategy (Figure 4.16) works as a function that accepts as input two variables (*MainList* and *NumOfCores*), reallocates the threads, and produces the updated *MainList*. The mechanism of the strategy includes the following steps:

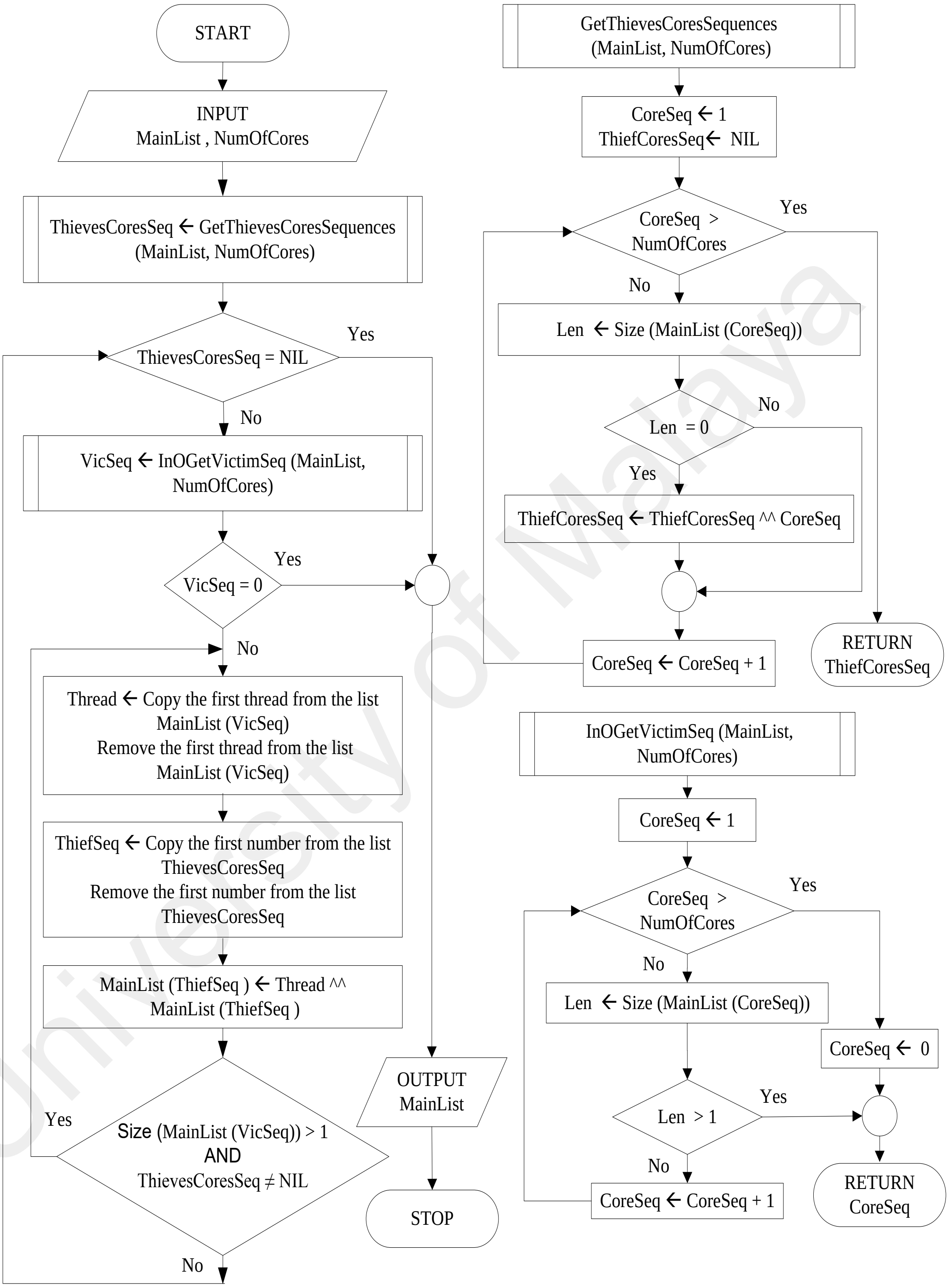
(a) Finding the set of thief cores' numbers. This process is done for only once, it returns a list of integer numbers that represent the set of thief cores' numbers. The function *GetThievesCoresSequences* is in charge of this process and the resulted list is given the name *ThievesCoresSeq*. The function simply checks the number of threads in each core. If a core is out of threads, the function assigns the core's sequence to the list. Cores with one or more threads are not included in the list.

(b) Finding the first encountered victim core, the researcher defines a victim core as any core that has more than one thread. The searching process is carried on from core No. 1 to core No. *NumOfCores*. Although, one victim core may be enough to satisfy all the thief cores, however, this process may be repeated when there is a need for an additional victim core. The function *InOGetVictimSeq* searches for the first victim core and stores its number in *VicSeq*. The function *InOGetVictimSeq* uses the variable *CorSeq* to index the cores. A simple loop is used to investigate the size of each core through *MainList* (*CoreSeq*). A value zero of *VicSeq* means there is no more victim cores. In other words, all the cores have zero or one thread.

(c) The redistribution process. The *InOrderSingleStealing* function pulls a thread from the victim core and assigns it to the variable *Thread*. Then, the function redistributes this thread to one of the thief cores, one thread for each thief core.

(d) In case some of the thief cores are still idle, at the same time, there is a chance to get another victim core then, steps (b), and (c) are repeated and so on.

(e) The mechanism stops when all the thief cores become non-idle or there are no more victim cores.


START
INPUT
MainList , NumOfCores
ThievesCoresSeq ← GetThievesCoresSequences (MainList, NumOfCores)
ThievesCoresSeq = NIL
Yes
No
VicSeq ← InOGetVictimSeq (MainList, NumOfCores)
VicSeq = 0
Yes
No
Thread ← Copy the first thread from the list MainList (VicSeq)
Remove the first thread from the list MainList (VicSeq)
ThiefSeq ← Copy the first number from the list ThievesCoresSeq
Remove the first number from the list ThievesCoresSeq
MainList (ThiefSeq) ← Thread ^^ MainList (ThiefSeq)
Size (MainList (VicSeq)) > 1 AND ThievesCoresSeq ≠ NIL
Yes
No
OUTPUT
MainList
STOP
GetThievesCoresSequences (MainList, NumOfCores)
CoreSeq ← 1
ThiefCoresSeq← NIL
CoreSeq > NumOfCores
Yes
No
Len ← Size (MainList (CoreSeq))
Len = 0
No
Yes
ThiefCoresSeq ← ThiefCoresSeq ^^ CoreSeq
CoreSeq ← CoreSeq + 1
RETURN
ThiefCoresSeq
InOGetVictimSeq (MainList, NumOfCores)
CoreSeq ← 1
CoreSeq > NumOfCores
Yes
No
Len ← Size (MainList (CoreSeq))
CoreSeq ← 0
Len > 1
Yes
No
CoreSeq ← CoreSeq + 1
RETURN
CoreSeq

Note: The rectangles with double sided bares represent functions calls. These two functions: GetThievesCoresSequences and InOGetVictimSeq are illustrated to the right. The two symbols ^^ are used to concatenate a thread with the list of threads.

Figure 4.16: The InOrderSingleStealing Strategy (IOSSS) located at the left.

4.2.2 The InOrderMultiStealing Strategy (IOMSS)

The IOMSS shares the IOSSS way in finding the victim core and thief cores. However, there are two differences between the two mentioned strategies. First, in the IOMSS, more than one thread can be assigned to the thief cores whereas in the IOSSS only a single thread can be assigned to the thief cores. Second, the IOMSS offers a way to balance the threads between the victim core and the thief cores. In other words, the IOMSS provides a more equitable way of threads distribution.

In mathematics, we can balance the values of any N different variables as follows: Let V = (X_1, X_2, X_3, X_4 ... X_n) be a set of non-negative integer numbers ($X_i \geq 0$). To balance the values in these variables, we need to calculate the following (Strang, 2011):

Let Sum = $\sum_{i=1}^{N} X_i$

Let C = Sum + K, where K is a constant (K ≥ 0). The value of C represents the smallest value such that: C mod N = 0

Let H = (C / N)

Now, the new value of the first (N – K) variables is H, while the value of the rest is H-1.

Example: Let V = (1, 3, 2, 0, 7, 9, 1, 4).

We have N=8, Sum = $\sum_{i=1}^{8} x_i$ → Sum = 27

Now, the smallest value of C that achieves (C mod N = 0) is 32.

Since C = Sum + K → K = 32 – 27 → K = 5 , H = (32 / 8) → H = 4

Now, the first 3 variables will have the value 4, that is X_1=4, X_2=4, X_3=4, while the rest of the variables will have the value 3, that is: X_4=3, X_5=3, X_6=3, X_7=3, X_8=3.

The IOMSS (Figure 4.17) works as follows:

(a) As in the IOSSS, the IOMSS creates a list of thief cores' numbers; then it searches for the first encountered victim sequence.

(b) The strategy evaluates the following variables:

I- *ProcessedCores*: It represents the value N, as mentioned above, plus one. The one stands for the victim core. In other words, the variable *ProcessedCores* represents the number of thief cores plus the chosen victim core.

II- Initially, the *C* variable holds the number of threads in the victim core. Then, the value of *C* is increased to achieve the condition (*C mod ProcessedCores = 0*).

III- *VictimThreads*: This variable emulates the variable *Sum*. It represents the number of threads in the victim core.

(c) The strategy calls the function *Redistributor* which is responsible for moving the threads from the victim core to the thief cores. The function performs the following actions:

I- Dividing the cores (the thief cores and the victim core) into two groups: *FirstGroup* and *SecondGroup*. The first group includes the victim core and (all or part of) the thief cores. The second group includes zero or the thief cores (the rest of thief cores). Following that, computing group's values stand for the number of threads that are going to be assigned to each core in the group. The variables *FirstGroupValue* and *SecondGroupValue* represent the number of threads that are going to be assigned to the first and second group respectively. The variable *FirstGroup* gets the value (N-K) while its value *FirstGroupValue* holds the value H.

The other two variables *SecondGroup* and *SecondGroupValue* get the values K and (H-1) respectively.

II- Following that, the function *Redistributor* removes (*VictimThreads* – *FirstGroupValue*) thread(s), i.e. (Sum – H) threads from the victim core and assigning it to the list *TempList* (a temporary list that combines the separated threads from their cores). As a result, the victim core retains its share. In the next step, the victim core is excluded from the computation since it did its role in the computations. The value of *FirstGroup* is updated to exclude the victim core.

III- Next, the strategy starts distributing the threads of the *TempList* to the first and the second groups. The distribution process includes first, fetching the number of each thief core one by one from the list *ThievesCoresSeq*. Second, a number of *FirstGroupValue* of threads are cut from the *TempList* and assigned to each thief core in the first group. The same process is repeated on the second group members who receive *SecondGroupValue* of threads from the *TempList*.

(d) Finally, the strategy rechecks the list of the thief cores. If it is still non-empty and there is a new candidate victim core, then the strategy is repeated. Otherwise, no more action is taken and the strategy stops.

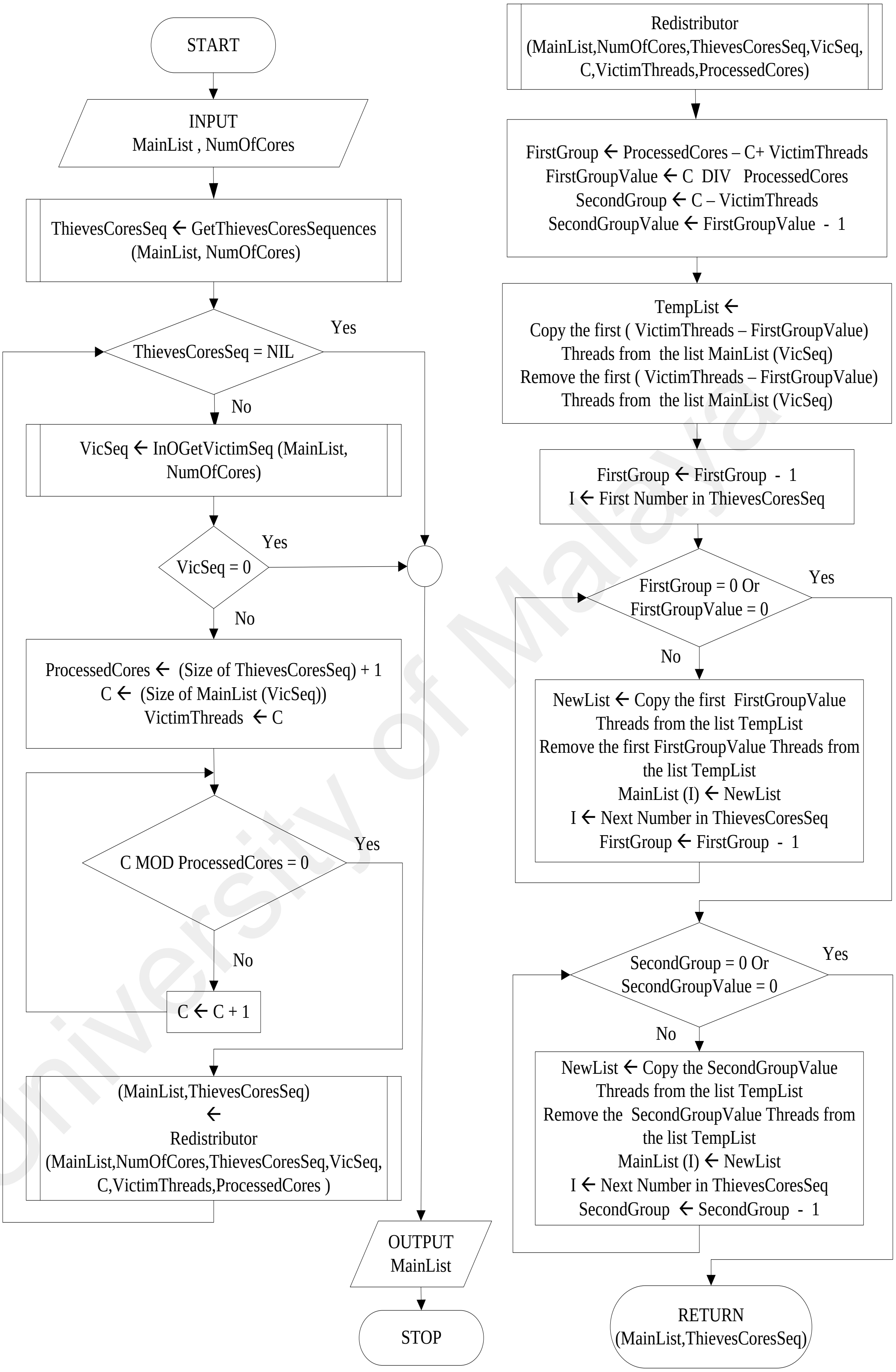

START
INPUT
MainList , NumOfCores
ThievesCoresSeq ← GetThievesCoresSequences (MainList, NumOfCores)
ThievesCoresSeq = NIL
Yes
No
VicSeq ← InOGetVictimSeq (MainList, NumOfCores)
VicSeq = 0
Yes
No
ProcessedCores ← (Size of ThievesCoresSeq) + 1
C ← (Size of MainList (VicSeq))
VictimThreads ← C
C MOD ProcessedCores = 0
Yes
No
C ← C + 1
(MainList,ThievesCoresSeq)
←
Redistributor
(MainList,NumOfCores,ThievesCoresSeq,VicSeq, C,VictimThreads,ProcessedCores)
OUTPUT
MainList
STOP
Redistributor
(MainList,NumOfCores,ThievesCoresSeq,VicSeq, C,VictimThreads,ProcessedCores)
FirstGroup ← ProcessedCores – C+ VictimThreads
FirstGroupValue ← C DIV ProcessedCores
SecondGroup ← C – VictimThreads
SecondGroupValue ← FirstGroupValue - 1
TempList ←
Copy the first (VictimThreads – FirstGroupValue)
Threads from the list MainList (VicSeq)
Remove the first (VictimThreads – FirstGroupValue)
Threads from the list MainList (VicSeq)
FirstGroup ← FirstGroup - 1
I ← First Number in ThievesCoresSeq
FirstGroup = 0 Or
FirstGroupValue = 0
Yes
No
NewList ← Copy the first FirstGroupValue
Threads from the list TempList
Remove the first FirstGroupValue Threads from
the list TempList
MainList (I) ← NewList
I ← Next Number in ThievesCoresSeq
FirstGroup ← FirstGroup - 1
SecondGroup = 0 Or
SecondGroupValue = 0
Yes
No
NewList ← Copy the SecondGroupValue
Threads from the list TempList
Remove the SecondGroupValue Threads from
the list TempList
MainList (I) ← NewList
I ← Next Number in ThievesCoresSeq
SecondGroup ← SecondGroup - 1
RETURN
(MainList,ThievesCoresSeq)

Figure 4.17: The InOrderMultiStealing Strategy (IOMSS) where the InOGetVictimSeq and GetThievesCoresSequences functions are already shown in Figure 4.16

4.2.3 The RichestFirstSingleStealing Strategy (RFSSS)

The mechanism of the RFSSS (Figure 4.18) is similar to the IOSSS (Figure 4.16). However, it differs in the way of finding the victim core. First, a search is conducted to cover all the victim cores. The chosen core is the one with the highest (richest) number of threads. The function *RichestFirstGetVictimSeq* is in charge of searching for the wealthiest core and then returns its number. This function checks all the cores one by one. The function returns the value zero if all the cores have one or zero threads; otherwise, the function returns the number of core that has highest number of threads.

The mechanism of this strategy outperforms its counterpart (IOSSS). This is based on the probability of having so many threads in the richest core to the extent that it meets the needs of all the thief cores. As a result, dealing with richest core instead of picking the first encountered victim core will definitely achieve better performance since the richest core may satisfy the desire of the thief cores. Or at least, dealing with richest core will reduce the number of calling the HLS in rebalancing the workload. On the other hand, the process of searching for the richest core takes time. Nevertheless, this process will not exceed O(n), where n is the number of cores.

4.2.4 The RichestFirstMultiStealing Strategy (RFMSS)

This strategy is a combination of the two strategies: RFSSS and IOMSS strategies. The way of finding the victim core is adopted from the RFSSS while the way of stealing and all its calculations have been taken from IOMSS. The power of this combination relies on two factors: First, providing a better way to find the victim core and not to rely on the in-order searching manner. Second, stealing more than one thread and let each thief core get a fair share of threads. Consequently, combining these two factors creates a better way of scheduling. Figure 4.19 shows the mechanism of the RFMSS.

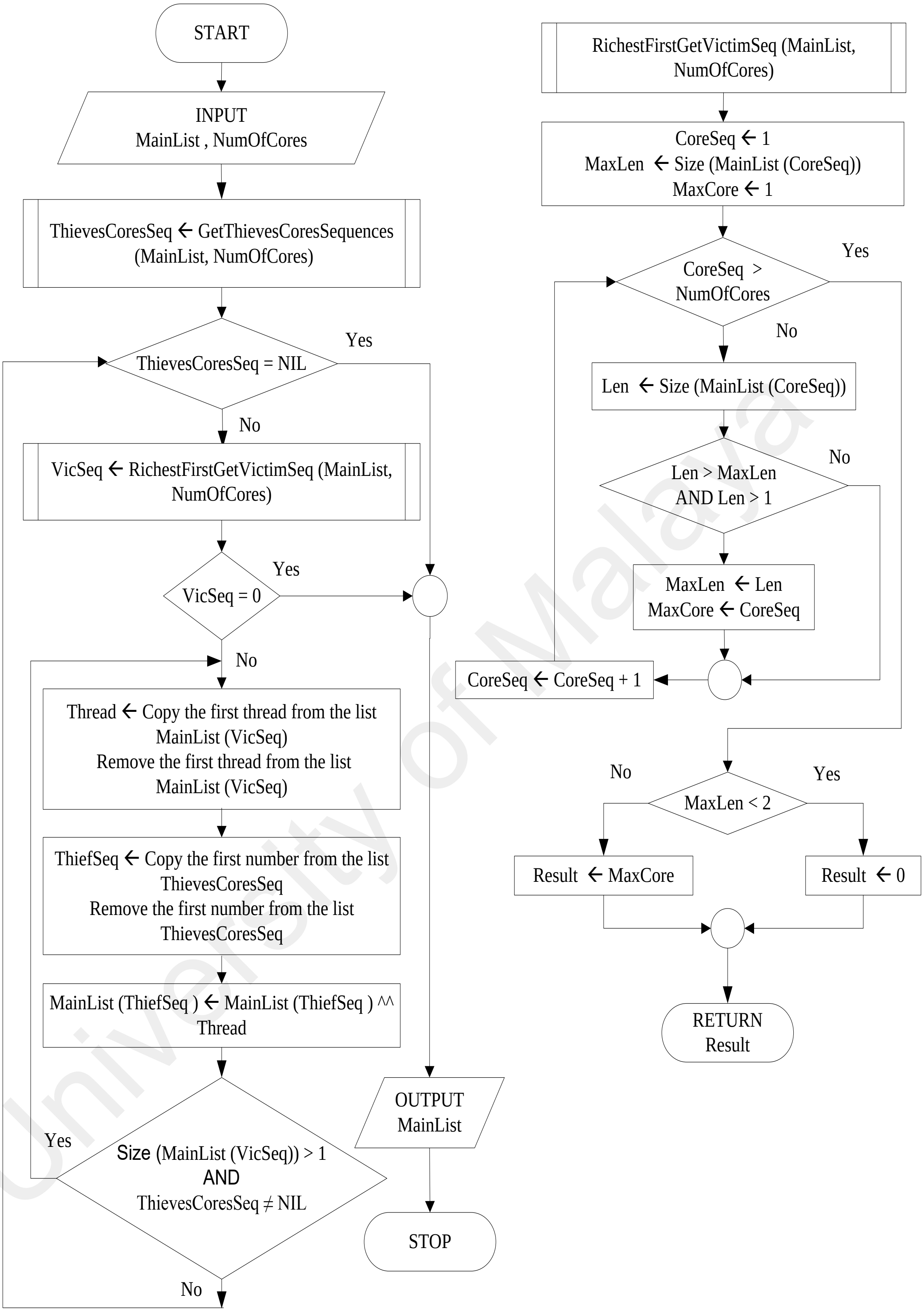

START
INPUT
MainList , NumOfCores
ThievesCoresSeq ← GetThievesCoresSequences (MainList, NumOfCores)
ThievesCoresSeq = NIL
Yes
No
VicSeq ← RichestFirstGetVictimSeq (MainList, NumOfCores)
VicSeq = 0
Yes
No
Thread ← Copy the first thread from the list MainList (VicSeq)
Remove the first thread from the list MainList (VicSeq)
ThiefSeq ← Copy the first number from the list ThievesCoresSeq
Remove the first number from the list ThievesCoresSeq
MainList (ThiefSeq) ← MainList (ThiefSeq) ^^ Thread
Size (MainList (VicSeq)) > 1 AND ThievesCoresSeq ≠ NIL
Yes
No
OUTPUT
MainList
STOP
RichestFirstGetVictimSeq (MainList, NumOfCores)
CoreSeq ← 1
MaxLen ← Size (MainList (CoreSeq))
MaxCore ← 1
CoreSeq > NumOfCores
Yes
No
Len ← Size (MainList (CoreSeq))
Len > MaxLen AND Len > 1
No
MaxLen ← Len
MaxCore ← CoreSeq
CoreSeq ← CoreSeq + 1
MaxLen < 2
No
Yes
Result ← MaxCore
Result ← 0
RETURN
Result

Figure 4.18: The RichestFirstSingleStealing Strategy (RFSSS) where the function GetThievesCoresSeq is already illustrated in Figure 4.16

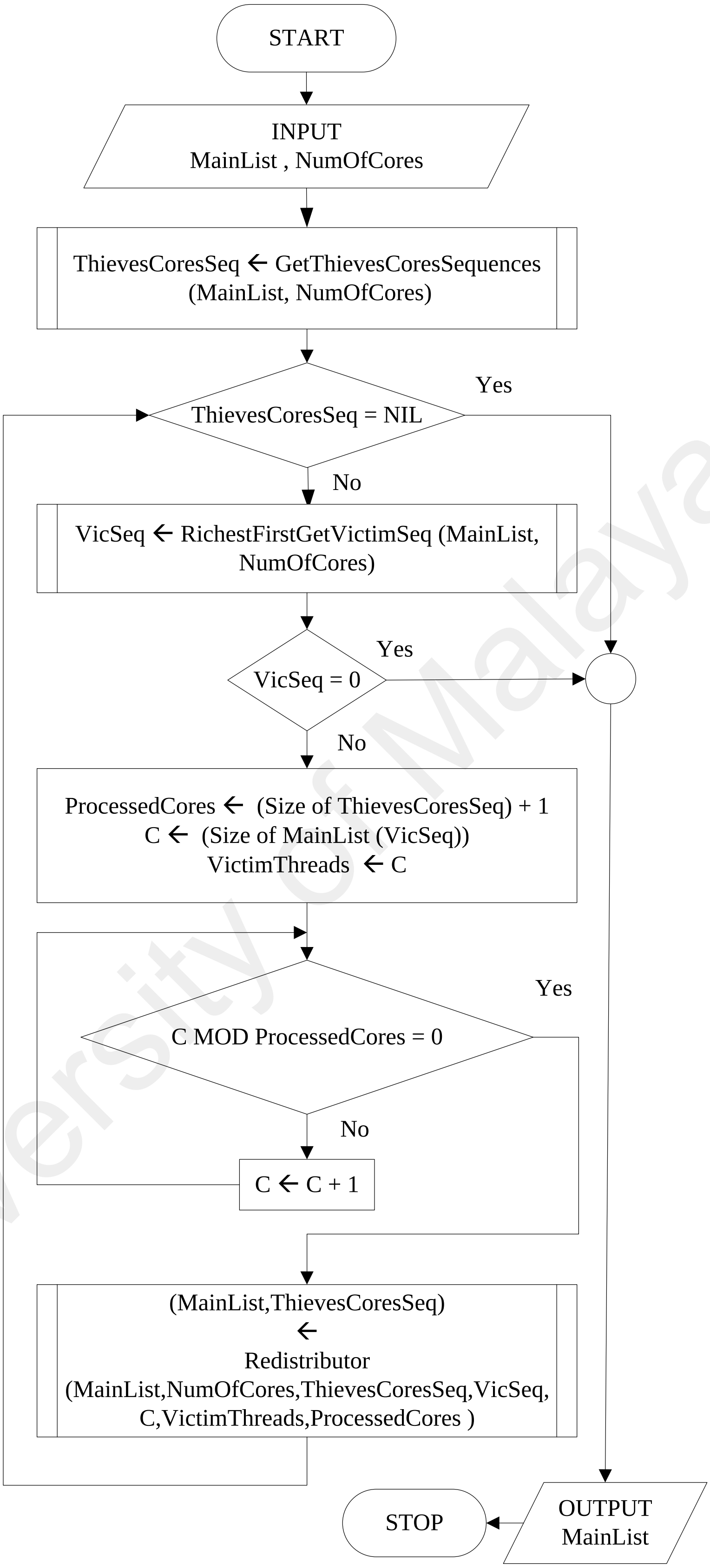

START
INPUT
MainList , NumOfCores
ThievesCoresSeq ← GetThievesCoresSequences (MainList, NumOfCores)
ThievesCoresSeq = NIL
Yes
No
VicSeq ← RichestFirstGetVictimSeq (MainList, NumOfCores)
VicSeq = 0
Yes
No
ProcessedCores ← (Size of ThievesCoresSeq) + 1
C ← (Size of MainList (VicSeq))
VictimThreads ← C
C MOD ProcessedCores = 0
Yes
No
C ← C + 1
(MainList,ThievesCoresSeq)
←
Redistributor
(MainList,NumOfCores,ThievesCoresSeq,VicSeq,
C,VictimThreads,ProcessedCores)
STOP
OUTPUT
MainList

Figure 4.19: The RichestFirstMultiStealing Strategy (RFMSS) where the Redistributor and the RichestFirstGetVictimSeq functions are already illustrated in Figure 4.17 (IOMSS) and Figure 4.18 (RFSSS) respectively

4.2.5 The CompleteMultiStealing Strategy (CMSS)

In the previous strategies, only one victim was in the spotlight. However, in the multicore environment, several cores may be busy at the same time. This motivates the researcher to take advantage of all these victim cores to provide a better way to balance threads distribution among the busy and the idle cores. As the name indicates, the CMSS deals with all the cores. Here, no victim core is excluded from the computations. However, victim cores with single threads will not lose their own threads. They may get extra threads but they will not give up their own threads.

The CMSS (Figure 4.20) calls the following functions:

(a) The *GetVicThie* function (Figure 4.20) is in charge of calculating *NumOfVictims ,VictimThreads,* and *NumOfThieves.* This function serially checks the list *MainLis*. The function checks the number of threads in each core. If the core is empty, then the variable *NumofThieves* is increased by one. If the core is non-empty, then the variable *NumOfVictims* is increased by one and this number of threads is accumulated in the variable *VictimThreads*

(b) The *CMSBalancer* (Figure 4.21) has the duty of redistributing the threads in all the cores. It performs the following actions:

I- Calculating the values of *ProcessedCores, C, VictimThreads, FirstGroup, SecondGroup, FirstGroupValue,* and *SecondGroupValue* through the *Calculations* function (Figure 4.21).

II- Extracting the extra threads from the first and second groups of victim cores through *GetExtraThreads* (Figure 4.22).

III- Updating the two groups of threads through using *UpdateGroups* function (Figure 4.23).

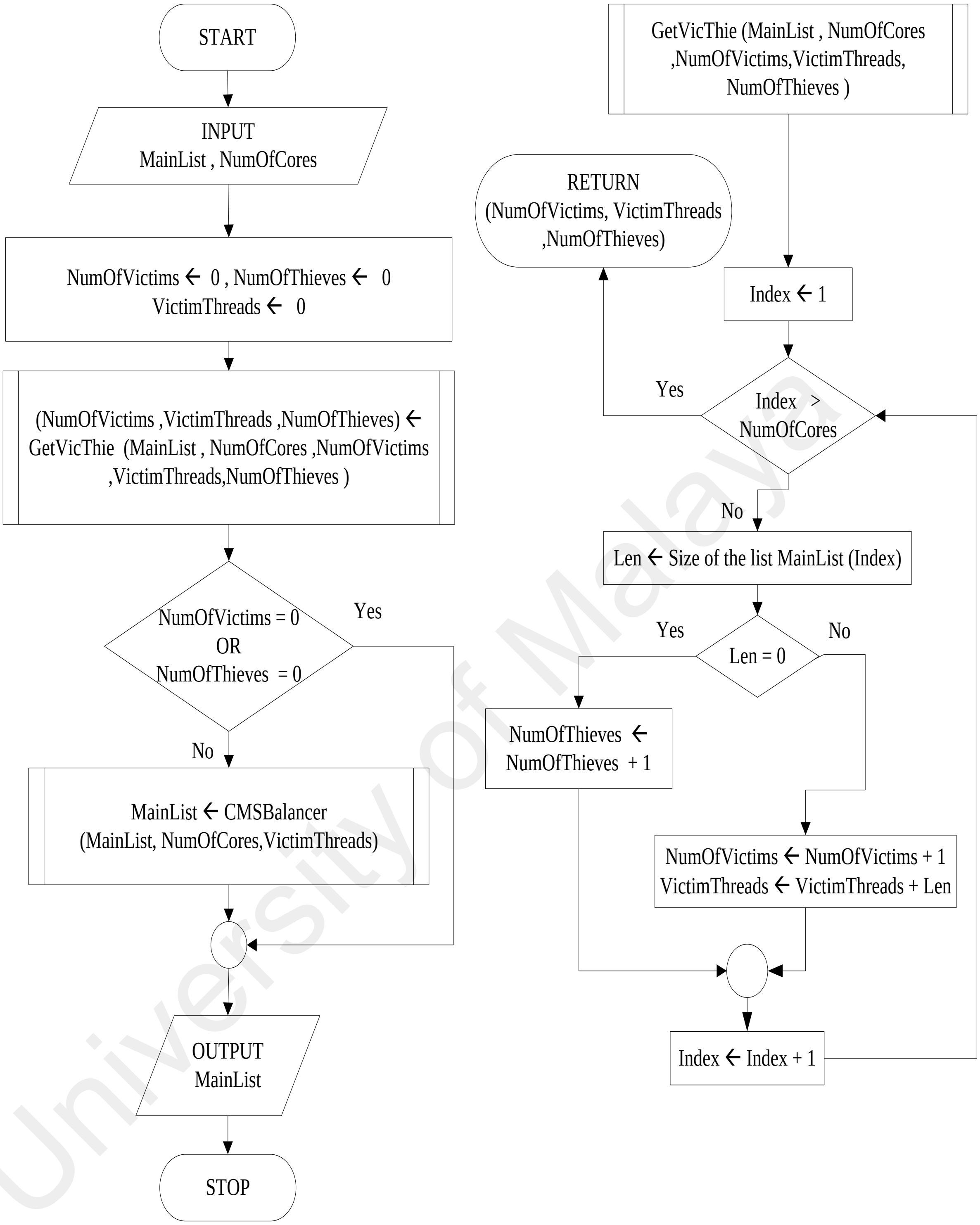

START
INPUT
MainList , NumOfCores
NumOfVictims ← 0 , NumOfThieves ← 0
VictimThreads ← 0
(NumOfVictims ,VictimThreads ,NumOfThieves) ← GetVicThie (MainList , NumOfCores ,NumOfVictims ,VictimThreads,NumOfThieves)
NumOfVictims = 0
OR
NumOfThieves = 0
Yes
No
MainList ← CMSBalancer (MainList, NumOfCores,VictimThreads)
OUTPUT
MainList
STOP
GetVicThie (MainList , NumOfCores ,NumOfVictims,VictimThreads, NumOfThieves)
RETURN
(NumOfVictims, VictimThreads ,NumOfThieves)
Index ← 1
Yes
Index > NumOfCores
No
Len ← Size of the list MainList (Index)
Yes
Len = 0
No
NumOfThieves ← NumOfThieves + 1
NumOfVictims ← NumOfVictims + 1
VictimThreads ← VictimThreads + Len
Index ← Index + 1

Figure 4.20: The CompleteMultiStealing Strategy (CMSS) with its function GetVicThie

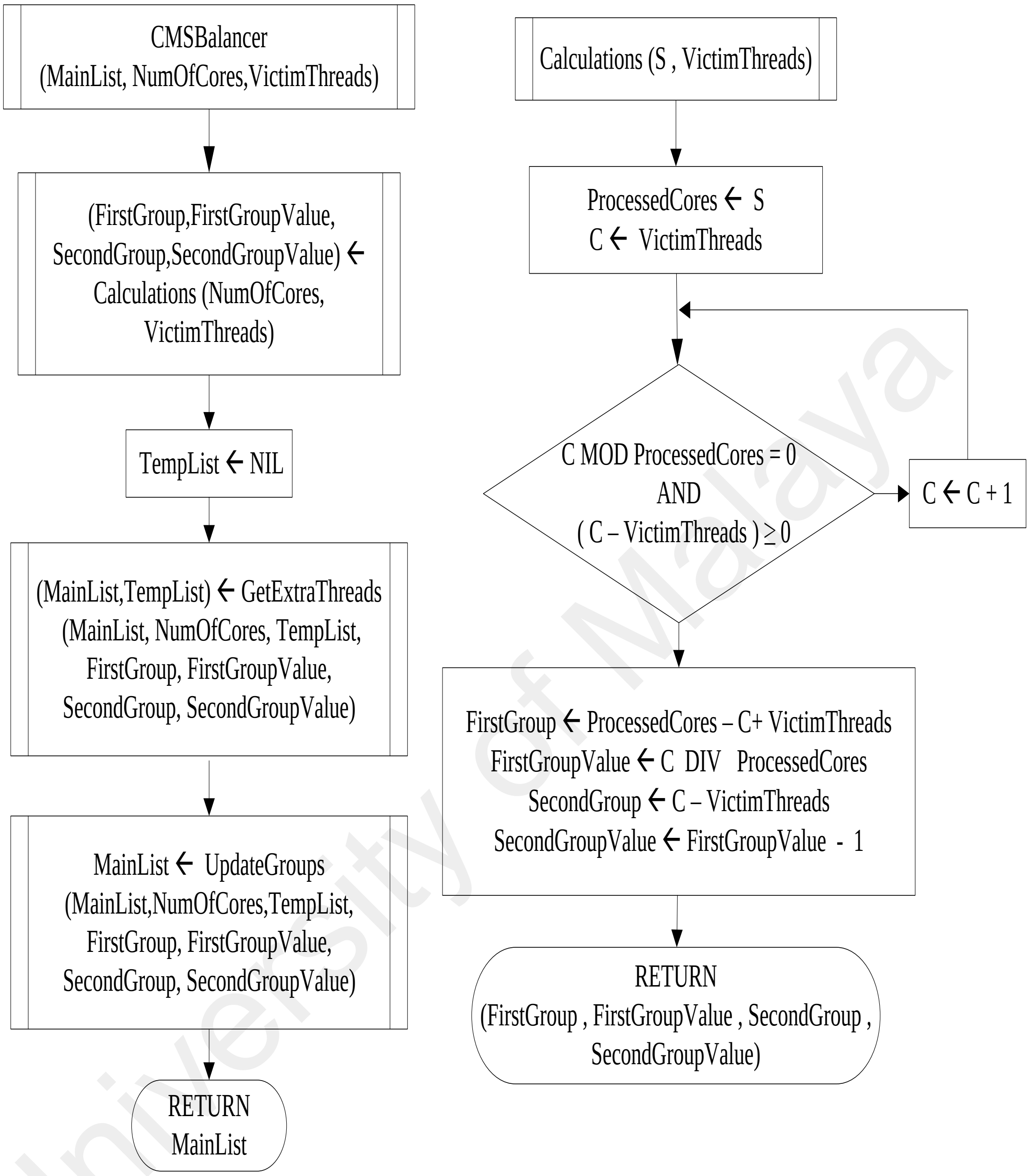

CMSBalancer
(MainList, NumOfCores,VictimThreads)
(FirstGroup,FirstGroupValue, SecondGroup,SecondGroupValue) ← Calculations (NumOfCores, VictimThreads)
TempList ← NIL
(MainList,TempList) ← GetExtraThreads (MainList, NumOfCores, TempList, FirstGroup, FirstGroupValue, SecondGroup, SecondGroupValue)
MainList ← UpdateGroups (MainList,NumOfCores,TempList, FirstGroup, FirstGroupValue, SecondGroup, SecondGroupValue)
RETURN
MainList
Calculations (S , VictimThreads)
ProcessedCores ← S
C ← VictimThreads
C MOD ProcessedCores = 0
AND
(C – VictimThreads) ≥ 0
C ← C + 1
FirstGroup ← ProcessedCores – C+ VictimThreads
FirstGroupValue ← C DIV ProcessedCores
SecondGroup ← C – VictimThreads
SecondGroupValue ← FirstGroupValue - 1
RETURN
(FirstGroup , FirstGroupValue , SecondGroup , SecondGroupValue)

Figure 4.21: The CMSBalancer function with its sub function Calculations

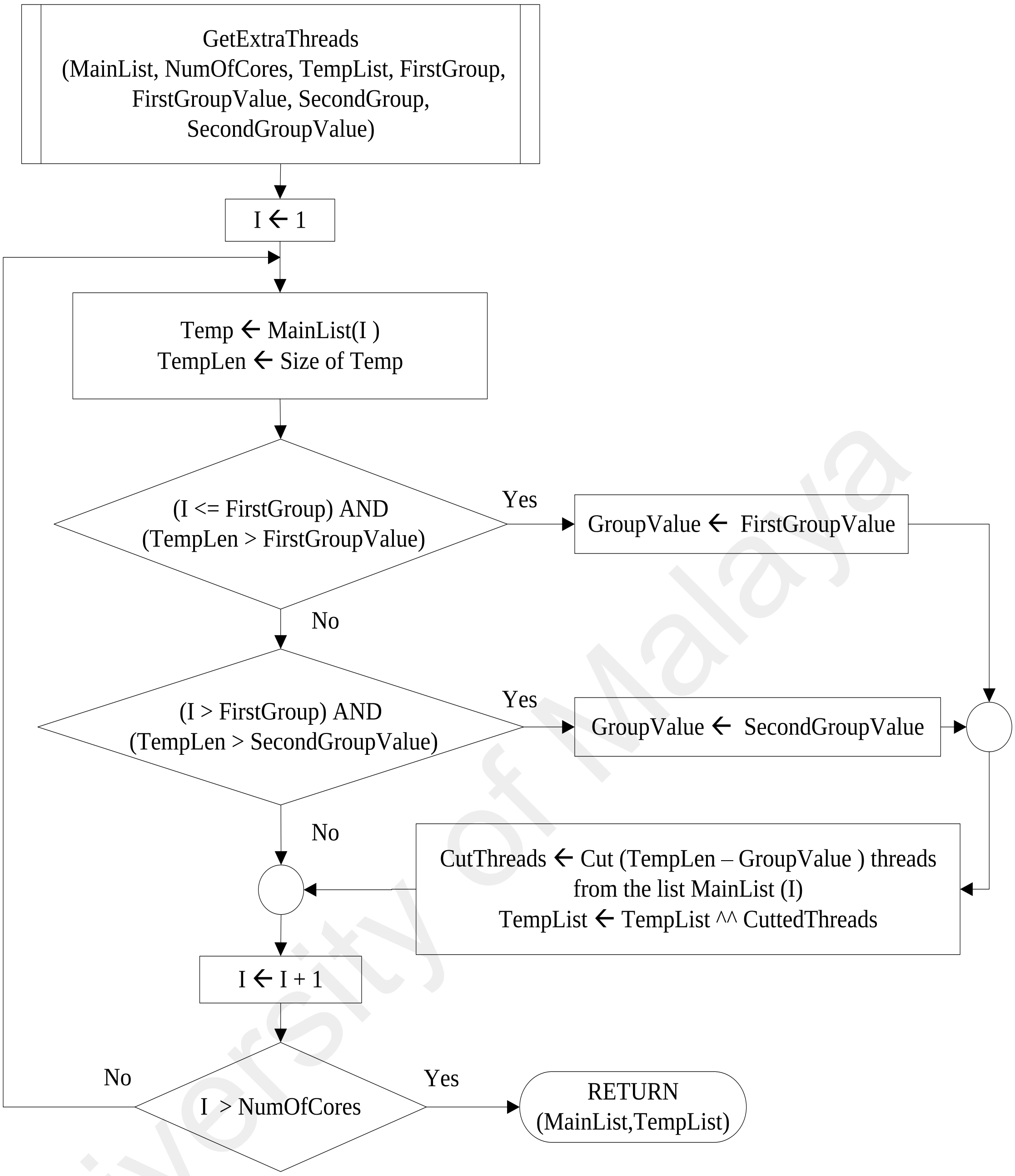

GetExtraThreads
(MainList, NumOfCores, TempList, FirstGroup, FirstGroupValue, SecondGroup, SecondGroupValue)
I ← 1
Temp ← MainList(I)
TempLen ← Size of Temp
(I <= FirstGroup) AND (TempLen > FirstGroupValue)
Yes
GroupValue ← FirstGroupValue
No
(I > FirstGroup) AND (TempLen > SecondGroupValue)
Yes
GroupValue ← SecondGroupValue
No
CutThreads ← Cut (TempLen – GroupValue) threads from the list MainList (I)
TempList ← TempList ^^ CuttedThreads
I ← I + 1
No
I > NumOfCores
Yes
RETURN (MainList,TempList)

Figure 4.22: The GetExtraThreads function

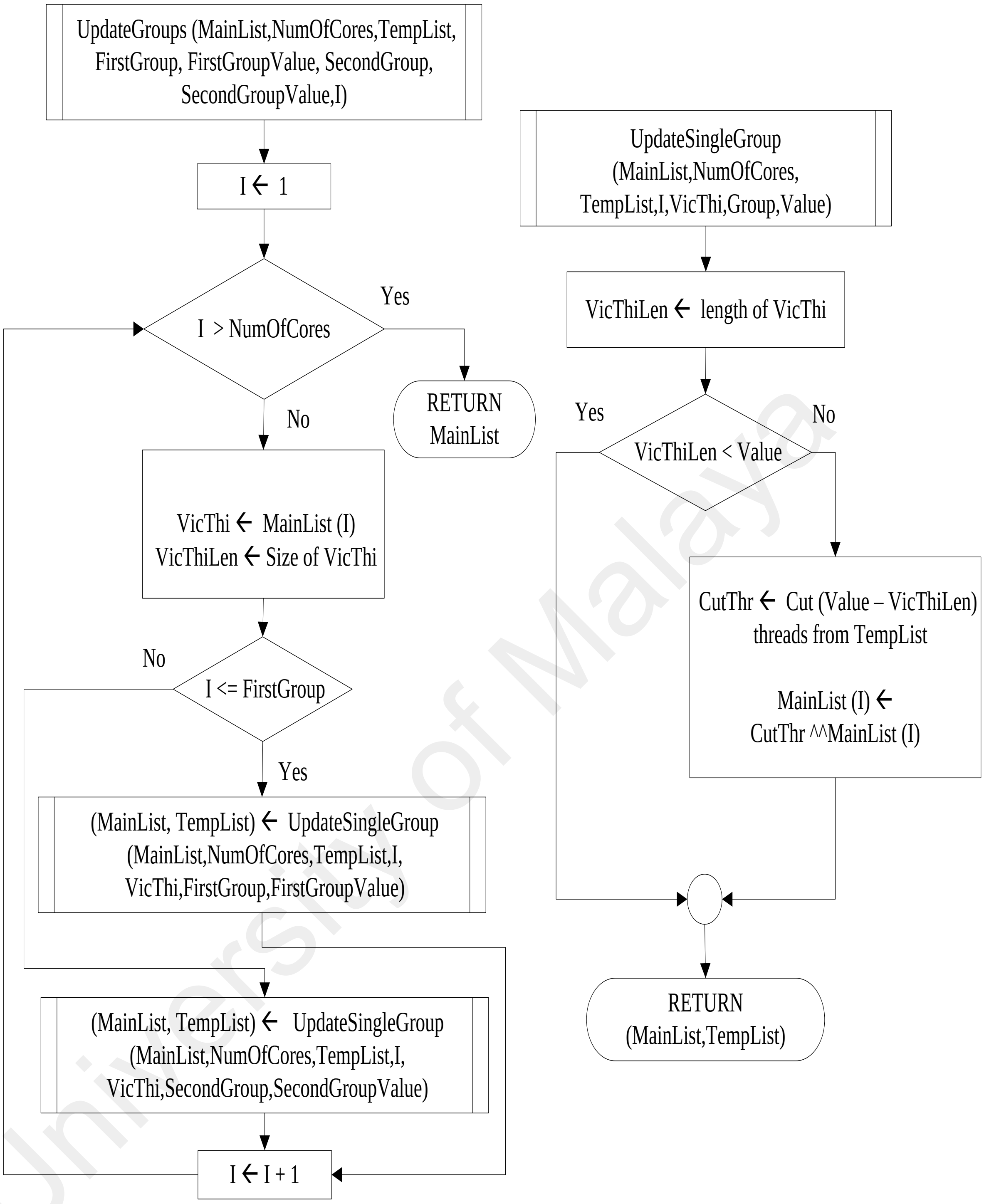

UpdateGroups (MainList,NumOfCores,TempList, FirstGroup, FirstGroupValue, SecondGroup, SecondGroupValue,I)
I ← 1
I > NumOfCores
Yes
RETURN MainList
No
VicThi ← MainList (I)
VicThiLen ← Size of VicThi
No
I <= FirstGroup
Yes
(MainList, TempList) ← UpdateSingleGroup (MainList,NumOfCores,TempList,I, VicThi,FirstGroup,FirstGroupValue)
(MainList, TempList) ← UpdateSingleGroup (MainList,NumOfCores,TempList,I, VicThi,SecondGroup,SecondGroupValue)
I ← I + 1
UpdateSingleGroup (MainList,NumOfCores, TempList,I,VicThi,Group,Value)
VicThiLen ← length of VicThi
Yes
No
VicThiLen < Value
CutThr ← Cut (Value – VicThiLen) threads from TempList
MainList (I) ← CutThr ^^MainList (I)
RETURN (MainList,TempList)

Figure 4.23: The UpdateGroups function with its sub function UpdateSingleGroup

4.2.6 The PartialMultiStealing Strategy (PMSS)

The CMSS deals with all the cores; there is no exemption for anyone of them. However, taking into account all the busy cores as victim cores may be considered non practical when these victim cores have few numbers of threads. It is better to leave the core handling its own threads since it is worthless to interrupt the core's work for a trivial number of threads. Therefore, as the name indicates, the PMSS restricts its dealing with the victim cores that have a plenty of threads. To achieve this goal, the researcher added another parameter named *PartialFactor* to the strategy. *PartialFactor* is given the value 3 which means any victim core having less than 3 threads will not join the set of victim cores. The value of the *PartialFactor* is not fixed, it can be changed.

The PMSS mechanism (Figure 4.24) works as follows:

(a) First, the scheduler creates a list called *PartialList*. This list includes victim cores' numbers (those have *PartialFactor* threads or more) and thief cores' numbers. The function *GetVicThiePartialList* (Figure 4.24) is in charge of creating such list, in addition to calculating *NumOfVictims*, *NumOfThieves*, and *VictimThreads*.

(b) Then, the scheduler calls the function *PMSBalance* which is responsible for balancing the threads among those cores in the *PartialList*. This function performs the following actions:

I- As shown in Figure 4.25, the *PMSBalance* function first computes the values *ProcessedCores, C, VictimThreads, FirstGroup, SecondGroup, FirstGroupValue,* and *SecondGroupValue* through the *Calculations* values by calling the function *Calculations* (Figure 4.21).

II- The function *PMSBalance* calls *PartialGetExtraThreads* (Figure 4.25) function which is in charge of extracting extra threads from the victim cores. The

PartialGetExtraThreads function deals only with those cores in the *PartialList*. The function *GetThreads* is dedicated to extract threads from each group of cores.

III- The function *PMSSBalance* calls *PartialUpdateGroups* (Figure 4.26) function which is in charge of redistributing the threads that have been collected in *TempList* to the thief cores.

4.2.7 Discussion Threads Distribution Fairness in CMSS and PMSS

The process of redistribution of threads in both CMSS and PMSS gives a privilege to those cores that come in the front in case of CMSS and those cores that occupy the first locations in the *PartialList* in the case of the PMSS. For the purpose of further clarification: Let V_1 = (2,6,0,1,8,0,0,2,0,6,0,1,13,15) be a set of fourteen cores' sizes, where each member in this set represents the number of threads in every core. To apply PMSS, cores that have single or two threads are excluded assuming that *PartialFactor* ≥ 3. So, V_2 = (6,0,8,0,0,0,6,0,13,15) is a set cores' sizes but only for those cores that have zero threads or more than two threads, that is V_2 = (X_1=6, X_2=0, X_3=8, X_4=0, X_5=0, X_6=0, X_7=6, X_8=0, X_9=13, X_{10}=15).

We have: N = 10, Sum = $\sum_{i=1}^{10} x_i$ → Sum = 48

C = Sum + K, the smallest value of C that achieves (C mod N = 0) is 50

K = C – Sum → K = 2 , H = (C/N) → H = 5

The new value of the first (N – K) variables is H, that is, each of the first 8 variables will get the value 5 while the last two variables get the value 4, that is, V_2 = (5, 5, 5, 5, 5, 5, 5, 5, 4,4) or V_2 = (X_1=5, X_2=5, X_3=5, X_4=5, X_5=5, X_6=5, X_7=5, X_8=5, X_9=4, X_{10}=4).

There is a notable variation in the values of victims' threads before and after redistribution. For instance, X_1 was 6 and then becomes 5 while the value of X_9 and X_{10} were 13 and 15; both went down to 4. There is no problem with thief cores, each has its share. However, there is a state of non-justice among the victim cores. This is why the PMSS is biased since the victim cores that come at the beginning of the *PatialList* exceed those come at the end of the list. To generate a non-biased version of the PMSS, the processing of the *PartialList* should be updated. The *PartialList* in the non-biased version (NonBiasedPMSS) is the list of pairs where each pair consists of (core number, number of threads). Adding any core to this list depends on the number of threads inside this core. Therefore, the wealthy victim cores occupy the first positions in this list. Consequently, any redistribution process will depend on the order of cores in this list. As a result, the wealthiest cores retain the extra threads. The same thing can be said for the CMSS; the distribution process is biased and gives privilege to those cores that occupy the first positions. To generate a non-biased version to the CMSS, the NonBiasedCMSS works in a similar way to the NonBiasedPMSS in creating a list of pairs of cores and their threads' sizes but this time for all the cores instead of a partial number of cores as in NonBiasedPMSS.

Both NonBiasedCMSS and NonBiasedPMSS can be seen as an extension to the CMSS and PMSS respectively. Those non-biased versions provide more justice in threads distribution. This may be of importance when the threads sizes are varied. On the other hand, having one extra thread in a certain number of cores may not be considered an important difference especially when threads sizes are equal and small, in addition to the cost of reordering the cores so that each core gets its fair share of threads. Anyway, both NonBiasedCMSS and NonBiasedPMSS share many functions that have already been described in CMSS and PMSS. Appendix II is dedicated for illustrating the mechanisms of NonBiasedCMSS and NonBiasedPMSS.

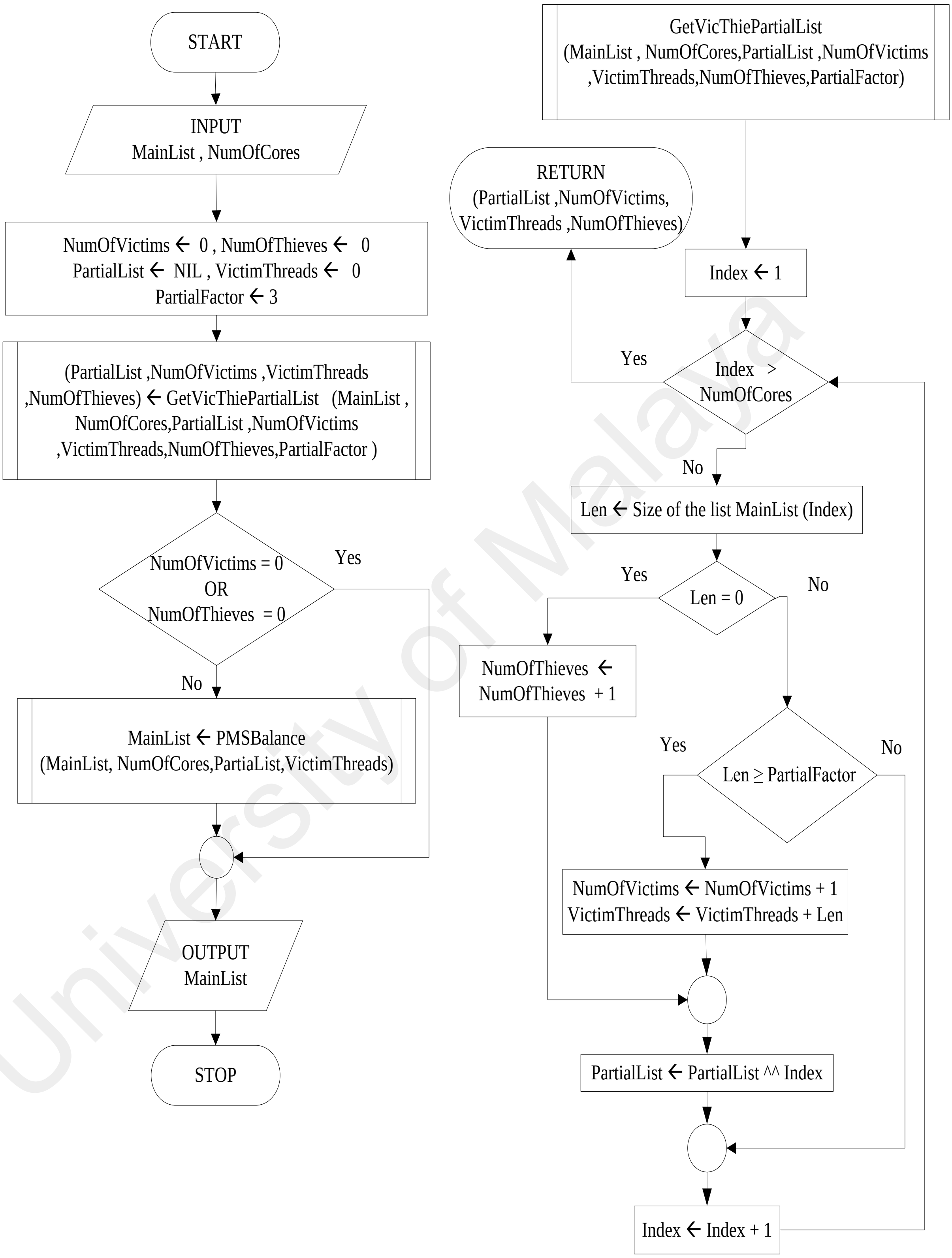

START
INPUT
MainList , NumOfCores
NumOfVictims ← 0 , NumOfThieves ← 0
PartialList ← NIL , VictimThreads ← 0
PartialFactor ← 3
(PartialList ,NumOfVictims ,VictimThreads ,NumOfThieves) ← GetVicThiePartialList (MainList , NumOfCores,PartialList ,NumOfVictims ,VictimThreads,NumOfThieves,PartialFactor)
NumOfVictims = 0
OR
NumOfThieves = 0
Yes
No
MainList ← PMSBalance
(MainList, NumOfCores,PartiaList,VictimThreads)
OUTPUT
MainList
STOP
GetVicThiePartialList
(MainList , NumOfCores,PartialList ,NumOfVictims ,VictimThreads,NumOfThieves,PartialFactor)
RETURN
(PartialList ,NumOfVictims,
VictimThreads ,NumOfThieves)
Index ← 1
Yes
Index >
NumOfCores
No
Len ← Size of the list MainList (Index)
Yes
Len = 0
No
NumOfThieves ←
NumOfThieves + 1
Yes
Len ≥ PartialFactor
No
NumOfVictims ← NumOfVictims + 1
VictimThreads ← VictimThreads + Len
PartialList ← PartialList ^^ Index
Index ← Index + 1

Figure 4.24: The PartialMultiStealing Strategy (PMSS) with its function GetVicThiePartialList

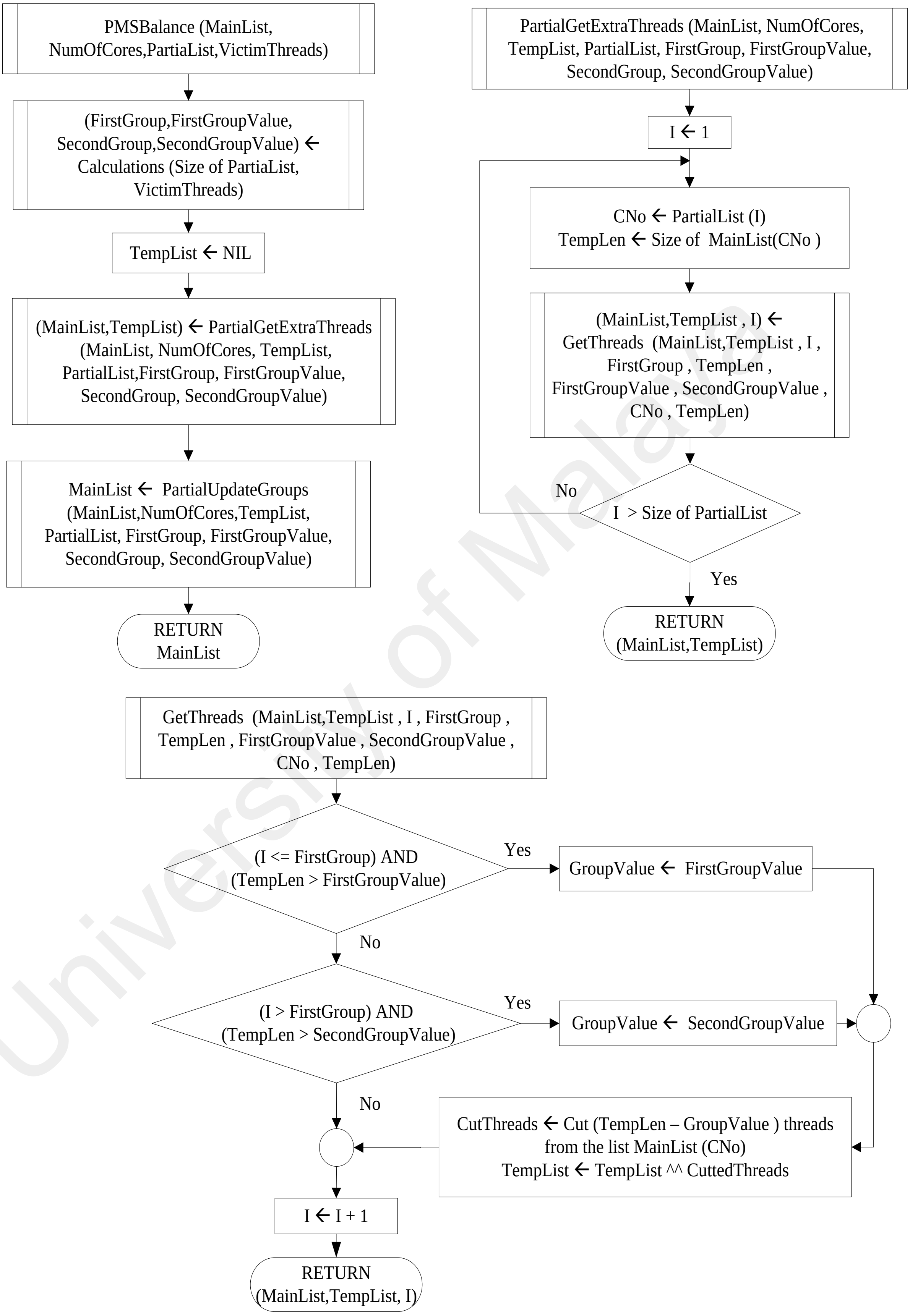

PMSBalance (MainList, NumOfCores,PartiaList,VictimThreads)
(FirstGroup,FirstGroupValue, SecondGroup,SecondGroupValue) ← Calculations (Size of PartiaList, VictimThreads)
TempList ← NIL
(MainList,TempList) ← PartialGetExtraThreads (MainList, NumOfCores, TempList, PartialList,FirstGroup, FirstGroupValue, SecondGroup, SecondGroupValue)
MainList ← PartialUpdateGroups (MainList,NumOfCores,TempList, PartialList, FirstGroup, FirstGroupValue, SecondGroup, SecondGroupValue)
RETURN MainList
PartialGetExtraThreads (MainList, NumOfCores, TempList, PartialList, FirstGroup, FirstGroupValue, SecondGroup, SecondGroupValue)
I ← 1
CNo ← PartialList (I)
TempLen ← Size of MainList(CNo)
(MainList,TempList , I) ← GetThreads (MainList,TempList , I , FirstGroup , TempLen , FirstGroupValue , SecondGroupValue , CNo , TempLen)
No
I > Size of PartialList
Yes
RETURN (MainList,TempList)
GetThreads (MainList,TempList , I , FirstGroup , TempLen , FirstGroupValue , SecondGroupValue , CNo , TempLen)
(I <= FirstGroup) AND (TempLen > FirstGroupValue)
Yes
GroupValue ← FirstGroupValue
No
(I > FirstGroup) AND (TempLen > SecondGroupValue)
Yes
GroupValue ← SecondGroupValue
No
CutThreads ← Cut (TempLen – GroupValue) threads from the list MainList (CNo)
TempList ← TempList ^^ CuttedThreads
I ← I + 1
RETURN (MainList,TempList, I)

Figure 4.25: The PMBalance, PartialGetExtraThreads and GetThreads functions

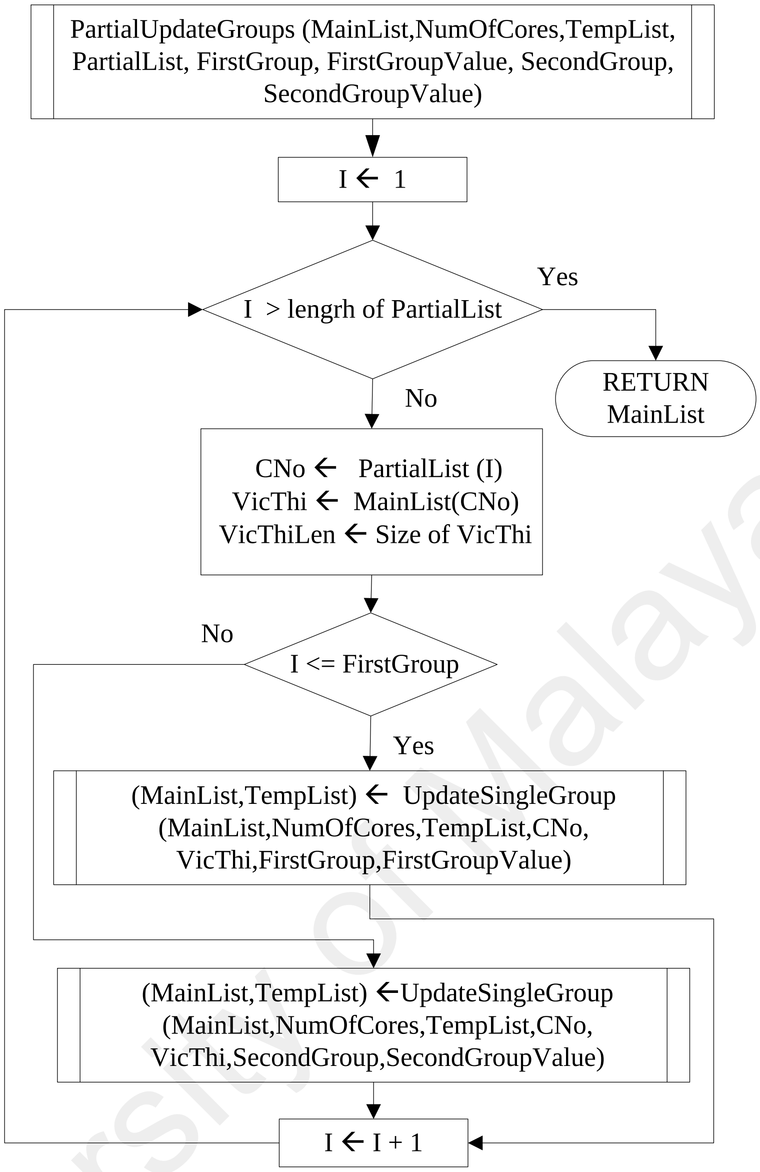

PartialUpdateGroups (MainList,NumOfCores,TempList, PartialList, FirstGroup, FirstGroupValue, SecondGroup, SecondGroupValue)
I ← 1
I > lengrh of PartialList
Yes
RETURN MainList
No
CNo ← PartialList (I)
VicThi ← MainList(CNo)
VicThiLen ← Size of VicThi
No
I <= FirstGroup
Yes
(MainList,TempList) ← UpdateSingleGroup (MainList,NumOfCores,TempList,CNo, VicThi,FirstGroup,FirstGroupValue)
(MainList,TempList) ←UpdateSingleGroup (MainList,NumOfCores,TempList,CNo, VicThi,SecondGroup,SecondGroupValue)
I ← I + 1

Figure 4.26: The PartialUpdateGroups function where the UpdateSingleGroup function is illustrated in Figure 4.23

4.3 Guards' Mechanisms

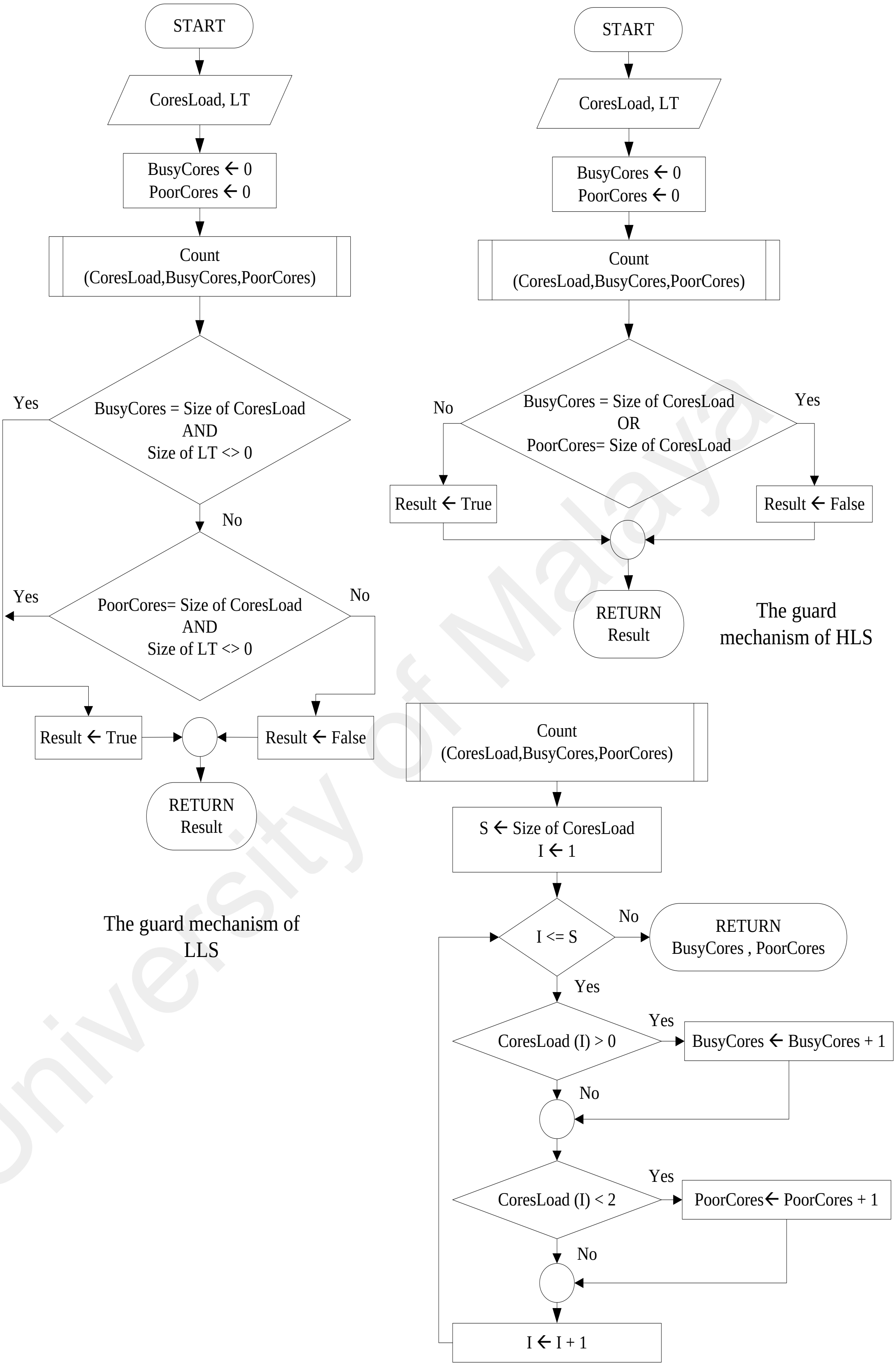
The LLSs have been designed to manage workload partitioning and manipulation while the HLSs are responsible for reallocating the workload among the modeled cores. In general, a guard is a kind of lock that prevents the LLS and the HLS from working when certain conditions happen. The researcher proposes two kinds of guards, one for the LLSs and one for the HLSs. To illustrate the mechanisms of these two guards, the researcher proposes three variables:

(a) BusyCores: This integer variable holds the number of cores that have one or more threads.

(b) PoorCores: It is also an integer variable, however, it counts the number of cores that have less than two threads.

(c) CoresLoad: It is a list of integer numbers representing cores' workload (number of threads) in each core.

Figure 4.27 illustrates the mechanisms of the LLS and HLS guards. In the LLS, the guard first counts the number of BusyCores and PoorCores. Then, the LLS guard enables the LLS to work only if one of the following two conditions is met: First, all the cores are busy. Here, the LLS is allowed to partition and manipulate its own threads since there is no need to interrupt its work to redistribute the threads among the cores because all the cores have work to do. Second, all the cores are poor with threads which means every core has zero or one thread. The guard allows the LLS to work if it has a thread since it does not make sense to freeze its activity when there is no wealthy core to steal from it. On the other hand, the HLS guard is simpler than the LLS guard. The HLS guard enables the HLS when the number of BusyCores does not match the number of cores which means we have victim(s) and thief(s) cores or when PoorCores does not match the number of cores which also means we have victim(s) and thief(s) cores.



The guard mechanism of HLS

The guard mechanism of LLS

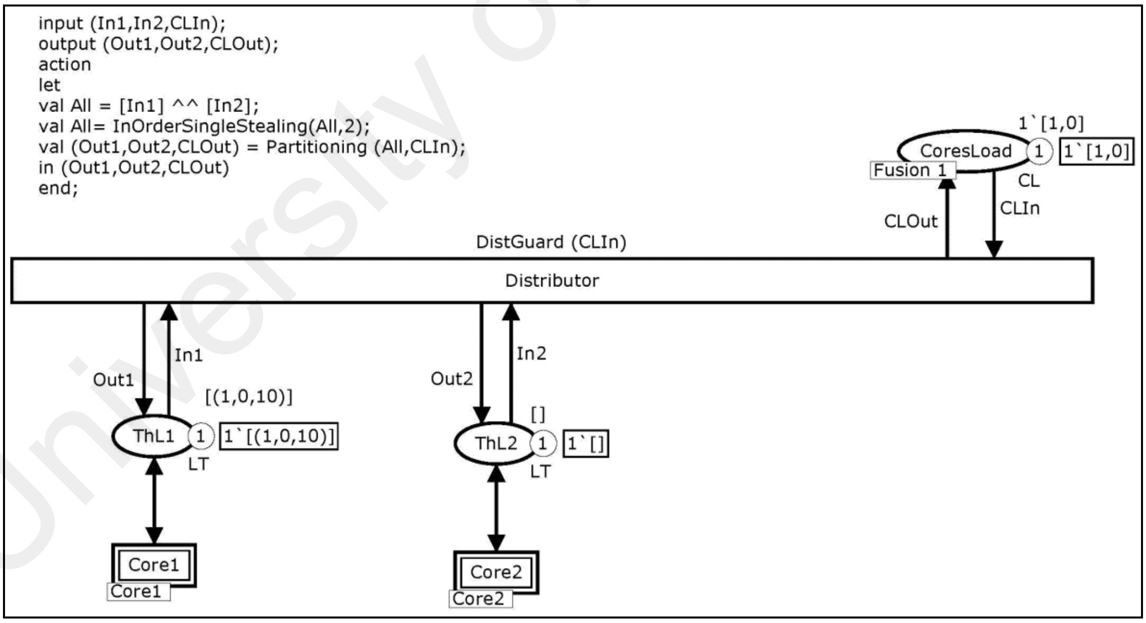
Figure 4.27: The Guard Mechanism

4.4 The CPN models

In this section, the researcher presents the CPN models that solve the D&C problems on a modelled multicore environment. The models apply the LLSs, HLSs, and Guard mechanisms that are previously mentioned in this chapter. For every D&C problem, the researcher designed nine hierarchical CPN models. They are: two-core, three-core, four-core, five-core, six-core, seven-core, eight-core, nine-core, and ten-core models. These nine models work under one of the HLSs.

4.4.1 The CPN Models of Fibonacci Series

The CPN main model for the Fibonacci Series is illustrated in Figure 4.28. This figure shows a hierarchical two-core model with three places (ThL1, ThL2, CoresLoad), one transition (Distributor), and two substituted transitions (Core1 and Core2).



```
input (In1,In2,CLIn);
output (Out1,Out2,CLOut);
action
let
val All = [In1] ^^ [In2];
val All= InOrderSingleStealing(All,2);
val (Out1,Out2,CLOut) = Partitioning (All,CLIn);
in (Out1,Out2,CLOut)
end;
```

Figure 4.28: A two-core CPN main model for solving Fibonacci Series problem

Initially, place ThL1 holds the main thread [(1,0,10)] while place ThL2 is empty ([]). Both places are of type LT which is defined as of type (Int*Int*Int) which indicates ThreadId, FatherId, and Argument, as explained in Figure 4.1. The main thread indicates that the model intends to calculate Fibonacci (10). The places ThL1 and ThL2 exchange threads with transition Distributor through the input parameters (In1, In2) and the output parameters (Out1, Out2).

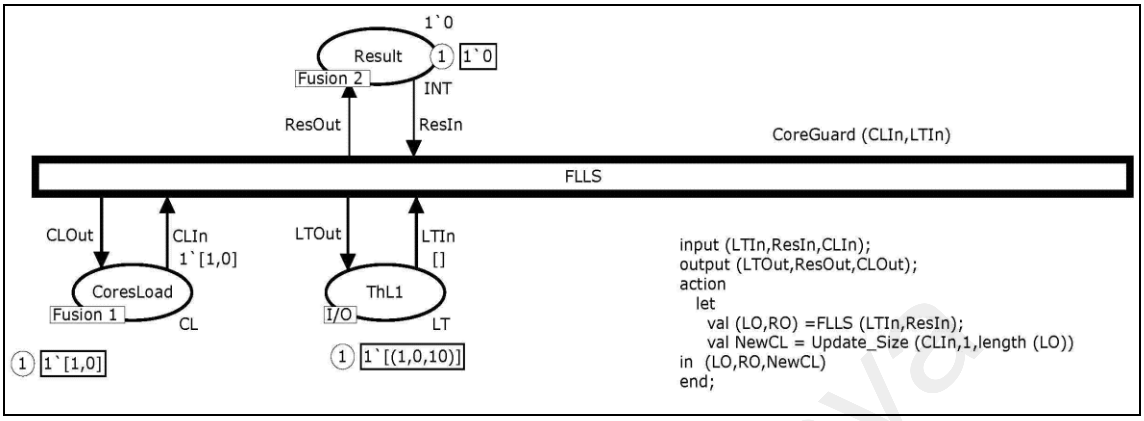
The CoresLoad place holds a list of integers (the type CL is defined as a list of integers) which indicates the current workload of the cores, therefore, initially this list has the value [1,0] since there is a single thread in the first core with null threads in the second one. The CoresLoad place communicates with transition Distributor through the parameters CLIn and CLOut.

The transition Distributor represents the HLS. The code segment of this transition works as follows: it merges the lists of threads from the cores into a single list of lists of threads All. Then it calls the InOrderSingleStealing strategy (IOSSS, Figure 4.16) to redistributes the threads. The Partitioning function separates the All list into sub lists, and assigns each sub list to a core. In addition, the Partitioning function computes the size of threads in each core and saves the results inside place CoresLoad. The transition Distributor has a guard called DistGuard which works as the HLS guard as shown in Figure 4.28.

The main model has two substitution transitions: Core1 and Core2. Figure 4.29 illustrates the contents of Core1 sub model which matches the structure of Core2 except in its threads. In addition, in every sub model there is a common place called Result which stores the results of the computations.

The places CoresLoad in the main and sub models represent one common place and it has a tag called Fusion. Fused places are a set of places that have the same type and data. That is, any change happens to one place is immediately reflected on the other fusion places that share the same fusion number. It is like a global variable that can be changed from the main program or from inside any sub routine. The purpose behind

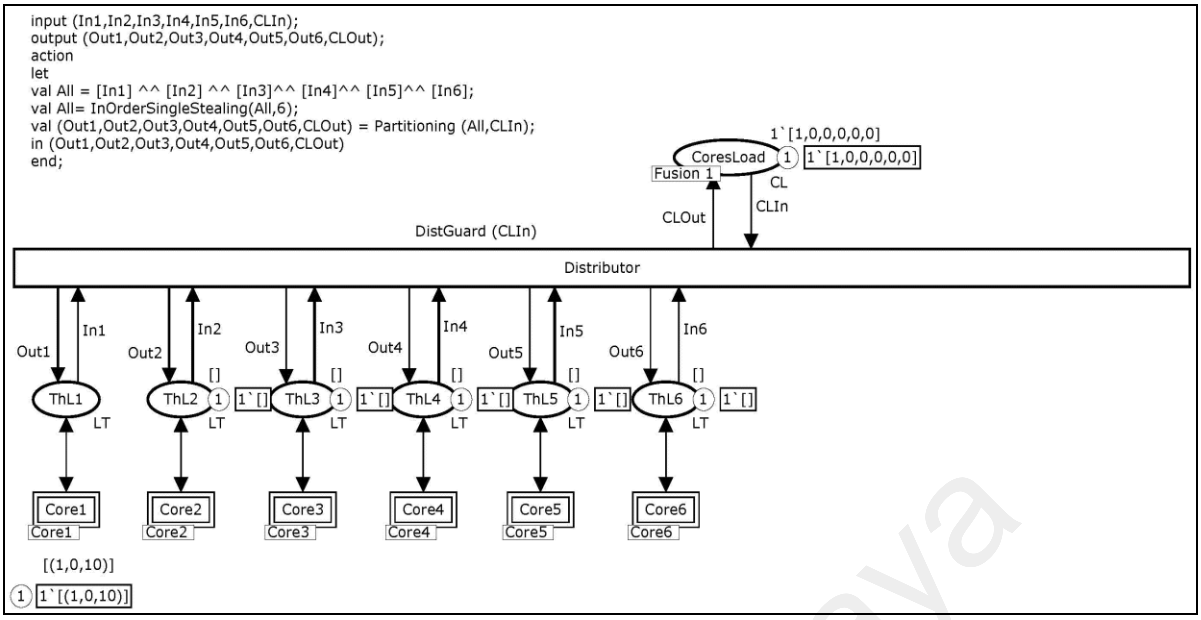
CoresLoad being fused is to let every core update (through the Update_Size function) its own size in the list of cores' sizes that is stored in the CoresLoad place.



```
input (LTIn,ResIn,CLIn);
output (LTOut,ResOut,CLOut);
action
  let
    val (LO,RO) =FLLS (LTIn,ResIn);
    val NewCL = Update_Size (CLIn,1,length (LO))
in  (LO,RO,NewCL)
end;
```

Figure 4.29: A CPN sub model (core model) for solving Fibonacci Series

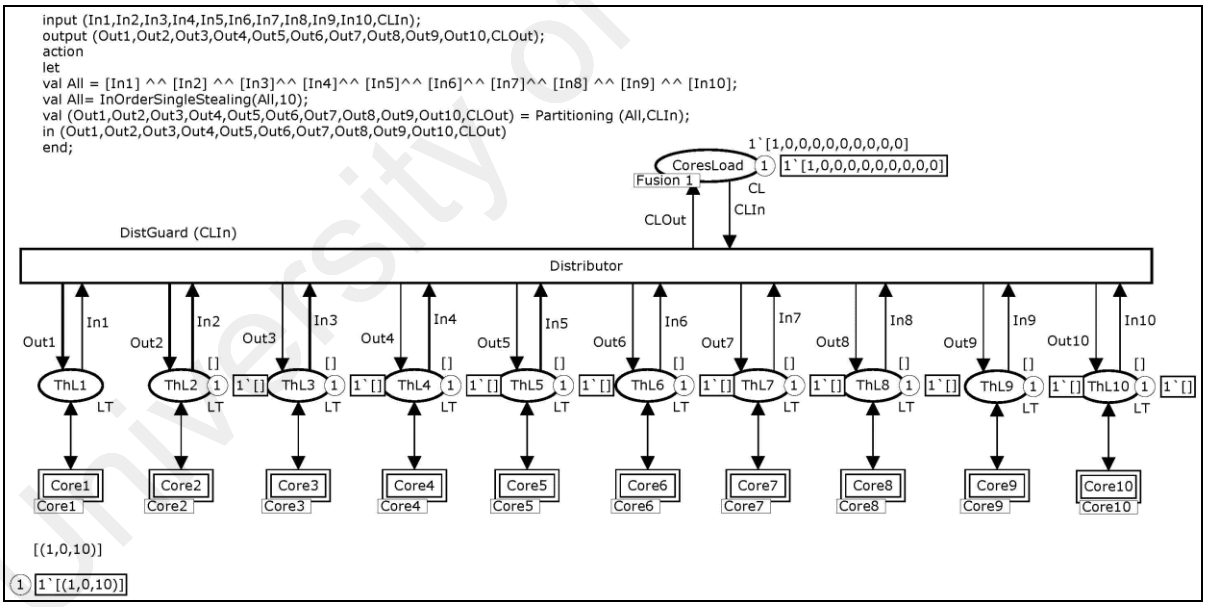
As a result, the CoresLoad is dynamically changed by the LLS inside each core and by the HLS via the transition Distributor. The same thing can be said for the place Result, it is a fused place shared by all the cores (sub models). The Distributor reads the contents of the place Result through ResIn, update it and send it back as ResOut. Figure 4.30 and Figure 4.31 show the CPN models for the same Fibonacci problem being solved on six and ten cores respectively. As can be seen clearly, the CoresLoad place deals with six cores in the first model while the same place deals with ten cores in the second model. All the nine models are dedicated for InOrderSingleStealing strategy (IOSSS) as shown in Figure 4.16.



```
input (In1,In2,In3,In4,In5,In6,CLIn);
output (Out1,Out2,Out3,Out4,Out5,Out6,CLOut);
action
let
val All = [In1] ^^ [In2] ^^ [In3]^^ [In4]^^ [In5]^^ [In6];
val All= InOrderSingleStealing(All,6);
val (Out1,Out2,Out3,Out4,Out5,Out6,CLOut) = Partitioning (All,CLIn);
in (Out1,Out2,Out3,Out4,Out5,Out6,CLOut)
end;
```

1`[1,0,0,0,0,0]
CoresLoad
1
1`[1,0,0,0,0,0]
Fusion 1
CL
CLOut
CLIn
DistGuard (CLIn)
Distributor
Out1 In1 Out2 In2 Out3 In3 Out4 In4 Out5 In5 Out6 In6
[]
ThL1 ThL2 ThL3 ThL4 ThL5 ThL6
1`[]
LT
Core1 Core2 Core3 Core4 Core5 Core6
[(1,0,10)]
1
1`[(1,0,10)]

Figure 4.30: A six-core CPN main model for solving Fibonacci Series problem



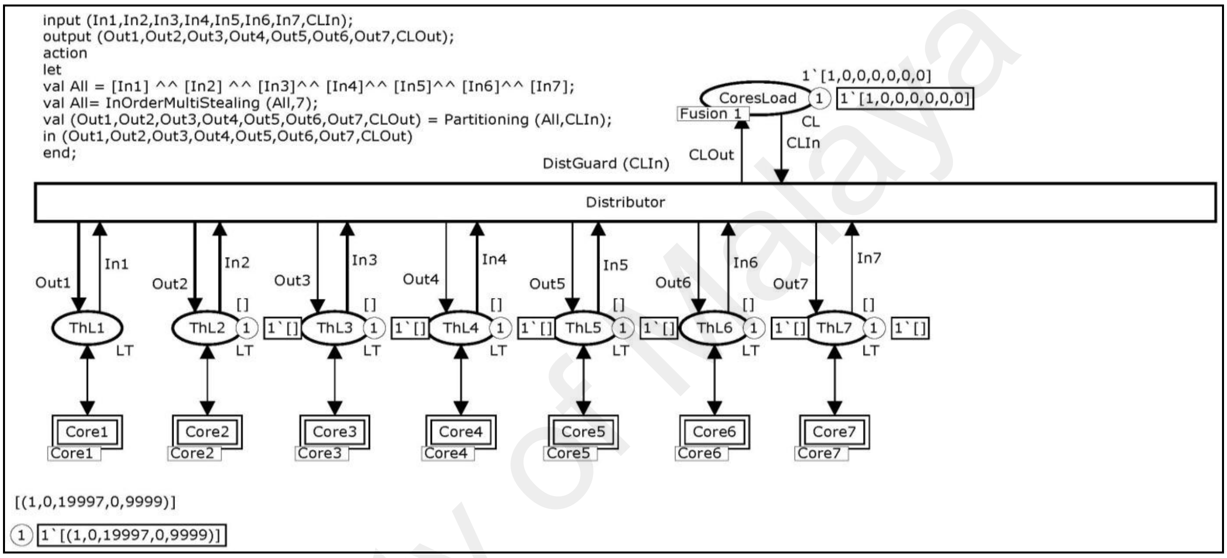
```
input (In1,In2,In3,In4,In5,In6,In7,In8,In9,In10,CLIn);
output (Out1,Out2,Out3,Out4,Out5,Out6,Out7,Out8,Out9,Out10,CLOut);
action
let
val All = [In1] ^^ [In2] ^^ [In3]^^ [In4]^^ [In5]^^ [In6]^^ [In7]^^ [In8] ^^ [In9] ^^ [In10];
val All= InOrderSingleStealing(All,10);
val (Out1,Out2,Out3,Out4,Out5,Out6,Out7,Out8,Out9,Out10,CLOut) = Partitioning (All,CLIn);
in (Out1,Out2,Out3,Out4,Out5,Out6,Out7,Out8,Out9,Out10,CLOut)
end;
```

1`[1,0,0,0,0,0,0,0,0,0]
CoresLoad
1
1`[1,0,0,0,0,0,0,0,0,0]
Fusion 1
CL
CLIn
CLOut
DistGuard (CLIn)
Distributor
Out1 In1 Out2 In2 Out3 In3 Out4 In4 Out5 In5 Out6 In6 Out7 In7 Out8 In8 Out9 In9 Out10 In10
[]
ThL1 ThL2 ThL3 ThL4 ThL5 ThL6 ThL7 ThL8 ThL9 ThL10
1`[]
LT
Core1 Core2 Core3 Core4 Core5 Core6 Core7 Core8 Core9 Core10
[(1,0,10)]
1
1`[(1,0,10)]

Figure 4.31: A ten-core CPN main model for solving Fibonacci Series problem

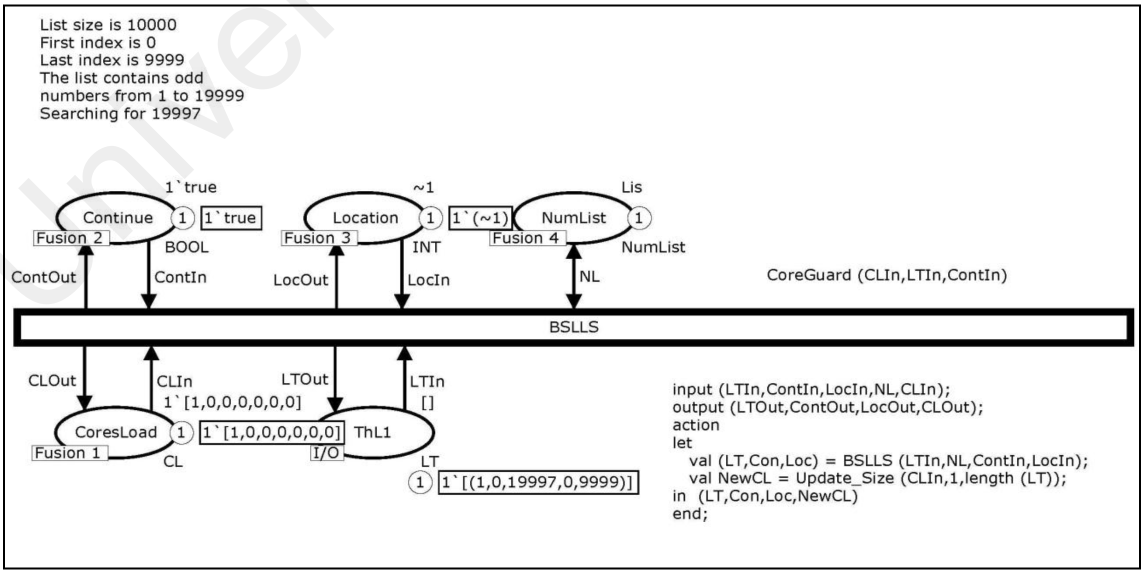
4.4.2 The CPN Models of the Binary Search

For the Binary Search problem, Figure 4.32 shows a sample of a main model that has seven cores. The strategy is InOrderMultiStealing (IOMSS) as shown in Figure 4.17 and the thread structure has been defined in Figure 4.4. The core's model (Figure 4.33) has a place called NumList which contains a list of integer numbers. This list as explained in the comment at the upper left corner consists of ten thousand integer numbers, indexed from 0 to 9999. The search is targeting the number 19997. Figure 4.34 shows the same sub model but with unfolding the content of place NumList. The places Location and Continue are used to store the values Loc and Found values as explained in Figure 4.5.



```
input (In1,In2,In3,In4,In5,In6,In7,CLIn);
output (Out1,Out2,Out3,Out4,Out5,Out6,Out7,CLOut);
action
let
val All = [In1] ^^ [In2] ^^ [In3]^^ [In4]^^ [In5]^^ [In6]^^ [In7];
val All= InOrderMultiStealing (All,7);
val (Out1,Out2,Out3,Out4,Out5,Out6,Out7,CLOut) = Partitioning (All,CLIn);
in (Out1,Out2,Out3,Out4,Out5,Out6,Out7,CLOut)
end;
```

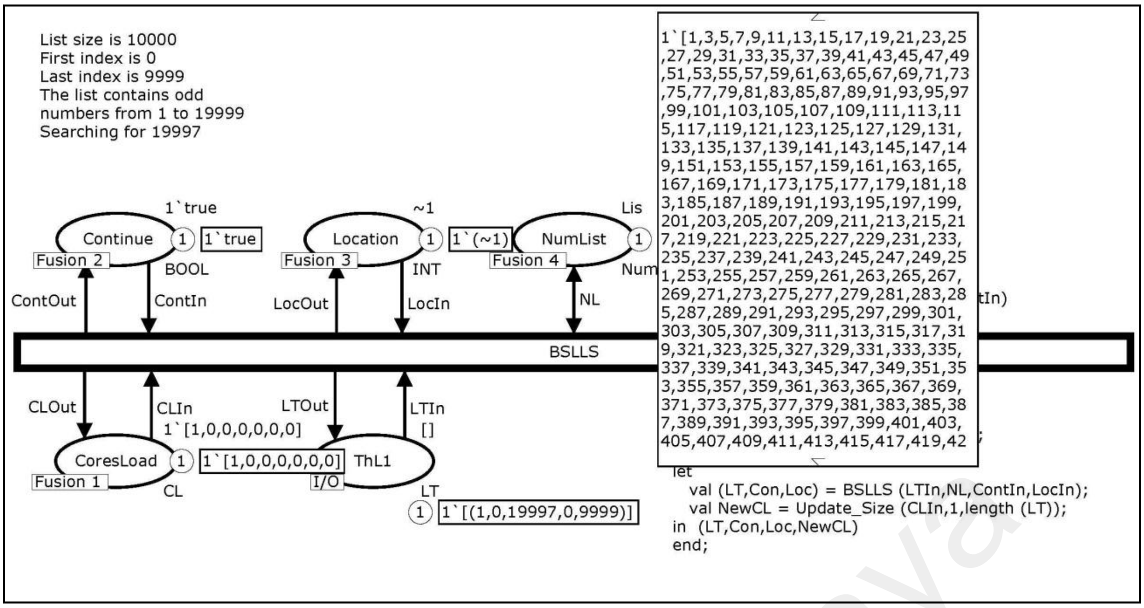
Figure 4.32: A seven-core CPN main model for solving Binary Search problem



List size is 10000
First index is 0
Last index is 9999
The list contains odd
numbers from 1 to 19999
Searching for 19997

```
input (LTIn,ContIn,LocIn,NL,CLIn);
output (LTOut,ContOut,LocOut,CLOut);
action
let
   val (LT,Con,Loc) = BSLLS (LTIn,NL,ContIn,LocIn);
   val NewCL = Update_Size (CLIn,1,length (LT));
in  (LT,Con,Loc,NewCL)
end;
```

Figure 4.33: A CPN sub model (core model) for solving Binary Search problem



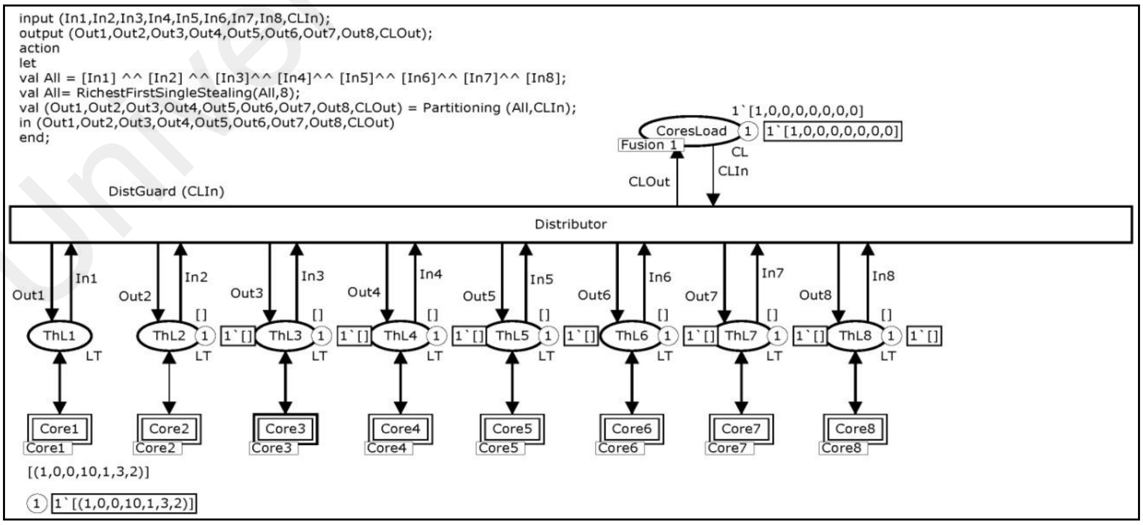
```
1`[1,3,5,7,9,11,13,15,17,19,21,23,25
,27,29,31,33,35,37,39,41,43,45,47,49
,51,53,55,57,59,61,63,65,67,69,71,73
,75,77,79,81,83,85,87,89,91,93,95,97
,99,101,103,105,107,109,111,113,11
5,117,119,121,123,125,127,129,131,
133,135,137,139,141,143,145,147,14
9,151,153,155,157,159,161,163,165,
167,169,171,173,175,177,179,181,18
3,185,187,189,191,193,195,197,199,
201,203,205,207,209,211,213,215,21
7,219,221,223,225,227,229,231,233,
235,237,239,241,243,245,247,249,25
1,253,255,257,259,261,263,265,267,
269,271,273,275,277,279,281,283,28
5,287,289,291,293,295,297,299,301,
303,305,307,309,311,313,315,317,31
9,321,323,325,327,329,331,333,335,
337,339,341,343,345,347,349,351,35
3,355,357,359,361,363,365,367,369,
371,373,375,377,379,381,383,385,38
7,389,391,393,395,397,399,401,403,
405,407,409,411,413,415,417,419,42
```

```
let
   val (LT,Con,Loc) = BSLLS (LTIn,NL,ContIn,LocIn);
   val NewCL = Update_Size (CLIn,1,length (LT));
in  (LT,Con,Loc,NewCL)
end;
```

Figure 4.34: A CPN sub model (core model) unfolding the list of numbers

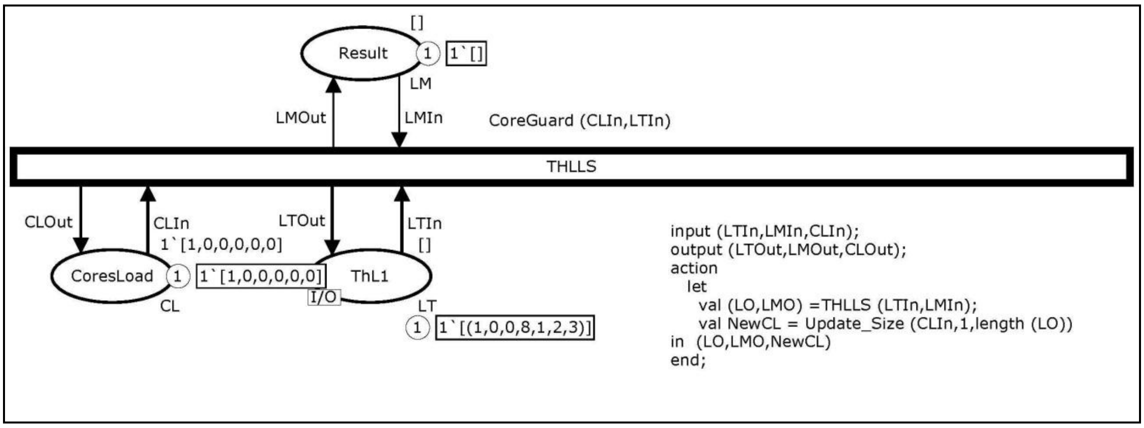
4.4.3 The CPN Models of the Towers of Hanoi

The main model of the Towers of Hanoi problem is shown in Figure 4.35; it is an eight-core model that applies the RichestFirstSingleStealing strategy as previously explained in Figure 4.18. The threads structure is explained previously in Figure 4.8. The Towers of Hanoi sub model is shown in Figure 4.36.



```
input (In1,In2,In3,In4,In5,In6,In7,In8,CLIn);
output (Out1,Out2,Out3,Out4,Out5,Out6,Out7,Out8,CLOut);
action
let
val All = [In1] ^^ [In2] ^^ [In3]^^ [In4]^^ [In5]^^ [In6]^^ [In7]^^ [In8];
val All= RichestFirstSingleStealing(All,8);
val (Out1,Out2,Out3,Out4,Out5,Out6,Out7,Out8,CLOut) = Partitioning (All,CLIn);
in (Out1,Out2,Out3,Out4,Out5,Out6,Out7,Out8,CLOut)
end;
```

Figure 4.35: A CPN eight-core main model for solving the Towers of Hanoi problem

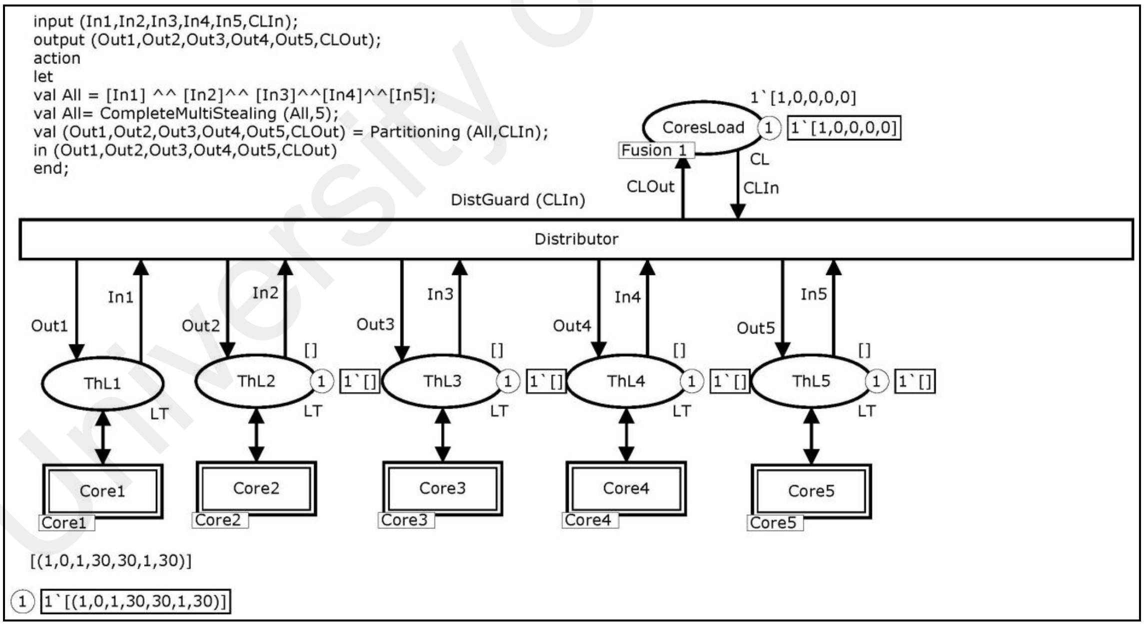


```
input (LTIn,LMIn,CLIn);
output (LTOut,LMOut,CLOut);
action
   let
     val (LO,LMO) =THLLS (LTIn,LMIn);
     val NewCL = Update_Size (CLIn,1,length (LO))
in  (LO,LMO,NewCL)
end;
```

Figure 4.36: A CPN sub model for the Towers of Hanoi problem

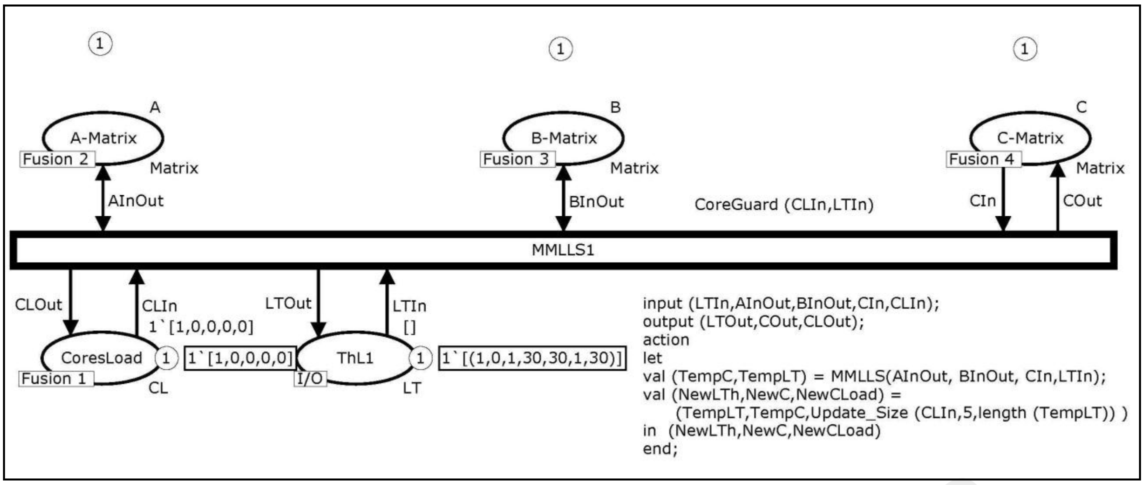
4.4.4 The CPN Models of the Matrix Multiplication

A sample of a main model with five cores is given in Figure 4.37. A sub model with folded matrices' places is illustrated in Figure 4.38 while Figure 4.39 shows the same sub model with unfolded matrices' places.



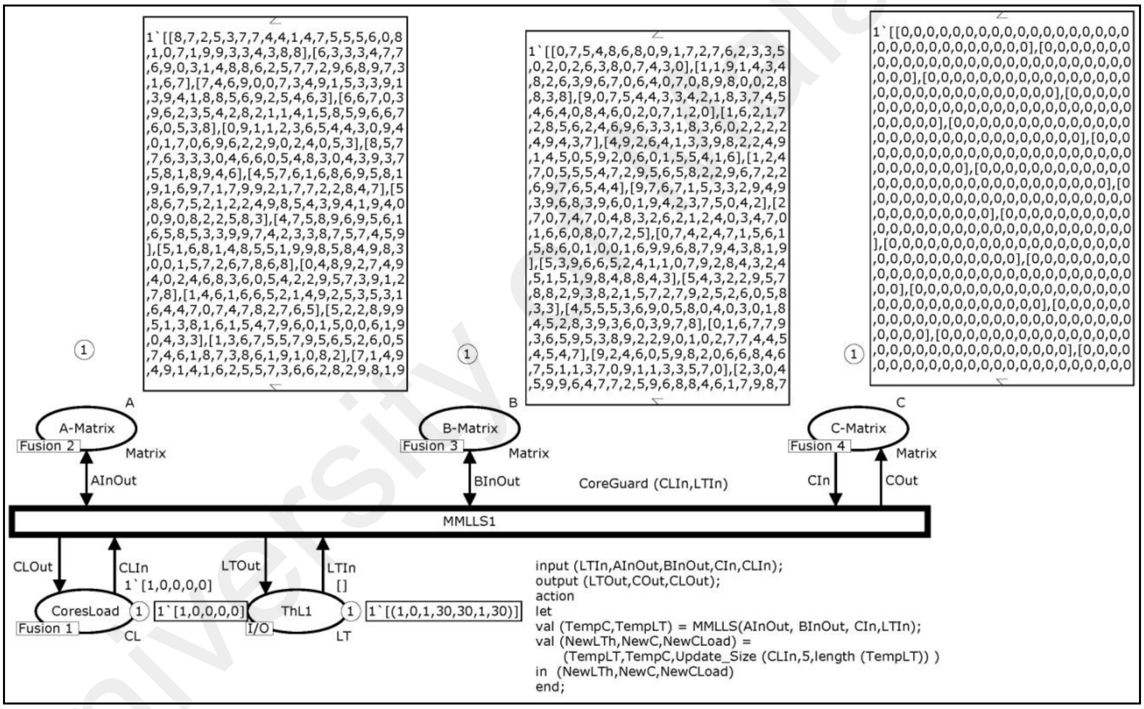
```
input (In1,In2,In3,In4,In5,CLIn);
output (Out1,Out2,Out3,Out4,Out5,CLOut);
action
let
val All = [In1] ^^ [In2]^^ [In3]^^[In4]^^[In5];
val All= CompleteMultiStealing (All,5);
val (Out1,Out2,Out3,Out4,Out5,CLOut) = Partitioning (All,CLIn);
in (Out1,Out2,Out3,Out4,Out5,CLOut)
end;
```

Figure 4.37: A five-core CPN main model for the Matrix Multiplication problem



```
input (LTIn,AInOut,BInOut,CIn,CLIn);
output (LTOut,COut,CLOut);
action
let
val (TempC,TempLT) = MMLLS(AInOut, BInOut, CIn,LTIn);
val (NewLTh,NewC,NewCLoad) =
     (TempLT,TempC,Update_Size (CLIn,5,length (TempLT)) )
in  (NewLTh,NewC,NewCLoad)
end;
```

Figure 4.38: A CPN sub model for the problem of Matrix Multiplication with folded matrices



```
input (LTIn,AInOut,BInOut,CIn,CLIn);
output (LTOut,COut,CLOut);
action
let
val (TempC,TempLT) = MMLLS(AInOut, BInOut, CIn,LTIn);
val (NewLTh,NewC,NewCLoad) =
     (TempLT,TempC,Update_Size (CLIn,5,length (TempLT)) )
in  (NewLTh,NewC,NewCLoad)
end;
```

Figure 4.39: A CPN sub model for the problem of Matrix Multiplication with unfolded matrices

4.5 Summary

This chapter is dedicated for explaining the methodology adopted by the researcher. The researcher proposed two types of schedulers: The Low-Level Schedulers (LLSs) and the High-Level Schedulers (HLSs). Each of the LLSs is dedicated for solving one of the

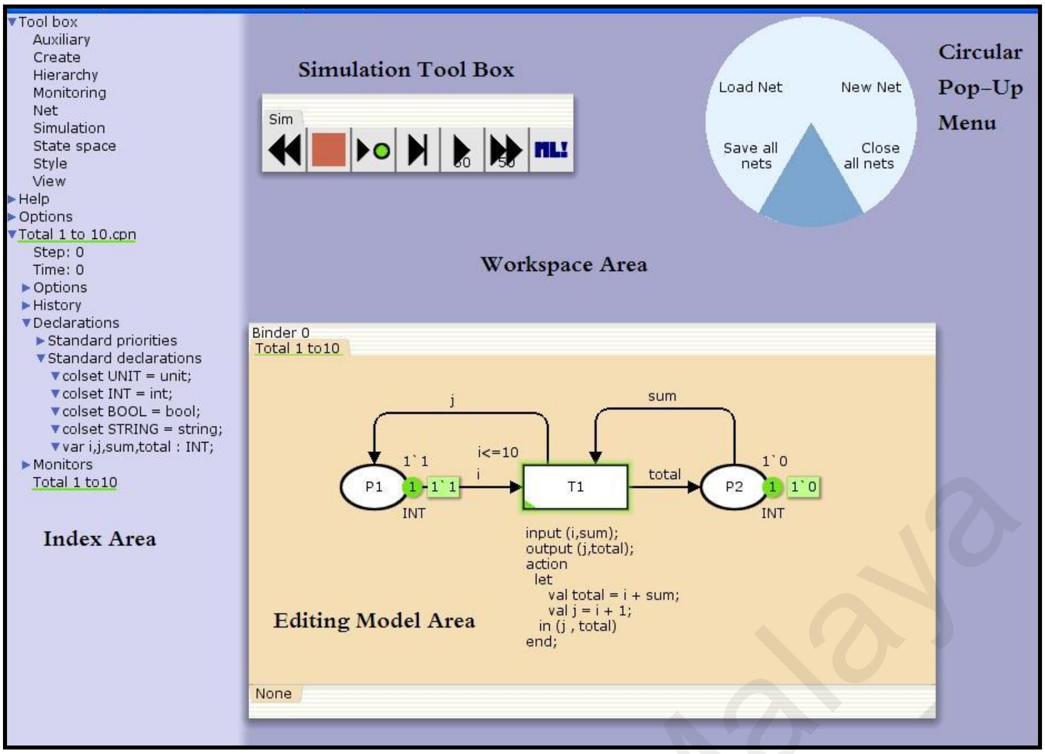
D&C problems: Fibonacci Series, Towers of Hanoi, Binary Search, and Matrix Multiplication. The researcher illustrated the suggested thread, the mechanism of partitioning and computing the threads, and an example for a tree of threads for each D&C problem. On the other hand, the HLSs represent the mechanisms (strategies) suggested by the researcher to redistribute the workload among the cores. The researcher suggested six strategies: The InOrderSingleStealing Strategy, the InOrderMultiStealing Strategy, the RichestFirstSingleStealing Strategy, the RichestFirstMultiStealing Strategy, the CompleteMultiStealing Strategy, and the PartialMultiStealing Strategy. In addition, the researcher demonstrated the Guard's Mechanism which is in charge of activating/deactivating both the LLSs and HLSs. Finally, this chapter showed the CPN models that include the LLSs, HLSs, and the Guards.

CHAPTER 5: SIMULATION RESULTS AND DISCUSSION

5.1 Introduction

Simulation is a representation of a system; it provides a brief but comprehensive way for real system substitution. Another definition of simulation is given by Brown et al. who defined simulation as: "*Provides opportunities to see effects of one's action. Provides some feedback and may develop some intuitive understanding*" (Peterson, 1977). In plain English, a simulation is the process of getting information about the behavior of a system without running the system in reality (Jerry, 1984). The simulation process helps in the detection of defects (inefficiency), getting reliable results, and it certainly saves a great deal of money. In addition, the simulation process explores the impacts of adjustment or changing to the system, makes sure all the system's variables are identified, and it inspires creative thinking. Simulation involves building a model that represents a system. Actually, both modeling and simulation are convergent in meaning. However, modeling is closely related with the abstract representation of the system's reality. Modeling helps in providing formal specification of the concept of the system, assumptions, and constraints. However, simulation is more related with implementation rather than abstraction. In fact, simulation is about implementing the models over time.

To put models in practice, these models have to be implemented; therefore, the development of computer-based software tools that provide the environment to construct and execute these models is a key factor in the success of the modeling and simulation processes. The researcher chooses CPN-Tool since this tool is designed to construct and execute concurrent models. CPN-Tool provides the facility to edit, simulate, monitor, and analyze a concurrent model. A screenshot of the CPN-Tool is given in Figure 5.1.



```
input (i,sum);
output (j,total);
action
 let
   val total = i + sum;
   val j = i + 1;
  in (j , total)
end;
```

Figure 5.1: A screenshot of CPN-Tool

CPN-Tool provides a GUI environment that enables the designer to interact with the design in a simple but effective way. The tool provides menu bars and pull-down menus that facilitate this interaction with the designer. The rectangular area on the left of the screenshot (Figure 5.1) represents the Index area. This area is a holder for the tool boxes that assist the designer to edit, simulate, carry out analysis…etc. The Index area also includes the standard declarations and the designer declarations written in SML (Jensen, et al., 2007). To the right of the Index area, there is the Workspace area. In this area, the designer can edit the CPN models. In addition, the designer can bring tool boxes from the Index area and release them such as the Simulation tool box. Besides, in the Workspace area, it is possible to call a Pop-up menu as shown in Figure 5.1. There are several references (Jensen, et al.; Jensen & Kristensen, 2009) that provide comprehensive information regarding CPN-Tool.

The execution of any CPN model by CPN-Tool consists of two processes that run at the same time. The first one is the simulation process which is compulsory to execute any model, while the second one, the monitoring process, is optional. Through the simulation process, the Tool executes the SML code of the HLSs, LLSs, and the Guards. In other words, the Tool executes the code associated with the transitions in the CPN models. This execution results in redistribution of the threads between the places of the models. The Tool graphically shows this redistribution, transfers the controls between the models' pages (cores), enables the user to interact with the models, etc. On the other hand, the monitoring process is responsible for extracting specific results designated by the user during the simulation process. These results represent the contents of the places that are dynamically changed during the simulation process. Precisely, the researcher focuses on monitoring the CoresLoad place. This place, as defined previously, reflects the current sizes of the threads inside all the cores. The monitoring process registers the content of this place in a text file which is used later to evaluate the performance of the HLSs.

For instance, Table 5.1 shows the results of solving the CPN model of Fibonacci problem on a three-core model using IOSSS for threads distribution. These results represent the CoresLoad values for the model which has been executed three times; first, using eight-thread partitioning; then using four-thread partitioning; and finally, using two-thread partitioning. As noticed in the table, each partitioning method starts with a main thread ([1,0,0]) and ends with the last step of execution where all the cores have zero threads ([0,0,0]).

Table 5.1: The contents of the CoresLoad places for the Fibonacci problem solved on three-core CPN model using eight, four, and two-thread partitioning

Eight-Thread Partitioning	Eight-Thread Partitioning Continue	Four-Thread Partitioning	Four-Thread Partitioning Continue	Two-Thread Partitioning	Two-Thread Partitioning Continue	Two-Thread Partitioning Continue
[1,0,0]	[6,3,2]	[1,0,0]	[7,3,2]	[1,0,0]	[2,3,1]	[1,2,2]
[8,0,0]	[5,3,2]	[4,0,0]	[6,3,2]	[2,0,0]	[2,3,2]	[1,1,2]
[6,1,1]	[5,3,1]	[2,1,1]	[6,6,2]	[1,1,0]	[2,3,3]	[1,1,1]
[6,8,1]	[4,3,1]	[2,1,4]	[6,7,2]	[2,1,0]	[2,3,4]	[0,1,1]
[6,11,1]	[4,2,1]	[2,1,7]	[5,7,2]	[1,1,1]	[2,3,3]	[0,1,0]
[6,11,8]	[4,2,0]	[2,4,7]	[4,7,2]	[1,1,2]	[2,3,2]	[0,2,0]
[13,11,8]	[3,2,1]	[2,7,7]	[4,6,2]	[1,1,3]	[2,3,1]	[1,1,0]
[14,11,8]	[3,2,0]	[5,7,7]	[4,5,2]	[1,2,3]	[3,3,1]	[1,2,0]
[14,11,9]	[2,2,1]	[8,7,7]	[4,5,1]	[1,2,4]	[3,3,2]	[1,1,1]
[14,11,8]	[9,2,1]	[9,7,7]	[3,5,1]	[1,3,4]	[4,3,2]	[1,1,0]
[14,10,8]	[9,2,8]	[8,7,7]	[3,5,2]	[2,3,4]	[4,4,2]	[1,0,0]
[13,10,8]	[8,2,8]	[7,7,7]	[3,5,1]	[3,3,4]	[4,4,1]	[2,0,0]
[12,10,8]	[8,1,8]	[6,7,7]	[3,4,1]	[3,4,4]	[4,4,0]	[1,1,0]
[11,10,8]	[8,0,8]	[5,7,7]	[3,4,0]	[3,4,5]	[3,4,1]	[0,1,0]
[10,10,8]	[7,1,8]	[4,7,7]	[2,4,1]	[4,4,5]	[3,5,1]	[0,2,0]
[10,10,7]	[7,1,7]	[7,7,7]	[2,4,2]	[4,4,6]	[3,5,0]	[1,1,0]
[9,10,7]	[7,1,6]	[7,10,7]	[2,4,1]	[5,4,6]	[2,5,1]	[0,1,0]
[9,10,6]	[6,1,6]	[7,9,7]	[2,4,0]	[5,5,6]	[1,5,1]	[0,0,0]
[9,9,6]	[5,1,6]	[7,8,7]	[1,4,1]	[4,5,6]	[2,5,1]	
[9,8,6]	[4,1,6]	[7,8,8]	[1,4,2]	[4,5,5]	[1,5,1]	
[8,8,6]	[3,1,6]	[6,8,8]	[1,4,1]	[3,5,5]	[0,5,1]	
[8,8,5]	[3,0,6]	[6,8,7]	[1,4,0]	[3,5,4]	[1,4,1]	
[7,8,5]	[2,1,6]	[5,8,7]	[1,3,1]	[3,5,3]	[2,4,1]	
[7,7,5]	[2,0,6]	[4,8,7]	[0,3,1]	[2,5,3]	[2,4,0]	
[7,8,5]	[1,1,6]	[4,8,6]	[1,2,1]	[2,6,3]	[1,4,1]	
[7,8,4]	[4,1,6]	[4,8,5]	[1,5,1]	[2,7,3]	[1,4,0]	
[7,8,3]	[4,0,6]	[4,7,5]	[1,6,1]	[2,7,4]	[1,3,1]	
[7,7,3]	[3,1,6]	[4,6,5]	[0,6,1]	[3,7,4]	[1,4,1]	
[7,6,3]	[3,0,6]	[3,6,5]	[1,5,1]	[3,8,4]	[1,4,0]	
[7,6,2]	[2,1,6]	[6,6,5]	[1,4,1]	[3,7,4]	[1,3,1]	
[6,6,2]	[2,1,5]	[6,6,4]	[0,4,1]	[3,6,4]	[0,3,1]	
[6,7,2]	[2,1,4]	[6,6,3]	[1,3,1]	[2,6,4]	[1,2,1]	
[5,7,2]	[2,0,4]	[6,6,6]	[0,3,1]	[2,5,4]	[1,2,0]	
[5,6,2]	[1,1,4]	[6,6,5]	[1,2,1]	[2,6,4]	[1,1,1]	
[5,6,1]	[0,1,4]	[6,6,4]	[1,2,0]	[2,5,4]	[0,1,1]	
[5,6,0]	[1,1,3]	[5,6,4]	[1,1,1]	[2,5,3]	[0,1,2]	
[4,6,1]	[0,1,3]	[5,6,3]	[0,1,1]	[2,5,2]	[1,1,1]	
[4,5,1]	[1,1,2]	[4,6,3]	[0,4,1]	[2,4,2]	[1,2,1]	
[11,5,1]	[0,1,2]	[4,6,2]	[1,3,1]	[2,4,3]	[2,2,1]	
[12,5,1]	[1,1,1]	[4,6,5]	[1,3,0]	[2,5,3]	[2,3,1]	
[11,5,1]	[1,1,0]	[4,6,4]	[1,2,1]	[2,6,3]	[2,3,0]	
[11,5,8]	[0,1,0]	[3,6,4]	[1,2,0]	[1,6,3]	[1,3,1]	
[11,4,8]	[0,0,0]	[3,7,4]	[1,1,1]	[2,6,3]	[0,3,1]	
[10,4,8]		[3,6,4]	[0,1,1]	[2,6,4]	[1,2,1]	
[10,4,7]		[2,6,4]	[0,1,0]	[3,6,4]	[2,2,1]	
[9,4,7]		[3,6,4]	[0,0,0]	[3,6,3]	[3,2,1]	
[9,5,7]		[3,5,4]		[3,5,3]	[3,2,0]	
[9,5,6]		[3,6,4]		[3,5,2]	[2,2,1]	
[8,5,6]		[3,6,3]		[2,5,2]	[2,3,1]	
[8,5,5]		[2,6,3]		[2,4,2]	[1,3,1]	
[8,5,4]		[1,6,3]		[2,3,2]	[1,3,2]	
[7,5,4]		[1,6,2]		[1,3,2]	[2,3,2]	
[6,5,4]		[1,5,2]		[0,3,2]	[2,3,1]	
[6,5,3]		[4,5,2]		[1,2,2]	[1,3,1]	
[6,4,3]		[7,5,2]		[1,3,2]	[1,3,0]	
[6,3,3]		[7,4,2]		[2,3,2]	[1,2,1]	

As shown in Table 5.1, each CoresLoad value (observation) consists of a list with three integer numbers which indicate the current numbers of threads in the three cores of the model. In all the CPN models, each core has a single transition (LLS), and in addition, each model has a HLS transition which employs threads' distribution. The CPN-Tool randomly picks one of the ready transitions and executes it. At the beginning, the tool has no choice other than executing the transition in the first core since other cores are free of threads. According to the partitioning method, 8, 4, or 2 threads are generated in the first core, following, that the IOSSS redistributes the threads among the cores. Starting from the next step, the choice of any core is non-deterministic, that is, the re-execution of the model will generate different sequence of cores' selection. However, when one of the cores becomes out of threads, the HLS redistributes the threads again. In this case, the Tool has no choice other than picking the HLS's transition since cores' transitions are deactivated by their guards.

In order to compare various results of the simulation and monitoring processes, the researcher proposes a new measurement to evaluate the results obtained from execution of the models. The Average of Execution Steps (AES) is simply the measurement the researcher used to compare between the results of execution of the models. The AES is computed as follows:

AES = Number of Execution Steps / Number of Cores

In the above equation, the Number of Execution Steps is equal to the number of CoresLoad observations starting from the initial state where only a single thread is located in one of the cores through the last observation where all the cores have zero threads. The best AES is the one with the lowest value. The results are sketched as MS Excel graphs (histograms) where the AES along with the number of cores represent the two axes of the histograms.

In Section 5.2, the researcher presents and discusses the results obtained from simulating and monitoring the Fibonacci CPN models. Sections 5.3, 5.4, and 5.5 are dedicated for presenting and discussing the execution of the Towers of Hanoi, Binary Search, and Matrix Multiplication CPN models respectively. Chapter discussion and summary are given in Sections 5.6 and 5.7.

5.2 The Results of Executing the Fibonacci CPN Models

There are two objectives behind the simulation and the monitoring processes: First, computing the results; second, recording the contribution of the threads at the threads' places. The first objective can be achieved through the simulation process while the second objective is achieved through the monitoring process. In the case of Fibonacci series, computing the results is represented by computing Fibonacci (n) where n ≥ 0. The result is computed and sited in the place Result (Figure 4.29). On the other hand, checking the contribution of the threads can be done by monitoring the CoresLoad place. This place, as defined previously, reflects the sizes of the threads' places. For instance, Table 5.1 shows the result of executing a CPN model designed for computing Fibonacci (10) using three cores. The general observation of the table shows that all the methods started with a single thread [1,0,0] in the first core and ended with zero threads [0,0,0] in all the cores. It is also clear that the eight threads partitioning takes fewer steps (least AES). The reason behind this is the ability in one step to generate more threads than the other two methods. As a result, the need for threads redistribution (to satisfy thief cores) becomes less comparing with the case when using four threads partitioning which is in turn less than the two threads partitioning. In the eight threads partitioning, the FLLS divides the threads into eight, four, and two threads; otherwise, the FLLS computes the thread into zero or one as Fibonacci (0) is zero and Fibonacci(1) is one. The same thing can be said for the four threads partitioning except there is no chance to divide threads into eight threads. The worst case is in the two threads partitioning, here, the victim thread has no choice other than creating two threads which are not sufficient to satisfy the hungry thief cores. Therefore, the IOSSS has to repeat the division process more times to please the thief cores, which no doubt causes the loss of time. It is important to note that the limit of threads that are generated in the three ways of partitioning: 2, 4, and 8. In general, since the FLLS generates a binary tree of threads, this tree can be extended to create 16 threads or multiples thereof , this is possible when the argument n in Fibonacci (n) is large enough. However, doing this will increase the time in generating the threads; in other words, the scheduler will waste precious time in dividing threads leaving thief cores in an idle situation. Therefore, to find an intermediate state, the researcher found that eight threads partitioning is the suitable one. Yet, in case if there are several hundreds of cores and there is a large value of n, then it will be more convenient to increase the number of divided threads.

The researcher solved the problem of calculating Fibonacci (10). For this purpose, nine CPN models have been designed: two-core model, three-core model, and so forth until ten-core model. First, the researcher solved the problem using two-thread partitioning. The execution of each model consists of ten trials, in each trial; a new AES is calculated and recorded with other AESs in a text file. Therefore, for the two-core model, there will be ten trials of computing ten AESs, the same thing for three-core model, and so forth until ten-core model. The above has been repeated for four-core and eight-core partitioning. Table 5.2 shows the trials of executions of the CPN models designed for computing Fibonacci (10) and adopting IOSSS as the HLS for threads distribution. Next, the researcher computes the average of every ten trials of each model as illustrated in Table 5.2. Finally, a histogram that shows the results in Table 5.3 is sketched using Ms Excel (Figure 5.2).

Table 5.2: The AES values for the Fibonacci (10) problem using IOSSS as the HLS and eight-thread, four-thread, and two-thread partitioning

Using IOSSS and Two-Thread Partitioning

Cores	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10
2	56.50	57.00	56.50	58.00	55.00	56.50	56.00	59.50	55.00	55.00
3	38.67	39.67	39.33	37.67	37.33	40.67	39.67	40.67	39.00	38.33
4	29.50	31.25	29.25	33.50	30.50	29.75	28.25	29.00	30.75	30.25
5	24.00	25.20	24.60	25.20	23.40	25.80	26.20	24.00	23.40	24.80
6	20.83	23.50	22.50	20.67	22.17	22.50	21.83	20.67	21.17	21.17
7	19.14	17.71	20.14	18.71	19.29	20.00	19.00	18.86	19.00	18.71
8	16.75	16.75	16.88	17.50	15.63	17.63	16.38	17.13	17.25	17.63
9	14.33	15.33	16.33	14.44	15.56	14.22	14.89	14.89	14.56	14.89
10	13.80	13.90	14.70	15.00	13.80	14.00	14.80	14.00	14.10	14.20

Using IOSSS and Four-Thread Partitioning

Cores	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10
2	46.00	47.50	45.50	46.00	47.00	48.50	48.50	49.50	46.00	50.00
3	30.33	32.00	30.00	31.67	31.33	30.00	30.33	32.33	31.00	31.00
4	24.50	23.25	24.50	23.00	23.00	24.25	23.50	24.00	23.75	23.50
5	19.80	21.40	19.80	21.80	19.80	21.00	19.80	19.60	20.80	19.80
6	17.83	16.17	18.00	17.33	17.00	16.33	18.00	16.83	17.50	17.67
7	14.71	14.29	15.43	14.71	15.57	15.86	15.57	16.14	15.43	16.43
8	13.88	13.00	12.38	13.50	14.13	13.38	13.25	13.88	13.13	15.00
9	11.67	13.11	11.67	13.22	12.00	12.33	11.22	11.33	12.56	12.44
10	10.20	10.90	11.30	10.80	10.80	10.10	10.80	10.10	10.40	11.10

Using IOSSS and Eight-Thread Partitioning

Cores	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10
2	47.50	50.00	49.50	45.50	44.50	51.00	47.00	48.00	48.50	48.00
3	32.00	29.33	30.00	34.00	36.00	32.67	32.33	31.33	37.00	33.33
4	22.00	23.00	24.00	22.75	23.75	25.25	23.00	24.25	25.00	23.25
5	18.20	18.80	18.40	19.00	19.60	18.60	19.00	19.80	19.00	19.20
6	15.83	16.83	15.17	16.50	17.00	15.83	17.00	15.50	16.50	15.33
7	13.29	13.29	13.14	13.00	12.43	13.71	13.43	13.14	12.86	14.00
8	11.00	11.13	11.63	11.13	11.75	11.25	11.38	11.63	11.38	11.63
9	10.33	10.22	10.44	11.00	10.44	10.44	10.56	10.67	11.00	11.44
10	9.30	10.30	10.00	10.40	9.50	10.80	9.50	9.90	9.90	9.50

Table 5.3: The averages of the AES values shown in Table 5.2

Cores	Using Two-Thread Partitioning	Using Four-Thread Partitioning	Using Eight-Thread Partitioning
2	56.50	47.45	47.95
3	39.10	31.00	32.80
4	30.20	23.73	23.63
5	24.66	20.36	18.96
6	21.70	17.27	16.15
7	19.06	15.41	13.23
8	16.95	13.55	11.39
9	14.94	12.16	10.66
10	14.23	10.65	9.91

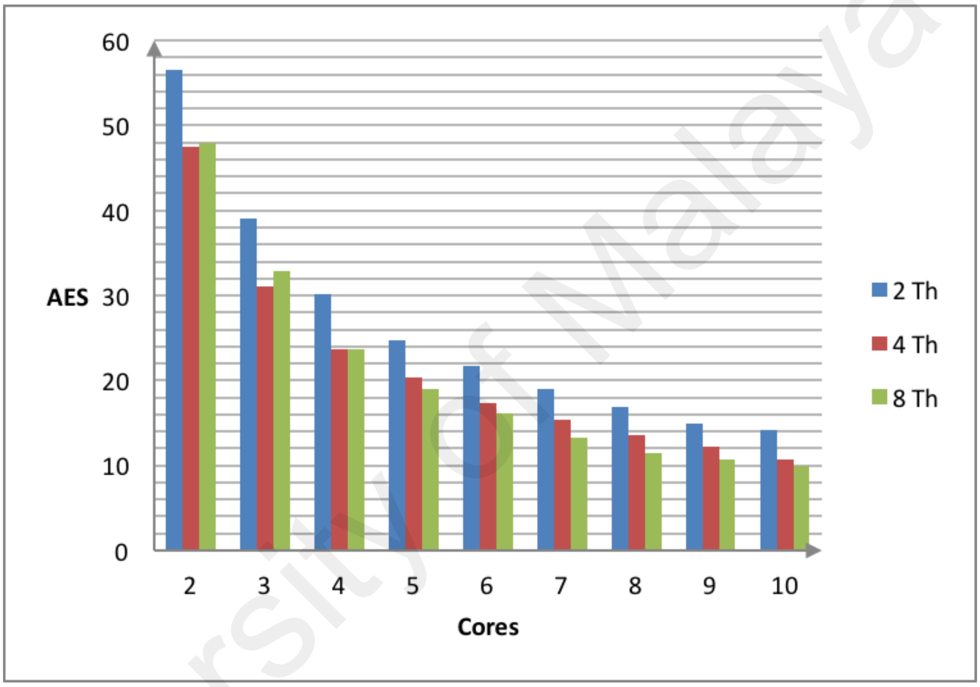


Figure 5.2: Sketching the AES values vs. number of cores for the Fibonacci (10) problem using eight-thread, four-thread, two-thread partitioning, and IOSSS for threads distribution

In general, the histogram shows that as the number of cores is increased, the performance becomes better. In other words, the values of the AES become less which is the target behind increasing the number of the cores. It is also noted that there is a clear difference between the two-thread method and the other two methods. This is due to the high number of divisions comparing with other two methods. On the other hand, the results of the four-thread and eight-thread methods are unstable in the 2-4 cores models but the difference becomes clear starting from the fifth core. Yet, the histogram reaches a stability point starting from the tenth core.

The weakness in the IOSSS is clear which is represented by the number of threads that each thief core gets. The thief core gets only a single thread at a time, processes it, and then it becomes a thief again, and after that the distribution process is invoked again. This scenario is repeated frequently which leads to a waste of time, as a result, it has a bad effect on the overall performance. In conclusion, the IOSSS is easy to implement and has no complicated calculations but it cannot achieve a high level of concurrency.

The researcher resolves the Fibonacci problem but this time for Fibonacci (13) and Fibonacci (15). This time, the researcher fixes the partitioning method to the eight-thread and applies various stealing strategies. The researcher uses the strategies: IOSSS, IOMSS, RFSSS, RFMSS, and CMSS. Table 5.4 and Table 5.5 show the AES values for computing Fibonacci (13) and Fibonacci (15). The results in the previous tables are sketched as histograms in Figure 5.3 and Figure 5.4 respectively.

Analyzing the results in the two histograms leads to the following facts:

1- Solving Fibonacci (15) consumes more steps than Fibonacci (13). This is evident from the values of AES. In Fibonacci (15) the maximum AES value is 509. This value appears in both IOSSS and RFSSS; however, in Fibonacci (13), the maximum AES value is 208. This is natural since increasing the size of any problem will definitely lead to an increase in the number of execution steps. The only exception happens in the Binary Search problem, since the repetition of model execution with different arguments (searching element) requires a different number of steps. The issue is related with the location of the element searched for.

Table 5.4: The AES values vs. strategies for the problem of solving Fibonacci (13) using eight-thread partitioning

Cores	IOSSS	IOMSS	RFSSS	RFMSS	CMSS
2	208.27	193.58	208.27	193.58	193.27
3	139.90	132.43	142.88	131.47	130.51
4	105.36	98.99	106.86	99.20	98.48
5	84.52	82.22	86.76	80.82	79.66
6	68.88	68.65	70.14	67.95	66.70
7	59.71	57.99	60.55	57.11	56.57
8	53.78	51.75	55.26	50.60	49.65
9	49.34	46.95	50.70	45.34	44.96
10	45.51	43.00	47.29	41.24	41.06

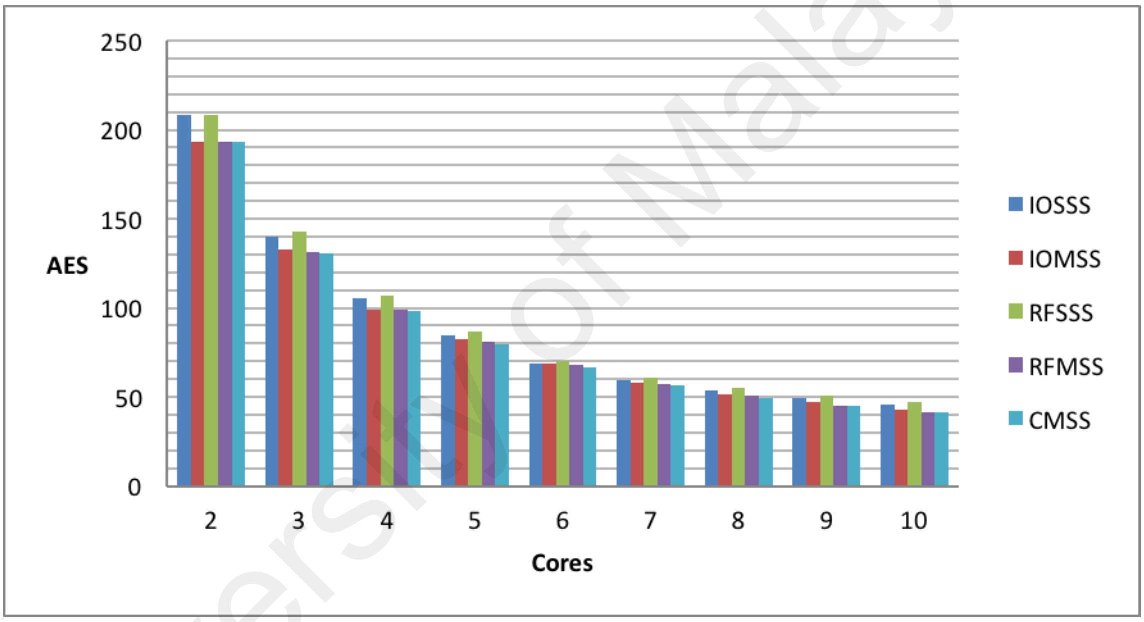


Figure 5.3: Sketching the AES values vs. number of cores for the problem of Fibonacci (13) using eight-thread partitioning

2- The general overview to the histograms indicates that as the number of cores increases, the performance becomes better in the sense of using less execution steps. This can be generalized to all the HLSs. The histogram descends smoothly from worst AES values at the two-core model to the best AES values at the ten-core model.

Table 5.5: The AES values vs. strategies for the problem of solving of Fibonacci (15) using eight-thread partitioning

Cores	IOSSS	IOMSS	RFSSS	RFMSS	CMSS
2	**509.47**	**479.76**	**509.47**	**479.76**	**479.15**
3	**341.62**	**324.59**	**353.40**	**322.61**	**321.60**
4	**257.39**	**242.60**	**265.97**	**242.92**	**241.49**
5	**207.32**	**199.03**	**214.78**	**196.08**	**194.24**
6	**169.78**	**166.57**	**173.34**	**164.19**	**162.15**
7	**146.03**	**141.07**	**147.24**	**139.34**	**138.42**
8	**130.32**	**125.49**	**131.79**	**122.63**	**121.32**
9	**118.78**	**113.38**	**119.96**	**109.59**	**108.98**
10	**109.06**	**103.40**	**110.85**	**99.13**	**98.50**

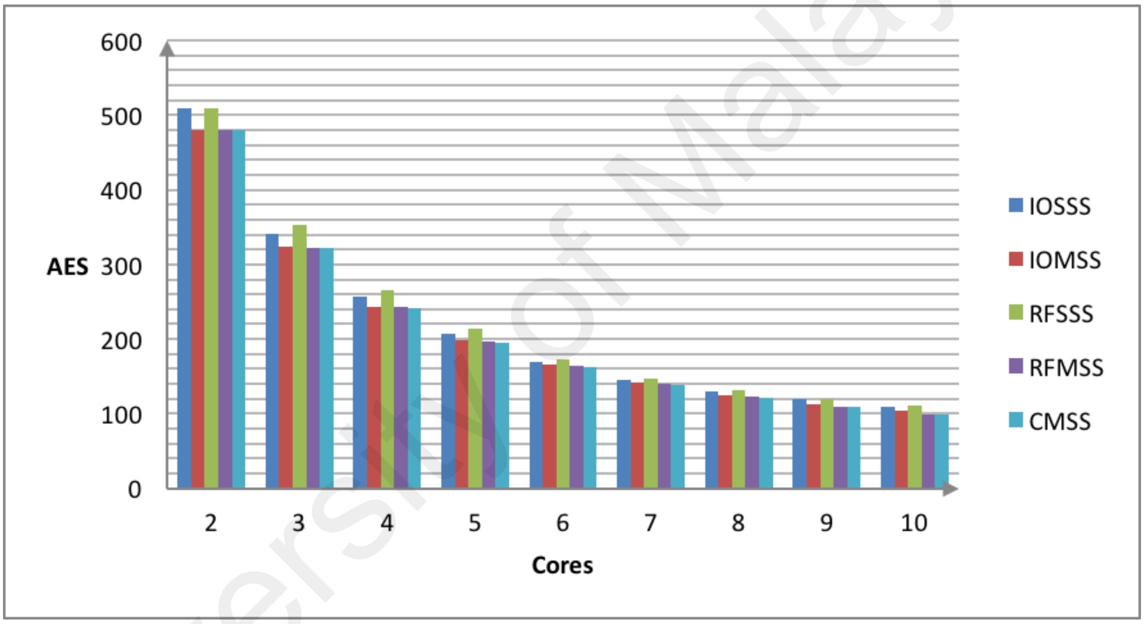


Figure 5.4: Sketching the AES values vs. number of cores for the problem of Fibonacci (15) using eight-thread partitioning

3- The rate of change in AES values in the low number of cores models is higher than their counterparts with a higher number of cores. For instance, the difference between the AES values between the two-core and three-core models is higher than the difference between the nine-core and ten-core models. This is because as the number of cores increases and with the same problem size, the single core's share of threads decreases; consequently, there will be less need to call the HLS. As a result, the AES values become convergent. In other words, the effect of the HLSs becomes less significant when the number of cores is relatively large comparing with the problem size. This is why the difference between the two and three-core models can be noticed clearly where the HLSs play a significant role in the redistribution process ultimately generating better results.

4- In order to compare the performances of the HLSs; the IOSSS and RFSSS show the worst performance since they consume relatively high values of AESs. The reason behind this is that both of them distribute only single threads to the thief cores. This has a bad effect on the overall performance since a core with a single thread quickly becomes a thief core again and the distribution process has to be invoked continuously. An interesting thing is the results of the IOSSS and RFSSS are convergent and sometimes identical. That is because the main thread resides on the first core and this thread precisely generates eight threads at the beginning of execution therefore there is a good chance to make this core one of the wealthiest cores of threads. As a result, the first core becomes the target of the RFSSS most of the time. As for the IOSSS, it is also the target of this strategy since IOSSS starts from left to right. This makes both IOSSS and RFSSS give convergent results, even so when there is a difference due to choosing another victim core by RFSSS. The number of threads in this victim core does not differentiate too much from the victim core chosen by IOSSS.

5- The number of threads generated in one step has a strong influence on the overall performance. The researcher has resolved the problems of Fibonacci (13) and Fibonacci (15) but this time using only two-thread partitioning. Table 5.6 and Figure 5.5 are dedicated for Fibonacci (13) while Table 5.7 and Figure 5.6 are belonging to Fibonacci (15). At first glance, a comparison between Figure 5.3 and Figure 5.5 shows a clear difference between the AES for the same problem of Fibonacci (13).

Table 5.6: The AES values vs. strategies for the problem of solving of Fibonacci (13) using two-thread partitioning.

Cores	IOSSS	IOMSS	RFSSS	RFMSS	CMSS
2	237.85	236.05	237.85	236.05	236.30
3	162.63	160.47	163.03	160.87	160.27
4	125.50	124.15	126.50	123.40	121.05
5	102.22	100.72	101.50	99.78	98.50
6	86.12	86.55	88.13	84.40	82.75
7	76.47	73.70	75.10	73.16	71.63
8	68.56	64.95	67.09	64.75	63.93
9	59.83	60.36	60.29	57.77	56.74
10	55.08	55.62	55.37	52.85	51.92

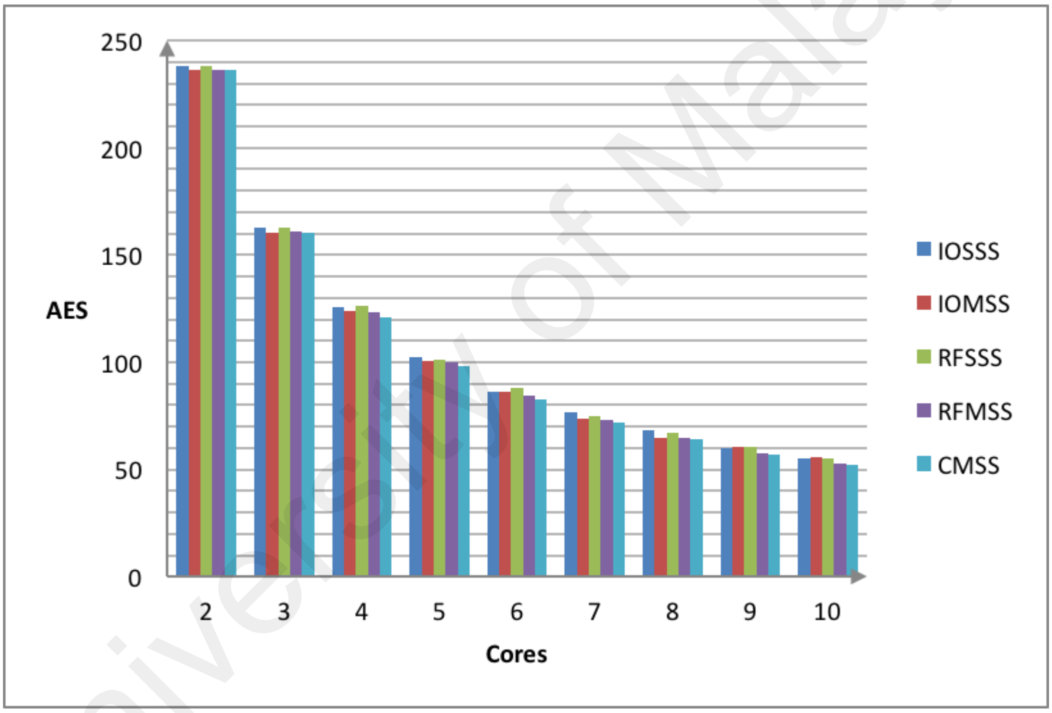


Figure 5.5: Sketching the AES values vs. number of cores for the problem of Fibonacci (13) using two-thread partitioning.

The results in Figure 5.5 show that more steps are needed to complete simulation comparing with results in Figure 5.3. The same thing can be said in the case of Fibonacci (15) where in Figure 5.6 the AES values need more steps than those in Figure 5.4.

Table 5.7: The AES values vs. the entire strategies for solving the problem of Fibonacci (15) using two-thread partitioning

Cores	IOSSS	IOMSS	RFSSS	RFMSS	CMSS
2	621.55	615.10	621.55	615.10	614.90
3	421.17	413.77	424.47	415.00	413.93
4	323.03	316.65	322.35	315.40	312.78
5	263.24	259.80	263.16	254.84	254.04
6	220.30	214.28	221.27	214.08	212.17
7	190.66	187.94	191.71	185.46	181.43
8	168.50	165.68	168.66	163.53	160.33
9	151.43	151.27	150.81	146.29	144.58
10	138.62	134.98	138.38	132.53	130.22

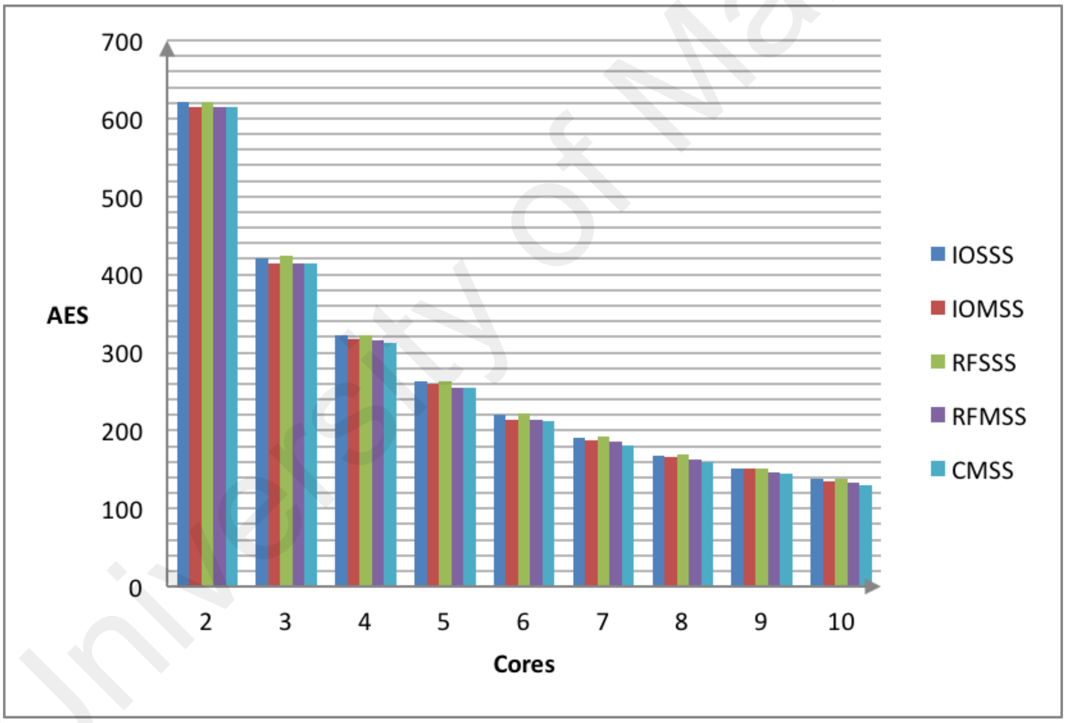


Figure 5.6: Sketching the AES values vs. number of cores for the problem of Fibonacci (15) using two-thread partitioning.

In general, we can conclude the following: a partitioning technique that can generate a higher number of threads in one step is the key to high level of concurrency between the cores of the model. However, this cannot be generalized for generating any number of threads; there should be a kind of balancing between the number of cores, the maximum number of generated threads in one step, and the time consumed in generating those threads.

5.3 The Results of Executing the Binary Search CPN Models

The Binary Search has a different technique than the Fibonacci technique. In the Fibonacci case, all the temporary results have to be taken into consideration; they are parts of the final result. However, in the Binary Search, the researcher builds the mechanism on dividing the searching area into a certain number of sections where each section is assigned to a thread, after that, the HLSs distribute those threads to the cores. The searching may be succeeded in one of the threads being executed by one of the cores; therefore, it is not compulsory to check all the threads. In other words, there is a chance to find the searched element in one of the threads. Consequently, the process stops immediately, and if there are any threads still waiting for their turn in processing, they will be discarded.

As in Fibonacci case, the researcher designed nine Binary Search CPN models (two-core model, three-core model, and so forth until ten-core model. To apply the searching technique, the researcher proposes an ordered list of 10000 integer numbers [1, 3, 5, 7... 19995, 19997, 19999]. This list is already created and saved in a fused place that is shared by all the cores. The index of the first element in the list is zero while the index of the last element in the list is 9999. The searching is carried out for the value 19997. The result of execution of the models results in the value 9998 which represents the location of the searched element in the list. In other words, it signifies the success of finding the element. On the other hand, returning the value ~1 means that the element is not found in the list. Regarding the HLSs and their role in distributing the threads, Table 5.8 shows the AES values of conducting the binary search while Figure 5.7 shows a histogram that reflects the values in Table 5.8.

Table 5.8: The AES values for the Binary Search Problem where the list size is 10000 and the list's values are: [1, 3, 5, 7,.. 19995,19997,19999] and searching is conducted for the value 19997 with the return list's value 9998

Cores	IOSSS	IOMSS	RFSSS	RFMSS	CMSS
2	56.70	38.90	56.70	38.90	38.10
3	41.90	26.67	41.90	26.17	26.33
4	32.45	20.23	32.45	19.63	19.53
5	26.34	16.56	26.34	16.20	15.38
6	22.13	14.45	22.13	13.03	12.90
7	18.96	11.97	18.96	11.56	11.29
8	16.75	10.45	16.75	9.93	9.89
9	14.69	9.93	14.69	9.09	9.06
10	13.20	8.45	13.20	8.01	8.00

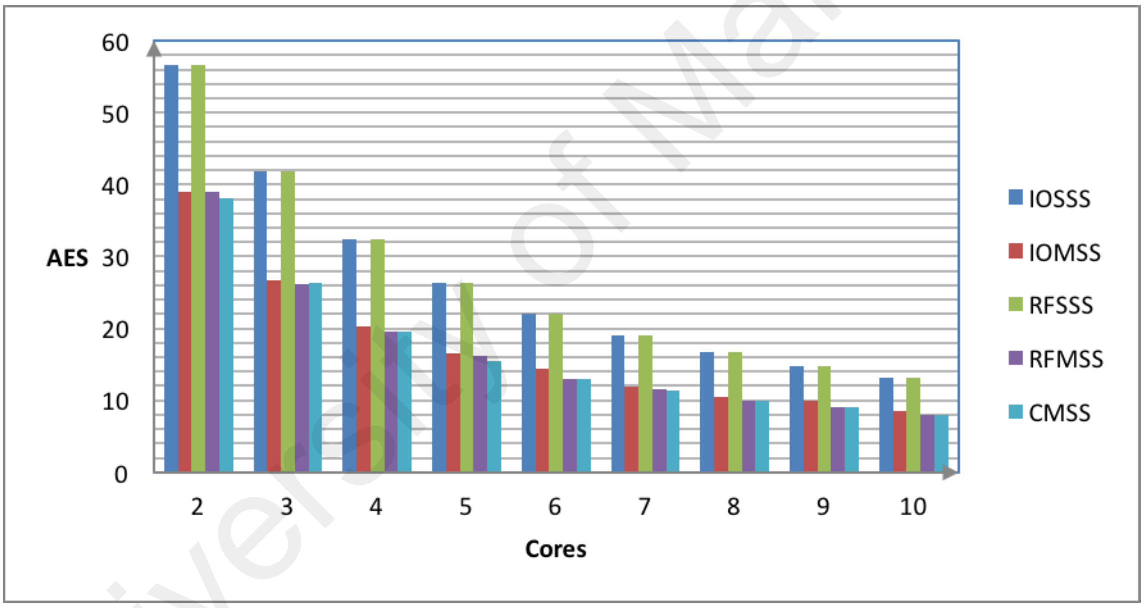


Figure 5.7: Sketching the AES values vs. number of cores for the Binary Search problem defined in Table 5.8

The researcher repeated the searching process but this time for the value 4. As expected, the result of the searching is ~1 since there are no even numbers in the list. The AESs' values are given in Table 5.9 while the histogram is illustrated in Figure 5.8.

Table 5.9: The AES values for the Binary Search Problem where list size is 10000 and the list's values are: [1, 3, 5, 7,.. 19995,19997,19999] and searching is conducted for the value 4 with the return value of ~1

Cores	IOSSS	IOMSS	RFSSS	RFMSS	CMSS
2	57.75	42.90	57.75	42.90	42.95
3	43.37	29.27	42.03	29.03	28.90
4	34.83	22.28	32.75	22.08	21.93
5	28.34	18.00	27.46	17.86	17.76
6	24.05	15.42	23.08	15.07	14.87
7	21.13	13.41	20.03	13.03	12.77
8	18.38	11.85	17.60	11.39	11.29
9	16.46	10.69	15.68	10.38	10.10
10	14.82	9.65	14.27	9.45	9.23

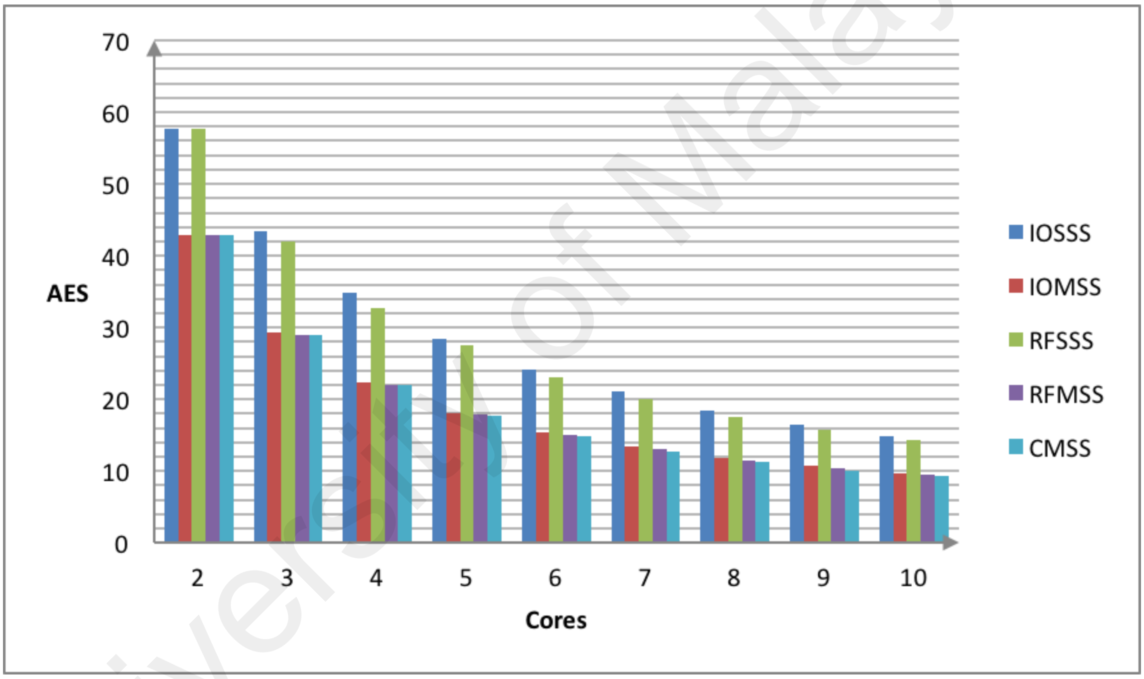


Figure 5.8: Sketching the AES values vs. number of cores for the Binary Search problem defined in Table 5.9

As in the Fibonacci case, the values of the AES are significantly improved with the increase in the number of cores. However, the binary search results show a clear difference from those of the Fibonacci results. This difference is represented precisely in the results of IOSSS and RFSSS. The results of those schedulers are so close and differ from the other schedulers while the results of all the schedulers in Fibonacci case are convergent.

The reason behind this is in Fibonacci, a thread may spawn 2,4, or 8 threads or may be more if the FLLS includes such partitioning, besides the reallocation of threads by the HLSs makes the cores semi saturated with threads. This matter creates a state of convergence between the HLSs, in other words, in the case of Fibonacci; the difference between the capacities of the HLSs on distributing the threads is not as clear as in Binary Search. In Binary Search, the BSLLS controls the size of the searching area assigned to the thread by the value of Delta (Figure 4.5). The mechanism of the BSLLS can generate so many threads with one step. As a result, starting from the first step of simulation, the BSLLS creates a large number of threads and stacked them at the core that holds the main thread. Adding to that, both the IOSSS and RFSSS steal and distribute single threads. Therefore, there will be too much calling to the scheduler since thief cores only get single threads. Therefore, the AES values for the IOSSS and RFSSS are relatively high. On the other hand, the other schedulers: IOMSS, RFSSS, and CMSS show more convergent results than IOSSS and RFSSS since they (IOMSS, RFSSS, and CMSS) have been built on stealing more than one thread.

5.4 The Results of Executing the Towers of Hanoi CPN Models

The Towers of Hanoi problem has a restricted approach comparing with Fibonacci and Binary Search in splitting the problem. Here, the THLLS has no choice other than computing a move and creating two sub threads at a time. In other words, THLLS lacks the ability of the FLLS in creating 4,8, or more threads at the same time. The same thing can be said for the BSLLS where the scheduler can create many searching areas (threads) that can be distributed under the HLS to the thief cores at the same time. The reason behind THLLS's inability is due to the accumulation of disks on each other. The game's player cannot move two disks at the same time; it should be one by one.

Therefore, the concurrent characteristic in this type of D&C problem is weak. As a result, the diversity in the HLSs has no effect on the results of simulation.

As with the previous two D&C problems, the researcher has designed two types of CPN models, one for solving the problem with seven disks, and the second one with nine disks. The output of the seven disks problem is given in Table 5.10. As explained in Section 4.1.3, the output of the Towers of Hanoi game is a set of disks' moves. A move consists of (Ord,DNo,Sou,Des), the number of moves is equal to (2^n) -1 where n is the number of disks. In Table 5.10, the number of disks is seven, therefore we have 127 moves. The THLLS (Figure 4.9) creates a binary tree with 127 threads; each thread has its move. The moves on the left side of the tree are numbered with the negative sign (~), the move at the root is numbered with zero, and the moves at the right hand side of the binary tree are numbered with positive sign. Therefore, for 127 moves, the first move is (~63,1,1,3) which indicates moving disk No. 1 from pillar No.1 to pillar No. 3. The next move is (~62,2,1,2) which includes moving disk No. 2 from pillar No.1 to pillar No. 2. The last move is (63,1,1,3) which consists of moving disk No. 1 from pillar No.1 to pillar No. 3. As for the nine disks example, the output is given in Table 5.11. Here, we have 511 moves, 255 moves with negative sign resident at the left side of the binary tree, and 255 moves with positive sign resident at the right hand side of the binary tree. Therefore, the first move is (~255,1,1,3) and the last one is (255,1,1,3).

Regarding the distribution of threads through the five strategies, Table 5.12 shows the AES values for solving the Towers of Hanoi game using seven disks. Figure 5.9 sketches the values in Table 5.12. For the nine disks example, Table 5.13 and Figure 5.10 are dedicated for the nine disks problem.

Table 5.10: The output (moves) of solving the problem of Towers of Hanoi with seven disks

~63,1,1,3	~62,2,1,2	~61,1,3,2	~60,3,1,3	~59,1,2,1	~58,2,2,3	~57,1,1,3	~56,4,1,2	~55,1,3,2
~54,2,3,1	~53,1,2,1	~52,3,3,2	~51,1,1,3	~50,2,1,2	~49,1,3,2	~48,5,1,3	~47,1,2,1	~46,2,2,3
~45,1,1,3	~44,3,2,1	~43,1,3,2	~42,2,3,1	~41,1,2,1	~40,4,2,3	~39,1,1,3	~38,2,1,2	~37,1,3,2
~36,3,1,3	~35,1,2,1	~34,2,2,3	~33,1,1,3	~32,6,1,2	~31,1,3,2	~30,2,3,1	~29,1,2,1	~28,3,3,2
~27,1,1,3	~26,2,1,2	~25,1,3,2	~24,4,3,1	~23,1,2,1	~22,2,2,3	~21,1,1,3	~20,3,2,1	~19,1,3,2
~18,2,3,1	~17,1,2,1	~16,5,3,2	~15,1,1,3	~14,2,1,2	~13,1,3,2	~12,3,1,3	~11,1,2,1	~10,2,2,3
~9,1,1,3	~8,4,1,2	~7,1,3,2	~6,2,3,1	~5,1,2,1	~4,3,3,2	~3,1,1,3	~2,2,1,2	~1,1,3,2
0,7,1,3	1,1,2,1	2,2,2,3	3,1,1,3	4,3,2,1	5,1,3,2	6,2,3,1	7,1,2,1	8,4,2,3
9,1,1,3	10,2,1,2	11,1,3,2	12,3,1,3	13,1,2,1	14,2,2,3	15,1,1,3	16,5,2,1	17,1,3,2
18,2,3,1	19,1,2,1	20,3,3,2	21,1,1,3	22,2,1,2	23,1,3,2	24,4,3,1	25,1,2,1	26,2,2,3
27,1,1,3	28,3,2,1	29,1,3,2	30,2,3,1	31,1,2,1	32,6,2,3	33,1,1,3	34,2,1,2	35,1,3,2
36,3,1,3	37,1,2,1	38,2,2,3	39,1,1,3	40,4,1,2	41,1,3,2	42,2,3,1	43,1,2,1	44,3,3,2
45,1,1,3	46,2,1,2	47,1,3,2	48,5,1,3	49,1,2,1	50,2,2,3	51,1,1,3	52,3,2,1	53,1,3,2
54,2,3,1	55,1,2,1	56,4,2,3	57,1,1,3	58,2,1,2	59,1,3,2	60,3,1,3	61,1,2,1	62,2,2,3
63,1,1,3								

Table 5.11: The moves of solving the problem of Towers of Hanoi with nine disks

~255,1,1,3	~254,2,1,2	~253,1,3,2	~252,3,1,3	~251,1,2,1	~250,2,2,3	~249,1,1,3	~248,4,1,2	~247,1,3,2
~246,2,3,1	~245,1,2,1	~244,3,3,2	~243,1,1,3	~242,2,1,2	~241,1,3,2	~240,5,1,3	~239,1,2,1	~238,2,2,3
~237,1,1,3	~236,3,2,1	~235,1,3,2	~234,2,3,1	~233,1,2,1	~232,4,2,3	~231,1,1,3	~230,2,1,2	~229,1,3,2
~228,3,1,3	~227,1,2,1	~226,2,2,3	~225,1,1,3	~224,6,1,2	~223,1,3,2	~222,2,3,1	~221,1,2,1	~220,3,3,2
~219,1,1,3	~218,2,1,2	~217,1,3,2	~216,4,3,1	~215,1,2,1	~214,2,2,3	~213,1,1,3	~212,3,2,1	~211,1,3,2
~210,2,3,1	~209,1,2,1	~208,5,3,2	~207,1,1,3	~206,2,1,2	~205,1,3,2	~204,3,1,3	~203,1,2,1	~202,2,2,3
~201,1,1,3	~200,4,1,2	~199,1,3,2	~198,2,3,1	~197,1,2,1	~196,3,3,2	~195,1,1,3	~194,2,1,2	~193,1,3,2
~192,7,1,3	~191,1,2,1	~190,2,2,3	~189,1,1,3	~188,3,2,1	~187,1,3,2	~186,2,3,1	~185,1,2,1	~184,4,2,3
~183,1,1,3	~182,2,1,2	~181,1,3,2	~180,3,1,3	~179,1,2,1	~178,2,2,3	~177,1,1,3	~176,5,2,1	~175,1,3,2
~174,2,3,1	~173,1,2,1	~172,3,3,2	~171,1,1,3	~170,2,1,2	~169,1,3,2	~168,4,3,1	~167,1,2,1	~166,2,2,3
~165,1,1,3	~164,3,2,1	~163,1,3,2	~162,2,3,1	~161,1,2,1	~160,6,2,3	~159,1,1,3	~158,2,1,2	~157,1,3,2
~156,3,1,3	~155,1,2,1	~154,2,2,3	~153,1,1,3	~152,4,1,2	~151,1,3,2	~150,2,3,1	~149,1,2,1	~148,3,3,2
~147,1,1,3	~146,2,1,2	~145,1,3,2	~144,5,1,3	~143,1,2,1	~142,2,2,3	~141,1,1,3	~140,3,2,1	~139,1,3,2
~138,2,3,1	~137,1,2,1	~136,4,2,3	~135,1,1,3	~134,2,1,2	~133,1,3,2	~132,3,1,3	~131,1,2,1	~130,2,2,3
~129,1,1,3	~128,8,1,2	~127,1,3,2	~126,2,3,1	~125,1,2,1	~124,3,3,2	~123,1,1,3	~122,2,1,2	~121,1,3,2
~120,4,3,1	~119,1,2,1	~118,2,2,3	~117,1,1,3	~116,3,2,1	~115,1,3,2	~114,2,3,1	~113,1,2,1	~112,5,3,2
~111,1,1,3	~110,2,1,2	~109,1,3,2	~108,3,1,3	~107,1,2,1	~106,2,2,3	~105,1,1,3	~104,4,1,2	~103,1,3,2
~102,2,3,1	~101,1,2,1	~100,3,3,2	~99,1,1,3	~98,2,1,2	~97,1,3,2	~96,6,3,1	~95,1,2,1	~94,2,2,3
~93,1,1,3	~92,3,2,1	~91,1,3,2	~90,2,3,1	~89,1,2,1	~88,4,2,3	~87,1,1,3	~86,2,1,2	~85,1,3,2
~84,3,1,3	~83,1,2,1	~82,2,2,3	~81,1,1,3	~80,5,2,1	~79,1,3,2	~78,2,3,1	~77,1,2,1	~76,3,3,2
~75,1,1,3	~74,2,1,2	~73,1,3,2	~72,4,3,1	~71,1,2,1	~70,2,2,3	~69,1,1,3	~68,3,2,1	~67,1,3,2
~66,2,3,1	~65,1,2,1	~64,7,3,2	~63,1,1,3	~62,2,1,2	~61,1,3,2	~60,3,1,3	~59,1,2,1	~58,2,2,3
~57,1,1,3	~56,4,1,2	~55,1,3,2	~54,2,3,1	~53,1,2,1	~52,3,3,2	~51,1,1,3	~50,2,1,2	~49,1,3,2
~48,5,1,3	~47,1,2,1	~46,2,2,3	~45,1,1,3	~44,3,2,1	~43,1,3,2	~42,2,3,1	~41,1,2,1	~40,4,2,3
~39,1,1,3	~38,2,1,2	~37,1,3,2	~36,3,1,3	~35,1,2,1	~34,2,2,3	~33,1,1,3	~32,6,1,2	~31,1,3,2
~30,2,3,1	~29,1,2,1	~28,3,3,2	~27,1,1,3	~26,2,1,2	~25,1,3,2	~24,4,3,1	~23,1,2,1	~22,2,2,3
~21,1,1,3	~20,3,2,1	~19,1,3,2	~18,2,3,1	~17,1,2,1	~16,5,3,2	~15,1,1,3	~14,2,1,2	~13,1,3,2
~12,3,1,3	~11,1,2,1	~10,2,2,3	~9,1,1,3	~8,4,1,2	~7,1,3,2	~6,2,3,1	~5,1,2,1	~4,3,3,2
~3,1,1,3	~2,2,1,2	~1,1,3,2	0,9,1,3	1,1,2,1	2,2,2,3	3,1,1,3	4,3,2,1	5,1,3,2
6,2,3,1	7,1,2,1	8,4,2,3	9,1,1,3	10,2,1,2	11,1,3,2	12,3,1,3	13,1,2,1	14,2,2,3
15,1,1,3	16,5,2,1	17,1,3,2	18,2,3,1	19,1,2,1	20,3,3,2	21,1,1,3	22,2,1,2	23,1,3,2
24,4,3,1	25,1,2,1	26,2,2,3	27,1,1,3	28,3,2,1	29,1,3,2	30,2,3,1	31,1,2,1	32,6,2,3
33,1,1,3	34,2,1,2	35,1,3,2	36,3,1,3	37,1,2,1	38,2,2,3	39,1,1,3	40,4,1,2	41,1,3,2
42,2,3,1	43,1,2,1	44,3,3,2	45,1,1,3	46,2,1,2	47,1,3,2	48,5,1,3	49,1,2,1	50,2,2,3
51,1,1,3	52,3,2,1	53,1,3,2	54,2,3,1	55,1,2,1	56,4,2,3	57,1,1,3	58,2,1,2	59,1,3,2
60,3,1,3	61,1,2,1	62,2,2,3	63,1,1,3	64,7,2,1	65,1,3,2	66,2,3,1	67,1,2,1	68,3,3,2

69,1,1,3	70,2,1,2	71,1,3,2	72,4,3,1	73,1,2,1	74,2,2,3	75,1,1,3	76,3,2,1	77,1,3,2
78,2,3,1	79,1,2,1	80,5,3,2	81,1,1,3	82,2,1,2	83,1,3,2	84,3,1,3	85,1,2,1	86,2,2,3
87,1,1,3	88,4,1,2	89,1,3,2	90,2,3,1	91,1,2,1	92,3,3,2	93,1,1,3	94,2,1,2	95,1,3,2
96,6,3,1	97,1,2,1	98,2,2,3	99,1,1,3	100,3,2,1	101,1,3,2	102,2,3,1	103,1,2,1	104,4,2,3
105,1,1,3	106,2,1,2	107,1,3,2	108,3,1,3	109,1,2,1	110,2,2,3	111,1,1,3	112,5,2,1	113,1,3,2
114,2,3,1	115,1,2,1	116,3,3,2	117,1,1,3	118,2,1,2	119,1,3,2	120,4,3,1	121,1,2,1	122,2,2,3
123,1,1,3	124,3,2,1	125,1,3,2	126,2,3,1	127,1,2,1	128,8,2,3	129,1,1,3	130,2,1,2	131,1,3,2
132,3,1,3	133,1,2,1	134,2,2,3	135,1,1,3	136,4,1,2	137,1,3,2	138,2,3,1	139,1,2,1	140,3,3,2
141,1,1,3	142,2,1,2	143,1,3,2	144,5,1,3	145,1,2,1	146,2,2,3	147,1,1,3	148,3,2,1	149,1,3,2
150,2,3,1	151,1,2,1	152,4,2,3	153,1,1,3	154,2,1,2	155,1,3,2	156,3,1,3	157,1,2,1	158,2,2,3
159,1,1,3	160,6,1,2	161,1,3,2	162,2,3,1	163,1,2,1	164,3,3,2	165,1,1,3	166,2,1,2	167,1,3,2
168,4,3,1	169,1,2,1	170,2,2,3	171,1,1,3	172,3,2,1	173,1,3,2	174,2,3,1	175,1,2,1	176,5,3,2
177,1,1,3	178,2,1,2	179,1,3,2	180,3,1,3	181,1,2,1	182,2,2,3	183,1,1,3	184,4,1,2	185,1,3,2
186,2,3,1	187,1,2,1	188,3,3,2	189,1,1,3	190,2,1,2	191,1,3,2	192,7,1,3	193,1,2,1	194,2,2,3
195,1,1,3	196,3,2,1	197,1,3,2	198,2,3,1	199,1,2,1	200,4,2,3	201,1,1,3	202,2,1,2	203,1,3,2
204,3,1,3	205,1,2,1	206,2,2,3	207,1,1,3	208,5,2,1	209,1,3,2	210,2,3,1	211,1,2,1	212,3,3,2
213,1,1,3	214,2,1,2	215,1,3,2	216,4,3,1	217,1,2,1	218,2,2,3	219,1,1,3	220,3,2,1	221,1,3,2
222,2,3,1	223,1,2,1	224,6,2,3	225,1,1,3	226,2,1,2	227,1,3,2	228,3,1,3	229,1,2,1	230,2,2,3
231,1,1,3	232,4,1,2	233,1,3,2	234,2,3,1	235,1,2,1	236,3,3,2	237,1,1,3	238,2,1,2	239,1,3,2
240,5,1,3	241,1,2,1	242,2,2,3	243,1,1,3	244,3,2,1	245,1,3,2	246,2,3,1	247,1,2,1	248,4,2,3
249,1,1,3	250,2,1,2	251,1,3,2	252,3,1,3	253,1,2,1	254,2,2,3	255,1,1,3		

Table 5.12: The AES values for the Towers of Hanoi problem using seven disks

Cores	IOSSS	IOMSS	RFSSS	RFMSS	CMSS
2	36.25	35.84	36.16	35.58	35.75
3	29.62	29.21	29.55	28.62	28.67
4	24.29	24.06	24.76	23.89	24.21
5	21.54	21.14	21.62	21.12	21.10
6	19.44	19.24	19.22	18.91	19.14
7	17.41	17.16	17.45	17.07	17.81
8	16.16	15.99	15.89	15.68	16.14
9	15.18	15.22	15.08	14.70	15.44
10	14.24	14.14	13.89	13.98	14.23

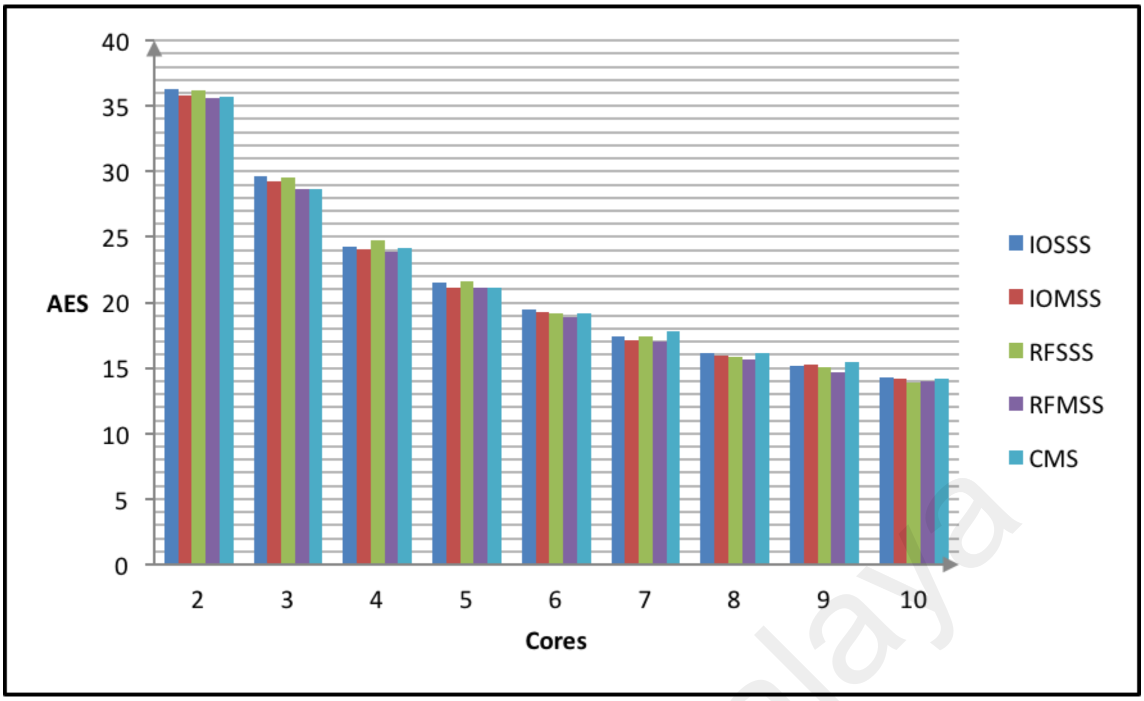


Figure 5.9: Sketching the results in Table 5.12

The results in the two figures show that the AES values in the nine disks example are higher than their counterparts in the seven disks example. This is natural, since adding more disks causes an increase in the number of moves. However, the two graphs demonstrate different behavior of the HLSs comparing with the results in Fibonacci and Binary Search results. Precisely, the results of the IOSSS which show in some cases better performance than the RFSSS, RFMSS, and CMSS performances.

Table 5.13: The AES values for the Towers of Hanoi problem using nine disks

Cores	IOSSS	IOMSS	RFSSS	RFMSS	CMSS
2	134.02	133.20	134.02	133.20	132.55
3	99.78	98.91	100.26	98.68	98.09
4	77.82	77.08	81.63	78.12	77.45
5	67.38	66.31	71.61	67.06	67.03
6	59.11	57.55	62.60	58.52	58.14
7	51.62	50.31	55.95	52.43	52.31
8	47.00	45.78	50.48	47.60	47.75
9	43.31	41.96	46.95	44.06	44.06
10	40.23	38.69	43.55	40.48	40.80

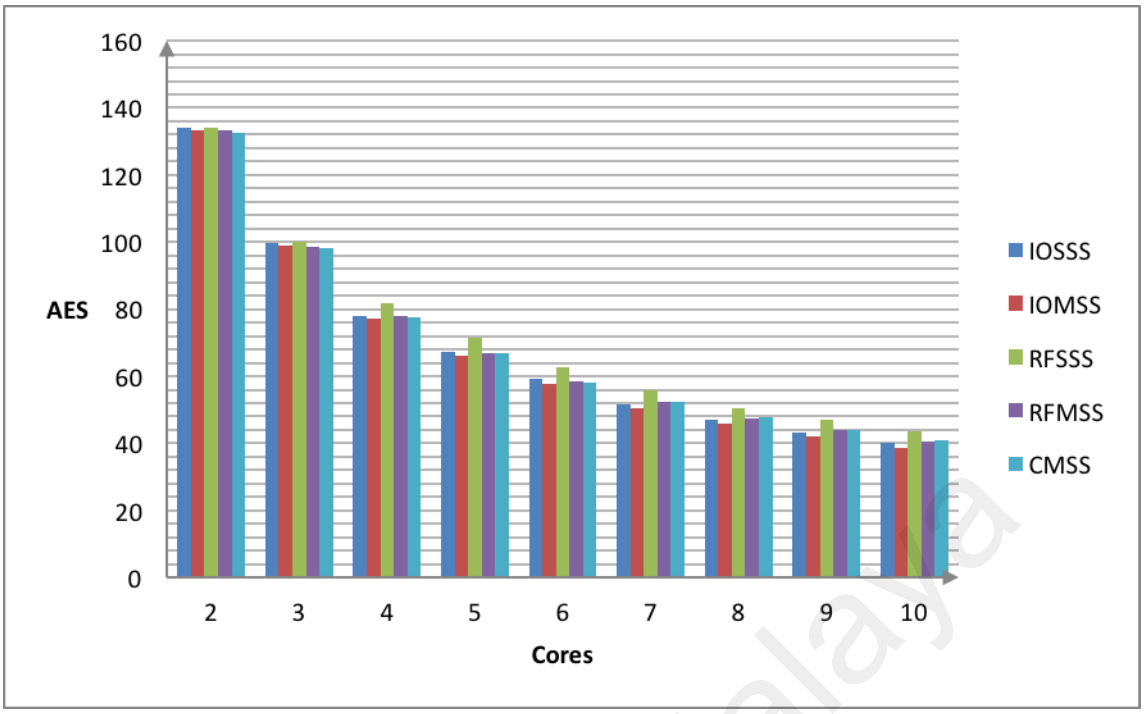


Figure 5.10: Sketching the results in Table 5.13

The reason behind this is in the THLLS mechanism where each core creates at the most only two sub threads, this leads to generating relatively heavy threads. By heavy thread, we mean a thread that is headed a heavy sub tree of threads in the Towers of Hanoi binary tree of threads (Figure 4.10). Comparing with the threads created by BSLLS and FLLS (eight-thread partitioning and more), the threads in those schedulers are lighter in the sense that they deal with smaller portions of the problem. It follows that the THLLS starts with the heavy sub threads and then accumulates the relatively smaller threads. As a result, heavy threads are settled at the bottom of the core's list of threads. Now, the IOSSS steals from the first encountered victim, digging inside it looking for threads to steal, as a result, the IOSSS relatively deals with heavy threads. On the other hand, the RFSSS search for the wealthy victim which leads to the core that has recently got new threads which are lighter than those targeted by IOSSS. Consequently, the performance of the IOSSS reaches the performance of the IOMSS, RFMSS, CMSS, and sometimes exceeds them. The heavy threads generate more sub threads, and this will definitely make the core busier for dealing with its own threads, accordingly, calling the HLS will be less. This is because as the cores become busy with their own threads, the need for calling the HLS will be less.

In conclusion, the main reason behind the improvement of the IOSSS is that this scheduler relatively deals with heavy threads while the other schedulers deal with lighter threads. What makes worse for the other schedulers is the amount of generated threads. Spawning only two threads at a time with the non deterministic behavior of the model makes all the lists of cores in convergent sizes.

5.5 The Results of Executing the Matrix Multiplication CPN Models

The MMLLS behaves like the BSLLS; it can generate many threads at the same time. A thread generated by the MMLLS holds a row number of the first matrix, the size of the element in that row, and the number of the column of the second matrix. The researcher solves two examples: First one is a 10 × 10 matrix and the second is a 20 × 20 matrix. The values of the first (A) matrix and the second (B) matrix are already created and saved inside the corresponding fused places. Table 5.14 shows the matrices' values of the first example while Table 5.15 shows the matrices' values of the second example. Table 5.16 and Figure 5.11 present the AES values for the Matrix Multiplication problem 10×10 while Table 5.17 and Figure 5.12 present the AES values for the Matrix Multiplication problem 20×20.

Table 5.14: The values of the input and output matrices for the 10 × 10 Matrix Multiplication example

Input Matrix A	[[0,8,2,0,3,3,9,0,8,5],[2,9,4,1,9,5,2,3,7,9],[7,6,6,7,0,2,0,9,9,3], [5,7,5,9,6,5,4,2,0,6],[8,0,8,1,6,0,5,3,8,5],[4,7,7,7,7,9,8,3,3,3], [9,2,8,0,6,2,1,4,5,2],[9,1,4,4,1,4,7,0,8,1],[1,7,9,4,5,0,5,2,4,6],[3,7,9,8,9,5,5,4,2,7]];
Input Matrix B	[[0,9,3,3,6,9,7,4,6,8],[6,5,5,4,4,8,6,6,2,7],[2,4,5,8,4,8,2,8,6,9], [7,0,2,9,7,7,5,4,3,0],[3,7,0,0,5,6,2,0,2,6],[4,5,3,0,7,7,0,6,8,1], [7,4,5,1,6,1,2,8,6,2],[7,4,6,4,1,7,4,9,8,4],[6,3,8,2,8,7,8,7,9,9],[5,5,2,2,6,8,2,0,8,7]];
Output Matrix C	[[209,169,178,83,224,224,150,210,224,220],[238,253,190,129,276,362,189,220,286,316], [237,205,233,216,252,372,246,296,304,292],[225,221,152,188,268,341,172,218,241,233], [170,227,183,140,244,301,185,223,277,301],[272,261,211,191,320,376,191,308,310,271], [129,220,156,130,200,291,165,203,233,273],[163,186,173,124,247,254,179,226,248,218], [206,185,172,172,223,290,159,220,226,264],[273,263,199,220,310,403,195,278,304,306]]

Table 5.15: The values of the input and output matrices for the 20 × 20 Matrix Multiplication example

Input Matrix A	[[2,1,7,2,3,3,9,6,1,6,2,2,3,0,8,0,3,4,2,2],[8,0,8,0,7,3,2,3,1,3,1,1,4,7,0,5,4,5,1,0], [8,7,9,2,1,6,3,5,5,0,6,4,0,0,0,3,0,4,2,8],[5,1,3,5,3,4,0,6,1,6,0,2,4,0,7,4,7,8,3,1], [9,5,0,5,9,7,5,1,4,4,3,0,7,3,9,5,8,5,9,1],[7,9,0,9,2,4,3,9,2,7,7,0,7,2,8,7,7,4,5,5], [0,7,8,9,2,3,2,1,5,5,6,8,6,5,1,8,9,9,4,8],[3,7,1,0,8,6,8,2,8,9,1,3,4,0,1,6,3,8,4,7], [4,8,8,4,2,0,4,6,9,7,1,2,2,1,1,0,5,3,1,1],[6,7,1,5,7,1,0,1,7,4,3,1,1,2,8,4,6,8,5,5], [9,3,0,8,7,8,9,6,6,2,0,3,4,8,0,5,4,9,5,1],[7,3,7,4,7,7,1,2,5,9,0,3,4,2,6,4,0,2,7,6], [4,2,2,8,7,3,3,1,7,6,0,4,0,3,9,2,9,1,4,2],[1,9,0,9,3,0,9,9,0,6,2,3,0,5,5,4,9,5,2,6], [3,4,1,7,2,0,2,5,8,7,9,5,0,1,7,8,0,7,2,7],[0,7,1,3,8,1,2,7,5,5,5,7,4,7,5,5,7,6,5,6], [2,9,7,5,7,5,9,9,4,3,0,4,9,2,7,8,6,7,7,3],[9,2,0,5,7,3,6,3,4,8,9,4,3,1,6,1,7,6,1,5], [2,8,1,4,1,1,4,3,2,8,8,9,9,9,2,3,5,7,8,4],[9,1,6,1,8,0,4,3,5,8,9,2,2,6,5,0,6,2,0,6]];
Input Matrix B	[[4,0,4,7,8,8,4,2,5,9,7,1,8,8,3,2,1,4,4,9],[8,0,3,1,2,3,7,4,7,1,2,1,8,4,1,2,4,7,6,4], [2,9,4,6,0,5,9,9,1,2,9,9,1,7,4,8,1,3,4,9],[7,8,7,5,7,9,2,6,2,9,8,5,3,0,3,0,1,7,4,4], [0,7,7,8,2,3,9,0,6,6,3,3,8,1,7,9,5,7,7,0],[0,6,5,7,3,6,7,3,1,5,5,6,8,1,8,2,6,5,2,1], [9,1,9,7,3,5,3,9,5,2,8,5,4,3,1,5,1,0,7,4],[7,8,6,9,0,7,5,7,2,9,8,9,6,8,4,0,7,0,5,3], [3,0,9,9,1,6,9,8,1,1,4,0,7,8,2,7,4,3,4,7],[7,8,6,4,9,1,8,8,2,4,7,6,2,7,7,3,0,4,8,9], [3,7,1,0,2,7,1,1,5,6,4,4,7,0,6,1,7,2,1,8],[4,1,6,4,9,1,2,3,8,5,6,6,3,7,6,9,9,2,2,7], [5,4,6,2,3,1,4,0,2,9,6,7,5,1,2,0,8,1,8,6],[4,1,8,9,0,3,8,2,9,7,2,2,8,9,5,0,1,1,4,4], [1,8,9,9,2,1,7,8,6,1,9,8,6,3,8,4,1,8,1,4],[2,2,4,9,5,2,6,1,1,0,9,1,0,9,0,8,9,6,7,9], [8,2,4,5,2,8,6,1,7,7,8,8,8,5,0,5,3,7,1,2],[2,2,8,2,5,1,0,6,2,6,9,6,1,8,4,8,2,8,1,3], [4,6,0,7,7,9,9,1,3,0,2,2,3,9,9,2,2,6,4,6],[5,7,6,7,0,7,5,3,7,4,9,7,4,2,6,1,0,3,9,1]];
Output Matrix C	[[299,355,411,399,216,285,364,378,248,291,477,409,301,310,303,270,188,259,293,323], [222,254,354,400,209,272,380,226,246,343,401,295,317,377,250,287,220,254,287,339], [293,311,364,409,223,397,386,328,279,319,471,340,364,372,303,283,263,280,322,392], [288,345,405,412,289,309,373,308,243,368,505,398,322,373,305,280,238,361,276,338], [393,411,543,597,385,474,581,341,405,460,589,429,551,460,437,366,335,503,424,460], [503,485,548,575,382,515,544,411,410,522,674,506,540,478,416,285,377,477,474,517], [473,445,571,542,381,479,545,415,431,484,690,527,457,529,404,441,395,478,462,543], [382,338,527,513,332,365,517,383,336,347,549,384,425,451,369,409,316,396,468,417], [359,281,401,386,224,335,421,386,250,306,436,333,348,391,235,285,206,278,323,384], [324,337,468,479,298,380,473,329,353,351,506,345,431,418,354,345,244,452,342,384], [420,334,593,614,375,481,509,385,376,517,587,402,511,517,379,366,339,409,431,435], [311,440,477,548,344,391,559,371,309,361,535,403,408,441,442,339,263,393,426,454], [330,349,475,507,300,378,480,346,339,344,488,367,416,372,345,330,216,397,312,356], [496,374,519,502,294,421,441,408,413,424,584,454,433,420,316,286,261,387,408,362], [352,393,493,482,328,378,417,401,325,356,570,381,371,433,372,340,312,380,373,473], [420,406,546,554,309,408,542,356,449,445,567,457,496,497,419,383,379,419,431,433], [503,490,648,674,369,490,645,492,426,486,731,574,533,568,438,464,438,497,536,528], [397,398,520,487,359,432,438,367,398,479,587,441,488,389,399,340,292,395,380,443], [476,378,514,471,395,407,494,348,453,476,557,455,472,512,426,319,375,382,437,522], [347,387,475,488,260,399,477,346,392,430,520,404,469,401,377,318,235,317,373,441]]

Table 5.16: The AES values for the Matrix Multiplication problem 10×10

Cores	IOSSS	IOMSS	RFSSS	RFMSS	CMSS
2	74.40	51.25	74.40	51.25	51.35
3	55.30	34.80	55.30	34.47	34.57
4	43.35	26.63	43.35	26.08	25.98
5	35.76	21.52	35.76	21.24	21.02
6	29.70	18.22	29.70	17.70	17.68
7	25.57	15.87	25.57	15.59	15.14
8	22.68	14.01	22.68	13.64	13.39
9	20.23	12.71	20.23	12.18	11.92
10	18.34	11.34	18.34	11.10	10.75

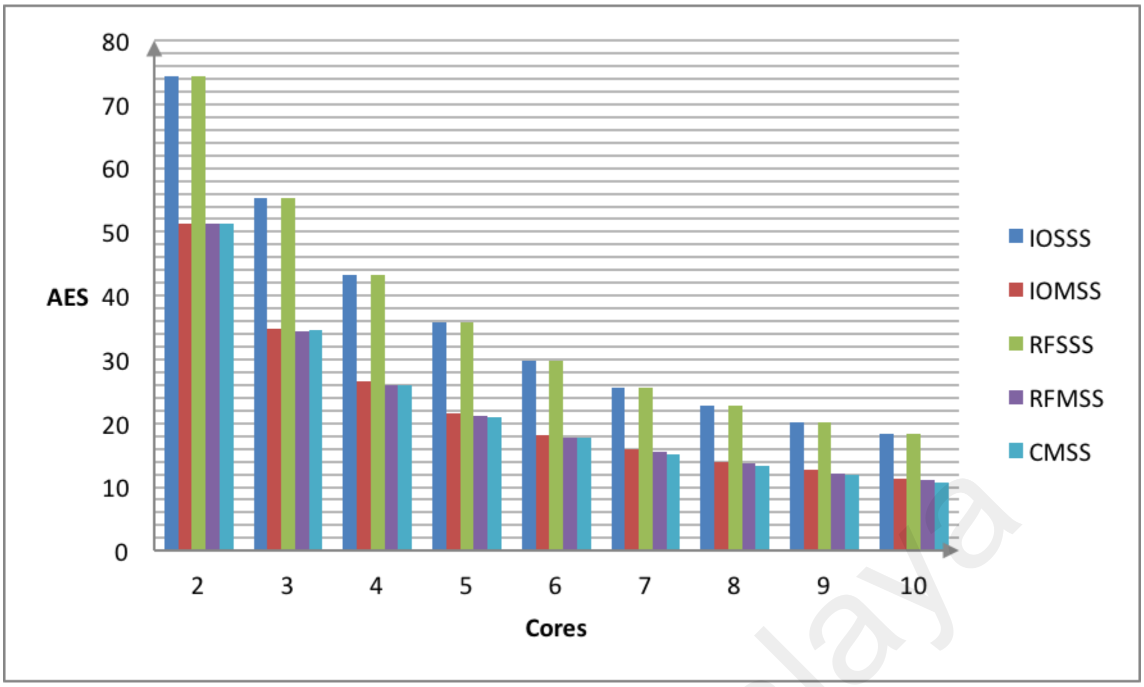


Figure 5.11: Sketching the results in Table 5.16

Table 5.17: The AES values for the Matrix Multiplication problem 20×20

Cores	IOSSS	IOMSS	RFSSS	RFMSS	CMSS
2	299.65	201.60	299.65	201.60	201.50
3	221.67	135.13	221.67	134.80	134.70
4	174.93	101.95	174.93	101.40	101.13
5	143.50	82.62	143.50	81.34	81.16
6	121.62	69.40	121.62	68.07	67.78
7	104.56	59.93	104.56	58.47	58.13
8	93.05	52.76	93.05	51.30	50.95
9	83.29	47.21	83.29	45.77	45.34
10	74.96	42.67	74.96	41.28	40.87

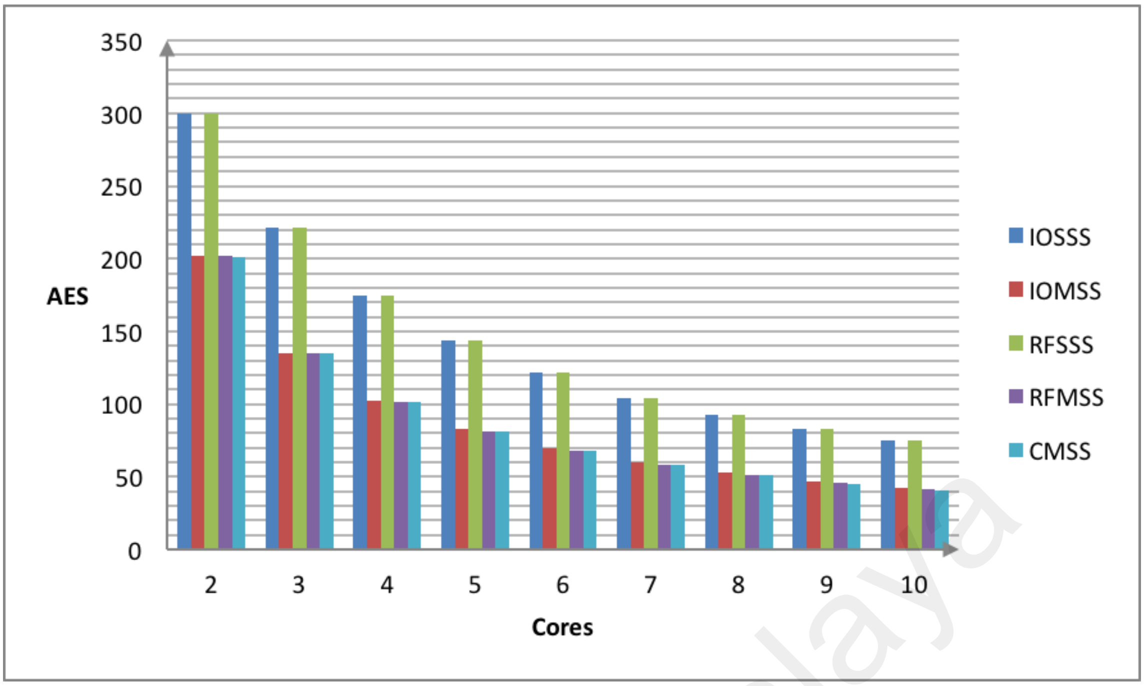


Figure 5.12: Sketching the results in Table 5.17

The results clearly show an improvement in the performance of the models when the number of cores increases, in addition, the AES values in the 20×20 example are higher than those in the 10×10 example due to the problem size. On the other hand, the results of the IOSSS and RFSSS are distinguishable since they steal single threads while the other schedulers show better performance.

5.6 Discussion

There are three major factors that have direct influence on the results, they are:

(a) The number of cores and problem size: Although these are two different factors, nevertheless, these two factors did not come up with something new. It was expected that, the increase in the number of cores with a fixed problem size will definitely lead to better results, i. e. low values of AES. The same thing can be said for the problem size. As the problem size increases, the AES values will also increase. However, the only exception may be happening in the case of binary search since this kind of D&C problem does not necessary depend on the number of cores and problem size comparing with other D&C problems. In binary search, the searched element may be found from the first step or after few steps, however in other D&C problems such as matrix multiplication, we have to multiply all the rows by all the columns. Therefore, the increase in the problem size has a strong influence on problems such as matrix multiplication, towers of Hanoi, Fibonacci; however, it may not have the same influence on the binary search problem.

(b) The stealing strategy: The stealing strategies have different issues:

I- The results of the strategies designed to steal a single thread at a time showed a poor performance compared with the good performance of those strategies designed to steal more than one thread at a time. The reason behind this goes to the extra number of execution steps in the single stealing strategies. On the other hand, the single stealing strategies are simple and easy to implement. In addition, the single stealing strategies may be preferred when the system deals with heavy threads (a thread that carries a lot of computations, and at the same time, the number of threads in all the cores are few. In such a case, it will be more convenient to use single stealing strategies such as IOSSS or RFSSS.

II- The location of the main thread is decided by the programmer prior to the simulation process. Choosing the first core has an effect on the IOSSS and IOMSS results. Changing the location of the main thread will weaken these two strategies since the stealing may not be from the wealthiest core. On the other hand, the results showed that IOSSS and RFSSS are convergent since the main thread resides in the first core and there is a good chance to keep the first core as the wealthiest core. However, changing the main thread’s location to another core, say in the middle of cores, will not have an effect on the RFSSS or RFMSS since the victim core is chosen to be the wealthiest core no matter its location.

III- The CMSS showed the best performance, however, this strategy is costly since it deals with all the cores. In the future, the number of cores may reach hundreds or even thousands. It will be highly costly to instruct the majority of the cores to give up some of their threads for a single or even few idle cores. This is why PMSS can be more convenient than CMSS in such cases.

(c) The number of generated threads in one step.

One of the important things that the researcher has achieved in this study is the development of partitioning techniques that suit multicore technology. The results showed that the performance of the model would be better when more threads can be generated in one step. This is because the number of stolen threads decreases when cores become saturated with threads. However, one of the D&C problems represents an exceptional case to what has been mentioned above. THLLS will not be able to generate more than two threads at a time, simply because there is no chance to move more than one disk at a time. The movements' generation should be serial, and this is the weakness of the concurrency requirement in this study. Therefore, we can conclude that not every D&C problem fully coincided with multicore technology.

An important issue that the researchers addresses in this section is that it would be vital to compare the results of this study with other studies' results for the sake of highlighting the strength and weakness points of the researcher's work. In other words, running a benchmark test that shows the quality of this study versus other studies would be significant, however, this is could not be achieved. The main reason behind this is due to the lack of finding similar studies. This is because this study has been built on three elements that work together to support the uniqueness of the results:

(a) Workload Partitioning Techniques

The subject of D&C problems is not new; there are too many papers that studied the characteristics of such problems. However, in this study, the researcher develops new techniques for partitioning the workload of D&C problems being solved on a multicore environment. Most researches that cover D&C problems focus on a single-processor or parallel processors architectures where each processor has its own memory. However, this study focuses on solving D&C problems on a multicore architecture where all the cores share a common memory. This will definitely generates a class of results that cannot be compared with other studies that have been implemented in totally different architectures.

(b) Workload Balancing Strategies

The work-stealing strategies developed by the researcher have strong influence on the results. The behaviour of the modelled systems varies depending on the adopted strategy. For instance, using a single-stealing strategy differs from multi-stealing strategy. The first strategy suits the systems that have relatively few number of heavy weight threads while the second strategy suits the systems that have many light weight threads. In spite that these strategies reach the same result; yet, these strategies consume a different number of execution steps which make any comparison with other studies illogical and unrealistic.

(c) The Modelling Language

The modelling language has a clear impact in supporting the uniqueness of the results generated by this study. The researcher has chosen CPN language for modelling the proposed models since this language is dedicated for modelling concurrent systems, and since we have more than one core working concurrently, then the CPN language would be the right choice for modelling. However, modelling concurrent systems is a difficult task due to the non deterministic behaviour of such systems. In CPN models, more than one transition can be executed at the same time and the process of choosing the transitions is completely randomized. That is, in each run, we may have a different sequence of transitions. This is in contrast with the imperative languages such as C-like languages where a pre determined sequence of execution steps has to be followed. In other words, a CPN model may run in a different sequence each time the model is executed because the language is built on picking transitions randomly.

In conclusion, the three elements: the workload partitioning techniques, workload balancing strategies, and the modelling language all lead to distinctive results. Any comparison with other studies must be built on the basis on having the same modelling language and the same utilized architecture.

5.7 Summary

The researcher started this chapter by giving a glimpse on the importance of simulation and modelling. Then, the researcher provided a quick look into the main features of CPN-Tool and its GUI. Following that, the researcher explained how the outputs of the simulation and monitoring processes have been registered. Next, the researcher presented the AES equation which represents the criteria adopted by the researcher in this study to obtain a trade-off between the results of simulation. Subsequently, the researcher showed the results of executing of the CPN models for every D&C problem. The researcher organized the results (AES values) insides tables and sketched them as histograms by using MS-Excel. The tables and histograms showed the relation between the increases in the number of cores versus the redistribution strategies proposed by the researcher.

CHAPTER 6: CONCLUSION AND FUTURE WORK

6.1 The Problem Addressed by This Study in Brief

This study addresses one of the important challenges facing the software both in the academic and the industry fields. It is the adaptation of software with the multicore environment. This adaptation is taken as a primary concern of software companies since the early models of multicore computers. The hardware industry succeeded in solving the obstacles that face the single-processor products through the replication of the processing elements. Nevertheless, software could not adapt easily with such development in the hardware. There is still a gap between the hardware and the software in the sense that software developers could not duplicate the speed of their products as the hardware manufacturers replicated the cores into dual, quad, and octal cores. Part of the problem goes back to the computer architecture itself where having a common main memory creates a kind of competition between the cores that are trying to utilize this memory at the same time. However, the main problem lies with the software itself where the majority of the current software being designed to be run on a single-processor environment, the process of mapping the software onto the multicore architecture becomes the developers' nightmare. Despite the improvements that have been made on the software toward the enhancement of the concurrency characteristic, yet software need to be more improved to achieve better performance.

6.2 The Purpose of the Thesis in Brief

In this thesis, the researcher has directed his effort to deal with a class of algorithms, i.e. D&C, which plays a major role in scientific and non-scientific applications. However, this class of software still lacks the adaptation with a multicore environment (Miller & Vandome, 2010). Precisely, the researcher found that the concurrent characteristic in D&C techniques is still low and needs to be improved. Therefore, this thesis is concerned with solving D&C problems (Fibonacci Series, Towers of Hanoi, Binary Search, and Matrix Multiplication) on a multicore environment in a way that can achieve a high level of concurrency.

6.3 The Achievement of Research Objectives and Thesis Contribution

In this section, the researcher reviews the objectives that have been stated in Section 1.4. The researcher re-examines these objectives and explains what has been achieved for these objectives. In addition, the researcher addresses the research's contributions that have been achieved in this study.

6.3.1 The Achievement of Research Objectives

The researcher has achieved the research objectives stated in Section 1.4 of this study as follows:

Objective (a): The proposed workload distribution scheduler was able to manage threads distribution among the cores. If any core becomes idle, the scheduler immediately checks whether there is any core that has extra threads, in such a case, the scheduler steals some of those extra threads and gives them to the idle core. As a result, no core is left idle unless the rest of the cores have a very small number of threads and at which the process of stealing threads becomes useless. In addition, the scheduler achieved the objectives of being scalable and concurrent. The scheduler showed complete flexibility in dealing with any number of cores (but only maximum ten cores had been tried) and only restricted by the modelling tool capabilities. Finally, concurrency always was one of the main objectives in this study. The scheduler mechanism in redistribution of the threads led to an increase of concurrency level among the cores.

Objective (b): The proposed core scheduler dealt with the D&C problems separately. In other words, the researcher achieved the second objective through designing a separate core scheduler for each D&C problem. In spite of the differences in the way of designing the threads and partitioning the workload, yet every proposed core scheduler creates a tree (binary or non binary) of threads. In addition, the third objective of this scheduler which is related to solving the D&C problems has also been achieved. That is to say, in addition to be able to partition the workload, the scheduler uses its threads to solve part of the given D&C problem.

Objective (c): This objective has been achieved through using the elements of CPN in designing the models. The researcher relied on the SML in programming the mechanisms of workload distribution scheduler and the core schedulers. Therefore, there were several mechanisms of the core schedulers' since we have different D&C problems. On the other hand, although we have one workload distribution scheduler, nevertheless, the researcher proposed different strategies written in SML to control workload distribution.

Objective (d): The researcher has employed the facilities of the CPN-Tool towards the reduction of the idleness of the cores. The use of the GUI of the tool clearly shows the execution of the schedulers. This has had a significant benefit in confirming the right execution of partitioning and redistributing the threads, in addition, to calculating partial results. Moreover, the GUI of the tool confirmed that no core remained idle, and at the same time, there was a chance to change that core to non-idle. In other words, the GUI of the tool had a significant impact on performing the simulation and monitoring processes.

6.3.2 Thesis Contribution

The research designed concurrent multithreaded models using CPN as a modelling language and CPN-Tool as the modelling tool. These models are able to solve D&C problems (Fibonacci Series, Binary Search, Towers of Hanoi, and Matrices Multiplications) on multicore environment. The research contribution can be stated as follows:

(a) For each D&C problem, the researcher designed nine models, they are: two-core, three-core, four-core, five-core, six-core, seven-core, eight-core, nine-core, and ten-core models. The models are expandable to add more cores.

(b) Every designed model has the ability to redistribute its threads by using one of the redistribution work-stealing based strategies: InOrderSingleStealing, InOrderMultiStealing, RichestFirstSingleStealing, RichestFirstMultiStealing, CompleteMultiStealing, PartialMultiStealing strategies. These mechanisms vary in their ability (efficiency) and simplicity; however, they all seek to balance the threads among the cores.

(c) The researcher has developed new distinct mechanisms to partition the workload of the D&C problems, i. e. a mechanism for each D&C problem. The mechanisms suit well the multicore environment because of their ability in partitioning the workload quickly which makes it easier for the redistribution strategies in reallocating the threads.

The mechanisms also show high flexibility in dealing with different sizes of D&C problems.

(d) The distribution strategies work in harmony along with partitioning mechanism to reduce the idleness of the cores and raising the concurrency level inside the models.

6.4 Conclusion of Workload Partitioning

(a) In this study, the researcher has proposed a representation to the D&C workload called it a “Thread”, where a thread is n-tuple of parameters. Every D&C problem is represented with a single thread. A thread can be divided into two or more sub threads. A thread has parameters which signify the elements of the D&C problem. For instance, a binary search thread holds the name of the array (list), the start index, the end index, and the searched element. In general, the threads’ parameters vary in their number and types depending on the type of D&C problem.

(b) To partition the workload, the researcher proposed several mechanisms (low-level schedulers) that partition the workload (main thread) into sub threads. Every D&C problem has its own scheduler that works on partitioning its threads. The way of partitioning and the number of generated threads vary from scheduler to scheduler; in addition, some schedulers may have different ways of partitioning. For instance, the researcher proposes two schedulers for the matrix multiplication problem: the MMLLS for generating two threads at a time, and DMMLLS for generating many threads at a time.

In general, the schedulers that produce only two sub threads at a time show the worst performance, while the one that produces many threads at a time prove its efficiency. The reason behind this is that, with the latter type of schedulers, more idle cores can be reinitiated to work at the same time. Therefore, the Towers of Hanoi scheduler was the worst since this game can produce a move and two sub threads at a time. The game is actually not suitable for concurrent environment since there is no chance to move two or more disks at a time. On the other hand, both Binary Search and Matrix Multiplication schedulers prove their efficiency since they can generate many threads at a time. In the Binary Search case, many searching areas can be checked at the same time. The same thing can be said in the Matrix Multiplication where the rows from the first matrix can be multiplied with the columns from the second matrix at the same time. On the other hand, the Fibonacci scheduler can be redirected to generate two, four, eight, sixteen, etc threads or more. The higher the number of generated threads at a time, the better the performance that can be achieved.

(c) The guard mechanism related to the core scheduler proves its effectiveness in enabling / disabling the core for processing threads. The mechanism unlocks the core when the core has threads inside it and there are no idle cores in the model, otherwise, the scheduler is locked to open the way to the HLS to redistribute the threads. The behaviour of the guard guarantees the achievement of the concurrency in execution and the justice in distributing the threads among the cores. In general, the guard mechanism is important for balancing the workload among the cores.

6.5 Conclusion of Workload Distribution

(a) The creation of threads will not be effected till the availability of mechanisms that distribute those threads to the cores. Therefore, the researcher proposed five mechanisms (high-level schedulers), namely IOSSS, IOMSS, RFSSS, RFMSS, PMSS and CMSS. These strategies work on the basis of work-stealing. However, in this study, the stealing process is centralized, in the sense that there is a special core that is responsible for coordinating stealing threads from the victim cores and submits these threads to the thief cores.

(b) The IOSSS is easy to implement and does not need extra calculation. This strategy can be so effective when the majority of the cores are wealthy with threads and only one or two cores are in an idle situation. Applying this strategy will not bother or interrupt other cores, and the thief cores can be fed easily from the first encountered wealthy core. However, if there are several thief cores, then applying this strategy is not worth while since as soon as a thief core gets its single thread, it turns to be thief again as soon as the core has finished processing its thread.

(c) The IOMSS is also easy to implement as the IOSSS, however, the IOMSS is more effective since it enforces the chosen victim core to give up half of its thread to the thief core. As a result, the thief core that has got the threads spends more time till it becomes a thief again comparing with its counterpart in the IOSSS where the thief core gets a single thread.

(d) The efficiency of the RFSSS lies between IOSSS and IOMSS. In general, the RFSSS performance is better than the IOSSS when there are more than one thief core. In this case, the IOSSS may deal with a poor victim; however, the RFSSS locates the wealthiest core. As a result, the RFSSS reduces the time of calling the distribution process. As in the IOSSS, the RFSSS can be useful when most of the cores are wealthy and few of the cores are idle.

The deficiency in the IOSSS and RFSSS is caused by the single stealing. The RFMSS overcomes this deficiency by allowing sharing the threads of the wealthiest core. The RFMSS shows better performance than the IOMSS since the thief core gets a share of threads higher than the share comes by the IOMSS since the chosen victim in IOMSS may not be the wealthiest victim.

(e) The CMSS can be evaluated as the best distribution strategy. It has the ability to balance the workload among all the cores; victims and thieves but that comes with a price. This strategy needs extra calculation to fit the amount of threads that each core must get, in addition, all the cores will be hindered for deciding which core gives and which core gets. As a result, this strategy suits the case where there are many thieves and many victims.

(f) The PMSS represents a special case in this study. This mechanism is based on excluding the cores that have a small number of threads and focusing on zero-thread cores and wealthy cores. It will be waste of time to interrupt poor cores especially when the system has several hundreds of cores. It is more convenient to spotlight on wealthy and zero-thread cores. However, since the CPN-Tool executes a single transition at a time, the PMSS could not achieve distinguished results. The PMSS results were identical to the CMSS; therefore, the researcher did not include these results with other strategies' results.

(g) The guard mechanism that is related to the HLS has special significance since it controls the distribution of threads among the cores. This guard opens the way to transfer the threads only when there are victim and thief cores at the same time, otherwise, the guard deactivates such transfer.

6.6 CPN Modelling of the Mechanism

The interesting thing in CPN is that the repeated execution of any model may happen in different paths. That is, the execution paths of the models are not unique. This is because of the non deterministic nature of this modelling language which makes it suitable to model multicore environment. However, the different execution paths lead to the same destination and generate the desired result. On the other hand, the CPN-Tool was the right choice for this study. The GUI provided by this tool enables the researcher to interact with model during the design, simulation, and monitoring processes. However, the tool is not able to execute two transitions at the same time. Although, this might be more helpful for the study, nevertheless, the tool provides a complete overview on the active transitions in the entire model.

6.7 Findings of the Simulation

(a) The simulation results have shown a high stability towards the increase in the number of cores. The simulation of all the models shows a gradual and stable improvement in the values of the average of execution steps as moving from a model with a low number of cores to one with a higher number of cores.

(b) The general observation of the low-level schedulers reveals that schedulers that produce only two threads at a time are not suitable for multicore environment in the sense that they cannot satisfy the cores' need. On the other hand, low-level schedulers with a higher number of threads produced at a time can easily satisfy the cores' need. In other words, the cores will be busy and they no longer need any rescheduling to the threads.

(c) The high-level schedulers vary in their influence depending on the number of threads that can be generated by the low-level schedulers. The results of the single stealing strategies converge with the multi stealing strategies when adopting the low-level schedulers with few numbers of threads. However, the performance of the high-level schedulers becomes distinctive when adopting the low-level schedulers with a high number of threads.

6.8 Future Work

6.8.1 Work Related to the Low-Level Schedulers

6.8.1.1 Thread’s Weight

Thread’s weight points to the amount of data that a thread is going to deal with. For instance, the thread designed for matrix multiplication is dedicated for multiplying a single row by a single column. For this point, a research can be conducted to investigate the extent of the benefit that can be achieved when designing a thread to multiply a single row by all the columns instead of one column. Increasing the weight of the thread might be useful in making the core that processes the threads more busy; however, when one of the cores becomes idle, then it would be difficult to steal parts of the threads. Therefore, there should be an accurate and practical study on the increase in the thread’s weight.

6.8.1.2 Number of Generated Threads

The number of threads that is generated by the low-level schedulers plays a main role in this study. This number, if well computed, will be the key to achieve a high level of concurrency. A research can be conducted to find a relation between the size of the problem and the number of the cores. That is, designing a function that can compute the suitable number of threads needed to be generated for a given problem size and a given number of cores.

6.8.2 Future Work Related with the High-Level Schedulers

6.8.2.1 Scenario of Launching the High-Level Scheduler

The process of launching a high-level scheduler is costly since it freezes all the cores from continuing their job. In this thesis, whenever a core becomes idle, it turns to be a thief and this thief core waits for the high-level scheduler to get thread(s). This scenario could be investigated in certain directions, such as, when one of the cores becomes idle;

then it would not be necessary to freeze all the cores. A single or a few number of victim cores can deal with this situation. Another point is, is it necessary to launch the high-level scheduler when only a single core becomes idle? In other words, is it always worthy to launch the high-level scheduler when one or two cores become idle? It is possible to highlight this point with more attention.

6.8.2.2 Diversity in High-Level Schedulers

In this thesis, the researcher suggests several high-level schedulers. However, a model can run only a predetermined distribution strategy. Now, since the contents of the cores are dynamically changed, the distribution strategy should also change to fit the current requirements of the model. For that reason, research can be conducted to make choosing of the distribution strategy dynamic, i.e. based on the number of threads in the cores, the high-level scheduler decides whether it should apply IOSSS or IOMSS, etc.

6.8.2.3 Choosing the Heaviest Threads

The high-level schedulers redistribute the threads between the cores. A research can be conducted regarding what threads should be moved and what threads should be kept inside the victim cores. For instance, in the Fibonacci case, is it worth to steal a thread that headed a large sub tree or it is better to leave it inside its core? An immediate answer to this question is to leave it inside its core since this will improve the locality. However, in the case of IOSSS or RFSSS in an environment that has few victim wealthy cores, it would be worthy to steal heavy threads so that the stolen threads make the thief cores busier.

6.9 Limitation of Schedulers

The main limitation in the LLSs is they are designed for solving D&C problems where problems can be partitioned into sub independent problems. This kind of technique does not suit a wide range of problems which makes the proposed LLSs unsuitable for non D&C problems. In addition, each LLS is oriented to solve a specific D&C problem, i.e. the design of the LLS is unique and closely related to the type of the D&C problem.

REFERENCES

Acar, U., Blelloch, A., & Blumofe, R. D. (2000). *The Data Locality of Work Stealing*. Paper presented at the The twelfth annual ACM symposium on Parallel algorithms and architectures Bar Harbor, Maine, United States

Acar, U. A., Charguéraud, A., & Rainey, M. (2013). *Scheduling parallel programs by work stealing with private deques.* Paper presented at the Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming.

Agrawal, K., Leiserson, C. E., He, Y., & Hsu, W. J. (2008). Adaptive work-stealing with parallelism feedback. *ACM Transactions on Computer Systems (TOCS), 26*(3), 7.

Arora, N. S., Blumofe, R. D., & Plaxton, C. G. (2001). Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems, 34*(2), 115-144. doi: 10.1007/s00224-001-0004-z

Belal, M. A Modified Work Stealing Algorithm Based on Randomized Spanning Trees Approach.

Berenbrink, P., Friedetzky, T., & Goldberg, L. A. (2001). *The natural work-stealing algorithm is stable.* Paper presented at the The 42th IEEE Symposium on Foundations of Computer Science (FOCS).

Blumofe, R. D. (1995). *Executing multithreaded programs efficiently.* Massachusetts Institute of Technology.

Blumofe, R. D., & Leiserson, C. E. (1999). Scheduling multithreaded computations by work stealing. *Journal of the ACM, 46*, 720-748.

Breshears, C. (2009). *The art of concurrency: A thread monkey's guide to writing parallel applications*: " O'Reilly Media, Inc.".

Bryant, R., & David Richard, O. H. (2003). *Computer systems: a programmer's perspective*: Prentice Hall.

Burton, F. W., & Sleep, M. R. (1981). *Executing functional programs on a virtual tree of processors.* Paper presented at the Proceedings of the 1981 conference on Functional programming languages and computer architecture.

Cao, Y., Sun, H., Qian, D., & Wu, W. (2011). *Stable Adaptive Work-Stealing for Concurrent Multi-core Runtime Systems.* Paper presented at the High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on.

Casavant, T. L., & Kuhl, J. G. (1988). A taxonomy of scheduling in general-purpose distributed computing systems. *Software Engineering, IEEE Transactions on, 14*(2), 141-154.

Casey, L. M. (1981). Decentralized Scheduling. *Journal of Australian Computer, 13*, 58-63.

Chase, D., & Lev, Y. (2005). *Dynamic circular work-stealing deque*. Paper presented at the The seventeenth annual ACM symposium on Parallelism in algorithms and architectures, Las Vegas, Nevada, USA.

Chen, Q., Guo, M., & Huang, Z. (2012). Adaptive Cache Aware Bi-tier Work-stealing in Multi-socket Multi-core Architectures.

Cheung, A. K. Y., & Jacobsen, H.-A. (2006). *Dynamic load balancing in distributed content-based publish/subscribe*: Springer.

Chhabra, A., Singh, G., Waraich, S. S., Sidhu, B., & Kumar, G. (2006). Qualitative parametric comparison of load balancing algorithms in parallel and distributed computing environment. *Proc. World Academy of Science, Engineering and Technology*, 39-42.

. Compare-and-swap. (4 February 2014), from http://en.wikipedia.org/wiki/Compare-and-swap

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (Third ed.): MIT Press.

Davis, R. I., & Burns, A. (2011). A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (CSUR), 43*(4), 35.

Diekmann, R., Monien, B., & Preis, R. (1997). *Load balancing strategies for distributed memory machines.* Paper presented at the Multi-Scale Phenomena and Their Simulation.

Dinan, J., Olivier, S., Sabin, G., Prins, J., Sadayappan, P., & Tseng, C.-W. (2008). A message passing benchmark for unbalanced applications. *Simulation Modelling Practice and Theory, 16*(9), 1177-1189.

Ding, X., Wang, K., Gibbons, P. B., & Zhang, X. (2012). *BWS: Balanced Work Stealing for Time-Sharing Multicores*. Paper presented at the EuroSys '12 Proceedings of the 7th ACM european conference on Computer Systems Bern, Switzerland.

Eager, D. L., Lazowska, E. D., & Zahorjan, J. (1986). A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance evaluation, 6*(1), 53-68.

Fatourou, P., & Spirakis, P. (2000). Efficient scheduling of strict multithreaded computations. *Theory of Computing Systems, 33*(3), 173-232.

Feitelson, D. G., & Rudolph, L. (1995). *Parallel job scheduling: Issues and approaches*. Paper presented at the Job Scheduling Strategies for Parallel Processing.

Frigo, M., Leiserson, C. E., & Randall, K. H. (1998). *The implementation of the Cilk-5 multithreaded language*. Paper presented at the ACM SIGPLAN Notices.

Gansner, E. R., & Reppy, J. H. (2004). *The standard ML basis library*: Cambridge University Press.

Gautier, T., Lima, J. V. F., Maillard, N., & Raffin, B. (2013). *Locality-Aware Work Stealing on Multi-CPU and Multi-GPU Architectures*. Paper presented at the 6th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG).

Geer, D. (2005). Chip makers turn to multicore processors. *Computer, 38*(5), 11-13. doi: 10.1109/mc.2005.160

Guo, Y., Barik, R., Raman, R., & Sarkar, V. (2009). *Work-first and help-first scheduling policies for async-finish task parallelism*. Paper presented at the Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on.

Halstead Jr, R. H. (1984). *Implementation of Multilisp: Lisp on a multiprocessor*. Paper presented at the Proceedings of the 1984 ACM Symposium on LISP and functional programming.

Held, G. (1986). *IBM PC and PC XT User's Reference Manual* (Vol. 2): Longman Higher Education.

Hendler, D., Lev, Y., Moir, M., & Shavit, N. (2005). A Dynamic-Sized Non-blocking Work Stealing Deque. *Journal of Distributed Computing, 18*(3), 189-207.

Hendler, D., & Shavit, N. (2002). Non-blocking Steal-Half Work Queues. *In The twenty-first annual symposium on Principles of distributed computing*

Hendrickson, B., & Devine, K. (2000). Dynamic load balancing in computational mechanics. *Computer Methods in Applied Mechanics and Engineering, 184*(2), 485-500.

Herlihy, M. (2007). *The multicore revolution: the challenges for theory*. Paper presented at the Proceedings of the 27th international conference on Foundations of software technology and theoretical computer science, New Delhi, India.

Jensen, K. (1998). Special section on coloured Petri nets. *Int. J. Software Tools Technol. Transfer, 2*(2), 95-191.

Jensen, K., Christensen, S., Kristensen, L. M., & Westergaard, M. CPN Tools Web Page, 2014, from http://cpntools.org/

Jensen, K., & Kristensen, L. M. (2009). *Coloured Petri Nets: Modeling and Validation of Concurrent Systems*: Springer-Verlag New York Inc.

Jensen, K., Kristensen, L. M., & Wells, L. (2007). Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT), 9*(3), 213-254.

Jerry, B. (1984). *Discrete-event system simulation*: Pearson Education India.

Karp, R. M., & Zhang, Y. (1993). Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM (JACM), 40*(3), 765-789.

Knuth, D. E. (1968). The Art of Computer Programming: Fundamental Algorithms, Vol. I: Addison-Wesley.

Kristensen, L. M. (2000). *State Space Methods for Coloured Petri Nets.* University of Aarhus.

Kwok, Y.-K., & Ahmad, I. (1999). Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR), 31*(4), 406-471.

Lea, D. (2005). The java. util. concurrent synchronizer framework. *Science of Computer Programming, 58*(3), 293-309.

Lu, W., & Adviser-Gannon, D. (2009). Exploiting multi-core processors for the service oriented architecture paradigm: parallel xml processing and concurrent service orchestration.

Mack, C. A. (2011). Fifty Years of Moore's Law. *Semiconductor Manufacturing, IEEE Transactions on, 24*(2), 202-207.

Mao, Z. M., So, H.-S. W., & Woo, A. (1998). JAWS: A Java work stealing scheduler over a network of workstations. *The University of California at Berkeley, Berkeley, USA, Technical*.

Mattheis, S., Schuele, T., Raabe, A., Henties, T., & Gleim, U. (2012). Work stealing strategies for parallel stream processing in soft real-time systems *Architecture of Computing Systems–ARCS 2012* (pp. 172-183): Springer.

Miller, F. P., & Vandome, A. F. (2010). *Divide and Conquer Algorithm*: Alphascript Publishing.

Mitzenmacher, M. (1998). *Analyses of load stealing models based on differential equations.* Paper presented at the Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures.

Mollick, E. (2006). Establishing Moore's law. *Annals of the History of Computing, IEEE, 28*(3), 62-75.

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE, 77*(4), 541-580. doi: 10.1109/5.24143

Nambiar, R., & Poess, M. (2011). Transaction Performance vs. Moore’s Law: A Trend Analysis *Performance Evaluation, Measurement and Characterization of Complex Systems* (pp. 110-120): Springer.

Neill, D., & Wierman, A. (2009). On the benefits of work stealing in shared-memory multiprocessors. *Department of Computer Science, Carnegie Mellon University, Tech. Rep*.

Nieuwpoort, R. v., Maassen, J., Kielmann, T., & Bal, H. E. (2001). Satin: Simple and efficient Java-based grid programming. *Scalable Computing: Practice and Experience, 6*(3).

Nogueira, L., Fonseca, J. C., Maia, C., & Pinho, L. M. (2012). *Dynamic Global Scheduling of Parallel Real-Time Tasks.* Paper presented at the Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on.

Nogueira, L. M., Pinho, L. M., Fonseca, J., & Maia, C. (2013). *On the use of Work Stealing Strategies in Real Time Systems.* Paper presented at the Workshop on High-performance and Real-time Embedded Systems, Berlin, Germany.

Olivier, S. L., & Adviser-Prins, J. F. (2012). Locality awareness for task parallel computation.

Osman, A., & Ammar, H. (2002). Dynamic load balancing strategies for parallel computers. *Sci. Ann. Cuza Univ., 11*, 110-120.

Peterson, J. L. (1977). Petri nets. *ACM Computing Surveys (CSUR), 9*(3), 223-252.

Quintin, J.-N., & Wagner, F. (2010). Hierarchical work-stealing *Euro-Par 2010-Parallel Processing* (pp. 217-229): Springer.

Raetz, G. (1987). *Sequent general purpose parallel processing system*. Paper presented at the Northcon\87.

Rainey, M. A. (2010). *Effective scheduling techniques for high-level parallel programming languages*: THE UNIVERSITY OF CHICAGO.

Robison, A., Voss, M., & Kukanov, A. (2008). *Optimization via reflection on work stealing in TBB.* Paper presented at the Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on.

Rudolph, L., Slivkin-Allalouf, M., & Upfal, E. (1991). *A simple load balancing scheme for task allocation in parallel machines.* Paper presented at the Proceedings of the third annual ACM symposium on Parallel algorithms and architectures.

Squillante, M. S., & Lazowska, E. D. (1993). Using processor-cache affinity information in shared-memory multiprocessor scheduling. *Parallel and Distributed Systems, IEEE Transactions on, 4*(2), 131-143.

Squillante, M. S., & Nelson, R. D. (1991). *Analysis of task migration in shared-memory multiprocessor scheduling* (Vol. 19): ACM.

Strang, G. (2011). Introduction to linear algebra.

Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal, 30*(3), 202-210.

Sutter, H., & Larus, J. (2005). Software and the concurrency revolution. *Queue, 3*(7), 54-62.

Tardieu, O., Wang, H., & Lin, H. (2012). *A work-stealing scheduler for X10's task parallelism with suspension.* Paper presented at the ACM SIGPLAN Notices.

Tchiboukdjian, M., Danjean, V., Gautier, T., Lementec, F., & Raffin, B. (2010). A work stealing algorithm for parallel loops on shared cache multicores. *HPPC 2010*, 18.

. Threading Building Blocks. Retrieved March 20, 2014, 2014, from https://www.threadingbuildingblocks.org/

Tucker, R., Barlow, N., & Stuart, L. (2012). THE BACKGROUND AND IMPORTANCE OF EXPLOITING MULTIPLE CORES: A CASE STUDY IN NEUROPHYSIOLOGICAL VISUALIZATION.

Tzannes, A. (2012). Enhancing Productivity and Performance Portability of General-Purpose Parallel Programming.

Ullman, J. (1998). *Elements of ML Programming*: Prentice-Hall.

Van Nieuwpoort, R. V., Kielmann, T., & Bal, H. E. (2000). *Satin: Efficient parallel divide-and-conquer in java.* Paper presented at the Euro-Par 2000 Parallel Processing.

Van Nieuwpoort, R. V., Kielmann, T., & Bal, H. E. (2001). *Efficient load balancing for wide-area divide-and-conquer applications.* Paper presented at the ACM SIGPLAN Notices.

Vrba, Ž., Espeland, H., Halvorsen, P., & Griwodz, C. (2009). Limits of work-stealing scheduling *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science* (Vol. 5798, pp. 280-299): Springer.

Wang, & Morris, R. J. T. (1985). Load sharing in distributed systems. *Computers, IEEE Transactions on, 100*(3), 204-217.

Wang, Y., Ji, W., Shi, F., & Zuo, Q. (2013). *A work-stealing scheduling framework supporting fault tolerance.* Paper presented at the Proceedings of the Conference on Design, Automation and Test in Europe.

Wells, L. (2002). *Performance analysis using coloured Petri nets.* Paper presented at the Modeling, Analysis and Simulation of Computer and Telecommunications Systems, 2002. MASCOTS 2002. Proceedings. 10th IEEE International Symposium on.

Wikipedia. (2013). Parallel Extensions, from http://en.wikipedia.org/wiki/Parallel_Extensions

Wikipedia. (2014a). Multi-core processor, 2015, from http://en.wikipedia.org/wiki/Multi-core_processor

Wikipedia (Ed.). (2014b). *Threading Building Blocks*.

Zamanifar, K., Nematbakhsh, N., & Sadjady, R. S. (2010). *A New Load Balancing Algorithm in Parallel Computing*. Paper presented at the Communication Software and Networks, 2010. ICCSN'10. Second International Conference on.

Zhang, B.-Y., Mo, Z.-Y., Yang, G.-W., & Zheng, W.-M. (2007). *Dynamic load-balancing and high performance communication in JCluster*. Paper presented at the Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International.

Zhang, B. Y., Yang, G. W., & Zheng, W. M. (2006). Jcluster: an efficient Java parallel environment on a large-scale heterogeneous cluster. *Concurrency and Computation: Practice and Experience, 18*(12), 1541-1557.

LIST OF PUBLICATIONS AND PAPERS PRESENTED

(a) Al-Obaidi, A., & Lee, S. P. (2011). A Concurrent Multithreaded Scheduling Model for Solving Fibonacci Series on Multicore Architecture. International Journal of Advancements in Computing Technology, 3(2), 24 – 37

(b) Al-Obaidi, A., & Lee, S. P. (2011, June 27- 29, 2011). A Concurrent Coloured Petri Nets Model for Solving Binary Search Problem on a Multicore Architecture. Paper presented at the The 2nd International Conference on Software Engineering and Computer Systems, University Malaysia Pahang, Malaysia.

(c) Al-Obaidi, A. (2012). A Concurrent Multi-Stealing Scheduler Model For Divide And Conquer Problems. Malaysian Journal of Computer Science 25(4), 177-195.

(d) Al-Obaidi, A., & Lee, S. P. (2012). A multithreaded scheduling model for solving the Tower of Hanoi game in a multicore environment. Maejo International Journal of Science and Technology, 6(2), 282-296.

(e) Al-Obaidi, A., & Lee, S. P. (2012, 3-5 July). A Partial Multi Stealing Scheduling Model for Divide and Conquer Problems. Paper presented at the International Conference on Computer and Communication Engineering (ICCCE 2012), Kuala Lumpur, Malaysia.

APPENDIX I

COLORED PETRI NETS

This appendix is dedicated for giving a brief explanation with examples to the modeling language that is used in this thesis. Colored Petri Nets (CPN) is a language designed for modelling and validating concurrent systems (Jensen & Kristensen, 2009). The language combines Petri Nets and SML where the aspect of Petri Nets is responsible for providing the basics for modeling concurrency and communication between the elements of the models, in addition to providing the graphical notations for the designed models. On the other hand, the SML enables CPN to define data types, writing expressions and writing user defined functions besides using a rich library of built-in functions. CPN is a dynamic language; in other words, a system modeled in CPN can be executed. It is possible to represent a system's states (places) and events (transitions) that can cause the system's states to be changed. In this appendix, the researcher focuses on two subjects related with CPN: the first one is about the main elements that construct any CPN model while the second subject is related to building hierarchical CPN.

I.1 CPN Main Elements

In CPN, we can distinguish three types of graphics which constitute the elements of the model:

(a) Places

In CPN, a place is an oval shape that holds tokens. It is also referred to as system state. A token is a combination of the occurrence of the data and the data itself. Figure I.1 shows an example of a place named P1. Places in CPN have several characteristics:

I- A place has a data type (color set). In CPN, each place has a compulsory color set that can be simple such as: Integer, Boolean, String, Enumerated, or it can be compound such as: Product, Record, and List. The inscription of the place's color set is located at the lower right corner of the place. The type of the place in Figure I.1 is INT (Integer).

II- A place in CPN may have an initial value called the initial marking which is located at the upper right corner. The inscription of the initial marking consists of the tokens that reside in the place before executing the model. In Figure I.1, the initial marking has only one token which is 1`1, that is, place P1 has one occurrence of data with the value one.

III- A place in CPN may have a current marking. The current marking consists of two shapes: a circle which includes the current number of tokens in the place and a rectangle which includes the tokens' details. In Figure I.1, the circle has the number one which means we have one token and the rectangle includes 1`1 which indicates that we have one data with value one. The main difference between the initial and current markings is the first one never changed during the simulation process while the second may change.

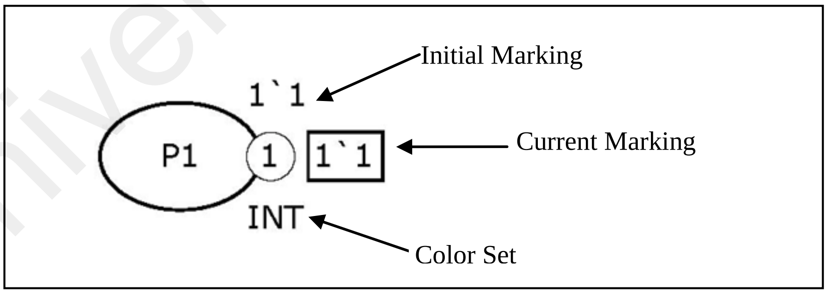
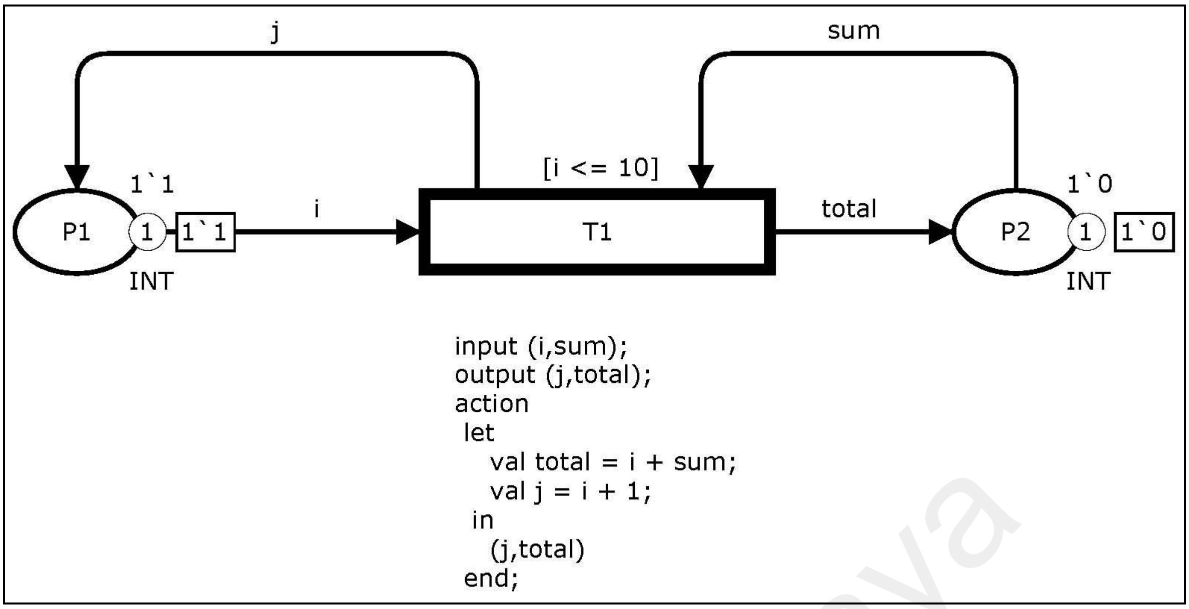


Figure I.1: An example of a place

Figure I.2 shows another example for a CPN. It has two places (P1 and P2). Both places of the same type, INT, however, they differ in the initial and current markings.



```
input (i,sum);
output (j,total);
action
 let
   val total = i + sum;
   val j = i + 1;
  in
   (j,total)
 end;
```

Figure I.2: An example of a CPN that is designed to find the summation of numbers from 1 to 10

(b) Transitions

A transition is the CPN element that when executed, can change the state of the system. A transition has a rectangle shape and it is connected with the places through directed arcs. Usually, transitions receive token(s) from input place(s) and send token(s) to the output places. In Figure I.2, there is a single transition called T1 which is connected with input place (P1) and the output place (P2). A transition may have the following characteristics:

I- A transition may have a guard which represents a Boolean expression. A value "True" of the guard allows the transition to be executed if there are enough tokens in the input places. Conversely, a guard with "False" value prevents its transition from being executed. In Figure I.2, the guard has the value [i ≤ 10] which means any token comes from the place P1 exceeds the value 10 will freeze the transition T1. Therefore, a guard works like a lock that enables and disables its transition.

II- A transition may have a code segment which enables the programmer to write code in CPN ML. This code segment is executed when the transition is executed. The code segment has input, output, and action sections. The input and output sections are dedicated for including the input and output arguments that are obtained from the input and output places. In Figure I.2, the code segment has the duty of finding the summation of the numbers 1 to10. At the beginning, the segment receives the value one from P1; it (segment) accumulates it inside the variable total and sends it to P2. Next, when the T1 is executed for the second time, it (T1) receives the value two (i = 2) and computes the summation (1 + 2) and sends it to P2. The process continues until the guard disables T1, at that time, P2 has the token (current marking) 1`55 which represents the summation for the numbers from 1 to 10.

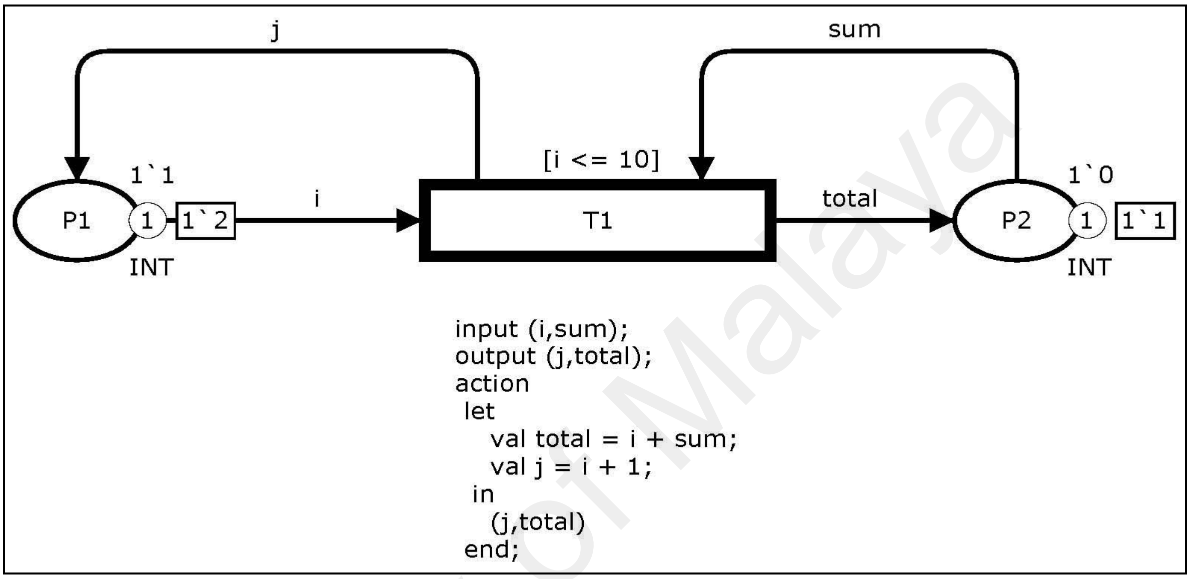
III- A transition is enabled to be executed (firing) when its guard inscription returns the value True and there are enough tokens in the input place(s), in other words, if the input place(s) have no tokens inside them, a transition cannot be executed. On the other hand, when an enabled transition fires, it transfers token(s) from the input places to the output places causing changes in the system state.

(c) Directed Arcs

The places and transitions cannot work alone. They need a kind of communication that transfers the tokens between them. The arcs are used to connect a place with a transition and a transition with a place. Any two places cannot be connected by arcs neither any two transitions. An arc may have an inscription as shown in Figure. I.2, the arc that connects P1 with T1 has the inscription i which indicates that i carries the token from P1 to T1.

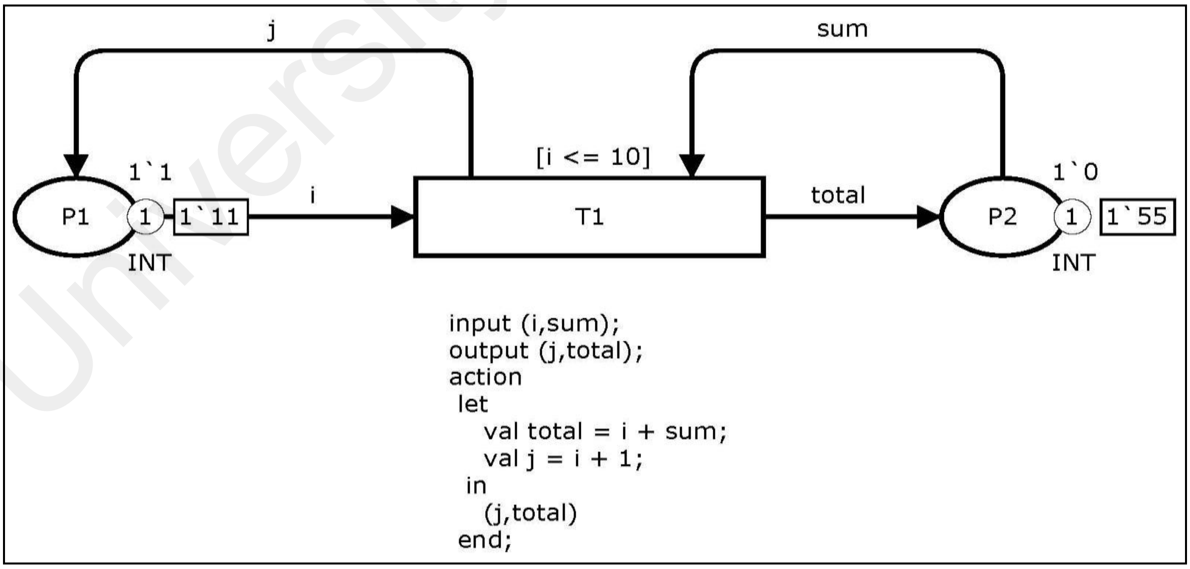
Executing the model in Figure I.2 makes T1 changes the state of P1 and P2 (Figure I.3). The current marking of P1 changes from 1`0 to 1`1, the same thing for P2, its current marking chages from 1`0 to 1`1. P1 works as a counter from 1 to 10 while P2 accumulates the numbers from 1 to 10. After executing T1 ten times, the final state

(Figure I.4) shows the P1 holds the value 11 while P2 holds the value 55 which is the summation of 1 to 10. T1 is no more able to be excuted since the value of i excced the value 10, this will violate the guard condition which say that the value of i ≤ 10. Therfore, T1 is deactivated and the simulation process is stopped.



```
input (i,sum);
output (j,total);
action
 let
    val total = i + sum;
    val j = i + 1;
  in
    (j,total)
 end;
```

Figure I.3: Executing the transition T1 for only once



```
input (i,sum);
output (j,total);
action
 let
    val total = i + sum;
    val j = i + 1;
  in
    (j,total)
 end;
```

Figure I.4: Executing the transition T1 for ten times.

In CPN, when there is more than one transition in the model, it is the simulator duty to search for the enabled transition and executes it. In case there is more than one enabled transition at the same time, this makes the simulator to choose one of them randomly.

Executing the model again from the beginning may leads to different transitions executions; in other words, there is no guarantee that executing the model again will generate the same order of transitions executions; this is because of the nondeterministic nature of CPN.

I.2 Hierarchical Structure

In CPN, it is possible to build a hierarchical structure. In this case, a main model may cooperate with sub models to interact with each other. It is similar to the relation between main and sub routine in programming languages which ultimately leads too model large concurrent systems. Figure I.5 gives an example to a simple hierarchical model. The main model (Figure I.5.a) consists of three places (Car, Color, Car and Color) and one transition (Connect). The places Car and Color each has four tokens, the ++ symbol is used to construct a multi-set of colors. The Connect transition has a double line on its border; this means that this is a substitution transition. In this case, there should be a substitution model (sub model) associated with this transition. The name of the substitution model appears as a small tag on the lower left corner of the substitution transition (Figure I.5.a). On the other hand, the substitution model (Figure I.5.b) receives the inputs from the places Car and Color and produces the output and send it to the Car and Color place. The relation between the main model and the substitution model is similar to the one between main program and sub routine in programming languages. In Figure I.5.b, the substitution model connect a car's name with a color (the ^ symbol is used for this reason). Figure I.5.c shows the main model after executing the Connect transition four times.

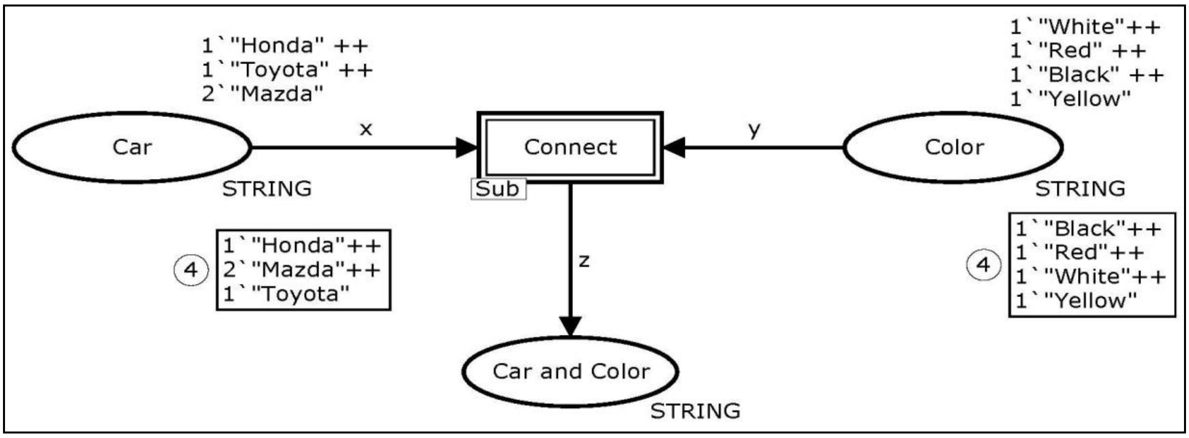


Figure I.5.a: A hierarchical CPN model. The main model

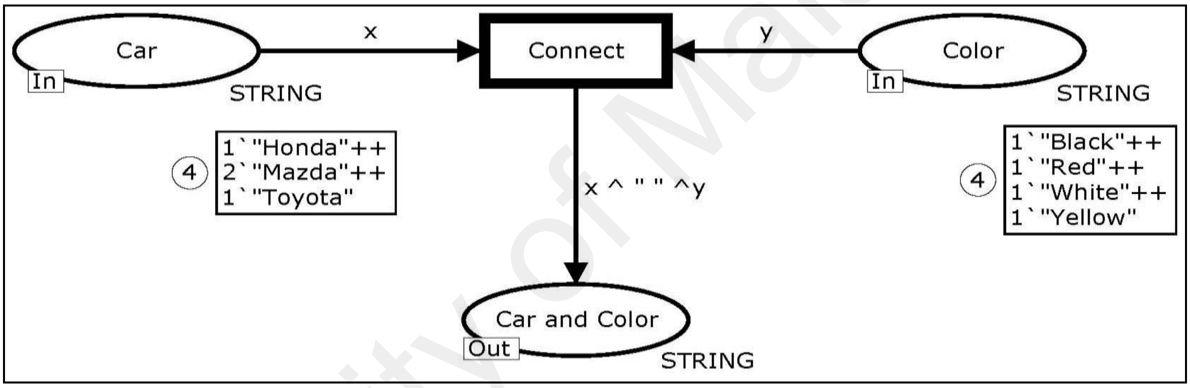


Figure I.5.b: A hierarchical CPN model. The sub model

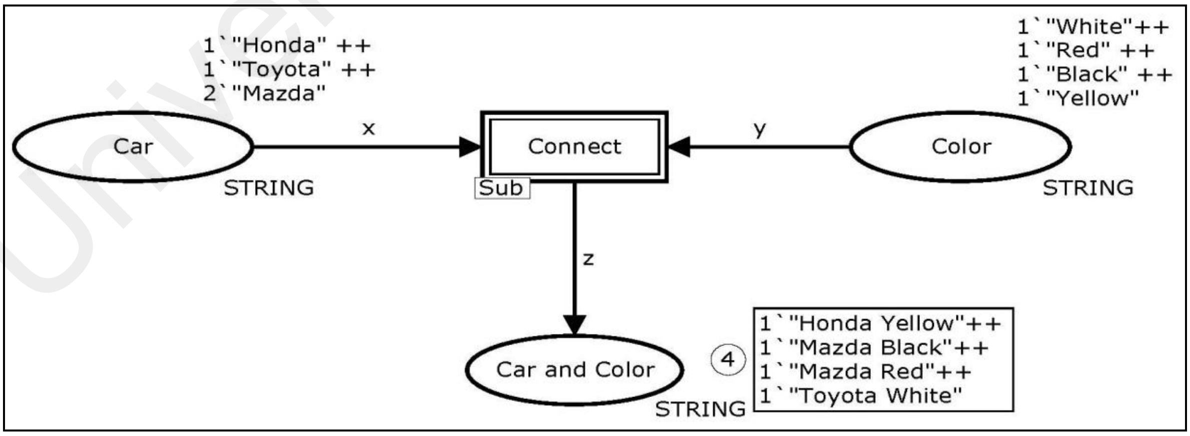


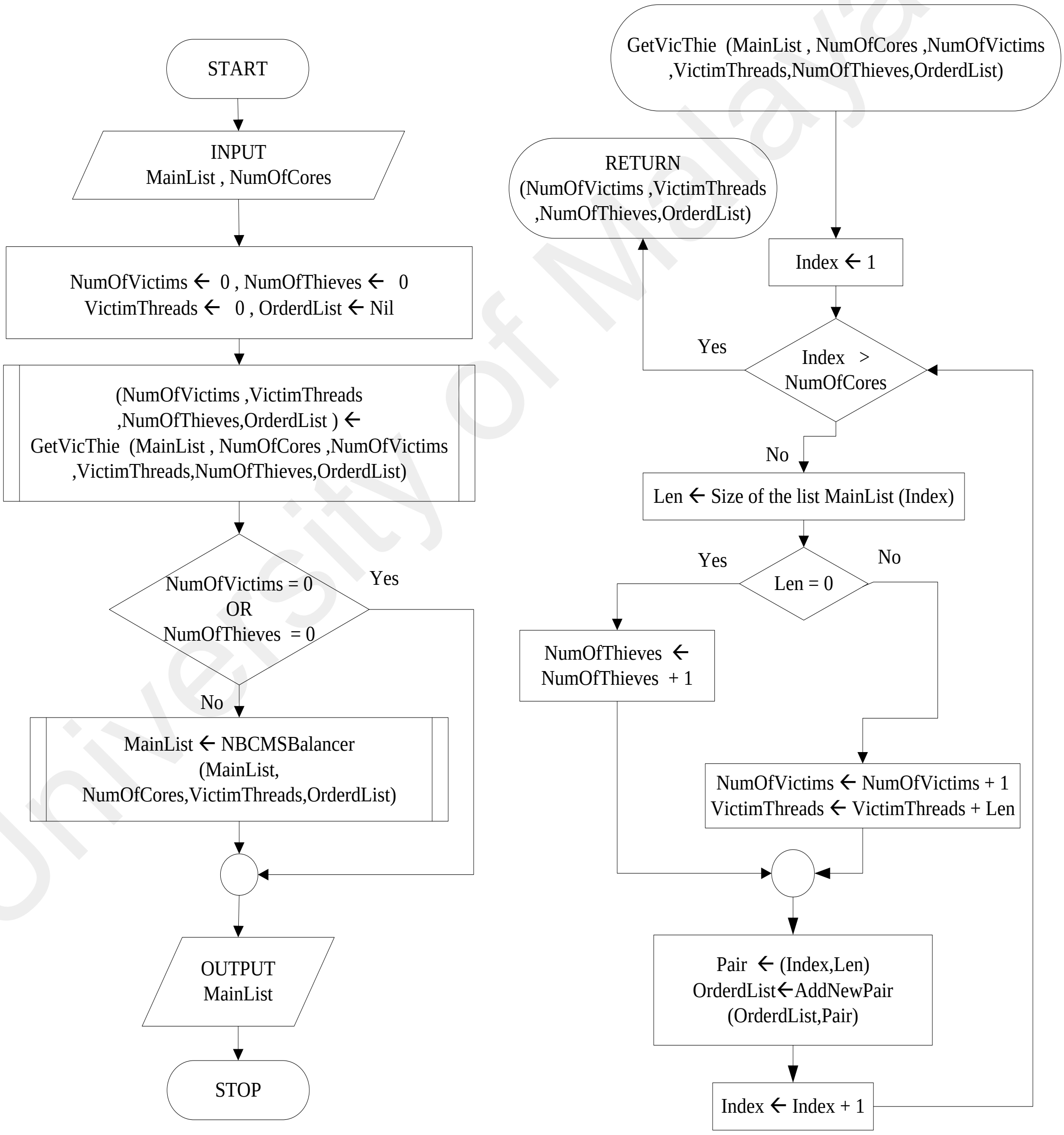
Figure I.5.c: A hierarchical CPN model. The main model after four executions to the Connect Transition

APPENDIX II

NonBiasedCMSS and NonBiasedPMSS

This appendix is dedicated for illustrating the mechanisms of NonBiasedCMSS and NonBiasedPMSS.

(a) The mechanism of NonBiasedCMSS



START
INPUT
MainList , NumOfCores
NumOfVictims ← 0 , NumOfThieves ← 0
VictimThreads ← 0 , OrderdList ← Nil
(NumOfVictims ,VictimThreads ,NumOfThieves,OrderdList) ←
GetVicThie (MainList , NumOfCores ,NumOfVictims ,VictimThreads,NumOfThieves,OrderdList)
NumOfVictims = 0
OR
NumOfThieves = 0
Yes
No
MainList ← NBCMSBalancer (MainList, NumOfCores,VictimThreads,OrderdList)
OUTPUT
MainList
STOP
GetVicThie (MainList , NumOfCores ,NumOfVictims ,VictimThreads,NumOfThieves,OrderdList)
RETURN
(NumOfVictims ,VictimThreads ,NumOfThieves,OrderdList)
Index ← 1
Yes
Index >
NumOfCores
No
Len ← Size of the list MainList (Index)
Yes
Len = 0
No
NumOfThieves ←
NumOfThieves + 1
NumOfVictims ← NumOfVictims + 1
VictimThreads ← VictimThreads + Len
Pair ← (Index,Len)
OrderdList←AddNewPair
(OrderdList,Pair)
Index ← Index + 1

Figure II.1: The NonBiasedCMSS and GetVicThie functions

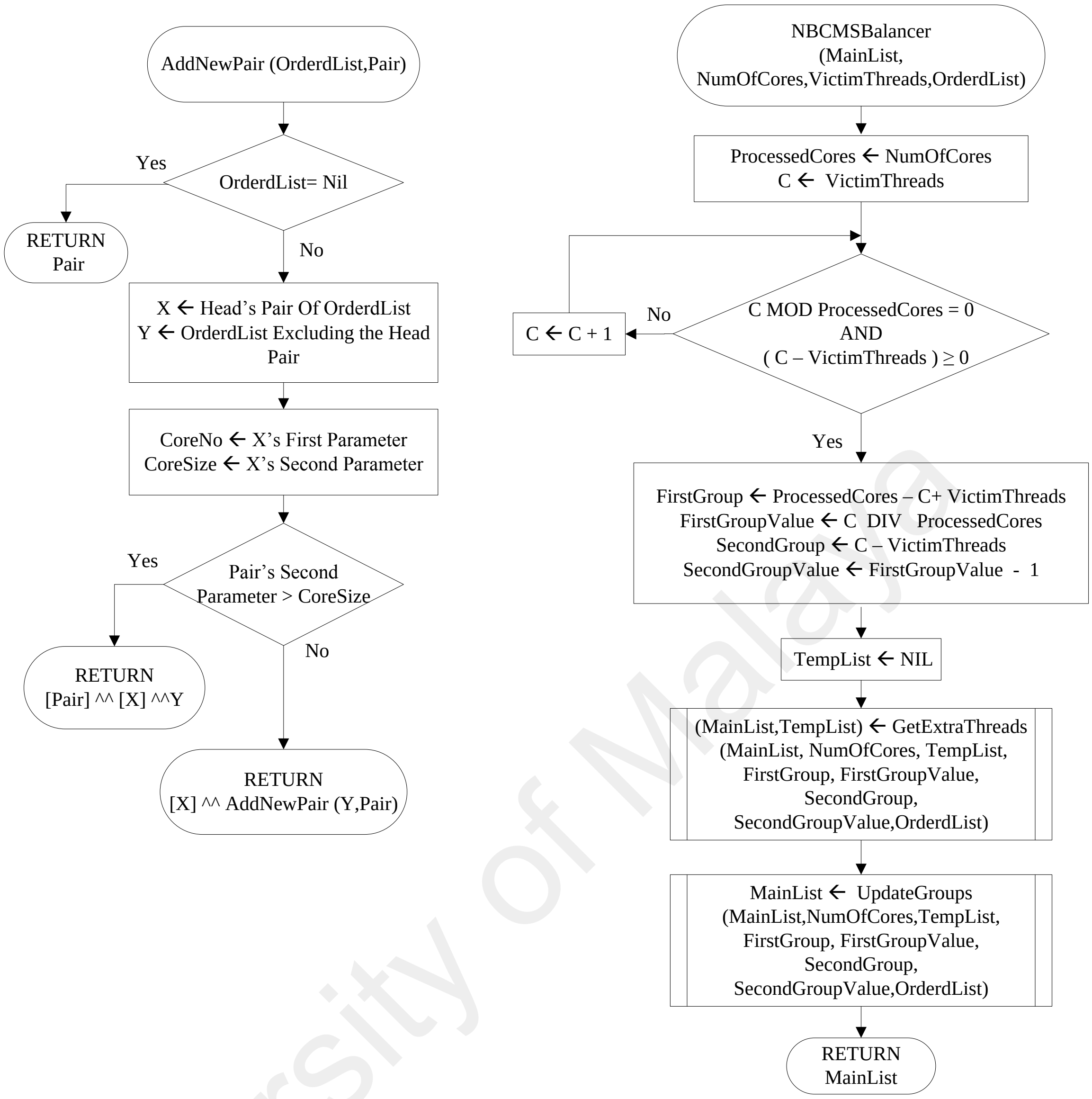

AddNewPair (OrderdList,Pair)
OrderdList= Nil
Yes
RETURN Pair
No
X ← Head's Pair Of OrderdList
Y ← OrderdList Excluding the Head Pair
CoreNo ← X's First Parameter
CoreSize ← X's Second Parameter
Pair's Second Parameter > CoreSize
Yes
RETURN [Pair] ^^ [X] ^^Y
No
RETURN [X] ^^ AddNewPair (Y,Pair)
NBCMSBalancer (MainList, NumOfCores,VictimThreads,OrderdList)
ProcessedCores ← NumOfCores
C ← VictimThreads
C MOD ProcessedCores = 0
AND
(C – VictimThreads) ≥ 0
No
C ← C + 1
Yes
FirstGroup ← ProcessedCores – C+ VictimThreads
FirstGroupValue ← C DIV ProcessedCores
SecondGroup ← C – VictimThreads
SecondGroupValue ← FirstGroupValue - 1
TempList ← NIL
(MainList,TempList) ← GetExtraThreads (MainList, NumOfCores, TempList, FirstGroup, FirstGroupValue, SecondGroup, SecondGroupValue,OrderdList)
MainList ← UpdateGroups (MainList,NumOfCores,TempList, FirstGroup, FirstGroupValue, SecondGroup, SecondGroupValue,OrderdList)
RETURN MainList

Figure II.2: The AddNewPair and NBCMSBalancer Functions

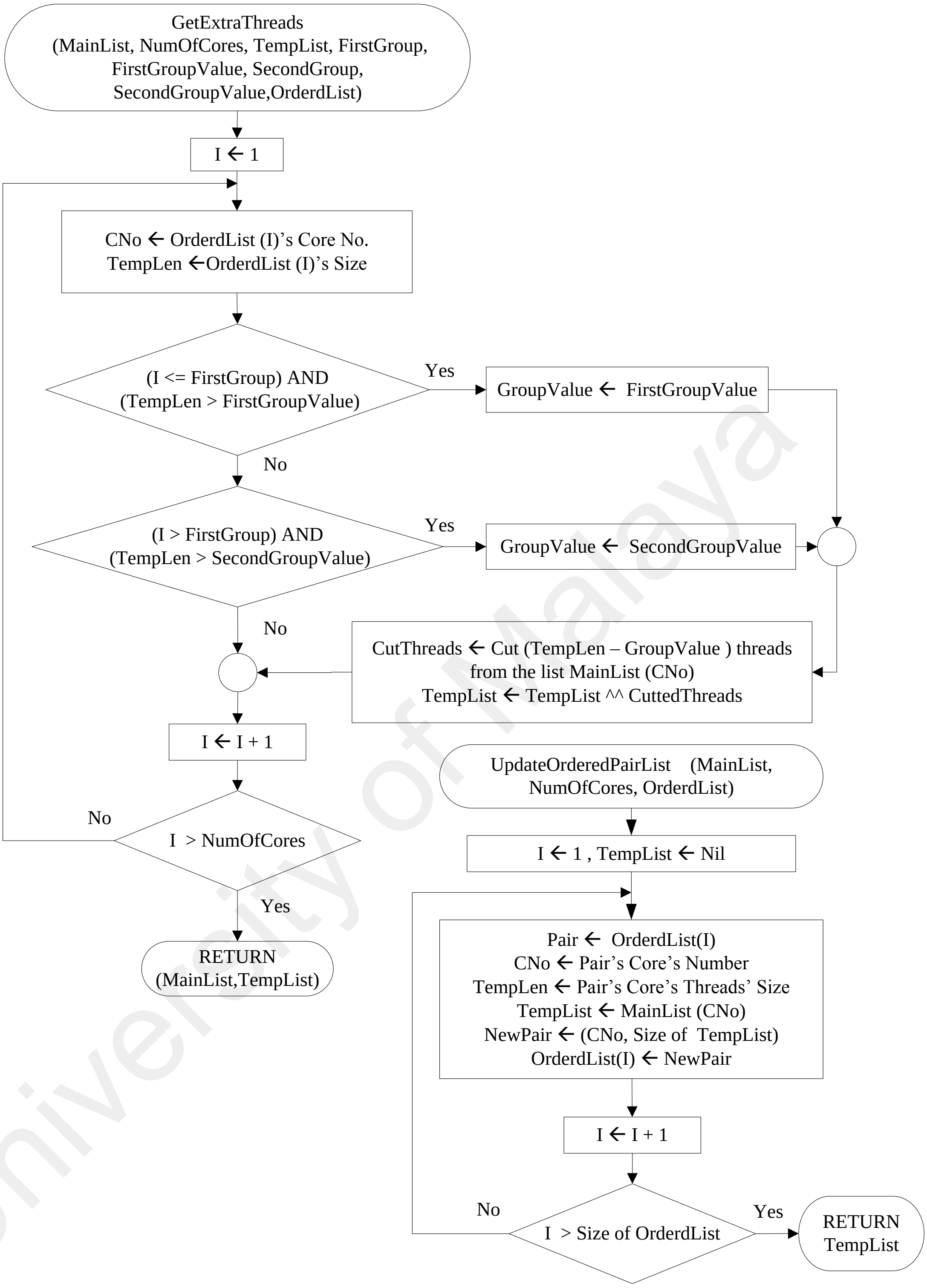

GetExtraThreads
(MainList, NumOfCores, TempList, FirstGroup,
FirstGroupValue, SecondGroup,
SecondGroupValue,OrderdList)
I ← 1
CNo ← OrderdList (I)'s Core No.
TempLen ←OrderdList (I)'s Size
(I <= FirstGroup) AND
(TempLen > FirstGroupValue)
Yes
GroupValue ← FirstGroupValue
No
(I > FirstGroup) AND
(TempLen > SecondGroupValue)
Yes
GroupValue ← SecondGroupValue
No
CutThreads ← Cut (TempLen – GroupValue) threads
from the list MainList (CNo)
TempList ← TempList ^^ CuttedThreads
I ← I + 1
No
I > NumOfCores
Yes
RETURN
(MainList,TempList)
UpdateOrderedPairList (MainList,
NumOfCores, OrderdList)
I ← 1 , TempList ← Nil
Pair ← OrderdList(I)
CNo ← Pair's Core's Number
TempLen ← Pair's Core's Threads' Size
TempList ← MainList (CNo)
NewPair ← (CNo, Size of TempList)
OrderdList(I) ← NewPair
I ← I + 1
No
I > Size of OrderdList
Yes
RETURN
TempList

Figure II.3: The GetExtraThreads and UpdateOrderedPairList functions

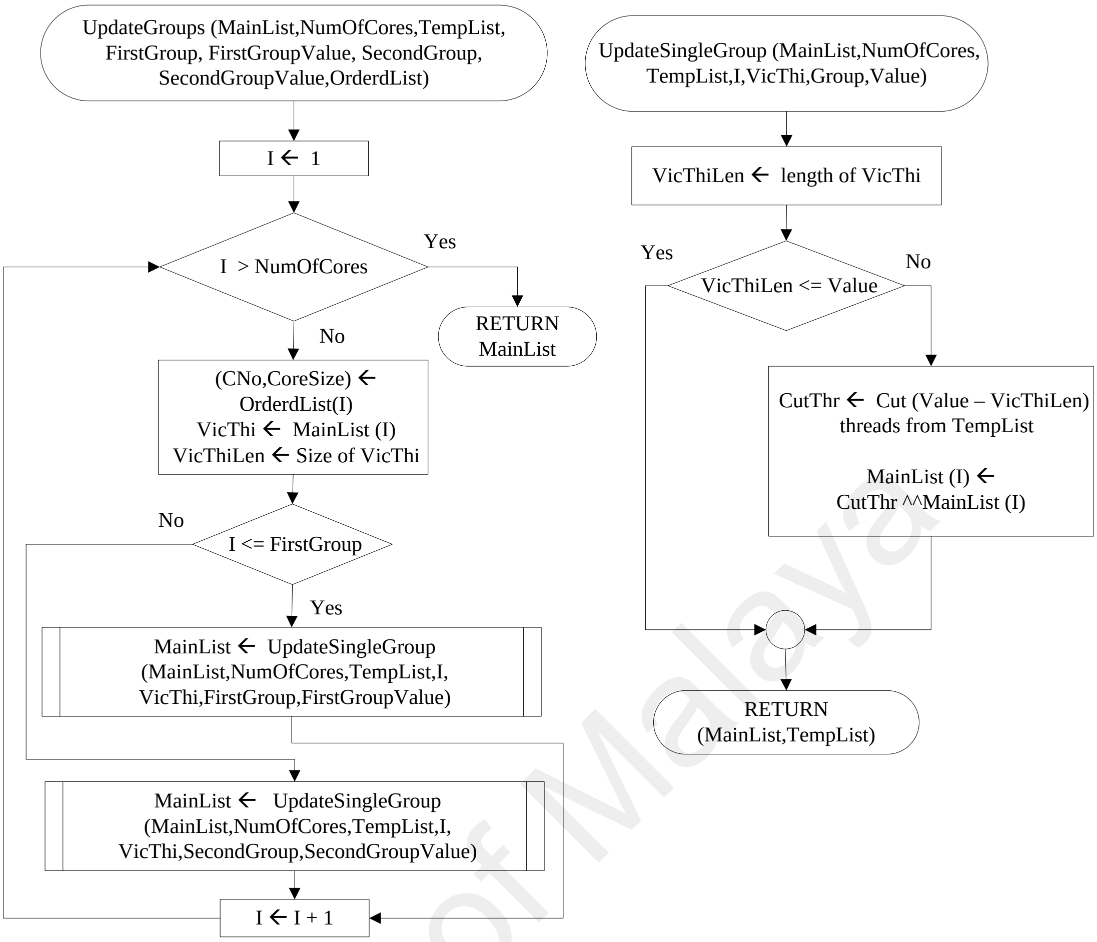

UpdateGroups (MainList,NumOfCores,TempList, FirstGroup, FirstGroupValue, SecondGroup, SecondGroupValue,OrderdList)
I ← 1
I > NumOfCores
Yes
RETURN MainList
No
(CNo,CoreSize) ← OrderdList(I)
VicThi ← MainList (I)
VicThiLen ← Size of VicThi
No
I <= FirstGroup
Yes
MainList ← UpdateSingleGroup (MainList,NumOfCores,TempList,I, VicThi,FirstGroup,FirstGroupValue)
MainList ← UpdateSingleGroup (MainList,NumOfCores,TempList,I, VicThi,SecondGroup,SecondGroupValue)
I ← I + 1
UpdateSingleGroup (MainList,NumOfCores, TempList,I,VicThi,Group,Value)
VicThiLen ← length of VicThi
Yes
No
VicThiLen <= Value
CutThr ← Cut (Value – VicThiLen) threads from TempList
MainList (I) ← CutThr ^^MainList (I)
RETURN (MainList,TempList)

Figure II.4: The UpdateGroups and UpdateSingleGroup functions

(a) The mechanism of NonBiasedPMSS

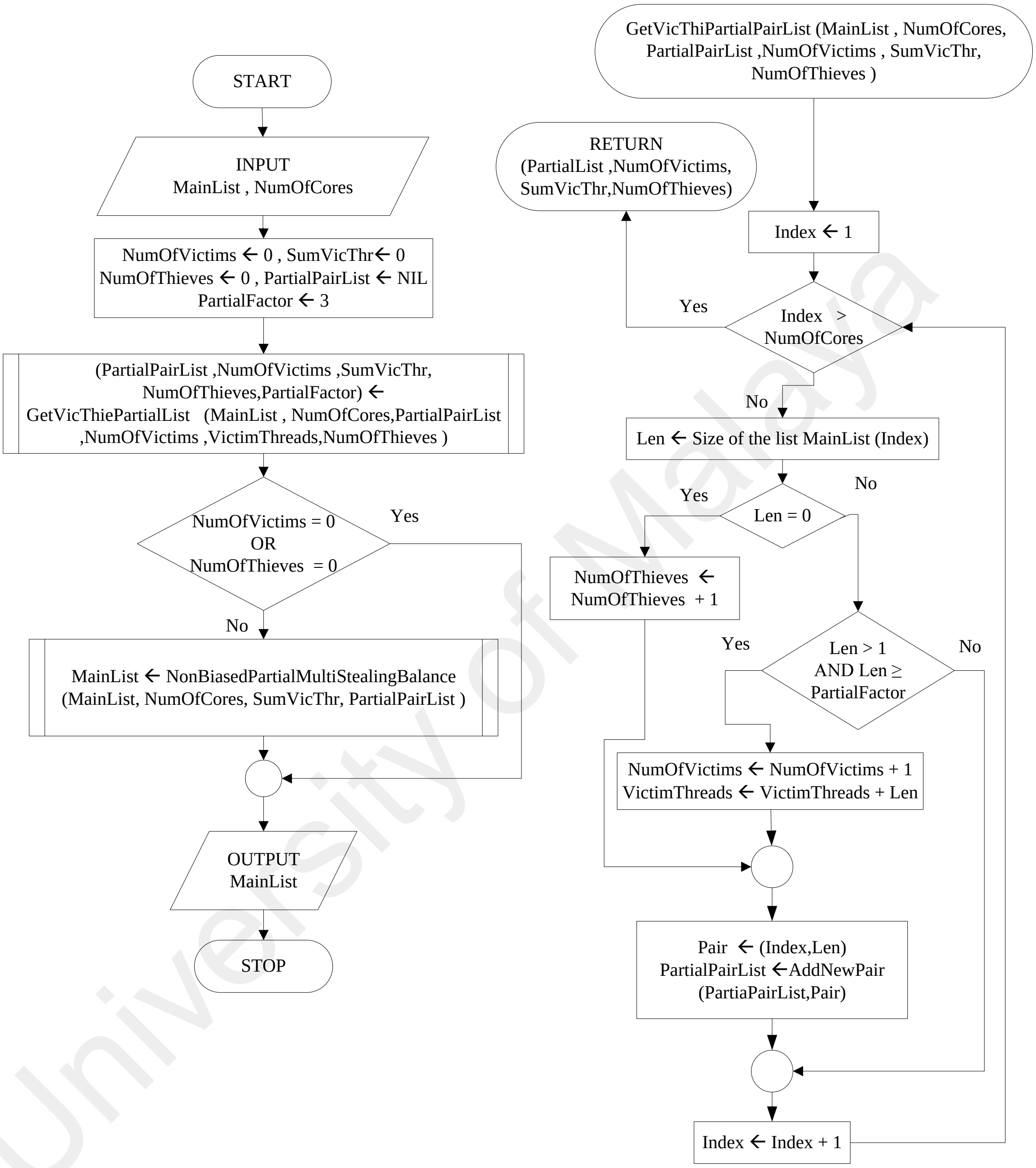

START
INPUT
MainList , NumOfCores
NumOfVictims ← 0 , SumVicThr← 0
NumOfThieves ← 0 , PartialPairList ← NIL
PartialFactor ← 3
(PartialPairList ,NumOfVictims ,SumVicThr, NumOfThieves,PartialFactor) ← GetVicThiePartialList (MainList , NumOfCores,PartialPairList ,NumOfVictims ,VictimThreads,NumOfThieves)
NumOfVictims = 0 OR NumOfThieves = 0
Yes
No
MainList ← NonBiasedPartialMultiStealingBalance (MainList, NumOfCores, SumVicThr, PartialPairList)
OUTPUT
MainList
STOP
GetVicThiPartialPairList (MainList , NumOfCores, PartialPairList ,NumOfVictims , SumVicThr, NumOfThieves)
RETURN
(PartialList ,NumOfVictims, SumVicThr,NumOfThieves)
Index ← 1
Index > NumOfCores
Yes
No
Len ← Size of the list MainList (Index)
Len = 0
Yes
No
NumOfThieves ← NumOfThieves + 1
Len > 1 AND Len ≥ PartialFactor
Yes
No
NumOfVictims ← NumOfVictims + 1
VictimThreads ← VictimThreads + Len
Pair ← (Index,Len)
PartialPairList ←AddNewPair (PartiaPairList,Pair)
Index ← Index + 1

Figure II.5: The NonBiasedPMSS and GetVicThiePartialList

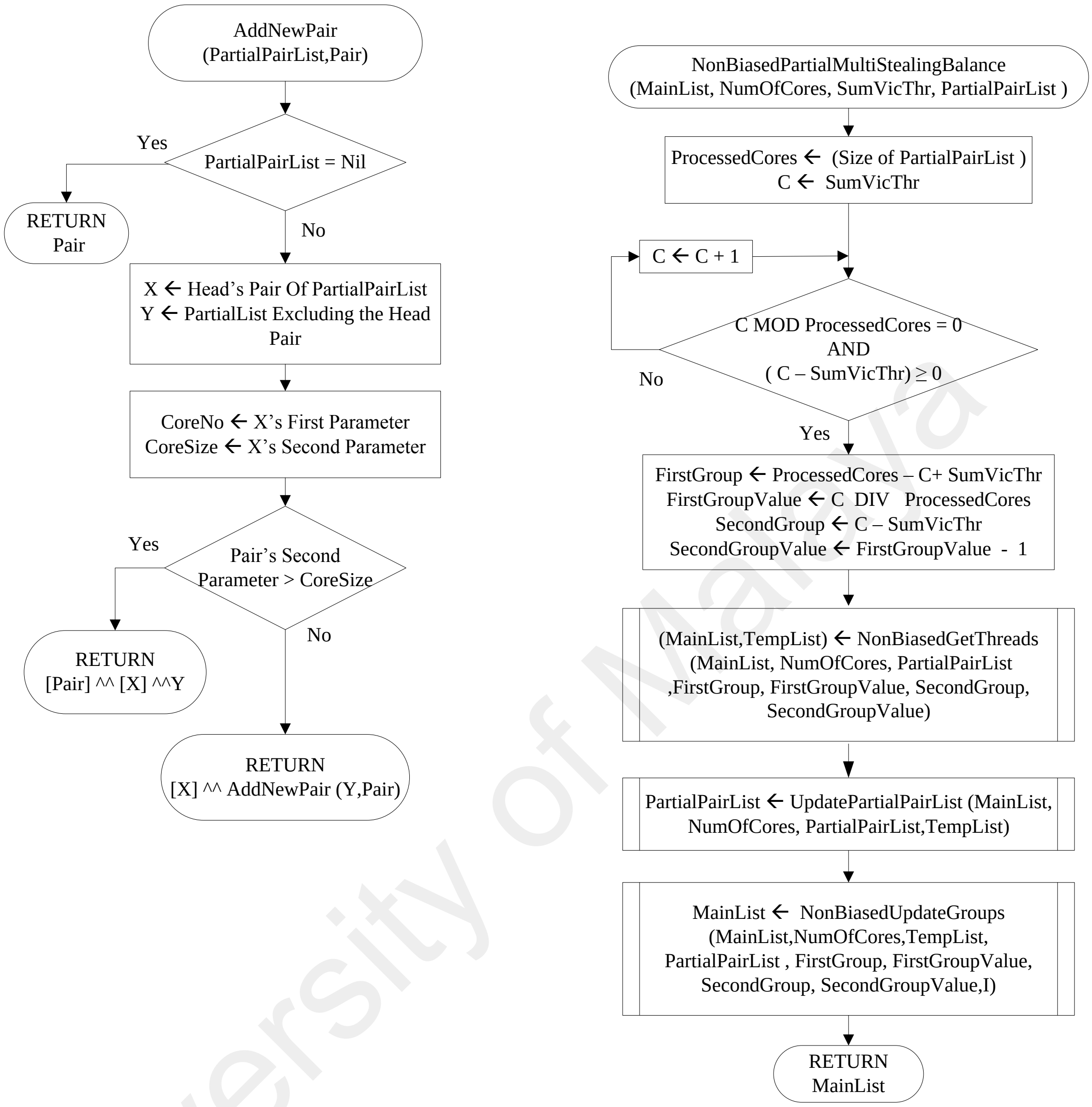
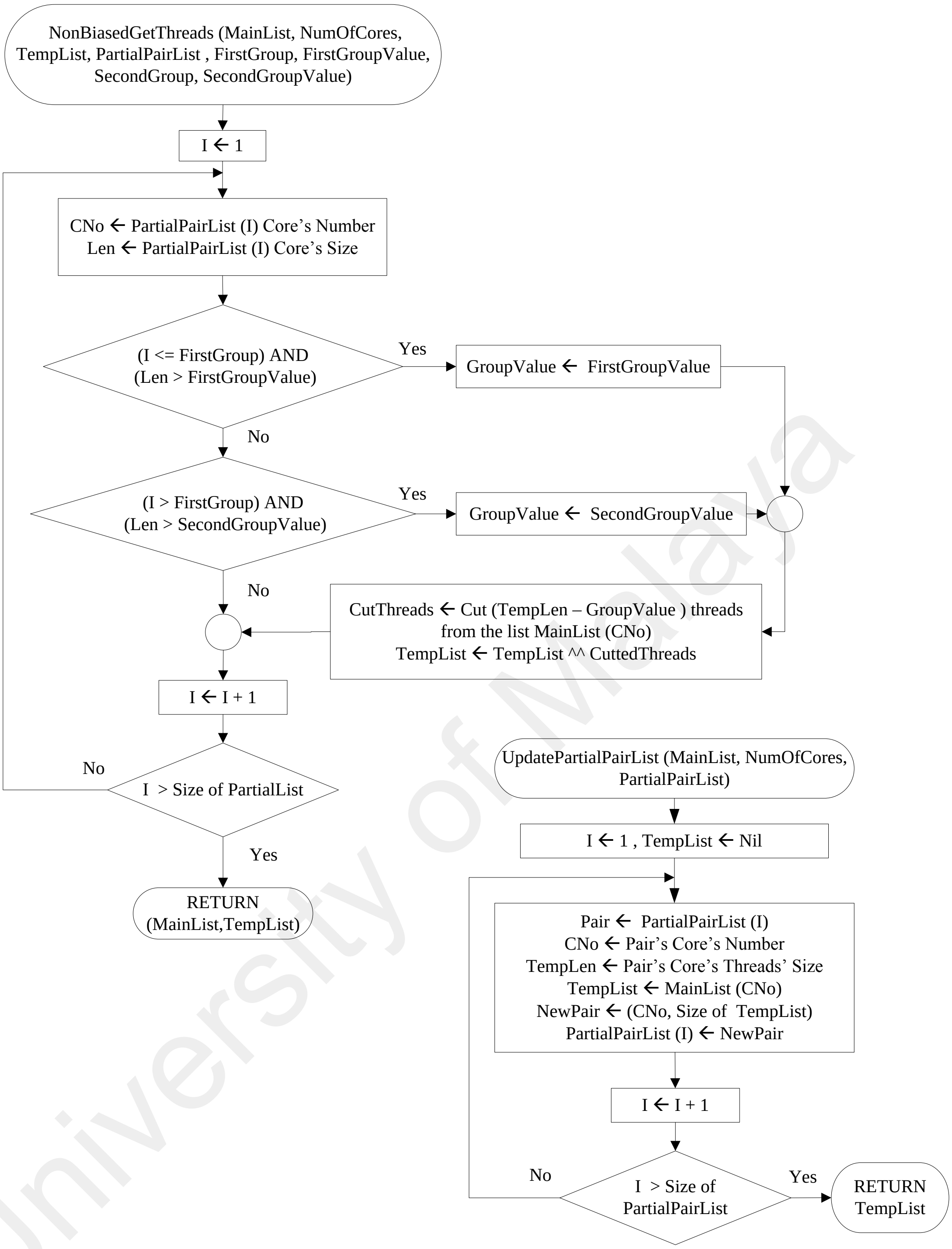

AddNewPair
(PartialPairList,Pair)
Yes
PartialPairList = Nil
RETURN
Pair
No
X ← Head's Pair Of PartialPairList
Y ← PartialList Excluding the Head Pair
CoreNo ← X's First Parameter
CoreSize ← X's Second Parameter
Yes
Pair's Second Parameter > CoreSize
RETURN
[Pair] ^^ [X] ^^Y
No
RETURN
[X] ^^ AddNewPair (Y,Pair)
NonBiasedPartialMultiStealingBalance
(MainList, NumOfCores, SumVicThr, PartialPairList)
ProcessedCores ← (Size of PartialPairList)
C ← SumVicThr
C ← C + 1
C MOD ProcessedCores = 0
AND
(C – SumVicThr) ≥ 0
No
Yes
FirstGroup ← ProcessedCores – C+ SumVicThr
FirstGroupValue ← C DIV ProcessedCores
SecondGroup ← C – SumVicThr
SecondGroupValue ← FirstGroupValue - 1
(MainList,TempList) ← NonBiasedGetThreads (MainList, NumOfCores, PartialPairList ,FirstGroup, FirstGroupValue, SecondGroup, SecondGroupValue)
PartialPairList ← UpdatePartialPairList (MainList, NumOfCores, PartialPairList,TempList)
MainList ← NonBiasedUpdateGroups (MainList,NumOfCores,TempList, PartialPairList , FirstGroup, FirstGroupValue, SecondGroup, SecondGroupValue,I)
RETURN
MainList

Figure II.6: The AddNewPair and NonBiasedPartialMultiStealingBalance



NonBiasedGetThreads (MainList, NumOfCores, TempList, PartialPairList , FirstGroup, FirstGroupValue, SecondGroup, SecondGroupValue)
I ← 1
CNo ← PartialPairList (I) Core's Number
Len ← PartialPairList (I) Core's Size
(I <= FirstGroup) AND (Len > FirstGroupValue)
Yes
GroupValue ← FirstGroupValue
No
(I > FirstGroup) AND (Len > SecondGroupValue)
Yes
GroupValue ← SecondGroupValue
No
CutThreads ← Cut (TempLen – GroupValue) threads from the list MainList (CNo)
TempList ← TempList ^^ CuttedThreads
I ← I + 1
No
I > Size of PartialList
Yes
RETURN (MainList,TempList)
UpdatePartialPairList (MainList, NumOfCores, PartialPairList)
I ← 1 , TempList ← Nil
Pair ← PartialPairList (I)
CNo ← Pair's Core's Number
TempLen ← Pair's Core's Threads' Size
TempList ← MainList (CNo)
NewPair ← (CNo, Size of TempList)
PartialPairList (I) ← NewPair
I ← I + 1
No
I > Size of PartialPairList
Yes
RETURN TempList

Figure II.7: The NonBiasedGetThreads and UpdatePartialPairList functions

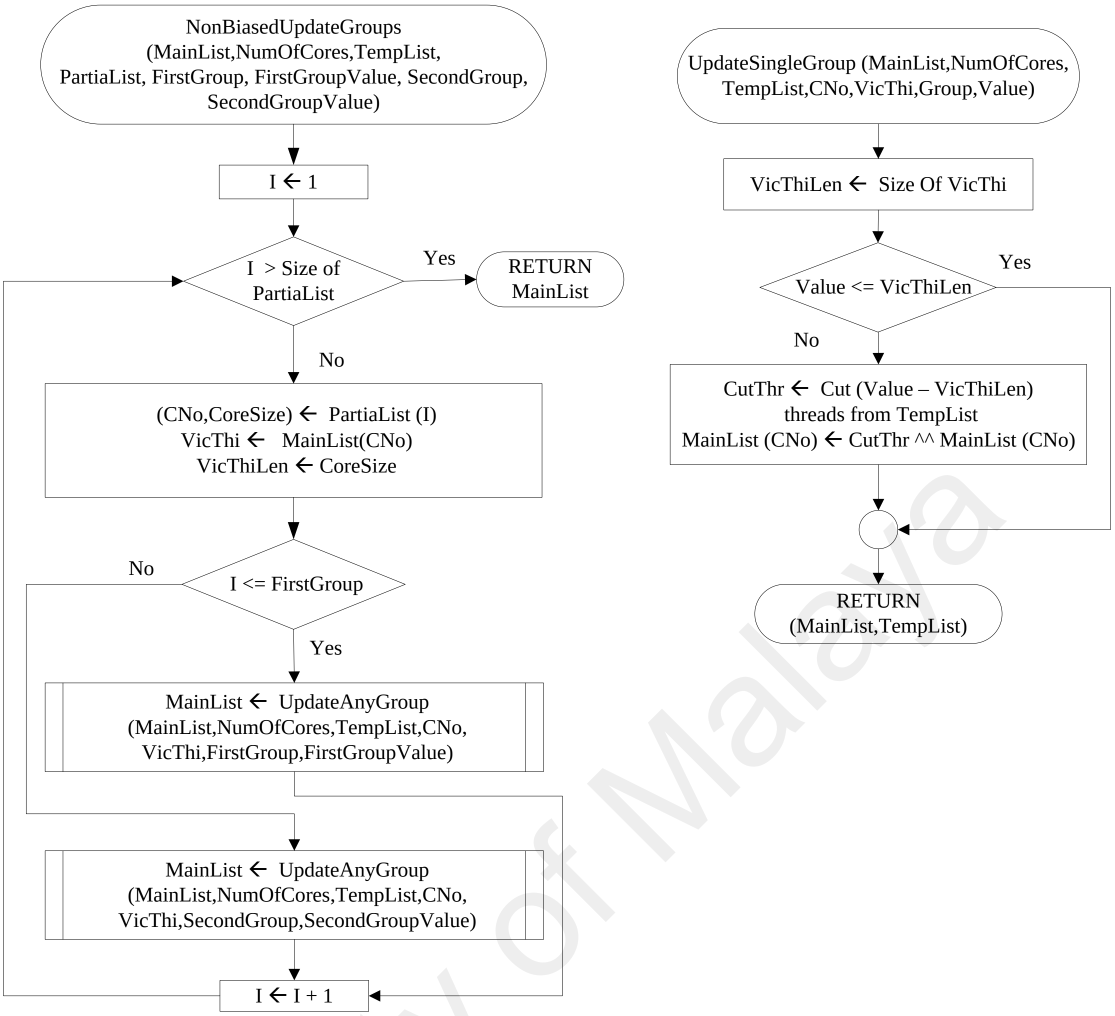

NonBiasedUpdateGroups (MainList,NumOfCores,TempList, PartiaList, FirstGroup, FirstGroupValue, SecondGroup, SecondGroupValue)
I ← 1
I > Size of PartiaList
Yes
RETURN MainList
No
(CNo,CoreSize) ← PartiaList (I)
VicThi ← MainList(CNo)
VicThiLen ← CoreSize
I <= FirstGroup
No
Yes
MainList ← UpdateAnyGroup (MainList,NumOfCores,TempList,CNo, VicThi,FirstGroup,FirstGroupValue)
MainList ← UpdateAnyGroup (MainList,NumOfCores,TempList,CNo, VicThi,SecondGroup,SecondGroupValue)
I ← I + 1
UpdateSingleGroup (MainList,NumOfCores, TempList,CNo,VicThi,Group,Value)
VicThiLen ← Size Of VicThi
Value <= VicThiLen
Yes
No
CutThr ← Cut (Value – VicThiLen) threads from TempList
MainList (CNo) ← CutThr ^^ MainList (CNo)
RETURN (MainList,TempList)

Figure II.8: The NonBiasedUpdateGroups and UpdateSingleGroup