A CONCURRENT MULTI-STEALING SCHEDULER MODEL FOR DIVIDE AND CONQUER PROBLEMS

Alaa M. Al-Obaidi [1] ***and Sai Peck Lee*** [2]
[1, 2] Department of Software Engineering, Faculty of Computer Science & Information Technology, University of Malaya, 50603 Kuala Lumpur, Malaysia
[1] alaa@siswa.um.edu.my
[2] saipeck@um.edu.my, Phone Number: +60379676361, Fax: +60379579249

ABSTRACT

Multicore architecture has dramatically changed the general direction of software development dedicated for personal computers. As such, it is important for software designers to keep pace with the evolving challenges that happen in the hardware side, for example in this context of multicore architecture, so that they can leverage on the advantages of multicore technology as much as possible while developing software. As one of the well-known techniques, Divide and Conquer has a natural adaptation with the multicore technology. The technique needs to be further developed to fit into this new environment. In this paper, we present a new concurrent multithreaded Colored Petri Nets model that provides a new approach for scheduling Divide and Conquer problems on a multicore environment. Two new schedulers have been developed to control the actions of the model. The Multi Stealing Scheduler (MSS) has been designed to redistribute threads among the modelled cores. The MSS is general, scalable and it can be used for any Divide and Conquer problem. The second scheduler is the Local Threads Scheduler (LTS) that has the duty of threads creation and division inside each modelled core. In addition, the LTS introduces a new recursive method to provide the necessary information to multiply two matrices. Two main things have been achieved: First, workload among the modelled cores becomes well balanced; second, the technique produces a high level of concurrency between the elements of the model, which greatly minimise the execution time.

Keywords: Concurrency, Multithreading, Colored Petri Nets, Divide and Conquer, Matrix Multiplication

1.0 INTRODUCTION

In computer science, Divide and Conquer (D&C) represents one of the vital techniques that can be used to solve a variety of problems [1, 2]. The solution to such kind of technique should be adapted to be more efficient in order to suit the multicore architecture. One of the main ideas is to provide a high-level concurrent multithreaded scheduler that can manage load distribution among the used cores. Matrix Multiplication 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, matrix inversion, etc. Therefore, to accelerate the execution of these problems, Matrix Multiplication should also be sped up to make use of these developments in the hardware side. It is apparent that there is a strong relation between D&C, multithreading, and concurrency. Any D&C problem can be divided into a finite number of threads that can be assigned to the available cores. The process of assigning besides the execution of these threads can be done concurrently, leading to highly execution rates.

The development of new concurrent methods that effectively utilise the dimensioning growth of the number of cores is the cornerstone for the optimum utilisation of the multicore technology. These new methods should fulfil several conditions such as: generality, scalability, high degree of concurrency, and fairness in load distribution. Modelling represents the first step towards building robust systems that are able to achieve these demands. Through modelling and modelling tools, many errors and weak points in the software can be identified and corrected. This study has two objectives: First, it aims to provide a general concurrent multithreaded scheduler that can balance load distribution in a centralised manner for any Divide and Conquer problem on a multicore architecture. The second objective aims to provide another scheduler that is specifically designed for multiplying two matrices. The two schedulers are working together in achieving one goal, that is, to design a new centralised, scalable, efficient, concurrent, multithreaded scheduler model for Divide and Conquer problems.

In this paper, we propose Multi Stealing Scheduler (MSS) to fairly organise threads distribution among n-modelled cores in a simulated multicore environment. The MSS is general enough to be applied for any D&C problem. The scheduler represents a centralised unit that balances cores' threads through stealing threads from the victim (non idle) cores and send them to the thief (idle) cores. The process of stealing allows multi threads to be stolen from one or more victim cores and redistributed to one or more thief cores. The main objective of this study is to make all the cores working concurrently as much as possible. As an extension to this paper, we also propose another scheduler, Local Threads Scheduler (LTS). The LTS is in charge of threads creation, division, and it provides the necessary information that assists in multiplying two-dimensional matrices in a multicore environment.

The rest of this paper is organised as follows: background for Work Stealing techniques, the modelling language, Colored Petri Net (CPN), and the modelling tool, CPN-Tool, are given in Section 2. In Section 3, scheduling Divide and Conquer problems on a multicore environment is explained in detail. Building the CPN model is fully described in Section 4. The results of the simulation process, discussion and the conclusion are included in the Sections 5, 6, and 7 respectively.

2.0 BACKGROUND

The Multi Stealing Scheduler is built on the basis of Work Stealing. The idea of Work Stealing came from the effort of Blumofe and Leiserson [3]. They designed an algorithm that is able to schedule fully-strict (well-structured) multithreaded computations. The results 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 that deal with multiprogramming. This is due to the algorithm's designed mechanism that deals with a fixed set of processors. Another major contribution represented in the work of Arrora et al [4] through designing a non fully-strict mechanism for multithreaded computation. Their work is considered an improvement to [3] in view of the fact that they succeeded in designing an algorithm that can deal with a multiprogrammed environment instead of a dedicated one. The achievement made by [4] was considered one of the best load balancing techniques in the academic as well as in the industrial fields. However, some problems have been encountered: First, the algorithm of Arrora et al faced the memory management problem. It can deal with only m/n threads inside its deques where m represents the total memory size and n represents the number of processes. The second problem is related with overflow that can easily occur due to the use of arrays in representing deques. The sizes of the arrays have to be adjusted many times. There is no easy way to free additional sizes of memory.

Several works have extended [4] in different ways. "Stealing The Half", a new idea introduced by [5]. As the name implies, a half number of the threads can be stolen in one trial. A locality-guided work stealing algorithm that improves the data locality of multithreaded computations by allowing a thread to have an affinity for a processor has been suggested by [6]. Hendler et al presented in [7] a new algorithm that can detect synchronisation conflicts by pointer-crossing instead of gaps between indexes as suggested by [4]. The algorithm does get rid of fixing the overflow problem [4] through adopting non-blocking dynamic-sized work stealing deques. Despite this, there was a kind of trade-off between space and time complexity since lists (main structure used to represent deques) have been used as an alternative to arrays. Chase and Lev [8] presented a simple lock-free work stealing deque. Their algorithm is based on using a cyclic array that can easily deal with overflows. Memory size is the only limitation to this algorithm; however, there is no need for garbage collection. Another contribution is represented by the work of [9]. Agrawal, et al presented an adaptive thread scheduler, called A-STEAL. Their scheduler performs better than [4] 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 [10]. Work Stealing has been formally proven to be optimal only for the restricted class of fully-strict computations. Recently, Ding et al presented in [11] a work-stealing scheduler for time-sharing multicore systems. Their scheduler has been designed to deal with two important drawbacks in the work of Arrora et al, significant unfairness and degraded throughput. The scheduler improves average system throughput and reduces average unfairness.

The common feature in the above studies is the decentralisation in managing threads distribution where the selection of the victim cores is done randomly. We have conducted four studies [12-15] regarding concurrent multithreading scheduling for D&C problems where the process of managing the threads is done in a centralised manner. That is, we suggested designing a separate hardware unit that is in charge of extracting threads from the victim cores and submit these threads to the thief cores. In the first two studies [12, 13], we suggested a simple scheduler that can manage threads distribution among the modelled cores. The scheduler concurrently steals threads from the victim cores and distributes them to the thief cores. However, in each stealing trial, only a single thread at a time is given to a thief core. In addition, we suggested specific inner schedulers (a scheduler per core) that can manage threads creation, division, and calculation inside each modelled core. The inner scheduler in [12] is directed to calculate Fibonacci series while the inner scheduler in [13] is dedicated for the Binary Search problem. Although the two models succeeded in reaching the final results, their load distribution was not efficient enough in view of the fact that only a single thread is given to the thief core at the same time. We often found that some cores have plenty of threads while other cores have only a single thread. The third study suggests searching for the richest core prior to any distribution process [14] while in the fourth study, we introduced the idea of partial stealing [15].

In this paper, we use Colored Petri Nets (CPN) as a graphical language for building and analyzing models of concurrent systems [16]. CPN has been developed from Petri Nets (PN) as being the origin of CPN [17, 18]. 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) [19, 20] in models. As a software tool, we use CPN-Tool [21] which is developed by Kurt Jensen [16]. 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) [21] as a language of writing declarations, expressions, and codes inside the model. CPN-ML has been built based on SML [19, 20].

3.0 SCHEDULING DIVIDE AND CONQUER PROBLEMS ON A MULTICORE ENVIRONMENT

In this paper, we present a new methodology for scheduling Divide and Conquer problems on a modelled multicore environment. The methodology of this study has been built on two schedulers: The first scheduler is the Multi Stealing Scheduler (MSS) that is designed to be general for any D&C problem. The scheduler represents a centralised unit that dynamically balances threads distribution among the modelled cores. The MSS is scalable; it can concurrently deal with *n* different cores. To get practical results, we introduce another scheduler, Local Threads Scheduler (LTS) that schedules threads creation, division, and managing to multiply two matrices. The LTS dynamically divides threads till reaching Leaf-Level threads (a Leaf-Level holds a row number of the first matrix and a column number of the second matrix). Along with LTS, each core has also three components: A Multiplier that represents the core's unit is responsible for multiplying rows by columns. In addition, each core has two Guards that control the activation of the LTS and the Multiplier. The two matrices, resulting matrix and Leaf-Level threads are kept inside a Shared Area. Moreover, the Shared Area holds a common value used by the Guards. Fig. 1 shows the relation between the elements of the model.

3.1. Multi Stealing Scheduler (MSS)

The MSS has been designed to be general for any class of D&C problems. The scheduler redistributes threads among the n modelled cores. The distribution process is centralised; it consists of stealing one or more threads from the victim cores, then delivering these stolen threads to the thief cores.

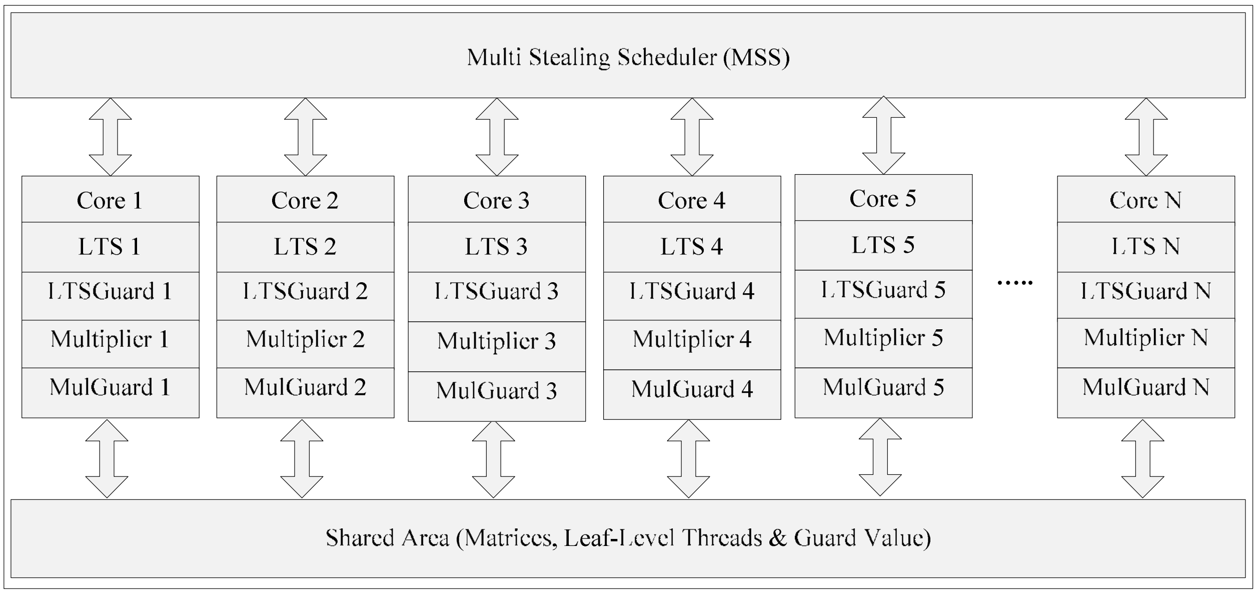


Fig. 1: The Elements of the Model

Accordingly, the statuses of idle cores can be minimized to the fullest extent. The modelled cores have been designed to deal with a different number of threads. At any instance, each core may have zero, one or more threads, besides, each individual modelled core has its own list which is used to retain its own threads. The cores' lists concurrently accept/give threads from/to the MSS.

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:
Let sum = $\sum_{i=1}^{n}(x_i)$
Let c = sum + k, where k is a constant ($k \geq 0$). 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 = $\sum_{i=1}^{8}(x_i) \Rightarrow$ sum = 27
Now, the smallest value of c that achieves (c mod n = 0) is 32. Since c = sum + k $\Rightarrow$ k = 32 – 27 $\Rightarrow$ k = 5
h = (32 / 8) $\Rightarrow$ h = 4
$\therefore$ 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

Although the new values (4, 4, 4, 3, 3, 3, 3, 3) seem balanced with unavoidable bias to some elements, however, there are many cases where these calculations are not only bias but also totally unfair. Let us consider the example where the original values of the variables are: (1, 0, 0, 0, 0, 0, 2, 3). Now, if we recalculate the values of n, sum, c, k and h, we will get the following: n = 8, sum = 6, c = 8, k = 2, h = 1. Therefore, the new values will be: (1, 1, 1, 1, 1, 1, 0, 0). It is obvious that the last two variables have been treated unfairly. Applying the same policy in threads redistribution among n modelled cores will generate an unbalanced situation. It is clear that some cores will sacrifice all their threads in return of feeding other cores. Therefore, this technique is totally rejected in our case.

The MSS that we suggest in this study treats the lists of threads fairly regardless of the original distribution of threads in the modelled cores. Regarding the previous example, our MSS produces the following values: (1, 1, 1, 1, 0, 0, 1, 1). The mechanism of the MSS is depicted in Fig. 2. We can distinguish four different parts in this scheduler:

Part I: Calculating the *Total* Number of Threads (First Loop). At the beginning, the scheduler needs to know the entire number of threads in the modelled cores. This resembles using Σ in the previous example.

Part II: Calculating the *Base* Value. In the MSS, the *Base* value represents the upper limit of threads for each core. The value is obtained by dividing the total number of threads by the number of cores. No core is allowed to have more than *Base* threads. In case we have too few number of threads compared with the number of cores, then the value of *Base* is adjusted to one.

Part III: Moving the Extra Threads. The second loop in the MSS is dedicated for extracting the extra threads from the cores that own more than *Base* threads. Thus, the multi stealing happens in this part. The *Temp* list will eventually hold a collection of extra threads gathered from wealthy (victim) cores.

Part IV: This part is in charge of balancing the number of threads in the entire cores. First, we calculate the value of the variable *Add*. This variable represents the extra number of threads that has to be distributed when all the cores have the same share (*Base*) of threads. Naturally, the value of *Add* ranges from zero to N-1. Then we enter a loop that has two functions:

A- Adjusting the number of threads in each core. If the number of threads is less than the *Base* value then the scheduler cuts (if available) a number of threads from the *Temp* list to fulfil the core's need.

B- Managing the extra threads. The scheduler checks whether the same core would get an extra thread or not. This can be done through adjusting the value of the variable *Add*. If the current value of *Add* is greater than zero, then a single thread is extracted from the *Temp* list and sent to the core; otherwise no extra thread is added.

3.2. Local Threads Scheduler (LTS)

We have *n* LTSs that work concurrently at the same time (Fig. 1). The LTS creates and manages two types of threads; Normal and Leaf-Level Threads.

3.2.1. Normal Threads

The LTS creates a tree of threads (Fig. 3). We have modelled the normal threads (symbolised by rectangles) as a 7-tuple: (*ThreadId, FatherId, StartRow, EndRow, N, StartColumn, EndColumn)*. The first two parameters hold the thread's number and the thread's father number. *ThreadId* and *FatherId* are denoted as "Tx" (x is a positive integer number).

ThreadId has the value of its ancestor's *ThreadId* multiplied by two, while *FatherId* has the value of its ancestor's *ThreadId*. As for the other parameters, a normal thread holds the necessary information to multiply two matrices. 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 initial and the final 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 first and last column numbers in the second matrix (B). Normal threads are created inside the modelled cores and they can be reallocated under the supervision of the MSS.

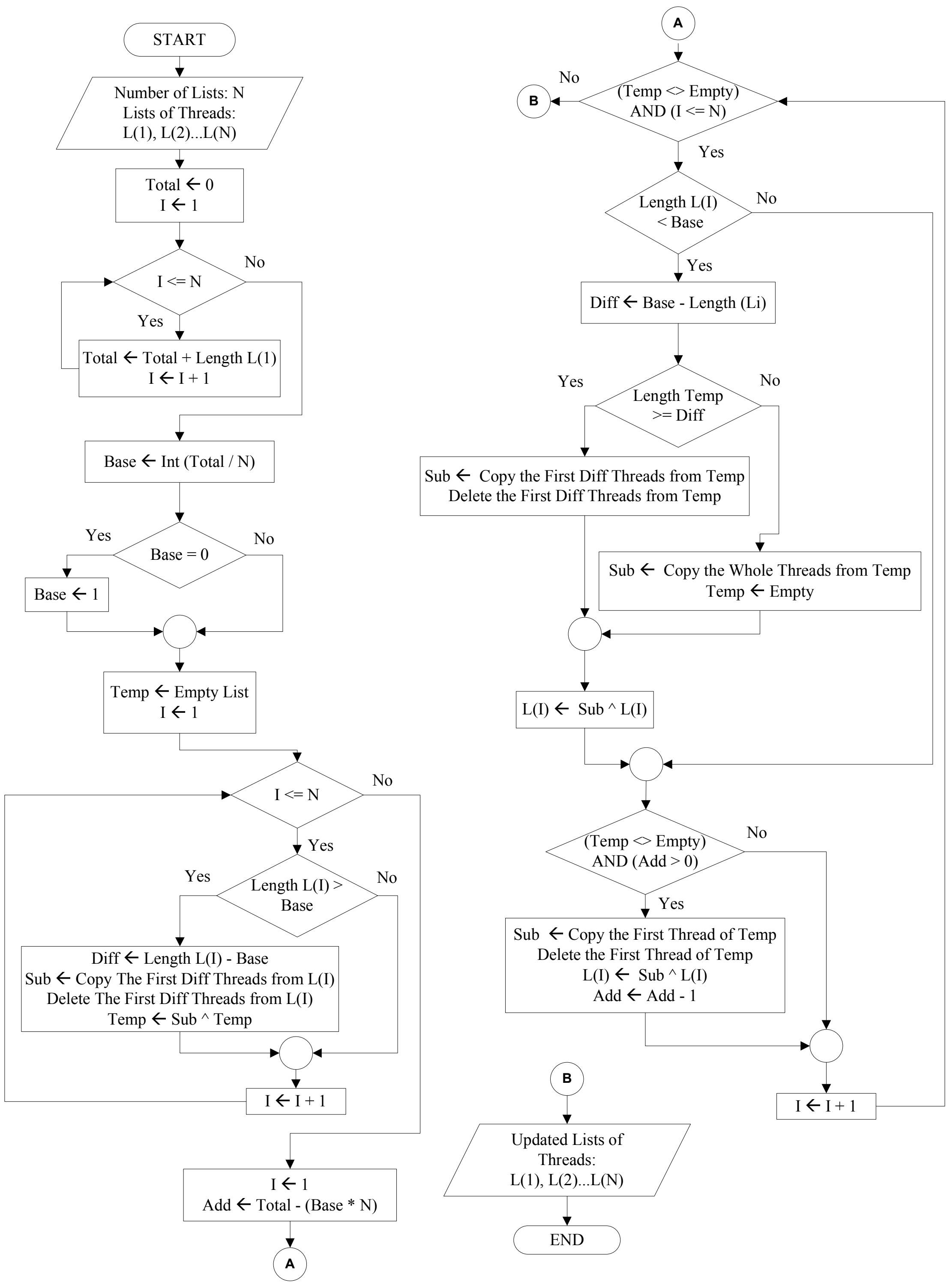

START
Number of Lists: N
Lists of Threads:
L(1), L(2)...L(N)
Total ← 0
I ← 1
I <= N
No
Yes
Total ← Total + Length L(1)
I ← I + 1
Base ← Int (Total / N)
Base = 0
Yes
No
Base ← 1
Temp ← Empty List
I ← 1
I <= N
No
Yes
Length L(I) > Base
Yes
No
Diff ← Length L(I) - Base
Sub ← Copy The First Diff Threads from L(I)
Delete The First Diff Threads from L(I)
Temp ← Sub ^ Temp
I ← I + 1
I ← 1
Add ← Total - (Base * N)
A
A
(Temp <> Empty) AND (I <= N)
No
B
Yes
Length L(I) < Base
No
Yes
Diff ← Base - Length (Li)
Length Temp >= Diff
Yes
No
Sub ← Copy the First Diff Threads from Temp
Delete the First Diff Threads from Temp
Sub ← Copy the Whole Threads from Temp
Temp ← Empty
L(I) ← Sub ^ L(I)
(Temp <> Empty) AND (Add > 0)
No
Yes
Sub ← Copy the First Thread of Temp
Delete the First Thread of Temp
L(I) ← Sub ^ L(I)
Add ← Add - 1
I ← I + 1
B
Updated Lists of Threads:
L(1), L(2)...L(N)
END

Fig. 2: Multi Stealing Scheduler (MSS). The inputs are lists of threads (L(1), L(2)…L(N)). Every list belongs to a core. The outputs are the same lists after reallocating the threads inside them. The symbol "^" is used to connect two lists.

3.2.2. Leaf-Level Threads

A Leaf-Level thread is a simple representation of the terminal normal thread and it (Leaf-Level thread) is used directly in finding the result of multiplying a single row by a single column. In multiplying a single row by a single column, we need only a row number, size of the elements in the row, and a column number. These three parameters are the elements of the Leaf-Level threads. This kind of threads is kept inside the Shared Area (Fig. 1). In Fig.3, an example for a Leaf-Level thread is (3,4,2). This thread is allocated for multiplying the elements of the third row (matrix A) by the elements of the second column (matrix B). Both the row and the column have four elements.

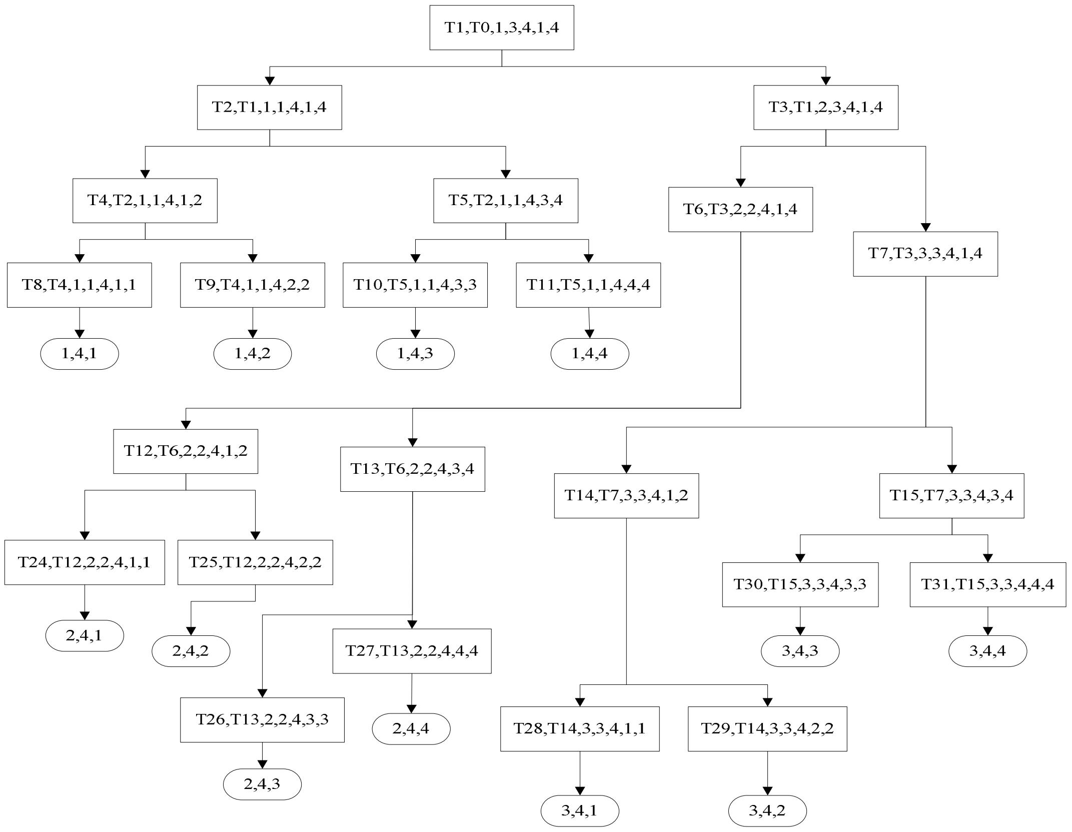


Fig. 3: An example of the division process of the Local Threads Schedulers intended for multiplying two matrices: $A_{3,4} \times B_{4,4} = C_{3,4}$. Normal threads are symbolised by rectangles while Leaf-Level threads are symbolised by ovals

3.2.3. LTS Mechanism

The essence of the LTS mechanism (Fig. 4) is in computing the values of the parameters: *StartRow, EndRow, StartColumn,* and *EndColumn*. The LTS is executed in each of the *n* modelling cores. The scheduler receives a list of normal threads and it (scheduler) continuously divides normal threads till generating Leaf-Level threads. To accelerate the process of division, any normal thread that has four normal grandchildren is directly divided into these four threads.

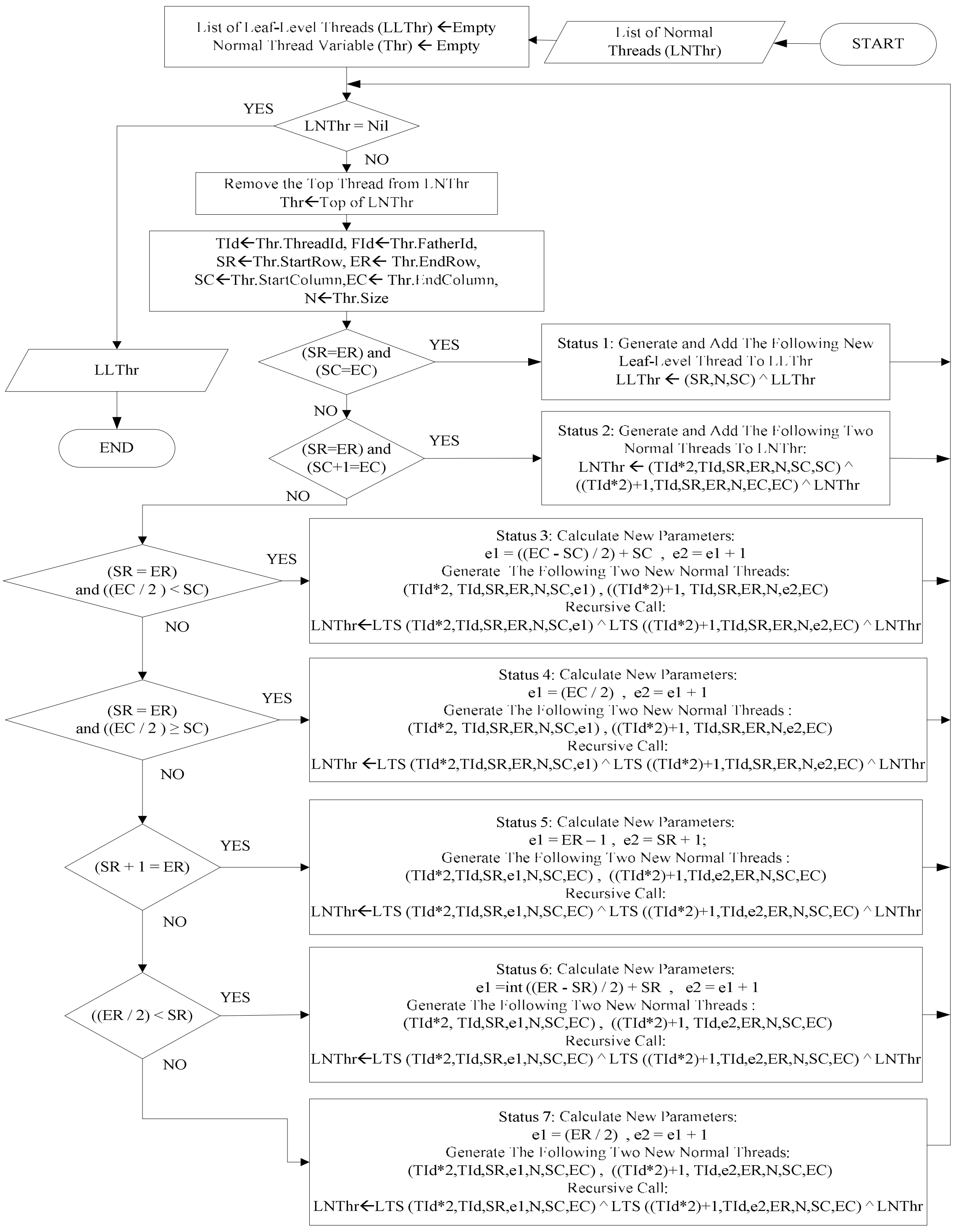

START
List of Normal Threads (LNThr)
List of Leaf-Level Threads (LLThr) ←Empty
Normal Thread Variable (Thr) ← Empty
LNThr = Nil
YES
NO
Remove the Top Thread from LNThr
Thr←Top of LNThr
TId←Thr.ThreadId, FId←Thr.FatherId, SR←Thr.StartRow, ER← Thr.EndRow, SC←Thr.StartColumn,EC← Thr.EndColumn, N←Thr.Size
LLThr
END
(SR=ER) and (SC=EC)
Status 1: Generate and Add The Following New Leaf-Level Thread To LLThr
LLThr ← (SR,N,SC) ^ LLThr
(SR=ER) and (SC+1=EC)
Status 2: Generate and Add The Following Two Normal Threads To LNThr:
LNThr ← (TId*2,TId,SR,ER,N,SC,SC) ^ ((TId*2)+1,TId,SR,ER,N,EC,EC) ^ LNThr
(SR = ER) and ((EC / 2) < SC)
Status 3: Calculate New Parameters:
e1 = ((EC - SC) / 2) + SC , e2 = e1 + 1
Generate The Following Two New Normal Threads:
(TId*2, TId,SR,ER,N,SC,e1) , ((TId*2)+1, TId,SR,ER,N,e2,EC)
Recursive Call:
LNThr←LTS (TId*2,TId,SR,ER,N,SC,e1) ^ LTS ((TId*2)+1,TId,SR,ER,N,e2,EC) ^ LNThr
(SR = ER) and ((EC / 2) ≥ SC)
Status 4: Calculate New Parameters:
e1 = (EC / 2) , e2 = e1 + 1
Generate The Following Two New Normal Threads :
(TId*2, TId,SR,ER,N,SC,e1) , ((TId*2)+1, TId,SR,ER,N,e2,EC)
Recursive Call:
LNThr ←LTS (TId*2,TId,SR,ER,N,SC,e1) ^ LTS ((TId*2)+1,TId,SR,ER,N,e2,EC) ^ LNThr
(SR + 1 = ER)
Status 5: Calculate New Parameters:
e1 = ER – 1 , e2 = SR + 1;
Generate The Following Two New Normal Threads :
(TId*2,TId,SR,e1,N,SC,EC) , ((TId*2)+1,TId,e2,ER,N,SC,EC)
Recursive Call:
LNThr←LTS (TId*2,TId,SR,e1,N,SC,EC) ^ LTS ((TId*2)+1,TId,e2,ER,N,SC,EC) ^ LNThr
((ER / 2) < SR)
Status 6: Calculate New Parameters:
e1 =int ((ER - SR) / 2) + SR , e2 = e1 + 1
Generate The Following Two New Normal Threads :
(TId*2, TId,SR,e1,N,SC,EC) , ((TId*2)+1, TId,e2,ER,N,SC,EC)
Recursive Call:
LNThr←LTS (TId*2,TId,SR,e1,N,SC,EC) ^ LTS ((TId*2)+1,TId,e2,ER,N,SC,EC) ^ LNThr
Status 7: Calculate New Parameters:
e1 = (ER / 2) , e2 = e1 + 1
Generate The Following Two New Normal Threads:
(TId*2,TId,SR,e1,N,SC,EC) , ((TId*2)+1, TId,e2,ER,N,SC,EC)
Recursive Call:
LNThr←LTS (TId*2,TId,SR,e1,N,SC,EC) ^ LTS ((TId*2)+1,TId,e2,ER,N,SC,EC) ^ LNThr

Fig. 4: Local Threads Scheduler (LTS)

The term "LTS" in Fig. 4 indicates calling the same algorithm with the adjusted parameters. That is, according to Fig. 3, when the scheduler processes the first thread (T1,T0,1,3,4,1,4), it will directly generates (T4,T2,1,1,4,1,2), (T5,T2,1,1,4,3,4), (T6,T3,2,2,4,1,4), and (T7,T3,3,3,4,1,4) without generating (T2,T1,1,1,4,1,4) and

(T3,T1,2,3,4,1,4). This process is repeated recursively. We can distinguish seven different statuses (Fig. 4) in calculating the above parameters.
In Status 1, when both the *StartRow* and *EndRow* are matched, and at the same time, *StartColumn* and *EndColumn* are matched also; a Leaf-Level thread can be directly generated and sent to the Shared Area (Fig. 1). Status 2 is reserved for normal threads that come one step before the terminal normal threads. The result is in generating two terminal normal (non Leaf-Level) threads sent back to their list. Statuses 3 and 4 are activated when we have a matching in the rows parameters with a contrast in the column parameters. We have divided the threads into two sets depending on the result of dividing *EndColumn* by 2.

All the mentioned statuses are dedicated for managing threads that have a matching in the rows parameters while statuses 5, 6, and 7 are customized for threads that have mismatching in the rows parameters. When *EndRow* exceeds *StartRow* in one value, Status 5 is called. Two normal threads are sending to the list; each one has equal values of *StartRow* and *EndRow*. Statuses 6 and 7 manage the threads that have more than one value between *StartRow* and *EndRow*. In both these two statuses, the LTS mechanism tries to narrow down the differences between the rows parameters. The result of dividing *EndRow* by 2 has been taken as the point of threads' division. Status 6 deals with threads that have less difference between *StartRow* and *EndRow,* while Status 7 is dedicated for those threads that have much difference between these parameters. The LTS output is a set of Leaf-Level threads. The entire cores keep the Leaf-Level threads in the Shared Area (Fig. 1).

3.3. The Multipliers

The Multiplier has the function of extracting the row number, number of elements in the row, and the column number from the Leaf-Level Threads. Then, it uses this information in reading the elements of the desired row (first matrix) and the desired column (second element). Finally, it multiplies the elements of the row by the elements of the column and save the resulting value in the resulting matrix. All the Multipliers operate concurrently in computing the elements of the new matrix.

3.4. The Guards

In Fig. 1, each core has two function units: LTS and Multiplier. To enable/disable the activation of these two units; Guards have been added. The reason behind using Guards is to enforce the MSS to redistribute threads when there is an urgent need. For instance, one or more cores become idle. At the same time, one or more cores are wealthy in threads. The Guards, temporarily, disable the activation of the LTSs and Multipliers so that they can enforce the MSS to redistribute the threads among the modelled cores in a fair manner. Two Guards have been assigned for each core; one for the LTS (LTSGuard) and the other for the Multiplier (MulGuard). Both Guards have the same mechanism. The only exception is that the LTSGuard deals with the list of normal threads while the MulGuard deals with the list of Leaf-Level threads.

The general mechanism of a Guard is illustrated in Fig. 5. Every Guard (LTSGuard / MulGuard) reads the same shared value called the GuardsValue. This value consists of n-tuples (L1, L2…Ln). The Lx represents Normal threads list size of core x. For instance, a GuardsValue with (1,0,2,3) means that core 1 has one Normal thread, core 2 has zero Normal threads, etc.

Every Guard uses the GuardsValue, the associated core number and its list of threads (Normal or Leaf-Level depending on whether it is LTSGuard or MulGuard) as shown in Fig. 5 to decide whether it should activate (Guard is Locked) or deactivate (Guard is Opened) its LTS / Multiplier. We can distinguish four different statuses in the Guard behavior:

First Status is dedicated for an empty list of threads. When the associated list (Normal or Leaf-Level) is empty, then its associated Guard (LTSGuard / MulGuard) immediately disables (Locked) the LTS / Multiplier. The variable "EmptyList" retains the value true.

Second Status, the Guard opens the way for the LTS / Multiplier as long as all the cores are wealthy in threads (AllAreBusy remains true); therefore, the Guard enables LTS / Multiplier.

Third Status is aroused when some cores have only a single thread; in addition, one or more (not all) cores have zero threads (AllLessThan2 remains true). In this case, the Guard permits the LTS / Multiplier for working (as long as its associated list is not empty) since there is no chance to steal threads from other cores.

Finally, Fourth Status treats what is remained. This includes the occurrences where some cores are wealthy (more than one thread) and others are idle (zero threads); as a result, a Guard should be locked.

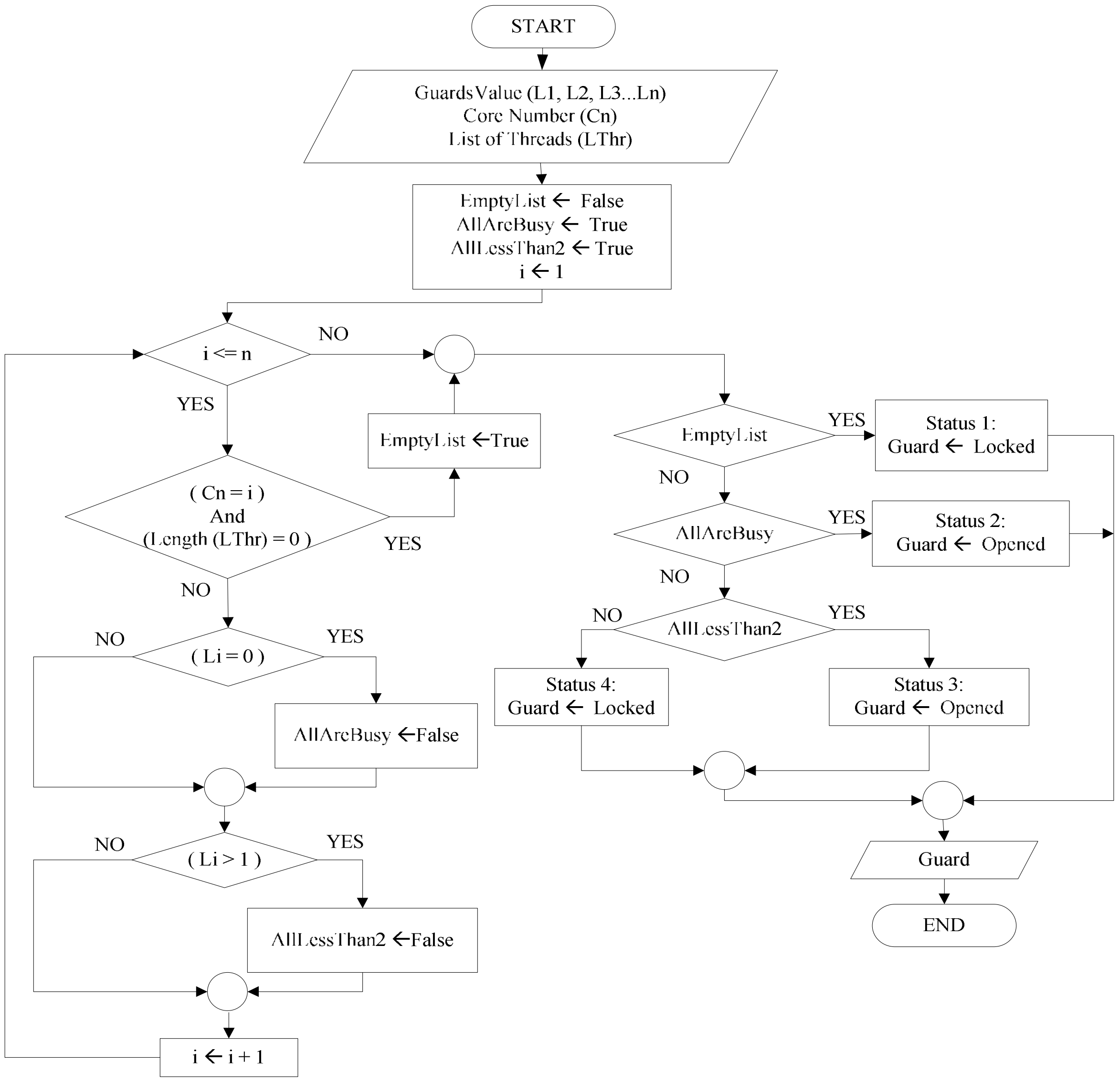


Fig. 5: Guards' Mechanism

4.0 A CPN MODEL FOR MULTICORE MULTI-STEALING SCHEDULER

We design a Colored Petri Nets (CPN) model that simulates the behaviour of the two schedulers, Multipliers and the Guards (Fig. 1). The model is hierarchical and scalable; it consists of a main page and three sub pages. The main page emulates the behavior of the Multi Stealing Scheduler (Fig. 2) while each one of the sub pages emulates a single core. Therefore, we have a CPN model with three modelled cores.

4.1. The Main Page

The main page (Fig. 6) of this model consists of four places (List 1, List 2, List 3, and GuardsValue), one transition (MSS), and three substituted transitions (Core1, Core2, and Core3).

4.1.1. Main Page Places

In CPN, a place is an oval shape that holds the data. In our model, List 1, List 2, and List 3 are of type *NormalList* (located at the lower right corner of every place). The type *NormalList* represents a list of the seven tuples mentioned in Section 3.2.1. Thus, the content of each place is a list of normal threads. In CPN, every place has an initial value usually located at the right corner of the place. Initially, List 2 and List 3 are empty (empty list is represented as [ ]), while List 1 has only one thread (the main thread).

At the top of each place, there is a circle attached to a rectangle; both are used to reflect the current content of each place. The circle carries the number of tokens (data) and the rectangle carries the data itself. In the case of List 1; there is only one list while there is an empty list in both List 2 and List 3. The content of List 1 is a single normal main thread: ("T1","T0",1,4,5,1,6). The main thread holds the information needed to multiply two matrices.

The dimensions of the first matrix are (4×5) and of the second matrix are (5×6). The parameter “T1” stands for the thread’s number while the parameter “T0” represents the thread’s predecessor number (in case of the main thread, there is no actual predecessor; even that, we keep its Id). The third and fourth parameters (1 and 4) stand for the numbers of the corresponding first and last row of the first matrix. The fifth parameter (number 5) corresponds to the number of columns and rows of the first and second matrices respectively. The last two parameters (1 and 6) stand for the first and last column numbers of the second matrix. The current content of each place changes during the simulation process while the initial contents never changed.

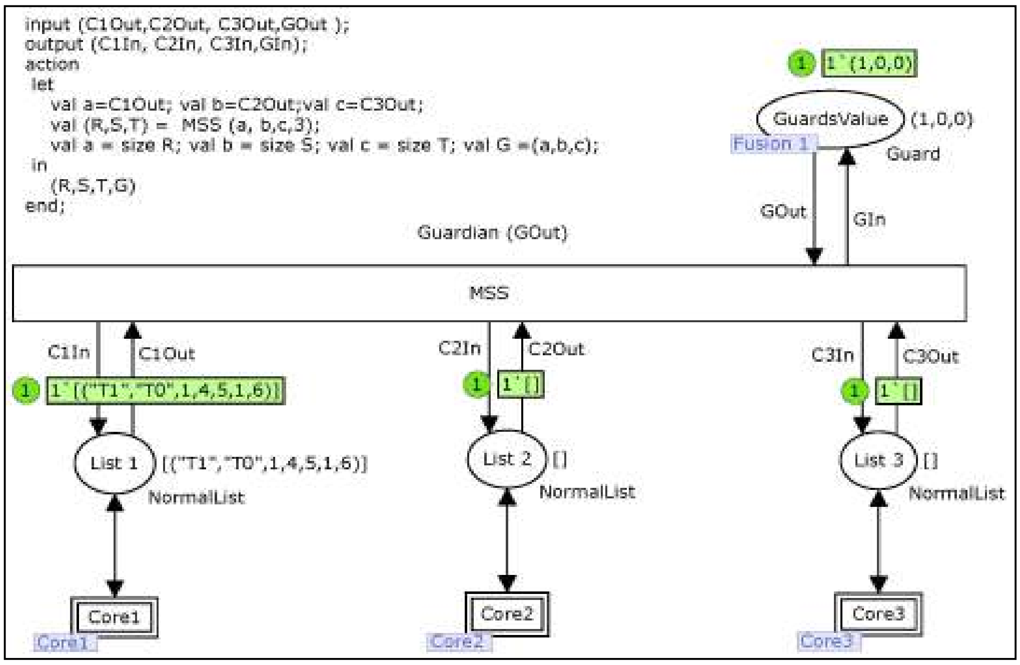
Place GuardsValue is dedicated for holding the current size of List 1, List 2, and List 3. The purpose of this place is to control the activation of the cores. The initial value of the place is (1,0,0) which indicates that there is only one thread in Core 1, second and third cores have zero threads. This value is used by the cores to enable/disable their transitions so that it will be possible to enforce the MSS Transition to redistribute the threads. The Fusion tag [21] at the lower left corner of the place indicates that this place is a shared place. In other words, places with the same fusion number in different cores share the same area.

4.1.2. Main Page Transition

In CPN, the model’s part that is in charge of doing actions (threads movement, division, etc in our model) is the Transition (such as MSS Transition in Fig. 6). Any transition can be executed if it has enough tokens (data) in its input places, and at the same time, the guard function (if exists) returns the value True. In our example, the Transition MSS has three input lists arguments, C1Out, C2Out, and C3Out (all of them of type *NormalList*), and it has a guard function (Guardian). The Guardian Function is a simple SML Boolean function that is designed to return True or False. The function returns True if and only if at least one of the input lists (that is one of the cores) has zero Normal threads, and at the same time, at least one of the same input lists has more than one Normal thread; otherwise, it returns False. It is obvious that there is no meaning behind redistributing the threads if all the cores are empty (idle) or all of them are non-empty (busy). The execution of the MSS Transition is represented by running the SML code located above the transition. The main thing in this code is the calling of the MSS function that is an SML function designed to execute the flowchart in Fig. 2. The output of this function is the balanced lists of threads represented by C1In, C2In, and C3In. In addition, the code calculates the size of threads of each core returned as GIn.

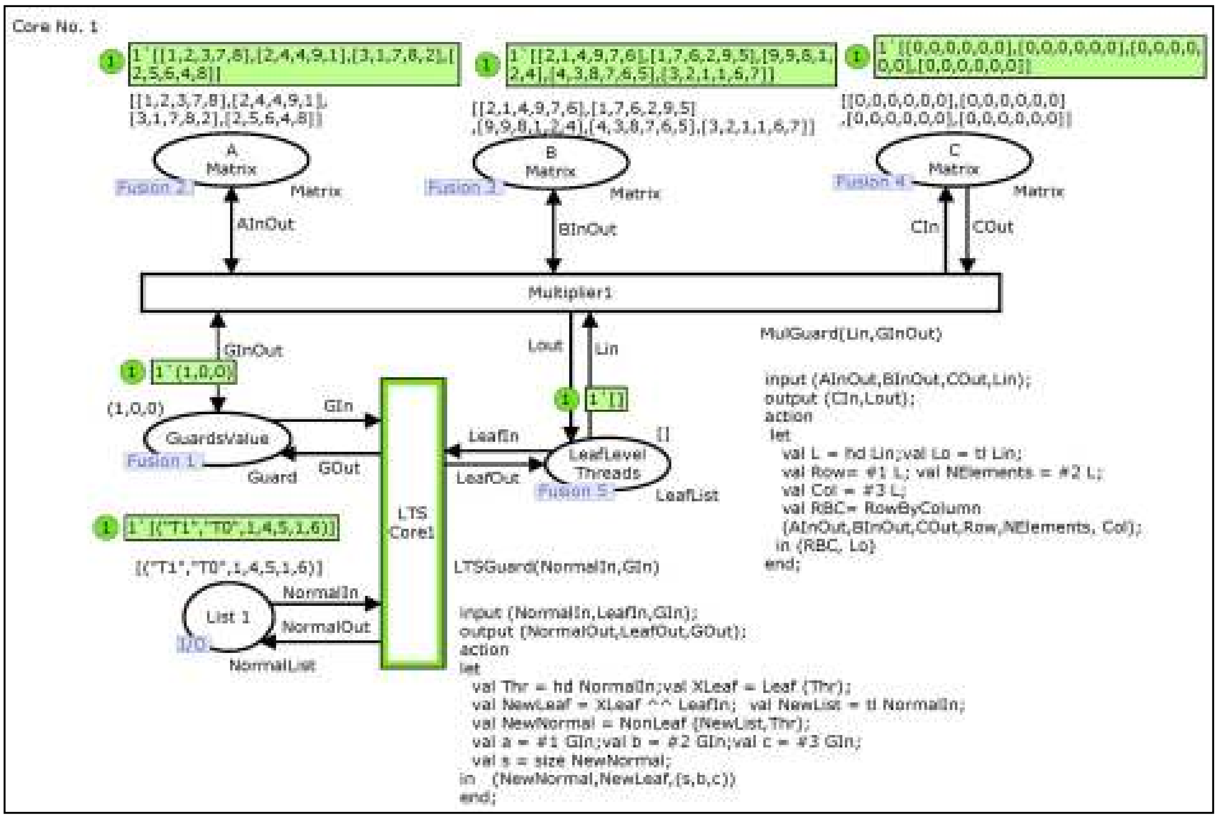
4.1.3. Main Page Substituted Transitions

At the bottom of the main page, three substituted transitions (Core1, Core2, and Core3) are located. In the hierarchical CPN models, a substituted transition represents a whole page. Thus the substituted transition is a replacement for a whole core. At the lower left corner of each substituted transition, there is a substituted tag; it is an indication to a subpage. Fig. 7 shows the components of Core 1; other cores (Core 2 and Core 3) will have exactly the same structure.



```
input (C1Out,C2Out, C3Out,GOut );
output (C1In, C2In, C3In,GIn);
action
 let
   val a=C1Out; val b=C2Out;val c=C3Out;
   val (R,S,T) =  MSS (a, b,c,3);
   val a = size R; val b = size S; val c = size T; val G =(a,b,c);
 in
   (R,S,T,G)
end;
```

Fig. 6: The CPN Model of the Main Page



```
input (AInOut,BInOut,COut,Lin);
output (CIn,Lout);
action
 let
   val L = hd Lin;val Lo = tl Lin;
   val Row= #1 L; val NElements = #2 L;
   val Col = #3 L;
   val RBC= RowByColumn
   (AInOut,BInOut,COut,Row,NElements, Col);
  in (RBC, Lo)
end;
```

```
input (NormalIn,LeafIn,GIn);
output (NormalOut,LeafOut,GOut);
action
let
  val Thr = hd NormalIn;val XLeaf = Leaf (Thr);
  val NewLeaf = XLeaf ^^ LeafIn;  val NewList = tl NormalIn;
  val NewNormal = NonLeaf (NewList,Thr);
  val a = #1 GIn;val b = #2 GIn;val c = #3 GIn;
  val s = size NewNormal;
in  (NewNormal,NewLeaf,(s,b,c))
end;
```

Fig. 7: The CPN Model of a Single Core

4.2. Core 1 (Sub Page 1)

There are six places: List 1, GuardsValue, LeafLevel Threads, A Matrix, B Matrix, and C Matrix. In addition, there are two transitions: LTS Core 1 and Multiplier1. Other cores will have exactly the same number of places and transitions.

4.2.1. Core 1 Places

• Place List 1 represents the same List 1 place in Fig. 6. The I/O tag located at the lower left corner of the place indicates the ability to get/remove normal threads to/from the core under the control of MSS in Fig. 6. CPN-Tool [21] provides an internal mechanism of adding/removing data (threads) from the main page to the sub pages.

• Place LeafLevel Threads holds a list of another kind of threads. A LeafLevel thread has the following structure: Row * Size * Column. It represents the number of the desired row of the first matrix followed by the number of elements in the row, followed by the number of the desired column of the second matrix.

• The GuardsValue place shared the same area with GuardsValue place in the main page.

• The places A Matrix and B Matrix hold the data (numbers) of the given two matrices respectively. The numbers inside the two matrices organised as lists of numbers. The output matrix, C Matrix, holds only zeros.

Places LeafLevel Threads, GuardsValue, A Matrix, B Matrix, and C Matrix are fused. Fusion places share the same area. That is, Places: A Matrix in Core1, A Matrix in Core2, and A Matrix in Core3 shared the same area. The same thing is applied for other fused places.

4.2.2. Core1 Transitions

• Transition LTS Core1 is in charge of the following:

1- Reading the contents of the Normal and LeafLevel threads lists through NormalIn and LeafIn respectively. In addition, the transition reads the sizes of the lists through GIn.

2- Updating the contents of the Normal and LeafLevel threads lists as illustrated in Fig. 4.

3- Updating of the content of GuardsValue place. This process is done according to the new size of the Normal List. The arguments of the GuardsValue Place say that X, Y, and Z are updated as follows: The transition updates only its own argument. That is, Transition LTS Core1 updates X, Transition LTS Core 2 updates Y, etc. The updating is sent back as GOut. The code below the LTS Core 1 Transition is in charge of executing all the mentioned actions of this transition.

The Transition LTS Core1 is accompanied with a guard. The LTSGuard function controls the activation of the transition. The mechanism of this function has been illustrated in Fig. 5.

• Transition Multiplier1 is responsible of multiplying a single row from A Matrix by a single column from B Matrix. The result of multiplication is stored in the appropriate location of the C Matrix. The code that is in charge of doing multiplication is located at the lower right corner of the transition. The transition is accompanied by a guard as in the previous transition. The MulGuard function works as LTSGuard with one difference. LTSGuard deals with Normal threads while MULGuard deals with LeafLevel threads.

5.0 THE RESULTS OF THE SIMULATION PROCESS

The model concurrently schedules threads creation, threads balancing, and matrices multiplications. An example of multiplying two matrices is given in Table 1:

Table 1 An example of multiplying two matrices

A Matrix, a 4×5 matrix. Never changed during the simulation	B Matrix, a 5×6 matrix. Never changed during the simulation	The C Matrix, a 4×6 resulted matrix. Initially it has only zeros. At the end of the simulation process, it will be as below
[1,2,3,7,8] [2,4,4,9,1] [3,1,7,8,2] [2,5,6,4,8]	[2,1,4,9,7,6] [1,7,6,2,9,5] [9,9,8,1,2,4] [4,3,8,7,6,5] [3,2,1,1,6,7]	[83,79,104,73,121,119] [83,95,137,94,118,100] [108,101,140,94,104,105] [103,119,126,70,143,137]

It is important to mention that CPN-Tool is a single thread tool. At any moment, the tool randomly picks one of the ready transitions (those transitions that have data in their input places, and at the same time, their Guard functions, if any, return true). Therefore, different executions of the model generate different sequences of active transitions. Nevertheless, all the execution trials lead to the same result (calculating the elements of the C Matrix).

The result of the simulation is shown in Table 2-A. The simulation process took sixty-nine steps. Prior to the simulation process, only one thread resides in Core 1 (Step 0). In the next step (step 1), Core 1's LTS is being selected by the tool; it divides its main thread into four sub threads. We notice that the MSS is the second transition chosen by the simulator; it redistributes the available threads between the cores as shown in step 2. The generation of the first Leaf-Level thread happened in step 5. In step 6, Core 2's LTS has been selected again; it exchanges one of its normal threads with two children's normal threads. The execution of the first Leaf-Level thread and getting a new value in the C Matrix has been done in step 8 through Multiplier2. In Table 2-B, we have manually collected all the possible transitions that can be done concurrently. It is clear that the model takes less time (only thirty-six steps). However, for larger arrays, we can write a simple function in CPN-ML to count the number of steps that can be run concurrently.

Table 2-A shows the sequence of transitions' executions along with contents of the cores and the current number of Leaf-Level threads. Table 2-B shows the possible concurrent activities that can be done at the same time. The S, S.T, C1, C2, C3, and L stand for Step, Selected Transition, Core 1, Core 2, Core 3 and Leaf-Level threads respectively.

Table 2-A

S	S.T	C1	C2	C3	L
0		1	0	0	0
1	LTS-Core1	4	0	0	0
2	MSS	2	1	1	0
3	LTS-Core2	2	4	1	0
4	LTS-Core3	2	4	4	0
5	LTS-Core2	2	3	4	1
6	LTS-Core2	2	4	4	1
7	LTS-Core2	2	3	4	2
8	Multiplier2	2	3	4	1
9	LTS-Core3	2	3	3	2

Table 2-B

S	S.T	C1	C2	C3	L
0		1	0	0	0
1	LTS-Core1	4	0	0	0
2	MSS	2	1	1	0
3	LTS-Core2 LTS-Core3	2	4	4	0
4	LTS-Core2	2	3	4	1
5	LTS-Core2	2	4	4	1
6	LTS-Core2	2	3	4	2
7	Multiplier2 LTS-Core3	2	2	3	2

10	Multiplier1	2	3	3	1
11	LTS-Core2	2	2	3	2
12	Multiplier3	2	2	3	1
13	Multiplier2	2	2	3	0
14	LTS-Core1	5	2	3	0
15	LTS-Core2	5	3	3	0
16	LTS-Core2	5	2	3	1
17	LTS-Core2	5	1	3	2
18	LTS-Core3	5	1	4	2
19	LTS-Core1	4	1	4	3
20	Multiplier1	4	1	4	2
21	Multiplier3	4	1	4	1
22	LTS-Core2	4	0	4	2
23	MSS	3	3	2	2
24	Multiplier3	3	3	2	1
25	Multiplier1	3	3	2	0
26	LTS-Core2	3	4	2	0
27	LTS-Core2	3	3	2	1
28	Multiplier1	3	3	2	0
29	LTS-Core2	3	2	2	1
30	Multiplier2	3	2	2	0
31	LTS-Core3	3	2	3	0
32	LTS-Core1	4	2	3	0
33	LTS-Core3	4	2	2	1
34	LTS-Core2	4	1	2	2
35	LTS-Core3	4	1	1	3
36	LTS-Core2	4	0	1	4
37	MSS	2	2	1	4
38	LTS-Core3	2	2	0	5
39	MSS	2	1	1	5
40	Multiplier2	2	1	1	4
41	Multiplier2	2	1	1	3
42	LTS-Core3	2	1	0	4
43	MSS	1	1	1	4
44	Multiplier1	1	1	1	3
45	Multiplier2	1	1	1	2
46	LTS-Core1	4	1	1	2
47	Multiplier3	4	1	1	1
48	LTS-Core2	4	0	1	2
49	MSS	2	2	1	2
50	LTS-Core3	2	2	0	3
51	MSS	2	1	1	3
52	Multiplier3	2	1	1	2
53	LTS-Core1	3	1	1	2
54	LTS-Core3	3	1	2	2
55	LTS-Core1	2	1	2	3
56	Multiplier3	2	1	2	2
57	Multiplier3	2	1	2	1
58	Multiplier1	2	1	2	0
59	LTS-Core3	2	1	1	1
60	Multiplier3	2	1	1	0
61	LTS-Core2	2	0	1	1
62	MSS	1	1	1	1

	Multiplier1 LTS-Core2				
8	Multiplier3 Multiplier2 LTS-Core1 LTS-Core2	5	3	3	0
9	LTS-Core2	5	2	3	1
10	LTS-Core2 LTS-Core3 LTS-Core1 Multiplier1	4	1	4	2
11	Multiplier3 LTS-Core2	4	0	4	2
12	MSS	3	3	2	2
13	Multiplier3 Multiplier1 LTS-Core2	3	4	2	0
14	LTS-Core2	3	3	2	1
15	Multiplier1 LTS-Core2	3	2	2	1
16	Multiplier2 LTS-Core3 LTS-Core1	4	2	3	0
17	LTS-Core3 LTS-Core2	4	1	2	2
18	LTS-Core3 LTS-Core2	4	0	1	4
19	MSS	2	2	1	4
20	LTS-Core3	2	2	0	5
21	MSS	2	1	1	5
22	Multiplier2	2	1	1	4
23	Multiplier2 LTS-Core3	2	1	0	4
24	MSS	1	1	1	4
25	Multiplier1 Multiplier2 LTS-Core1 Multiplier3 LTS-Core2	4	0	1	2
26	MSS	2	2	1	2
27	LTS-Core3	2	2	0	3
28	MSS	2	1	1	3
29	Multiplier3 LTS-Core1 LTS-Core3	3	1	2	2
30	LTS-Core1 Multiplier3	2	1	2	2
31	Multiplier3 Multiplier1 LTS-Core3	2	1	1	1
32	Multiplier3 LTS-Core2	2	0	1	1
33	MSS	1	1	1	1

63	Multiplier3	1	1	1	0
64	LTS-Core3	1	1	0	1
65	Multiplier3	1	1	0	0
66	LTS-Core1	0	1	0	1
67	LTS-Core2	0	0	0	2
68	Multiplier3	0	0	0	1
69	Multiplier2	0	0	0	0

34	Multiplier3 LTS-Core3	1	1	0	1
35	Multiplier3 LTS-Core1 LTS-Core2	0	0	0	2
36	Multiplier3 Multiplier2	0	0	0	0

6.0 DISCUSSION

The model has been executed several times. All the trials have led to the same result that is represented in calculating the elements of the C Matrix. However, different trials have different sequences of transitions' activations; this is due to the non-deterministic nature of the Colored Petri Nets. Nevertheless, all the trials generate the same number of Leaf-Level Threads, which is 24 threads, that is exactly matched the number of elements in the C Matrix.

One of the major differences that distinguishes this study from the previous studies in [12-15] is the way of stealing. Blumofe and Leiserson [3] have built their algorithm on the basis of allowing the thief core choose its victim in a random manner. Consequently, the thief core steals threads from the selected victim core. This scenario has been followed by the subsequent researches. However, the MSS in this study as the schedulers in [12-15] are in charge of this mission. In other words, we present a centralised unit (MSS) that is responsible of controlling threads movement among the core. As the number of cores per chip increases, we may reach to the level of having several thousands of cores on a single chip. The management of this number of cores needs to be controlled. As a result, having a random method in choosing cores may no longer suit this advancement in computer architecture. The MSS can deal with any number of cores. It is a scalable scheduler that can concurrently redistribute threads among the modelled cores regardless of the type of problems in which those cores are involved. That is, the scheduler can be used to solve any Divide and Conquer problem.

The MSS has been designed to solve the limitations founded in the previous schedulers [12-15]. The scheduler in [12, 13] was designed to assign only a single for each thief core while in the MSS more than one threads can be assigned to each thief core at the same time. In addition, in [12, 13], the chosen victim core may not be the best choice since the searching for it is done serially, as a result, threads are extracted from the first encountered victim core. This will extend the time needed to balance the number of threads because we have to repeat the process of balancing several times while in the MSS we achieve balancing from the first trial. In [14], we partially solved the wasted time problem in [12, 13] through searching for the richest core first. However, in certain cases, the richest core itself may be at the end of the cores list. Although that [14] shows a better performance than [12, 13], yet it doesn't satisfy our need for a significant change. In [15], we presented a partial multi stealing scheduler that focuses on wealthy victim cores and thief cores only. The poor victim cores have been execluded. Compared with MSS, the presented partial multi stealing scheduler consumes less time, however, this can be seen as a special case. However, the MSS is more general and it is adequate with any situation of threads distribution.

The result of reallocating is fair enough. However, we may see that those cores that come first in order may get extra threads but no more than one extra thread per core. This is due to the additional number of threads that remains after redistributing the majority number of them.

The entire Local Threads Schedulers (LTSs) and Multipliers are working concurrently. At any moment, in any core, we may find both the LTS and the Multiplier working at the same time or only one of them is functioning; this is due to the availability of Normal and Leaf-Level threads in the core.

The computation of the parameters: StartRow, EndRow, StartColumn, and EndColumn in Fig. 4, greatly increased the independence in computing the elements of the resulted matrix (C Matrix). Starting from the root, the LTS generates distinct, independent, and dividable/non-dividable threads that are able to be reallocated by the MSS.

In the LTS, the ability to divide a single Normal thread into four Normal Threads has highly minimised the time needed in reaching Leaf-Level threads. This can be a starting point to generate eight, sixteen, or more threads directly. However, this will depend on the size of the problem itself. In addition, the LTS algorithm will have more steps to facilitate the direct generation of eight threads, sixteen threads, etc. Anyway, the generation of more than two threads in one step has no doubt improved the performance of the model.

The Guarding Mechanism that we added to the model easily freezes the activities for LTSs and Multipliers when there is a need to reallocate threads. The two types of Guards: LTS Guard and Multiplier Guard test the sizes of the lists of threads represented as GuardsValue, and then decide whether they should activate or deactivate the transitions. In case there is a need for reallocating the threads, the entire actions of the core are disabled by the Guards, i.e. the cores are no longer be able to divide threads and multiply rows by columns. This will leave no choice to the model other than reallocating the threads and then unlocking other activities.

We can classify the results in Table 2-A into the following groups:

•**Group 1 (MSS Distributions' Steps)**: The MSS has been activated in the following steps 2, 23, 37, 39, 43, 49, 51, and 62 (subtotal is 8 steps). The steps clearly showed the reallocating of threads among the modelled cores using MSS (Fig. 2).

•**Group 2 (Normal Threads Generations' Steps)**: The generation of 4 / 2 /1 normal thread(s) have been taken place in the steps: 1, 3, 4, 6, 14, 15, 18, 26, 31, 32, 46, 53, and 54 (subtotal is 13 steps).

•**Group 3 (Leaf-Level Threads Generations' Steps)**: Leaf-Level Threads are created in the steps: 5, 7, 9, 11, 16, 17, 19, 22, 27, 29, 33, 34, 35, 36, 38, 42, 48, 50, 55, 59, 61, 64, 66 and 67 (subtotal is 24 steps).

•**Group 4 (Multipliers Actions' Steps)**: The process of computing new C Matrix elements occurred in the steps: 8, 10, 12,13, 20,21, 24, 25, 28, 30, 40, 41, 44, 45, 47, 52, 56, 57, 58, 60, 63, 65, 68 and 69 (subtotal is 24 steps).

In Table 2-B, all the possible transitions that can occur concurrently are grouped together. It is clear that some transitions can be run at the same time with other transitions. Only thirty-six steps are needed to achieve the same task. For instance, in step 25 (Table 2-B), five transitions can be executed at the same time in any sequence. Prior to this step, we had four Leaf-Level threads, thus any one of the three Multipliers can be executed in any sequence. LTS-Core1 generates four Normal threads while LTS-Core2 generates a single Leaf-Level thread. All the above actions can be done in any sequence, and at the same time, without interfering between each other. However, in other steps, certain actions have to be done individually as in step number one (only Core 1 has a thread), also in step number four where LTS-Core2 has previously executed concurrently with LTS-Core3 in step 3, and the same transition (LTS-Core2) has been chosen again by the simulator in step five. Thus it has to be executed individually since it cannot execute the same transition twice at the same time. In addition, all the MSS have to be executed individually.

Adding more cores will no doubt decrease the number of steps especially when we improve the LTS algorithm to generate eight or more threads in one step. This will definitely increase the number of possible concurrent transitions among the modelled cores. However, any increase in the number of cores should be accompanied with an increase in generating more threads in one step. The reason is that adding more cores without a suitable increase in threads will make some cores idle even after applying the MSS.

7.0 CONCLUSION

In this study, we introduce a new Colored Petri Nets model for solving Divide and Conquer problems on a multicore environment. The model is scalable, i.e. it can be easily expanded by adding more cores. In addition, the model is concurrent and it uses multithreaded in scheduling the activities of the cores. Two new schedulers have been developed to achieve the mission of the model. The Multi Stealing Scheduler effectively redistributes threads among the modelled cores through stealing more than one thread from different cores and redistributing them among the idle cores. The Multi Stealing Scheduler is an open technique; it can be used to control threads distribution for any Divide and Conquer problem. The Local Threads Scheduler manages threads creation, division, and cooperating with Multi Stealing Scheduler in reallocating threads. In addition, the Local Threads Scheduler provides a new recursive approach in getting the necessary information to multiply two matrices. Besides the two schedulers, we have developed a new mechanism that controls the activation of the model's elements. The Guard mechanism proves its efficiency in enforcing the Multi Stealing Scheduler to redistribute the threads among the modelled cores when there is a need. This has greatly reduced the time some cores being in an idle situation as in the previous studies that used Work Stealing technique. The result of the simulation shows a high percentage of concurrency among the different actions of the model.

REFERENCES

[1] F. P. Miller and A. F. Vandome, *Divide and Conquer Algorithm*: Alphascript Publishing, 2010.

[2] T. H. Cormen, *et al.*, *Introduction to Algorithms*, Third ed.: MIT Press, 2009.

[3] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM,* vol. 46, pp. 720-748, Nov. 1999.

[4] N. A. Arora, *et al.*, "Thread Scheduling for Multiprogrammed Multiprocessors," in *The Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Puerto Vallarta, Mexico, 1998, pp. 119-129.

[5] D. Hendler and N. Shavit, "Non-blocking Steal-Half Work Queues," in *In The twenty-first annual symposium on Principles of distributed computing*, Monterey, California, 2002, pp. 280 – 289.

[6] U. Acar, *et al.*, "The Data Locality of Work Stealing," presented at the The twelfth annual ACM symposium on Parallel algorithms and architectures Bar Harbor, Maine, United States 2000.

[7] D. Hendler, *et al.*, "A Dynamic-Sized Non-blocking Work Stealing Deque," *Journal of Distributed Computing,* vol. 18, pp. 189-207, 2005.

[8] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," presented at the The seventeenth annual ACM symposium on Parallelism in algorithms and architectures, Las Vegas, Nevada, USA, 2005.

[9] K. Agrawal, *et al.*, "Adaptive work-stealing with parallelism feedback," *ACM Transactions on Computer Systems (TOCS),* vol. 26, p. 7, 2008.

[10] Ž. Vrba, *et al.*, "Limits of work-stealing scheduling," in *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science*. vol. 5798, ed: Springer, 2009, pp. 280-299.

[11] X. Ding, *et al.*, "BWS: Balanced Work Stealing for Time-Sharing Multicores," in *EuroSys '12 Proceedings of the 7th ACM european conference on Computer Systems* Bern, Switzerland, 2012, pp. 365-378.

[12] A. Al-Obaidi and S. P. Lee, "A Concurrent Multithreaded Scheduling Model for Solving Fibonacci Series on Multicore Architecture," *International Journal of Advancements in Computing Technology,* vol. 3, pp. 24 - 37, 2011.

[13] A. Al-Obaidi and S. P. Lee, "A Concurrent Colored Petri Nets Model for Solving Binary Search Problem on a Multicore Architecture," in *The 2nd International Conference on Software Engineering and Computer Systems*, University Malaysia Pahang, Malaysia, 2011.

[14] A. Al-Obaidi and S. P. Lee, "A multithreaded scheduling model for solving the Tower of Hanoi game in a multicore environment," *Maejo International Journal of Science and Technology,* vol. 6, pp. 282-296, 2012.

[15] A. Al-Obaidi and S. P. Lee, "A Partial Multi Stealing Scheduling Model for Divide and Conquer Problems," in *International Conference on Computer and Communication Engineering (ICCCE 2012)*, Kuala Lumpur, Malaysia, 2012, pp. 153-157.

[16] K. Jensen and L. M. Kristensen, *Colored Petri Nets: Modeling and Validation of Concurrent Systems*: Springer-Verlag New York Inc, 2009.

[17] J. Peterson, "Petri Nets," *ACM Computing Surveys,* vol. 9, pp. 223-252, 1977.

[18] T. Murata, "Petri Nets: Properties, Analysis and Applications," in *Proceedings of the IEEE*, 1989, pp. 541 - 580.

[19] E. Gansner and J. Reppy, *The Standard ML Basis Library*: University of Cambridge, 2002.

[20] J. Ullman, *Elements of ML Programming*: Prentice-Hall, 1998.

[21] K. Jensen*, et al. CPN Tools Web Page*. Available: http://cpntools.org/

Biography

Alaa M. Al-Obaidi is currently a PhD student at the Faculty of Computer Science and Information Technology, University of Malaya. The general idea of his research is about developing new concurrent multithreaded schedulers' models that fit the multicore environment. This paper represents his fifth contribution in this area of research.

Sai Peck Lee is a professor at the Department of Software Engineering, University of Malaya. She got her Ph.D. degree in Computer Science from Université Paris 1 Panthéon-Sorbonne. Her current research interests include Object-Oriented Techniques and CASE tools, Software Reuse, Requirements Engineering, and software quality. She is a member of IEEE and a founding member of Informing Science Institute. She is in several experts review panels, both locally and internationally.