Maejo Int. J. Sci. Technol. **2012**, *6*(02), 282-296

Maejo International Journal of Science and Technology

ISSN 1905-7873
Available online at www.mijst.mju.ac.th

Full Paper

A multithreaded scheduling model for solving the Tower of Hanoi game in a multicore environment

Alaa M. Al-Obaidi * **and Sai Peck Lee**

Department of Software Engineering, Faculty of Computer Science & Information Technology, University of Malaya, Kuala Lumpur, Malaysia

* Corresponding author, e-mail: alaa@siswa.um.edu.my

Received: 10 May 2011 / Accepted: 1 August 2012 / Published: 7 August 2012

Abstract: Modern computer systems greatly depend on multithreaded scheduling to balance the workload among their working units. One of the multithreaded scheduling techniques, the work-stealing technique has proven effective in balancing the distribution of threads by stealing threads from the working cores and reallocating them to the non-working cores. In this study, we propose a new strategy that extends the work-stealing technique by enabling it to select the richest core prior to any redistribution process. In order to obtain practical results, we applied this new strategy of balancing threads to one of the divide-and-conquer problems, the Tower of Hanoi game. A multithreaded scheduling model which is a hierarchical model was designed to work under the control of this new strategy. A modelling tool was used to simulate and verify the designed model. The proposed model was shown by the simulation process to exhibit consistency and stability in reaching the desired result. Scalability, concurrency, simplicity and fair load distribution among the modelled cores are the main beneficial characteristics of this model.

Keywords: multithreaded scheduling, Tower of Hanoi, work-stealing technique, divide-and-conquer problem, coloured Petri nets

INTRODUCTION

The general trend in the multicore industry is to increase the number of cores per chip. The rapid growth in the number of cores per chip imposes new requirements for software designers who need to make their products more adaptable with the new developments taking place on the hardware side of the industry. In order to address this challenge, multithreaded scheduling techniques have been developed to provide a solution for managing the continuing increase in the

Maejo Int. J. Sci. Technol. **2012**, *6*(02), 282-296

number of cores. These techniques are in charge of assigning and distributing the workload among the cores. For instance, the work-stealing scheduling technique has proven effective in balancing the distribution of threads in multicore environments. The technique aims to balance the distribution of threads by stealing threads from the working cores and reallocating them to the non-working cores [1-9].

However, in view of the increasing number of cores, the work-stealing technique may not be sufficient to meet the demands of multicore technology. Hence, motivated by the need to meet these demands, the development of new software strategies that are built on the basis of the work-stealing technique is the main aim of this study. We aim to design a multithreaded scheduling model that improves the work-stealing technique by enabling it to select the richest core (i.e. the one that has the most threads) when balancing thread distribution among working and non-working cores.

In this paper, we present a new strategy for balancing the distribution of threads. The richest selection scheduling (RSS) strategy extends the work-stealing technique so that it can choose the core that has the most threads from among the available cores when there is a need to redistribute threads. We have designed a multithreaded scheduling model, which is a hierarchical model that consists of two types of schedulers. The first scheduler is called a High-Level Scheduler, which is in charge of applying the RSS strategy. This scheduler was implemented through designing a new algorithm, the RSS algorithm. The algorithm is in charge of making all the cores work concurrently by stealing a certain number of threads from the working (victim) cores and reallocating them to the non-working (thief) cores. Where two or more victim cores are available, the richest core (the one that has the most threads) is preferred. The second scheduler is called the Core Scheduler. This scheduler is in charge of thread creation and calculation. It is also used to create and calculate the moves for the Tower of Hanoi game. A new algorithm, the Tower of Hanoi multithreaded scheduling (THMS) algorithm was designed for this purpose. We applied these algorithms by using coloured Petri net (CPN) [10-13] as the modelling language and coloured Petri nets meta language (CPN-ML) [14-16] as the coding language. We used CPN-Tool [17] as a software tool to enable us to create, simulate and verify the correctness of the designed model.

The Tower of Hanoi

The general idea behind any divide-and-conquer problem can be summarised as the continuous partitioning of a given problem till a certain condition is achieved [18-19]. After that, the conquering part is executed in a manner that depends on the nature of the problem. The Tower of Hanoi computer game is based on a puzzle that was first published by a French mathematician (François Éduoard Anatole Lucas) in 1883 [18]. The game 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 1 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 $(2^n) - 1$.

Maejo Int. J. Sci. Technol. **2012**, *6*(02), 282-296

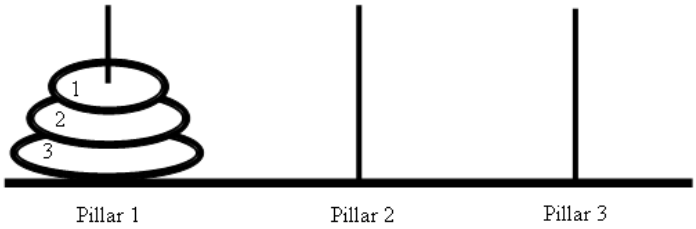


Figure 1. The Tower of Hanoi pillars with three disks

Prior to the development of multicore technology, the solution to such problem had to be done serially. That is, the generation of the steps had to be made one by one. However, with the advent of multicore technology, solutions can be done faster through multithreading and concurrency techniques [20]. However, the modelling of multithreaded concurrent systems represents a great challenge due to the non-deterministic nature of such systems, in addition to the difficulty of thread synchronisation The divide-and-conquer technique, concurrency, and multithreaded scheduling have the following in common: the main problem can be divided into several parts, and each part can be assigned to a thread. The ability to allocate a core for each thread makes all the threads work concurrently.

RELATED WORK

The main idea of work stealing is attributed to Blumofe and Leiserson [2]. They designed an algorithm that is able to schedule well-structured multithreaded computations. Although the result was good, the algorithm could not deal with the new environments where multiprogramming is used. This is due to the design of the algorithm's mechanism which deals with a fixed set of processors. Arora et al. [3] made an improvement by designing an algorithm that can deal with a multiprogrammed environment instead of a dedicated one. However, their algorithm encountered some problems with respect to memory management. Overflow easily occurred due to the use of arrays in representing deques. The sizes of the arrays had to be adjusted many times [4]. Several improvements have been built on the development made by Arora et al., e.g. 'stealing the half', a new idea that was introduced by Hendler et al. [5]. As the name implies, half of the threads can be stolen in one trial. Two other contributions based on Arora et al. are 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 suggested by Acar et al. [6], and a simple lock-free work-stealing technique proposed by Chase and Lev [7]. Chase and Lev's algorithm is based on using a cyclic array that can easily deal with overflows; memory size is the only limitation to their algorithm.

We have developed a scheduling strategy which has been designed on the principle of work stealing and is simple and direct [8-9]. In this strategy, which was applied to solve two of the divide-and-conquer problems, namely Fibonacci Series and Binary Search, the search for the victim cores is simply achieved through investigating each core starting from core number 1 to the end of the core list. Although this technique succeeds in balancing thread distribution in a direct and simple way, it lacks selection efficiency. In certain cases, the scheduler may pick a victim core that has too few threads compared with other victim cores. This leads to unnecessary repetitions of the distribution process because the process itself may not satisfy the needs of the thief cores.

Maejo Int. J. Sci. Technol. **2012**, *6*(02), 282-296

CPN and CPN-Tool

CPN is a graphical discrete-event language designed to model and validate concurrent systems [10-13]. CPN has been developed from Petri nets [21-22], the main difference between Petri nets and CPN being the addition of types to CPN as well as the ability to write expressions and functions in standard meta language (SML) [14-16]. The CPN model is an executable model in the sense that the process of execution shows the different states of the system represented by the model.

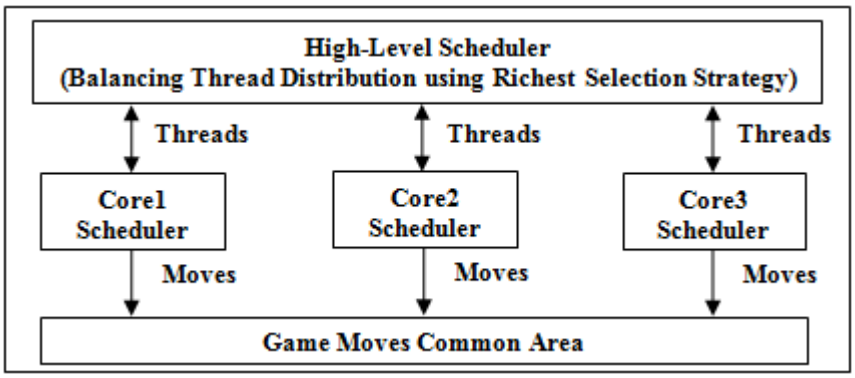
CPN-Tool is a software tool that is designed to create, simulate and validate CPN [17]. CPN-Tool is a GUI tool that provides all the interaction methods such as menus and toolbars in addition to giving feedback messages when errors are encountered during the process of checking the syntax of codes. The ability to execute models is one of the main advantages of this tool. In addition, the tool supports hierarchical modelling, which simplifies complicated designs. Additionally, CPN-Tool provides a monitoring mechanism which permits observation of the behaviour of the elements of the model. CPN-Tool uses CPN-ML for writing declarations, expressions and codes [17]. CPN-ML is a language that can be used to write net inscriptions. These inscriptions include expressions on the arcs, codes that control transitions as well as declarations of the types and variables that are included in the CPN model. The CPN-ML has been built based on SML [14-16]. The tool has proved successful in the world of modelling [17].

DESIGN METHODOLOGY

In this study, we propose a concurrent multithreaded scheduling model that is able to balance thread distribution among a set of modelled cores through applying a new strategy, namely the RSS strategy. In addition, the model is able to schedule the computations of moves in the Tower of Hanoi game. The methodology is based on building two types of schedulers: the High-Level Scheduler and the Core Scheduler (Scheme 1). Basically, there is only one high-level scheduler while there are as many core schedulers as the number of modelled cores (three in the case of Scheme 1). The High-Level Scheduler exchanges threads with the Core Schedulers while the latter are in charge of creating these threads in addition to generating game moves and keeping them in a common area.

The thread is designed as a 7-tuple: *ThreadId*, *FatherId*, *Disk-No*, *Order*, *Source*, *Through* and *Destination*, the first two representing the thread's identifier and the thread's father identifier respectively. *ThreadId* and *FatherId* are denoted as *x* (x = a positive integer). The third parameter, *Disk-No*, holds the number of the disk. The fourth parameter, *Order*, represents a sequence number that is reserved for the game moves. The last three parameters, *Source*, *Through* and *Destination*, represent the numbers of the pillars.

Maejo Int. J. Sci. Technol. **2012**, *6*(02), 282-296



Scheme 1. Scheme of the model with three cores

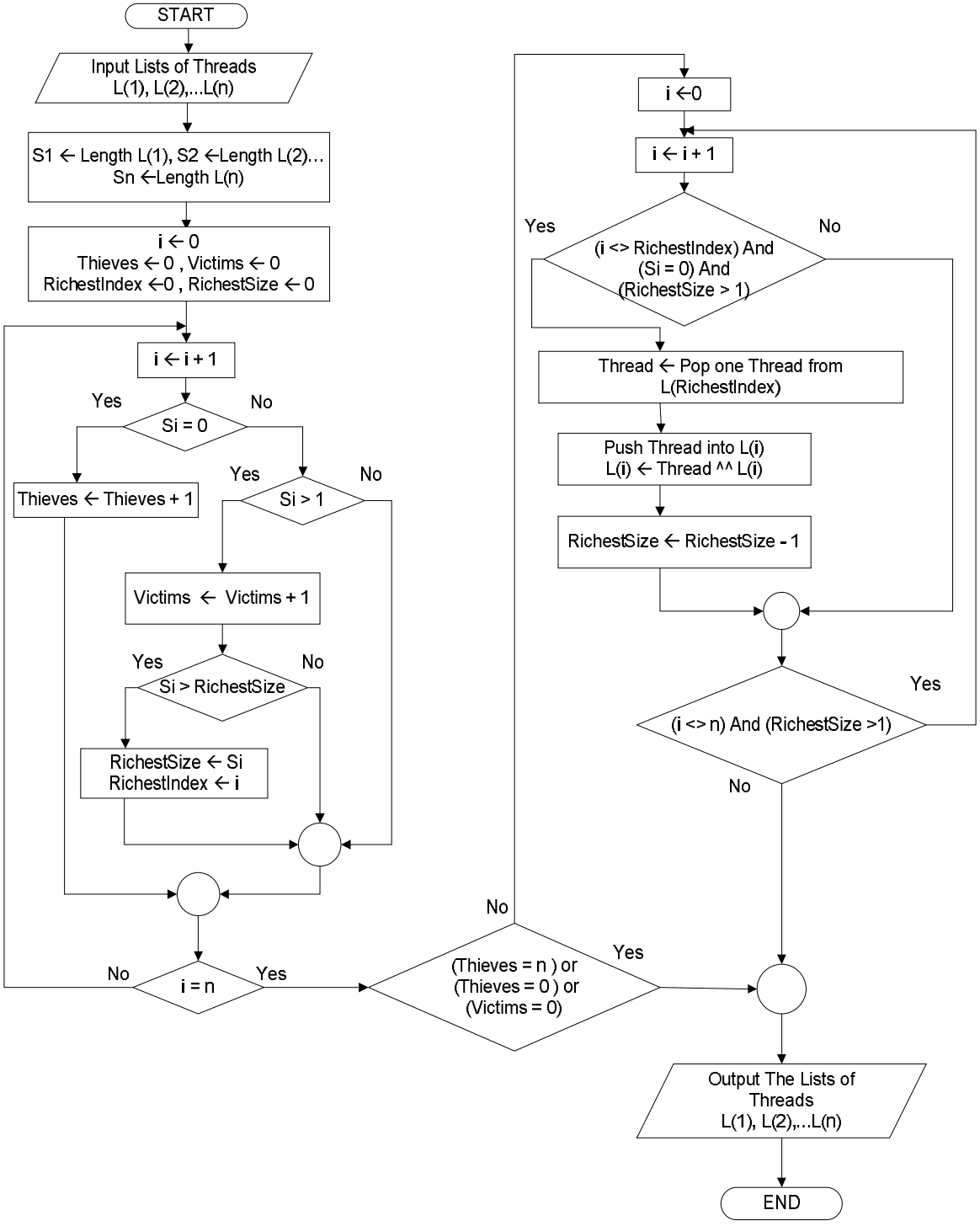
The game move is modelled as a 4-tuple: *Order*, *Disk-No*, *Source* and *Destination*. It holds the information of moving a single disk numbered as *Disk-No* from pillar number *Source* to pillar number *Destination*. The *Order* parameter is necessary for putting all the moves in order in the common area. Since each modelled core concurrently generates its own set of moves, it becomes necessary to add a parameter that controls the arranging of these moves.

The High-Level Scheduler (Scheme 2) has the role of controlling thread distribution among the modelled cores. The process of distribution can be achieved through stealing threads from the working (victim) core that has the highest number of threads and then reallocating those threads to a set of non-working (thief) cores.

The mechanism of the Core Scheduler, as shown in Scheme 3, creates a binary tree of threads and game moves. The scheduler divides any given thread into left and right descendant threads and the division process continues until no thread can be divided. The number 0 is chosen as the *Order* number of the main thread. Therefore, the *Order* numbers of the left-thread children will be less than their counterparts on the right side. Thus, some threads' identifiers have negative *Order* numbers (in CPN-ML, the negative sign is ~) and the other threads' identifiers have positive *Order* numbers (an example being given in Figure 5). The scheduler also generates game moves; a game move is configured from the thread itself. It is an abstract representation of the thread by holding the *Disk-No* and the numbers of the *Source* and *Destination* pillars. All the cores' schedulers keep their game moves in a common area, as shown in Scheme 1. The schedulers are independent; they can work concurrently to exploit the cores to reduce the overall execution time.

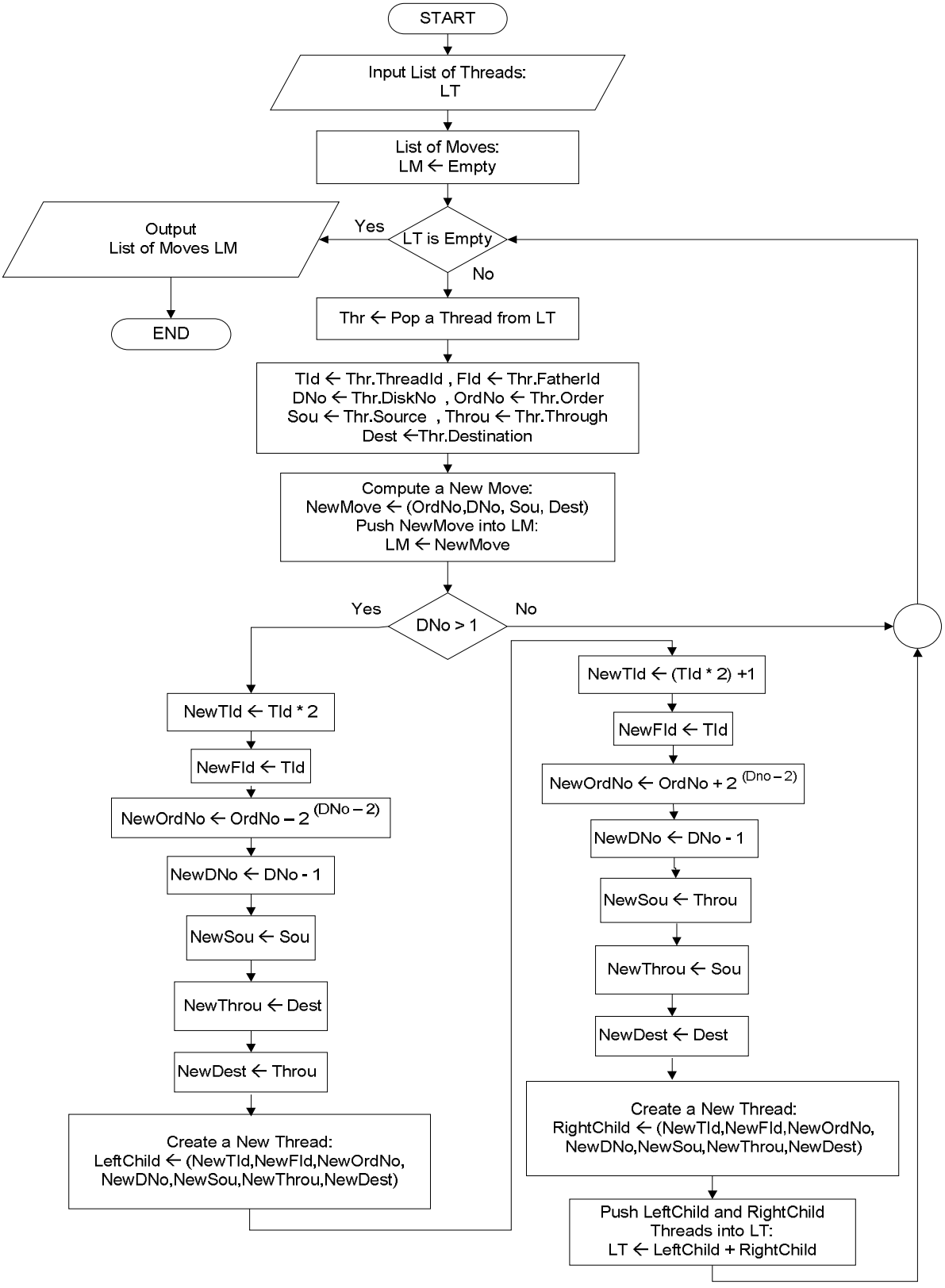
Scheme 4 shows an example of a binary tree of threads and game moves for a model that has three disks. Threads and game moves are included inside rectangles and ellipses respectively. The first resulting game move consists of moving disk 1 from pillar 1 to pillar 3; the second move consists of moving disk 2 from pillar 1 to pillar 2; etc. The first parameter in each move is used to put the generated move in its proper order.

Maejo Int. J. Sci. Technol. **2012**, *6*(02), 282-296



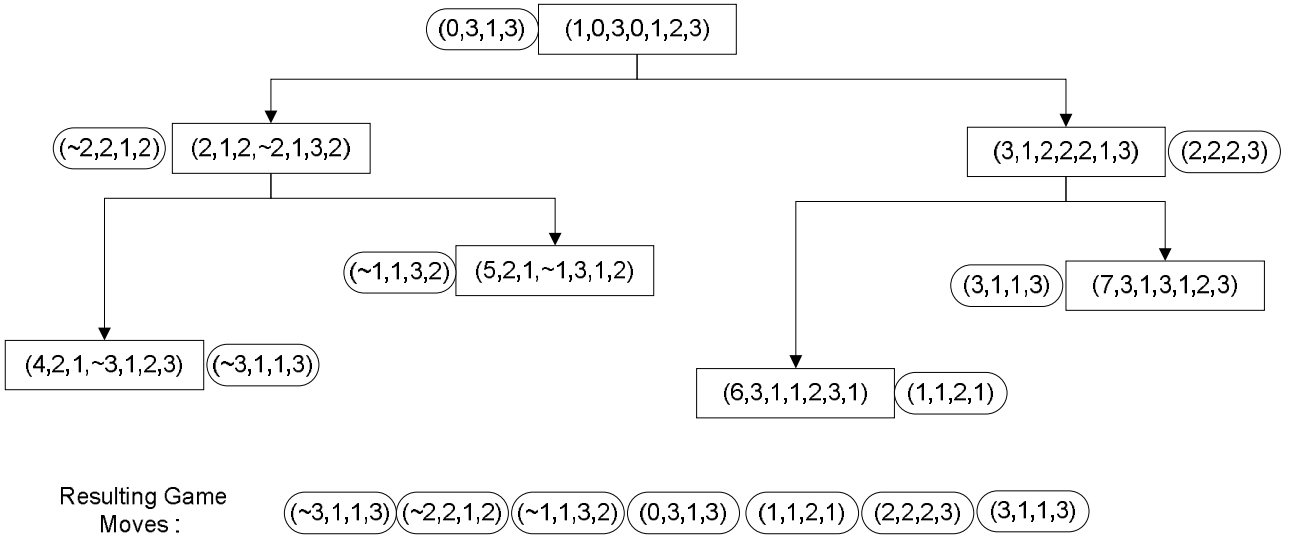
Scheme 2. High-Level Scheduler mechanism – RSS strategy

Maejo Int. J. Sci. Technol. **2012**, *6*(02), 282-296



Scheme 3. Core Scheduler mechanism – scheduling of moves in the Tower of Hanoi game

Maejo Int. J. Sci. Technol. **2012**, *6*(02), 282-296



Scheme 4. Example of scheduling the threads and moves of Tower of Hanoi game with three disks

Building the CPN Model

The CPN model of the main page is shown in Scheme 5. The model simulates Scheme 1. It includes three places (Core1List, Core2List and Core3List), one transition (Coordinator) and three substituted transitions (Core1, Core2 and Core3). In CPN a place is an oval shape which holds data. In our model each place holds a list of threads. The place usually is accompanied by the following three items:

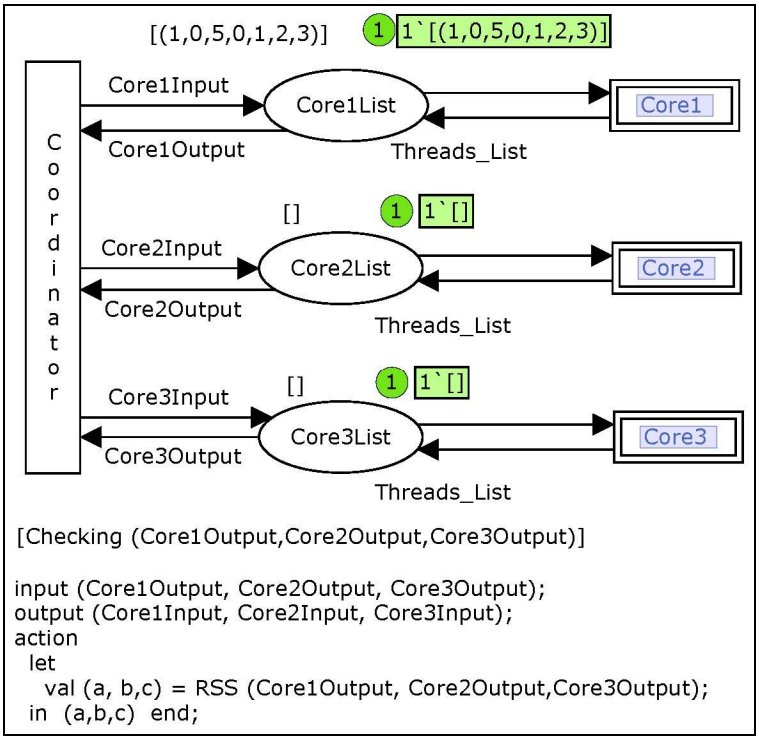
1) An initial value which is located on the top left of the place. The place Core1List contains a list with a single thread called the main thread ([(1, 0, 5, 0, 1, 2, 3)]). The first two parameters, i.e. 1 and zero, symbolise *ThreadId* and the thread's *FatherId* respectively. Number 5 represents the number of disks; for example, we are planning to move five disks from pillar 1 to pillar 3. The fourth parameter, zero, stands for the *Order* parameter of the first (main) thread. The remaining three parameters represent the pillar numbers 1, 2 and 3. The cores, Core2List and Core3List, have the initial value '[]', which means that the cores initially have no threads (empty list of threads).

2) A current value which is located on the top right of the place. The current values are frequently changed during the simulation process while the initial values never change. A current value is symbolised by a circle and a rectangle. The rectangle displays the value while the circle shows the number of values. In the case of the list of threads, we have one value, that is, a single list of threads.

3) A data type *Threads-List* which is located on the bottom right of the place. This data type is designed in CPN-ML to indicate the type of data inside the place.

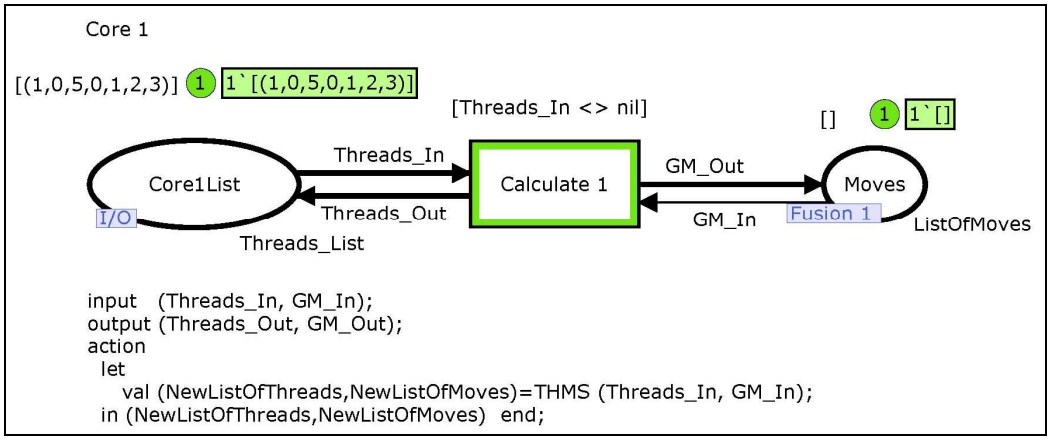
In CPN, transitions represent the action units. The transition Coordinator is in charge of executing high-level scheduling (described in Scheme 2). To activate the transition Coordinator, the Checking function should return a Boolean true value. This function works as a guard; it returns true only if there is at least one victim and at least one thief. The code below the Checking function represents instructions that will be carried out when transition Coordinator is executed by the CPN-Tool's simulator. The transition reads the lists of the cores through Core1Output, Core2Output and

Maejo Int. J. Sci. Technol. **2012**, *6*(02), 282-296



Scheme 5. CPN model of the main page

Core3Output, updates the lists and then sends the feedback as Core1Input, Core2Input and Core3Input. The transition executes the RSS function to do the updating task. This CPN-ML function has been designed to execute the mechanism of the high-level scheduling strategy, which results in a fair distribution of the threads. Finally, to the right of each place there is a substitute transition: Core1, Core2 and Core3. Each substitute transition corresponds to a core scheduler (Scheme 1). The structure of a substitute transition is illustrated in Scheme 6, which represents Core1. Other substitute transitions have the same structure.



Scheme 6. CPN model of Core1

Maejo Int. J. Sci. Technol. **2012**, *6*(02), 282-296

The model in Scheme 6 shows the content of Core1. The core consists of two places: Core1List and Moves, in addition to one transition: Calculate 1. The Calculate-1 transition reads a list of threads (stored in Core1List) which is represented by Threads_In, updates it and sends it back as Threads_Out. The I/O symbol at the lower-left corner of the place Core1List represents a port tag which is a mechanism offered by the CPN-Tool to connect places from different pages. It allows the Coordinator to add/take threads to/from the places. This means that the place Core1List in Scheme 6 is just a copy of the place Core1List in Scheme 5. This kind of hierarchy simplifies the communication between the pages' places in the model.

The Calculate-1 transition also reads a list of ordered game moves represented by GM_In (stored in the place Moves), updates it and then sends it back as GM_Out. The place Moves has an initial value which consists of an empty list '[]'. In addition, the place Moves has a Fusion-1 tag. Fused place is one of the CPN-Tool mechanisms used in creating shared areas [17]. In our example, the place Moves in each core will share the same game moves list. The Calculate-1 transition has a guard expression ([Threads_In <> nil]) located at the top of the transition. This Boolean expression controls the activation of the transition. The expression returns true (that is, allows the transition to be executed) only if the list of threads is not empty. The remaining cores have exactly the same structure as Core1 except that they start with empty lists of threads. The THMS function applies the mechanism shown in Scheme 3 – that is, this function is the CPN-ML copy of the Core Scheduler. The function receives and updates two parameters, a list of threads and a list of game moves.

Thus, both the RSS and THMS functions control the movements of the threads within the elements of the model. The RSS strategy is implemented by the RSS function while generating the threads, and the movements of the game are implemented by the THMS function.

RESULTS OF SIMULATION

CPN-Tool is a single thread tool and it chooses transitions randomly. At every stage, the tool checks the available active transitions. An active transition must fulfil two conditions: first, its input places are not empty; and second, the transition's guard function (if exists) returns true. Next, the tool picks one of the transitions that is ready in order to execute it. If the tool picks the Coordinator transition, this will cause the reallocation of the threads in the entire model (applying the RSS function). However, when the tool picks one of the Calculate transitions, then new threads and game moves are generated (applying the THMS function).

The following represents an example of a simulation process in the model. The main thread is [(1,0,5,0,1,2,3)] located in Core1. We plan to move five disks from pillar 1 to pillar 3 with the help of pillar 2. Initially, the places Core2list and Core3List are empty. The simulation process starts by executing the only active transition, i.e. Calculate 1 in Core1. Table 1 lists all the selected transitions, cores' contents and game moves during the simulation process.

Maejo Int. J. Sci. Technol. **2012**, *6*(02), 282-296

Table 1. List of transitions selected by CPN-Tool during the simulation process in addition to the current contents of the places. (Transitions Calculate 1, Calculate 2, Calculate 3 and Coordinator have been shortened to C1, C2, C3 and Coo respectively.)

Seq.	Transition	Core-1 thread	Core-2 thread	Core-3 thread	Generated move
1	-	(1,0,5,0,1,2,3)	Nil	Nil	Nil
2	C1	(2,1,4,~8,1,3,2),(3,1,4,8,2,1,3)	Nil	Nil	(0,5,1,3)
3	C1	(4,2,3,~12,1,2,3), (5,2,3,~4,3,1,2),(3,1,4,8,2,1,3)	Nil	Nil	(~8,4,1,2)
4	Coo	(3,1,4,8,2,1,3)	(4,2,3,~12,1,2,3)	(5,2,3,~4,3,1,2)	Nil
5	C2	(3,1,4,8,2,1,3)	(8,4,2,~14,1,3,2), (9,4,2,~10,2,1,3)	(5,2,3,~4,3,1,2)	(~12,3,1,3)
6	C1	(6,3,3,4,2,3,1),(7,3,3,12,1,2,3)	(8,4,2,~14,1,3,2), (9,4,2,~10,2,1,3)	(5,2,3,~4,3,1,2)	(8,4,2,3)
7	C2	(6,3,3,4,2,3,1),(7,3,3,12,1,2,3)	(16,8,1,~15,1,2,3), (17,8,1,~13,3,1,2), (9,4,2,~10,2,1,3)	(5,2,3,~4,3,1,2)	(~14,2,1,2)
8	C2	(6,3,3,4,2,3,1),(7,3,3,12,1,2,3)	(17,8,1,~13,3,1,2), (9,4,2,~10,2,1,3)	(5,2,3,~4,3,1,2)	(~15,1,1,3)
9	C1	(12,6,2,2,2,1,3),(13,6,2,6,3,2,1), (7,3,3,12,1,2,3)	(17,8,1,~13,3,1,2), (9,4,2,~10,2,1,3)	(5,2,3,~4,3,1,2)	(4,3,2,1)
10	C2	(12,6,2,2,2,1,3),(13,6,2,6,3,2,1), (7,3,3,12,1,2,3)	(9,4,2,~10,2,1,3)	(5,2,3,~4,3,1,2)	(~13,1,3,2)
11	C1	(24,12,1,1,2,3,1),(25,12,1,3,1,2,3), (13,6,2,6,3,2,1),(7,3,3,12,1,2,3)	(9,4,2,~10,2,1,3)	(5,2,3,~4,3,1,2)	(2,2,2,3)
12	C2	(24,12,1,1,2,3,1),(25,12,1,3,1,2,3), (13,6,2,6,3,2,1),(7,3,3,12,1,2,3)	(18,9,1,~11,2,3,1), (19,9,1,~9,1,2,3)	(5,2,3,~4,3,1,2)	(~10,2,2,3)
13	C2	(24,12,1,1,2,3,1),(25,12,1,3,1,2,3), (13,6,2,6,3,2,1),(7,3,3,12,1,2,3)	(19,9,1,~9,1,2,3)	(5,2,3,~4,3,1,2)	(~11,1,2,1)
14	C2	(24,12,1,1,2,3,1),(25,12,1,3,1,2,3), (13,6,2,6,3,2,1),(7,3,3,12,1,2,3)	Nil	(5,2,3,~4,3,1,2)	(~9,1,1,3)
15	C1	(25,12,1,3,1,2,3),(13,6,2,6,3,2,1), (7,3,3,12,1,2,3)	Nil	(5,2,3,~4,3,1,2)	(1,1,2,1)
16	C1	(13,6,2,6,3,2,1), (7,3,3,12,1,2,3)	Nil	(5,2,3,~4,3,1,2)	(3,1,1,3)
17	C1	(26,13,1,5,3,1,2),(27,13,1,7,2,3,1), (7,3,3,12,1,2,3)	Nil	(5,2,3,~4,3,1,2)	(6,2,3,1)
18	Coo	(27,13,1,7,2,3,1),(7,3,3,12,1,2,3)	(26,13,1,5,3,1,2)	(5,2,3,~4,3,1,2)	Nil
19	C3	(27,13,1,7,2,3,1),(7,3,3,12,1,2,3)	(26,13,1,5,3,1,2)	(10,5,2,~6,3,2,1), (11,5,2,~2,1,3,2)	(~4,3,3,2)
20	C2	(27,13,1,7,2,3,1),(7,3,3,12,1,2,3)	Nil	(10,5,2,~6,3,2,1), (11,5,2,~2,1,3,2)	(5,1,3,2)
21	Coo	(27,13,1,7,2,3,1),(7,3,3,12,1,2,3)	(10,5,2,~6,3,2,1)	(11,5,2,~2,1,3,2)	Nil
22	C2	(27,13,1,7,2,3,1),(7,3,3,12,1,2,3)	(20,10,1,~7,3,1,2), (21,10,1,~5,2,3,1)	(11,5,2,~2,1,3,2)	(~6,2,3,1)
23	C3	(27,13,1,7,2,3,1),(7,3,3,12,1,2,3)	(20,10,1,~7,3,1,2), (21,10,1,~5,2,3,1)	(22,11,1,~3,1,2,3) , (23,11,1,~1,3,1,2)	(~2,2,1,2)
24	C2	(27,13,1,7,2,3,1),(7,3,3,12,1,2,3)	(21,10,1,~5,2,3,1)	(22,11,1,~3,1,2,3) , (23,11,1,~1,3,1,2)	(~7,1,3,2)
25	C3	(27,13,1,7,2,3,1),(7,3,3,12,1,2,3)	(21,10,1,~5,2,3,1)	(23,11,1,~1,3,1,2)	(~3,1,1,3)
26	C2	(27,13,1,7,2,3,1),(7,3,3,12,1,2,3)	Nil	(23,11,1,~1,3,1,2)	(~5,1,2,1)
27	Coo	(7,3,3,12,1,2,3)	(27,13,1,7,2,3,1)	(23,11,1,~1,3,1,2)	Nil
28	C1	(14,7,2,10,1,3,2),(15,7,2,14,2,1,3)	(27,13,1,7,2,3,1)	(23,11,1,~1,3,1,2)	(12,3,1,3)
29	C3	(14,7,2,10,1,3,2),(15,7,2,14,2,1,3)	(27,13,1,7,2,3,1)	Nil	(~1,1,3,2)

Maejo Int. J. Sci. Technol. **2012**, *6*(02), 282-296

Table 1. (Continued)

Seq.	Transition	Core-1 thread	Core-2 thread	Core-3 thread	Generated move
30	C2	(14,7,2,10,1,3,2),(15,7,2,14,2,1,3)	Nil	Nil	(7,1,2,1)
31	Coo	(15,7,2,14,2,1,3)	(14,7,2,10,1,3,2)	Nil	Nil
32	C1	(30,15,1,13,2,3,1), (31,15,1,15,1,2,3)	(14,7,2,10,1,3,2)	Nil	(14,2,2,3)
33	C2	(30,15,1,13,2,3,1), (31,15,1,15,1,2,3)	(28,14,1,9,1,2,3), (29,14,1,11,3,1,2)	Nil	(10,2,1,2)
34	Coo	(30,15,1,13,2,3,1), (31,15,1,15,1,2,3)	(29,14,1,11,3,1,2)	(28,14,1,9,1,2,3)	Nil
35	C2	(30,15,1,13,2,3,1), (31,15,1,15,1,2,3)	Nil	(28,14,1,9,1,2,3)	(11,1,3,2)
36	C3	(30,15,1,13,2,3,1), (31,15,1,15,1,2,3)	Nil	Nil	(9,1,1,3)
37	Coo	(31,15,1,15,1,2,3)	(30,15,1,13,2,3,1)	Nil	Nil
38	C2	(31,15,1,15,1,2,3)	Nil	Nil	(13,1,2,1)
39	C1	Nil	Nil	Nil	(15,1,1,3)

Now if we remove the indices from the moves, we get the following moves: (1,1,3),(2,1,2),(1,3,2),(3,1,3),(1,2,1),(2,2,3),(1,1,3),(4,1,2),(1,3,2),(2,3,1),(1,2,1),(3,3,2),(1,1,3),(2,1,2),(1,3,2),(5,1,3),(1,2,1),(2,2,3),(1,1,3),(3,2,1),(1,3,2),(2,3,1),(1,2,1),(4,2,3),(1,1,3),(2,1,2),(1,3,2),(3,1,3),(1,2,1),(2,2,3),(1,1,3). In the above list, each move is of the following structure: (Disk No, Source Pillar and Destination Pillar).

DISCUSSION

The model was executed several times. All the trials led to the same result, viz. the generation of the correct sequence of moves. However, different trials have different sequences of transition activations. This is due to the non-deterministic nature of the CPNs. Nevertheless, all the trials generate the same number and sequence of game moves.

The RSS strategy can be applied to any divide-and-conquer problem. The strategy is scalable; it can be easily expanded to deal with any number of cores. Moreover, it is an improvement on the In-Order strategy [8-9] with respect to the searching method. The In-Order strategy picks the first encountered victim regardless of the threads' richness. Although simple and direct, the In-Order strategy becomes inactive when it chooses a victim core that has only a few threads, which may happen frequently. On the other hand, the RSS strategy picks the wealthiest core to ensure that a maximum number of threads can be moved to the thief cores.

In Table 1 the simulator needs 39 steps to reach the final list of moves. Stealing happens in steps 4, 18, 21, 27, 31, 34 and 37. Steps 18 and 27 take advantage of the RSS strategy. The stealing in these steps happens based on a search for the richest core, which is preferred. In the other stealing steps (steps 4, 21, 31, 34 and 37) we can see that there is either only one victim core or the victim cores have the same degree of richness. The stealing and the distribution processes obviously consume some time. However, with the steady increase in the number of cores per chip currently taking place, this kind of sacrifice should be accepted in view of the great advantage that we can gain from ensuring that all the cores are as busy as possible.

Maejo Int. J. Sci. Technol. **2012**, *6*(02), 282-296

Concurrency is one of the main features in this study. The cores work concurrently in most of the steps in Table 1. From step 4 to step 39, many steps have the potential to occur at the same time with other steps. These concurrent steps are: (5 and 6), (8 and 9), (10 and 11), (14 and 15), (19 and 20), (22 and 23), (24 and 25), (28, 29, and 30), (32 and 33), (35 and 36), and (38 and 39). Any increase in the number of cores will definitely increase the number of steps that can occur concurrently. On the other hand, it was observed that Core 3 is less active than Core 1 and Core 2. Despite the fact that in every new run the simulation process starts with Core 1, there will be different sequences of active transitions. In other words, the continuous running of the model will generate different percentages of core activation. This is due to the non-deterministic nature of the CPNs, yet all the runs end with the same result.

The simplicity of the design is another feature of the model. It is so easy to add a new core to the model. New cores can be easily copied and pasted in the model. This is one of the advantages of CPN-Tool. In general, the tool can easily support the expansion of the model by replicating the sub-pages as long as the model is hierarchically well designed.

The language of coding, CPN-ML, was found to play a significant role in supporting the mechanism of the model. This is because firstly CPN-ML is free of side effects. This advantage eliminates the problems of side effects present in other studies that use imperative languages. Using CPN-ML makes the behaviour of the model much more understandable. Second, a huge number of built-in functions are included in this language. This makes it easier to deal with structures such as lists which we use to store the threads and the game moves. In addition, as a functional language derived from SML, CPN-ML uses linked lists when creating lists. Compared with other languages, the use of linked lists has two main advantages:

(1) In linked lists there are no problems such as overflow or managing array indices. Previous studies used fixed arrays when storing threads. This led to the possibility of having the overflow problem even when circular arrays were used to solve the overflow. However, in functional languages there are no such problems. The entire memory is under the control of the programmer. The only limitation is the size of the memory.

(2) A garbage collection mechanism is supported with the linked lists in functional language. Compared with other languages such as C and C++, modelling using CPN-ML releases the designer from any kind of memory management problems. This mechanism relieves the burden of writing additional codes to manage cells of memory that are no longer in use. Without such a mechanism there would be more overheads to the design specification.

CONCLUSIONS

In this paper, we present a new concurrent multithreaded model that extends the work-stealing scheduling technique by developing a new strategy that improves its performance. The RSS strategy provides a new means of balancing the threads among the modelled cores. It is based on balancing thread numbers in the model through moving threads from the wealthiest (victim) core to the non-working (thief) cores. The strategy has several beneficial features:

- Simplicity: Implementing the strategy is simple and does not need any complicated calculations.

Maejo Int. J. Sci. Technol. **2012**, *6*(02), 282-296

• Scalability: The mechanism of the strategy has been designed to be unlimited. The model can be easily expanded to deal with any number of cores.
• Generality: This strategy can deal with any kind of divide-and-conquer problems. Although in this paper we used this strategy to balance the threads of the moves in the Tower of Hanoi game, the strategy can deal with any divide-and-conquer problem.
• Concurrency: One of the main motivations behind the development of this strategy was to make all the cores work as much as possible. This strategy is superior to the previous one, the In-Order strategy, in the increase of the number of concurrent steps among the modelled cores.

FUTURE WORK

One of the future plans in this area is the development of more work-stealing strategies. It is envisaged that these future strategies should lead to better performance especially with regard to the balancing of the threads. In addition, compared to those strategies that currently exist, these new strategies should lead to a shorter run-time.

REFERENCES

1. S. Hofmeyr, C. Iancu and F. Blagojevic´, "Load balancing on speed", Proceedings of 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, **2010**, Bangalore, India, pp.147-158.
2. R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing", *J. ACM*, **1999**, *46*, 720-748.
3. N. S. Arora, R. D. Blumofe and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors", Proceedings of 10th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), **1998**, Puerto Vallarta, Mexico, pp.119-129.
4. D. Hendler, Y. Lev, M. Moir and N. Shavit, "A dynamic-sized nonblocking work stealing deque", *Distrib. Comput.*, **2005**, *18*, 189-207.
5. D. Hendler and N. Shavit, "Non-blocking steal-half work queues", Proceedings of 21st Annual Symposium on Principles of Distributed Computing, **2002**, Monterey, California, USA, pp.280-289.
6. U. A. Acar, G. E. Blelloch and R. D. Blumofe, "The data locality of work stealing", Proceedings of 12th Annual ACM Symposium on Parallel Algorithms and Architectures, **2000**, Bar Harbor, Maine, USA, pp.1-12.
7. D. Chase and Y. Lev, "Dynamic circular work-stealing deque", Proceedings of 17th Annual ACM Symposium on Parallelism Algorithms and Architectures, **2005**, Las Vegas, Nevada, USA, pp.21-28.
8. A. M. Al-Obaidi and S. P. Lee, "A concurrent multithreaded scheduling model for solving Fibonacci series on multicore architecture", *Int. J. Adv. Comput. Technol.*, **2011**, *3*, 24-37.
9. A. M. Al-Obaidi and S. P. Lee, "A concurrent coloured Petri nets model for solving binary search problem on a multicore architecture", Proceedings of 2nd International Conference on Software Engineering and Computer Systems, **2011**, University Malaysia Pahang, Malaysia, pp.463-477.

Maejo Int. J. Sci. Technol. **2012**, *6*(02), 282-296

10. K. Jensen and L. M. Kristensen, "Coloured Petri Nets: Modelling and Validation of Concurrent Systems", Springer, Dordrecht**, 2009**.
11. K. Jensen, "An introduction to the practical use of coloured Petri nets", in "Lectures on Petri Nets II: Applications, Advances in Petri Nets" (Ed. W. Reisig and G. Rozenberg), Springer-Verlag, London, **1998**, pp.237-292.
12. L. M. Kristensen and S. Christensen, "Implementing coloured Petri nets using a functional programming language", *Higher-Order Symbol. Comput.*, **2004**, *17*, 207-243.
13. N. A. Mulyar and W. M. P. van der Aalst, "Patterns in colored Petri nets" , Working Paper, **2005**, Eindhoven University of Technology, Eindhoven, Netherlands.
14. J. Reppy, "Standard ML of New Jersey (SML/NJ)", Bell Laboratories, New Jersey, USA, http://www.smlnj.org/ (Accessed: January 2012).
15. E. R. Gansner and J. H. Reppy, "The Standard ML Basis Library", University of Cambridge Press, Cambridge, **2002**.
16. J. D. Ullman, "Elements of ML Programming", Prentice-Hall, Upper Saddle River, **1998**.
17. CPN Group, "CPN Tools", Eindhoven University of Technology, Eindhoven, Netherlands, http://cpntools.org/ (Accessed: January 2012).
18. T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, "Introduction to Algorithms", MIT Press, Cambridge, **2001**.
19. F. P. Miller and A. F. Vandome, "Divide and Conquer Algorithm", Alphascript Publishing, Beau Bassin, **2010**.
20. K. M. Kavi, A. Moshtaghi and D. J. Chen, "Modeling multithreaded applications using Petri nets", *Int. J. Parallel Program.*, **2002**, *30*, 353-371.
21. J. L. Peterson, "Petri nets", *ACM Comput. Surv.*, **1977**, *9*, 223-252.
22. T. Murata, "Petri nets: Properties, analysis and applications", *Proc. IEEE*, **1989**, *77*, 541-580.