Abstract

The emergence of multicore technology has led to essential changes in the hardware design of personal computers. These changes are represented by an increased growth of cores per chip. As a matter of fact, this growth has imposed new directions in planning not only in the hardware but also in the software side. This study is about developing new algorithms that make a class of software (Divide and Conquer) more adaptable with multicore architecture. We propose one novel algorithm and one enhanced algorithm that prove their ability to perform concurrent multithreaded scheduling for solving Fibonacci series problem on a multicore environment. To put these algorithms to work, we have designed a hierarchical model that is controlled by these algorithms. The designed model has been simulated successfully through a modelling tool. The modelled design solved several shortcomings in the previous studies. Furthermore, it provides a new and efficient way in load-distribution other than incurring a slight overhead to handle sorting from time to time.

Keywords: Divide and Conquer, Multithreading Scheduling, Work Stealing, Concurrency, Fibonacci.

1. Introduction

Divide and Conquer (D&C) problems [1] [2] represent a group of problems that can be solved by partitioning a main problem into sub problems in a recursive manner. Starting from the top level downward, the process of partitioning continues until it reaches to the leaf-level in which a solution can be obtained directly. Partial results can then be recursively combined to ultimately formulate the result of the main problem. Fibonacci series [3] is an example of D&C problems that we focus in this paper. The series is given as: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, etc. The first two terms are 0 and 1, the other terms can be calculated as: Tn = Tn-1 + Tn-2 , where n ≥ 2 and Tn is the nth term.

Prior to multicore technology, such algorithm can be solved serially step by step. As the number of cores per chip in personal computers increases, software engineers rely more on multithreading in leveraging the efficiency of executing programs by using concurrency [4] [5] [6]. Modelling concurrent multithreaded systems represents a great challenge due to the nondeterministic nature of such systems besides the difficulty in synchronizing the threads. There is a strong relation between concurrent systems, multithreading as well as D&C problems. It is apparent that the partitioning process along with joining partial results can be done concurrently. Assigning each partition to a thread will make more than one thread working at the same time. In the environment of multicore, each thread can be assigned to a core; eventually we get a concurrent system. The process of mapping D&C problems from a single processor environment to a multicore environment puts some burden of efforts on the software designers. They have to develop new algorithms which help them in keeping all the cores working at the same time as much as possible. The solution is in finding a suitable scheduling algorithm that can cope with the new technology.

In this paper, first, we present a new algorithm, Fibonacci Multithreaded Scheduling (FMS), which is able to schedule threads inside a modelled core to calculate a Fibonacci series. The FMS algorithm is basically built on D&C technique in partitioning modelled tokens (threads). Second, we present a new algorithm, Enhanced Work Stealing Scheduling, (EWSS) algorithm which works as a coordinator...
between the modelled cores. The EWSS algorithm controls the movements and threads’ distribution among the cores and it is responsible for making all the cores working concurrently. We claim that the new modelled cores along with the new enhancement achieve high concurrency rates through the simulation process that we have conducted. We applied these algorithms by using Coloured Petri Net (CPN) [7] [8] [9] [10] as a language of modelling, Coloured Petri Nets Meta Language (CPN-ML) [11] as a language of coding and CPN-Tool [12] as a software tool which enables us to create, simulate and verify the correctness of the designed models.

The rest of this article is organized as follows: Section 2 represents a background of Work-Stealing scheduling algorithm besides some information about Coloured Petri Nets (CPN), CPN-Tool and CPN-ML. Section 3 is dedicated for the problem statement. In Section 4, we demonstrate the research methodology that we propose. The proposed concurrent multithreaded scheduling model for solving Fibonacci series is explained in Section 5. Section 6 is dedicated for building the CPN model. The results of simulation are included in Section 7. Sections 8 and 9 are devoted for the discussion and conclusion. Future work is included in Section 10. Finally, an appendix which comprises all the CPN-ML code has been added at the end.

2. Background

There are two types of scheduling algorithms in a multicore environment: Work Sharing and Work Stealing. In Work Sharing, the scheduler continuously attempts to transfer threads from the core that has generated them to other cores hopefully in a fair load distributing among the cores. In Work Stealing, on the other hand, the idle cores attempt to steal threads from the working cores. Movement of threads between cores happens less frequently in Work Stealing than in Work Sharing. In Work Stealing, each core is accompanied by a local memory which is organized as a deque. A core can push threads into the deque from one end and pop threads from the other end. When a core becomes idle (out of threads), it tries to steal thread(s) from another deque using the other end of that deque. In general, each core has two statuses: either it is a thief (trying to steal threads from other cores) or a victim (other cores trying to steal from it).

The work of Blumofe and Leiserson [13] can be considered as a landmark in work stealing scheduling. They presented an algorithm that is able to schedule fully-strict (well-structured) multithreaded computations. Although the algorithm did well in the area that needs static space partition, it faces problems in the modern environments that support multiprogramming. The reason behind this drawback is the supposition of the availability of a fixed set of processors to achieve computation. Another important achievement of Work Stealing is the contribution of Arora et al [14]. They designed a thread scheduler for a shared-memory multiprocessors environment. They improved the work of Blumofe and Leiserson [13] by considering non fully-strict multithreaded computations in addition they dealt with a multiprogrammed environment instead of a dedicated one. However the huge success it gained as best choice algorithm for load balance both in academic and industrial fields, the algorithm of Arora et al faced two problems: it introduced memory management problem; having n processes with m as total memory size, the algorithm can deal with m/n threads in the deque at the most. In addition, overflows can easily occur; this means that the size of the array (deque) must be adjusted to solve the overflow problem. There is no simple way to free additional memory and continue. An expensive overflow mechanism must be added to fix this situation [15].

The work of Arora et al [14] has been extended by several ways: Hendler and Shavit presented in [16] the idea of stealing the half. In their algorithm, the process can steal up to the half number of items in the deque. The data locality of Work Stealing has been studied by Acar et al [17]; they presented 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. A major development on the algorithm of Arora et al has been proposed by Hendler et al in [15]; their algorithm detects synchronization conflicts by pointer-crossing instead of gaps between indexes as in the algorithm of Arora et al. The algorithm builds non-blocking dynamic-sized work stealing deques. It eliminates the need for any kind of application for fixing the overflow problem as in the algorithm of Arora et al. However, since lists represent the main structure in this algorithm, there is a kind of trade-off between space and time complexity for the reason that the work needed to maintain lists. A simple lock-free work stealing
deque has been presented by Chase and Lev [18]. They used a cyclic array to store the elements. The algorithm can easily deal with overflow due to the cyclic nature of its data structure. There are no limits in this algorithm other than memory and integer sizes. The algorithm is simple and does not need garbage collector.

The principle of Work Stealing has been implemented in several programming languages. JAWS (Java Work Stealer) has been presented in [19]. It allows programmers to write parallel programs in pure Java that can run on a network of workstations. JAWS has been implemented as a user-level Java library which schedules user threads over a network of workstations using a Work Stealer algorithm. However, due to certain overheads such as lower communication cost, JAWS could not achieve optimal performance. Satin [20] represents a system for running programs on distributed memory systems. It extends Java with several primitives for D&C programming using Work Stealing to distribute the jobs. Jcluster [21] is another Java based system that provides a parallel environment that is suitable for a large-scale heterogeneous cluster. It implements a task scheduler based on a Transitive Random Stealing algorithm. The proposed scheduler can be seen as an improvement to work stealing algorithm. Java language [22] has developed its java.util.concurrent packages in its 7th release by adding a framework for fork-join style parallel decomposition. The new framework provides a natural means for partitioning many algorithms to efficiently make use of hardware parallelism. The use of Work Stealing has reduced the contention for the working deques.

Coloured Petri Nets [7] [8] [9] [10] is a graphical discrete-event language designed to model and validate concurrent systems. In addition to concurrency, communication and synchronization between the elements of the nets have a significant role in controlling the execution of the model. CPN has been developed from Petri Nets [23] [24] [25] as being the origin of CPN. The main difference between Petri Nets and CPN is the addition of types to CPN besides the ability to write expressions and functions written in Standard Meta Language (SML) [26]. CPN model is an executable model in the sense that the process of execution shows different states of the system that is represented by the model.

CPN-Tool [12] is a software tool developed by Kurt Jensen [7] [8] [27] at University of Arhus (Denmark). The tool provides all the necessary facilities to create, simulate and validate Coloured Petri Nets. CPN-Tool is a GUI tool that provides all the interaction methods such as menus and toolbars besides giving feedback messages when errors have been encountered during the process of code’s syntax checking. CPN-Tool uses Coloured Petri Nets Met a Language (CPN-ML) as a language of writing declarations, expressions and codes. CPN-ML [11] is a language for writing nets inscriptions which includes expressions on the arcs, codes that control transitions as well as the declarations of the types and variables that are included in the net. It has been built based on SML [26] [28] [29].

3. Problem statement

The emergence of multicore technology opens a new path in the process of algorithms design. Generally, the growing number of cores imposes a noticeable change in the design phase. New enhancements have been suggested to make the algorithms suitably fitting the new architecture; D&C algorithm has its share in these developments. The lack of investing the growing number of cores in enhancing these algorithms will make no obvious differences when applying these algorithms on a multicore environment. A key factor to these enhancements is through designing a multithreaded scheduler that makes all the cores working concurrently. The adoption of a modelling language that provides the specifications of such schedulers is a major step towards applying these schedulers in real life. In addition, the simulation of the generated models will no doubt catch many of design bugs and ensure the stability of the design. This study aims to design a concurrent multithreaded scheduler model for D&C problems (Fibonacci Series, Matrix Multiplications, etc) that can achieve high expectations if applied on a multicore environment. As a case study, we present a concurrent multithreaded scheduler model for calculating Fibonacci sequence.

4. Research methodology

The general research plan that we propose for building a concurrent multithreaded scheduling model for Divide and Conquer problems consists of three phases: High-Level Scheduling Phase, Core
Scheduling Phase and Threads Computing Phase. Fig. 1 shows these phases and the relation between them.

4.1. High-level scheduling phase

High-Level Scheduling Phase has the duty of controlling threads redistribution among the cores. The process of redistribution can be achieved through stealing the threads from a certain set of cores, called the victim cores, to another set of cores called the thief cores. Victim cores are those cores that have one or more threads while thief cores are those who have no threads. We have developed new strategies to enhance the way of stealing (moving). In this paper, we propose a new strategy, Single-Step Stealing Strategy. This strategy simply can be applied as follows: When one or more cores are idle and there is a chance to allow them to steal, then the victim core(s) permit all the thief cores to steal in one step. The stealing process happens in a clock wise. Assuming that all the cores are organized on a single line, each thief core will check the non idle (victim) core to its right, then the other next core, etc. In case only one victim is available, this victim core will be the target for the thief core. When all the cores are busy or idle, nothing happens. The Single-Step Stealing Strategy has been applied through a function written in CPN-ML which receives, as inputs, list of threads that belong to the modelled cores. The output of the function will be the updating of these lists. A major achievement in this function is the ability to redistribute the threads in the entire cores in one step, further, the function works concurrently with the other activities of the cores. In other words, while the High-Level Scheduling Phase balances the threads among the cores, each core can execute its own Core Scheduling Phase.

![Figure 1](image.png)

Figure 1. The General Research Plan

4.2. Core scheduling phase

Core Scheduling Phase is responsible for scheduling operations inside each core besides facilitating the communication with the High-Level Scheduling Phase. A major significant in the Core Scheduling Phase is that all the cores work concurrently in scheduling their internal activities. The phase is in charge of performing three tasks: Threads creation, computing leaf-level threads and import / export threads. In fact, the process of threads division is done in this phase. The division process includes the creation of two threads; Left and right. The left one has the same number as of its direct ancestor thread number multiplied by two. The right child holds the left child number plus one. A local memory organized as a stack (represented as a list of threads) is used to temporarily store the divided threads. The mechanism of this phase has been designed to create and PUSH/POP threads into the stack. In addition, the phase calculates leaf-level threads (leaf-level threads are those threads that are designed to
A Concurrent Multithreaded Scheduling Model for Solving Fibonacci Series on Multicore Architecture
Alaa M. Al-Obaidi, Sai Peck Lee

be calculated directly) then send them to the Threads Computing Phase. Finally, the Core Scheduling Phase cooperates with the High-Level Scheduling Phase in importing/exporting threads. Threads can be easily taken (stolen)/added from/to the stacks and redistributed to other cores resulting in a fair load distribution. The modelling tool, CPN-Tool, supports such cooperation through using Fusion Sets [12].

4.3. Threads computing phase

Threads Computing Phase has the duty of calculating and joining two sibling threads into one thread. This phase receive its inputs from the Core Scheduling Phase. In fact, the process of conquering is done in this phase through recursive joining of the sibling threads. The process of conquering continues till reaching the main thread which will hold the final result. The mechanism of this phase is controlled by a recursive technique which has different approaches depending on the kind of the problem.

5. A Proposed concurrent multithreaded scheduling model for solving Fibonacci series

In this study, we propose a concurrent model that is able to schedule Fibonacci series as multithreading scheduling. We can visualize the model as the one in Fig. 2. The model consists of three sub models namely: Coordinator Model, Core Model, and Ready Threads Model.

![Figure 2. The Fibonacci Concurrent Multithreaded Scheduling Model](image)

5.1. Fibonacci coordinator model

The Fibonacci Coordinator Model represents the High-Level Scheduling Phase; it works under the control of a new algorithm, Enhanced Work Stealing Scheduling (EWSS), which represents the new development that we have achieved on work stealing algorithm. This model is responsible for coordinating threads moving between the cores in order to achieve load balance. The function EWSS as listed in the Appendix receives three parameters. Each parameter stands for a list of threads that belong to one core. The function EWSS redistributes the threads by moving threads from the victim cores to the thief cores. The Single-Step Stealing Strategy has been applied in this function. The EWSS function is built on the base of Pattern Matching technique which has the ability to redistribute the threads in one move (step). The statuses in bold are the only statuses which include threads redistribution. The rest of the statuses do not include redistribution either because there is no need or there is no possibility to steal (no thief cores).

5.2. Fibonacci core model

The Core Scheduling Phase is represented in the core model. The core model is controlled by a new algorithm; Fibonacci Multithreaded Scheduling (FMS) that achieves threads creation, computing leaf-level threads and the movement of the leaf-level threads to the Ready Threads Model (Threads Computing Phase). Every core schedules its activities by executing its own copy of FMS function as listed in the Appendix resulting in a concurrent working of the entire cores. The thread is modelled as a
4-tuple: (ThreadId, FatherId, Value, ReturnValue), where the first two parameters represent the thread’s number and the thread’s father number. ThreadId and FatherId are denoted as “Tx” (x is a positive integer number). The third parameter, Value, holds the argument “n”, as Fibonacci (n) = Fibonacci (n-1) + Fibonacci (n-2). The last parameter, ReturnValue, carries the result of the function.

The FMS function receives two parameters: StackList and Ready Thread. The first parameter represents the list of threads (stack of threads). The second parameter holds the list of computed threads (leaf-level threads). The function uses a one let-in-end construct in its body. The function pops the thread at the top of the stack then it does one of the following: if the popped thread is a leaf-level thread then it computes the return value (Fibonacci (0) \leftarrow 1, Fibonacci (1) \leftarrow 1) after that it sends the computed thread to the Ready Threads Model. If the popped thread is not a leaf-level thread then it creates the children, finally it pushes the two new children threads again in the stack. The result of this function is the updating of the same two parameters. The Get_Left_Child and Get_Right_Child functions are used to create the left and right child. The difference between these two functions is that the first one calculates the left child number through multiplying ThreadId by two while the second function (Get_Right_Child) has the same number as the left child plus one. The CPN-ML codes of these two functions are given in the Appendix.

5.3. Fibonacci ready threads model

Threads Computing Phase is represented here. Ready Threads Model is a common model which receives the computed leaf-level threads from the modelled cores. It reduced the received leaf-level threads into one thread, that is, the main thread which holds the final result. The model works concurrently with the Coordinator and Cores models. The mechanism of this model is controlled by an algorithm called Reduction. The Reduction algorithm is represented as a function which has two responsibilities: Sorting the threads and then reducing them according to ThreadId. The process of sorting is done through the function Merge Sort listed in the Appendix, while the process of reduction is done through the function ROT (Reduced Ordered Threads). The ROT function recursively joins two successive (sibling) threads into one thread. The process of joining includes the creating of the direct descendent of any two sibling threads along with its parameters. Eventually, the ROT function ended its job with a single thread; that is the main thread which has the final result.

6. Building the CPN Fibonacci model elements

The main page for the CPN Fibonacci Model is shown in Fig. 3. This page includes three places (Stack_List_1, Stack_List_2, and Stack_List_3), 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 (stack) of threads. The initial value of the place is located at the top of each place. Place Stack_List_1 contains a list with a single thread, ("T1","T0",5,0) as an example, where “T1” and “T0” symbolize ThreadId and Thread’s father Id respectively. The number 5 represents thread’s parameter (as the value n in Fibonacci (n)). The zero stands for the returned result. At the end of the simulation process, this zero will hold the final result of Fibonacci (5). The initial value of Core 2 and Core 3 is a nil list ([]) since the execution starts with one main thread located in Core 1. Each core has also a current value. This value will be changed during the execution of the model.

In CPN, transitions represent the action units. Transition Coordinator is in charge of executing the High-Level Scheduling Phase (EWSS function). The transition reads the lists of the cores through (Core1 In, Core2 In, and Core3 In), update the lists, and then send the feedback as Core1 Out, Core2 Out, and Core3 Out. To activate the transition Coordinator, the Checking guard function as listed in the Appendix should return Boolean True value. The function returns true only if there is at least one victim and at least one thief. In Fig. 3, the code below the Checking function represents the code that will be carried out when transition Coordinator is executed by the CPN-Tool’s simulator. As illustrated in the figure, the transition executes the EWSS function which results in a fair distribution of the threads. To the right of each place there is a substituted transition: Core1, Core2, and Core3. Each substituted transition corresponds to a page similar to the one in Fig. 4 which represents Core1.
The page in Fig. 4 shows the content of Core 1. The core consists of two places (Stack_List_1 and Ready Threads) and one transition (Calculate). Although the two places have different types’ names (stack and ThrList), however, they represent one type, that is, list of threads (refer to the Appendix). The I/O symbol at the lower left corner of the place Stack_List_1 represents a port tag which is a kind of mechanism offered by the CPN-Tool to connect places from different pages. It allows the Coordinator to add/take threads to/from the place. This means that the place Stack_List_1 in Fig. 3 is just a copy of the place Stack_List_1 in Fig. 4. This kind of hierarchic simplifies the communication between the pages’ elements in the model. Transition Calculate reads the stack’s list and the ready threads list through Stack1_Out and RT1_Out respectively. The transition updates these two lists through FMS function and it returns the updated lists through Stack1_In and RT1_In. The Fusion 1 tag at the lower left corner of place Ready Threads symbolizes that this place (and the other Ready Threads places in Core 2 and Core 3) works as a sink for the place Ready Threads in the Ready Threads page as shown in fig. 5. That is all ready (leaf level) threads generated by the cores are sent to the Ready Threads place.
The Ready Threads page (Fig. 5) consists of one place (Ready Threads) and one transition (Reduce). The Reduce transition is supported with a guard represented as a Boolean expression (length Thread_List_Out > 1). The guard activates the transition only if there is more than one thread in the Ready Threads List. Executing the Reduce transition will run the Reduction function which includes two functions: Merge Sort and ROT (Reduce Ordered Threads). The first function sorts the list of threads in ascending according to the ThreadId and the second function recursively reduce the threads into one thread which holds the final result.

7. The results of simulation

CPN-Tool is a single thread tool, it chooses transitions randomly. At every stage, the tool checks the active transitions (those who have enough tokens in their input transitions to enable their transition, beside that the guards, if exist, return true). The following represents an example of a simulation process that we have achieved on the model. The main thread is [(T1,T0,5,0)] located in Core 1. We want to calculate Fibonacci (5). Initially all the places (cores) except Stack_List_1 are empty (status 1, Table 1).
Table 1. The Results of Simulation

<table>
<thead>
<tr>
<th>No</th>
<th>Selected Transition</th>
<th>Stack_List_1 Core1</th>
<th>Stack_List_2 Core2</th>
<th>Stack_List_3 Core3</th>
<th>Ready Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Calculate Core1</td>
<td>[(T1,T0,5,0)]</td>
<td>Nil</td>
<td>Nil</td>
<td>Nil</td>
</tr>
<tr>
<td>2</td>
<td>Calculate Core1</td>
<td>[(T2,T1,4,0), (T3,T1,3,0)]</td>
<td>Nil</td>
<td>Nil</td>
<td>Nil</td>
</tr>
<tr>
<td>3</td>
<td>Calculate Core1</td>
<td>[(T4,T2,3,0), (T5,T2,2,0), (T3,T1,3,0)]</td>
<td>Nil</td>
<td>Nil</td>
<td>Nil</td>
</tr>
<tr>
<td>4</td>
<td>Coordinator</td>
<td>[(T3,T1,3,0)]</td>
<td>[(T4,T2,3,0)]</td>
<td>[(T5,T2,2,0)]</td>
<td>Nil</td>
</tr>
<tr>
<td>5</td>
<td>Calculate Core1</td>
<td>[(T6,T3,2,0), (T7,T3,1,0)]</td>
<td>[(T4,T2,3,0)]</td>
<td>[(T5,T2,2,0)]</td>
<td>Nil</td>
</tr>
<tr>
<td>6</td>
<td>Calculate Core1</td>
<td>[(T12,T6,1,0), (T13,T6,0,0), (T7,T3,1,0)]</td>
<td>[(T4,T2,3,0)]</td>
<td>[(T5,T2,2,0)]</td>
<td>Nil</td>
</tr>
<tr>
<td>7</td>
<td>Calculate Core5</td>
<td>[(T12,T6,1,0), (T13,T6,0,0), (T7,T3,1,0)]</td>
<td>[(T4,T2,3,0)]</td>
<td>[(T10,T5,1,0), (T11,T5,0,0)]</td>
<td>Nil</td>
</tr>
<tr>
<td>8</td>
<td>Calculate Core2</td>
<td>[(T12,T6,1,0), (T13,T6,0,0), (T7,T3,1,0)]</td>
<td>[(T4,T2,3,0)]</td>
<td>[(T10,T5,1,0), (T11,T5,0,0)]</td>
<td>Nil</td>
</tr>
<tr>
<td>9</td>
<td>Calculate Core1</td>
<td>[(T13,T6,0,0), (T7,T3,1,0)]</td>
<td>[(T8,T4,2,0), (T9,T4,1,0)]</td>
<td>[(T10,T5,1,0), (T11,T5,0,0)]</td>
<td>[(T12,T6,1,1)] The fourth parameter has been changed to 1 since Fibonacci(1) is 1</td>
</tr>
<tr>
<td>10</td>
<td>Calculate Core1</td>
<td>[(T7,T3,1,0)]</td>
<td>[(T8,T4,2,0), (T9,T4,1,0)]</td>
<td>[(T10,T5,1,0), (T11,T5,0,0)]</td>
<td>[(T12,T6,0,0), (T12,T6,1,1)]</td>
</tr>
<tr>
<td>11</td>
<td>Ready Threads</td>
<td>[(T7,T3,1,0)]</td>
<td>[(T8,T4,2,0), (T9,T4,1,0)]</td>
<td>[(T10,T5,1,0), (T11,T5,0,0)]</td>
<td>[(T6,T3,2,1)]</td>
</tr>
<tr>
<td>12</td>
<td>Calculate Core3</td>
<td>[(T7,T3,1,0)]</td>
<td>[(T8,T4,2,0), (T9,T4,1,0)]</td>
<td>[(T11,T5,0,0)]</td>
<td>[(T10,T5,1,0), (T6,T3,2,1)]</td>
</tr>
<tr>
<td>13</td>
<td>Ready Threads</td>
<td>[(T7,T3,1,0)]</td>
<td>[(T8,T4,2,0), (T9,T4,1,0)]</td>
<td>[(T11,T5,0,0)]</td>
<td>[(T6,T3,2,1), (T10,T5,1,1)]</td>
</tr>
<tr>
<td>14</td>
<td>Calculate Core1</td>
<td>Nil</td>
<td>[(T9,T4,1,0)]</td>
<td>[(T11,T5,0,0)]</td>
<td>[(T7,T3,1,1), (T6,T3,2,1), (T10,T5,1,1)]</td>
</tr>
<tr>
<td>15</td>
<td>Coordinator</td>
<td>[(T8,T4,2,0)]</td>
<td>[(T9,T4,1,0)]</td>
<td>[(T11,T5,0,0)]</td>
<td>[(T7,T3,1,1), (T6,T3,2,1), (T10,T5,1,1)]</td>
</tr>
<tr>
<td>16</td>
<td>Calculate Core1</td>
<td>[(T16,T8,1,0), (T17,T8,0,0)]</td>
<td>[(T9,T4,1,0)]</td>
<td>[(T11,T5,0,0)]</td>
<td>[(T7,T3,1,1), (T6,T3,2,1), (T10,T5,1,1)]</td>
</tr>
<tr>
<td>17</td>
<td>Calculate Core1</td>
<td>[(T17,T8,0,0)]</td>
<td>[(T9,T4,1,0)]</td>
<td>[(T11,T5,0,0)]</td>
<td>[(T16,T8,1,1), (T7,T3,1,1), (T6,T3,2,1), (T10,T5,1,1)]</td>
</tr>
<tr>
<td>18</td>
<td>Calculate Core2</td>
<td>[(T17,T8,0,0)]</td>
<td>Nil</td>
<td>[(T11,T5,0,0)]</td>
<td>[(T9,T4,1,1), (T16,T8,1,1), (T7,T3,1,1), (T6,T3,2,1), (T10,T5,1,1)]</td>
</tr>
<tr>
<td>19</td>
<td>Ready Threads</td>
<td>[(T17,T8,0,0)]</td>
<td>Nil</td>
<td>[(T11,T5,0,0)]</td>
<td>[(T3,T1,3,2), (T9,T4,1,1), (T10,T5,1,1), (T16,T8,1,1)]</td>
</tr>
<tr>
<td>20</td>
<td>Calculate Core1</td>
<td>Nil</td>
<td>[(T11,T5,0,0)]</td>
<td>[(T11,T5,0,0), (T17,T8,0,0), (T3,T1,3,2)]</td>
<td>[(T17,T8,0,0), (T3,T1,3,2), (T9,T4,1,1), (T10,T5,1,1), (T16,T8,1,1)]</td>
</tr>
<tr>
<td>21</td>
<td>Calculate Core3</td>
<td>Nil</td>
<td>Nil</td>
<td>[(T11,T5,0,0), (T17,T8,0,0), (T3,T1,3,2)]</td>
<td>[(T9,T4,1,1), (T10,T5,1,1), (T16,T8,1,1)]</td>
</tr>
<tr>
<td>22</td>
<td>Ready Threads</td>
<td>Nil</td>
<td>Nil</td>
<td>[(T3,T1,3,2), (T9,T4,1,1), (T5,T2,2,1), (T16,T8,1,1), (T17,T8,0,0)]</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Ready Threads</td>
<td>Nil</td>
<td>Nil</td>
<td>[(T3,T1,3,2), (T9,T4,1,1), (T5,T2,2,1), (T16,T8,1,1), (T17,T8,0,0)]</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Ready Threads</td>
<td>Nil</td>
<td>Nil</td>
<td>[(T3,T1,3,2), (T9,T4,1,1), (T5,T2,2,1), (T14,T2,3,2)]</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Ready Threads</td>
<td>Nil</td>
<td>Nil</td>
<td>[(T3,T1,3,2), (T2,T1,4,3)]</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>Ready Threads</td>
<td>Nil</td>
<td>Nil</td>
<td>[(T1,T0,5,5)]</td>
<td></td>
</tr>
</tbody>
</table>
8. Discussion

We have simulated the model several times. Each time we started a new simulation, we got different series of executed transitions. However, all these simulations led to the same final result (status 26). The results of simulation show consistency towards reaching the ending thread. No errors have been detected nor any kind of ambiguity appeared during the execution time. Although CPN-Tool is a single thread tool in the sense that the tool has to move from one transition to another, the concurrent execution of the model appeared clearly in the results. The concurrent execution happened in the status 4-20. We can notice in any one of these statuses that we have more than one active core. Generally, in each of the status 4-20, we have more than one transition that can be executed concurrently.

Design simplicity and scalability are other significant achievements in this study since it is easy to expand the model by adding new cores. Only a slight change may be needed to the Enhanced Work Stealing code to include the new cores.

CPN-ML as a functional language uses lists as their main data structure. The language is supported by a huge number of list’s built-in functions that process the list and return results in a short time. In addition, CPN-ML is free of side effects as in imperative languages. Eliminating side effects makes the behaviour of a program much understandable and predictable.

The internal structure of the stack’s list and thread’s list proves its robustness in processing the threads. CPN-ML, as a functional language derived from SML, uses linked list in creating lists. Using linked lists has several advantages, as in the following, comparing with other data structures such as arrays.

First, using fixed array suffers from the problem of overflow as in previous studies even so when circular arrays have been used to solve this problem. Managing the indices and sometimes copying data from one region to another to spare more space causes a lot of overheads. However, in functional languages, there are no such problems since the entire space is under the control of the language except when the total required space exceeds the size of the memory.

Second, memory management in functional languages is much easier than in other languages like C and C++. Modelling using CPN-ML releases the designer from any kind of memory management problems. The mechanism of garbage collector is responsible for returning areas of memory that are no longer in use. The lack of such mechanism in languages such as C and C++ adds more overhead since codes have to be included to reorganize the memory.

In our study, we used stacks as local memories comparing with using queues or deques as in other studies. We found that using a stack has several useful characteristics: First it is much easier to deal with one end instead of two (less code). Second, for reasons related to security and management, stacks are preferable than queues. Finally, as we used CPN-ML as a language of modelling, representing a stack as a list enables us to use all the lists’ built-in functions.

The Enhanced Work Stealing algorithm plays the main role in balancing threads’ load among the cores. The algorithm has a privilege comparing with other Work Stealing techniques which is the ability to make more than one core steal at the same time (one step). In case we add more cores, the Enhanced Work Stealing has to be expanded slightly to deal with the new expansion. It is likely to have more than one idle core and more than one non-empty core. The pattern matching technique allows the redistribution of loads (threads) among the cores to be performed in one step in a fair manner. The Reduction function consumes some time in getting the results since each time the simulator executes the Reduction function, threads must be sorted first. As we have designed the Reduce Ordered Threads algorithm to work on ordered lists, improving the Reduce Ordered Threads algorithm to work on non-sorted lists will greatly reduce the time needed for executing Reduction function.

9. Conclusion

This paper presents a new model for solving one of the Divide and Conquer problems, i.e. Fibonacci series. The model has been designed to be adaptable to a multicore environment through concurrent multithreaded scheduling. A new algorithm in addition to one enhanced algorithm has been designed to support the work of the designed model. Fibonacci Multithreaded Scheduling (FMS) algorithm is
A Concurrent Multithreaded Scheduling Model for Solving Fibonacci Series on Multicore Architecture
Alaa M. Al-Obaidi, Sai Peck Lee

designed to control the creation and movement of threads within each modelled core. FMS function is responsible for the dividing part of the Divide and Conquer (D&C) general strategy. Reducing Ordered Thread (ROT) algorithm is designed to reduce the resulted threads to one final thread that holds the ultimate result. The ROT function is in charge of the conquer part of the D&C general strategy. Finally, the Enhanced Work Stealing, EWSS, is designed to control threads distribution among the modelled cores. The EWSS function is responsible for the concurrent part in this model.

Scalability, accuracy, simplicity, and fair distribution of load are the main characteristics of the model. The simulation process of the model proved its correctness and stability. The results of simulation show effective concurrent execution of the model. In many cases, more than one modelled core can be executed at the same time.

In this study, we succeeded in solving several shortcomings happened in the previous studies such as limitation of the number of threads, overflow, side effects, language complexity, indices and, memory management. Concurrent execution and fair distribution of threads among the cores were the main targets in this research. However, executing the sorting routine adds some time to the total time of simulation. In addition, our policy in controlling threads movement does not take into consideration the richness of the cores when we want to redistribute the load.

The general idea behind this model opens the door for designing new models that solve different Divide and Conquer problems. A key factor in these new models is the concurrent multithreaded scheduling that proves its effectiveness in dealing with a multicore environment.

10. Future work

This study can be expanded by solving different Divide and Conquer problems. In addition, the Work Stealing technique can be developed through adopting new strategies in distributing threads among the cores.

11. Acknowledgement

We would like to thank Mr. Michael Westergaard from University of Arhus, who is responsible for managing cpntools-support mailing list. He helped us in clarifying certain issues related with using CPN-Tool.

colset INT = int; colset STRING = string;
colset ThreadId=STRING; colset FatherId=ThreadId;
colset Value= INT; colset ReturnValue = INT;
colset Thread = product ThreadId * FatherId * Value * ReturnValue;
colset Stack= list Thread;
colset ThrList= Stack;

var Stack1_Out,Stack1_In, Stack2_Out,Stack2_In,
Stack3_Out,Stack3_In, Core1_Out, Core2_Out,
Core3_Out,Core1_In, Core2_In, Core3_In:
Stack;

var   RT1_Out,RT1_In, RT2_Out,RT2_In,
RT3_Out,RT3_In, Thread_List_In,
Thread_List_Out: ThrList;

fun EWSS([],[],[])=([],[],[])
|EWSS([],[],a::nil)=([],[],a::nil)
|EWSS([],a::nil,[])=(a::nil,[],[])
|EWSS([],a::nil,b::nil)=(a::nil,b::nil,[])
|EWSS([],a::nil,c::(d::e))=(a::nil,c::nil,d::e)

fun Get_Thread_Id(st)=valOf (Int.fromString (String.extract (st,1,NONE)));

fun Get_Father((x1:STRING,x2:STRING,x3:INT,x4:INT),
(y1:STRING,y2:STRING,y3:INT,y4:INT))=
("T"^Int.toString(Get_Thread_Id(x1)) div 2),
"T"^Int.toString(Get_Thread_Id(x1) div 4),x3+1,x4+y4);

fun Father_Token(x1:STRING,x2:STRING,
x3:INT,x4:INT)=Get_Thread_Id(x2);
fun Thread_Token(x1:STRING,x2:STRING,
x3:INT,x4:INT)=Get_Thread_Id(x1);

fun ROT ([])= []
|ROT (x::nil)=[x]

fun Checking (a,b,c)  =
if (length a =0 orelse length b=0 orelse length c=0)
andalso   (length a >1 orelse length b>1 orelse
length c>1) then true else false;

fun Get_Left_Child (p1)= "T"^Int.toString (valOf
(Int.fromString (String.extract  (p1,1 NONE)) )*2);

fun Get_Right_Child (p1)= "T"^Int.toString (valOf
(Int.fromString(String.extract (p1, 1, NONE)))*2 +
1);

fun Get_Thread_Id(st)=valOf (Int.fromString
(String.extract (st,1,NONE)));

fun FMS (StackList,ReadyThread) = let
val Head = hd StackList;
val Par1= #1 Head; val Par2= #2 Head;
val Par3= #3 Head;  val Par4= #4 Head;
val New_Ready_Thread=if (Par3 < 2) then
[(Par1,Par2,Par3,Par3)] else empty;
val Left_Child=if Par3 > 1 then 
[(Get_Left_Child (Par1),Par1,Par3 - 1,0)] else empty;
val Right_Child= if Par3 > 1 then
[(Get_Right_Child (Par1), Par1, Par3 - 2,0)] else
empty;
val Updated_Stack_List= if Par3 > 1 then
Left_Child^^Right_Child^^tl StackList  else empty;
val New_Stack_List= if Par3 > 1 then
Updated_Stack_List  else tl StackList;
in (New_Stack_List, New_Ready_Thread  ^^
ReadyThread) end;

13. References


