International Journal of Computer Sciences and Engineering Open Access
Research Paper Volume-3, Issue-1 E-ISSN: 2347-2693

A Deterministic Parallel Computing Approach Optimised for Multicore Architecture

Shahid Iqbal
www.ijcaonline.org

Received: 26/Dec/2014 Revised: Jan/8/2015 Accepted: Jan/20/2015 Published: Jan/31/2015

Abstract --- In software computing, computation is done either deterministic or non-deterministic approach. Deterministic approach includes the constraint like dependency of data in which no random computation is involved. This paper talks about how to achieve more parallelism in context of a deterministic computation approach for the dual-core architecture. In this paper a new Scheduling algorithm which termed as "LA Scheduling" algorithm and its associated component has been presented which is mainly optimised for dual core architecture. Simulation result shows that it helps in reducing the response time of a program and average speedup has been increased.

Keywords --- Parallel Computing, Multicore Architecture, Scheduling Algorithm

1. INTRODUCTION

In computing science, a lot of computation is involved in order to generate some useful result. Based on the degree of dependency or independency of data, computation is majorly divided into two types, i.e. deterministic and non-deterministic computation/algorithm. A deterministic algorithm is an algorithm which, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states . So in deterministic approach, no random computation is involved. In this, the overall computation is depending on the particular sequence of execution of data. This paper talks about how to achieve more parallelism in context of a deterministic computation approach for the dual-core architecture.

2. ABOUT MULTICORE

The past few years have witnessed a persistent increase in the number of cores per CPU. At its simplest, multicore is a design in which a single physical processor contains the core logic of more than one processor. The multicore design puts several such processor "cores" and packages them as a single physical processor. The goal of this design is to enable a system to run more tasks simultaneously and thereby achieve greater overall system performance.

Programs are made up of execution threads. These threads are sequences of related instructions these cores are responsible for executing the threads which are the smallest unit of CPU utilisation. Multicore processors, as the name implies, contain two or more distinct cores in the same physical package. Figure 2 shows how this appears.

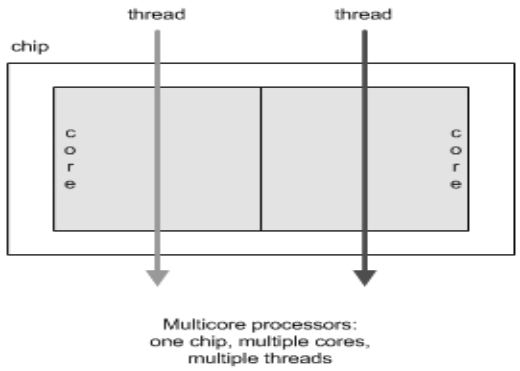

Fig. 1 showing the multicore example

In this design, each core has its own execution pipeline. And each core has the resources required to run without blocking resources needed by the other software threads. The example in figure 1 shows a two-core design which is in general called as dual-core architecture, there is no inherent limitation in the number of cores that can be placed on a single chip. The multicore design enables two or more cores to run at somewhat slower speeds and at much lower temperatures. The combined throughput of these cores delivers processing power greater than the maximum available today on single-core processors and at a much lower level of power consumption.

3. DESIGN COMPONENTS

In order to utilise these cores maximum amount of time, we need better scheduling algorithms for the deterministic systems. In this paper we tried to evaluate one of the deterministic computations. In computing science the deterministic approach can be shown by Directed Acyclic Graph (DAG) in which the nodes of a DAG are computational tasks and the edges are the dependency among tasks. In figure 2, which shows the very basic DAG, in which total three tasks(1,2,3) are described and task 3 cannot be executed or scheduled before the task 1 and task 2 completes their execution and execution basically depends on the scheduling criteria.

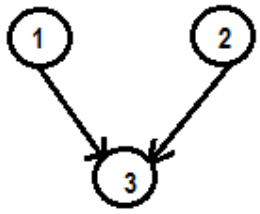

Fig. 2 A very basic DAG of three tasks

From software perspective, each task can be visualised as a software module or a function. And each module cannot be schedule for execution until or unless it has scheduled all its predecessor dependent function or module.

3.1 Thread as Computation

In general-purpose software engineering practice, we have reached a point where one approach to concurrent programming dominates all others, namely, threads. From computation point of view, we considered threads, as a model of computation. Threads are sequential processes that share memory. They represent a key concurrency model supported by modern computers. In our problem, we considered the thread as computational model because parallel architecture is responsible for thread level parallelism. For each task, corresponding one thread can be generated so that it could make use of the parallel computer architecture efficiently. So from now onward task and thread are interchangeable. We have restricted the number of core is 2, which in general termed as dual-core architecture as well.

The effective scheduling of the deterministic approach is always a NP hard problem for multicore architecture. One of the simplest scheduling mechanisms for any number of cores is that resolve the dependency and sequentially schedule that tasks/thread of that program but the response time of a program would be larger and in this mechanism it won't effectively utilise the other cores as well. For multicore architecture we need to busy both of the cores (in case of dual-core architecture) as much as possible in order to reduce the response time of the program without violating the constraint associated with it. We termed our new scheduling algorithm as LA algorithm but before going to scheduling algorithm we need some pre-processing steps and some extra hardware components before actual scheduling being done.

3.2 Pre-processing

1. In this first pre-processing step, it takes the raw problem as an input. In this step, it executes the Kahn Topological sorting algorithm which takes the DAG as an input and returns the List **L** of sorted task. So **L** is set of task to be computed. The Kahn Topological sorting algorithm is explained in [3].

2. The second pre-processing step is to calculate the number of out degree edges in a DAG for each task. This can be done by any simple traversal algorithm. So the **OUTDEG** is a set of natural numbers which represents the number of outwards edges for each task.

3.3 Queue Component

We need two hardware queue which maintains the **L** and **OUTDEG** which we have termed as Q_L[] and Q_OUTDEG[] respectively and it should be in such a way that

Q_L[i] = Vi

Q_OUTDEG[i] = Di

Where Vi belongs to **L** and Di belongs to **OUTDEG**

3.4 Scheduling Algorithm

The pseudocode of the scheduling algorithm which we termed as LA scheduling is mentioned below, which takes the **L** and **OUTDEG** and schedules the tasks/thread on two different cores **M** where M= {0,1}. The prior assumption of this algorithm is that worst case execution time unit is constant for each thread and they all are available at particular instant of time.

LA (L , OUTDEG)

1. Initialise i ← 0

2. Schedule the tasks/vertex while **L** is not empty

3. For node Vi in **L**
   Initialise j ← 0
   Initialise n
   n ← Di , where Di belongs to **OUTDEG** and Vi belongs to **L**

4. If n ≥ 2 then do this
   Mj ← Vi
   Delete Vi from **L**
   While (n!=0)
   {
   i++
   if(Di < 2)
   {
   Mj ← Vi
   update j, where j={0,1}
   Delete Vi from **L**
   }
   else
   {
   n ← Di + 1
   Mj ← Vi
   Delete Vi from **L**
   }
   n--
   }
   i++

Else do this
   Mj ← Vi
   Delete Vi from **L**
   i++

4. EXAMPLE CONSIDERATION

The problem is shown in figure 3, which shows the DAG with nine nodes or task and each task has been assigned a unique integer. As explained earlier, if each task is computed by one thread, then total nine threads will be scheduled on two cores in order to execute. After two pre-processing steps mentioned in section 3.2 and applying the LA scheduling mentioned in section 3.4, the scheduling diagram is shown in figure 4.

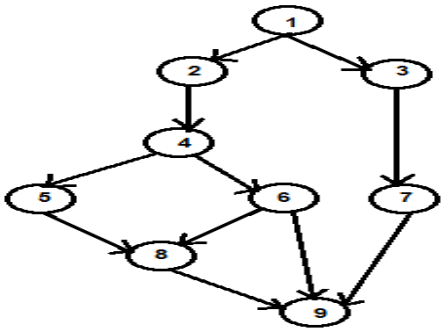

Fig. 3 DAG with nine nodes/tasks

1	2	4	7	6	8
	3		5		9

0 10 20 30 40 50 60

Fig. 4 Scheduling of threads on two cores **M**

4.1 Simulation

In our simulation as shown in figure 5 and 6, we restricted the maximum execution time for each thread is 10 time unit. We took different simulation samples for different set of DAG structure having fixed number of tasks which is nine in our case. As can be seen that the response time of a program can be reduced using the LA scheduling and after applying the Amdahl's law of parallel computation, the average speedup found to be 1.29.

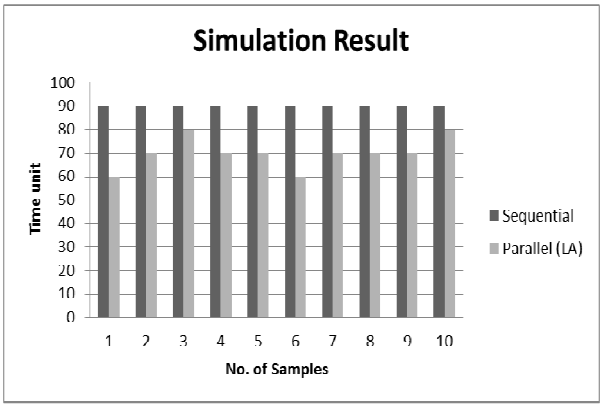

Fig. 5 Comparison of response time (Sequential Vs Parallel (LA))

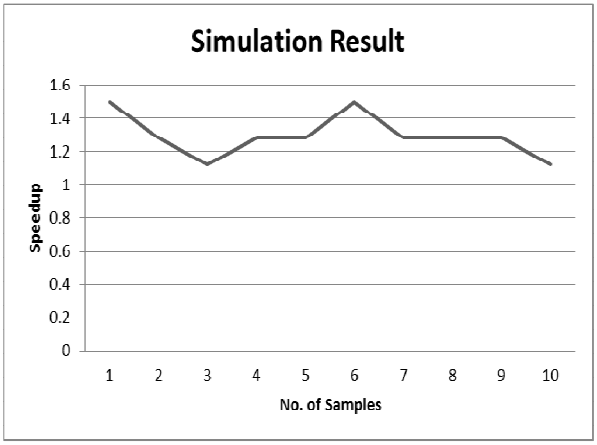

Fig. 6 Speedup diagram

5. CONCLUSION

With persistent increase in the number of cores for last few years, we need better core utilisation algorithms. In this paper we presented one of the efficient scheduling algorithm and some of its associated parameter which works efficiently in deterministic computation approach where no random computation is involved for the dual-core architecture. Our simulation result shows that response time of a program can be reduced effectively and average speedup has also been increased.

REFERENCES

[1] Cormen, Thomas H., et al. Introduction to algorithms. Vol. 2. Cambridge: MIT press, 2001.

[2] Karpinski, Marek, and Rutger Verbeek. "On randomized versus deterministic computation." Automata, Languages and Programming. Springer Berlin Heidelberg, 1993. 227-240.

[3] Kahn, Arthur B. "Topological sorting of large networks." Communications of the ACM 5.11 (1962): 558-562.

[4] Lee, Edward A. "The problem with threads." Computer 39.5 (2006): 33-42.

[5] Silberschatz, Abraham, et al. Operating system concepts. Vol. 4. Reading: Addison-Wesley, 1998.

[6] Bosilca, George, et al. "DAGuE: A generic distributed DAG engine for high performance computing." Parallel Computing 38.1 (2012): 37-51.

[7] Intel Corporation https://software.intel.com/en-us/articles/multi-core-processor-architecture-explained October 2008

