Jurnal Teknologi, 37(C) Dis. 2002: 55–66

DYNAMIC MULTIPROCESSOR SCHEDULING MODEL FOR THE RECONFIGURABLE MESH COMPUTING NETWORKS

SHAHARUDDIN SALLEH[1], NUR ARINA BAZILAH AZIZ[2], NOR AFZALINA AZMEE[3] & NURUL HUDA MOHAMED[4]

Abstract. Task scheduling is a combinatorial optimisation problem that is known to have large interacting degrees of freedom and is generally classified as NP-complete. Most solutions to the problem have been proposed in the form of heuristics. These include approaches using list scheduling, queueing theory, graph theoretic and enumerated search. In this paper, we present a dynamic scheduling method for mapping tasks onto a set of processing elements (PEs) on the reconfigurable mesh parallel computing model. Our model called the *Dynamic Scheduler on Reconfigurable Mesh* (DSRM) is based on the Markovian *m/m/c* queueing system, where tasks arrive and form a queue according to Poisson distribution, and are serviced according to the exponential distribution. The main objective in our study is to produce a schedule that distributes the tasks fairly by balancing the load on all PEs. The second objective is to produce a high rate of successfully assigned tasks on the PEs. These two requirements tend to conflict and they constitute the maximum-minimum problem in optimisation, where the maximum of one causes the other to be minimum. We study the effectiveness of our approach in dealing with these two requirements in DSRM.

Key Words: Reconfigurable mesh, task scheduling, load balancing, and parallel computing

Abstrak. Masalah penjadualan kerja ialah satu masalah pengoptimuman kombinatorik yang diketahui mempunyai darjah kebebasan berinteraksi yang besar. Oleh itu, masalah ini selalunya dikategorikan sebagai NP-lengkap. Kebanyakan penyelesaian bagi masalah ini menggunakan heuristik. Penyelesaiannya termasuk pendekatan berasaskan penjadualan tersenarai, teori giliran, teori graf dan pencaraian berenumerasi. Dalam kertas ini, satu kaedah penjadualan dinamik dicadangkan bagi memeta satu set kerja kepada satu set pemproses dalam rangkaian jaring boleh-konfigurasi. Model kami dipanggil *Dynamic Scheduler on Reconfigurable Mesh* (DSRM). Model ini berasaskan sistem giliran *m/m/c* di mana kerja-kerja yang tiba menunggu giliran masing-masing mengikut taburan Poisson, dan diservis mengikut taburan eksponen. Objektif utama dalam kajian ini ialah untuk menghasilkan jadual yang mempunyai taburan kerja seimbang pada pemproses-pemproses. Objektif kedua ialah untuk mempertingkat kadar pengagihan kerja ke tahap maksimum untuk memastikan kejayaan. Kedua-dua objektif ini merupakan satu masalah dikenali sebagai maksimum-minimum, di mana kejayaan dalam satu objektif boleh menyebabkan kemerosotan dalam objektif yang satu lagi. Keberkesanan pendekatan ini dikaji melalui model simulasi DSRM.

Kata Kunci: Jaring boleh-konfigurasi, penjadualan kerja, pengseimbangan beban dan perkomputeran selari

[1,2,3&4] Parallel Algorithm Research Group, Faculty of Science (Mathematics), Universiti Teknologi Malaysia 81310 Skudai, Johor.

1.0 INTRODUCTION

Task scheduling is a combinatorial optimisation problem that is known to have large interacting degrees of freedom and is generally classified as NP-complete [1]. Most solutions to the problem have been proposed in the form of heuristics. These include approaches using list scheduling, queueing theory, graph theoretic and enumerated search. Task scheduling is defined as the scheduling of tasks or modules of a program onto a set of autonomous processing elements (PEs) in a parallel network, so as to meet some performance objectives [1]. The main objective in task scheduling is to obtain a scheduling model that minimizes the overall execution time of the processing elements. Another common objective is distribute the tasks evenly among the processing elements, an objective known as *load balancing*. Task scheduling applications can be found in many areas, for example, in real-time control of robot manipulators [1], flexible manufacturing systems [1], and traffic control [1].

In terms of implementation, task scheduling can be classified as either static or dynamic. In static scheduling, all information regarding the states of the tasks and the processing elements are known beforehand prior to scheduling. In contrast, all this information is not available in dynamic scheduling and it is obtained *on the fly*, that is, as scheduling is in progress. Hence, dynamic scheduling involves extra overhead to the processing elements where a portion of the work is to determine the current states of both the tasks and the processing elements.

In this paper, we consider the task scheduling problem on the reconfigurable mesh architecture. A *reconfigurable mesh* is a bus-based network of N identical PE[k], for k = 1, 2, ..., N, positioned on a rectangular array, each of which has the capability to change its configuration dynamically according to the current processing requirements. Figure 1 shows a 4×5 reconfigurable mesh of 20 processing elements. Due to its dynamic structure, the reconfigurable mesh computing model has attracted researchers on problems that require fast executions. These include numerically-intensive applications in computational geometry [2], computer vision and image processing [3] and algorithm designs [4].

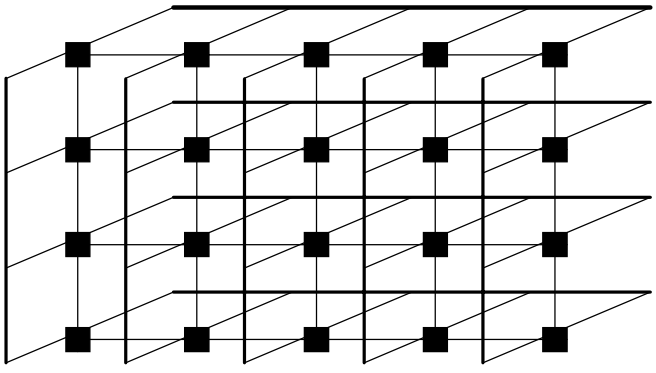

Figure 1 A reconfigurable mesh of size 4×5

This paper is organized into five sections. Section 1 is the introduction. Section 2 is an overview of the dynamic scheduling problem, while Section 3 describes our model which is based on the reconfigurable mesh computing model. The simulation results of our model are described in Section 4. Finally, Section 5 is the summary and conclusion.

2.0 DYNAMIC TASK SCHEDULING PROBLEM

Dynamic scheduling is often associated with real-time scheduling that involves periodic tasks and tasks with critical deadlines. This is a type of task scheduling caused by the *nondeterminism* in the states of the tasks and the PEs prior to their execution. Nondeterminism in a program originates from factors such as uncertainties in the number of cycles (such as loops), the and/or branches, and the variable task and arc sizes. The scheduler has very little *a priori* knowledge about these task characteristics and the system state estimation is obtained on the fly as the execution is in progress. This is an important step before a decision is made on how the tasks are to be distributed.

The main objective in dynamic scheduling is usually to meet the timing constraints, and performing load balancing, or a fair distribution of tasks on the PEs. Load balancing improves the system performance by reducing the mean response time of the tasks. In [5], load balancing objective is classified into three main components. First, is the *information rule* which describes the collection and storing processes of the information used in making the decisions. Second, is the *transfer rule* which determines when to initiate an attempt to transfer a task and whether or not to transfer the task. Third, is the *location rule* which chooses the PEs to and from which tasks will be transferred. It has been shown by several researchers [5,6,7] that with the right policy to govern these rules, a good load balancing may be achieved.

Furthermore, load balancing algorithms can be classified as *source-initiative* and *server-initiative* [6]. In the source-initiative algorithms, the hosts where the tasks arrive would take the initiative to transfer the tasks. In the server-initiative algorithms, the receiving hosts would find and locate the tasks for them. For implementing these ideas, a good load-balancing algorithm must have three components, namely, the information, transfer and placement policies. The information policy specifies the amount of load and task information made available to the decision makers. The transfer policy determines the eligibility of a task for load balancing based on the loads of the host. The placement policy decides which eligible tasks should be transferred to some selected hosts.

Tasks that arrive for scheduling are not immediately served by the PEs. Instead they will have to wait in one or more queues, depending on the scheduling technique adopted. In the *first-in-first-out* (FIFO) technique, one PE runs a scheduler that dispatches tasks based on the principle that tasks are executed according to their arriving time, in the order that earlier arriving tasks are executed first. Each dispatch PE maintains its own waiting queue of tasks and makes request for these tasks to be executed

to the scheduler. The requests are placed on the schedule queue maintained by the scheduler. This technique aims at balancing the load among the PEs and it does not consider the communication costs between the tasks. In [6], a queueing model has been proposed where an arriving task is routed by a task dispatcher to one of the PEs. An approximate numerical method is introduced for analyzing two-PE heterogeneous models based on an adaptive policy. This method reduces the task turnaround time by balancing the total load among the PEs. A central task dispatcher based on the single-queue multiserver queueing system is used to make decisions on load balancing. The approach is efficient enough to reduce the overhead in trying to redistribute the load based on the global state information.

Several *balance-constrained* heuristics, such as in [7], consider communication issues in balancing the load on all PEs. The approach adds balance constraint to the FIFO technique by periodically shifting waiting tasks from one waiting queue to another. This technique performs local optimisation by applying the steepest-descent algorithm to find the minimum execution time. The proposed *cost-constraint* heuristic further improves the load balancing performance by checking the uneven communication cost and quantify them as the time needed to perform communication.

Our performance index for load balancing is the *mean response time* of the processing elements. The response time is defined as the time taken by a processing element to response to the tasks it executes. In general, load balancing is said to be achieved when the mean response time of the tasks is minimized. A good load balancing algorithm tends to reduce the mean and standard deviation of the task response times of every processing elements in the network.

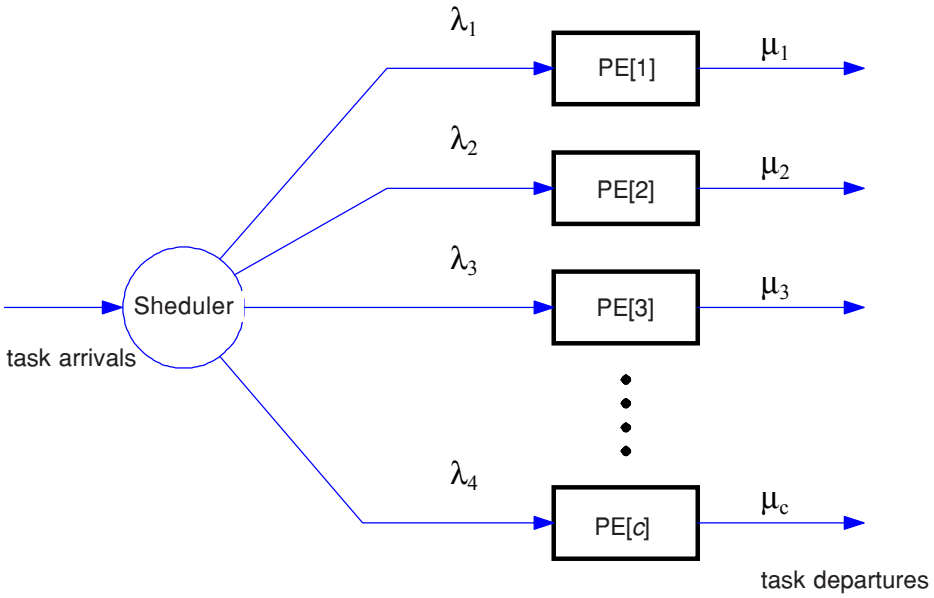

Figure 2 The *m/m/c* queueing model

In our work, task scheduling is modeled as the *m/m/c* Markovian queueing system. An algorithm is proposed to distribute the tasks based on the probability of a processing element receiving a task as the function of the mean response time at each interval of time and the overall mean turnaround time. Tasks arrive at different times and they form a FIFO queue. The arrival rate is assumed to follow the Poisson distribution with a mean arrival rate of λ. The service rate at processing element k is assumed to follow the exponential distribution with mean μ_k. Our idea is illustrated through a simulation model called DSRM which is explained in Section 4.

In general, the mean response time R for tasks arriving at a processing element is given from the Little's law defined in [8], as follows:

$$R = \frac{N}{\lambda} \tag{1}$$

where N is the mean number of tasks at that processing element. In a system of n processing elements, the mean response time is given as follows [8]:

$$R_k = \frac{1}{\mu_k - \lambda_k} \tag{2}$$

where l_k is the mean arrival rate and is the mean service rate at the processing element k. It follows that the mean response time for the whole system is given as follows [8]:

$$R = \frac{1}{n^*} \sum_{k=1}^{n} R_k \tag{3}$$

where $n^* = \sum_{k=1, \lambda_k \neq 0}^{n} 1$.

3.0 RECONFIGURABLE MESH COMPUTING MODEL

Our computing platform consists of a network of 16 processing elements arranged in a reconfigurable mesh. A suitable realization for this model is the message-passing transputer-based system where each node in the system is a processor which includes a memory module. In addition, each processor in the system has communication links with other processors to enable message and data passing.

3.1 Computational Model

The computational model is a 4×4 network of 16 processing elements, $PE[k]$, for $k =$ 1, 2, ..., 16, as shown in Figure 3. Each processing element in the network has four ports, denoted as $PE[k].n$, $PE[k].s$, $PE[k].e$ and $PE[k].w$, which represent the north,

south, east and west communicating links respectively. These ports can be dynamically connected in pairs to suit some computational needs.

Communication between the processing elements in the reconfigurable mesh can be configured dynamically in one or more buses. A bus is a doubly-linked list of processing elements, with every processing element on the bus being aware of its immediate neighbours. A bus begins in a processing element, pass through a series of other processing elements and ends in another processing element. A bus that passes through all the processing elements in the network is called the *global bus*, otherwise it is called a *local bus*. Figure 3 shows two local buses $B(1) = \{2,1,5,9,13,14\}$ and $B(2) = \{12,16,15,14,10,6,2,3,7,8,4\}$, where the numbers in the lists represent the processing element numbers arranged in order from the first (starting) processing element to the last (end). As an example, from Figure 3, communication between *PE*[9] and *PE*[9] on the bus *PE*[9] is made possible through the link {*PE*[9].s, *PE*[9].*n*}

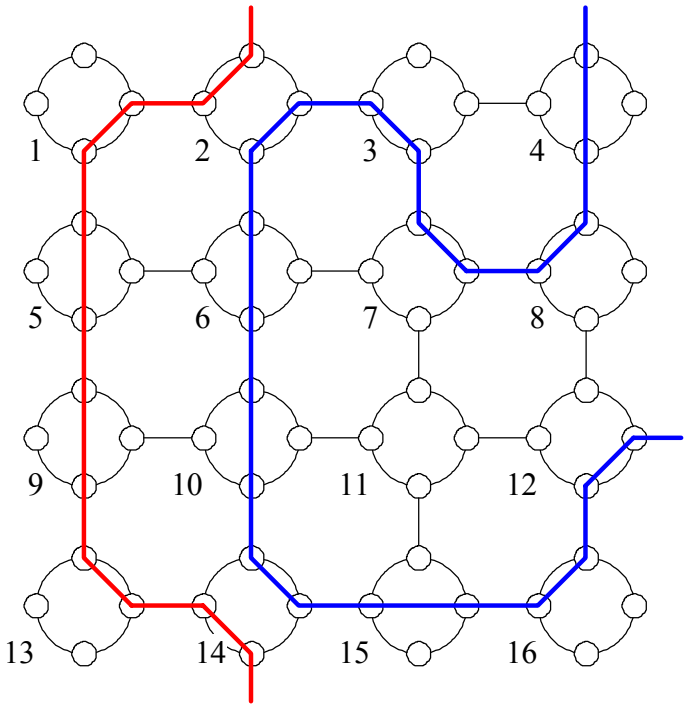

Figure 3 A 4 × 4 reconfigurable mesh network with two subbuses

The processing elements in a bus cooperate to solve a given problem by sending and receiving messages and data according to their controlling algorithm. A *positive direction* in a bus is defined as the direction from the first processing element to the last processing element, while the *negative direction* is the opposite. Note that the contents in the list of each bus at any given time t can change dynamically according to the current computational requirements.

3.2 Scheduling Model and Algorithm

In the model, *PE*[1] assumes the duty as the *controller* to supervise all activities performed by other processing elements in the network. This includes gathering information about the incoming tasks, updating the information about the currently executing tasks, managing the buses and locating the positions of the PEs for task assignments.

In our model, we assume the tasks to be nonpreemptive, independent and have no precedence relationship with other tasks. Hence, the computational model does not consider the communication cost incurred as a result of data transfers between tasks. We also assume the tasks to have no hard or soft executing deadlines. At time $t = 0$, the controller records Q_0 randomly arriving tasks, for $0 < Q_0 < Q$, and immediately places them in a FIFO queue, where Q is a predefined maximum number of tasks allowed. Each task *Task[i]* is assigned a number i and a random length, denoted as *Task[i].length*. The controller selects q_0 connected PEs to form the bus $B(0)$ and assigns the Q_0 tasks to the q_0 PEs in $B(0)$. At this initial stage, the controller creates the bus list S to consist of a single bus $B(0)$, that is, $S = \{B(0)\}$. The PEs then start executing their assigned tasks, and their status are updated to "busy". Each PE broadcasts the information regarding its current execution status and the task it is executing to the controller, and the latter immediately updates this information.

This initial operation is repeated in the same way until the stopping time $t = StopTime$ is reached. At time t, Q_t random new tasks arrive and they are immediately placed in the FIFO queue. The queue line is created in such a way that every task will not miss its turn to be assigned to a PE. There are some Q_w tasks who failed to be assigned from the previous time slots, and these tasks are automatically in the front line. Hence, at any given time t, there are $Q_t + Q_w$ tasks in the queue, of which all Q_w tasks are in front of the Q_t tasks. In an attempt to accommodate these tasks, the controller forms m buses in the list $S = \{B(0), B(1), ..., B(m)\}$. Each bus $B(j)$ has q_j connected PEs and this number may change according to the current processing requirements. The controller may add or delete the contents of each bus $B(j)$, depending on the overall state of the network. A PE in a bus that has completed executing a task may be retained or removed from this bus, depending on the connectivity requirements for accommodating the tasks. The controller also checks the status of other PEs not in the list S. These PEs are not "busy" and may be added to the connecting buses in S. At the same time, some PEs may be transferred from one bus in to another bus. In addition, the controller may also add or delete one or more buses in the list to accommodate the same processing needs. Finally, when the buses have been configured a total of q_t "free" PEs are then assigned to the q_t tasks in the front queue. When all the tasks have been completely executed, the controller compiles the information in its database to evaluate the mean arrival time λ_k, the mean executing time μ_k, and the mean response time R_k of each PE[k] in the network.

Our algorithm for scheduling the tasks dynamically on the reconfigurable mesh is summarised as follows:

At t=0, the controller records Q_0 newly arriving tasks;
The controller selects q_0 connected PEs at random to form the bus $B(0)$;
The Q_0 new tasks are assigned to the PEs in $B(0)$;
The controller flags the assigned PEs in $B(0)$ as "busy";
The controller creates the bus list $S = \{B(0)\}$;
The controller updates the state information of the PEs in $S = \{B(0)\}$;
for t = 1 to StopTime
    Q_t new tasks arrive while Q_w tasks still waiting;
    The controller places all the $Q_t + Q_w$ tasks in the FIFO queue;
    The controller checks the state information of the PEs in $B(j)$ of
      the list $S = \{B(0), B(1), ..., B(m)\}$, where $B(j) \subseteq S$:
    The controller checks the state information of the PEs not in S;
    The controller decides if the contents of $B(j)$ need to change;
    The controller decides if the list S needs to change;
    The controller selects "free" PEs, assign them to the buses in S;
    The controller assigns q_t PEs to the q_t front tasks;
    The controller updates the state information of the PEs in S;
The controller evaluates λ_k, μ_k, and of R_k PE[k] in S;

4.0 SIMULATION AND ANALYSIS OF RESULTS

The simulation is performed on an Intel Pentium II personal computer. A C++ Windows-based simulation program called *Dynamic Scheduler on Reconfigurable Mesh* (DSRM) has been developed to simulate our model. DSRM assumes the tasks to have no partial orders, no communication dependence, no timing constraints and are nonpreemptive. Figure 4 shows a sample run of some randomly arriving tasks on a 4 × 4 network. In DSRM, every time tick t is a discrete event where between 0 to 10 randomly determined number of tasks are assumed to enter the queue waiting to be assigned to the PEs. For each task, its arrival time (randomly determined), length (randomly determined) and completion time, is displayed as a dark or light bar in the Gantt chart.

DSRM has some flexible features which allow a user-defined mesh network sizes of $m \times n$, where . In addition, DSRM also displays the status of each processor in the network at time as a square. A square with a light border indicates the processor is busy as it has just been assigned a task, while a square with a dark border indicates the processor is also currently busy as it is still executing a previously assigned task. An unmarked square indicates the processor is currently idle and, therefore, is ready for assignment. Figure 4 shows an instance of this discrete event at t = 20. PE[3] is busy as it has just been assigned with Task 98, while PE[7] is also busy as it is still executing Task 92. In contrast, PE[11] is currently idle and is waiting for an assignment.

Results from a sample run of 209 successfully assigned tasks on a 4 × 4 network are shown in Table 1. Due to its dynamic nature, not all the tasks that arrive at time t

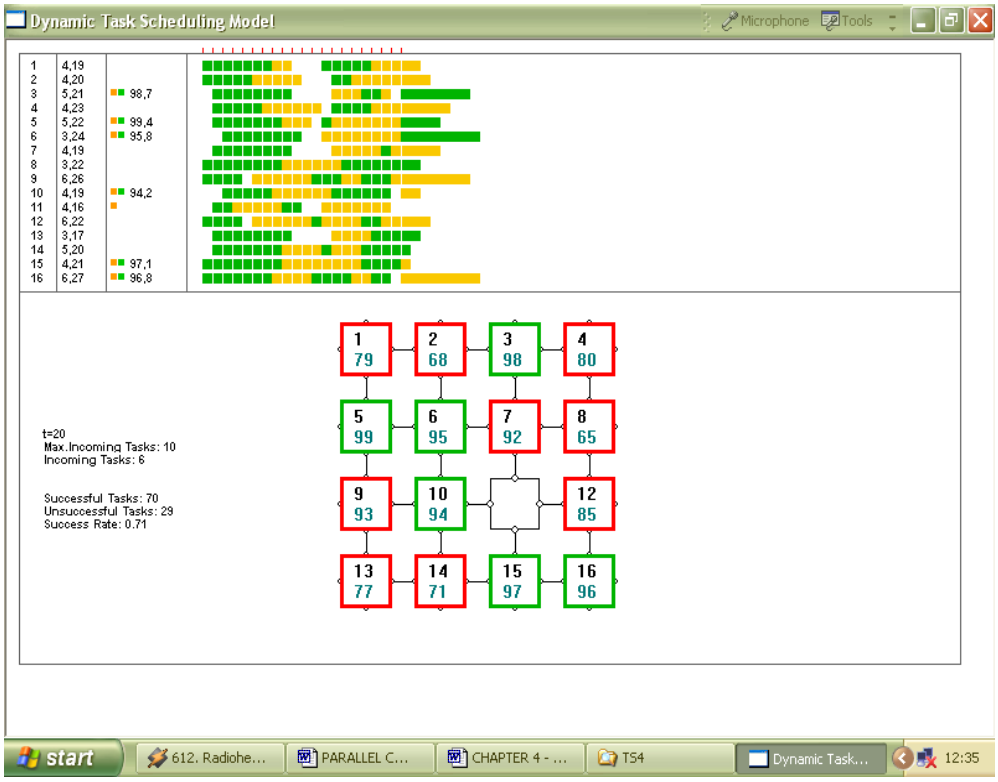

Figure 4 Sample run from DSRM

managed to be assigned successfully on the limited number of processors. In this sample, 35 tasks failed to be assigned and this gives the overall success rate of 85.7%, which is reasonably good. In general, the overall success rate can be improved by controlling factors such as reducing the maximum number of arriving tasks at every time tick t and increasing the network size. In addition, it is possible to have a 100% success rate by bringing forward the unsuccessfully assigned tasks at time t to enter the queue at time $t + 1$, $t + 2$ and so on. These factors normally imposes some timing constraints on the tasks, such as the execution deadline, and are presently not supported in DSRM.

The results from Table 1 show a fairly good distribution of tasks on the processors with a mean of 13.0625, with *PE*[1] having the highest number of tasks (17), while *PE*[5] has the lowest assignment (9). The standard deviation is 2.3310, while the overall mean response time is 1.880. The tasks have a total execution time of 824 time units, with a mean of 51.5 and a standard deviation of 5.1720 on each processor. The table also shows the performances of each processor in the network, in terms of its mean arrival time, mean service time and mean response time, which describes a reasonably good distribution.

Table 1 Sample run of 209 successful randomly generated tasks on 16 PEs

PE	No. of Tasks	Total Exec. Time	Mean Arrival Time	Mean Service Time	Mean Response Time
1	17	45	0.261538	0.377778	8.60294
2	16	53	0.246154	0.301887	17.9427
3	12	60	0.184615	0.2	65
4	11	45	0.169231	0.244444	13.2955
5	9	48	0.138462	0.1875	20.3922
6	12	50	0.184615	0.24	18.0556
7	10	54	0.153846	0.185185	31.9091
8	16	52	0.246154	0.307692	16.25
9	12	54	0.184615	0.222222	26.5909
10	11	44	0.169231	0.25	12.381
11	14	50	0.215385	0.28	15.4762
12	15	52	0.230769	0.288462	17.3333
13	15	60	0.230769	0.25	52
14	13	58	0.2	0.224138	41.4286
15	11	44	0.169231	0.25	12.381
16	15	55	0.230769	0.272727	23.8333
Total	209	824			
Mean	13.0625	51.5			1.880
Std.Dev.	2.3310	5.1720			

5.0 SUMMARY AND CONCLUSION

This paper describes dynamic task scheduling model implemented on the reconfigurable mesh computing model. The model is illustrated through our simulation program called *Dynamic Simulator on Reconfigurable Mesh* (DSRM) which maps a randomly generated number of tasks onto a network of $m \times n$ processors at every unit time t based on our scheduling algorithm. DSRM produces reasonably good load balancing results with a high rate of successful assigned tasks, as demonstrated in the sample run.

DSRM considers the tasks to have no partial orders, no communication dependence, no timing constraints and are nonpreemptive. These important factors will be considered in our future work as they are necessary in order for the model to be able to support many real-time and discrete-event requirements.

REFERENCES

[1] El-Rewini, H., T.G. Lewis and H.H. Ali. 1994. *Task scheduling in parallel and distributed systems. Prentice Hall.*

[2] Olariu, S., J.L. Schwing and J. Zhang. 1995. A fast adaptive convex hull algorithm on two-dimensional

processing element arrays with a reconfigurable bus system. *Computational Systems Science and Engineering.* 3:131-137.

[3] Olariu, S., J.L. Schwing and J. Zhang. 1995. Fast computer vision algorithms for reconfigurable meshes. *Image and Vision Computing*, 10(9):610-616.

[4] Nakano, K. and S. Olariu. 1998. An efficient algorithm for row minima computations on basic reconfigurable meshes. *IEEE Trans. Parallel and Distributed Systems.* 9(8).

[5] Lin, H. and C.S. Raghavendran. 1991. A dynamic load-balancing policy with a central task dispatcher. *IEEE Trans. Software Engineering*. 18(2):148-158.

[6] Chow, Y. and W.H. Kohler. 1979. Models for dynamic load balancing in heterogeneous multiple processing element systems. *IEEE Trans. Computers*. 28(5):354-361.

[7] Saletore, V. 1990. A distributed and adaptive dynamic load balancing scheme for parallel processing of medium-grain tasks. *Proc. of DMCC-5*, Portland, Oregon. 994-999.

[8] Kobayashi, H. 1978. *Modeling and analysis: an introduction to system evaluation methodology*. Addison-Wesley.