A simulation Study of Dynamic Resource Allocation Scheme for Distributed transactions

Bibhu D. Sahoo
Department of Computer Science Engineering and Applications
Regional Engineering College, Rourkela-769 008, Orissa.
E-mail: bibsss@rec.ren.nic.in

Abstract

Most distributed systems nowadays are consists of various nodes having different functions and/or different processing capabilities and speeds. We have considered a heterogeneous distributed system consists of a set of nodes(autonomous computers) with same functionality but different processing capability. This paper is aimed to find out the number of autonomous computer required to design a distributed computing platform (DCP) for a specific problem domain. The transactions initiated in any node routed to the central job scheduler (leader) for execution. The leader once elected is assumed to be fault-free during the assignment and execution of a submitted task. Scheduler model uses *Dijkstra algorithm* that takes half of that of *Bellman-Frod* algorithm for Single Origin Shortest Path problem(SOHP). Simulation are being done on a Pentium-II computer system with the model developed using Boroland C++.

1. Introduction

A distributed computing system (DCS) is defined as a computing system consisting of at least two autonomous processors connected by a network. Within a DCS, many processors may share resources; which includes files, printers and CPU's. Since many processes may try to access a particular resource or may demand multiple resources at the same time, access to the resources are scheduled to avoid conflict and to optimize the resource utilization for optimum performance.

The set of all computation processors (called PEs) has been partitioned into subsets such that one controller (Coordinator or leader) currently controls all PEs in a subset. One of the main advantages of distributed systems over stand-alone systems is that balancing the workload of the system among the computers (nodes) can improve system performance. The model of network structure is being realized using a two dimensional adjacent matrix. A class of transaction with varying size is being executed on the network.

A fundamental problem posed for such a heterogeneous system is the choice of static and dynamic policies. A distributed system (also called network) is represented as an undirected connected graph, where the nodes represent processor and the edges represents bi-directional communication links. The scheduling decision made by the schemes is based on task deadline and resource requirement. Also, the notion of guarantee underlines all scheduling decisions: when a task arrives at a node, the local scheduler at that node attempts to guarantee that the task will complete execution before its deadline, on that node. If the attempt fails, the scheduling components on individual nodes cooperate to determine which other node in the system has sufficient resource surplus to guarantee the task.

In general, Dynamic task scheduling schemes have been applied extensively in experimental distributed systems and have shown significant potential for performance improvement [10]. Following present trends of Internet and Network computing using

Java as well as price reduction in hardware, it is anticipated that large-scale Distributed-computing system incorporating dynamic task scheduling will rapidly be employed in a large scale in the future. A distributed parallel-computing platform with multiple computers connected to Internet forms a Java-Internet Computing Environment (JICE)[8]. The task scheduling schemes can schedule processes among the idle systems in the entire global network, with the use of multithreading and remote method invocation (RMI) interface provided with Java.

2. Distributed Algorithms

Distributed algorithms [3] are algorithms designed to run on hardware consisting of many interconnected processors. Some of the attributes of these distributed algorithms includes (i) the inter-processor communication (IPC) method, (ii) the timing model, (iii) the failure model and (iv) the problem addressed in this project work. The distributed algorithms are designed either for fixed connection networks (as arrays, tress, and hypercubes etc) or networks with some type of uncertainty and independence [7]. Task scheduling in distributed computing systems consists of local scheduling and global scheduling. Local scheduling involves assignment of tasks to time-slices of a single PE whereas global scheduling involves deciding where a task should be executed. These scheduling schemes are realized using *scheduling algorithms*, which are broadly classified as **Static** and **Dynamic**. Static Schemes use enumerative, graph-theoretic, mathematical and quadratic programs. They use apriori knowledge about task behavior and do not obtain information about dynamically changing states. We are concerned with the Dynamic Schemes in the ensuing sections. In our experiment we first out the average execution time of distributed transactions over a network structure and then try to find out the relation between the number of sub-transactions and number of processors on the network, with a fixed topology.

Various scheduling schemes can be used to find out the completion time of a distributed transaction. We have used *Dynamic Schemes* that make few assumptions about task characteristics and obtain information about the system task before making a task scheduling decision. This is a most realistic practice in distributed computation. The applicability of task scheduling algorithm [2] depends on the amount of information available about the attributes of given network viz. (i) no network information is available at all, (ii) an upper bound on the number of processors in the network (N) is available, (iii) the exact number of processors in the network is available (size), and (iv) the topology of the network is available (topology). We have conducted both theoretical and simulation studies on Dynamic Scheduling. The distributed network model is implemented as a graph. Transaction assignment and executions were simulated on a Pentium-II machine by using C++.

3. The Distributed system Model

The distributed (system) network studied is modeled as an anonymous network by an undirected, connected, simple graph $G = (V, E)$, where the vertex set, $V=\{v_1 ,v_2, \ldots, v_n\}$ represents the processors in the network and the edge set E represents the bi-directional links among the processors. An edge $e \in E$ is represented by (u, v), if e connects u - for all $(u, v) \in V$. Let G denote the set of all such graphs (networks).

Each processor is assumed to have unlimited computational power, it has sufficiently large local memory and can access and change its memory content instantaneously. In executing a given sequential algorithm, a processor depending on the current memory content - either changes its memory content, sends a message via one of its ports or receives a message via a port - for each step of the algorithm. The processors are anonymous in the sense that they do not have identity numbers[5], and the processors run the same deterministic algorithm. Although we label the processors in V by unique name $v_1,\ldots.v_n$, these names are used only for description purposes, and the processors do not know their names. In other words, the algorithm, which a processor executes, does not use its identity number to make a decision or to compute a value. We have assumed that all processors have the same execution speeds and ensure fairness, i.e. only after a job reaches termination, a processor takes the next job for execution in finite time.

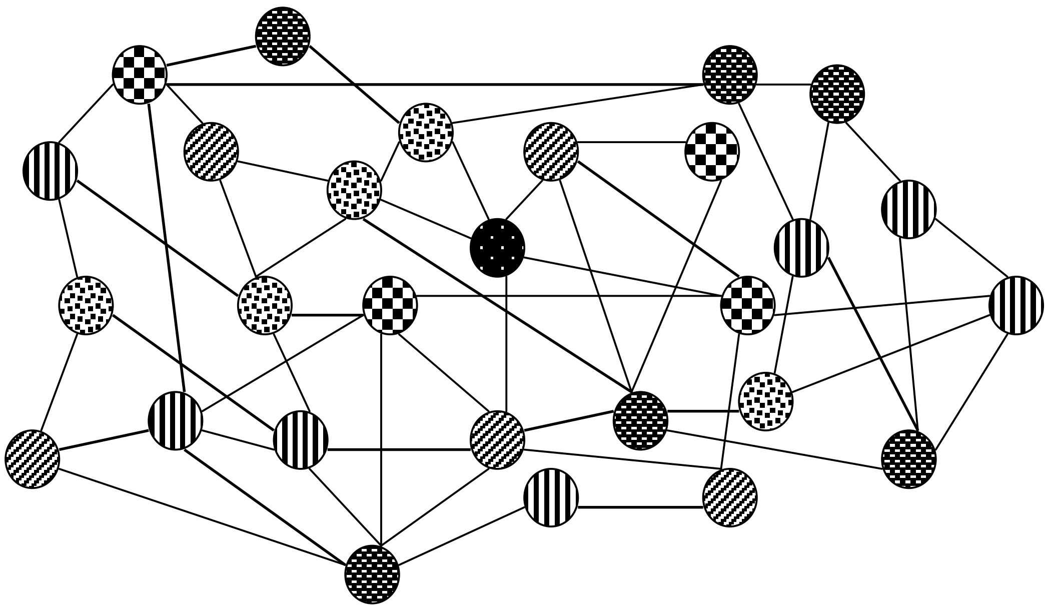

Figure 1. Distributed network (processor) model with processors having unlimited computational capability

Communication is carried out by sending messages through links which are nothing but the edges $e \in E$. A processor v is equipped with *deg(v)* number of input/output ports, one for each link incident to it- named 1, …, *deg(v)* - where *deg(v)* denotes the degree of v. Let port *j* be processor *u*'s port for the link (u,v). When processor *u* executes the instruction "*send message M via port j*", M is sent to the input queue of processor v through link *e* in finite time with no error and in the FIFO order. Messages sent through the link are placed in the input queue in the order they are sent. In order to receive a message placed in an input queue, the '*receive*' instruction is used. By the instruction *"receive message M from port j"* executed by processor u, the first message in the input queue for link e is transferred to the variable M (stored in u's local memory). If the input queue is empty, a special symbol is returned to M (acknowledgement).

Our algorithm is initiated at one of the processors named "*Leader Processor*" (*LP*) as shown in figure 1. A processor (*P*) at which the algorithm is not initiated gets involved with computations only after receiving a message from another processor. Processors can send/receive messages to/from processors that are only adjacent to it (i.e., connected by an edge). The algorithm proceeds as follows: Each active processor performs local computations if any and sends out messages to *LP*. We assume that any *P* can receive messages from neighbors at any instance. Thus, no messages are lost once they are delivered to a processor. Similarly, no messages are lost on any of the communication links and are guaranteed of delivery with an arbitrary but finite amount of time. Messages communicated over the same edge to the same destination are received in the order they are sent. The termination of an algorithm can be determined in one of two ways: (i)Each processor determines the termination depending on a local condition – may be the failure of the node, in which case it will not become active again or (ii)Termination is detected depending on global condition such as the completion of all local processing and the absence of messages in transit; a special termination detection step is used in this case. In this study we have selected a group of transactions that obey the second condition for termination.

4. Implementation methodology

Implementation requires generation of an "arbitrary" graph with a given number of node (n). More specifically, the first step in our problem is to generate (or select) a graph G at random. The problem is not entirely straightforward because a graph may be written down in (usually) many isomorphic forms and different graphs can have different numbers of isomorphism. Our goal is to reduce the execution time of a job through equitable distribution of workload among the processors in a distributed system.

4.1 Task Allocation

An important problem that arises in distributed computer systems is the *task allocation problem*. Many heuristic approaches that provide suboptimal solutions have been attempted in a number of studies. However, for practical problems, it is difficult to evaluate how accurate these solutions are, because efficient algorithms that have been studied are limited to very small sized problems. In this section we have presented an algorithm that may be extended to large-sized problems. It has been developed to solve the task allocation problem in a distributed computer system that meets the following specification

a) *The processors for message transmission use ideal communication links. This means that they are fault-free and bi-directional.*

b) *The capacities of processors and links are assumed to be unlimited.*

In case of a two-processor system it has been shown that a polynomial-time algorithm may find the optimal assignment very efficiently. However, for an arbitrary number of processors, the problem is known to be *NP– complete* [1,11].

4.2 Problem Model

Let P = {P_1, P_2,…., P_m} be the set of the *m* identical processors of the distributed system. A distributed process is defined as the set of tasks T = { T_1, T_2…. T_n} to be run on the distributed system .We assume that the communication cost between two tasks executed by the same processor is negligible. Let

Q_{tp} ($t \in \{1...n\}$, $p \in \{1...m\}$) be the execution cost of task T_t when it is assigned to processor P_p.

X_{ip} ($t \in \{1…n\}$, $p \in \{1…m\}$) be the decision Boolean variable

$$= \begin{cases} 1 \text{ if task } T_t \text{ is assigned to processor } P_p \\ 0, \text{ otherwise.} \end{cases}$$

In the problem model the constraints in this scheduling are *assignment constraints*, that is each task must be assigned to one and only one processor. Our purpose is to make the best use of resources in this distributed system based on FCFS scheduling. This means that for a given distributed process we have to minimize execution and communication costs. We also do not take into account precedence relationships among tasks.

4. 3 Control Abstraction of the Dynamic Scheduler

// Sched.H - The User Defined Header File

// *N* - the number of nodes in the network.

// *source* - The single origin where all the transactions are submitted.

// Input data files:

i. **DATAT×T**, where T×T is the set of the Transaction Set stored in transdat[ ][ ].

ii. **WTEDN×N** gives the Weighted Matrix[ ][ ] of the network .

iii. **ADJ N×N** used to generate the Weighted Matrix[ ][ ].

// sched.h includes the prototype for the CLASS TRANSACTION and the function *policy( )*. This function uses the Djikstra's algorithm for scheduling transactions.

// CONGESTION is taken to occur when all the nodes in the network are busy executing tasks.

// The Main program file is BUFF.CPP based on ***algorithm-1:Buff*** , which calls another program SCH.CPP based on ***algorithm-2:Sch***

ALGORITHM_1: *Buff*

- NRO – number of rows in the transaction table
- NCOL - number of columns in the transaction table
- VALMAX - Maximum time units that the server can schedule at a time. This value is taken to be 1000 TIME UNITS.
- TSCHED - Our assumption on the time for scheduling a task to the intended node. This value is fixed at 50 TIME UNITS.

CLASS BUFFER

```
{
private:
        int schedbuf[ ] ;  // The scheduling buffer, which contains the current
                              transactions at any instant of time.
        int transdat[ ];
        int totaltime ;    // The total time taken for scheduling transdat[ ][ ]
                              completely.
        int optnode ;      //  Optimal destination found out by policy( )
        int optcost ;      //  Optimal cost found out by policy( )
        int waiting ;      //  the waiting time for the scheduler either when
                              (a) The scheduling buffer is full or
                              (b) When CONGESTION occurs.
        int totwait ;          // the total time the scheduler SLEEPS.
        int large ;            // the last largest transaction in execution after scheduling of
                               all jobs in transdat[ ][ ] is completed
public:
        buf_read( ) ;    // To read values from DATATxT and WTEDNxN
        jobcomput( ) ; //  To check the time left out in each  transaction  at the
                           executing node.
        policy2( ) ;     // Our scheduling policy which is invoked  whenever
                              •  When any deadline at a BUSY node < = TSCHED or
                              •  When CONGESTION occurs.
        jobcal( );            // The core function where jobs are scheduled and the
                              number of tasks the scheduling buffer has to take at the next
                              iteration is determined. Here policy( ) and policy2() are
                              invoked.
        caltime( );           // calculates the totaltime.
        ccongestion_chk( );   // checks for CONGESTION in the network.
};  // CLASS BUFFER ends.
```

MAIN( )

```
{
1. Invoke buf_read( );
2. Invoke job_cal( );
        {
        while (ntransn < NRO * NCOL )
                {
                while( sum of values in the scheduling buffer <= VALMAX )
                        store (schedbuf[ ] ← transdat[ ]);

                        FOR (the first new transaction to the last transaction which has
                                arrived in the scheduling buffer , schedbuf[ ])
                        {
                        Invoke congestion_chk( );
                        If ( CONGESTION occurred)
                                {
                                call policy2( );
                                get the values of optnode and optcost;
                                increment totwait appropriately ;
                                }
                        else
                                {
                                call policy( ) from ALGO SCH.CPP;
                                get the values of optnode and optcost;
                                }
3. Send the current task to optnode.
4. Call jobcomput( ) for manipulating next iteration.
 }
} // END main while
5. Invoke caltime( );
       {calculate large ;
        totaltime = large+totwait+(number of tasks scheduled by policy()) * TSCHED
        }
} // end Algorithm_1: Buff
```

ALGORITHM_2: *Sch*

This program uses the Djikstra's algorithm[3,11] to find the shortest path to the node, which is not BUSY from the *source*. The resulting destination is stored at *optnode* and the associated cost at *optcost*. These are returned to the algorithm Buff whenever *policy( )* is called.

5. Simulation outcomes and discussion

The performance of the scheduler is assessed by plotting the execution times for different sizes of transaction sets to varying sizes. The network dimension is taken along the X-axis and the time units on the Y-axis. We have used three set of sample transaction for study, which runs on a network of 50 processors. The sample transaction models are in Table 1(a), 1(b) and 1(c).

Transaction set-I			
Transactions	*Sub-Transactions (Time Units)*		
	t1	**t2**	**t3**
T1	350	600	850
T2	400	250	900
T3	700	550	650

Table 1.1(a)

Transaction set-II					
Transactions	*Sub-Transactions (Time Units)*				
	t1	**t2**	**t3**	**t4**	**t5**
T1	250	400	750	300	0
T2	850	600	550	800	950
T3	350	450	750	650	450
T4	300	250	550	600	0
T5	400	500	650	0	0

Table 1(b)

Transaction set-III								
Transactions	*Sub-Transactions (time units)*							
	t1	t2	t3	T4	T5	t6	t7	t8
T1	650	700	600	300	550	250	400	0
T2	400	300	850	350	250	350	250	400
T3	250	450	350	650	500	250	450	600
T4	300	800	600	700	350	400	0	0
T5	550	700	750	250	850	0	0	0
T6	450	600	800	950	0	0	0	0
T7	250	350	850	400	350	0	0	0
T8	450	350	400	950	0	0	0	0

Table 1.1(c)

Table 1 Transactions along with sub-transactions and their expected execution times.

Simulation Results

The results were obtained after performing simulations for three categories of the problem addressed:

Category I: Determining the performance of the scheduler with all transactions in Table 1, without considering the cost of communication.

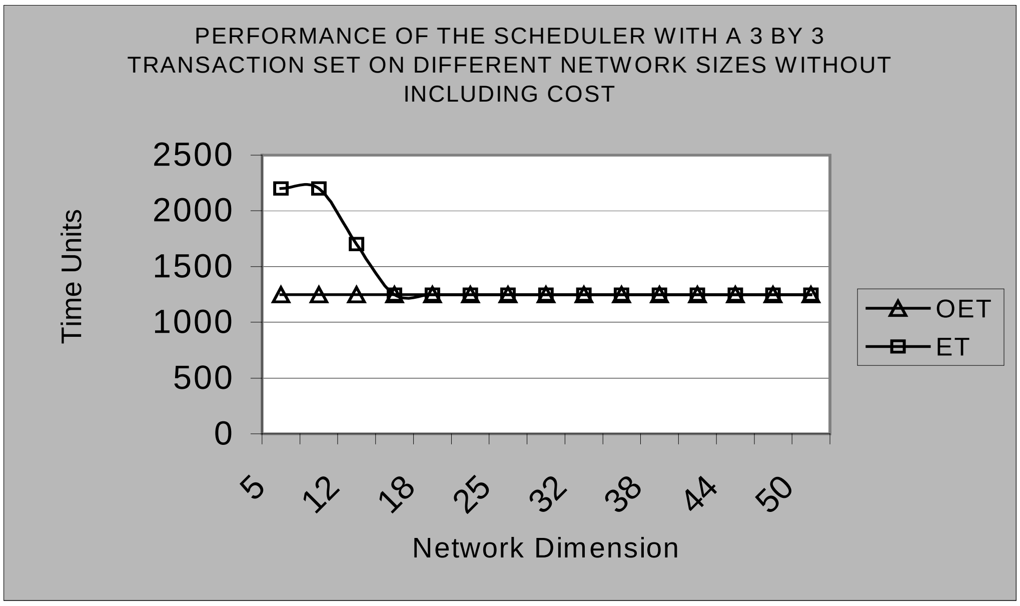

Figure 2 obtained with data in table 1.1(a)

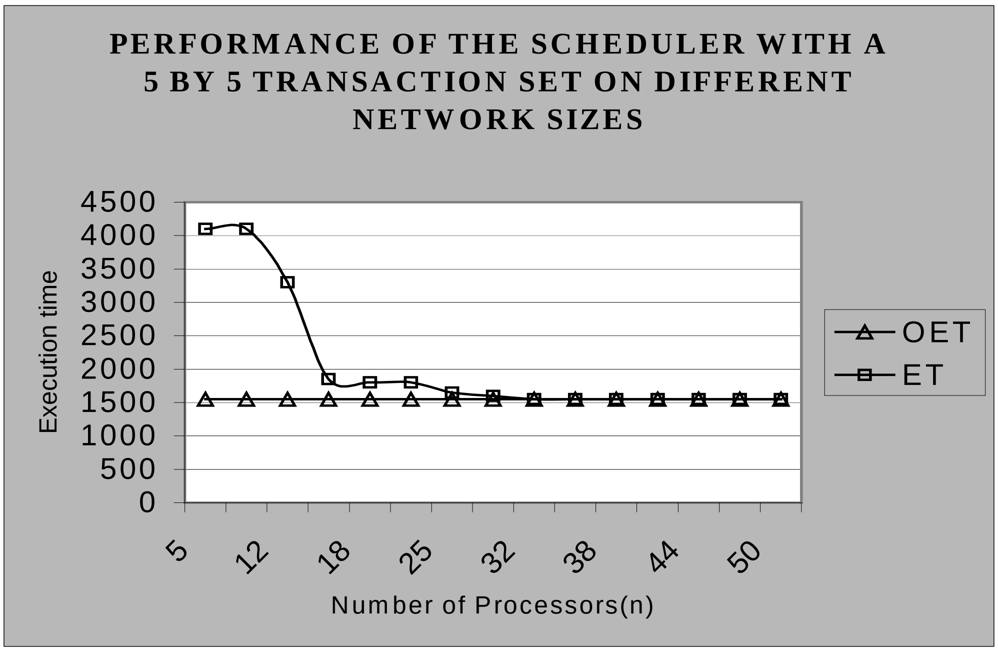

Figure 3 obtained with data in table 1.1(b)

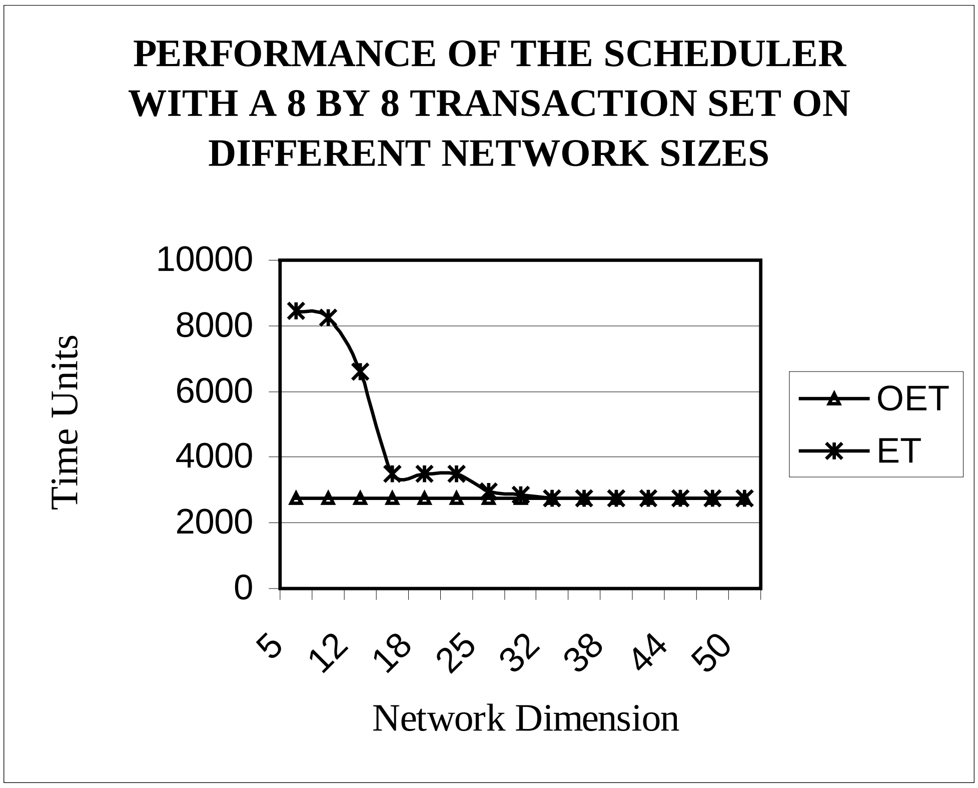

Figure 4 obtained with data in Table 1.1(c)

An "*Optimal Execution Time*" **(OET)** value is first calculated and used as the base for comparing the performance of the Scheduler. The Scheduler is then subjected to all three tables 1.1(a), 1.1(b) and 1.1 (c) and the "Execution Time" **(ET)** curve is obtained with varying network sizes as shown in figures 2,3 and 4. It is found that as the network dimension is increased the ET approaches OET.

Category II: Determining the performance of the scheduler after including the 'cost' variable. Here, an "*Average Execution Time*" **(AET)** value is calculated and used along with OET. Interestingly, we observe here those ET approaches **AET** and gradually move towards **OET** for considerably very large values of the network dimension. The curve has three distinct portions, (1)Portion A where the performance of the Scheduler is poor, (2) Portion B where the performance is average, and (3) Portion C where the performance is optimal. This optimal performance is attributed to our model where the nearest nodes are utilized continuously every time after they complete a task.

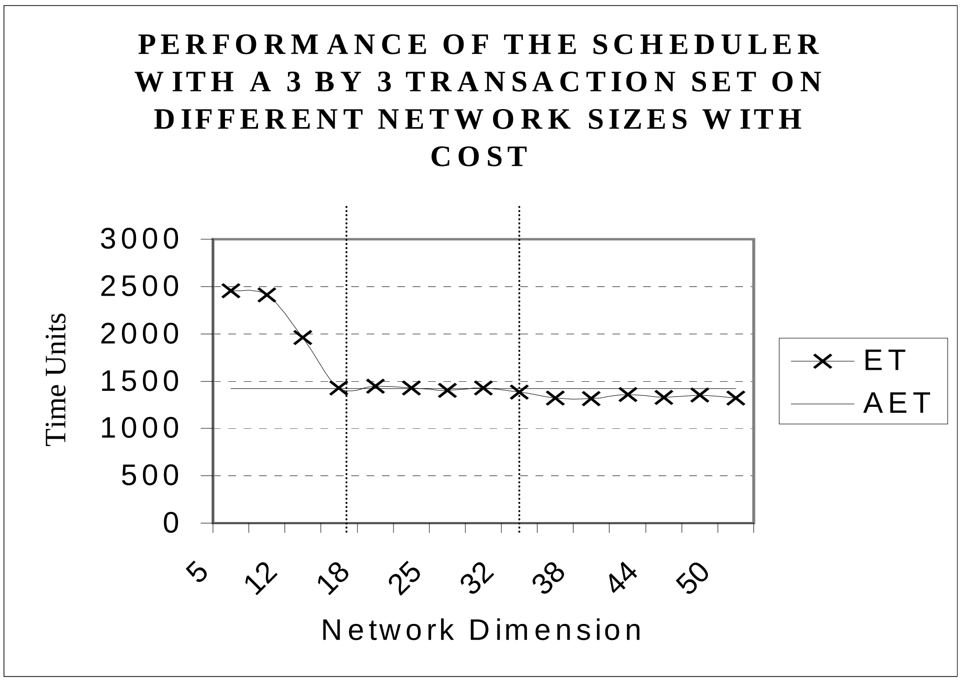

Figure 5 obtained with data in Table 1.1(a)

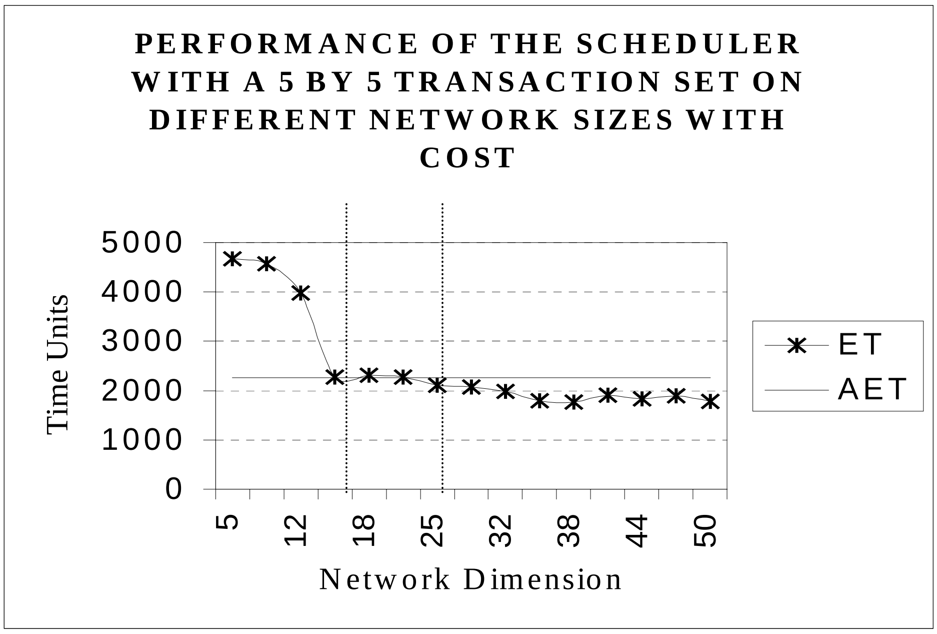

Figure 6 obtained with data in Table 1.1(b)

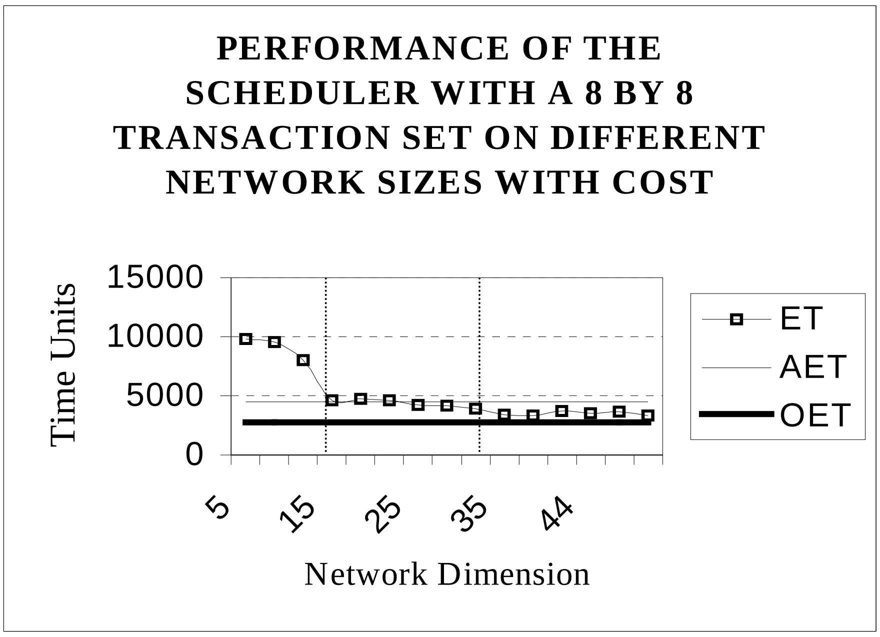

Figure 7 obtained with data in Table 1.1(c)

Category III: Comparison of the performance of the scheduler on different topologies obtained by decreasing the availability of the nodes from the *LP*.

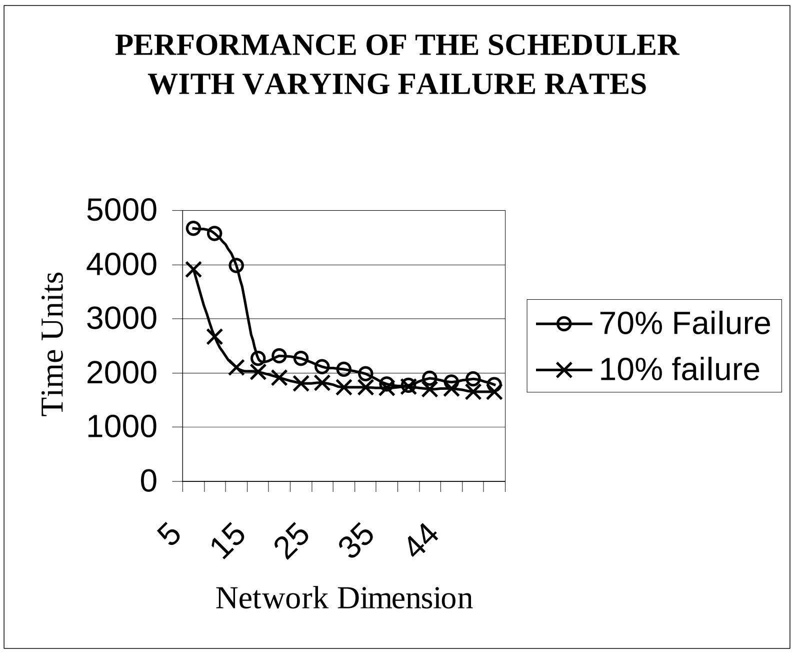

Figure 8 obtained with data in Table 1.1(b)

By increasing the failure of nodes accessible from *LP* from 10% to 70%, we plot two curves. As expected, the curve with the 10% failure rate reaches AET faster than the other. Also, the portion A of the curve with 10% failure rate has a lower starting value.

Work load of a *PE* at particular instant is defined as the total time needed to complete execution of all the tasks waiting at a PE at that instance (including the task, if any, that is currently being executed on the PE). This time includes the total computation time of all these tasks and overhead such as communication delay, synchronization delay and network queuing delay associated with the execution of the tasks. Here we have not taken into account these overhead problems. An attempt can be made further to find out the optimal number of processors that can be made available prior to the submission of a task for execution.

6. Conclusion

One of the major advantages of distributed systems over stand-alone systems is that balancing of workload of the system among the computers (nodes) can improve system performance. Although dynamic load-balancing strategies have the potential of performing better than static strategies, they are inevitably more complex. Their complexity and overhead may negate their benefits. Moreover, the same problem can be discussed on several distributed system models [9]. Selecting a set of transactions to simulate a distributed algorithm is a major factor. In this project distributed transactions of different dimensions were used for simulation. This scheduling policy also can be applicable to hierarchically clustered data networks [4]. The whole network can be treated as *Multi Origin Shortest Path Problem (MOSP),* which can be solved as a set of *Single Origin Shortest Path Problem (SOSP)* by applying the concept used in this work.

The performance of scheduling schemes can be optimized with the use of soft-computing paradigms like Genetic Algorithm [6], Evolutionary computation and Genetic Programming. So, we plan to retain the transaction model and the data sets used in this project and determine the behavior of the scheduler after replacing the Djikstra’s algorithm that has been used for scheduling, with an algorithm based on the Genetic Approach. We expect an improvement in the performance of the scheduler with the Genetic Approach and reduction of complexity in the scheduling decisions as compared to the simulated scheduling policies discussed in this paper.

7. References

[1] Krithi Rama Mritham, John A. Stankovic and Wei Zhao, “Distributed Scheduling of Tasks With Deadlines and Resource Requirements”, IEEE Transactions on Computers, Vol. 38, No. 8, August 1989, pp. 1110-1123.

[2] Kang G. Shin and Ming-Syan Chen, “On the Number of Acceptable Task Assignments in Distributed Computing Systems”, IEEE Transactions on Computers, Vol. 39, No. 1, January 1990, pp. 99-100.

[3] Nancy A. Lynch, “Distributed Algorithms”, Morgan Kaufmann Publications, 1996.

[4] Shan Zhu and Garng M. Huang “A New Parallel And Distributed Shortest Path Algorithm for Hierarchically Clustered Data Networks” IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 9, September1998, pp. 841-855.

[5] Masafumi Yamashita, and Tsunehiko Kameda, “Leader Election Problem On Networks in which Processor Identity Numbers Are Not Distinct”, IEEE Transactions On Parallel And Distributed Systems, Vol. 10, No. 9, September 1999, pp. 877-887.

[6] Albert Y. Zomaya, Chris Ward, and Ben Macey "Genetic Scheduling for Parallel Processor Systems Comparative Studies and Performance Issues", IEEE Transactions On Parallel And Distributed Systems, Vol. 10, No. 8, August 1999 pp. 795-813.

[7] Theodora A. Varvarigou, E. Anagnostou and Sudhir R. Ahuja, "Reconfiguration Models and Algorithms for Stateful Interactive Processes", IEEE Transactions on Software Engineering, Vol. 25, No. 3, May/June 1999, pp. 401-415.

[8] C. M. Chung, P. S. Shin and S. D. Kim, "An Effective Configuration Method for Java Internet Computing Environment", Parallel Processing letters, Vol. 10, No. 1, January 2000, pp. 74-86.

[9] Y. Zhang, H. Kameda, S.L. Hung, "Comparison of dynamic and static load balancing strategies in heterogeneous distributed systems", IEE Proc. on Computer and Digital Technique, Vol. 144, No. 2, March 1997, pp. 100-106.

[10] H. G. Rotithor, "Taxonomy of dynamic task scheduling schemes in distributed computing systems", IEE Proc. on Computer and Digital Technique, Vol. 141, No. 1, January 1994, pp. 1-10.

[11] Thomas H. Cormen, Charles E. Leisrson, and Ronald L. Rivest, "Algorithms", Prentice hall of India, 1998.