Jurnal Teknologi, bil. 25, Disember 1996 hlm. 69–79
©Universiti Teknologi Malaysia

TASK ALLOCATION IN A MULTIPROCESSOR SYSTEM USING FUZZY LOGIC

SHAHARUDDIN SALLEH
Department of Mathematics
BAHROM SANUGI
Research and Consultation Unit
HISHAMUDDIN JAMALUDDIN
Faculty of Mechanical Engineering
Universiti Teknologi Malaysia
Karung Berkunci 791
80990 Johor Bahru, Johor
Malaysia

Abstract. Task scheduling for multiprocessors is a job–sequencing problem generally classified as NP–complete or NP–hard. Optimal solutions to the problem using some well–known algorithms can only be obtained in some restricted cases. In most cases, however, this is not possible. Therefore, near–optimal solutions to the problem have been developed using heuristics. This paper proposes a new heuristic using fuzzy logic to achieve near optimum load balancing for the task allocation problem in a multiprocessor system. Task allocation is a restricted case of task scheduling where the tasks have no precedence relations with others and the priority order of execution is ignored. The tasks are assumed to be non–preemptable, have no execution deadlines and have no interprocessor communication. It is possible to apply the fuzzy concepts since the problems involved are difficult to model mathematically. Much of its power of fuzzy logic is derived from its ability to draw conclusion and generate responses based on incomplete and imprecise informations.

1.0 INTRODUCTION

A multiprocessor system is a system consisting of several processors, all of which share the same memory block, (Quinn [1], Lester [2]). Each processor is capable of executing instructions on its own but will have to share data and other information through this memory block. It can have access to this memory block directly and at a very fast rate. In some systems, the shared–memory block is partitioned into several modules in order to improve its performance and to avoid problems such as memory contention which results from congestion due to simultaneous access to the memory by the nodes.

Task scheduling in a multiprocessor system refers to the orderly mapping of tasks to a set of processors, or nodes, to satisfy all the system requirements and constraints. Task allocation is a restricted case of task scheduling where the orders of execution, or precedence relations between tasks, are not considered. Normally, for every task only one node is chosen to do the job. Therefore, all nodes in the system will compete for the job. Several criteria that determine the current states of the nodes are evaluated and the node that fulfils all the requirements to the highest level will be awarded the task.

Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX

Task scheduling and allocation for multiprocessor systems is a problem known generally to be either NP–hard or NP–complete. Several optimal solutions to these problems can be found in some restricted cases. In most cases, however, it is not possible to find their optimal solutions. Several heuristics that represent their near–optimal solutions have been developed. A well–written introductory literature on these problems can be found in, (El-Rewini [3]) which describes basic static and dynamic approaches such as the task graph, list scheduling, insertion, duplication and clustering techniques. In Rotithor [4], several dynamic scheduling techniques for both multiprocessor and multicomputer systems were evaluated and compared. These methods differ according to the system requirements and models. In Ramamritham [5], random scheduling, bidding, focused–addressing and flexible algorithms were proposed for hard real–time scheduling of both periodic and nonperiodic tasks in a loosely–coupled system. These algorithms take into consideration the deadlines and resource requirements of the hard real–time systems. In El Mouhamed [6], the least communication algorithm was proposed for the nonuniform memory access (NUMA) shared–memory system. While in Stone [7], the network flow algorithm was applied to find the minimum cutset in assigning tasks to nodes so as to achieve the least interprocessor communication.

Fuzzy logic is one of the most powerful tools for intelligent systems, (Kosko [8]) and has been found useful in solving problems that are difficult to model mathematically. Much of its power is derived from its ability to draw conclusion and generate responses based on vague, ambiguous, incomplete and imprecise qualitative data. Its mechanism is based on logical inference of rules in processing this non–numeric information to generate crisp or numeric output. Fuzzy logic has wide applications in the design of controllers, operations research, expert systems and help in various decision making processes.

Parallel and distributed computing has its applications in a wide area of real–time engineering problems. One particular instance is scheduling the robot inverse dynamics computation through mapping on a multiprocessor system, (Lee & Chen [9]). In summary, the problem is about the computation of the required generalised forces (torques) from an appropriate manipulator dynamics model using the Newton–Euler equation of motion on the measured displacement and velocity data. The mapping involves graph partitioning and scheduling to minimise the objective function which is defined in terms of the sum of the processor finishing time and the interprocessor communication time.

This paper presents some preliminary results obtained from using fuzzy logic to achieve a reasonably good load balancing for task allocation in a multiprocessor system. Fuzzy approach is possible due to the fact that a mathematical model is difficult to find and that the information gathered during an execution fits the description to such an approach.

2.0 CHARACTERISTICS OF TASK ALLOCATION PROBLEMS

Task scheduling for parallel and distributed systems is approached in two distinct ways, namely static and dynamic, and sometime as a hybrid of the two. Static scheduling is deterministic in nature in that it is computed off–line. All the characteristics of the tasks are known *a priori*, that is, before the execution begins. Therefore, its optimal or sub–optimal solutions may be obtained in a reasonably straight forward way through methods in graph theoritic, mathematical programming and heuristics. In dynamic scheduling, all this information is not known beforehand and has to be determined on the fly, that is, as the execution is in progress. The information is subject to unpredictable changes due to factors such as variable looping and branching in the program. This factor makes it more difficult to implement as it imposes a heavy overhead to the system. In most cases, heuristic approaches are proposed in its implementation as its optimal solutions are not possible. Because of this factor, dynamic scheduling finds its application in most real–time systems.

The performance of the system in task allocation is measured in several metrics, (Xu & Parnas [10]). Firstly, the system assigns tasks so as to minimise the schedule length, that is the optimal finishing time. Secondly, the system may want to allocate tasks according to the most feasible schedule by scheduling according to earliest-deadline first, least laxity and rate–monotonic. Thirdly, some algorithms try to schedule so as to minimise the number of processors in the system. This is necessary since an increase in the number of processors usually leads to a decrease in execution performance, (Quinn [1]). Finally, the system allocates the tasks as evenly as possible on nodes so that no nodes will be too idle while some others will be too overworked. This fourth performance measure is to achieve *load balancing* which is the subject of this paper.

3.0 THE SYSTEM MODEL

The main objective of the model is to perform task allocation with load balancing, that is, to achieve a fairly distributed cumulative execution time on all nodes. The computing model is assumed to be a simple multiprocessor system with the number of nodes selectable from 2 to 10. Each node has a very limited memory to process instructions and data, and will only communicate with others through an interconnection network to the shared–memory block. Communication between nodes, however, will not be considered for the time being. The model assumes that the shared–memory block stores the *global scheduler* which coordinates all task assignments to the nodes. The global scheduler serves as a *fuzzy scheduler* and, therefore, keeps all the information about the incoming tasks. In addition, each node has a *local scheduler* which represents the node in making bids and to receive further instructions from the global scheduler. All incoming tasks are randomly generated by the computer, are assumed to be independent, non–preemptable, have no execution deadline and have no precedence relations with one another. The only resources used by all nodes are their CPUs. In addition, all nodes are assumed to be homogeneous and execute at the same speed. A multiprocessor model with 10 processors is illustrated in Figure 1.

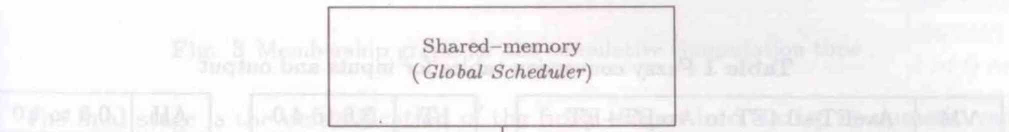

Fig. 1 The computing model

Each incoming task is characterised by its *execution time*, *ET*, which is the time needed to execute the task. When this new task arrives, the global scheduler obtains its information and conveys it to all local schedulers in the form of requests for bidding. Every local scheduler then responds by supplying two information regarding the state of its node. First is the *cumulative execution time* of all previous tasks already completed in the node, *Node[k].CumET* (k is the node number). Second is the offer by its node to the task on the idle time from the time the new task arrives to the time it can begin execution at its node, *Node*[k].*Idle*. This idle time depends on the node availability and is computed as the period from the time the node finishes a task to the time it can start executing a new task there. A value close to 0 means the task can start almost immediately at the node, while a bigger value means a longer waiting time before it can start. The *cumulative computation time, Node[k].CumCT* is the time taken by a node to complete all its jobs. The maximum Node[k].CumCT among the nodes gives the *total computation time*, which is the amount of time required by the system to complete all task assignments. Both, the task execution time ET and the node idle time Node[k].Idle are generated randomly by the computer in this model.

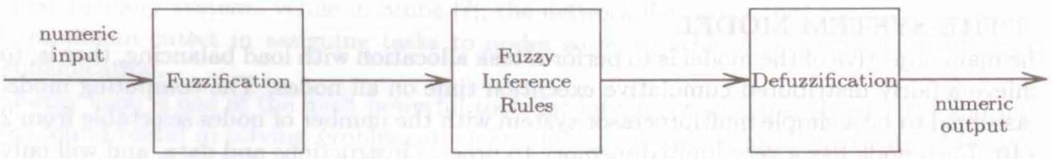

Fig. 2 The fuzzy scheduler

Scheduling of tasks using fuzzy logic involves three orderly steps as illustrated in Figure 2, namely fuzzification, the application of fuzzy inference rules and defuzzification. During the fuzzification process, the numeric input values are read and transformed into their corresponding fuzzy variables (or linguistics) based on a predefined set of rules, as shown in Table 1. These fuzzy inputs called *antecedents*, form their corresponding membership function graphs, usually in the form of a triangle. In classifying the fuzzy variables for the first input, VM, MM, AV, ML and VL, the average of Node[k].CumET is calculated first and is given by *AveET*. This average value is updated once at every arrival of a new task. The second antecedent is made up of four fuzzy variables, LT, AT, MT and VT, classified according to the value of Node[k].Idle.

Table 1 Fuzzy conversion tables for inputs and output

VM	AveET+0.4ET to AveET+ET
MM	AveET to AveET + 0.6ET
AV	AveET-0.2ET to AveET+ 0.2ET
ML	AveET-0.6ET to AveET
VL	AveET-ET to AveET-0.4ET

LT	3.6 to 4.0
AT	2.8 to 3.8
MT	2.4 to 3.2
VT	0 to 2.8

AH	0.6 to 1.0
AL	0.4 to 0.8
RL	0.2 to 0.5
RH	0 to 0.4

The second stage is applying the fuzzy inference rules in Table 2 to both antecedents to generate a *consequence*. Each rule is expressed as (ante1,ante2;consequence) which means *IF ante1 AND ante2 THEN consequence*. The consequence, or fuzzy output, is made up of four fuzzy variables, AH, AL, RL and RH which represent acceptance or rejection low/high, classified based on numeric values from 0 to 1, as shown in Table 1. A value close to 1 means the bidding node has a strong chance of being accepted while a decreasing value represents a weaker chance. The process involves the mapping of ante1 and ante2 to their respective membership degree values on their graphs. These degree values are compared and the minimum of the two is then projected onto the membership function of their consequence. The area between this value, the graph and the horizontal axis, usually in the shape of a trapezium, then represents the output of one inference rule.

Table 2 Fuzzy inference rules

	LT	AT	MT	VT
VM	RH	RH	RL	RL
MM	RH	RL	AL	AL
AV	RL	RL	AL	AH
ML	AL	AH	AH	AH
VL	AH	AH	AH	AH

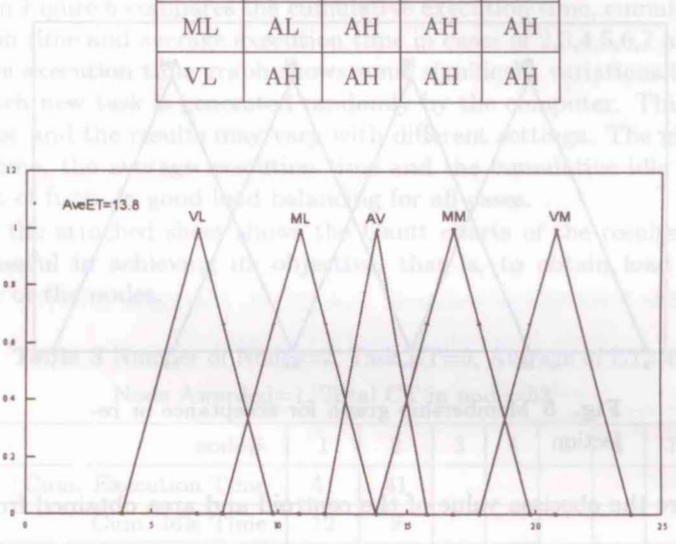

Fig. 3 Membership graph for the cumulative computation time

The final stage is the defuzzification of the fuzzy output into a crisp or numeric value. There are several defuzzification schemes and this model uses the most popular method called the centroid method. For each area generated by an inference rule, its centroid and area are calculated. This process is repeated for other inference rules where the inputs are applied to obtain an area of overlapped trapeziums. The abscissa value of the centroid of these overlapped areas, $\bar{x}$ is determined by

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i A_i}{\sum_{i=1}^{n} A_i} \tag{1}$$

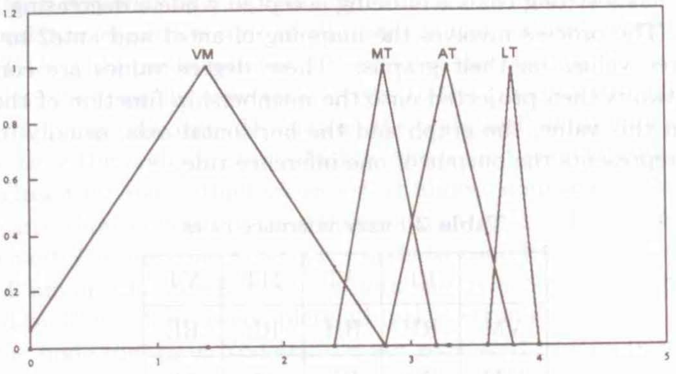

Fig. 4 Membership graph for the idle time before start

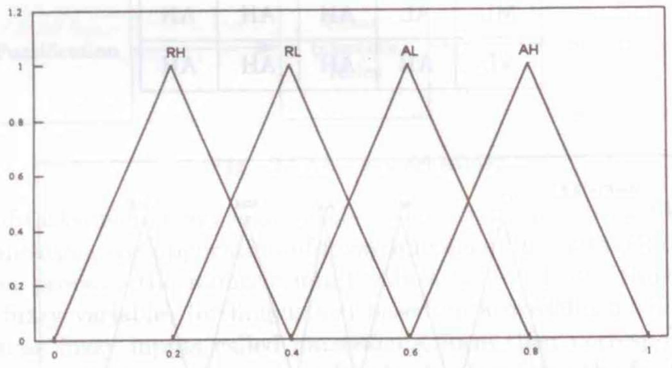

Fig. 5 Membership graph for acceptance or rejection

where $\bar{x}$ and A_i are the abscissa value of the centroid and area obtained from the ith rule respectively.

The defuzzification process generates a centroid value for every bidding node which ranges from 0 to 1. These centroid values are compared and the node with the maximum value is declared the winner. In the event that the maximum centroid values are the same in some nodes the award will be made to the node that has the least cumulative execution time. The global scheduler informs the local scheduler of the winning node and this node will receive the task. The process then repeats for the next task.

The algorithm for the task allocation problem is summarised as follows:

Step 1: Fuzzification

Read inputs Node[k].CumET,Node[k].Idle

Transform these crisp inputs into their fuzzy sets for relations using Table 1

Determine their degrees from the membership functions of Figures 3,4

Step 2: Applying The Inference Rules

Find the minimum degree value from Step 1

Project this value onto their consequence relation graph using Table 2, Figure 5

Step 3: Defuzzification
Find the centroid and area of the trapezium formed

Step 4: Repeat Steps 1,2,3 for other relations using the same inputs

Step 5: Find the final centroid of all overlapping areas using formula (1)

Step 6: Award the task to the node with the maximum centroid value in Step 5

4.0 SIMULATION RESULTS AND ANALYSIS

A multiprocessor system model with 2 to 8 nodes is simulated on an Intel-80486 computer. The program TS6.EXE, written in C++ runs the simulation and generates results for the allocation of 15 tasks, as shown in Tables 3 to 9. Entries in each table show quite a fairly distributed cumulative execution time. These results are made possible because of the fuzzy effect which assigns new tasks with high priorities to nodes with low cumulative execution time and high idle time.

The graph in Figure 6 compares the cumulative execution time, cumulative idle time, total computation time and average execution time in cases of 2,3,4,5,6,7 and 8 node models. The cumulative execution time graph shows some significant variations because the execution time of each new task is generated randomly by the computer. This graph represents one specific case and the results may vary with different settings. The graphs for the total computation time, the average execution time and the cumulative idle time of the nodes show the effect of fuzzy in good load balancing for all cases.

Figure 7 in the attached sheet shows the Gantt charts of the results. In all cases, the model is successful in achieving its objective, that is, to obtain load balancing on the execution time of the nodes.

Table 3 Number of Nodes=2, Task ET=5, Average of ET=41.0
Node Awarded=1, Total CT in node=53

node#	1	2	3	4	5	6	7	8
Cum. Execution Time	41	41						
Cum. Idle Time	12	9						
Cum. Computation Time	53	50						
Number of Tasks	7	8						
Centroid Value	0.8	0.6						

Table 4 Number of Nodes=3, Task ET=1, Average of ET=18.0
Node Awarded=1, Total CT in node=30

node#	1	2	3	4	5	6	7	8
Cum. Execution Time	14	20	20					
Cum. Idle Time	11	10	6					
Cum. Computation Time	25	30	26					
Number of Tasks	6	6	3					
Centroid Value	0.8	0.6	0.6					

Table 5 Number of Nodes=4, Task ET=5, Average of ET=20.5

Node Awarded=2, Total CT in node=29

node#	1	2	3	4	5	6	7	8
Cum. Execution Time	16	23	24	19				
Cum. Idle Time	8	6	2	4				
Cum. Computation Time	24	29	26	23				
Number of Tasks	5	4	3	3				
Centroid Value	0.8	0.8	0.52	0.8				

Table 6 Number of Nodes=5, Task ET=2, Average of ET=13.2

Node Awarded=5, Total CT in node=22

node#	1	2	3	4	5	6	7	8
Cum. Execution Time	13	14	14	14	11			
Cum. Idle Time	1	2	8	2	7			
Cum. Computation Time	14	16	22	16	18			
Number of Tasks	3	2	4	2	4			
Centroid Value	0.701	0.7	0.7	0.7	0.8			

Table 7 Number of Nodes=6, Task ET=6, Average of ET=15.5

Node Awarded=3, Total CT in node=22

node#	1	2	3	4	5	6	7	8
Cum. Execution Time	17	14	20	10	16	16		
Cum. Idle Time	2	5	2	4	6	4		
Cum. Computation Time	19	19	22	14	22	20		
Number of Tasks	2	2	4	2	3	2		
Centroid Value	0.6	0.8	0.8	0.8	0.673	0.673		

Table 8 Number of Nodes=7, Task ET=5, Average of ET=10.9

Node Awarded=7, Total CT in node=19

node#	1	2	3	4	5	6	7	8
Cum. Execution Time	8	6	16	7	13	12	14	
Cum. Idle Time	2	3	2	2	6	7	3	
Cum. Computation Time	10	9	18	9	19	19	17	
Number of Tasks	2	1	2	2	2	4	2	
Centroid Value	0.8	0.8	0.42	0.8	0.6	0.7	0.8	

Table 9 Number of Nodes=8, Task ET=9, Average of ET=9.8
Node Awarded=8, Total CT in node=18

node#	1	2	3	4	5	6	7	8
Cum. Execution Time	7	11	9	7	8	13	9	14
Cum. Idle Time	6	5	2	5	1	2	2	4
Cum. Computation Time	13	16	11	12	9	15	11	18
Number of Tasks	2	2	2	2	1	3	1	2
Centroid Value	0.737	0.6	0.7	0.737	0.8	0.5	0.754	0.8

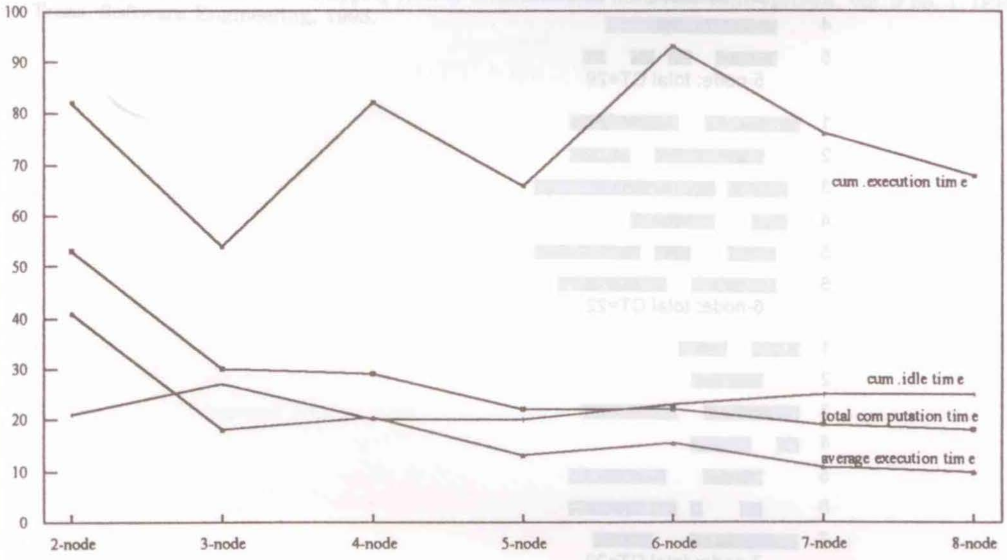

Fig. 6 Comparison on performances based on the number of nodes

5.0 CONCLUSIONS

As illustrated in this work, fuzzy logic has a tremendous potential in solving the task allocation and scheduling problems. Initially, the work reports on its use in achieving load balancing which proves to be successful although the method is restricted to cases where the tasks are independent, non-preemptable, have no precedence relations among them and have no execution deadline, as well as using CPUs as the only resources with no interprocessor communications. A more comprehensive model that includes other real-time requirements will be developed in this research later.

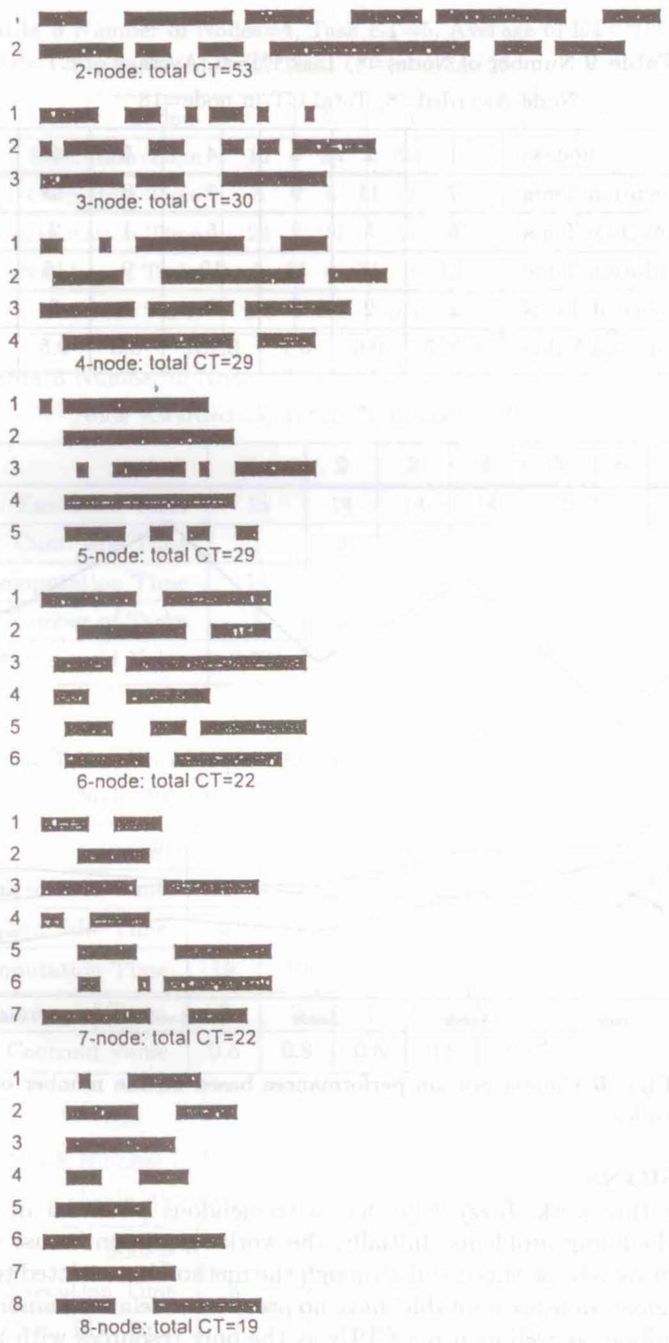

Fig. 7 Gantt chart for cases of 2,3,4,5,6,7 and 8 nodes

REFERENCES

[1] M. J. Quinn, *Parallel computing: theory and practice*, McGraw-Hill, New York, 1994.

[2] B. P. Lester, *The art of parallel programming*, Prentice-Hall, New York, 1993.

[3] H. El-Rewini, T. G. Lewis & H. Ali, *Task scheduling in parallel and distributed computing*, Prentice-Hall, New York, 1994.

[4] H. G. Rotithor, *Taxonomy of dynamic task scheduling schemes in distributed computing systems*, vol. 141 no. 1, IEE Proc. Comput. Digit. Tech., January 1994.

[5] K. Ramamritham, J. A. Stankovic & W. Zhao, *Distributed scheduling of tasks with deadlines and resource requirements*, vol. 38 no. 8, IEEE Trans. Computers, 1989.

[6] M. Al-Mouhamed, *Analysis of macro-data flow dynamic scheduling of nonuniform memory access architectures*, vol. 4 no. 8, IEEE Trans. Parallel and Dist. Systems, 1993.

[7] H. S. Stone, *Multiprocessor scheduling with the aid of network flow algorithms*, vol. 3 no. 1, IEEE Trans. Software Engineering, 1977.

[8] B. Kosko, *Neural networks and fuzzy systems: a dynamical system approach to machine intelligence*, Prentice-Hall, New York, 1992.

[9] C. S. G. Lee & C. L. Chen, *Efficient mapping algorithms for scheduling robot inverse dynamics computation on a multiprocessor system*, vol. 20 no. 3, IEEE Trans. Systems, Man and Cybernetics, 1990.

[10] J. Xu & D. L. Parnas, *On satisfying timing constraints in hard real–time systems*, vol. 9 no. 1, IEEE Trans. Software Engineering, 1993.