Matematika, 1999, Jilid 15, bil. 2, hlm. 95–109

Fuzzy Logic Model for Dynamic Multiprocessor Scheduling

Shaharuddin Salleh
Department of Mathematics
Faculty of Science
Universiti Teknologi Malaysia
81310 UTM Skudai, Johor, Malaysia

Bahrom Sanugi
Department of Mathematics
Faculty of Science
Universiti Teknologi Malaysia
81310 UTM Skudai, Johor, Malaysia

Hishamuddin Jamaluddin
Department of Applied Mechanics
Faculty of Mechanical Engineering
Universiti Teknologi Malaysia
81310 UTM Skudai, Johor, Malaysia

Abstract In this paper, we propose a dynamic task scheduling technique based on fuzzy logic. The main objective of the work is to implement load balancing in scheduling tasks on a network of processing elements. The fuzzy engine we propose is capable of processing inputs from incomplete and ambiguous data that arises from the current state of the processors. In the model, an arriving task is placed in a central queue based on the first-come-first-serve rule. When the task is ready to be assigned, its information is passed to the processors for bidding. One processor acts as the global scheduler to monitor the overall activities, while all others have local schedulers for managing their own activities. The latter supplies information on its current state and follows whatever decision given by the former. The two components work together and the global scheduler uses the fuzzy logic mechanism in making decision on the task assignment. Our experimental work shows promising results in achieving the objective.

Keywords Load balancing, fuzzy logic, task scheduling, multiprocessor and transputer.

Abstrak *Dalam artikel ini, satu teknik penjadualan kerja secara dinamik menggunakan logik fuzi dicadangkan. Objektif utamanya ialah untuk mendapatkan pengseimbangan beban dalam proses penjadualan kerja di dalam satu rangkaian yang terdiri daripada beberapa pemproses. Enjin fuzi ini berupaya memproses input daripada data-data yang tidak jelas atau tidak tentu yang terbit daripada keadaan semasa pemproses-pemproses. Dalam model ini, kerja yang baru tiba beratur menunggu giliran dalam satu barisan pusat berasaskan peraturan sampai-dahulu-didahulukan. Apabila kerja ini sampai gilirannya, maklumat semasanya dihantar ke pemproses-pemproses untuk dinilai. Sebuah pemproses berfungsi sebagai penjadual hakiki untuk mengawal aktiviti-aktiviti keseluruhan, sementara yang lain-lain sebagai penjadual tempatan untuk mengurus aktiviti pada pemprosesnya. Penjadual tempatan sentiasa memberi maklumat semasanya dan menurut perintah daripada penjadual hakiki. Kedua-dua komponen ini berkerjasama dalam menjayakan aktiviti-aktiviti penjadualan berasaskan mekanisma logik fuzi. Beberapa keputusan menggalakkan untuk mencapai objektif berjaya dihasilkan melalui model ini.*

Katakunci *Pengseimbangan kerja, logik fuzi, penjadualan kerja, multipemproses dan transputer.*

1 Fuzzy Logic Background

Fuzzy logic, as described by Zadeh [9,10] and Kosko [4], is one of the most powerful tools for designing autonomous intelligent systems. It has been found to be useful in solving problems that are difficult to model mathematically. Much of the power of fuzzy logic is derived from its ability to draw conclusion and generate responses based on vague, ambiguous, incomplete, and imprecise qualitative data. The mechanism is based on logical inference of rules in processing non-numeric information to generate crisp or numeric output. Fuzzy logic has a wide range of applications, for example, in the design of control systems and in various decision making processes.

Fuzzy logic contributions in the area of information technology could be in the form of approximate reasoning, where it provides decision-support and expert systems with powerful reasoning capabilities bound by a minimum number of rules. Theoretically, fuzzy logic is a method for representing analog processes, or natural phenomena that are difficult to model mathematically on a digital computer. The processes are continuous in nature, and are not easily broken down into discrete segments. With a reliable set of inference rules, inputs are converted into their fuzzy representations during the *fuzzification* process, and the output generated is then converted back into the "crisp", or numerically precise solutions during the *defuzzification* process. The rules that determine the fuzzy and crisp predicates for both the fuzzification and the defuzzification processes are constructed from sources like past history, neural network/neuro-fuzzy training and numerical approximation.

This paper presents a dynamic scheduling model based on fuzzy logic for parallel processing systems. Fuzzy logic provides a powerful tool for representing the solution space of the problem that arises from the imprecise information of its input derived from the current states of both the processing elements (PEs) and the arriving tasks. This information can be interpreted by the fuzzy engine which performs the analysis and then makes decision

on the assignment of the tasks to the PEs. Section 2 describes the nature of the dynamic scheduling problem in general. Section 3 describes our model which includes the precedence relationship of the tasks as the constraint to the problem. The performance of the model is evaluated through the experimental work in Section 4. Section 5 describes the implementation of fuzzy scheduling models on several transputer network models and, finally, Section 6 gives the summary and conclusion.

2 The Dynamic Task Scheduling Problem

Task scheduling, as described in El-Rewini et al. [3], 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. Dynamic scheduling is a form of task scheduling caused by the nondeterminism factor 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. Dynamic scheduling is often associated with real-time scheduling that involves periodic tasks and tasks with critical deadlines.

The main objective in dynamic scheduling is usually to meet the timing constraints, and perform load balancing, or a fair distribution of tasks on the PEs. Load balancing improves the system performance by reducing the average job response time of the tasks. In Lin and Raghavendra [5], load balancing involves three 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 job and whether or not to transfer a job. Third, is the *location rule* which chooses the PEs to and from which jobs will be transferred. It has been shown by several researchers that with the right policy to govern these rules, a good load balancing may be achieved.

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-come-first-serve (FCFS) technique, one PE runs a scheduler that dispatches tasks on a first-in-first-out basis to all other PEs. Each dispatched 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 scheduled queue maintained by the scheduler. This technique aims at balancing the load among the PEs and it does not consider contraints such as communication overhead. Chow and Kohler [2] proposed a queueing model where an arriving job is routed by a job 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 job turnaround time by balancing the total load among the PEs. In [2], a central job 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 Saletore [6], consider communication issues in balancing the load on all PEs. The approach adds balance constraint to the FCFS

technique by periodically shifting waiting tasks from one waiting queue to another. This technique performs local optimization by applying the steepest-descent algorithm to find the minimum execution time. The *cost-constraint* heuristic in [6] further improves the load balancing performance by checking the uneven communication cost and quantify them as the time needed to perform communication.

3 Fuzzy Model for Dynamic Task Scheduling

Task scheduling involves a difficult factor as the main constraint: the precedence relationships between the tasks. Task scheduling applications can be found in many areas, for example, in real-time control of robot manipulators, flexible manufacturing systems, and traffic control [4]. In this section, a fuzzy logic model for dynamic task scheduling that performs load balancing is proposed. This section describes the computing system and the proposed approach to deal with task precedence constraints in dynamic scheduling.

Our earlier model (Salleh and Zomaya [8]) is a static model based on fuzzy logic that takes imprecise information for its input derived from the current states of both the PEs and the arriving tasks. This information is passed to the fuzzy engine which performs the analysis and then makes decisions on the assignment of tasks to the PEs. The present approach modifies this model to handle the more challenging dynamic environment.

3.1 Model and Relevant Terminology

The computing platform for simulating dynamic task scheduling assumes a multiprocessor system with K fully-connected PEs [7,8]. A suitable realization for this model is the message-passing transputer-based system where each transputer represents a processing element with a processor and a memory module each, and has communication links with other transputers.

We provide some terminology to be used throughout this paper. The task scheduling problem evolves from the need to map J tasks TS_j for $j = 1, 2, \ldots J$ from a parallel program optimally onto a target machine, which consists of K processing elements PE_k for $k = 1, 2, \ldots K$, connected in a network. Each task TS_j, represented in the set $\mathrm{TS} = \{\mathrm{TS}_j \mid j = 1, 2, \ldots, J\}$, is a sequential unit of work in a program whose size can be as small as one line of code and up to the size of a single function or procedure.

The processing elements (or processors) are represented as the set $\mathrm{PE} = \{\mathrm{PE}_k \mid k = 1, 2, \ldots, K\}$, which are connected in a network with an arbitrary interconnection configuration represented by the matrix $\tilde{p}$. The *length* (size) of TS_j, denoted as l_{jk}, is defined as the elapsed time for the execution of the task sequentially on PE_k. This length is also referred to as the task *execution time* or the task *worst-case computation time.* The value of l_{jk} for TS_j depends on the processing speed of the processor PE_k in use, and may vary on different processors. The task TS_j initiated at time $t = \mathrm{TS}_j.\mathrm{at}$ is said to have arrived at that time. This task is not immediately executed as it has to wait in a queue. The *actual start time* for the execution of TS_j is denoted $\mathrm{TS}_j.\mathrm{ast}$, while its *completion time* is $\mathrm{TS}_j.\mathrm{ct}$. For the execution in PE_k, we obtain $l_{jk} = \mathrm{TS}_j.\mathrm{ct} - \mathrm{TS}_j.\mathrm{ast}$.

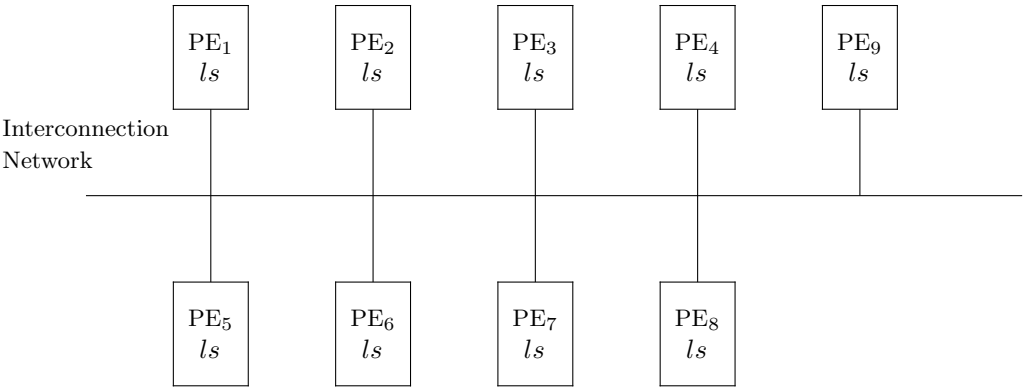

Figure 1. The Fully-Connected Computing System Model with 9 PE's

Figure 1 shows a network of fully-connected computing system $\Pi = \{\text{PE}, \tilde{p}\}$ with 9 processing elements labeled as PE_k for $k = 1, 2, \ldots, 9$. The interconnection matrix $\tilde{p}$ has entries $p_{ij} = 1$ if PE_l and PE_j are adjacent, and 0 otherwise. Each processing element in the network has its own memory to process instructions and data. All PEs in the computing model are also assumed to be homogeneous with the same execution speed. This assumption implies that l_j is used instead of l_{jk} for the task length.

In this work, we assume the incoming tasks are randomly generated, independent, non-preemptive and have some communication requirements with other tasks. Besides, the tasks are characterized with random arrival times but have no execution deadlines. These task characteristics can be summarized as $\Gamma = \{\text{TS}, \tilde{c}\}$ where

$$\text{TS} = \{l_j, \text{TS}_j.\text{at}, \text{TS}_j.\text{rt}, \text{TS}_j.\text{ast}, \text{TS}_j.\text{ct}, \text{TS}_j.\text{st} \mid j = 1, 2, \ldots, J\} \tag{1}$$

Our scheduling model in [7] consists of a FCFS central queue for newly arrived tasks and a host of PEs (servers) to receive these tasks, as shown in Figure 2. The queue is based on Poisson distribution on the M/M/K Markovian model with K servers and a mean arrival rate of λ. A newly arrived task enters the queue and waits for its turn to be assigned. This task is dispatched by the scheduler to a PE with the probability p_k based on the adopted scheduling policy. Each PE_k for $k = 1, 2, \ldots, K$, processes tasks assigned to it at a mean service rate of μ_k according to an exponential distribution.

As the name suggests, the assignment of a task is made based on competition among the PEs. The model stipulates that one PE acts as the controller which stores the *global scheduler* while all other PEs each have a *local scheduler*. Figure 1 illustrates this concept where the controller is PE9. The controller PE receives and stores all information about the incoming tasks, processes them and then makes decision on their assignments to other PEs in the network. The local scheduler in each PE manages the processor resources, supplies information on the state of the processor to the global scheduler, receives instructions from the global scheduler on when to execute an assigned task and keeps information on all its assigned tasks.

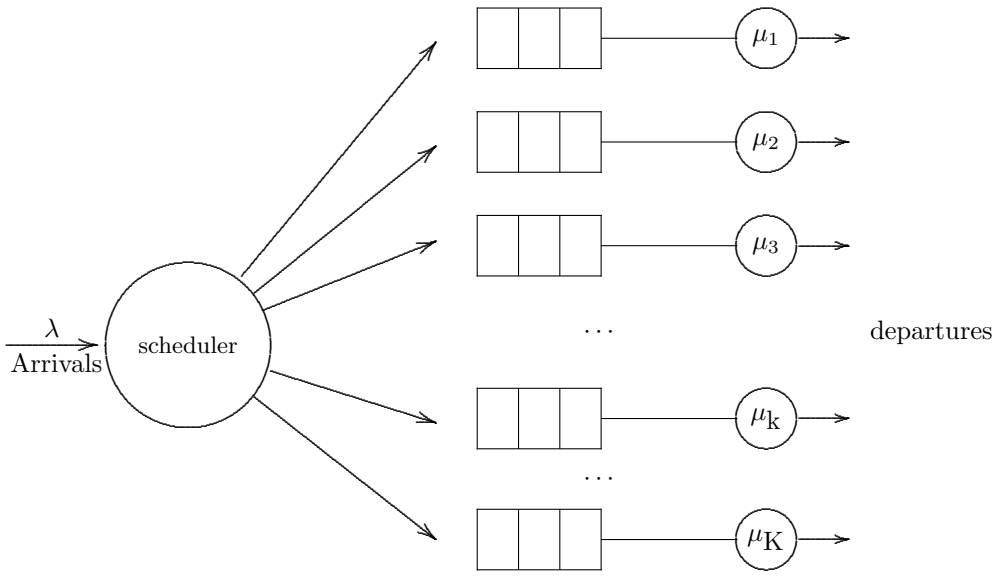

Figure 2. The M/M/K Queueing Model

When a new task arrives, the controller PE obtains the task's information and puts the task in the central queue. This task is not immediately assigned unless the queue is empty. The task at the front of the FCFS queue is selected for assignment. The controller PE broadcasts its information to all other PEs in the network requesting the bidding of this task. Every local scheduler then responds by supplying two pieces of information regarding the state of its PE. First, is the *processor execution length* PE_k.pel, which is the length of all the completed tasks in PEk defined as follows:

$$\mathrm{PE}_k.\mathrm{pel} = \sum_{j=l}^{J} l_{jk} = \sum_{j=1}^{J} s_{jk} l_j \tag{2}$$

where $k = 1, 2, \ldots, K$ is the PE number and $l_{jk} = 0$ if $\mathrm{TS}_j \notin \mathrm{PE}_k$. In the above equation, s_{jk} is a parameter which determines whether the task TS_j belongs to PE_k or not, defined as follows:

$$s_{jk} = \begin{cases} 1 & \text{if } \mathrm{TS}_j \in \mathrm{PE}_k \\ 0 & \text{if } \mathrm{TS}_j \notin \mathrm{PE}_k \end{cases} \tag{3}$$

Second, is the *processor delay time,* PE_k.del, which is the waiting time before TS_j can start executing. This waiting time is the delay caused by

1. The dynamic nature of scheduling which adds overhead to the system by delaying some processes.

2. The precedence relationship between the offered task TS_j, and its predecessors TS_i. If both TS_i and $\mathrm{TS}_j \in \mathrm{PE}_k$ then $\mathrm{PE}_k.\mathrm{del} = \mathrm{TS}_j.\mathrm{lrt} - \mathrm{TS}_{Lk}.\mathrm{ct}$, otherwise the method in the next section is used to determine this value.

3. The unsuccessful bidding by PE_k on the task prior to TS_j, which creates an unnecessary waiting time before another offer is received.

4. The state of PE_k, whether it is busy executing another task or not, at the required time.

For $PE_k.del \geq 0$, a value close to 0 means the task can start executing almost immediately at the PE, while a larger value means a longer waiting time before it can actually start. We now discuss the method to determine the value of PE_k.del.

3.2 Task Precedence Relationship

Every partial order between any two tasks incurs a significant delay due to the precedence rule, that a task cannot start executing until its predecessors have completed their execution, and synchronization for data transfer. A scheduled task will not be able to start executing until it has received all the required data from its predecessors. This means that the value of PE_k.del is expected to be significant depending on the network platform. Therefore, it is expected that the simulation will generate a schedule with both high PE_k.pct and PE_k.pil for all $k = 1, 2, \ldots, K$.

Figure 3 shows the chronological movement of $TS_j \in PE_k$ from the time it arrives at time $t = t_0$ to its completion at $t = t_3$. The task is ready for assignment at $t = t_1$ but it still has to wait until $t = t_2$ before starting its execution at an assigned processor.

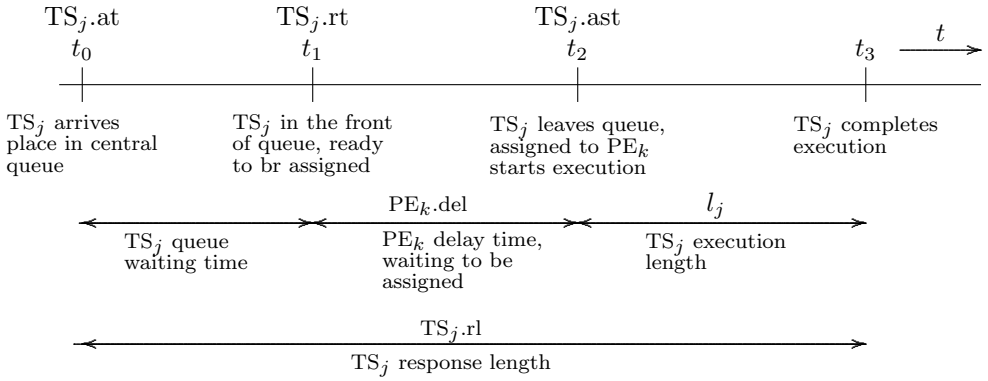

Figure 3. Movement of Task $TS_j \in PE$

In the FCFS queue of our model, it is assumed that a task arrives at the time no earlier than the arrival time of all its predecessors. This is necessary to guarantee the successful completion of all tasks and also to avoid any waiting period at the PE. The execution status of a task at time t is denoted as TS_j.st. A task TS_j is said to be *ready* to be assigned to a processor if it has received all the required data from the predecessor tasks. The *ready time* TS_j.rt for TS_j is defined as the earliest time TS_j can be assigned to any available processor. We further define the ready time for TS_j as either high ready time or low ready time. The *high ready time* TS_j.hrt of a ready task TS_j is the highest value of the sum of the predecessor task energy and its communication to TS_j. The *low ready time* TS_j.lrt is the next highest value. The processor with TS_j.hrt may skip some or all of the communication

cost from its latest predecessor to the incoming task to enable it to start executing the incoming task at the time $t = \mathrm{TS}_j.\mathrm{lrt}$. For other processors, the earliest time they can execute the task TS_j is at $t = \mathrm{TS}_j.\mathrm{hrt}$. The following relationships can be made

$$\begin{aligned} \mathrm{TS}_j \text{ high ready time} &= \mathrm{TS}_j.\mathrm{hrt} = \max_{i=1}^{J} \{\delta_{ij}\mathrm{TS}_i.\mathrm{ct} + c_{ij}\} \\ \mathrm{TS}_j \text{ low ready time} &= \mathrm{TS}_j.\mathrm{lrt} = \max_{i\neq 1}^{J} \{\delta_{ij}\mathrm{TS}_i.\mathrm{ct} + c_{ij}\} \end{aligned} \tag{4}$$

for $i, k = 1, 2, \ldots, J$. In the above equation, $\tilde{c} = \{c_{ij}\}$ is the communication matrix with positive values if $\mathrm{TS}_i < \mathrm{TS}_j$, and $c_{ij} = 0$ otherwise. The parameter δ_{ij} represents the partial order $\mathrm{TS}_i < \mathrm{TS}_j$ defined as follows

$$\delta_{ij} = \begin{cases} 1 & \text{if } \mathrm{TS}_i < \mathrm{TS}_j \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

In addition, we define the processor ready time $\mathrm{PE}_k.\mathrm{prt}$ as the earliest time PE_k becomes available and is ready to accept TS_j. The value of PEk.prt is determined as follows

$$\mathrm{PE}_k.\mathrm{prt} = \begin{cases} \max\{\mathrm{TS}_{Lk}.\mathrm{ct}, \mathrm{TS}_j.\mathrm{hrt}\} & \text{if } k \neq s \\ \max\{\mathrm{TS}_{Lk}.\mathrm{ct}, \mathrm{TS}_j.\mathrm{hrt}\} & \text{if } k = s \end{cases} \tag{6}$$

where TS_{Lk} denotes the latest task executed or is still executing in PE_k and s is the subscript of the assigned PE. It follows that $\mathrm{TS}_j.\mathrm{rt} = \mathrm{PE}_s.\mathrm{prt}$ and $\mathrm{TS}_j.\mathrm{ast} \geq \mathrm{PE}_s.\mathrm{prt}$ for PE_s.

The above method can be used to determine $\mathrm{PE}_k.\mathrm{del}$ in a static manner, that is, if the values of $\mathrm{T}_j.\mathrm{lrt}$, $\mathrm{TS}_j.\mathrm{hrt}$ and $\mathrm{PE}_k.\mathrm{prt}$ are all known beforehand. In dynamic scheduling, this information is not always available at the time TS_j is ready to be assigned. In dynamic scheduling, $\mathrm{PE}_k.\mathrm{del}$ has to be evaluated first before the start time for TS_j is determined. We now discuss a method to evaluate $\mathrm{PE}_k.\mathrm{del}$.

3.3 The Fuzzy Scheduler

Scheduling of tasks using fuzzy logic involves three orderly steps [4,9,10], as illustrated in Figure 4, namely, the *fuzzification* of the input variables, the application of *fuzzy inference rules* and the *defuzzification* of the results. 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. These fuzzy inputs, called *antecedents,* form their corresponding membership function graphs, which are commonly represented as triangles.

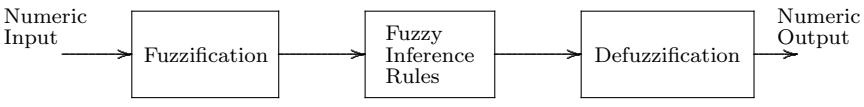

Figure 4. Structure of the Fuzzy Scheduler

The fuzzy variables for the first input $\mathrm{PE}_k.\mathrm{pel}$ are denoted as E1, E2, E3, E4 and E5. The real values that correspond to this antecedent are user defined and can be changed

interactively at runtime during the simulation. The mean of the processor execution length, denoted as PE_k.pel or μ_{pel}, is defined as follows

$$\mu_{\mathrm{pel}} = \frac{1}{K} \sum_{k=1}^{K} \mathrm{PE}_k.\mathrm{pel} \tag{7}$$

This mean value is updated every time a new task arrives to represent changes in PE_k.pel for $k = 1, 2, \ldots, K$. Therefore, the variables in the first antecedent change values at every update of μ_{pel}. The second antecedent from the input PE_k.del is made up of four fuzzy variables: I1, I2, I3 and I4.

The second stage involves the application of the fuzzy inference rules to both antecedents to generate a *consequent.* Each rule is expressed as follows

$$(antecedent1, antecedent2; consequent)$$

which means *IF antecedent1 AND antecedent2 THEN consequent.* The consequent, 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. A value close to 1 means the bidding PE has a strong chance of being accepted while a decreasing value represents a weaker chance. The process involves the mapping of *ante*1 and *ante*2 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 consequent graph. 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.

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* [4,9,10]. The defuzzification process generates a centroid value for every bidding PE whose range is from 0 to 1. These centroid values are compared and the PE with the maximum value is declared the winner. In the event that the maximum centroid values are the same in some PEs, the award goes to the PE with the most minimum PE_k.pel. The task is then dispatched to the winning PE and its execution starts immediately once that PE is ready. The task next in line in the queue is now put for grab and the bidding process repeats.

Our fuzzy model for the task allocation problem is summarized as Algorithm TS_FL, as follows

/* Algorithm TS_FL */
For j=1 to J
  If TS_j.st is 'waiting' or 'ready' or 'executing'
  1. **If** TS_j is initiated (arrives at $t \geq \mathrm{TS}_j$.at)
    1.1 Place TS_j in the FCFS queue to generate its priority list.
    1.2 Set TS_j.st to 'waiting'.
  2. **If** TS_j in the front of the queue is ready ($t \geq \mathrm{TS}_j$.rt)
    2.1 Remove TS_j from the queue. Set TS_j.st to 'ready'.
    2.2 **(Fuzzification)**

Evaluate PE_k.pel and μ_{pel}
Evaluate PE_k.del using the method in Section 3.2.
Transform these crisp inputs into their fuzzy sets using Table 1.
Determine their degrees from the membership functions.

2.3 **(Applying The Inference Rules)**
Find the minimum degree value from Step 2.2.
Project this value onto their consequence graph.

2.4 **(Defuzzification)**
Find the centroid and area of the trapezium formed.

2.5 Repeat Steps 2.2, 2.3 and 2.4 for other relations using the same inputs.
2.6 Find the final centroid of all overlapping areas in Step 2.5.
2.7 Award the task to the PE with the maximum centroid value in Step 2.6.

3. **If** TS_j is assigned to PE_s // PE_s is the assigned PE
3.1 Set TS_j.st to 'executing'.
3.2 Set PE_s.st to 'busy'.
3.3 Update the Gantt charts.
4. **If** TS_j is executing ($TS_j.ast < t < TS_j.ct$)
4.1 Update the Gantt charts.
5. **If** TS_j completes its execution ($t = TS_j.ct$)
5.1 Set TSj.st to 'completed'.
5.2 Reset PEs.st to 'available'.
5.3 Update the Gantt charts.

4 Simulation Results

The simulation program called TS_FL.EXE, written in C, implements the fuzzy scheduling model using Algorithm TS_FL. The program runs the simulation and generates results for successive cases of 36, 100 and 200 tasks. The start time is $t = 0.0$ and the end is at $t =$ TSL.ct, for the last scheduled task TS_L. For the purpose of this simulation, the precedence relationship of the tasks are predetermined in [7]. Other inputs are obtained in [7], as follows

1. l_j: random number between 1 and 8, generated by the program.
2. TS_j.at: random number, generated by the program.
3. Communication cost c_{ij} from a predecessor TS_i to TS_j: random number between 1 and 3, generated by the program.

Figures 5(a)-(c) show the performance results on cases of 2, 3, 4, 5, 6, 7 and 8 processors. The graphs in Figure 5(a) show evenly distributed μ_{pel} with standard deviation $0.707 \leq \sigma_{pel} \leq 3.347$. There is a fairly good distribution of load on all PEs as reflected on their low σ_{pur} values and relatively equal values of PE_k.pur in the models. The trend of the distribution favors large number of tasks and the use of more PEs, where their σ_{pel} and σ_{pur} values are lower.

Figure 5(a) shows the comparison graphs for μ_{pel} and PE completion time, respectively, on the PE models. The results show that there is a drastic improvement of both the load distribution and completion time when the number of PEs are increased. The speedup graphs in Figure 5(b) tend to support this argument. Finally, the graphs for μ_{pur} in Figure 5(c) show better stability in the distribution of work load when the number of tasks is large, although its value decreases when there are more processors in the network.

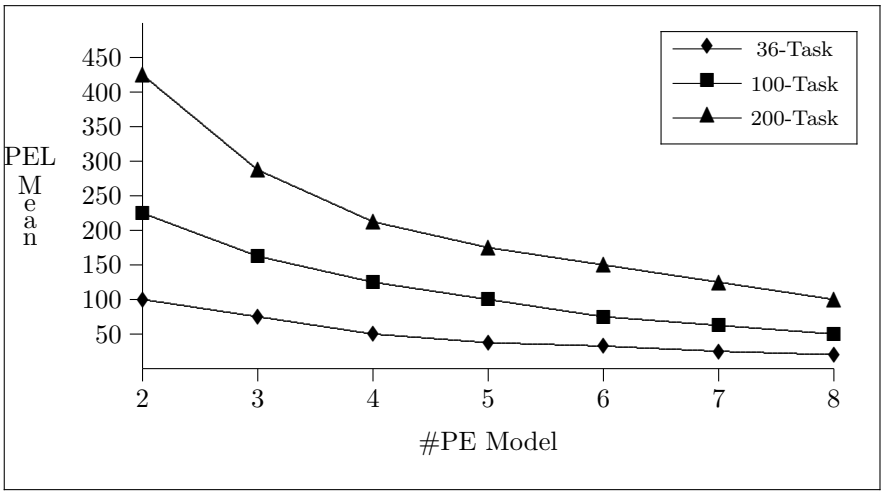

Figure 5(a). Comparison of μ_{pel} on the PE Models

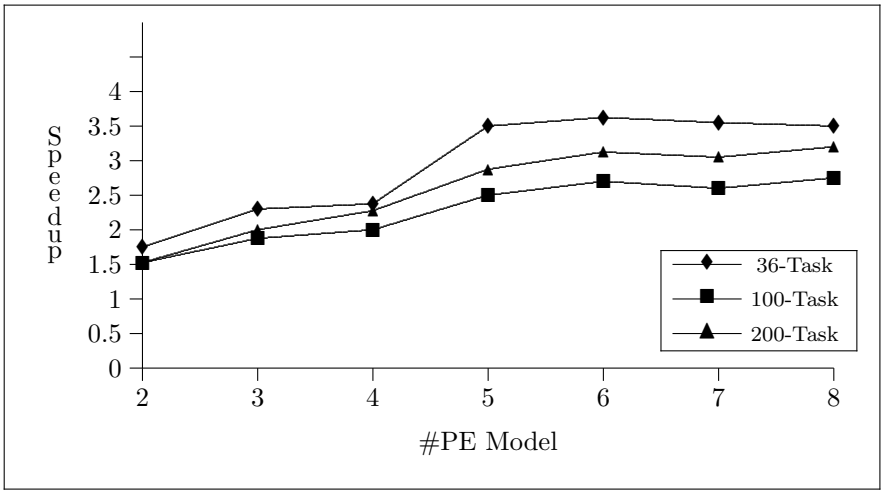

Figure 5(b). Comparison of the Speedup on the PE Models

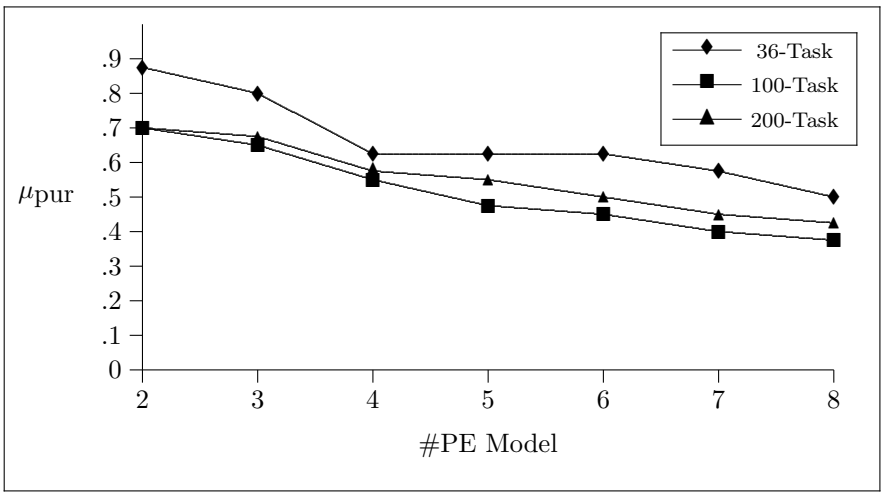

Figure 5(c). Comparison of μ_{pur} on the PE Models

5 Implementation on the Transputer Network

In this section, TS_FS.EXE is implemented using four network models. In the Computer Systems Architects, CSA [1] transputer system, we limit the experiment to five processing elements from which the network models F4, R4, S4 and L4 are made possible. These multi-hop models are the fully-connected, ring, star and linear networks respectively using 4 processing elements, as shown in Figure 6.

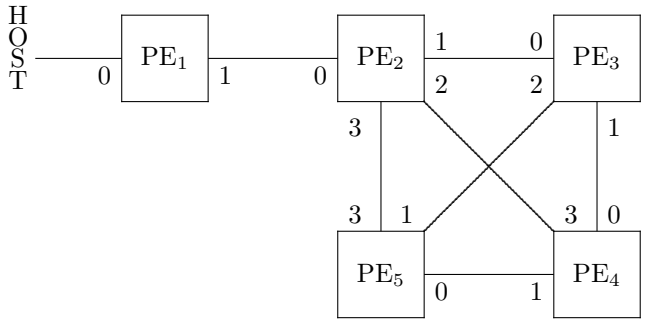

Figure 6. Our Transputer Network

In the above figure, PE_1 is the root transputer or PC/Link, which provides the interface between a front-end computer and the transputer network. Each transputer PE_k is a 32-bit, RISC-based INMOS T805 processor. It has 4 bidirectional serial links labeled as 0, 1, 2 and 3 in the figure. The transputer allows multitasking, concurrent processing of jobs and high-speed communication through message-passing in its network.

For the fuzzy scheduler model, PE_1 is the global scheduler while PE_2, PE_3, PE_4 and

PE_5 are the local schedulers. As the global scheduler, PE_1 runs Algorithm TS_FL, controls all the scheduling activities in the network and makes decision on task assignments. PE_1 will constantly communicate with other PEs in making request for task bidding and gets the feedback before making any decision. Due to the hardware limitation, any message between PE_1 and other PEs must hop through PE_2. This has the impact of slowing down some movement although the difference is not very significant. Communications between the transputers are provided through the channel functions ChanIn() and ChanOut().

The application from Mandelbrot set graphics program [1] is used in the experiment. This program draws fractals from the Mandelbrot set recursively. The sequential code of the program is partitioned into 36 dependent tasks TS_j for $j = 1, 2, \ldots, 36$. Other inputs for the 36 tasks are obtained as follows:

1. l_j is the amount of time required to execute TS_j. The value of l_j is only known on the fly, that is, as the task is executing.

2. TS_j.at: random number, generated by the program. Its value determines the position of TS_j in the central queue.

3. Communication cost c_{ij} is based on the amount of time required to transfer data from TS_i to TS_j. As in (1), its value is only known at the time of execution.

A task TS_j is randomly initiated at $t = TS_j$.at and is immediately placed in the central queue. The global scheduler in PE_1 obtains the information on its arrival time and stores it in its database. When the task TS_j in the front of the queue needs to be scheduled, the global scheduler transmits the information to the relevant PEs. The local scheduler in each PE then responds by providing information on their PE_k.pel. This information is the total execution length already performed in the PE which is in the local scheduler database.

The other information, however, is not immediately available. The global scheduler knows who the predecessors TS_i of TS_j are, but how much data is to be transferred is not known. The method discussed in Section 3.2 is applied. The global scheduler evaluates the value of each PE_k.del by assuming TS_j is in PE_1. The available time for PE_k depends on factors such as the completion of data transfer from its predecessors TS_i and the current processing status of PE_k.

Upon receipt of this information, the global scheduler decides which PE will be awarded with TS_j based on Algorithm TS_FL. The global scheduler then notifies the selected PEs of the decision and the actual start time for its execution TS_j.ast. Algorithm TS_FL-T below summarizes the implementation of TS_FL on the transputer network.

```
/* Algorithm TS_FL-T */
Generate a central queue of tasks from input (2) above.
From t=t_b to t=t_e
    For j=1 to J
    Apply Step 1 of TS_FL.
        If t≥TS_j.at
            PE_1 broadcasts request for bids to PE_k, for k=2,3,...,K.
            TS_j transfers c_ij to PE_1 by assuming TS_j ∈PE_1 and TS_i <TS_j.
    Determine PE_k.del.
        PE_k, for k=2,3,...,K supplies their PE_k.pel and PE_k.del.
```

Apply Steps 2 of TS_FL.
Apply Steps 3, 4 and 5 of TS_FL.

The results obtained from Algorithm TS_FL-T on the F4 network are shown in Table 1. The schedule generates the schedule length SL = 114.35 on PE_2, and a reasonably good load balancing with $\mu_{\text{pel}} = 51.48$ and $\sigma_{\text{pel}} = 3.88$. The table shows the overall performances of the network models. In terms of speedup, the mean of PE utilization rate and the graph completion time, the F4 model performs the best while the L4 model is the worst. The graphs of Figure 7 further illustrate these results.

Table 1: Comparison in Performance

Network Model	Speedup	σ_{pct}	SL	μ_{pur}
F4	1.801	7.467	114.4	0.48
R4	1.726	9.016	119.3	0.47
S4	1.740	8.847	118.3	0.47
L4	1.638	9.602	125.7	0.45

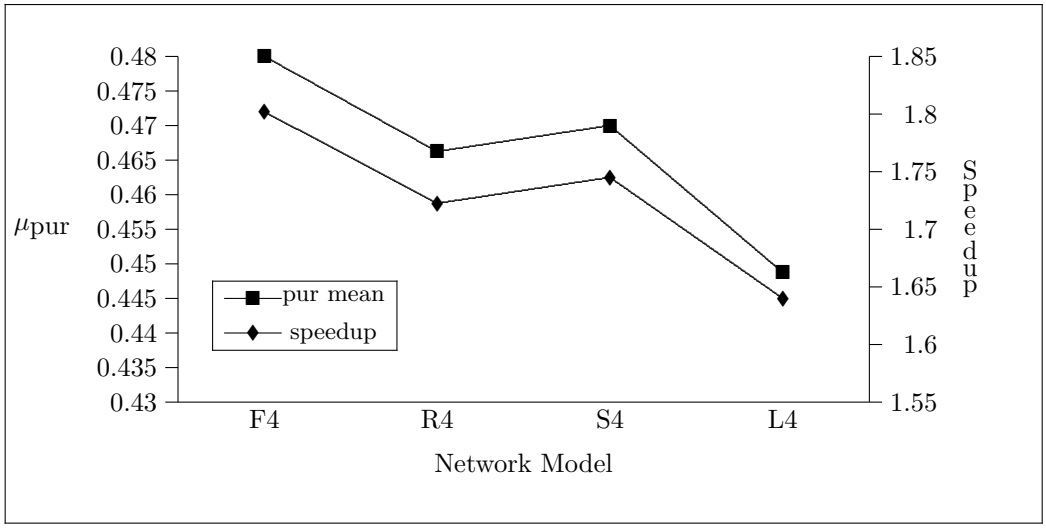

Figure 7. Comparison of μ_{pur} and Speedup on the Network Models

6 Summary and Conclusion

This paper described a study on the dynamic task scheduling problem using fuzzy logic. Fuzzy logic has been applied in processing the incomplete and uncertain inputs in the state of the processors and the task, and then generate some decision on the task assignment. It

has been shown in both the simulation and real-time implementation that the fuzzy-based scheduler performs well in achieving load balancing.

The nondeterministic nature in dynamic scheduling makes the problem very difficult to solve. The scheduler needs to make fast decision on task assignment based on arbitrary and incomplete information on the current state of the task and the processor. This difficulty arises during runtime and it increases overhead to the system as the scheduler will have to keep a log of all assignment activities in order to progress further.

Our approach in using fuzzy logic produces some useful results that meets the load balancing performance objective. In the model, one processor is assigned to be the global scheduler and all others handle their own local schedulers. At any time t the fuzzy scheduler takes inputs from each processor the current execution load PE_k.pel and its delay length PE_k.del. The first input balances the load on all processors by placing higher chance of task assignment to the processor with small load. The second input tries to minimize the delay by giving the processor with small delay a higher chance. Through the fuzzification and defuzzification processes, these two variables are moderated and this generates the decision for the task assignment.

Reference

[**1**] ———, *Logical Systems C for the Transputer Version 89.1 User Manual*, Provo, Utah, 1990. Computer Systems Architects.

[**2**] Y.Chow and W.H.Kohler, *Models for Dynamic Load Balancing in Heterogeneous Multiple Processor Systems*, IEEE Trans. Computers, **28**, no.5, (1979), 354–361. **26** (1963), 115–148.

[**3**] H.El-Rewini, T.G.Lewis and H.H.Ali, *Task Scheduling in Parallel and Distributed Systems*, Prentice Hall, 1994.

[**4**] B.Kosko, *Neural Networks and Fuzzy Systems: A Dynamical Systems Approach to Machine Intelligence*, Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[**5**] H.Lin and C.S.Raghavendran, *A Dynamic Load-balancing Policy with a Central Job Dispatcher*. IEEE Trans. Software Engineering, vol.18, no.2, 1991, pp.148-158.

[**6**] V.Saletore, *A Distributed and Adaptive Dynamic Load Balancing Scheme for Parallel Processing of Medium-grain Tasks*, Proc. of DMCC-5, Portland, Oregon, 1990, pp.994-999.

[**7**] S.Salleh, *Fuzzy and Annealing Models for Task Scheduling in Multiprocessor Systems*, Ph.D Thesis, Dept. of Mathematics, Universiti Teknologi Malaysia, 1997.

[**8**] S.Salleh and A.Y.Zomaya, *Using Fuzzy Logic for Task Scheduling in Multiprocessor Systems* Proc. 8th ISCA Int. Conf. on Parallel and Distributed Computing Systems, Orlando, Florida, 1995, pp.45-51.

[**9**] L.A.Zadeh, *Fuzzy Algorithms* Information and Control, vol.12, 1968, pp.94-102.

[**10**] L.A.Zadeh, *Fuzzy Logic* IEEE Computer, 1988, pp.83-92.