DISTRIBUTED OPTIMIZATION ALGORITHMS FOR QUASI-STATIC THRESHOLD LOAD BALANCING

Kyoo Jeong Lee
GTE Laboratories Incorporated
40 Sylvan Road
Waltham, MA 02254

Don Towsley
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 87-113
October 27, 1987

DISTRIBUTED OPTIMIZATION ALGORITHMS FOR QUASI-STATIC THRESHOLD LOAD BALANCING*

Kyoo Jeong Lee†
GTE Laboratories Incorporated
40 Sylvan Road
Waltham, MA 02254

Don Towsley
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

October 27, 1987

Abstract

In this paper we study the problem of how to efficiently determine the load balancing parameters on-line for a class of threshold load balancing policies in distributed computer systems. We formulate the optimal load balancing problem for each policy, in a static environment, as an integer optimization problem with the mean response time of a job as a performance metric. This kind of problem is inherently difficult to solve exactly (strictly speaking, integer programming problems are NP-complete). We develop heuristic distributed integer optimization algorithms by which each node computes its own load balancing parameters on-line. These algorithms are iterative in nature and each iteration requires a simple computation at each node. In time-varying systems these algorithms can be executed in the background so that they track system variations in a quasi-static manner. Numerical experiments show that the algorithms adapt well in such environments.

*This work was partially supported by the National Science Foundation and the Office of Naval Research under grants ECS-8406402 and N00014-87-K-0796.

†This work was performed while Kyoo Jeong Lee was at the University of Massachusetts.

1 Introduction

In this paper we study the problem of how to determine the load balancing parameters efficiently on-line for a class of threshold load balancing policies in distributed computer systems that have been proposed in the literature [7,8,18,19,22]. We consider systems of autonomous host computers interconnected by a communication network. Each node executes a threshold policy that obeys the following rules. At the time of a job arrival, each node compares a workload indicator to a threshold. If the workload indicator lies below the threshold, the node processes the job locally; otherwise the job is a candidate for transfer to another node for remote processing. This remote node may be chosen according to some probability distribution or as a result of state information obtained from that node. It is important to carefully choose the thresholds and the distribution rule by which to select remote nodes in order for a threshold policy to provide good performance. We propose and evaluate several distributed on-line algorithms whose objectives are to choose values for the load balancing parameters that will provide good performance. These algorithms are derived from a class of distributed algorithms developed to solve nonlinear optimization problems [2,3,9,12].

Although there is a large class of threshold policies, we will focus on those policies that use the number of jobs in the node as a workload indicator and that choose remote nodes for job transfer according to a probability distribution. These policies have been shown to perform as well as more sophisticated policies (*e.g.*, policies that probe other nodes) when each node provides sufficient concurrent processing and allows multiprogramming [19]. Also such policies are appropriate for systems that have large communication delay, thus making probing information outdated [22].

For such policies, we formulate the optimal load balancing problem, in a static environment, as an integer optimization problem with the mean response time of a job as a performance metric. This kind of problem is known to be inherently difficult to solve exactly (mathematically speaking, integer programming problem is NP-complete [23]). Hence a heuristic algorithm is required for its solution. For time-varying systems, this

optimization computation can be invoked every time there is a change in the system environment (*e.g.*, job arrival statistics vary over time or a node is temporarily disconnected from the network). Two approaches exist for obtaining the load balancing parameters on-line. The first approach is to require one of the nodes (or all nodes) to solve the static optimization problem to obtain the load balancing parameters. This process is repeated whenever there is a change in the system environment. If a centralized algorithm is used to solve this optimization problem, it may incur a large amount of overhead in terms of CPU processing. An additional problem with the centralized approach is that the system is vulnerable to the failure of the node executing the algorithm.

We consider a second approach in this paper where all of the nodes in the system execute a distributed algorithm in order to obtain a near optimal solution to the load balancing problem. The motivation behind the use of a distributed algorithm is that it requires simple computation at each node and that it more naturally adapts to a time-varying environment. Specifically, we use techniques developed in decentralized optimization for the minimum delay routing in communication networks [9]. The resulting algorithm is iterative in nature and the load balancing parameters are updated at each iteration. Each node executes the load balancing policy with these new parameters until the next iteration, and during this period estimates its throughput and incremental delay (first derivative of queue length with respect to throughput). These quantities are then exchanged and used to update the load balancing parameters at the next iteration. The algorithm executes in the background at each node and requires little computational resources. It modifies the load balancing parameters as the system workload changes over time. To illustrate the behavior of the algorithm, we consider numerical examples where each node and the communication network are modeled by single-server queueing systems. The results show that the mean response time of a job stabilizes within a neighborhood of the optimal performance in a static environment after a finite number of algorithm iterations. We also study the adaptivity of the algorithm in systems that change *slowly* over time (quasi-static systems) and observe that the algorithm behaves well in such environments.

The following section describes the system model and the class of decentralized

threshold load balancing policies under study. Section 3 formulates the optimal load balancing problem as an integer optimization problem and presents heuristic distributed integer optimization algorithms by which each node obtains its own load balancing parameters on-line. Section 4 contains numerical examples that show the behavior of the algorithm. Section 5 discusses another approach to the problem considered in this paper where *soft* threshold policies are used. Finally, section 6 summarizes the paper.

2 System Model and Load Balancing Policies

In this section we describe the system model and the class of decentralized threshold load balancing policies under study.

2.1 Model Description

The system model consists of N autonomous host computers interconnected by a communication network (see Figure 1). The communication network can be either a local area network or a store-and-forward network. Jobs arrive at each node according to some arrival process with rate ϕ_i, $i = 1, 2, ..., N$. Jobs arriving at node i can be processed at any one of the nodes in the set S_i where $S_i = \{k|$ node k can process jobs originating at node $i\}$ which may be due to hardware/software requirements of different jobs. In order to simplify the problem we assume that this subset depends on the node that the job arrived at and that jobs originating at node i can always be processed locally (*i.e.*, $i \in S_i$). If a job is chosen for remote processing, it is transferred from the source node to a processing node and the results are returned to the source node. Communication delays are incurred during both transfers (*e.g.*, packetization, unpacketization, transmission and queueing delays). We assume that each node has a separate processor (communication server) that handles job transfers between computers. Consequently, the node processor is not affected by the job transfer between nodes. Nodes in the system may have different processing rates and/or job arrival processes. An example of this kind of system is an interactive transaction processing system consisting of multiple computers (*e.g.*, collection of Service

Control Points for database query processing in a Common Channel Signaling network [11]).

2.2 Load Balancing Policies

We consider a class of threshold load balancing policies that only use local state information in making decisions for job transfer. These are referred to as *sender-initiated* policies. We define this class of *decentralized threshold policies* as follows.

Decentralized Threshold Policies

- If a node receives a job from the external world when its workload indicator (*e.g.*, queue length) is above a threshold, it sends the job to other node for remote processing. The choice of a destination node is made according to some probability distribution. Otherwise, it processes the job locally.
- Jobs arriving from other nodes are always accepted at the destination nodes.

This class of policies has been studied in the literature [7,8,22,18,19]. Policies in this class may use different workload indicators and/or use different processor scheduling policies. Figure 2 shows job flows at node i. In Figure 2, Δ_i denotes the rate at which jobs are transferred from node i for remote processing, γ_i denotes the rate at which jobs arrive from other nodes, and β_i denotes the rate at which jobs are accepted for local execution.

One reason why we are interested in this class of load balancing policies is the simplicity of implementation. However, since the job transfer decision is independent of the state of other nodes, it is possible that an arriving job from other nodes may find a busy destination node. In order to avoid this undesirable situation, the source node may probe possible destination nodes to see whether they are busy or not. It then sends the job to a node which is not busy. Such policies have been studied in [7,8,22,19]. The benefit of probes has been shown to be negligible in systems where each node provides sufficient concurrent processing and allows multiprogramming [19] and/or in systems with large communication delay where probing information may be out of date [22].

Load balancing policies divide jobs within the system into two classes; namely *local jobs* and *remote jobs*. Local jobs are those processed at the origination node and remote jobs are those processed at some other node after being transferred through the communication network. We denote the number of local jobs, the number of remote jobs and the number of jobs at node i as $L_i^{(l)}$, $L_i^{(r)}$ and L_i respectively ($L_i = L_i^{(l)} + L_i^{(r)}$). Among the class of decentralized threshold policies, the following two policies are typical examples.

Policy SLO (Sender-initiated LOcal) [18]

- If node i receives a job from the external world with $L_i^{(l)} \geq T_i$, the node sends the job to node j in the set S_i for remote processing with probability P_{ij} where $\sum_{k \in S_i} P_{ik} = 1$ ($P_{ii} = 0$). Otherwise, it processes the job locally.
- Jobs arriving from other nodes are always accepted at the destination node.
- Each node schedules jobs to the processor according to a preemptive priority discipline where local jobs are given a higher priority than remote jobs.

Policy SR (Sender-initiated Random) [7]

- If node i receives a job from the external world with $L_i \geq T_i$, the node sends the job to node j in the set S_i for remote processing with probability P_{ij} where $\sum_{k \in S_i} P_{ik} = 1$ ($P_{ii} = 0$). Otherwise, it processes the job locally.
- Jobs arriving from other nodes are always accepted at the destination node.

Jobs can be processed by any scheduling algorithm (*e.g.*, First-Come-First-Served, Last-Come-First-Served and Processor Sharing) if they have the same priority. Policy SLO is a *selfish* policy in the sense that each node assigns higher priority to local jobs than to remote jobs. Policy SLO uses the number of local jobs as a workload indicator whereas Policy SR uses the total number of jobs. The parameters T_i's and P_{ij}'s need to be determined so that the system obtains a good performance. They should also be modified as the system

environment changes over time. The main focus of this paper is on how to determine these parameters efficiently on-line in time-varying systems.

3 Distributed Parameter Selection Algorithms

In this section we formulate the optimal load balancing problem for each policy, in a static environment, as an integer optimization problem. We then develop several distributed integer optimization algorithms by which each node obtains its own load balancing parameters to the above problem.

3.1 Optimal Load Balancing Problem

Let $f_i(\beta_i, \gamma_i)$ and $g(\Gamma)$ denote the mean queue length at node i and the communication network respectively where $\Gamma = \sum_{i=1}^{N} \gamma_i$. Let D be a random variable that denotes the response time of a job. Using Little's result [14] we obtain the mean response time of a job in the system as follows.

$$E[D] = \frac{\sum_{i=1}^{N} f_i(\beta_i, \gamma_i) + g(\Gamma)}{\Phi} \tag{1}$$

where $E[.]$ denotes the expectation operator and $\Phi = \sum_{i=1}^{N} \phi_i$. In general, it is difficult to obtain an exact expression for $E[D]$ even when simple models are used for nodes and the communication network.

The optimal load balancing problem can be stated as follows.

MINIMIZE $E[D]$ ..... (P1)

with respect to

T_i's and P_{ij}'s,

subject to

$T_i \geq 0$ and T_i is an integer, $i = 1, 2, ..., N$,

$$\sum_{k \in S_i, k \neq i} P_{ik} = 1, \qquad i = 1, 2, ..., N,$$

$$P_{ij} \geq 0, \qquad i, j = 1, 2, ..., N,$$

$$P_{ii} = 0 \text{ and } P_{ij} = 0, \qquad i = 1, 2, ..., N \text{ and } j \notin S_i.$$

This is an integer optimization problem with respect to integer and real variables. In general, this kind of problem is inherently difficult to solve exactly and a heuristic algorithm is required for its solution.

3.2 Distributed Integer Optimization Algorithms

One way of solving problem (P1) is to relax the integer constraints and to solve the problem using any nonlinear optimization technique [21]. The mean response time of a job thus obtained can be used as a lower bound to the solution to problem (P1). We can then adjust the noninteger thresholds to obtain a *feasible* solution. Because the thresholds must take integer values, it does not appear possible to develop a distributed algorithm that performs this task. Instead, we present an algorithm that ensures that the thresholds take integer values at all times.

We first reformulate problem (P1) by relaxing the integer constraints and develop a distributed optimization algorithm for its solution. This algorithm is then modified to deal with integer constraints. Finally, we show how these algorithms can be used in time-varying systems.

3.2.1 Relaxation of Integer Constraints

For notational convenience, let the steady-state job flow from node i to node j be $x_i(j)$. If node j cannot process jobs originating at node i, $x_i(j) = 0$. Note that $x_i(i) = \beta_i$ and $\sum_{k=1, k \neq i}^{N} x_k(i) = \gamma_i$. For given values of T_i's and P_{ij}'s, the steady-state job flows $x_i(j)$'s can be obtained. However, it may not be possible to obtain integer-valued T_i's from a set of feasible $x_i(j)$'s. For the moment we ignore this problem and treat $x_i(j)$'s as nonnegative real variables. Problem (P1) can then be rewritten as follows.

MINIMIZE $E[D]$ ..... (P2)

with respect to

$$x_i(j)\text{'s},$$

subject to

$$\sum_{k \in S_i} x_i(k) = \phi_i, \qquad i = 1, 2, ..., N,$$

$$x_i(j) \geq 0, \qquad i, j = 1, 2, ..., N,$$

$$x_i(j) = 0, \qquad i = 1, 2, ..., N \text{ and } j \notin S_i.$$

Note that problem (P2) is very similar, in form, to the multicommodity flow problem in networks [9].

Let $\delta_{i,j}$ denote the Kronecker delta function (*i.e.*, $\delta_{i,j} = 1$ for $i = j$, and $\delta_{i,j} = 0$ otherwise). Necessary conditions for the optimal solution of problem (P2) can be derived from the Kuhn-Tucker conditions [21] as follows: For all $j \in S_i$,

$$\frac{df_j(\beta_j, \gamma_j)}{dx_i(j)} + (1 - \delta_{i,j}) \frac{dg(\Gamma)}{dx_i(j)} \begin{cases} = \lambda_i, & \text{for } x_i(j) > 0; \\ \geq \lambda_i, & \text{for } x_i(j) = 0, \end{cases} \tag{2}$$

where λ_i is some constant (Lagrange multiplier). Note that the multiplicative term $(1-\delta_{i,j})$ is introduced since local jobs do not experience communication delay. Relation (2) indicates that from node i's point of view the incremental delay incurred for jobs processed at node j due to the job flow from node i to node j should be equal for all j if there is a positive job flow from node i to node j. On the other hand, if there is no job flow from node i to node j, the incremental delay should be no less than the above value. Note that since $df_j/dx_i(j) = df_j/dx_k(j)$ for $i \neq j \neq k$, $\lambda_i = \lambda_k$ if $x_i(j) > 0$ and $x_k(j) > 0$. This means that if there are positive job flows from nodes i and k to node j, the incremental delay at node j seen by nodes i and k should be equal.

Using relation (2), we develop a distributed optimization algorithm that solves problem (P2). This algorithm is an extension of the work by Gallager [9] that developed a distributed algorithm for solving the minimum delay routing problem in communication

networks. The algorithm is iterative in nature and, at each iteration, each node reroutes a small amount of job flows from all other nodes to the node that has the minimum incremental delay. The amount of rerouted job flow is proportional to the difference in the incremental delays.

We define A_i as,

$$A_i = \min_{k \in S_i, k \neq i} \{df_k(\beta_k, \gamma_k)/dx_i(k) + dg(\Gamma)/dx_i(k)\}. \tag{3}$$

Let $h_i(j)$ be the difference between the incremental delay of node j for $j \in S_i$ and the minimum incremental delay seen by node i including its own incremental delay and,

$$h_i(j) = \frac{df_j(\beta_j, \gamma_j)}{dx_i(j)} + (1 - \delta_{i,j})\frac{dg(\Gamma)}{dx_i(j)} - \min\{A_i, \frac{df_i(\beta_i, \gamma_i)}{dx_i(i)}\}. \tag{4}$$

Let $\Delta_i(j) = \min\{x_i(j), \eta h_i(j)\}$ where η is a step size parameter. Note that $x_i(j)$ is the maximum amount of job flow that can be rerouted from node i to node j. Initially, each node i sets a set of feasible $x_i(j)$'s arbitrarily. Node i then executes the following algorithm at each iteration.

Algorithm A

- Gathering the incremental delay information for all $j \in S_i$, node i computes $x_i^1(j)$ where,

$$x_i^1(j) = \begin{cases} x_i(j) - \Delta_i(j), & \text{for } j \neq k_{min}(i); \\ x_i(j) + \sum_{k \neq k_{min}(i)} \Delta_i(k), & \text{for } j = k_{min}(i), \end{cases} \tag{5}$$

  and $k_{min}(i)$ denotes the site that yields the minimum incremental delay seen by node i (*i.e.*, $\min\{A_i, \frac{df_i(\beta_i,\gamma_i)}{dx_i(i)}\}$).

- Perform a convergence test by checking $\sum_{k \in S_i}\{x_i(k) - x_i^1(k)\}^2 < \epsilon$ where ϵ is a given positive constant. If the test is satisfied, terminate the iteration. Otherwise, go to the next step.

- Redistribute the workload according to $x_i^1(j)$'s for $j \in S_i$.

To implement Algorithm A, the incremental delay information is required along with a protocol to exchange this information between nodes.

The choice of the parameter η affects the convergence rate of the algorithm [9,6]. A large value for η yields a fast convergence rate of the algorithm. However, we cannot choose an arbitrarily large value for η since the algorithm may not converge in that case. See [3] for a discussion on choosing step sizes. The convergence proof of this algorithm can be found in Gallager [9]. The algorithm converges linearly to the optimal solution provided that the function, $E[D]$, is a convex functionof the control parameters (linear convergence means that the tail of the error sequence of intermediate solutions forms a geometric sequence). Superlinear convergence can be obtained if one uses second derivative information [2]. Convergence of an asynchronous implementation of this algorithm was discussed by Tsitsiklis and Bertsekas [31].

A similar distributed optimization algorithm has been developed by Heal to solve the economic planning problem [12] and extensions of this algorithm have been used to solve static load balancing and file allocation problems in distributed computer systems [16,15]. In this algorithm, the average incremental delay (instead of the minimum incremental delay) is compared with the incremental delay of each node in order to update $x_i(j)$'s. We define A_i^* as,

$$A_i^* = \sum_{k \in S_i} \frac{df_k(\beta_k, \gamma_k)/dx_i(k) + (1 - \delta_{i,k})dg(\Gamma)/dx_i(k)}{|S_i|}, \tag{6}$$

where $|S_i|$ denotes the cardinality of the set S_i. Let $\Delta_i(j)^* = \eta^* h_i(j)^*$ where η^* is a step size parameter and,

$$h_i(j)^* = \frac{df_j(\beta_j, \gamma_j)}{dx_i(j)} + (1 - \delta_{i,j})\frac{dg(\Gamma)}{dx_i(j)} - A_i^*. \tag{7}$$

We denote the new algorithm as Algorithm B which is defined in the same way as Algorithm

A except for the rules to update $x_i(j)$'s as,

$$x_i^1(j) = \begin{cases} x_i(j) - \Delta_i(j)^*, & \text{for } j \in K_i; \\ x_i(j), & \text{otherwise.} \end{cases} \tag{8}$$

The set K_i is determined iteratively so that the feasibility constraint is satisfied [12,16].

Unfortunately, the above algorithms cannot be used directly to solve problem (P1) since we cannot control the steady-state job flows continuously using integer-valued T_i's. If the cost functions are convex, the algorithms can be used off-line to obtain lower bounds on the best achievable performance. In the next subsection we develop heuristic algorithms to account for the integer constraints. They use ideas similar to those used in Algorithms A and B.

3.2.2 Heuristic Algorithms

One algorithm closely follows the idea of Algorithm A where each node compares its own incremental delay to the minimum incremental delay of other nodes seen by itself to determine whether to increase or decrease its local job throughput (*i.e.,* increase or decrease the threshold by one). This adjustment of thresholds may change the job flows abruptly. The transfer probabilities are determined according to the *slack* processing power of nodes where the slack processing power of a node is defined as the difference between the maxmum processing capacity of the node and the local job throughput.

Initially, each node i sets T_i and P_{ij}'s for $j \in S_i$ ($P_{ii} = 0$) to some arbitrary values. Node i then executes the following algorithm at each iteration.

Algorithm C

- After gathering the incremental delay information for all $j \in S_i$, node i computes the value of A_i.
- If $df_i/dx_i(i) > A_i(1 + \theta_i)$ where θ_i is a given positive constant, set $T_i := T_i - 1$. Else if $df_i/dx_i(i) < A_i(1 - \theta_i)$, then set $T_i := T_i + 1$. Otherwise, set $T_i := T_i$. (The

parameter θ_i must be tuned to prevent threshold change due to a slight imbalance in incremental delays. Numerical experience indicates that $\theta_i = 0$ works well.)

- Using the values of $\mu_j - x_j(j)$, determine P_{ij} for all $j \in S_i$ ($P_{ii} = 0$) as,

$$P_{ij} = \frac{\mu_j - x_j(j)}{\sum_{k \in S_i, k \neq i} \{\mu_k - x_k(k)\}}, \tag{9}$$

where μ_j denotes the maximum processing rate of node j. Note that $\mu_j - x_j(j)$ denotes the *slack* processing power of node j.

Note that in the above algorithm we do not route all incremental job flows from each node to the node that has the minimum incremental delay as in Algorithm A. Numerical results in the next section show that the use of slack processing power to determine the transfer probabilities yields better mean response time performance.

In Algorithm C the minimum incremental delay is used to update the threshold parameters. We next present another algorithm based on Algorithm B where average incremental delay is used instead of the minimum incremental delay. We denote the new algorithm as Algorithm D which is defined in the same way as Algorithm C except that A_i is replaced by A_i^*.

In order to implement these algorithms, efficient estimators for incremental delay and slack processing power are required. The incremental delay information at each node can be obtained in one of several ways. One way is to measure the local job throughput and the remote job throughput at each node, and use closed-form incremental delay formulas if they are available. We assume that the incremental delays due to remote job flow are not affected by the integer threshold constraints. However, incremental delays due to local job flow are indeed affected by the integer threshold constraints since local job flow is directly controlled by the threshold parameter. In this case we use the following backward difference formula to approximate the incremental delay due to local job flow.

$$df_i/dx_i(i) \approx \frac{[f_i]_{T_i} - [f_i]_{T_i-1}}{[x_i(i)]_{T_i} - [x_i(i)]_{T_i-1}}, \quad i = 1, 2, ..., N \tag{10}$$

where $[f_i]_{T_i}$ and $[x_i(i)]_{T_i}$ denote the mean queue length and the local job throughput at node i when the threshold is T_i. This approximation, in fact, yields a smaller value than the true incremental delay. Another option is to use a forward difference formula as follows.

$$df_i/dx_i(i) \approx \frac{[f_i]_{T_i+1} - [f_i]_{T_i}}{[x_i(i)]_{T_i+1} - [x_i(i)]_{T_i}}, \quad i = 1, 2, ..., N \tag{11}$$

This approximation, on the other hand, overestimates the true incremental delays. In the next section we compare versions of Algorithm C that use the two formulas. Efficient algorithms for obtaining the data required to compute equations (10) and (11) are found in [5,28,24].

Incremental delays due to remote job flows can be computed using one of a number of different techniques such as perturbation analysis [13] and likelihood ratio estimation [10,25,24]). The slack processing power is easily obtained by measuring local job throughput. The minimum number of events that is to be measured to obtain a good estimate for incremental delays and slack processing power is dependent on the specific applications of the system and can be determined empirically (*e.g.*, a window of at least 100 job completions is required to make a good estimate).

Algorithms C and D can be triggered either by specific events or periodically. In the first case, each node measures its incremental delay and slack processing power, and reports them to other nodes if they differ from previously reported values by a certain margin. In the second case, each node measures and reports its incremental delay and the slack processing power to other nodes periodically. In either case, the overhead of exchanging this information is small.

In general, Algorithms C and D do not converge to the optimal solution. (Note that there is no convergence test at each iteration.) However, we conjecture that after a finite number of algorithm iterations, the mean response time of a job stabilizes within a neighborhood of the optimal performance in a static environment. This kind of behavior has been observed through many numerical examples some of which are shown in the next section. When the system environment varies *slowly* over time, the above algorithms can be executed in the background. In such a case, when there is an imbalance in incremental

delays due to a change in the system environment, the algorithm corrects the imbalance properly so that system performance may improve. In the next section we study the behavior of these algorithms where there are step changes in the system environment. This study qualitatively reveals the algorithms' adaptivity in time-varying systems.

4 Numerical Results

In this section we provide numerical examples. We first compare the performance of Algorithms C and D, and study the effects of various parameters and the incremental delay approximations used in the algorithm. We then look at examples where the system environment changes over time. Although we provide numerical results for a small set of system parameters, we have observed that the conclusions drawn in this section hold for a wide range of system parameters.

We model each node i, $i = 1, \cdots, N$, and the communication network as single-server queueing systems that have exponential service time with mean $1/\mu_i$ and $1/\mu_{ch}$ time units respectively. Jobs arrive at node i according to a Poisson process with rate ϕ_i and the system executes Policy SLO. For this model, we use the results in [19] to obtain a closed-form expression for the mean response time of a job. The analysis carried out in [19] is based on the Poisson assumption on the remote job arrivals at each node. It has been shown that this analysis becomes exact as the number of nodes in the system approaches infinity. Simulation studies showed that this approximation is good even in systems containing ten nodes when we are interested in obtaining the mean response time performance. Using this closed-form expression, we can study the behavior of the algorithms analytically.

We consider a system that contains three equal size classes of nodes where nodes in each class have the same external workload and job processing power. Let $\mu^{(i)}$, $\phi^{(i)}$, and $u^{(i)}$ denote the job processing rate, external job arrival rate and utilization of nodes in class i without load balancing respectively (*i.e.*, $u^{(i)} = \phi^{(i)}/\mu^{(i)}$). These nodes are interconnected by a communication network with mean service time of $1/\mu_{ch} = 0.01$ time units. We assume that there are no site constraints. We look at the following two examples.

- Example 1: $\mu^{(1)} = \mu^{(2)} = \mu^{(3)} = 1.0$, $u^{(1)} = 0.7$, $u^{(2)} = 0.9$ and $u^{(3)} = 1.1$
- Example 2: $\mu^{(1)} = 1.0$, $\mu^{(2)} = 2.0$, $\mu^{(3)} = 4.0$, $u^{(1)} = 0.9$, $u^{(2)} = 0.8$ and $u^{(3)} = 0.9$

Unless stated otherwise, the following parameters are used in the algorithms; initial thresholds randomly drawn between 5 and 10, the parameter θ equal to 0, and the backward difference approximation for incremental delays.

In the following figures, NLB (No Load Balancing) denotes the mean response time of a job in the system without load balancing. Similarly, SLB (Static Load Balancing) denote the mean response time under the probabilistic static load balancing of Tantawi and Towsley [29]. Briefly, under the static load balancing each node chooses a destination node (including itself) for execution of external job arrivals according to some probability distribution, regardless of its own workload. These probability parameters are then adjusted to obtain the optimal performance. Finally, CLB (Conjectured Lower Bound) denotes the mean response time obtained by solving problem (P1) with relaxed integer constraints using Algorithm A. Since we were unable to prove the convexity of $E[D]$, we attempted different initial solutions in order to obtain the global optimal solution. In all cases, the algorithm yields the same solution regardless of the initial solutions tried. This provides circumstantial evidence that $E[D]$ has single local minimum. We conjecture that CLB provides a lower bound to the optimal system performance obtainable by Policy SLO.

Figure 3 compares the performance of Algorithms C and D. The performance of the system under both algorithms approaches the lower bound initially and then stabilizes in a neighborhood of the lower bound. Algorithm C yields a lower mean response time than Algorithm D in the stabilized region. In the figure Algorithm C^* denotes the case where all incremental job flows are routed to the node that has the minimum incremental delay in Algorithm C (instead of using slack processing power). This yields a higher mean response time than Algorithm C. In the following we focus on the behavior of Algorithm C only. Figure 4 shows the effect of initial threshold values where case 1 corresponds to initial thresholds drawn randomly between 5 and 10 and case 2 corresponds to initial

thresholds drawn randomly between 10 to 20. When the initial thresholds are large, more iterations are required for the algorithm to stabilize. This is due to the fact that the optimal thresholds are usually small (*e.g.*, less than six). Figure 5 shows the effect of the parameter θ. This parameter was introduced to prevent a change in the threshold due to a slight imbalance between incremental delays. It shows that $\theta = 0$ works well. Figure 6 compares the effect of the backward and forward difference approximations for incremental delays. The backward approximation underestimates incremental delays whereas the forward approximation overestimates them. Each node usually transfers more jobs when the forward approximation is used, resulting in a performance degradation.

We next study the behavior of Algorithm C subject to step changes in the system environment. When there is a step change during the execution of the algorithm, the situation is the same as in the static environment case except that Algorithm C now uses the T_i's and $P_{i,j}$'s last calculated in the previous environment as initial values. If the change is small, these may not be far from the optimal values in the new system environment. We consider the following examples.

- Example 3: $\mu^{(1)} = \mu^{(2)} = \mu^{(3)} = 1.0$.

  Initially, nodes in all classes have the same utilization of 70%. Right after the $15th$ iteration, the utilization of nodes in class 2 changes to 90% (*i.e.*, $\phi^{(2)}/\mu^{(2)} = 0.9$). Right after the $25th$ iteration, the utilization of nodes in class 2 returns to 70%.

- Example 4: $\mu^{(1)} = \mu^{(2)} = \mu^{(3)} = 1.0$.

  Initially, nodes in all classes have the same utilization of 70%. Right after the $15th$ iteration, the utilization of nodes in class 2 changes to 110%. Right after the $25th$ iteration, the utilization of nodes in class 2 returns to 70%.

- Example 5: The same as Example 3 except that $\mu^{(1)} = 1.0$, $\mu^{(2)} = 2.0$ and $\mu^{(3)} = 4.0$.

- Example 6: The same as Example 4 except that $\mu^{(1)} = 1.0$, $\mu^{(2)} = 2.0$ and $\mu^{(3)} = 4.0$.

In the figures to follow *System* shows the behavior of Algorithm C and CLB denotes the conjectured lower bound. Due to the workload changes right after the $15th$ and $25th$

iterations, incremental delays and slack processing powers have been changed. This information is then used to update T_i's and P_{ij}'s in the next iteration. For a change of increasing workload, Figure 7 (a) and Figure 8 (a) show that the algorithm adapts to the change smoothly. For a large amount of increasing workload, Figure 7 (b) and Figure 8 (b) show that the algorithm still adapts to the change quickly although it results in slow response time performance in a transient period of two algorithm iterations. In both cases, the algorithm adapts smoothly to changes in which the workload decreases.

5 Discussion

This paper has focused on the problem of how to determine the load balancing parameters efficiently on-line for a class of decentralized threshold policies that have been proposed in the literature. One difficulty with threshold policies is that the optimal load balancing problem is an integer optimization problem which is difficult to solve exactly.

One way of avoiding integer optimization is to introduce new policies where the control parameters can take on real values. For example Policy SR can be modified as follows.

Policy SR^*

- If node i receives a job from the external world with $L_i = T_i$, the node sends the job to other node with probability q_i. On the other hand, if node i receives a job from the external world with $L_i \geq T_i + 1$, it sends the job to other node with probability one. Otherwise, it processes the job locally. In case of job transfer, node j in the set S_i is chosen with probability P_{ij} where $\sum_{k \in S_i} P_{ik} = 1$ ($P_{ii} = 0$).
- Jobs arriving from other nodes are always accepted at the destination node.

We refer to this as a *soft* threshold policy. Using this scheme, we can control the steady-state job flows continuously by a proper choice of T_i's, q_i's, and P_{ij}'s. Therefore the optimal load balancing problem is just a nonlinear optimization problem and Algorithm A (or B)

can be used directly for its solution. However, Lazar [17] showed that, for a certain class of queueing systems, the resulting mean queue length as a function of throughput is piecewise concave between points corresponding to integer thresholds. Furthermore, the derivatives at those points are discontinuous. Hence, problem (P2) is not a convex programming problem. We have applied Algorithm A to this problem with poor results. Although our preliminary work in this area is not promising, this approach requires further investigation.

6 Conclusions

In this paper, we studied the problem of how to determine the load balancing parameters efficiently on-line for a class of threshold load balancing policies that only use local state information in making job transfer decisions. We formulated the optimal load balancing problem for each policy, in a static environment, as an integer optimization problem. We then devised heuristic distributed integer optimization algorithms for its solution whose underlying idea comes from a distributed algorithm developed for the minimum delay routing in communication networks. The algorithm is iterative in nature and each iteration requires simple computation from each node. All that is required for the implementation of the algorithm are the measurement of incremental delay and slack processing power, and a protocol for exchange of this information between nodes. To illustrate the behavior of the algorithm, we considered numerical examples where each node and the communication are modeled by single-server queueing systems. The results show that after a finite number of algorithm iterations the performance of the system, in a static environment, is stabilized within a neighborhood of the optimal performance. This algorithm also shows a good adaptivity in a time-varying environment. Although not considered here, we believe that the decentralized optimization algorithm approach may be applicable to threshold policies that use probes to obtain nonlocal state information. This will be a topic for future research. It also remains as a future project to implement the optimization algoriithm with the necessary incremental delay estimators and study its performance.

References

[1] A. Barak and A. Shiloh, "A distributed load-balancing policy for a multicomputer," *Software - Practice and Experience*, vol. 15, pp. 901-913, 1985

[2] D.P. Bertsekas, E.M. Gafni, and R.G. Gallager, "Second derivative algorithms for minimum delay distributed routing in networks," *IEEE Trans. on Comm.* vol.COM-32, pp.911-919, Aug. 1984

[3] D.P. Bertsekas, "On the Goldstein-Levitin-Polyak gradient projection method," *IEEE Trans. on Automatic Control*, vol.AC-21, No.2, pp.174-184, April 1976

[4] R.M. Bryant and R.A. Finkel, "A stable distributed scheduling algorithm," *Proceedings of Dist. Comp. Systems Sym.*, pp. 314-323, 1981

[5] C.G. Cassandras, "On-line optimization for a flow control strategy," *Proc. of the 25th CDC Conference*,1986.

[6] C.G. Cassandras, M.V. Abidi, and D.F. Towsley, "Distributed routing with on-line marginal delay estimation," submitted to *INFOCOM'88.*

[7] D.L. Eager, E.D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 662-675, May 1986

[8] D.L. Eager, E.D. Lazowska, and J. Zahorjan, "A comparison of receiver-initiated and sender-initiated adaptive load sharing," *Performance Eval.*, vol. 6, pp. 53-68, March 1986

[9] R.G. Gallager, "A minimum delay routing algoithm using distributed computations," *IEEE Trans. on Comm.*, vol. COM-25, pp.73-85, Jan. 1977

[10] P.W. Glynn and J.L. Sanders, "Monte-Carlo optimization of stochastic systems: two new approaches", *Proceedings of 1986 ASME Computers in Engineering Conference*, 1986

[11] R. Hass and R. Robrock, "Intelligent network of the future," *Proceedings of Globecom*, pp. 1311-1315, 1986

[12] G.M. Heal, "Planning without prices", *Review of Economic Studies*, vol. 36, No.3, pp. 347-363, 1969

[13] Y.C. Ho and C.G. Cassandras, "A new approach to the analysis of discrete event dynamic systems," *Automatica*, Vol. 19, 2, pp. 149-167, 1983.

[14] L. Kleinrock, Queueing Systems, Vol. I: Theory, Wiley, 1975

[15] J.F. Kurose and R. Simha, "A microeconomic approach to optimal file allocation," *Proc. 6th International Conf. on Distr. Comp. Systems,* May 1986.

[16] J.F. Kurose and S. Singh, "A distributed algorithm for optimal static load balancing in distributed computer systems", *Proceedings of INFOCOM'86*, Miami, Florida, 1986

[17] A.A. Lazar, "Optimal flow control of a class of queueing networks in equilibrium," *IEEE Trans. on Automatic Control*, vol.AC-28, No.11, pp.1001-1007, Nov. 1983

[18] K.J. Lee and D.F. Towsley, "A comparison of priority-based decentralized load balancing policies," *Proceedings of Performance'86 and ACM Sigmetrics 1986 Joint Conference*, Raleigh, North Carolina, 1986

[19] K.J. Lee, Load Balancing in Distributed Computer Systems, Ph.D. Dissertation, Dept. of Electrical and Computer Engineering, Univ. of Massachusetts, 1987

[20] M. Livny and M. Melman, "Load balancing in homogeneous broadcast distributed systems," *Proceedings of ACM Comp. Net. Perf. Symposium*, pp. 47-55, 1982

[21] D. Luenberger, Introduction to Linear and Nonlinear Programming, Addison-Wesley, 1973

[22] R. Mirchandaney, D.F. Towsley, J.A. Stankovic, "Analysis of the effects of delays on load sharing," to appear in *IEEE Trans. on Computers*

[23] C.H. Papadimitriou and K. Steiglitz, Combinatorial Optimization, Prentice-Hall, New Jersey, 1982

[24] S. Pulidas, D. Towsley, J.A. Stankovic, "Design of efficient parameter estimators for decentralized load balancing policies, " COINS Tech Report 87-79, Univ. Massachusetts, Aug. 1987.

[25] M.I. Reiman and A. Weiss, "Sensitivity analysis for simulations via likelihood ratios," *Proc. of 1986 Winter Simulation Conference.*

[26] E.S. Silva and M. Gerla, "Load balancing in distributed systems with multiple classes and site constraints," *Proc. of Performance'84*, E.Gelenbe (Editor), pp.17-33, 1984

[27] J.A. Stankovic, "Simulations of three adaptive, decentralized controlled, job scheduling algorithms," *Computer Networks*, vol. 8, pp.199-217, 1984

[28] S. Strickland, C.G. Cassandras, "Augmented chain analysis of Markov and semi-Markov processes," *Proc. 25th Allerton Conf.*, Sept. 1987.

[29] A.N. Tantawi and D.F. Towsley, "Optimal static load balancing in distributed computer systems," *Journal of ACM*, vol. 32, pp. 445-465, April 1985

[30] A.N. Tantawi and D.F. Towsley, "A general model for optimal load balancing in star network configuration," *Proceedings of Performance'84*, pp. 277-291, Paris, France, 1984

[31] J.N. Tsitsiklis and D.P. Bertsekas, "Distributed asynchronous optimal routing in data networks," *Proceedings of 23rd Conf. on Decision and Control*, Las Vegas, Nevada, Dec. 1984

[32] Y.T. Wang and R.J.T. Morris, "Load sharing in distributed systems," *IEEE Trans. on Computers*, vol. C-34, pp. 204-217, March 1985

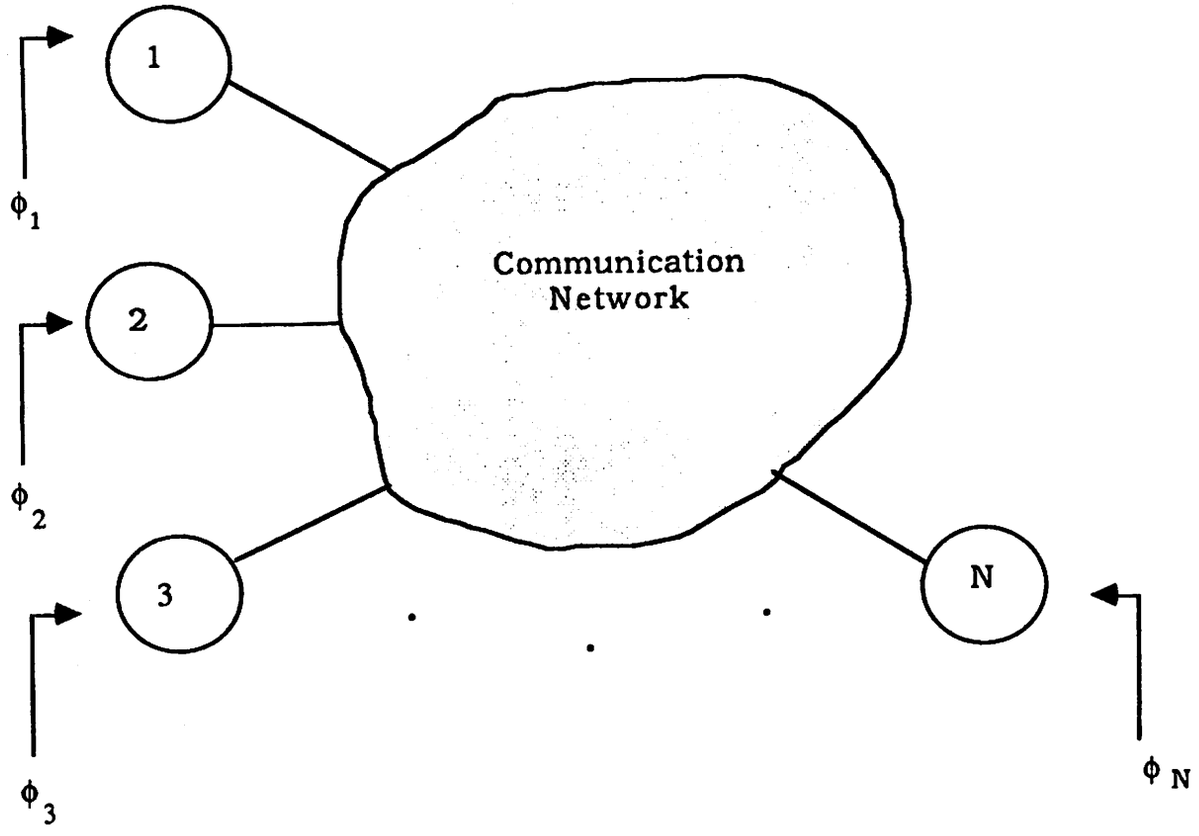


Figure 1 System model

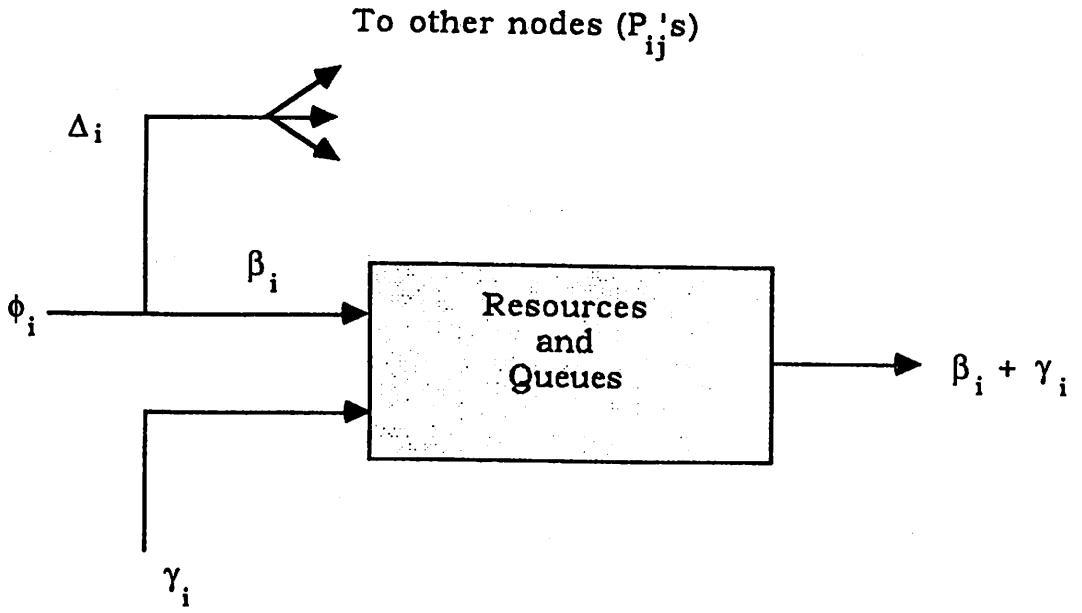
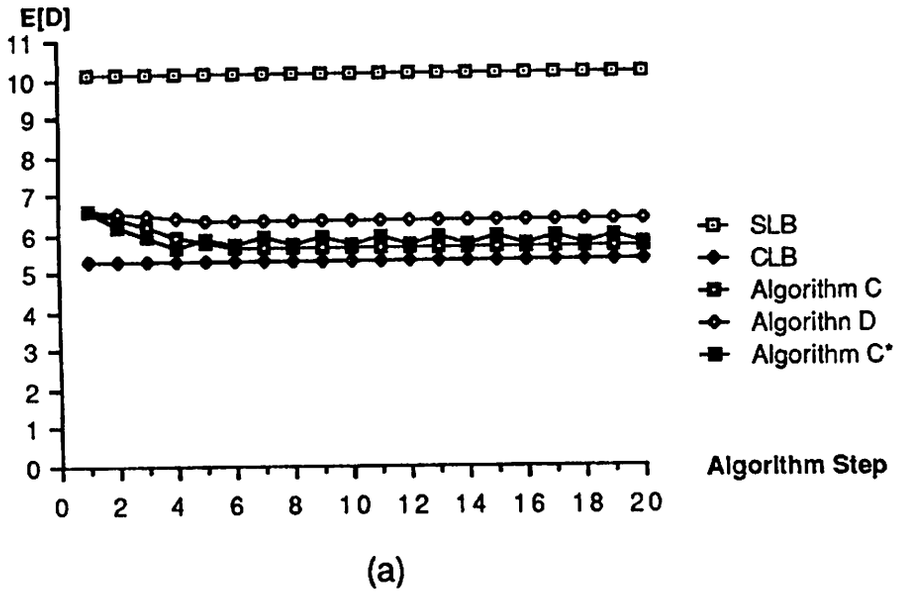
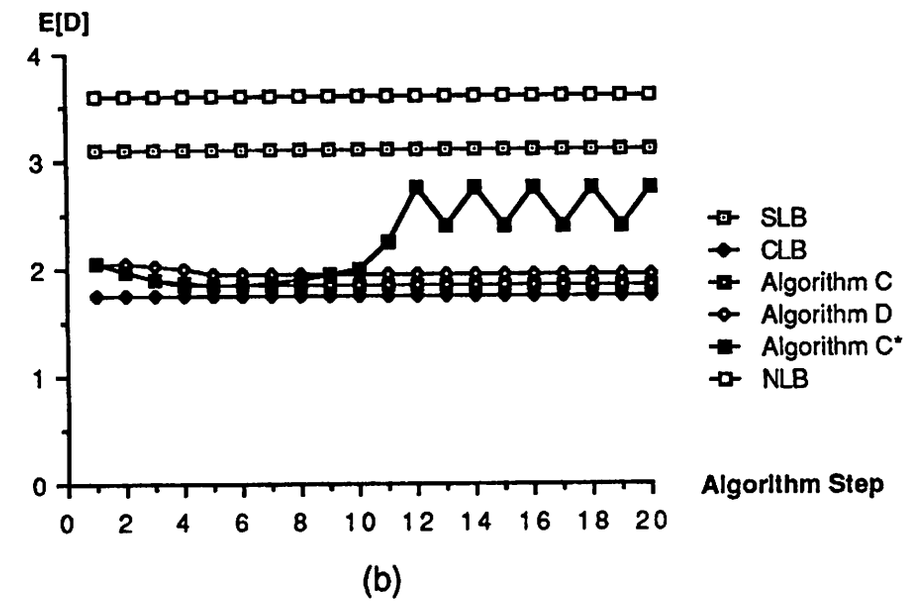


Figure 2 Job flows at node i



(a)



(b)

Figure 3 Comparison of Algorithms C and D
(a) Example 1
(b) Example 2

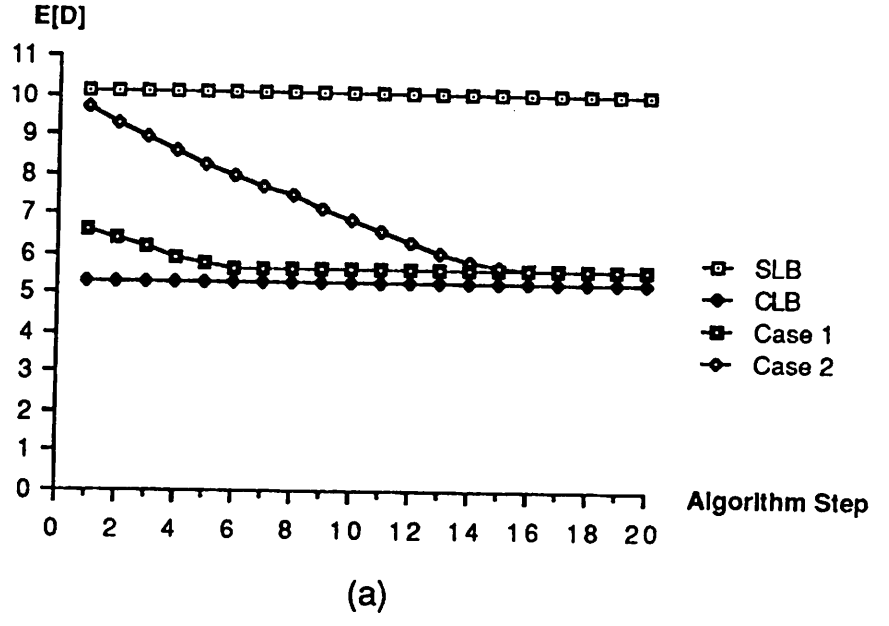


Figure 4 Effect of initial thresholds in Algorithm C
(a) Example 1
(b) Example 2

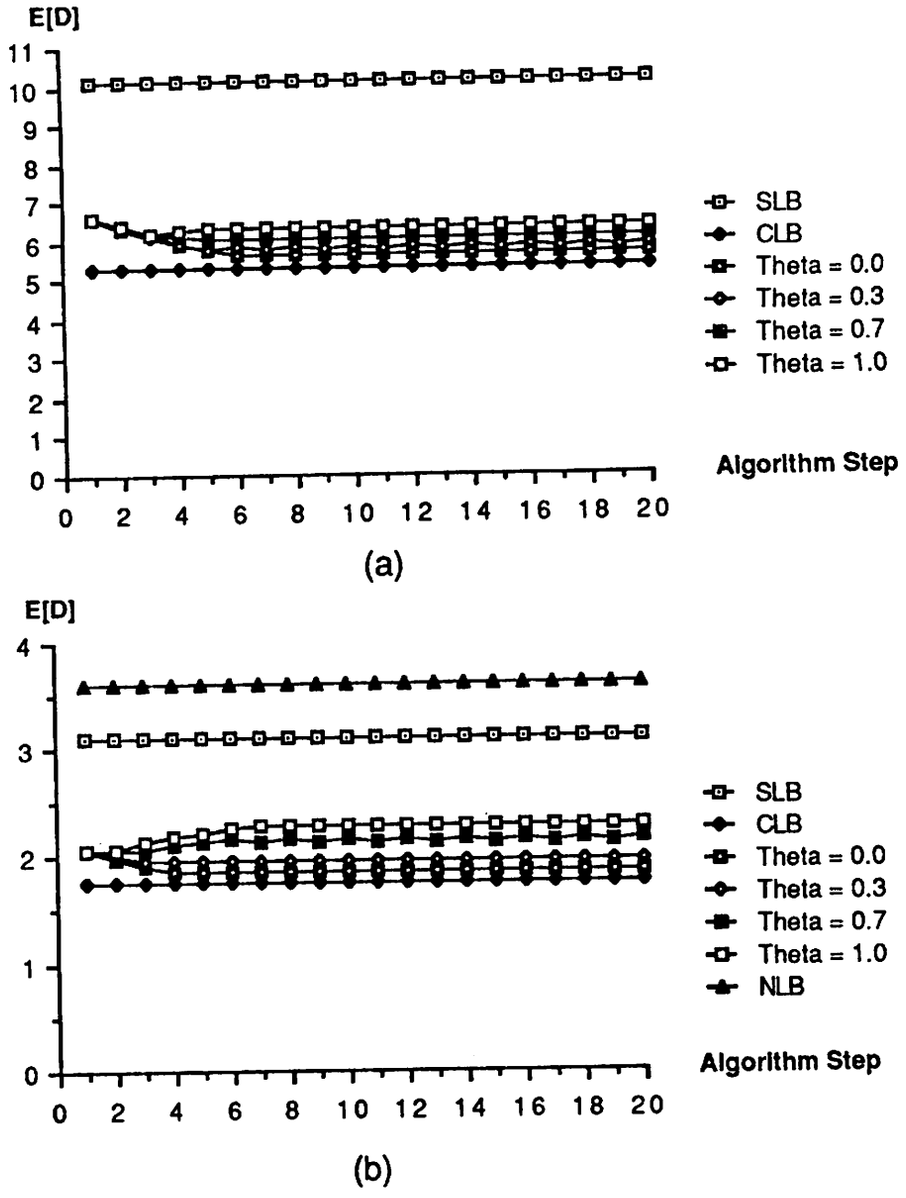


Figure 5 Effect of parameter θ in Algorithm C
(a) Example 1
(b) Example 2

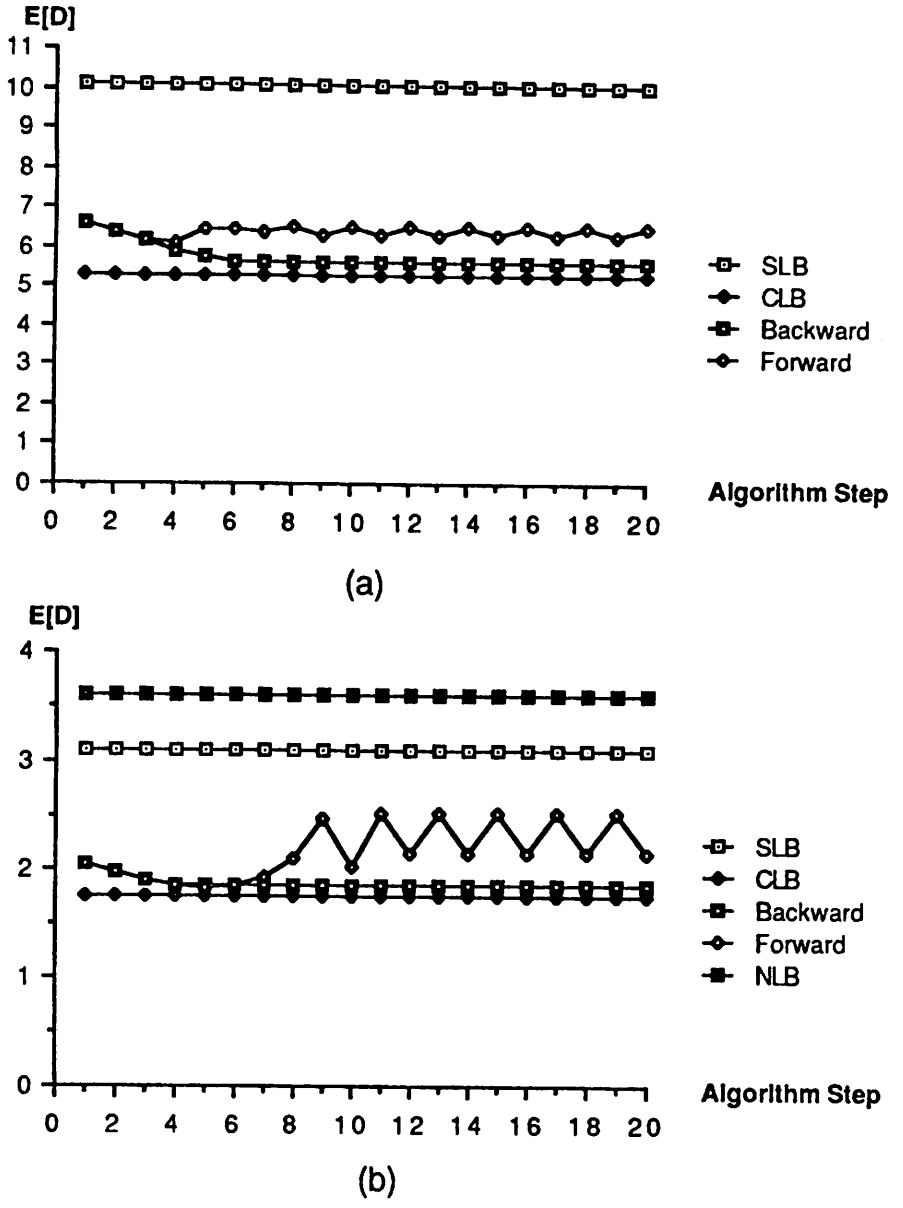


Figure 6 Effect of incremental delay approximation in Algorithm *C*
(a) Example 1
(b) Example 2

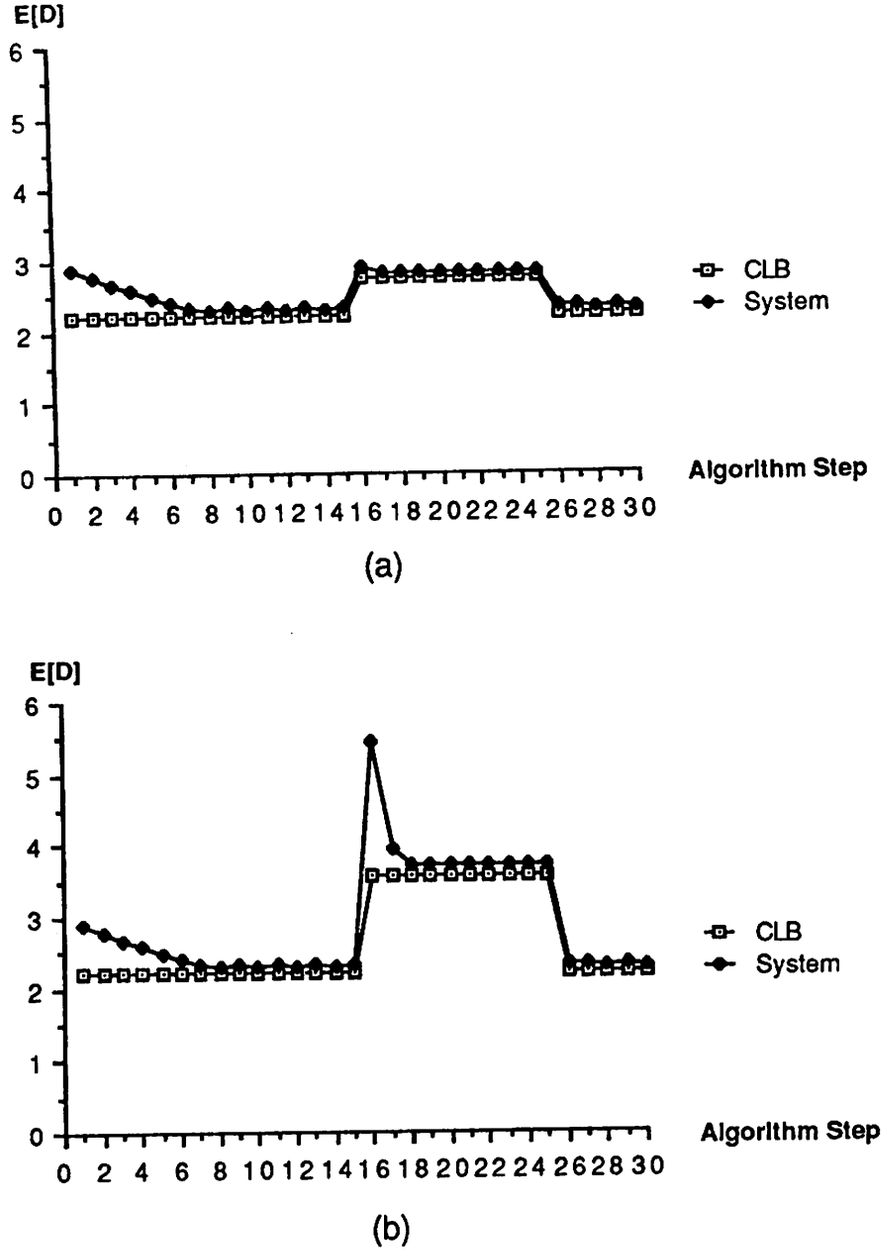


Figure 7 Behavior of Algorithm C in a time-varying environment
(a) Example 3
(b) Example 4

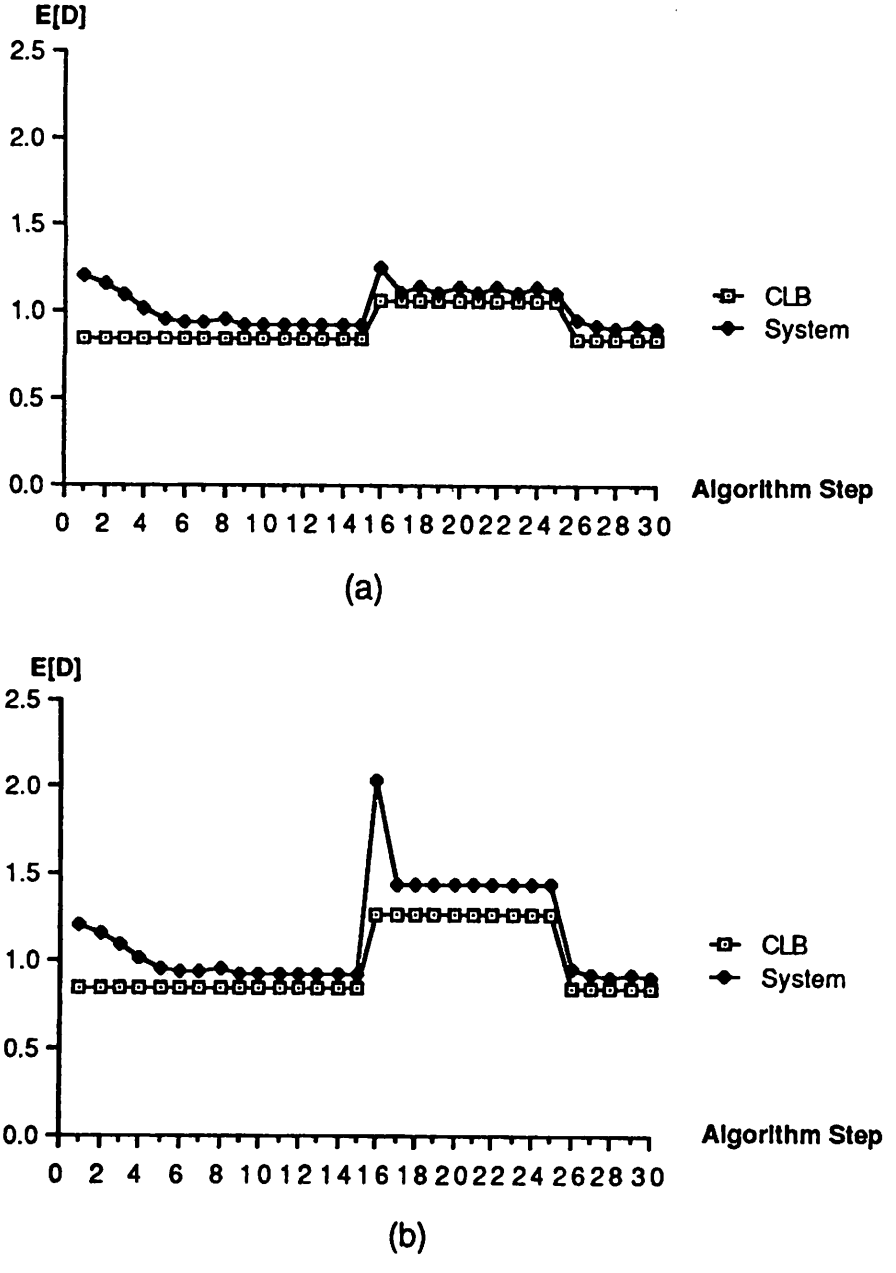


Figure 8 Behavior of Algorithm C in a time-varying environment
(a) Example 5
(b) Example 6