
ELSEVIER

Parallel Computing 26 (2000) 1213–1230

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Rate of change load balancing in distributed and parallel systems ☆

Luis Miguel Campos *, Isaac D. Scherson

Department of Information and Computer Science, University of California, Irvine, CA 92697, USA

Received 1 October 1998; received in revised form 1 April 1999; accepted 1 June 1999

Abstract

Dynamic load balancing (DLB) is an important system function destined to distribute workload among available processors to improve throughput and/or execution times of parallel computer programs either uniform or non-uniform (jobs whose workload varies at run-time in unpredictable ways). Non-uniform computation and communication requirements may bog down a parallel computer if no efficient load distribution is effected. A novel distributed algorithm for load balancing is proposed and is based on local rate of change (RoC) observations rather than on global absolute load numbers. It is a totally distributed algorithm and requires no centralized trigger and/or decision makers. The strategy is discussed and analyzed by means of experimental simulation. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Rate of change load balancing; Dynamic load balancing; Distributed systems; Parallel systems; Simulation; Dynamic work load; Resource management

1. Introduction

We consider the problem of resource management in a multiprocessor system whose operating system supports time sharing among a multiplicity of parallel and/or strictly sequential jobs. We focus on load balancing as an efficient strategy to improve throughput or speed up execution of the set of jobs while maintaining high

☆ This research was supported in part by NASA under grant number NAG5-2561, and by the Irvine Research Unit in Advanced Computing.

* Corresponding author.

E-mail addresses: lcampos@ics.uci.edu (L.M. Campos), isaac@ics.uci.edu (I.D. Scherson).

PII: S0167-8191(00)00036-3

processor utilization. Furthermore, we consider problems with uneven and unpredictable computation and communication requirements. Dynamic load balancing (DLB) is hence essential for the efficient use of multiprocessor systems when running many jobs, some of which correspond to non-uniform problems with unpredictable load estimates [1].

In this paper we consider therefore DLB strategies for minimizing the average completion time of applications running in parallel and improve the utilization of the processing elements (PEs). We consider both the case of distributed systems (network of workstations (NOW)) and highly parallel multicomputer systems. Load Balancing has the potential of improving the application's overall performance by redistributing the workload among PEs. However the load balancing activity comes at the expense of useful computation, incurs communication overhead and requires memory space to maintain load balancing information. To justify the use of load balancing strategies, the accuracy of each balancing decision must be weighed against the amount of added processing and communication incurred by the balancing process.

Load balancing strategies fall broadly into either one of two classifications, namely static or dynamic. A multicomputer system with static load balancing distributes tasks across PEs before execution using a priori known task information and the load distribution remains unchanged at run time. A multicomputer system with DLB uses no a priori task information and satisfies changing requirements by making task distribution decisions during run-time. DLB in turn can be either centralized or distributed. In centralized DLB a single PE is responsible for maintaining global load information and for making load balancing decisions. In distributed DLB, decisions are made locally by each PE and load information is maintained across all PEs which share the responsibility of achieving global load balance.

As load balancing can be viewed as a system function, we also distinguish between implicit and explicit load balancing. Implicit load balancing refers to load balancing performed automatically by the system, whereas explicit means that it is up to the user do decide which tasks should be migrated and when. Most systems only allow for explicit load balancing. Examples include [2,3]. A few systems have implicit load balancing policies, however they are strictly non-preemptive policies. Examples include [4–6]. In this paper, we focus on dynamic, implicit load balancing strategies.

Previously proposed load balancing strategies cast the problem as one of equalizing the absolute number of load units among all PEs. The proposed rate of change load balancing algorithm (RoC-LB) proposed here constitutes a departure from this classical approach. We define load balancing as the activity of migrating load units from one PE to another so that all PEs have at least one load unit at all times. The rationale is that if any given PE is busy executing tasks then the load differential with respect to other PEs is irrelevant since no performance gain can be obtained by transferring load. Hence the decision to initiate load transfers should not depend on a PE's absolute number of load units, but on how the load changes in time.

Our novel approach [7], uses the RoC of the load on each PE to trigger any load balancing activity. It can be described as a dynamic, distributed, on demand,

preemptive and implicit load balancing strategy. Dynamic, because it does not assume any prior task (unit of load in this study) information and must satisfy changing requirements by making task distribution decisions at run-time. Distributed, because all load balancing decisions are made locally and asynchronously by each processor. On demand, because only PEs that "need" tasks are allowed to initiate any migration activity. Preemptive, because running tasks may be suspended, moved to another PE and restarted. Implicit, because all load balancing activity is done by the system, without user assistance.

Furthermore, our proposed RoC-LB strategy achieves the goal of minimizing processor idling times without incurring into unacceptably high load balancing overhead. It does so by striking a balance between the cost of evaluating load information which now is a local activity to each PE, and the cost of transferring tasks across the system.

Moreover, RoC-LB does not create overloads on PEs. These are created by the running applications and RoC-LB will balance the overload when other PEs become available because they are underloaded.

We conduct an experimental analysis of RoC-LB by comparing it against three other well known DLB methods. A special purpose event-driven simulator was used show that our strategy is more effective across a range of network topologies and a variety of workloads.

2. Dynamic RoC-LB

Most reported load balancing algorithms fall under the general diffusion paradigm where workload is redistributed among processors to equalize the absolute number of load units (tasks) across PEs. The goal being to keep processors busy at all times and attempt to load them equally. However, the main goal of maintaining high processor utilization can be achieved by attempting to keep PEs busy regardless of the absolute value of the number of tasks (workload) assigned to each. Furthermore, by relaxing the equal workload constraint, better non-disruptive load balancing may become feasible. Such is the case of the algorithm presented here. Consider an ensemble of intercommunicating processors each of which has an assigned workload. As tasks are executed, some terminate while others spawn new tasks, thus changing the local workload. Predicting local changes in workload can be done by dynamically tallying the RoC of the number of tasks in each processor. Using past information (trend), a processor can estimate its own future overload or its future task starvation, and decide to initiate a task export or import respectively.

2.1. Goals

The rational behind our algorithm, dubbed RoC-LB, is as follows: if any given PE is busy executing tasks then the load differential with respect to any other PE is irrelevant since no performance gain can be obtained by load transfer. Moreover the decision factor to initiate a load transfer request should not be a PE's absolute

load but instead how much its load has changed since the previous time interval. Our goal is then to minimize processor idling time without incurring high load balancing overhead. To do so, an optimal tradeoff between the processing (cost of evaluating load information to determine task migration) and communication (cost of acquiring load information and informing other PEs of load migration decisions) overhead and the degree of knowledge used in the balancing process must be sought.

2.2. LB algorithm

Referring to the pseudocode in appendix and the data structures shown in Fig. 1, the load balancing problem can be thought of as a four phase decision process, namely:

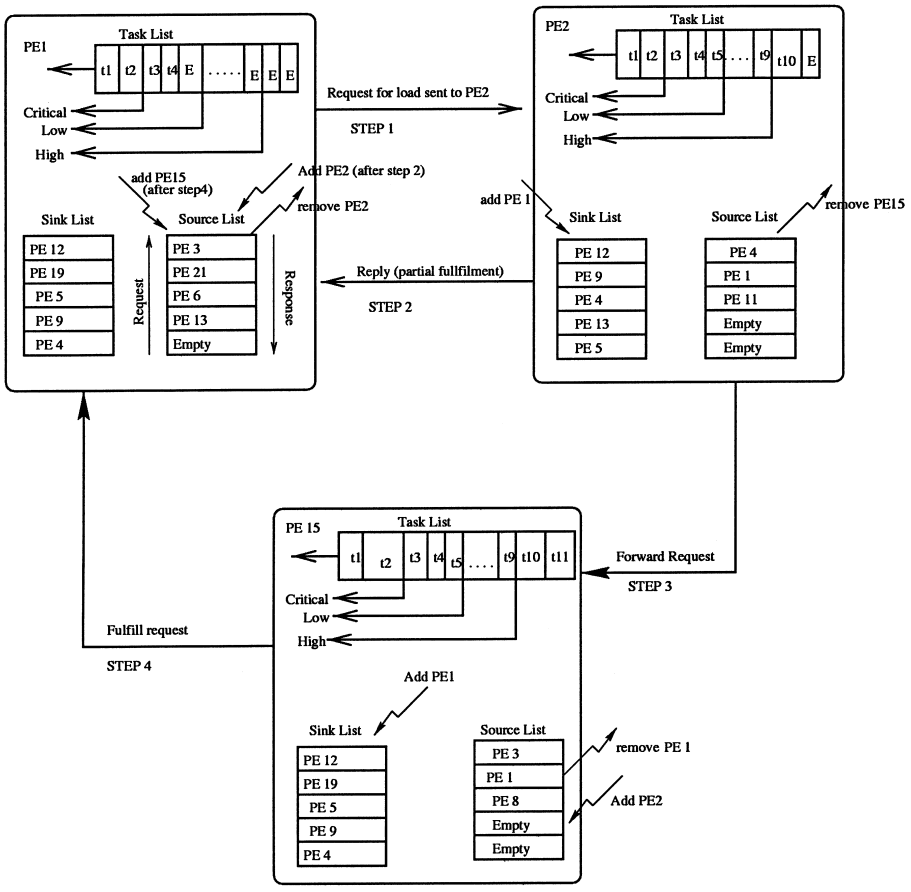

Fig. 1. Load transfer update process.

- when to initiate task migration;
- where, which PE, to send the task migration request;
- how many tasks to migrate;
- which tasks to migrate.

In RoC-LB these four phases occur asynchronously at each PE and are purely distributed. Specific examples which explain the instantiation of Figs. 1 and 2 are given in Section 2.4.

2.2.1. The when phase

At the beginning of each sample interval, each PE calculates the change in its load since the previous interval. Let us call this quantity the difference in load (DL). Two key observations should be made at this point. First, each PE may calculate this quantity independently from other PEs in the system, i.e. there is no concept of global synchronous clock in the system. Second, the length of the sample interval may vary with time for a given PE, depending, for instance, on the number of load requests received or network traffic. As a consequence different PEs may use different sampling intervals at any given time. The sampling interval is therefore an adaptive parameter. The sampling interval is usually measured as a multiple of the time slice duration. The finer the sampling the faster we detect the need for load balancing, but the greater overhead we incur.

Each PE uses its own DL as a predictor for how many tasks will finish in subsequent intervals. Each PE assumes that DL will remain the same forever. Given that

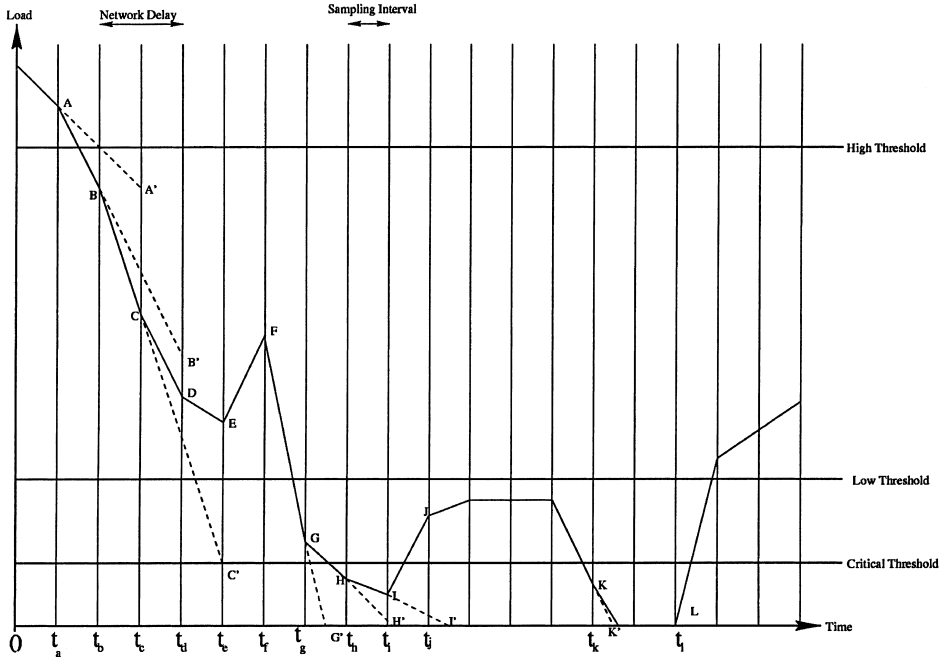

Fig. 2. Possible load variation on a PE.

it calculates the number of sampling intervals it will take to reach an idle state (no tasks to process). If the number of intervals (times its duration) is less than the network delay (ND), then the PE will initiate a migration request.

From the above, one can conclude that only when DL is negative (reduction in load) will a PE consider requesting load from other PEs. Network delay is an adaptive parameter. It is defined as the time it takes between the initiation of a load request and the reception of load. It may vary with time and each PE may use a different value depending on its own record of past load requests. Initially, before any request has been made, ND is set to an arbitrary (positive) value. In our simulations it was set to the maximum time it would take to transfer 1MB between any two PEs.

There are two exceptions to the general rules described above. The first exception is the situation in which a PE will initiate a request for load even though it does not predict it will reach the idle state. In order to understand why it chooses to do so, let us introduce the three thresholds used by the algorithm. high threshold (HT) and low threshold (LT) are used to determine the load status of the processor. If a PE's load is greater or equal to HT it is considered a Source PE. If on the other hand it is less or equal to LT it is considered a sink PE. If its load lies between these two thresholds then the PE is in a neutral state. If however a PE's load falls below a critical threshold (CT) the PE immediately initiates a request for load regardless of the predicted future load based upon the current DL value. We decide to request load even though we do not expect to reach the idle state since even a small change in load at this level will result in immediate task starvation by the PE. The only exception to this rule is that if a PE has already a request pending in the network it will not issue another until either load is received from other PE(s) or the request comes back as unfulfilled. This last observation applies in every case even when a DL's value would deem necessary to issue another request. The other exception to the general rule is the situation when a PE's load level is above HT. In that case even if the value of DL predicts that the PE will reach an idle state we do not initiate a load request because at this load value the value of DL necessary to force the PE to become idle must be quite high. There is a good chance that such a value is rare and short lived in which case during the next sample interval the newly calculated DL will be such that it does not warrant an initiation of a transfer request. By delaying any action until the load value falls below HT we are immune to any spikes in load that may occur over time.

2.2.2. The where phase

Each PE keeps two local tables containing system load information. One contains information regarding the location of sink PEs, called sink table the other of source PEs, called source table. Any PE that initiates a request for load is considered to be a sink by the receiving PE(s). This is true regardless of the level of load at the time the request for load transfer was issued. Selecting the PE to which send a load request is a simple operation. The sink PE (request initiator) selects a source PE from its source table (the first entry in the table) and sends a message requesting load to it. Initially the table is empty since no information regarding the state of the machine is known and therefore a PE is chosen at random. This is not to say that the transfer protocol

is random. It only means that when no information is known regarding the load of any PE in the system then every PE is as likely to be considered a source PE as any other and therefore we chose one among all possible ones at random. This is a very special situation, usually encountered only during startup. Also notice that the PE identifier chosen this way is compared against the entries in its sink table as to prevent any message to be sent to a sink PE.

The selection process described above is used by all PEs alike, whether they are the request initiator or the recipient of a load transfer message who might need to select a source PE to whom forward the message to.

2.2.3. The how many phase

At the beginning of each sample interval each PE calculates its DL. Using DL it computes how many tasks it would have, assuming DL would stay constant, after a length of time equal to ND. Let us call this quantity predicted load (PL). If PL is greater or equal to zero then the PE does not expect to become idle within the next ND period and therefore does not initiate a request for load. If on the other hand PL is lesser than zero the PE requests load, according to the mechanism described in Section 2.2.2, in the amount of **abs**(PL). On the receiver side, a PE will only transfer tasks if its load is above the HT level, in which case it transfers tasks above this value up to the requested transfer amount.

2.2.4. The which phase

Finally, to answer the question of which tasks to migrate, we have decided to migrate older tasks because these tasks have a higher probability of living long enough to amortize their migration cost [8] Age in this case refers to the CPU time a task has used thus far and not how long ago the task was created measured in wall time clock.

2.3. Table update algorithm

Each PE maintains two local tables with information representing its view of the system's load distribution. A tradeoff has to be sought between the temporal accuracy of this information and the overhead required to gather such information. Our proposed algorithm uses a distributed approach to disseminate the global knowledge by relying on each PE's local load information and a low overhead update mechanism.

2.3.1. The message structure

The load request message itself is composed by the following fields:

Sink processor identifier (*SPI*). Unique system wide identifier. Necessary since the message might be forwarded and source PE(s) need to be able to identify the original sink PE.

Number of load units requested (*NLUR*). Updated every time the request is partially fulfilled.

Load state of preceding PE (*LSP_PE*). Used to do updates to the local tables every time the message is forwarded with minimum cost.

Number of forwards (*NF*). Used to avoid infinite looping of requests. The request message is dropped when either NF reaches a predefined maximum value or NLUR becomes zero.

2.3.2. Update protocol

The PE responsible for the initiation of a load transfer request performs the following actions:

When it initiates the request:

1. Creates a message with SPI equal to its own PE id. Initializes NF to zero and NLUR to the number of tasks being requested.
2. Selects the source PE to whom the message will be sent. Removes the entry associated with the selected PE from its source table.
3. Changes its message status to WAITING, preventing it from issuing another request before receiving a reply.

The reason for removing the entry associated with the selected PE in step 2 above is that if the selected PE is able to fill the request (either partially or completely) then it will be added again to the table when the reply arrives. If on the other hand the source is no longer able to fill the request it simply means is no longer a source and therefore should not be an entry in the table. By updating the table upon receiving every reply we are able to keep information about who is currently a source in the system with minimum overhead. Note that, even though the table may be obsolete on some PEs, the process converges to up to date tables over time.

Upon receiving a reply:

1. Adds the id of the sender PE to its source table.
2. If the reply completely fulfills the original request it changes its message status to READY.

When a PE receives a transfer request message it performs the following actions:

1. Updates its own local tables by adding SPI to its sink PE's table and removing it from the source PE's table if present. If it is a forward message, it further updates its local tables based on LSP_PE.
2. Checks its own load status. If it is a source PE.
   - If it can completely fulfill the load transfer request it sends the load to SPI and removes the message from the system.
   - If it can only fulfill the load transfer request partially decrements NLUR by the amount of load units that is able to satisfy.
3. If it is not a source PE or cannot completely fulfill the load transfer request increments NF. If NF is equal to a predefined maximum value it discards the message and replies to SPI stating that the request has been dropped, otherwise sets LSP_PE to its current load status and forwards the message to a source PE selected from its source table.

By incrementing NF and limiting the maximum value it can reach to a predefined value, say NF_Max, we guarantee a deadlock free strategy since any message is

known to be forwarded at most NF_Max times. NF_Max was chosen to be 8 in our simulations.

In Fig. 1 we show the pseudocode for the algorithm, and in Section 2.4 we describe the process by which the internal data structures are updated and a possible load balancing scenario.

2.4. Examples

Fig. 1 corresponds to the situation where a PE (PE_1 in our case) initiates a request for load. In step 1, PE_1 creates a load request message and sends it to PE_2 which is the first entry in its source table and removes that entry from the table. When PE_2 receives the message, it checks its own load and concludes that it can only satisfy the request partially. It adds PE_1 to its sink table, decrements the message field NLUR by the amount of tasks that can be transferred to PE_1, increments the message counter NF and forwards the message to PE_{15} which is the first entry in its own source table. Upon reception of reply from PE_2 by PE_1, an entry for PE_2 is again added to PE_1's source table. (steps 2 and 3). PE_{15} eventually receives the forward message, adds PE_2 to its source table, removes PE_1 from its source table and adds it to its sink table. Since it can completely satisfy the remaining of the request, it removes the message from the system and replies to PE_1 (step 4). When PE_1 receives the reply from PE_{15} it adds this PE's id to its source list and the update mechanism is complete.

Fig. 2 shows a possible load balancing scenario. Let us explain in detail the several keys events depicted in the figure. For this example we chose the network delay to be constant and equal to two times the sampling interval duration. At time $t = t_a$ the PE calculates DL and uses this value to predict its load, A', at $t = t_c$ assuming DL is kept constant. Since A' is $\geqslant 0$ the PE does not issue a load request. The same reasoning applies to points B to F. At $t = t_g$ the value of DL predicts that the load at this particular PE will fall below zero, within the next ND and therefore a request for load is initiated. A reply to the request initiated at this point, $t = t_g$, will not arrive until a network delay interval has elapsed which corresponds to point I in the figure. However the value of DL changes during the next interval (becomes smaller: the slope becomes less steep) and as a consequence at $t = t_h$ the PE still has load to process. Notice that at $t = t_h$ the new value of DL still leads to a PL that is $\leqslant 0$ (point H'). Also notice that at this point, the amount of load available to the PE is now below CT. Either of these two conditions when met lead to the initiation of a load request by the PE. However since in this case there is already a request pending the PE will not issue another one. Between $t = t_i$ and $t = t_j$ the previous load request is satisfied as expected (notice the change in slope, corresponding to the arrival of tasks from other PEs). At time $t = t_k$ the PE predicts again that its load will go below zero and initiates another request. This time the load does actually fall to zero and the PE remains idle for over a sampling interval, or until $t = t_l$ at which point the PE starts processing the tasks transferred from other PE(s).

3. Experimental results

3.1. DLB strategies

RoC-LB is compared against three well known DLB strategies: the gradient model (GM) [9], the sender initiated diffusion (SID) [10] and the central job dispatcher (LBC) [11]. A complete description of the GM, SID and LBC can be found in the literature. For convenience a brief description of each follows below.

GM: In this strategy lightly loaded PEs inform other PEs of their state, and heavily loaded PEs respond by sending a portion of their load to the nearest lightly loaded PE in the system. When execution begins, every PE computes its total load. Two threshold values are used to classify a PE as lightly, heavily, or moderately loaded. The GM strategy uses the concept of proximity, defined as the minimum distance between the current PE and the nearest lightly loaded PE in the system, to initiate the load balancing process. Proximity is measured in terms of the distance between any two directly connected processors. Initially every PE sets its proximity to $d_{\max}$, a constant equal to the network's diameter. A PE's proximity is set to zero if becomes lightly loaded. All other PEs P_i with nearest neighbors n_j compute their proximity as:

$$\text{proximity}(P_i) = \min(\text{proximity}(n_j)) + 1.$$

The GM employs a gradient map of the proximities of underloaded processors in the system to guide the migration of tasks between overloaded and underloaded processors.

SID: The sender-initiated strategy uses a nearest-neighbor approach with overlapping neighborhood domains to achieve global load balancing. A PE is considered overloaded when its load, l_i, is greater than a preset threshold L_{high}. Only overloaded PEs initiate load distribution. The average load in a domain, L_{avg}, is used to determine how many tasks to transfer to the sender's neighbors.

$$L_{\text{avg}} = \frac{1}{K+1}\left(l_p + \sum_{k=1}^{K} l_k\right),$$

where l_p is the load of the sender, K the total number of neighbors, and l_k is the load of PE k. Each neighbor is assigned a weight h_k, according to

$$h_k = \begin{cases} L_{\text{avg}} - l_k & \text{if } l_k < L_{\text{avg}}, \\ 0 & \text{otherwise.} \end{cases}$$

These weights are summed to determined the total deficiency H_{d}, and the proportion of the sender's excess load which is assigned to neighbor $kas\delta_k$ is calculated by:

$$\delta_k = \lceil (lp - L_{\text{avg}})h_k/H_{\text{d}} \rceil.$$

Global balancing is achieved as tasks from heavily loaded neighborhoods diffuse into lightly loaded areas in the system.

LBC: In the central job dispatcher strategy, one of the network processors acts as a central load balancing master. The dispatcher maintains a table containing the number of waiting tasks in each processor. Whenever a task arrives at or departs from a processor, the processor notifies the central dispatcher of its new load state. When a state change message is received or a task transfer decision is made, the central dispatcher updates the table accordingly. The network bases load balancing on this table and notifies the most heavily loaded processor to transfer tasks to a requesting processor. The network also notifies the requesting processor of the decision.

3.2. Simulation models and parameters used

To compare the relative performance of RoC-LB against the other strategies, experimental simulation was used. A special-purpose event driven simulator was constructed. A complete description of the simulator can be found in [12]. For completeness we describe in here the main features of the simulator. The simulator accepts two types of inputs. The first describes the architectural features of the machine being simulated whereas the second aims at describing the workload that is to be run. There are currently two ways of describing the input needed to characterize the machine architecture to the simulator. We can either use a high-level model, such as Log$P$, where we hide for instance all the details related to a particular interconnect topology, or we can use a low-level model where we describe in detail all the components being modeled in our PP system. For the experiments described in this paper we used the low-level model form of input. The machine parameters used in our simulation are as follows:

Topology: Mesh, Hypercube, Fully connected and NOW with an arbitrary interconnection between nodes.

Number of PEs: 16.

Link Bandwidth: 10 Mbps for the NOW and 100 Mbps for the other topologies.

Time slice: 1 s.

Other simulation parameters such as operating system overhead, cache size, etc. are the same as the ones used in [13,14].

To characterize the workload we can use three different models, namely the probabilistic model,the algorithmic/programming language model, and the direct acyclic graph (DAG) model. For the simulations described in here we used only the probabilistic model. In this model a workload description consists of two major components: *job arrival* and *job structure*. The first component describes how jobs are submitted to the system over a period of time. The second component is that of modeling the work requirements of each job. This can be done in a monolithic manner, or else the internal structure of each job can be specified. The most common and clearly identifiable structures are the computational structure (parallelism and barrier synchronization), interprocess communication, memory requirements, runtime, I/O needs, etc.

We decided to use synthetic workloads, as opposed to real workloads traces, because the former offers the convenience of a much more manageable experi-

mental medium that is free of the idiosyncratic site-specific behavior of production traces.

We based our synthetic generated workloads on the work done Harchol-Balter and Downey [8] and Jann et al. [15]. More specifically, we model the inter-arrival time using a Hyper Erlang distribution of Common Order [15] and a task's lifetime using a used-better-than-new-in-expectation type of distribution [8]. For the other simulation parameters, such as the application's degree of parallelism/spawning factor and communication frequency we used Normal and Uniform distributions respectively. This choice of distributions is ad-hoc since no study based on empirical observations could be found in the literature. The remaining parameters of the simulation are algorithm specific. For the RoC-LB the table size was chosen to be 5 and ht, lt and ct, were 25, 10, 4 respectively.

3.3. Experiments description

Three experiments were performed for this paper. In the first one we simulated a stable situation where an initial set of tasks (representing several competing applications) was distributed uniformly among all 16 processors. In the second scenario, the same initial set of tasks was divided among only half of the PEs with the remaining being idle. In both experiments, new applications were not allowed to be submitted to the system. All new tasks were generated internally due to the non-uniform nature of the applications being simulated. As mentioned before, the spawning factor was given by a Normal distribution. Applications are differentiated by both their spawning factor and degree of parallelism. For the third experiment, the possibility of arbitrary arrival times for new jobs was allowed, and the tasks that compose an application were distributed randomly among all PEs. Internal task generation was also allowed naturally. The performance metric we chose was normalized performance, as used in [10,11]. Normalized performance (NP) determines the effectiveness of the load balancing strategy (such that $\mathrm{NP} \to 0$ if the strategy is ineffective and $\mathrm{NP} \to 1$ if the strategy is effective). This is a comprehensive metric; it accounts for the initial level of load imbalance as well as the load balancing overheads. NP is formally defined as:

$$\mathrm{NP} = \frac{(T_{\mathrm{noLB}} - T_{\mathrm{bal}})}{(T_{\mathrm{noLB}} - T_{\mathrm{opt}})}$$

where T_{noLB} is the time to complete the work on a multicomputer network without load balancing, T_{opt} is the time to complete the work on one processor divided by the number of processors in the network and T_{bal} is the time to complete the work on a multicomputer network with load balancing. For each experiment ten runs were performed and the average value for the performance metric was recorded.

3.4. Comparison analysis

Referring to Tables 1–3 and corresponding Figs. 3–5 we can make three main observations:

Table 1
Stable initial distribution, no new arrivals

	Mesh	Hypercube	Fully connected	Network of workstations
Gradient model	0.66	0.69	0.73	0.52
Sender initiated diffusion	0.7	0.71	0.74	0.59
Central job dispatcher	0.59	0.62	0.63	0.37
Rate of change model	0.79	0.79	0.87	0.61

Table 2
Unstable initial distribution, no new arrivals

	Mesh	Hypercube	Fully connected	Network of workstations
Gradient model	0.72	0.74	0.76	0.54
Sender initiated diffusion	0.65	0.65	0.68	0.52
Central job dispatcher	0.65	0.66	0.65	0.45
Rate of change model	0.74	0.75	0.81	0.60

Table 3
Stable initial distribution, new arrivals allowed

	Mesh	Hypercube	Fully connected	Network of workstations
Gradient model	0.7	0.77	0.81	0.54
Sender initiated diffusion	0.73	0.76	0.81	0.58
Central job dispatcher	0.52	0.53	0.53	0.42
Rate of change model	0.82	0.82	0.88	0.62

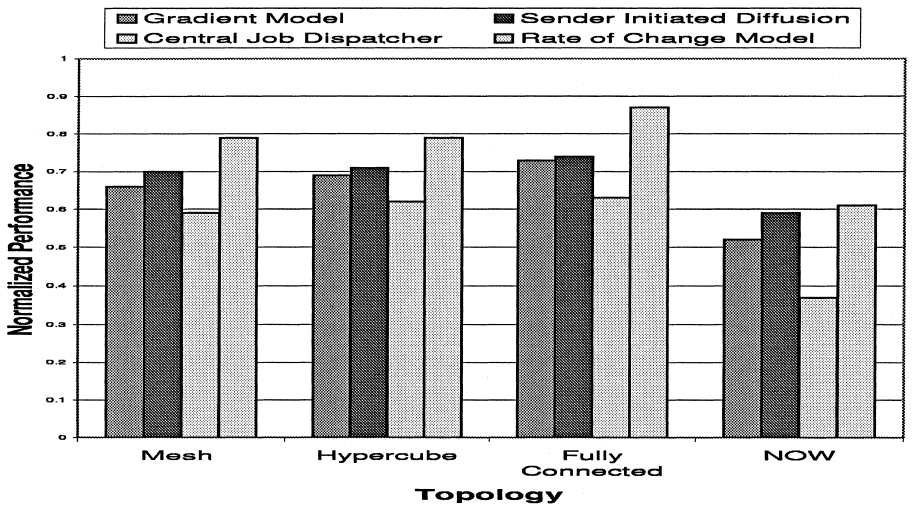

Fig. 3. Initial tasks distributed uniformly among all PEs (stable situation). New job arrivals not allowed.

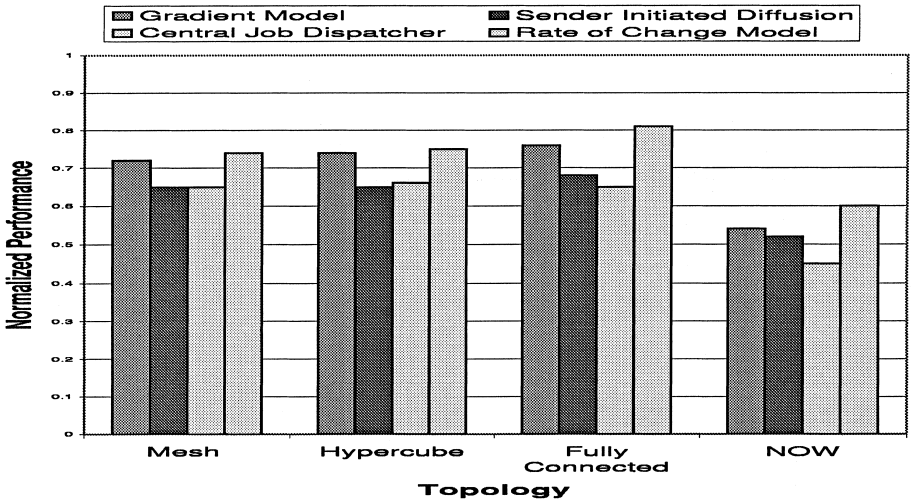

Fig. 4. Initial tasks distributed uniformly among half of the PEs (unstable situation). New job arrivals not allowed.

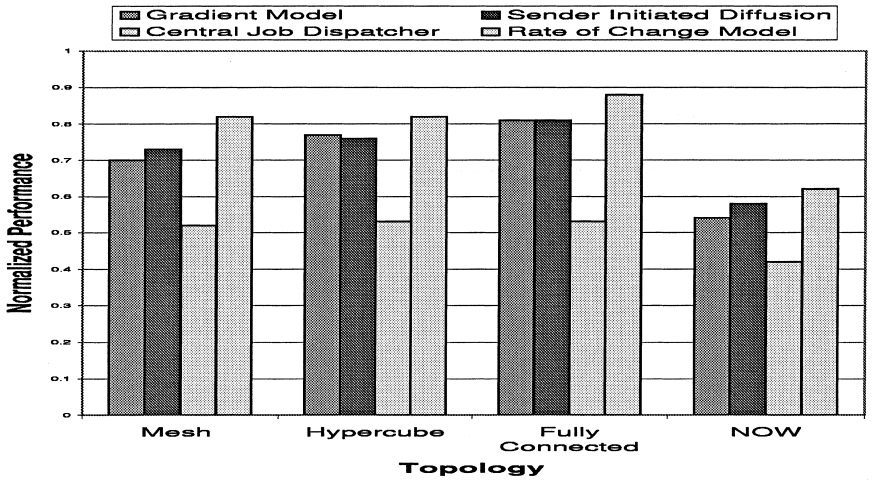

Fig. 5. Initial tasks uniformly distributed among all PEs. Arbitrary arrival of new jobs allowed.

1. In each experiment, a different one of the three prior methods is better than the other two. In experiments 1 and 3 the SID method shows better performance, except for the case of a hypercube topology in which the Gradient method performs slightly better (less than 2%), whereas in experiment 2 the Gradient method performs better than the remaining two across all topologies.
2. RoC-LB is best in *all* cases.

3. RoC-LB's performance is independent of network topology, choice of workload and initial distribution of load. It converges to high resource utilization in all cases.

The lower results obtained for the NOWs should not be surprising due to the higher cost of task migration. The fact that the best results were always obtained for the fully connected topology can be explained by the fact that the mean distance of a PE to another one is greater in a mesh and in a hypercube than in a fully connected topology. So, when a source PE is chosen, in a fully connected topology, this is always at distance 1, whereas in a hypercube, the mean distance of a PE is 2, and even greater in a mesh. To fully validate our approach, DAGs generated from real world parallel applications must be tested.

In summary, RoC-LB proves to be a better approach to load balancing in parallel/distributed systems for the following reasons:

- Does not require synchronization among processors to balance load. It is truly distributed. Decisions on when, where to, how many and which tasks to transfer are made locally.
- Has smallest overhead by virtue of its local decision making process.
- Yields a better utilization of resources independent of network topology and choice of workload.

Acknowledgements

We would like to thank the anonymous referees for their useful comments and advice in improving this article. This work was supported by NASA under grant NAG5-2561 and by the Irvine Research Unit in Advanced Computing.

Appendix A. RoC pseudocode

```
/*Code executed in every processor at every sample interval*/
loadBalancingDecision()
   {
   rate_of_change = current_load - previous_load;
   if ((request_pending = = FALSE) AND
      ( (current_load < critical_threshold) OR
        (current_load – rate_of_change * network_delay < 0) ) )
      {
        initiateRequest();
        request_pending = TRUE;
        my_status = SINK;
      }
   }
/*end transfer Initiation*/
```

```
/* Code executed at the PE responsible for request */
/* initiation */
initiateRequest()
   {
      /* Find a source processor from the table.If empty */
      /* randomly pick one */
      source_PE = selectSourceFromTable();

      /* Fill in request information */
      request.id = my_id;
      request.count = 0;
      request.number_units = high_threshold - current_load;

      /*Since this is the PE initiating the request the next */
      /* field is invalid. */
      /*It will be used if the message is forward */
      request.status_previous_PE = INVALID;

      /* Update table by removing the PE chosen from the source */
      /* table and send request */
      updateTables();
      sendRequest(request);
   }
/* end of request initiation*/

/* Code executed when a PE receives a request for transfer */
/* of load */
requestFilling(message)
   {
      /* Satisfy the request either in whole or partially */
      number_units_satisfied = satisfyRequest();

      /* send message to originator after updating units needed*/
      if ( number_units_satisfied >= 0 )
   {
      message.number_units -= number_units_satisfied;
      sendMessageToOriginator(message);
   }
/* if request not filled completely forward message */
   if ( message.number_units != 0 )
   {
   /* Update message */
   message.count++;

   /*Update tables (source/sink) using the information */
   /* contained in message.*/
   /*Remove transfer originator if present and update based*/
```

```
    /* on message.status_previous_PE */
    updateTables();
    message.status_previous_PE = my_status;
    if ( message.count == MAX_FORWARD )
       {
         /* If the message has been forward MAX_FORWARD times */
         /* then I will indicate that to the original PE as */
         /* indicated by the field message.id */
         sendFinalMessageToTransferOriginator(message);
       }
       else
       {
         /* Choose a source PE from my source table and forward */
         /* message to it*/
         source_PE = selectSourceFromTable();
         forwardMessage(source_PE);
       }
    }
  }
  /* End of filling of request */
```

References

[1] M.H. Willebeek-LeMair, A.P. Reeves, Region growing on a hypercube multiprocessor, in: Proceedings Third Conference Hypercube Concurrent Computations and Applications, 1988, pp. 1033–1042.

[2] E.R. Zayas, Attacking the process migration bottleneck, in: Eleventh ACM Symposium on Operating Systems Principles, 1987, pp. 13–24.

[3] G. Thiel, Locus operating system a transparent system, Computer Communications 14 (6) (1990) 336–346.

[4] A. Tanenbaum, R. vanRenesse, H. vanStaveren, G. Sharp, Experiences with the ameoba distributed system, Communications ACM 33 (1990) 336–346.

[5] M. Litzkow, M. Livny, M. Mutka Condor, A hunter of idle workstations, in: Eighth International Conference on Distributed Computing Systems, 1992.

[6] D.S. Milojicic, Load distribution: implementation for the Mach microkernel, Ph.D. Thesis, University of Kaiserslautern, Kaiserslautern, Germany, 1993.

[7] I.D. Scherson, L.M. Campos, A distributed dynamic load balancing strategy based on rate of change, in: Eighth International Parallel Computing Worshop, 1998.

[8] M. Harchol-Balter, A.B. Downey, Exploiting process lifetime distributions for dynamic load balancing, ACM Transactions on Computer Systems 15 (3) (1990) 253–285.

[9] F.C.H. Lin, R.M. Keller, The gradient model load balancing method, IEEE Transactions on Software Engineering 13 (1) (1991) 32–38.

[10] M.H. WillebeekLeMair, A.P. Reeves, Strategies for dynamic load balancing on highly parallel computers, IEEE Transactions on Parallel Distributed Systems 4 (9) (1993) 979–993.

[11] P.K.K. Loh, W.J. Hsu, C. Wentong, N. Sriskanthan, How network topology affects dynamic load balancing, IEEE Parallel and Distributed Technology 4 (3) (1996) 25–35.

[12] Luis Miguel Campos, Resource management techniques for parallel and distributed systems, Ph.D. Thesis, University of California at Irvine, 1999.

[13] V.L. Reis, I.D. Scherson, Impacts of network latency on parallel virtual memory management, in: Symposium on Performance Evaluation of Computer and Telecommunication Systems, 1998.
[14] L.M. Campos, V.L. Reis, I.D. Scherson, Swap file organizations for parallel virtual memory, in: Parallel and Distributed Computing and Systems, 1997, pp. 209–214.
[15] J. Jann, P. Pattnaik, H. Franke, F. Wang, J. Skovira, J. Riordan, Modeling of workload in MPPs, in: Third Workshop on Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 1291, pp. 95–116, 1997.