Report Number: WUCS-95-05

1995-01-01

A General Matrix Iterative Model for Dynamic Load Balancing

Mark A. Franklin and Vasudha Govindan

Effective load balancing algorithms are crucial in fully realizing the performance potential of parallel computer systems. This paper proposes a general matrix iterative model to represent a range of dynamic load balancing algorithms. The model and associated performance measures are used to evaluate and compare vairous load balancing algorithms and derive optimal algorithms and associated parameters for a given application and multiprocessor system. The model is parameterized to represent three load balancing algorithms - the random strategy, diffusion and complete redistribution algorithms. The model is validated by comparing the results with measured performance on a realistic workload. The parallel... **Read complete abstract on page 2.**

A General Matrix Iterative Model for Dynamic Load Balancing

Mark A. Franklin and Vasudha Govindan

Complete Abstract:

Effective load balancing algorithms are crucial in fully realizing the performance potential of parallel computer systems. This paper proposes a general matrix iterative model to represent a range of dynamic load balancing algorithms. The model and associated performance measures are used to evaluate and compare vairous load balancing algorithms and derive optimal algorithms and associated parameters for a given application and multiprocessor system. The model is parameterized to represent three load balancing algorithms - the random strategy, diffusion and complete redistribution algorithms. The model is validated by comparing the results with measured performance on a realistic workload. The parallel N-body simulation applicaton used for this purpose has a number of interesting properties and is representattive of a wide class of realistic scientific applications. The performance of the three algorithms are compared and optimal algorithm parameters derived for the application. The random strategy outperforms both the diffusion (12% better) and the redistribution (30% better) algorithms and its performance is within 25% of the ideal load balance case. General performance models such as the one presented in this paper can be used to guide the algorithm designer in choosing the best algorithm and associated parameters for a given environment.

A General Matrix Iterative Model for Dynamic Load Balancing

Mark A. Franklin and Vasudha Govindan

WUCS-95-05

February 1995

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130-4899

Submitted to IEEE Transactions on Parallel and Distributed Systems.

A General Matrix Iterative Model for Dynamic Load Balancing[†]

Mark A. Franklin
jbf@random.wustl.edu

Vasudha Govindan
vasu@wuccrc.wustl.edu

Computer and Communications Research Center
Campus Box 1115, One Brookings Drive
Washington University, St. Louis, Missouri 63130

Abstract

Effective load balancing algorithms are crucial in fully realizing the performance potential of parallel computer systems. This paper proposes a general matrix iterative model to represent a range of dynamic load balancing algorithms. The model and associated performance measures are used to evaluate and compare various load balancing algorithms and derive optimal algorithms and associated parameters for a given application and multiprocessor system. The model is parameterized to represent three load balancing algorithms - the random strategy, diffusion and complete redistribution algorithms. The model is validated by comparing the results with measured performance on a realistic workload. The parallel N-body simulation application used for this purpose has a number of interesting properties and is representative of a wide class of realistic scientific applications. The performance of the three algorithms are compared and optimal algorithm parameters derived for the application. The random strategy outperforms both the diffusion (12% better) and the redistribution (30% better) algorithms and its performance is within 25% of the ideal load balance case. General performance models such as the one presented in this paper can be used to guide the algorithm designer in choosing the best algorithm and associated parameters for a given environment.

Keywords: Dynamic load balancing, Performance model, Matrix iterative model, Parallel N-body simulation

---

[†]Submitted to IEEE Transactions on Parallel and Distributed Systems.

[*]This research has been sponsored in part by funding from the NSF under Grant CCR-9021041 and ARPA under contract DABT-93-C0057.

1 Introduction

Effective processor and communication resource utilization is essential in fully realizing the performance potential of parallel computer systems, and central to this is the development of appropriate load balancing (or load sharing [7]) algorithms. The various load balancing algorithms presented in the literature[4, 7, 13, 14] generally focus on distributing the workload in some equal fashion among the available processors. In this paper, we focus on load balancing of cooperating tasks belonging to a single application running on multiple processors[2, 8, 11, 19]. We develop a simple but effective matrix iterative model to represent a wide range of load balancing algorithms. The model and associated performance measures are used to quantitatively compare various load balancing algorithms. For a given application and multiprocessor system, the model is used to choose a suitable algorithm and to tailor the algorithm for "best" performance. The model is validated using a parallel N-body simulation implementation on a network of workstations.

The load balancing model presented in this paper serves as an effective performance evaluation tool in the design of efficient algorithms for parallel computing systems. Following are the highlights of the model and performance scheme presented in this paper:

- The matrix iterative formulation is compact and intuitive and can be parameterized to represent a range of load balancing algorithms. Unlike other models in literature for analyzing load balancing algorithms, our model is easily applicable to a heterogeneous processor systems. We apply the model to a parallel application running on a heterogeneous workstation network.

- The performance evaluation scheme incorporates all aspects of load balancing. The load balancing algorithm is evaluated taking into account the performance benefits as well as the overhead (in terms of computation and communication costs) of load balancing. The characteristics of the parallel application and the computing platform are also incorporated into the performance model. The performance measure derived therefore reflects the overall performance of the system.

- The model is validated by comparing the results with measured performance on a realistic workload. The parallel N-body simulation application used for this purpose has a number of interesting properties and is representative of a wide class of realistic scientific applications.

Load balancing algorithms can be broadly classified into *static* and *dynamic* strategies [4, 12]. In static schemes, assignment of tasks to processors is made before execution and it is fixed throughout the execution time. Dynamic (or Adaptive) load balancing schemes periodically reassign tasks as needed during execution to achieve balance. Dynamic load balancing strategies can also be classified as centralized or distributed based on where load control is exercised. Centralized load balancing algorithms have a "master" node which allocates tasks to other processors thus maintaining a balanced workload. In distributed algorithms, each processor makes an independent load balancing decision based

on a combination of its own load and some system-wide load information. Another classification of dynamic schemes relates to where task transfer between processors is initiated. In sender initiated schemes heavily loaded nodes initiate transfer of tasks to other nodes. In receiver initiated schemes, lightly loaded nodes request other nodes to send tasks. The principal focus of this paper is the analysis of dynamic, distributed, sender-initiated load balancing schemes. The model presented here, however, can be extended to represent other load balancing schemes.

There have been several studies on the performance analysis and modeling of load balancing algorithms[1, 5, 6, 7, 13, 17]. Eager et.al[7] use analytic queueing models to compare the performance of adaptive load sharing algorithms of varying complexity. Based on their analysis, they conclude that extremely simple adaptive algorithms yield dramatic performance improvements. The models, however, simplify several aspects of a real system. It is hard to represent complex system constraints and overheads in analytic queueing models. Ahmad et.al[1] present and, using simulation techniques, solve a queueing oriented state transition model of a range of sender initiated balancing techniques. The model focuses on the balanced scheduling of independent tasks. The state transition model formulation and techniques differ from the iterative approach taken here. The iterative approach used in this paper is more flexible and can be used to model load balancing on a heterogeneous system of dependent tasks.

Cybenko[6] presents a model for a simplified diffusion load balancing scheme where the movement of tasks between processors is analogous to the physical process of diffusion, and using matrix iterative techniques shows that the load distribution converges to the uniform distribution. The model does not take into account task dependencies and load balancing costs and is thus of limited use in real parallel environments. Willbeck-LeMair and Reeves[17] study the performance benefits and overheads of five dynamic load balancing algorithms. The five algorithms illustrate the tradeoff between the accuracy of load balancing decisions and the costs incurred by the balancing processes. The paper presents a comprehensive *qualitative* analysis of the various phases of load balancing for each of the five algorithms. Performance of these algorithms on an Intel iPSC/2 hypercube are presented. Our work is also focused on the tradeoff between performance benefits and overheads of load balancing. Our load balancing model and performance measures provide a platform for *quantitative* study of this tradeoff. The model is general and is applicable to a wide range of algorithms and applications.

Here, we note that none of the above models extend to heterogeneous processor systems[1]. The matrix iterative model presented here effectively incorporates processor heterogeniety in load balancing decisions. The next section, begins by presenting a matrix iterative model for load balancing. The model can be parameterized to reflect any dynamic, sender-initiated load balancing algorithm. In section 3, three load balancing schemes are considered - the random strategy, the diffusion algorithm and the complete redistribution algorithm, and the model is parameterized to represent these algorithms.

[1] In this context, we assume that the processors are functionally identical but differ in their computing ability.

Using the parallel N-body simulation application as an example (section 4), the model is validated (section 5) and used to evaluate the three load balancing algorithms and derive an optimal set of load balancing parameters. Parallel N-body simulations have non-uniform, dynamically changing computation and communication requirements and are representative of realistic scientific applications.

2 A Load Balancing Algorithm Model

In this section, a general matrix iterative model is developed to represent and evaluate a range of dynamic load balancing techniques. We assume that the application can be broken into a large number of tasks (large compared to the number of processors). The tasks have identical computation and communication requirements and can be executed on any of the available processors. The processors may be of different computation and communication capabilities.

The set of processors in the system is denoted by $\mathbf{P} = \{P1, P2, P3 \ldots\}$. The computing ability of processor Pi is denoted by C_i, a number proportional to the number of operations executed by the processor in unit time. The set $\mathbf{P}$ is ordered in decreasing order of processor power (*i.e.*, $C_1 > C_2 > C_3 \ldots$). The time taken to execute a task on Pi is inversely proportional to its computing ability, C_i. To achieve load balance in a heterogeneous system, the processors are allocated tasks proportional to their respective computing abilities. For example, if Pi is twice as fast as Pj (*i.e.*, $C_i = 2C_j$), Pi should be allocated twice as many tasks as Pj so that they take approximately the same time to execute the tasks.

The load balancing algorithm is invoked at instants denoted by k ($k = 1, 2, 3, \ldots$). The algorithm may be invoked at regular intervals, at the end of each iteration in iterative applications, or may be triggered by an external event. The matrix iterative model developed in this paper keeps track of the task distribution at each processor in the system at each of these instants k, when the load balancing algorithm is invoked. The state of the system at k is represented by the task distribution vector, $w(k)$, where $w_i(k)$ is the number of tasks associated with processor Pi at k. The model computes the task distribution, $w(k)$ for all k in terms of the task transfer between processors due to load balancing, and due to the arrival and departure of tasks at each processor resulting from the dynamics of the application. The load imbalance in the system is mainly due to the application dynamics and, in some systems, due to the dynamically changing processor capabilities.

Task distribution at time instant $(k+1)$, $w(k+1)$, can be expressed as a function of (1) $w(k)$, the task distribution at k, (2) the task transfer due to load balancing, and (3) the arrival and departure of tasks between the kth and the $(k+1)$th invocation due to

the dynamics of the application. For processor Pi,

$$\begin{aligned} w_i(k+1) \;=\; w_i(k) \;&-\; \text{tasks sent by } Pi \text{ (due to load balancing)} \\ &+\; \text{tasks received by } Pi \text{ (due to load balancing)} \\ &+\; \text{tasks arriving during } [k, k+1) \text{ (due to application)} \\ &-\; \text{tasks departing during } [k, k+1) \text{ (due to application).} \end{aligned} \tag{1}$$

The number of tasks transferred between processors due to load balancing is determined by the load balancing algorithm used while the arrival and departure of tasks depends on the dynamics of the application program. The load balancing algorithm attempts to transfer tasks to compensate for the imbalance resulting from the dynamically changing application requirements and can be described in terms of the following three phases with the various algorithms differing in the implementation of these phases. The different implementations reflect the tradeoffs between balancing accuracy and algorithm overhead costs.

1. *State Information:* Load balancing algorithms have different system state information requirements and also differ in how they acquire the information. Some algorithms require a measure of the average workload in all or a subset of processors in the system, while others may require the actual workload distribution over all processors in the system. $\bar{w}(k)$ (defined more precisely later) is a vector representing the ideal (load balanced) task distribution where the number of tasks on each processor is proportional to its computing ability. However, in algorithms where the state information gathering scheme is based on partial information, each processor's perception of the system load may be different. We represent this "approximate" version of the ideal task distribution by the vector $\tilde{w}(k)$.

2. *Processor Participation:* When a load balancing algorithm is invoked, each processor decides if it wants to participate in load balancing based on the system state information available. In some algorithms, only the heavily loaded processors participate in load balancing while in others, all processors are involved in redistributing the workload. Most algorithms define a task threshold, $w^*(k)$. Processors participate in load balancing if their load is greater than the threshold. The processor participation in load balancing at the kth invocation of the algorithm is represented by a matrix $A(k)$. For a p processor system, the $p \times p$ matrix $A(k)$ consists of ones and zeroes such that a diagonal element of $A(k)$ is set to one if the corresponding processor decides to participate in load balancing; all other elements are set to zero. The selection of a matrix representation here simplifies the expressions presented later.

3. *Task Migration:* Each processor that decides to participate in load balancing has to compute (a) the number of tasks to transfer and, (b) the task destinations. The migration decisions are typically based on the system state information available. The techniques used in the decision making process range from simple heuristics (e.g.,

random destination selection) to complex optimization algorithms (e.g., simulated annealing[18].). Task migration decisions are represented by a $p \times p$ matrix (for a p-processor system), $M(k)$. M_{ij} is the fraction of its tasks that processor Pi decides to send to Pj (*i.e.*, Pi sends $M_{ij}(k)w_i(k)$ tasks to Pj). Often, task migration decisions are constrained by the structure imposed by the algorithm or the system topology. We define a topology matrix H to incorporate these constraints. H is a $p \times p$ matrix describing the allowable task migration paths. An element of H, H_{ij} is equal to one if processors Pi and Pj are allowed to exchange tasks; otherwise, H_{ij} is set to zero. Therefore, in constructing matrix M, M_{ij} is nonzero only if the corresponding element in the H matrix, H_{ij} is equal to one.

Table 1 describes the variables used in the general load balancing model developed here. Equation 1 expressing the number of tasks on Pi at instant $(k+1)$th can be written as follows:

$$w_i(k+1) = w_i(k) - \sum_{(j \neq i)} M_{ij}(k)A_{ii}(k)w_i(k) + \sum_{(j \neq i)} M_{ji}(k)A_{jj}(k)w_j(k) + \lambda_i(k) - \mu_i(k) \quad (2)$$

where $\lambda(k)$ and $\mu(k)$ are vectors denoting the application task arrival and departure rates respectively and represent the dynamically changing workload requirements of the application. The two summation terms represent load balancing activity. The first summation term corresponds to the sums of all tasks transferred from processor Pi to other processors while the second summation represents the sum of all tasks transferred to Pi from other processors. Defining $M_{ii}(k) = -\sum_{j \neq i} M_{ij}(k)$, Equation 2 can be rewritten as:

$$w_i(k+1) = w_i(k) + \sum_{all\ j} M_{ji}(k)A_{jj}(k)w_j(k) + \lambda_i(k) - \mu_i(k) \quad (3)$$

In vector form, this becomes:

$$w(k+1) = w(k) + [M(k)]^T A(k)w(k) + \lambda(k) - \mu(k) \quad (4)$$

The matrices $M(k)$ and $A(k)$ reflect the transfer of tasks initiated by the load balancing algorithm and can be parameterized to represent various load balancing schemes. $M(k)$ and $A(k)$ are typically functions of the task distribution, $w(k)$, and the decision making scheme of the load balancing algorithm. The matrix formulation derived here is simple and intuitive and can represent a range of load balancing algorithms. The algorithms can be compared quantitatively by solving (either analytically or numerically) the corresponding models.

Load balancing decisions may be based on local or global knowledge of the system state information. The "ideal" (load balanced) distribution, $\bar{w}(k)$, represents global state information since it is computed based on the load distribution on all processors in the system. $\bar{w}_i(k)$, the "ideal" load on processor Pi is given by:

$$\bar{w}_i(k) = \frac{C_i}{\sum_{all\ Pj} C_j} \times \sum_{all\ Pj} w_j(k). \quad (5)$$

Table 1: Variable Definitions

Computing System:	
$\mathbf{P}$	The set of processors, $\{P1, P2, P3, \ldots Pp\}$
C_i	A number indicating computing power of processor Pi.
H	Topology Matrix.
	$H_{ij}(k) = \begin{cases} 1 & \text{if } Pi \text{ and } Pj \text{ are allowed to exchange tasks} \\ 0 & \text{otherwise} \end{cases}$
Task Distribution:	
$w(k)$	Task distribution vector at k.
	$w_i(k)$ = Number of tasks on processor Pi.
$\bar{w}(k)$	Ideal (load balanced) task distribution
	Each processor has tasks proportional to its computing power
	$\bar{w}_i(k)/C_i = \bar{w}_j(k)/C_j$
$\tilde{w}(k)$	Ideal task distribution based on local knowledge
	Pi believes that it should have $\tilde{w}_i(k)$ tasks for ideal task distribution.
$w^*(k)$	Task threshold vector.
	Pi participates in load balancing if $w_i(k) > w^*{}_i(k)$.
	The actual definition of this threshold is algorithm dependent.
Application Dynamics:	
$\lambda(k)$	Task arrival vector.
	$\lambda_i(k)$ = No. of tasks arriving at Pi during $[k, k+1)$
$\mu(k)$	Task departure vector.
	$\mu_i(k)$ = No. of tasks departing from Pi during $[k, k+1)$
Load Balancing Decisions:	
$M(k)$	Load balancing task transfer matrix.
	$M_{ij}(k)$ $w_i(k)$ is the number of tasks transferred from Pi to Pj at k.
$A(k)$	Processor participation matrix
	$A_{ij}(k) = \begin{cases} 1 & \text{if } i = j \text{ and } Pi \text{ participates in load balancing} \\ 0 & \text{otherwise} \end{cases}$

The first term is the fraction of the total compute power associated with processor Pi and the second term ($\sum w_j(k)$) is the total number of tasks in the system.

Acquiring global information is often expensive (due to the additional communication involved) or may result in stale information (due to communication delays). Several algorithms use local load information, the task distribution of processors in a certain neighborhood, as an approximation of the overall system load. The 1-step neighborhood of processor Pi is denoted by the set $\mathrm{N}_i^{(1)}$ and the set is defined by the topology matrix H. Two processors Pi and Pj are 1-step neighbors if the topology matrix entries $H_{ij} = H_{ji} = 1$. Therefore, $\mathrm{N}_i^{(1)}$ consists of all such processors "adjacent" to Pi as defined by the topology matrix. Pi collects load information from its 1-step neighbors and computes its load in an "ideal" distribution. The ideal distribution based on (1-step) local information is denoted by $\tilde{w}(k)$ and is computed as follows:

$$\tilde{w}_i(k) = \frac{C_i}{C_i + \sum_{all\ Pj \in N_i^{(1)}} C_j} \left[w_i(k) + \sum_{all\ Pj \in N_i^{(1)}} w_j(k) \right] \tag{6}$$

Again, the first term denotes the fraction of the total compute power in the set $Pi \cup N_i^{(1)}$ associated with Pi and the second term is the total number of tasks in the set $Pi \cup N_i^{(1)}$. Note that 2-step, 3-step, etc., neighborhoods can also be defined based on how much overhead seems reasonable in a given situation. Load balancing algorithms use global ($\bar{w}(k)$) or local ($\tilde{w}(k)$) load information based on the cost, availability and accuracy of global and load information.

3 Load balancing algorithms

3.1 The Random Load Balancing Algorithm

The random strategy is based on a set of simple heuristics. A processor Pi decides to send some of its tasks to another processor if the number of tasks assigned to it, $w_i(k)$, is greater than a certain threshold number of tasks, $w^*{}_i(k)$. The threshold is typically based on the system load information at each processor. If global information is available, w^* is a function of $\bar{w}(k)$ (e.g., $w^*{}_i(k) = 1.1 \times \bar{w}_i(k)$). The threshold may also be based on local load information (e.g., $w^*{}_i(k) = 1.2 \times \tilde{w}_i(k)$). The number of tasks sent out is moderated by the parameter α; a fraction α of the number of tasks over the threshold value is sent to other processors. Thus, processor Pi sends out $\alpha(w_i(k) - w^*{}_i(k))$ tasks if $(w_i(k) > w^*{}_i(k))$. It chooses a destination processor randomly from the set of "neighbor" processors as defined by the set $N_i^{(1)}$. Figure 1 summarizes the random load balancing strategy.

An element of the matrix $A_{ij}(k)$, defined in section 2, representing processor participation in load balancing, is equal to 1 if $i = j$ and $w_i(k) > w^*{}_i(k)$. All other elements in A are equal to zero. Processor Pi may receive tasks from each of its neighbor processors. This can be expressed by defining the matrix $M(k)$ such that an element of the matrix,

Each Processor Pi of p processors :
If $(w_i(k) \geq w^*(k))$
begin
Pick at random, a processor $j \in N_i^{(1)}$
Send $\alpha(w_i(k) - w^*{}_i(k))$ tasks to Pj;
end.

Figure 1: The Random Load Balancing Strategy

$M_{ij}(k)$ is α if H_{ij} is 1 and if Pi selects neighbor processor Pj as the recipient of its tasks. Equation 2 can now be rewritten as:

$$\begin{aligned} w_i(k+1) &= w_i(k) - \alpha A_{ii}(k)\ (w_i(k) - w^*{}_i(k)) \\ &\quad + \ \textstyle\sum_{j \neq i} \alpha A_{jj}(k) M_{ji}(k)\ (w_j(k) - w^*{}_i(k)) \\ &\quad + \ \lambda_i(k) - \mu_i(k) \end{aligned} \tag{7}$$

The diagonal elements of $M(k)$, $M_{ii}(k)$, are set to $-\alpha$ indicating that $\alpha(w_i(k) - w^*{}_i(k))$ tasks are sent out of processor Pi. The above equation reduces to the vector form,

$$w(k+1) = w(k) + [M(k)]^T A(k)[w(k) - w^*(k)] + \lambda(k) - \mu(k) \tag{8}$$

Equation 8 is similar to equation 4 with $w(k)$ in the second term replaced by $[w(k) - w^*(k)]$ (*i.e.,* the task transfer due to load balancing is proportional to the difference between the current task distribution and the threshold distribution). The selection of w^* and α determines the number of tasks transferred, and therefore, the balance achieved and the cost for load balancing. In section 5, we use equation 8 along with performance measure defined in section 4.2 to evaluate the load balancing scheme and to compute optimal values of α and w^* for the parallel N-body simulation example.

3.2 Diffusion Algorithm

The diffusion algorithm is analogous to the physical process of diffusion where tasks flow between processors with excess tasks diffusing to neighboring processors that have fewer tasks. Cybenko[6] showed that the diffusion algorithm applied to a system with uneven load distribution and identical processors will eventually result in a balanced load distribution. Each processor examines the task average of all its neighbor processors and sends out tasks if its load is greater than a certain threshold, a function of the local load information. It sends out tasks to all neighbors that have load less than the average local load (appropriately weighted by the processor's computing capability). This differs from the previously described random algorithm where tasks were sent out to a random neighbor. Figure 2 describes the diffusion algorithm in pseudocode form.

Each processor Pi:
If $(w_i(k) \geq w^*{}_i(k))$
begin
Compute ith row of $M(k)$ such that:
If $(j \in N_i^{(1)})$ and $(w_j(k) < w^*{}_i(k) \times \frac{C_j}{C_i})$
$M_{ij}(k) \propto \tilde{w}_i(k) \times \frac{C_j}{C_i} - w_j(k)$, and $\sum M_{ij}(k) = 1$
Send $M_{ij}(k) \times (w_i(k) - \tilde{w}_i(k))$ tasks to Pj.
end.

Figure 2: Diffusion load balancing algorithm

Pi decides to send out tasks if its load, $w_i(k)$ is greater than the certain threshold, $w^*{}_i(k)$. The threshold is typically a function of the local load information, $\tilde{w}_i(k)$. Therefore, a diagonal element of the processor participation matrix, $A(k)$, is equal to one if the corresponding processor load is greater than its task threshold; all other elements are zero (*i.e.*, $A_{ij}(k) = 1$ if $i = j$ and $w_i(k) > w^*{}_i(k)$).

If Pi decides to send out tasks, it sends all tasks above the local load average, (*i.e.*, $[w_i(k) - \tilde{w}_i(k)]$ tasks; $\tilde{w}_i(k)$ defined in Equation 6). The task destination is computed based on the task distribution of its neighbors. Pi sends tasks to Pj if the load on Pj is less than the local load average, $\tilde{w}_i(k)$, weighted appropriately by Pj's computing ability (*i.e.*, if $w_j(k) < \tilde{w}_i(k) \times (C_j/C_i)$). The number of tasks sent to Pj is proportional to the difference, $(\tilde{w}_i(k) \times (C_j/C_i) - w_j(k))$. The matrix $M(k)$ is defined as follows:

$$M_{ij}(k) \quad \propto \quad \tilde{w}_i(k) \times \frac{C_j}{C_i} - w_j(k), \tag{9}$$

and,

$$\sum_{i \neq j} M_{ij}(k) = 1 \tag{10}$$

The diagonal elements of M, $M_{ii}(k)$, are all set to -1 because $w(k) - \tilde{w}_i(k)$ tasks are transferred out of processor Pi. The load balancing equation 4 can be rewritten for the diffusion load balancing algorithm as follows:

$$w(k+1) = w(k) + [M(k)]^T A(k)[w(k) - w^*(k)] + \lambda(k) - \mu(k) \tag{11}$$

The equation 11 is similar to equation 4 with $w(k)$ in the second term replaced by $[w(k) - w^*(k)]$. The selection of w^* determines the number of tasks transferred and therefore, the balance achieved and the cost for load balancing. In section 5, we use equation 11 along with performance measure defined in section 4.2 to evaluate the load balancing scheme and to compute optimal values of w^* for the parallel N-body simulation example.

If $(w_i(k) > w^*_i(k))$ for any Pi,
begin
Find $M(k)$ such that:
$w(k) + [M(k)]^T w(k) \simeq \bar{w}(k)$
For all $j \neq i$, if $M_{ij}(k) > 0$,
Send $M_{ij}(k) w_i(k)$ tasks to Pj.
end.

Figure 3: The Complete Redistribution Algorithm

3.3 Complete Redistribution

The complete redistribution algorithm presented here requires complete (global) state information (*i.e.*, $w_i(k), \forall i$). The algorithm is activated if the load on any of the processors exceeds the threshold $w^*_i(k)$. The condition reflects imbalance in the overall system. All processors participate in load balancing. When the algorithm is activated, the processors use a set of heuristics or an optimization algorithm to compute the load balancing matrix $M(k)$ that results in a uniform, balanced task distribution. The processors send out tasks according to the corresponding entries in the $M(k)$ matrix. The redistribution algorithm is described in pseudocode form in Figure 3.

Since all processors participate in load balancing, the processor participation matrix, $A(k)$, is a diagonal matrix of ones if the load balancing algorithm is activated (*i.e.*, $A_{ii}(k) = 1 \ \forall i, if w_j(k) > w^*_j(k)$ for some j and $A_{ij}(k) = 0$ if $i \neq j$). The load balancing matrix $M(k)$ may be computed in several ways. For example, all tasks may be "pooled" together on a master processor and then redistributed evenly using some static load balancing algorithm. Alternately, the computation of $M(k)$ can be formulated as an optimization problem where optimum transfer of tasks is computed subject to certain constraints. Each processor Pi executes the optimization algorithm to compute the ith row of $M(k)$ such that the norm

$$\| \ (w(k) \ + \ [M(k)]^T w(k)) \ - \ \bar{w}(k) \ \| \tag{12}$$

is minimized, subject to the constraints defined by the topology matrix H. Another option is to run the optimization algorithm on a selected processor and communicate the results to all the other processors. After computation of $M(k)$, all processors would participate and transfer tasks as required. In our implementation of the algorithm for the N-body simulation application, the task information is communicated to processor $P1$ and $P1$ computes the M matrix using the orthogonal recursive bisection technique[8] and communicates the appropriate rows of the matrix to the other processors. The processors then participate in task transfer as required.

The load balancing model for redistribution algorithm is the same as that of equation 4 and is repeated below.

$$w(k+1) = w(k) + [M(k)]^T A(k) w(k) + \lambda(k) - \mu(k) \tag{13}$$

The matrix $M(k)$ is computed such that the term (12) is minimized.

4 Case Study: The N-body Problem

4.1 Parallel N-body Simulations

In this section a parallel implementation of the N-body problem is briefly described. The load balancing model is applied to this problem and relative performance of the random, diffusion and redistribution load balancing algorithms is evaluated and compared with the experimental results. The model is thus validated and its applicability to a real application demonstrated. Optimal load balancing parameters for the application are derived and the general approach to obtaining such parameters is discussed.

N-body techniques study the evolution of a system of particles where there exists a force (e.g., gravitational) between every pair of particles. Such techniques are widely applied to a number of problems in astrophysics, fluid dynamics, etc. They require dynamically changing, non-uniform, intense computation and long range communication and are therefore, good candidates for use as parallel computer benchmarks. Due to their non-uniform, time varying computational requirements, some form of dynamic load balancing is necessary to obtain good performance on parallel systems.

The N-body simulation discussed here is an implementation of the Barnes-Hut hierarchical algorithm[3] for a two dimensional gravitational N-body problem. The input consists of the mass, initial position and initial velocity of the particles distributed over a finite physical domain. The simulation proceeds over a number of time-steps (or iterations), each time-step computing the net force on every particle and updating its position and other attributes. The simulation enables us to study the evolution of such a system over time.

The computing platform consists of up to 16 Sun/Sparc workstations networked by a standard ethernet. The application program is implemented under the PVM[16, 10] environment. PVM is a programming environment that enables a set of networked computers to be used as a single concurrent computing resource. The processors in our network are of unequal computing abilities.

In the parallel implementation of N-body simulation, the entire physical domain is divided into smaller regions and each processor is allocated one such region. Each processor is responsible for all the computation associated with the particles present in its region. However, to compute the forces on a single particle, the position of every other particle needs to be known and this involves exchange of information between every pair of processors. The iterations in the N-body simulation algorithm are denoted by k. It is assumed that the load balancing algorithms are invoked every iteration but are executed only if the load imbalance exceeds the thresholds specified by the algorithms. The count k in the load balancing model coincides with the iteration count. The computations associated with a single particle are referred to as a "task" with each processor having as many tasks as there are particles allocated to it. The task distribution is denoted by the

```
On each processor, each iteration:
    Begin
    Exchange particle information with other processors.
    Spawn off a task for every particle.
    Each task:
        Compute resultant force on particle.
        Update position and velocity of particle.
        If particle crosses processor boundary,
          Send particle to appropriate processor
    Receive particles (if any) from other processors.
    Execute load balancing algorithm.
    End.
```

Figure 4: N-Body Algorithm

vector $w(k)$.

The initial distribution is load balanced, (*i.e.,* the number of particles allocated to a processor is proportional to its computing capabilities). At the beginning of each iteration, processor Pi has a certain number of particles (and associated tasks) within its spatial domain. The exchange of particle information between processors acts as an implicit synchronization point at each iteration. Each task computes the resultant force on a particle and updates its position and velocity. The processor then checks for its particles that have crossed out of its space domain and transfers such particles to the appropriate processors. This domain crossing causes an imbalance in the workload distribution among processors with model variables $\lambda(k)$ and $\mu(k)$ representing the system particle dynamics at iteration k in terms of their processor (and therefore spatial domain) association. Estimates of $\lambda(k)$ and $\mu(k)$ can be obtained by collecting relevant data on a test run of the application or some scaled down version of the application. For the N-body simulation example, there is a strong correlation between the number of tasks on a processor, $w_i(k)$, and the number of tasks that leave the processor, $\mu_i(k)$. Typically, the tasks leaving a processor are equally likely to go to any one of its adjacent processors.

Load balancing, in this case, consists of attempting to reassign spatial domains for each processor so that the number of particles associated with the processors is balanced. This effectively is a task transfer operation and is represented by the matrix $M(k)$ in the model. For the N-body application, only pairs of processors that are allocated adjacent domains in space are allowed to exchange tasks. This is captured by appropriately defining the topology matrix, H. The topology matrix constraints the load balancing activity and the m matrix.

The dynamics of the N-body simulations depend on the initial distribution of particles. Our workload simulates the evolution of 1024 particles with an initial mass, velocity and spatial distribution based on the Plummer's model[15]. Plummer's model is a standard

Table 2: Performance Measure: Definitions

max_iter	total number of iterations.
$T^{(p)}_{total}$	total execution time. on p processors.
$T^{(p)}_{appl,i}(k)$	time for application task execution for iteration k, on Pi, for a p processor run.
$T^{(p)}_{l.b,i}(k)$	time for load balancing on Pi for iteration k, for a p processor execution
f_{task}	No. operations in a single application task.

method for generating initial conditions for the simulation of star clusters and results in a non-uniform spatial distribution of the particles.

4.2 A Performance Measure

Load balancing algorithms maintain a balanced workload by transferring work from heavily loaded processors to lightly loaded processors. However, the algorithms themselves incur both communication and computation costs. Below we define a performance measure that takes into account both the costs and benefits of load balancing. In doing this, it is assumed that a good performance model of the application program is available.

The basic performance measure used is the total execution time, T_{total}. Other performance indicators such as processor utilization, message traffic, resource contention, etc. can also be employed based on application and system requirements. For iterative applications (such as the N-body simulations), the total execution time is determined for a fixed number of iterations, max_iter. Each iteration involves execution of the application and the load balancing algorithm. The total execution time on the p processor set $\{P1, P2, \ldots Pp\}$ is given by:

$$T^{(p)}_{total} = \sum_{k=1}^{max\_iter} \max_{all\ i} [\ T^{(p)}_{appl,i}(k) + T^{(p)}_{l.b,i}(k)\ ] \tag{14}$$

The max term above results from processor synchronization at the end of each iteration. $T^{(p)}_{appl}$ is a function of the computation and communication requirements of the application and the task distribution at iteration k. $T_{l.b}(k)$ is a function of the load balancing algorithm complexity and its computation and communication costs (e.g., a function of number of tasks transferred).

The application is divided into a number of identical tasks, and the tasks are distributed over the available processors. Processor Pi is allocated $w_i(k)$ tasks in iteration k and if f_{task} is the compute and communication load associated with a single application

task expressed in terms of number of operations[2], processor Pi takes $w_i(k) \times f_{task}/C_i$ time to execute all tasks for iteration k. Incorporating this into equation 14, we have:

$$T_{total}^{(p)} = \sum_{k=1}^{max\_iter} \max_i [w_i(k) \times f_{task}/C_i + T_{l.b,i}^{(p)}(k)] \tag{15}$$

An ideal load balancing algorithm maintains a "perfectly balanced" task distribution (where the number of tasks on any processor is proportional to its computing capability, given by $\bar{w}(k)$) and incurs zero cost. Since $\bar{w}_i(k)/C_i = \bar{w}_j(k)/C_j$, the total execution time for ideal load balancing (with no overhead costs) can be written as:

$$T_{total,ideal}^{(p)} = \sum_{k=1}^{max\_iter} \bar{w}_1(k) \times f_{task}/C_1 \tag{16}$$

Often, in parallel implementations, the speedup obtained over a single processor is a useful performance measure. The speedup is defined in the standard manner as:

$$\text{Speedup}(p) = \frac{T_{total}^{(1)} \text{ on processor } P1}{T_{total}^{(p)} \text{ on processors } \{P1, P2, \ldots Pp\}} \tag{17}$$

Ideal speedup is calculated using Equation 17 with $T_{total}^{(p)}$ set equal to $T_{total,ideal}^{(p)}$ from Equation 16. The maximum speedup that can be obtained on a $p-$processor execution, however, is limited by the ratio of the combined computing ability of the p processors to that of processor $P1$. The maximum speedup on processors $\{P1, P2, \ldots Pp\}$ is given by:

$$\text{Speedup}_{max}(p) = [\sum_{i=1}^{p} C_i \ ] \ / \ C_1. \tag{18}$$

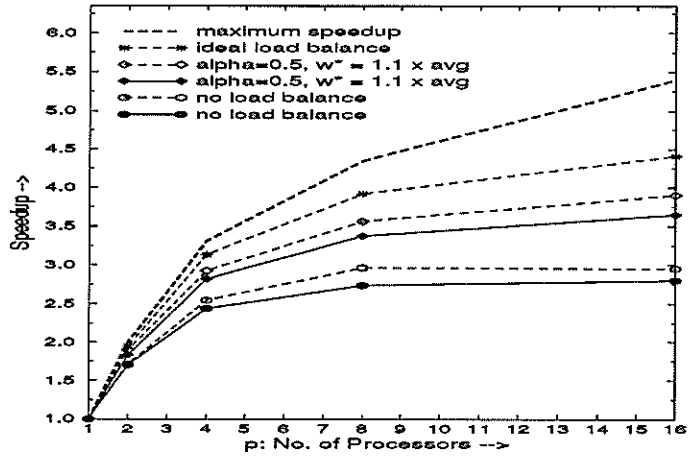
Note that if all processors are of equal computing power, the maximum speedup is given by $\text{speedup}_{max}(p) = p$. In evaluating the load balancing algorithms, a good performance model for the application is necessary to evaluate f_{task} of equation 15, and with this available, the total time and speedup can be established. A mean value performance model for the N-body simulation application on a network of workstations was developed in [9]. Measured and model predicted values were found to be within 10% of each other.

5 Model and Experimental Results

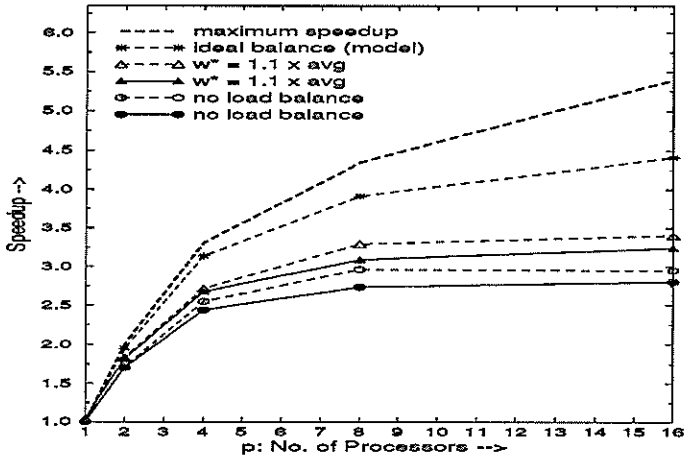
5.1 Model Validation

In this section validation results for the load balancing model (applied to the N-body application) are presented. These studies contrast model derived performance data with experimentally obtained data.

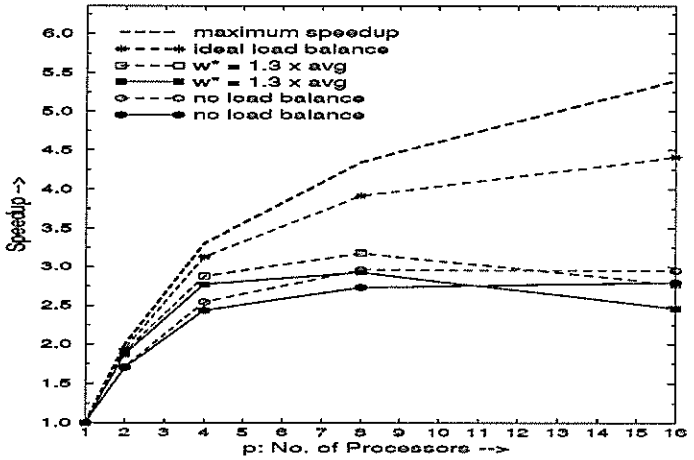
[2] F_{task} includes the number of compute operations in each task and the number of operations "equivalent" to the communication load of the task.



(a) Random Strategy



(b) Diffusion Algorithm



(c) Complete Redistribution

Figure 5: Comparing Model and Measured Speedups
(Solid lines represent measured values and dashed lines represent model values.)

The performance of load balancing algorithms is bounded by the two extreme cases of "no balance" and "ideal balance". The "no balance" case corresponds to application execution without any load balancing. To evaluate the "no balance" performance the model is solved (numerically) with the matrix $M(k)$ in equation 4 set to zero, and (for the experimental data) the application is run without load balancing. In an ideally balanced system, the number of tasks allocated to a processor is proportional to its compute speed. All the processors therefore spend almost equal time for each iteration. The performance model of [9] is used in conjunction with equation 16 to obtain the execution time and speedup associated with the ideal balance case.

For both the experiment and the model, the initial workload distribution is perfectly balanced (*i.e.*, the task distribution, $w(0) = \bar{w}(0)$). Performance data is gathered for the first 100 iterations with each data point in the graphs being the average of several independent runs such that the result falls within a 95% confidence interval.

To validate the model, model and measured values for the total time, $T_{total}^{(p)}$, over different processor populations are obtained and the speedup (equation 17) calculated. Figure 5a shows the comparison for the random load balancing algorithm. No balance, ideal balance and the random load balancing case ($\alpha = 0.5$ and $w^*(k) = 1.1 \times \bar{w}(k)$) are shown. Figure 5b shows the comparison for the diffusion load balancing algorithm. The figure shows speedup curves for no balance, ideal balance and for the diffusion algorithm with $w^* = 1.1 \times \tilde{w}(k)$. Figure 5c shows the comparison for the redistribution algorithm. Again, no balance, ideal balance and redistribution case (with $w^*(k) = 1.3 * \bar{w}(k)$) are shown.

The difference between predicted and measured speedups is well below 20% of the measured value for all three algorithms. This level of error is reasonable considering that the underlying system is a network of time shared workstations subject to non-uniform computational loads and communication delays due to spurious network traffic. The performance model used and the characterization of μ and λ also contribute to the error. Although the comparison shown here is for only a few parameter values, similar results apply for the entire range of parameters considered.

Figure 6 shows the model predicted speedups of the three load balancing algorithms along with the speedups for the ideal and no balance cases. For the no balance case, speedup increases up to about 8 processors and then decreases slightly. This is due to the increased communications costs and poor balance when more processors are present. Load balancing improves the performance with the speedup (with random load balancing) at 16 processors being within 25% of the ideal balance case. Balancing would have a greater positive effect if the underlying interconnection network were faster and had less contention. The speedup curves associated with the random and diffusion load balancing algorithms are similar. The complete redistribution algorithm shows some performance improvement for small processor systems (2 to 4 processors) but the high execution costs causes performance degradation for larger systems. It performs worse than the no balance case. The random strategy is generally better than the diffusion and the complete redistribution algorithms and, at 16 processors, it is about 12% better than diffusion algorithm

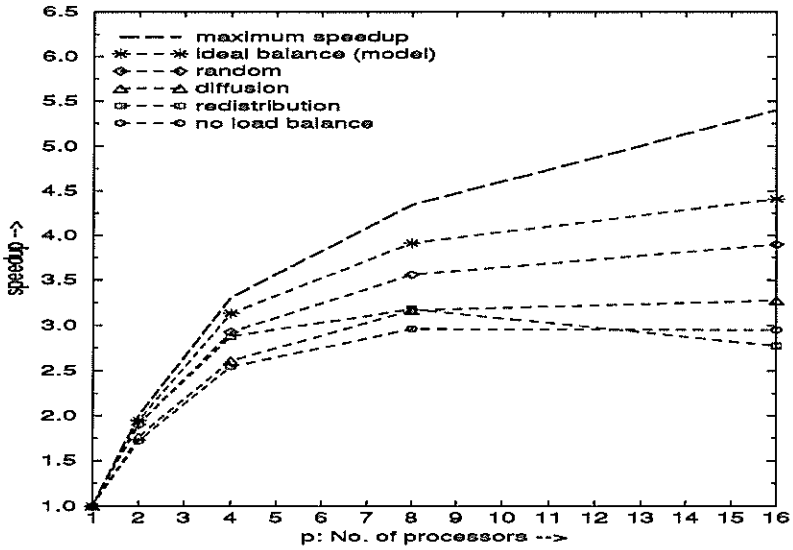


Figure 6: Load balancing algorithm performance

and about 30% better than the complete redistribution algorithm. The performance of the three load balancing algorithms reflects the trade-off between the load balance achieved by the algorithms and the costs incurred in executing them.

The random strategy uses minimum state information (task averages) and involves minimum processing for decision making (random destination selection). Also, the tasks are sent to a single neighbor processor. Therefore, the execution costs for load balancing are very small (typically about 1% of the application execution time per iteration in the N-body simulation example). For the diffusion algorithm, each processor uses the actual task distribution of its neighbors to compute the number of tasks that are sent to each of its neighbors. The cost for acquiring state information, processing cost for task migration decisions and the cost incurred in the actual transfer of these tasks to one or more neighbors (as opposed to a single neighbor processor in the random strategy) are much more for the diffusion algorithm than for the random strategy. Further, the load balance achieved by the diffusion algorithm is only marginally better than the random strategy. Due to this disparity in cost and performance, the random strategy yields a better overall performance. The complete redistribution algorithm brings the system to a near perfect balance every time it is activated. But, the costs incurred by the algorithm are prohibitively high. The overall performance is therefore poorer than the random and the diffusion algorithms, and sometimes poorer than the no load balance case as well. The large communication costs incurred in a networked system like the one used in our experiment may be the major cause for this performance degradation. The relative performance of these algorithms may be different for different applications and computing platforms. Our load balancing and performance models can be parameterized to evaluate performance for any given system or application.

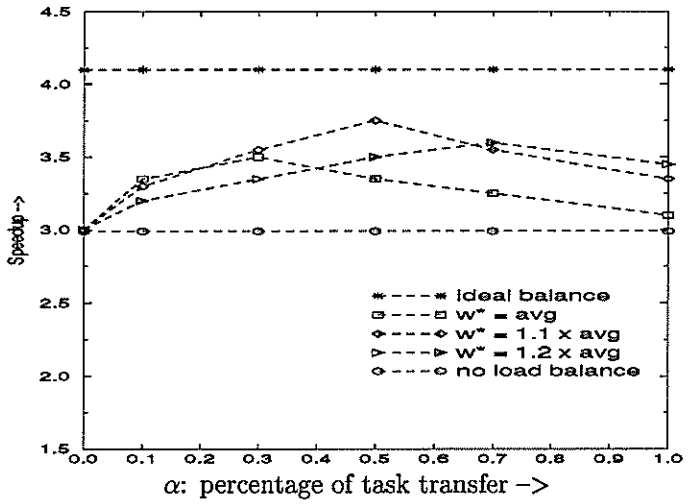


Figure 7: Random Strategy: Effect of α and w^*

5.2 Parameter Selection

Maximizing the parallel performance of a given application requires the selection of the "best" load balancing algorithm and optimum algorithm parameters. Each load balancing algorithm is quantitatively represented using model equation 4. We first identify the control parameters for the algorithm and specify a range for each of these parameters. The model is then numerically solved over the parameter ranges and the corresponding performance evaluated. The set of parameters that yield the best performance for a given system is chosen as the optimal set. The results of this process are presented.

The parameters that control the random algorithm are: α, the percentage of task transfer and w^*, the task threshold at which processors send out tasks. α ranges from 0 to 1. w^*, typically a function of the ideal balance task distribution vector, $\bar{w}(k)$, ranges from $\bar{w}(k)$ to $1.5 \times \bar{w}(k)$. Both α and w^* effect the number of tasks transferred and therefore, the balance achieved and the load balancing costs. Figure 7 shows the effect of these parameter values on the performance of the random algorithm for the eight processor case. Performance for the 16-processor case is similar. The performance for smaller systems (2 and 4 processors) is much less sensitive to parameter changes. For small values of α, fewer tasks are transferred and the workload may remain unbalanced. Larger values of α result in higher task transfer and is more likely to balance the workload, but incurs high load balancing costs, and, since tasks are transferred to a random processor, may result in overloading the recipient processor causing performance degradation. The performance improves with increasing α, reaches a maximum and further increases in α results in poorer performance. For the eight processor case, the best α values are between 0.3 and 0.7.

The w^* parameter also affects the number of tasks transferred and hence the balance and the costs incurred. A higher w^* causes fewer tasks to be transferred. For small systems (not shown in Figure 7), this results in poorer performance (a speedup of 1.8

Table 3: Diffusion Algorithm: Effect of w^*
For 8-processor system

w_i^*	Speedup
$1.0 \times \tilde{w}_i$	2.87
$1.1 \times \tilde{w}_i$	3.08
$1.2 \times \tilde{w}_i$	3.26
$1.3 \times \tilde{w}_i$	3.21
$1.4 \times \tilde{w}_i$	3.11

Table 4: Complete Redistribution: Effect of w^*
For 8-processor system

w^*	Speedup
$1.0 \times \bar{w}$	2.44
$1.1 \times \bar{w}$	2.76
$1.2 \times \bar{w}$	2.95
$1.3 \times \bar{w}$	3.07
$1.4 \times \bar{w}$	2.94
$1.5 \times \bar{w}$	2.83

on 2 processors for $w^* = 1.2 \times \bar{w}$ as opposed to 1.95 for $w^* = \bar{w}$). For larger systems, the performance improves for higher threshold values. For example, the results for an 8 processor system (Figure 7) show that a threshold of $1.1 \times \bar{w}$ performs better than the $w^* = \bar{w}$ case. For 16 processor systems (not shown in the figure), a threshold of $1.3 \times \bar{w}$ yields the best performance. This is because a higher w^* value eliminates system instabilities (e.g., thrashing) and incurs lower load balancing costs as fewer tasks are transferred and these effects are more prominent in large processor systems than in smaller systems.

Similar effects of threshold parameter, w^* can be seen in both diffusion and complete redistribution algorithms. Table 3 shows the speedup obtained with diffusion algorithm for different w^* values for the 8-processor case. $w^*(k) = 1.2 \times \tilde{w}(k)$ yields best performance. The performance of complete redistribution algorithm for different values of w^* is shown in Table 4 (8 processor case). $w^*(k) = 1.3 \times \bar{w}(k)$ yields best performance. The choice of w^* reflects the tradeoff between the load balance achieved and the cost incurred in load balancing. Since diffusion and redistribution algorithms incur higher execution costs than the random algorithm, the optimal thresholds are also higher than for the random algorithm.

Based on the results cited above, a set of optimal control parameters can be derived for the algorithms and the given workload. Table 5 gives the optimal parameters to be used for the N-body simulation example for the random, diffusion and redistribution

Table 5: Optimal parameters: N-body simulation example

n	no balance (speedup)	Random			Diffusion		Redistribution		ideal (speedup)
		α	w^*	speedup	w^*	speedup	w^*	speedup	
2	1.71	0.9	$\bar{w}$	1.76	$\tilde{w}$	1.76	$1.1 \times \bar{w}$	1.90	1.94
4	2.54	0.7	$\bar{w}$	2.92	$1.1 \times \tilde{w}$	2.61	$1.2 \times \bar{w}$	2.87	3.13
8	2.96	0.5	$1.1 \times \bar{w}$	3.75	$1.2 \times \tilde{w}$	3.25	$1.3 \times \bar{w}$	3.07	3.91
16	2.99	0.5	$1.2 \times \bar{w}$	3.95	$1.2 \times \tilde{w}$	3.27	$1.4 \times \bar{w}$	2.75	4.41

load balancing algorithms. The corresponding performance is also given. The "best" performance obtained with load balancing is compared with the no balance and the ideal balance case. The random strategy results in speedups within 30% of the ideal balance case. Redistribution algorithm performance is somewhat poorer but shows considerable improvement over the no balance case.

6 Conclusions and Future Work

A matrix iterative model was developed to represent a range of sender-initiated load balancing schemes. The model was parameterized to represent the random strategy, the diffusion algorithm and the complete redistribution algorithm. An N-body simulation example was used to validate the model for the three load balancing schemes with errors between model and measured values less than 20%.

Based on the model and the performance measures defined, we proposed a methodology for choosing a near optimal load balancing algorithm and associated parameters for a given application and multiprocessor system. This is important since the difference between optimal parameters and say, average parameter selections (e.g., $\alpha = 0.3$, $w^* = \bar{w}$) can be as much as 15%. For the N-body simulation example shown, the random strategy outperforms both the diffusion (12% better) and the redistribution (30% better) algorithms and its performance is within 25% of the ideal balance case.

Performance modeling tools for parallel applications and computing systems are important in the design of high performance computing systems. The work reported in this paper is part of an ongoing research effort towards developing effective performance models that incorporate application requirements, computing platform characteristics, and the impact of performance improvement support techniques like load balancing. The load balancing model presented in this paper is used to evaluate various load balancing schemes and to compare their performance. Such performance modeling and evaluation can be used to guide the the algorithm designer in choosing the best algorithm and associated parameters for a given application and platform.

References

[1] Ishfaq Ahmad, Arif Ghafoor, and Kishan Mehrotra. Performance Prediction of Distributed Load Balancing on Multicomputer Systems. In *Proceedings of Supercomputing '91*, 1991.

[2] Mikhail J. Atallah, Christina Lock Black, Dan C. Marinescu, Howard Jay Siegel, and Thomas L. Casavant. Models and Algorithms for Coscheduling Compute-Intensive Tasks on a Network of Workstations. *Journal of Parallel and Distributed computing*, 16, 1992.

[3] Joshua E. Barnes and Piet Hut. A Hierarchical O(N log N) Force Calculation Algorithm. *Nature*, 324(4), Dec. 1986.

[4] Thomas L. Casavant and Jon G. Kuhl. A taxonomy of scheduling in general purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.

[5] Yuan-Chieh Chow and Walter H. Kohler. Models for dynamic load balancing in a heterogeneous multiple processor system. *IEEE Transactions on Computers*, C-28(5):354–361, May 1979.

[6] George Cybenko. Dynamic Load Balancing for Distributed Memory Multiprocessors. *Journal of Parallel Distributed Computing*, 7(1):279–301, 1989.

[7] Derek L. Eager, Edward D. Lazowska, and John Zoharjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, 12(5), May 1986.

[8] Geoffrey C. Fox et. al. *Solving Problems on Concurrent Processor*. Englewood Cliffs, NJ, 1988.

[9] Mark Franklin and Vasudha Govindan. The N-body Problem: Distributed System Load Balancing and Performance Evaluation. In *Proceedings of the 6th International Conference on Parallel and Distributed Computing Systems*, October 1993. Louisville, Kentucky, USA.

[10] Al Geist, Adan Beguelin, Jack Dongarra, Weicheng Jiang, and Vaidy Sunderam. *PVM 3 User's Guide and Reference Manual.* Oak Ridge National Laboratoty, Oak Ridge, Tennessee 37830, ornl/tm-12187 edition, May 1996.

[11] Reinhard V. Hanxleden and L. Ridgway Scott. Load Balancing on Message Passing Architectures. *Journal of Parallel and Distributed Computing*, 13(3):312–324, 1991.

[12] A. J. Harget and I. D. Johnson. *Distributed Computer Systems; Part 1*, chapter Load Balancing Algorithms in Loosely Coupled Distributed Systems: A Survey. Butterworth and Co, 1990.

[13] Orly Kremien and Jeff Kramer. Methodical Analysis of Adaptive Load Sharing Algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 3(6), November 1992.

[14] Frank C. H. Lin and Robert Keller. The Gradient Model Load Balancing Method. *IEEE Transactions on Software Engineering*, SE-13(1), January 1987.

[15] M. Henon S. J. Aarseth and R. Wielen. Comparison of Numerical Methods for the Study of Star Cluster Dynamics. *Astronomy and Astrophysics*, 37(183), 1974.

[16] Vaidy S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2, december 1990.

[17] Marc H. Willebeek-LeMair and Anthony P. Reeves. Strategies for Load Balancing for Highly Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(9), September 1993.

[18] Ellen Witte, Roger Chamberlian, and Mark Franklin. Parallel simulated annealing using speculative computation. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):483–494, october 1991.

[19] Jian Xu and Kai Hwang. Heuristic Methods for Dynamic Load Balancing in a Message-Passing Multicomputer. *Journal of Parallel and Distributed Computing*, 18:1–13, 1993.