Dynamic Load Balancing in a Network of Workstations

95.515F Research Report

By: Shahzad Malik (219762)

November 29, 2000

Table of Contents

1 Introduction

With the prevalence of powerful workstations and rapid advances in high-speed computer network technologies, the use of a network of workstations (NOW) as a virtual parallel machine (VPM) has become a viable alternative to expensive, dedicated parallel machines. However, certain differences between the two types of machines need to be considered [DAND97a]:

Heterogeneous Nodes:
Nodes in a multiprocessor system typically consist of homogeneous processors, while workstations in a VPM may be different from each other in architecture, operation system, CPU speed, memory size, and available disk space.

Load on Nodes:
In a multiprocessor system, the workload is more predictable and can be controlled by a dedicated scheduler. However, the load on each workstation in a NOW can vary from time to time based on the workload brought about by the workstation owner as well as other users.

Load on Network:
Dedicated parallel machines are usually connected using fixed topologies (eg. Mesh or hypercube) and implement dedicated, high-speed interconnection networks and switching techniques. VPMs, on the other hand, tend to rely on standard LAN technologies that typically exhibit high overhead and low bandwidth compared to the dedicated parallel machines.

Node Failures:
Since owners of workstations have control of that specific network node, a VPM on a NOW must take fault-tolerance and fault-recovery into consideration (for example, if a workstation owner resets the machine).

Thus, when using a NOW for parallel tasks, one has to cope with the dynamic behaviour of the compute nodes, the network load, and the application tasks. These can lead to local load imbalances, which hamper the application's execution speed and the overall system performance [ALBA96].

Considering the serious issues mentioned above, one begins to question the viability of a NOW as an alternative to a dedicated parallel processing machine. However, studies have shown that 80% of workstations in a typical corporate network are idle depending on the time of day [DAND97a]. As a result, it becomes worthwhile to investigate some solutions that help to overcome some of the performance bottlenecks that a NOW may encounter when performing parallel processing. Load balancing is one such solution.

2 Load Balancing

Load balancing is defined as the allocation of the work of a single application to processors at run-time so that the execution time of the application is minimized [SIEG94]. Since the speed at which a NOW-based parallel application can be completed depends on the computation time of the slowest workstation, efficient load balancing can clearly provide major performance benefits. The two major categories for load-balancing algorithms are *static* and *dynamic*.

2.1 Static Load Balancing

Static load balancing algorithms allocate the tasks of a parallel program to workstations based on either the load at the time nodes are allocated to some task, or based on an average load of our workstation cluster. The advantage in this sort of algorithm is the simplicity in terms of both implementation as well as overhead, since there is no need to constantly monitor the workstations for performance statistics. However, static algorithms only work well when there is not much variation in the load on the workstations [LEE95]. Clearly, static load balancing algorithms aren't well suited to a NOW environment, where loads may vary significantly at various times in the day, based on the issues discussed earlier.

2.2 Dynamic Load Balancing

Dynamic load balancing algorithms make changes to the distribution of work among workstations at run-time; they use current or recent load information when making distribution decisions [DAND97b].

As a result, dynamic load balancing algorithms can provide a significant improvement in performance over static algorithms. However, this comes at the additional cost of collecting and maintaining load information, so it is important to keep these overheads within reasonable limits [DAND97b]. The remainder of this report will focus on such dynamic load balancing algorithms.

3 Load Balancing Algorithms

The choice of a load balancing algorithm for a VPM is not always an easy task. Various algorithms have been proposed in the literature, and each of them varies based on some specific application domain. Some load balancing strategies work well for applications with large parallel jobs, while others work well for short, quick jobs. Some strategies are focused towards handling data-heavy tasks, while others are more suited to parallel tasks that are computation heavy.

While many different load balancing algorithms have been proposed, there are four basic steps that nearly all algorithms have in common [ZAKI96]:

1. Monitoring workstation performance (load monitoring)
2. Exchanging this information between workstations (synchronization)
3. Calculating new distributions and making the work movement decision (rebalancing criteria)
4. Actual data movement (job migration)

3.1 Balancing Strategies

There are three major parameters which usually define the strategy a specific load balancing algorithm will employ. These three parameters answer three important questions: i) who makes the load balancing decision, ii) what information is used to make the load balancing decision, and iii) where the load balancing decision is made.

3.1.1 Sender-Initiated vs. Receiver-Initiated Strategies

The question of who makes the load balancing decision is answered based on whether a *sender-initiated* or *receiver-initiated* policy is employed. In *sender-initiated* policies, congested nodes attempt to move work to lightly-loaded nodes. In *receiver-initiated* policies, lightly-loaded nodes look for heavily-loaded nodes from which work may be received [DAND97b].

Figure 1 shows the relative performance of a *sender-initiated* and *receiver-initiated* load balancing

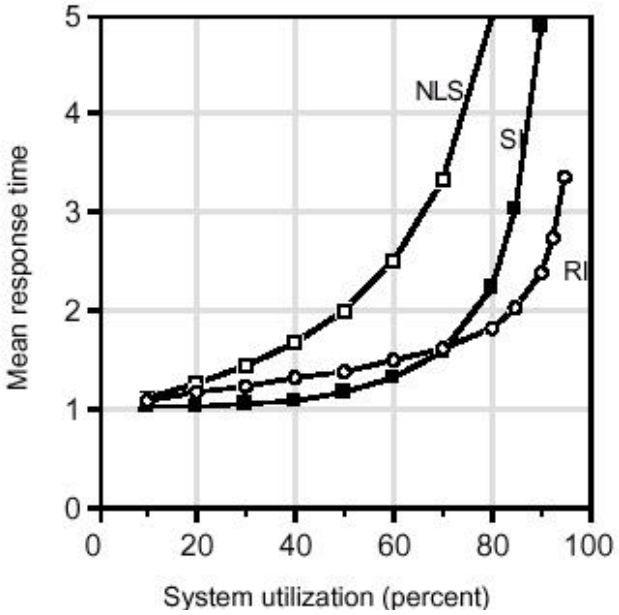

Figure 1 – Performance of Sender-Initiated vs. Receiver-Initiated Strategies [DAND97b]

algorithm on a homogeneous 32-node system. As can be seen, both the *sender-initiated* and *receiver-initiated* policies perform substantially better than a system which has no load sharing. What we also see is the *sender-initiated* policy performing better than the *receiver-initiated* policy at low to moderate system loads. [DAND97b] reasons that at these loads, the probability of finding a lightly-loaded node is higher than that of finding a heavily-loaded node. Similarly, at high system loads, the *receiver-initiated* policy performs better since it is much easier to find a heavily-loaded node. As a result, *adaptive* policies have been proposed which behave like *sender-initiated* policies at low to moderate system loads, while at high system loads they behave like *receiver-initiated* policies.

3.1.2 Global vs. Local Strategies

Global or *local* policies answer the question of what information will be used to make a load balancing decision [ZAKI96]. In *global* policies, the load balancer uses the performance profiles of all available workstations. In *local* policies workstations are partitioned into different groups. In a heterogeneous NOW, the partitioning is usually done such that each group has nearly equal aggregate computational power [ZAKI96]. The benefit in a local scheme is that performance profile information is only exchanged within the group.

The choice of a global or local policy depends on the behaviour an application will exhibit. For global schemes, balanced load convergence is faster compared to a local scheme since all workstations are considered at the same time. However, this requires additional communication and synchronization between the various workstations; the local schemes minimize this extra overhead. But the reduced synchronization between workstations is also a downfall of the local schemes if the various groups exhibit major differences in performance. [ZAKI96] notes that if one group has processors with poor performance (high load), and another group has very fast processors (little or no load), the latter will finish quite early while the former group is overloaded.

3.1.3 Centralized vs. Distributed Strategies

A load balancer is categorized as either *centralized* or *distributed*, both of which define where load balancing decisions are made. In a centralized scheme, the load balancer is located on one master workstation node and all decisions are made there. In a distributed scheme, the load balancer is replicated on all workstations.

Once again, there are tradeoffs associated with choosing one location scheme over the other. For centralized schemes, the reliance on one central point of balancing control could limit future scalability. Additionally, the central scheme also requires an “all-to-one” exchange of profile information from workstations to the balancer as well as a “one-to-all” exchange of distribution instructions from the balancer to the workstations. The distributed scheme helps solve the

scalability problems, but at the expense of an "all-to-all" broadcast of profile information between workstations. However, the distributed scheme avoids the "one-to-all" distribution exchange since the distribution decisions are made on each workstation.

Figure 2 shows the performance of four different matrix multiplication algorithms on 4 and 16 processors respectively. All algorithms were receiver-initiated while the other two parameters were adjusted, resulting in the four different algorithms consisting of a Global Centralized Algorithm (GCDLB), Global Distributed Algorithm (GDDLB), Local Centralized Algorithm (LCDLB), and a Local Distributed Algorithm (LDDLB). For the 4 processor situation, the global schemes perform significantly better than the local schemes, as would be expected due to the faster convergence. However, with 16 processors, the differences are not as pronounced since we are seeing the benefits of the reduced communication costs associated with the local schemes.

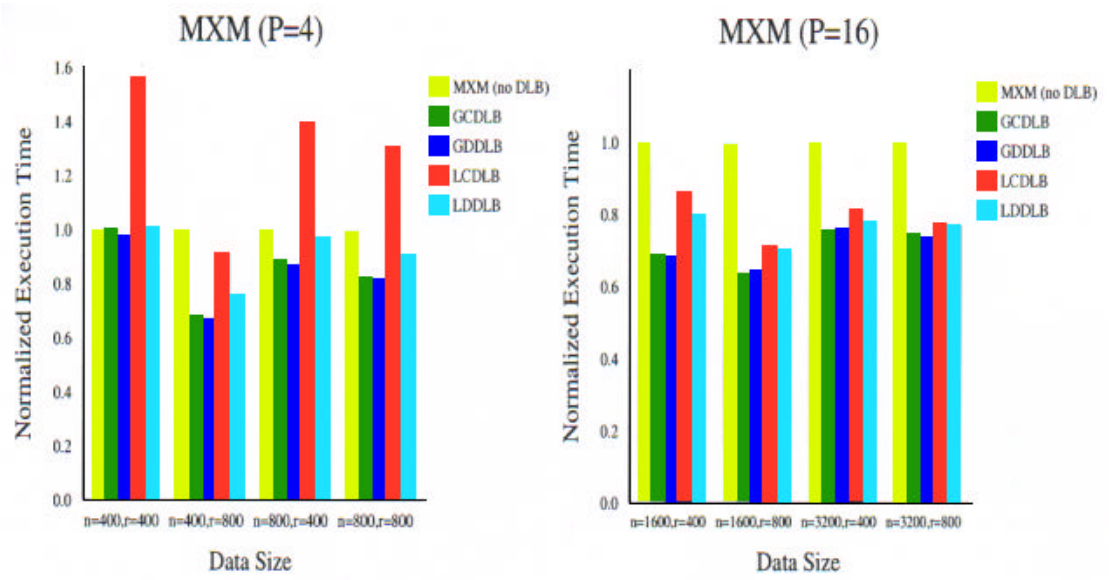

Figure 2 – Matrix Multiplication on 4 and 16 processors [ZAKI96]

3.2 Load Monitoring

In order to make optimal balancing and work distribution decisions, a load balancer needs to take some or all of the following information into consideration [ALBA96]:

- available capacity on each node (CPU, memory, disk space)
- current load of each node
- required capacity for each task
- network connectivity and capacity
- communication pattern for each task (if applicable)

[DAND97b] notes that the choice of load index to use when measuring the performance of a workstation has a considerable effect on the performance of the load balancer as a whole.

As with choosing a load balancing strategy, the choice of a load index largely depends on an application's requirements. A commonly used load index in many algorithms is the difference between the time a workstation starts a particular task and ends that same task (ie. computation time). This effectively allows a load balancer to compare the time each processor is spending on various tasks. Thus the idea behind the use of this load index is to have all workstations complete their tasks at roughly the same time. If this can be achieved, the VPM will be balanced in its tasks. However, this particular load index doesn't take network communication time into consideration. For equal sized tasks this isn't a major concern, but for loop scheduling algorithms that may adjust the granularity of the tasks that get allocated to the various workstations, this becomes an important consideration.

Thus, in order to take communication time into consideration, some algorithms prefer to use a response time for each workstation and compare it to an average response time for all workstations. In other words, let $Ttotal_i$ represent the total time it takes for a slave machine *i* to receive its task, perform the computation, and return the results. This value is then used to compute an average $Ttotal_{avg}$ across all workstations. Thus if a particular workstation has a $Ttotal_i$ value below the average $Ttotal_{avg}$, then this information can be used to possibly reduce the amount of work that is sent to workstation *i*. Similarly, if workstation *i* has a $Ttotal_i$ value above the $Ttotal_{avg}$, then work should potentially be reduced for this particular slave. The downfall to this approach is that it assumes network load is relatively constant, which is not true. Networks tend to experience bursts of activity, which could affect the timings taken by the balancer [DAND97b].

3.3 Rebalancing Criteria

There are two major issues that need to be considered before a decision can be made regarding attempting to balance the load:

1. Moving a task to a workstation will increase its load index. To avoid the oscillation of movement, a load balancer must be sure that the movement of a task from one workstation to another does not make the other workstation have a larger load index than that of the workstation on which the task presently resides.
2. The measure of success of dynamic load balancing is the net reduction of execution time achieved by applying the load balancing algorithm. Thus, the load balancer needs to make sure that the potential performance gain associated with moving a task from one workstation to another is more than the cost of performing an actual job migration.

3.4 Checkpointing and Job Migration

When the owner of a workstation claims control of his/her node in the network, migrating the job to another workstation is desirable to simply restarting it on another workstation [DAND97b]. This implies that the state of a task should be captured, after which it is started on a target machine and initialized with the captured state. Correct migration is difficult since the interactions of the task with its environment need to be taken into account [ALBA96]. Issues such as dealing with open files, handling communications with other tasks, and checkpointing overhead all need to be taken into consideration by a job migration system. From the perspective of a load balancer, migration is an important issue since the balancer needs to consider the overhead associated with job migration in order to measure the potential cost when making a balancing decision (see Section 3.3 regarding Rebalancing Criteria).

The full details of checkpointing and job migration are an entire topic by themselves, so they will not be discussed here. However, it is sufficient to note that a load balancer can make use of a job migration and checkpointing facility as long as *migration costs* can be computed for a specific algorithm. It is also worthwhile to mention that a migration system is not a requirement in a load balancer, since some studies have shown that migration doesn't yield any significant performance benefits due to the extra checkpointing and state-capturing overhead [DAND97b].

4 An Example Load Balancing Algorithm

Most load balancing algorithms are designed based on the performance requirements of some specific application domain. For example, applications that exhibit lengthy parallel jobs usually benefit in the presence of a job migration system. However, applications with shorter tasks usually don't warrant the expense of job migration and thus are better handled with clever loop scheduling algorithms where the task granularity changes dynamically, as defined in [DAND97a]. As a result, only one algorithm will be described here in order to provide an overview of the various issues that a typical algorithm must take into consideration. However, all algorithms closely follow the four basic load-balancing steps outlined at the beginning of Section 3.

4.1 Single Program Multiple Data Computation Model

The Single Program Multiple Data (SPMD) paradigm implies that all the workstations run the same code, but operate on different sets of data. The motivation for using SPMD programs is that they can be designed and implemented easily, and they can be applied to a wide range of applications such as numerical optimization problems and solving coupled partial differential equations [LEE95].

The SPMD computation model is depicted in Figure 3. Each task is divided into operations or iterations. Workstations execute the same operation asynchronously, using data available in the workstation's own local memory. This is followed by a data exchange phase where information can be exchanged between workstations (if required), after which all workstations wait for synchronization. Thus each "lock-step" of an SPMD program contains 3 phases:

1. Calculation Phase: each task will do the required computation. There is no communication between workstations at this point.
2. Data Distribution Phase: each task will distribute the relevant data to other tasks that need it for the next lock-step.
3. Synchronization Phase: this phase ensures that all tasks have completed the same lock-step. Otherwise there will be problems with tasks using the wrong data.

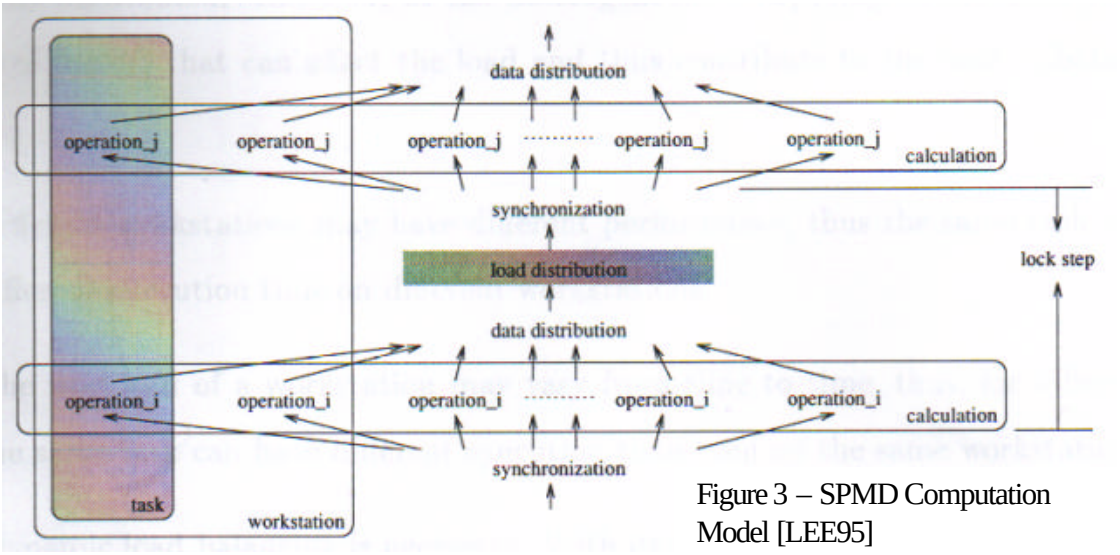

Figure 3 – SPMD Computation Model [LEE95]

If an SPMD program were to be executed on a homogeneous multiprocessor system, the workload would be balanced for the entire computation (assuming that all tasks were initially evenly distributed). However, in a NOW, there are various other factors that can affect the load and thus contribute to load imbalances.

Thus, within the SPMD paradigm in a VPM, we would like to reduce the execution time of the program by dynamically shifting/migrating tasks from one workstation to another at the end of each lock-step, if required. There are 2 things that need to be considered:

1. determine if there is a need to rebalance the load
2. find the best distribution of tasks

4.2 SPMD Load Balancer

[LEE95] has developed a global, centralized, sender-initiated load balancing algorithm for large, computation-heavy SPMD programs using the following parameters:

Tcompute$_i$ – the interval between the time at which the first task on workstation *i* starts execution and the time at which the last task on the same workstation completes the computation and waits for the synchronization. This value is thus the dynamic workload index for the algorithm, since there is a direct relation between *Tcompute$_i$* and a workstation's load.

Assuming a program can be decomposed into N tasks and there are P workstations, then we have $N = ? n_i$ (for i = 1 to P). Thus n_i is the number of tasks on workstation *i*.

Ttask$_i$ – the average computation time for each task on a workstation, defined as:

$$Ttask_i = Tcompute_i / n_i \quad \text{(equation 1)}$$

Thigh – the maximum of *Tcompute* over all workstations, defined as:

$$Thigh = \max \{ Tcompute_i \} \; (1 <= i <= P)$$

Tlow – the minimum of *Tcompute* over all workstations, defined as:

$$Tlow = \min \{ Tcompute_i \} \; (1 <= i <= P)$$

A common approach taken for dynamic load balancing on a NOW is to predict future performance based on past information [ZAKI96]. In the SPMD algorithm, *Ttask* can be used to update *Tcompute*. Thus if *m* tasks are moved to workstation *i*, we can solve equation1 to give us an estimation of *Tcompute$_i$*:

$$Ttask_i \times (n_i + m)$$

The estimation is based on the current workload of the workstation, and it is valid because all tasks in an SPMD program are executing the same code. *Tcompute* will be recalculated after each task reallocation, with *Thigh* and *Tlow* updated accordingly.

4.3 Meeting the Rebalancing Criteria

Therefore, in order to balance the load, tasks from workstations that have a longer *Tcompute* will be moved to the workstations with a shorter *Tcompute*. However, the algorithm must also take into account the rebalancing criteria as discussed in Section 3.3.

For the first rebalancing criteria, assume that we have a workstation *k* that has the current highest *Tcompute*. Further assume that *L* represents the number of lock steps remaining, and m_i represents the number of tasks workstation *i* transmits or receives. Therefore, in order to guarantee that moving a task from workstation *k* to a new workstation doesn't cause the new workstation's *Tcompute* to be greater than *k*'s, we check the following condition:

$$\mathbf{Ttask_k \times n_k > \min \{Ttask_i \times (n_i + 1)\} \; (1 <= i <= P, \; i != k)}$$

If this is true, then moving one task from workstation *k* to another workstation will not cause the oscillating effect that was mentioned in Section 3.3.

For the second rebalancing criteria (where we need to verify that attempting to balance the load provides some performance gain), we can compute the following:

OldThigh – represents the previous *Thigh* value before the current load balancing decision

NewThigh – represents the new *Thigh* value that is computed after meeting criteria 1 (which is guaranteed to be lower than *OldThigh*)

Therefore the gain associated with performing load balancing would be:

$$Gain = (OldThigh - NewThigh) \times L$$

Assuming that our load balancer knows *Toverhead*, the cost of performing job migration of the SPMD tasks, we can now check criteria 2 using the following:

$$Gain >= Toverhead \times \max\{m_i\} \quad (1 <= i <= P)$$

If this is true, then we can conclude that it is worthwhile to perform the load balancing and job migration.

4.4 Iterative Algorithm

The final iterative algorithm, as outlined by [LEE95], can be summarized as follows:

$OldThigh$ = max $\{Tcompute_i\}$ $(1 <= i <= P)$
WHILE criteria 1 and 2 are **TRUE DO**
    FOR i = 1 to P
        $Tcompute' = (n_i + 1) \times Ttask_i$
    ENDFOR
    move a task from workstation with *Thigh* to the one with the smallest *Tcompute'*
    Update *Tcompute* of the workstations involved in task migration
    $NewThigh$ = max $\{Tcompute_i\}$ $(1 <= i <= P)$
ENDWHILE

Each iteration of the while loop attempts to move one task from the most heavily loaded workstation to the most lightly loaded node, as long as the rebalancing criteria are being met. After some movement, the load monitoring variables are recomputed and the while loop repeats. This continues until the algorithm detects that there are no more tasks that can be redistributed without degrading performance. In other words, the while loop iterates until the system is as balanced as possible given the current timing information.

5 Cluster Management Software and Load Balancing

Cluster Management Software (CMS) is typically installed onto clusters of workstations in order to manage and schedule applications that run on these systems. They provide an increased and reliable throughput of user applications on the systems they manage by performing load balancing, utilizing spare CPU cycles, providing fault tolerance features, etc [BAKE96].

Typically batch jobs are given to the CMS via text files that specify such things as the job name, maximum runtime, and desired platform [BAKE96]. A master scheduling system, which has an overall view of the NOW, is then given this job information. This is where a load balancing system comes into play.

5.1 LoadLeveler

LoadLeveler is an example of a CMS, which distributes jobs to a cluster of workstations or to nodes of a multiprocessor machine [BAKE96]. It's load balancer is implemented using a global, centralized strategy whereby a *Central Master* is responsible for assigning and scheduling jobs to nodes as well as continuously maintaining state information [DAND97b]. Thus when a job is first submitted to LoadLeveler, its requirements (such as memory, disk space, operating system, etc) are compared to all the workstation statistics currently available to LoadLeveler. Once a suitable node or nodes are located, the job is dispatched. Additionally, LoadLeveler supports checkpointing and job migration to facilitate the load balancer as well as allow for fault tolerance. Finally, LoadLeveler supports a scalable architecture, since as workstations are added to the cluster these additional resources are automatically added to the pool of available workstations (transparently to the user).

5.2 Condor

Condor is a public domain CMS which was developed as a research project at the University of Wisconsin. It uses a combination centralized and distributed load balancing scheme, whereby each workstation is responsible for scheduling its own jobs while the actual load distribution is performed at a centralized controller [DAND97b]. Thus the central controller periodically polls each workstation and collects state information consisting of a node's current load as well as the number of jobs in the node's queue. This information is used to compute a workload index at the central controller, which is then passed along to each workstation in order to perform its own local job scheduling. Additionally, Condor supports checkpointing, whereby a disk image of an active process is stored on the workstation in order to facilitate both fault tolerance and process migration [COND00].

6 Conclusion

Load balancing is an important issue in a virtual parallel machine built using a low-cost network of workstations. The most difficult aspect of load balancing in a network of workstations involves deciding on which algorithm to use. Hundreds of various algorithms have been proposed, and each one has its own specific motivations and design decisions that result in trade-offs that aren't always suited to every imaginable task.

This report described many of the design issues that are commonly considered when deciding on a load balancing algorithm (such as global/local, centralized/distributed, etc.), as well as the tradeoffs associated with the various parameters and strategies. Additionally, this report outlined the details of an algorithm targeted towards SPMD style programs in order to present a concrete example of the details associated with an actual load balancing implementation. Finally, the load balancing features for two cluster management software packages (LoadLeveler and Condor) were described briefly.

References

[ALBA96] Van Albada, G.D., Clinckemaillie, J. *Dynamite-blasting Obstacles to Parallel Cluster Computing*, Technical Report, Department of Computer Science, University of Amsterdam, The Netherlands, 1996.

[BAKE96] Baker, M., Fox, G., Yau, H. "Review of Cluster Management Software", *NHSC Review*, 1996 Volume, First Issue, July 1996.

[COND00] Condor Homepage (http://www.cs.wisc.edu/condor)

[DAND97a] Dandamudi, S., Piotrowski, A. "A Comparative Study of Load Sharing on Networks of Workstations", *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, New Orleans, October 1997.

[DAND97b] Dandamudi, S. *Sensitivity Evaluation of Dynamic Load Sharing in Distributed Systems*, Technical Report TR 97-12, Carleton University, Ottawa, Canada.

[LEE95] Lee, B. *Dynamic Load Balancing in a Message Passing Virtual Parallel Machine*. Technical Report, Division of Computer Engineering, School of Applied Science, Nanyang Technological University, Singapore, 1995.

[OVER96] Overeinder, B.J., Sloot, P.M.A. "A Dynamic Load Balancing System for Parallel Cluster Computing", *Future Generation Computer Systems*, 12, pp. 101-105, May 1996.

[SIEG94] Siegell, B., Steenkiste, P. "Automatic Generation of Parallel Programs with Dynamic Load Balancing". *IEEE Symposium on High Performance Distributed Computing*, August 1994.

[ZAKI96] Zaki, M., Li, W., Parthasarathy, S. "Customized Dynamic Load Balancing for a Network of Workstations". *Proceedings of HPDC '96*, 1996.