Cost-Efficient Location of Resources Using Multicasting for Load Distribution

J. Tan*

School of Computer Science & Software Engineering, Monash University
900 Dandenong Road, Caulfield East, Vic. 3145, Australia
Phone: +613 9903-1405, Fax: +613 9903-1077
E-mail: jtan@broncho.ct.monash.edu.au

*The work reported in this paper has been funded in part by the Co-operative Research Centre Programme through the Australian Government's Department of Industry, Science and Resources.

Abstract

In a distributed system, load balancing or load sharing is a desirable feature that allows a heavily-loaded processor to migrate excess tasks to less busy or idle processors. This paper suggests a technique for locating the appropriate processor for migrated tasks without excessive overhead. In a heterogeneous environment, this location problem is crucial since processors vary and their capability to satisfy a task's requirements may change from time to time. But locating an appropriate processor generally requires considerable communication overhead while state information is gathered about these processors. This paper presents a mechanism called loaded multicasting to locate an appropriate processor to host and execute a migrating task. Multicasting is the key since it provides selectivity while minimizing communication costs. Modeling and simulation experiments are being attempted and prototyping should subsequently follow, but results are still not available. Despite the absence of such results, this paper discusses the approach through examples of load sharing and load balancing algorithms augmented with loaded multicasting.

Keywords: *distributed scheduling, distributed systems, load balancing, load distribution, load sharing, multicasting, resource management.*

1 Introduction

A *distributed system* is a loosely-coupled system of processors that cooperate with one another without the benefit of shared memory, while a *parallel system* is tightly-coupled, uses shared memory, and normally uses a homogeneous set of processors. Both systems harness the power of parallel and distributed processing, but there are advantages in using distributed systems from a heterogeneous network of computers rather than parallel systems using homogeneous processors:

- A distributed system normally costs less to set up [5, 15] than parallel systems which usually require customized components;
- The heterogeneity of the participating processors can provide a variety of capabilities and resources that applications may exploit;
- Code from preexisting non-distributed applications can be reused for distributed applications; and
- Resources used for local applications may be shared with distributed applications [12, 7].

Successful distributed system projects on top of heterogeneous networks include the Nectar project in Carnegie Mellon University [12] and the Beowulf project of the NASA Goddard Space Flight Center [10], among others. Such distributed systems are sometimes referred to as *multicomputers* [12, 15], and can be used for a variety of computing applications.

A significant research area in distributed systems involves how to distribute processing workload across the system – normally referred to as either load sharing or load balancing. *Load sharing* ensures that busy processors can migrate load to idle processors, while *load balancing* ensures a global workload balance among the processors. This cooperative mechanism of distributing workload is sometimes referred to as *load distribution* [6]. *Workload*, or simply, *load*, refers to programs to processing activities, typically referred to as units of *jobs, processes* or, as will be used predominantly in this paper, *tasks.*

Several load distribution solutions assume that tasks may execute on any of the processors in the distributed system [9, 8, 2, 4, 7]. But is not necessarily the case for heterogeneous distributed systems. A major issue in such systems, addressed by a *location policy* [6, 11], is choosing the appropriate processor to delegate workload to. Making this choice should consider three things: the performance index of the distributed system as a whole, the performance of the task, and the overhead incurred.

There is sufficient reason to be concerned about performance issues regarding metrics such as throughput, mean turnaround time, mean waiting time and mean response time, among others. There should also be concern for overhead, such as communication or computational overhead, since overhead can cause performance degradation. However, the performance concerns should not be limited to the performance metrics mentioned. An issue that bears further consideration is in providing tasks with the *resources* they require to execute properly. Such resources may be in the form of libraries, physical devices, communication links, interpreters, compilers, computation or information services, a specific type of CPU, and so on. Resources may be defined as items that tasks must use in order to execute and meet their goals. In a heterogeneous distributed system, these resources may be available on one processor but not on others. So locating these resources is itself an issue. One of the ideals in distributed systems is in making the location of resources transparent. This can be done through resource managers and various other schemes such as distributed name servers [1]. A desired result of this would be the ability to generate tasks on any processor regardless of where the task may eventually be migrated to for various reasons, such as to provide it with access to required resources.

With all these considerations, the location policy of concern in this paper suggests that, in migrating a task for the sake of load sharing or load balancing, an appropriate processor must be found such that:

- this processor can execute the task within a minimal amount of waiting and execution time;
- the processor can satisfy the resource requirements of the task;
- minimal overhead is incurred; and
- there is a reasonable degree of transparency.

2 The Load Distribution Model

The distributed system D consists of x nodes, y tasks and z resource types, where x is a reasonable number of nodes, y is an arbitrarily large number, and z is a reasonable number of resource types. Communication is provided by some reasonably fast and reliable local area network C, e.g., a bus running at 100 Mbps, that supports *IP multicasting*, although there may be other multicast-capable network protocols that can be applied. TCP/IP, the Transmission Control Protocol over the Internet Protocol, is a widely-used network protocol, and multicasting is supported at the IP level. Multicasting allows one message be sent to a group of nodes with only one transmission. This is facilitated by associating a unique *multicast address* to each multicast group, and having its members resolve that multicast address to themselves. A *multicast message* intended for a particular group must then be sent with that group's unique address as its destination address. A node may be a member of several groups at the same time, and may join or leave groups any time. It is assumed that routers do not exist between the nodes, as this will complicate multicasting by requiring special facilities such as multicast routers [3]. This network constraint is possible in a multicomputer environment such as Beowulf [10] or Nectar [12]. This constraint is also important since the absence of multicast routers will isolate the multicast traffic within this network, although there are other ways of minimizing the effects of multicast across segments, although such considerations are not covered in this paper.

2.1 Nodes

The set $N = \{n_1, n_2, ..., n_x\}$ is the set of x nodes, identified by unique node addresses, and may each contain multiple processors or computers, but will be considered as single entities in this model. The value of x, is not expected to reach hundreds. For example, Nectar had only 26 nodes by 1996 [12], while the Beowulf of NASA and Caltech had 16 nodes each [10]. As a formalism, each node n_i maintains an address set a_i for all $i = 1..x$, where a_i contains the addresses that n_i answers to. a_i therefore contains at least one element: n_i's unique node address. In addition to that, however, n_i may also resolve additional addresses: multicast addresses, i.e., of multicast groups that n_i belongs to. It is understood that at some point, having too many elements in a_i will degrade the performance of n_i as it must resolve too many addresses to itself. The activities of a node may be simplified into two: generating as well as executing *tasks*.

2.2 Tasks

The set $T = \{t_1, t_2, ..., t_y\}$ is the set of all tasks in D. The task is the basic unit of execution, and its sole purpose is to utilize resources and be executed by a node for an unknown period of time. Each task is a distinct entity. A task is generated by nodes in an unpredictable rate, and is executed by nodes in an unpredictable amount of time. There is a desirable twofold transparency for tasks as to which node generates them and which node executes them. A task t may be generated by some node n_h, even if it cannot execute t, and must subsequently migrate it elsewhere.

2.3 Resources of D

A *resource* is a reusable and relatively stable hardware or software component that is requested, used and released by tasks during their execution [6]. Each distinct resource is managed by a node that provides resource management services. Identical resources are considered as *resource instances* of a distinct *resource type*. The concept of resource types enhances the distributed system since it becomes possible to disregard the actual instance and location of resources to allocate to tasks. To simplify, the term "resource" will be synonymous with the "resource instance", while resource types will be referred to explicitly.

The set $R(D) = \{r_1, r_2, ..., r_z\}$ is the set of resource types in D. While several instances of a resource type may exist, $R(D)$ counts only one resource type to represent those instances. $R(D)$ must be well known in D, meaning that all nodes must be aware of the same, up-to-date contents of $R(D)$. $R(D)$ is seldom updated, and only for two reasons:

- To add a new resource type r_f, i.e., when a resource instance of resource type r_f is becomes available for the first time; or
- To remove an existing resource type r_f, i.e., when the only instance of resource type r_f becomes removed permanently. Note, however, that *temporary* unavailability of all instances of resource type r_f should not cause its removal from $R(D)$.

Methods of updating $R(D)$ and ensuring that it is well known are not dealt with in this paper.

2.4 Resources of a Node

$R(n_i)$ is the set of resources locally available to a node n_i at some point in time. Unlike $R(D)$, a resource may be added to or removed from $R(n_i)$ for temporary periods. Thus, $R(n_i)$ dynamically changes from time to time, for the following reasons:

- To add a resource r_{fg}, i.e., r_{fg} is made locally available to n_i for the first time or r_{fg} was temporarily unavailable but is now locally available once more; or
- To remove a resource r_{fg}, i.e., r_{fg} has become temporarily or permanently unavailable locally, such that it is not known when it will be locally available once more, e.g., device failure, r_{fg} has become physically and permanently removed from n_i, etc.

2.5 Resource Requirements of a Task

A task is expected to use a number of resources in its execution. But it is not known which ones *will* be used at any given runtime. But to facilitate the location of these resources, the worst case is assumed. $R(t)$ is the set of *resource requirements* of task t, and consists of all resource types that *may* be used by t, which makes $R(t)$ constant for t. Some preprocessing is necessary to identify the consistency of $R(t)$. For the sake of location transparency of resources, $R(t)$ consists of resource types, not resource instances.

2.6 Safety Rules on Resources

There are two safety rules regarding resources of D: (1) $R(D)$ must contain all resource types that represent all resource instances in D, i.e., $R(n) \subseteq R(D)$, for all nodes $n \in N$, and (2) the resource types required by a task must exist in the distributed system, i.e., $R(t) \subseteq R(D)$, for all tasks $t \in T$.

2.7 Heterogeneity by Resources

Nodes in D vary in what resource types are locally available to them, i.e., there may exist nodes n_i, $n_j \in N$ such that $R(n_i) \neq R(n_j)$, although it is possible for them to have similar resources, i.e., $R(n_i) \cap R(n_j) \neq \emptyset$. Heterogeneity of nodes in this model is therefore in the context of resource availability among nodes in D.

2.8 Qualified Nodes

A migrating task t may exist such that n_i may execute t but n_j cannot. This results from the fact that $R(n_i) \cap R(t) = R(t)$ while $R(n_j) \cap R(t) \neq R(t)$, i.e., one or more resources needed by t are locally available to n_i but not to n_j. Node n_i is therefore *qualified* to execute t at that point in time and is a candidate to host t. On the other hand, n_j is *disqualified* from executing t, or *unqualified* to execute t, at that given point in time, and t should not be migrated to n_j. Time is relevant since $R(n_i)$ and $R(n_j)$ may change from time to time.

Another aspect of disqualification is if the node's processor or CPU is unavailable, such as if its workload is already heavy. Due to its heavy workload of running tasks with unpredictable run-times, thus being unable to guarantee when the processor will be available once more, this "resource" must be removed from $R(n_j)$ for the time being, making the node disqualified from hosting tasks that require this "resource", i.e., any task. But in implementation, this wastes the use of a resource type to represent the node's availability. It would be simpler for a node to simply make all its resources unavailable, i.e., $R(n_j = \emptyset$, if it cannot host additional migrated tasks.

2.9 The Need to Migrate

Given a task t that has been generated in node n_h, these conditions will require the migration of t to another node:

- n_h is not qualified to execute t, i.e., $R(n_h) \cap R(t) \neq R(t)$, so it is necessary to migrate t to a qualified node for execution; or
- n_h is qualified to execute t, but it is foreseen that system performance will improve if t is migrated to another node, for reasons such as:
  - Some node has become idle and is thus underutilized;
  - n_h is overloaded, thus giving it a poor performance index; or
  - For the sake of global workload balance, the load of n_h must be reduced.

2.10 Migration

The model assumes that once a task has begun its execution, then it may not subsequently migrate to another node. Multiple migration due to on-demand resource requests may considerably complicate task migration and load distribution, as well as introduce considerable overhead due to the latency associated with locating a qualified and available host as well as actual migration of tasks through the network medium. For this reason, qualified nodes are identified on the basis of their ability to satisfy *all* the resource requirements of the migrating task.

2.11 The Goal: Resource Matching

The goal of the location policy described in this model can be defined thus: in the event that a task t is to be migrated to another node, there must be found a *qualified node* q, to accept and execute t. But since several nodes may be qualified to execute t at the same time, the goal may generalize into a search for a non-empty set Q, where each node $q \in Q$ is qualified to execute t, i.e., $R(q) \cap R(t) = R(t)$ for all $q \in Q$. The process of meeting the goal presented may be referred to as *resource matching* [13].

3 Loaded Multicasting

Multicasting minimizes communication overhead by allowing multicast messages to reach all members of a particular multicast group with only one transmission. Of course, it ma be that the message must subsequently be divided into several transmission packets at the lower levels of the network protocol stack, although, there is still no redundancy in transmission. *Loaded multicasting* is a modification of multicasting that: (1) loads state information on the multicast address or addresses of a node, and (2) loads the requirements of a migrating task on the message that is transmitted to locate qualified nodes. A node's *state information* in this paper refers to information about the node's workload and its capabilities, i.e., the locally available resources.

In this paper, multicasting in TCP/IP networks is explored as the basis for loaded multicasting. Multicasting in TCP/IP, or multicast IP, is based on RFC-1112 [3]. TCP, the Transmission Control Protocol, sits on top of IP, the Internet Protocol. TCP provides a reliable stream-oriented full-duplex stream between two sockets, and while multicasting is not supported in TCP, it is supported at the IP level.

All IP addresses are 32 bits long, but the four higher-order bits of multicast IP addresses must be set to "1110", leaving 28 lower-order bits for address variation. Loaded multicasting uses the remaining 28 bits to store special information: (1) state information of nodes reflected on their *loaded multicast addresses*; and (2) requirements of migrating tasks reflected on *loaded multicast messages*. In TCP/IP networks, messages are accepted by nodes if the destination address of the message matches the address of the node. In loaded multicasting, this match implies that the capabilities of the node matches the requirements of the migrating task for which the message was sent to find an appropriate host. One variation may be that the message was sent by a node to offer its resources, thereby advertising its willingness to host migrated tasks.

3.1 One Bit : One Resource Type

In [14], each of those 28 lower-order bits of the multicast address reflected a single resource type. It can be proven that this is an ineffective approach. Assume this simple load sharing scheme:

In this distributed system with 28 resource types, a heavily-loaded node will join one or more multicast groups corresponding to the resource types required by its excess tasks. The resource requirements of excess tasks are used to derive corresponding loaded multicast addresses. The node will then wait for a loaded multicast message from some idle node that is offering a matching set of resource types which it can provide.

Consider the excess tasks t_1, t_2 and t_3 of the heavily-loaded node n_h, and their corresponding resource requirements:

$R(t_1) = \{r_1, r_3\}$,
$R(t_2) = \{r_1, r_3\}$,
$R(t_3) = \{r_5, r_6\}$

Node n_h must therefore join at least two multicast groups, with these addresses:

1110 1010 0000 0000 0000 0000 0000 0000, for r_1 and r_3, and
1110 0000 1100 0000 0000 0000 0000 0000, for r_5 and r_6

The four higher-order bits are always "1110" as required in IP multicasting. The 28 lower-order bits are used as flags to indicate relevant resource types. Note that bit locations are numbered 1 to 28 to correspond with resource types r_1 to r_{28}. The first address shows bit locations 1 and 3 (of the 28 lower-order bits) are set to 1, since $R(t_1) = R(t_2) = \{r_1, r_3\}$. The second address shows bit locations 5 and 6 (of the 28 lower-order bits) are set to 1, since $R(t_3) = \{r_5, r_6\}$.

Assume that some node n_i has become idle. It will form and send out a multicast message known as an *offer*, reflecting which resource types are locally available, i.e., $R(n_i)$. For example, if r_1 and r_3 are both available in n_i, the offer should be addressed with this multicast address:

1110 1010 0000 0000 0000 0000 0000 0000, for r_1 and r_3

It matches with the address of the first of the two multicast groups that n_h is member of so n_h can send tasks t_1 and t_2 to n_i.

But what if all resource types $r_1, r_2, .., r_{28}$ were all available to n_i? Then the destination multicast address of the offer from n_i would instead be:

1110 1111 1111 1111 1111 1111 1111 1111,
i.e., all resource types are available

This address would no longer match with any of the multicast groups joined by the heavily-loaded n_h! And if n_h were to compensate by subscribing to all multicast groups where bit locations 1 and 3 (of the 28 lower-order bits) are set to 1, simple calculations will show that this means joining $2^{28} - 3$ multicast groups, which is unacceptable.

3.2 Resource Type Subgroups

It is necessary, therefore, to modify the addressing scheme of loaded multicasting. The set of resource types $R(D)$ will be divided into s *resource type subgroups*, corresponding to s multicast groups that nodes may join. And just as $R(D)$ should be well-known to all nodes, so should the resource type subgroups.

For example, assuming $s = 7$, and assuming a total of 28 resource types, subgroups can be identified using these multicast addresses:

1110 1000 0000 0000 0000 0000 0000 0000 for group 1, for r_1, r_2, r_3 and r_4
1110 0000 1000 0000 0000 0000 0000 0000 for group 2, for r_5, r_6, r_7 and r_8
1110 0000 0000 1000 0000 0000 0000 0000 for group 3, for r_9, r_{10}, r_{11} and r_{12}
1110 0000 0000 0000 1000 0000 0000 0000 for group 4, for r_{13}, r_{14}, r_{15} and r_{16}
1110 0000 0000 0000 0000 1000 0000 0000 for group 5, for r_{17}, r_{18}, r_{19} and r_{20}
1110 0000 0000 0000 0000 0000 1000 0000 for group 6, for r_{21}, r_{22}, r_{23} and r_{24}
1110 0000 0000 0000 0000 0000 0000 1000 for group 7, for r_{25}, r_{26}, r_{27} and r_{28}

The heavily-loaded n_h in the load sharing example above should then join multicast group 1 since tasks t_1 and t_2 require resource types r_1 and r_3, and it must also join multicast group 2, since task t_3 requires resource types r_5 and r_6. The offer from idle processor n_i would therefore be addressed to multicast group 1, since resource types r_1 and r_3 are available in n_i.

However, this scheme does not provide specific information about resource types in the loaded multicast addresses. The solution is to carry the specifics in the message data field of loaded multicast messages, which are IP datagrams. The minimum size of an IP datagram is from five 32-bit words to a maximum of 64KB. This also raises the upper limit for the number of resource types in $R(D)$ significantly, assuming a one-bit to one resource-type correspondence in the data field. The tradeoff is that nodes may receive loaded multicast messages that do not pertain to them. For example, assume that in the given example of n_h and n_i above, there was another heavily loaded node n_j with excess task t_{11} which it requires resource types r_1 and r_2. So node n_j

must join multicast group 1. Note that the offer from the idle n_i would also be accepted by n_j even if n_i cannot provide r_2 which t_{11} needs.

So using resource type subgroups, although the number of resource types in D have increased significantly, and the number multicast groups has been limited, the matching loaded multicast messages to addresses is no longer exact. But note that the overhead incurred is internal and network traffic is not adversely affected – please recall that D is implemented on a single network without routers between nodes.

And while it is now possible to divide $R(D)$ to as many as 2^{28} resource type subgroups, since there are 2^{28} unique multicast addresses possible, this may be excessive, although the proof of this will not be dealt with in this paper. But the approach advocated here is simply to divide $R(D)$ to a reasonable number s of resource type subgroups such that subgroups may consists of roughly the same number of resource types that *actually* exist or are *projected* to exist. As stated, 2^{28} resource type subgroups is too large.

4 Load Distribution with Loaded Multicasting

Two load distribution algorithms are presented below that apply resource matching. Both apply the use of loaded multicasting with well known resource type groups, as described in the earlier section. While the algorithms do not look special at first glance, note that *no state probing or state gathering* is required prior to the process of sharing or balancing load.

4.1 Load Sharing

The goal in this load sharing scheme is for overloaded nodes to delegate tasks to idle nodes that are qualified for those tasks. A heavily-loaded node n_h will join one or more multicast groups corresponding to the resources required by excess tasks. For each excess task, its resource requirements are used to derive the address of the corresponding multicast group to join. Multiple tasks may have the same resource requirement and therefore correspond to the same multicast group.

An idle node n_i forms and sends out a loaded multicast message known as an *offer*, which reflects which resource types are locally available, i.e., $R(n_i)$. The offer's address is made to reflect the appropriate resource type subgroups.

The heavily-loaded node n_h will receive the offer if the resource type subgroup of the resources required by its excess task or tasks matches the resource type subgroup of the resources being offered by n_i. And if the specific resource types match, i.e., $R(n_h) \cap R(t) = R(t)$, then excess tasks which apply will immediately be migrated to the idle node who made the offer. There may be several approaches regarding how many and which of the excess tasks, if there are several, will be sent first.

4.2 Load Balancing

This load balancing scheme uses a lenient definition of global balance. Each node maintains a local load threshold. If the current local workload is below the local load threshold, then the node is classified as lightly-loaded. If the current local workload exceeds this threshold at the generation of a new task, this node has become a heavily-loaded node, and must initiate the attempt to migrate this new task to a lightly-loaded node.

A heavily loaded node n_h forms a loaded multicast message known as a *call* such that the call's destination address reflects the subgroup of the resource type needed by its excess task t, i.e., $R(t)$. It transmits the call once and sets a timeout value.

The lightly-loaded node n_i can be located since it joins the multicast groups corresponding to the subgroups of resource types it can offer, based on $R(n_i)$. When it receives the call from a heavily-loaded node, implying that the resource type subgroups match, the lightly-loaded node must check the data field of the call to check if the specific resource types required are locally available. If so, it informs the caller. The excess task is immediately sent to the first lightly-loaded and qualified node that responds to the call before the timeout set by the heavily-loaded node lapses. Subsequent responses are turned down. If the timeout lapses then some action to handle this event must be taken, e.g., reevaluate the local workload – perhaps it is now below the threshold, re-transmit the call, abort the task and raise an exception, etc..

5 Conclusions

The paper described loaded multicasting applied to load sharing or load balancing in a distributed system using a TCP/IP network. It also enhanced early efforts in loaded multicasting [13, 14] and provides a more effective solution, albeit with some overhead. The approach tolerates heterogeneity of nodes in terms of what resources are available in individual nodes, which is considered a significant issue. The solution also avoids overhead incurred by gathering of state information and simplifies advertising, bidding and other forms of negotiation present in other load sharing or load balancing algorithms. As a side effect of using well known resource types and resource type subgroups, some degree of transparency is achieved, since applications may be written that refer to resource types without referring to their actual locations in the system. However, the study did not consider the following:

1. What was the actual performance of the given algorithms in terms of task execution and network overhead?
2. Can a well-known hashing function [1] replace the use of well known resource types and the resource type subgroups?
3. How can changes in workload be handled during load distribution or during migration?
4. How can fault tolerance be implemented?
5. How can the algorithms be enhanced to allow tasks to migrate in mid-execution?

These and other improvements should be attempted and subsequently prototyped in an actual system such as Nectar or Beowulf. It would also be interesting to see how loaded multicasting can be applied to other applications such as distributed blackboard systems and multi-agent systems [14].

Acknowledgements. Early work on this project was through the author's master's thesis, which was made possible through a scholarship grant from De La Salle University and the Philippine Government's Department of Science & Technology, through its Engineering & Science Education Project. Subsequent work was supported through a joint scholarship from Monash University's School of Computer Science & Software Engineering and the school's Distributed Systems & Software Engineering Research Group, including partial funding from the Co-operative Research Centre Programme through the Australian Government's Department of Industry, Science and Resources.

References

[1] Ahammad, M., Ammar, M. H., Bernabéu-Aubán, J. M., and Khalidi, M. Y. Using multicast communication to locate resources in a LAN-based distributed system. In *Proc. IEEE 13th Conference on Local Computer Networks* (October 1988), pp. 193–202.

[2] Decker, T. Virtual data space - a universal load balancing scheme. In *Proceedings of the 4th International Symposium on Solving Irregularly Structured Problems in Parallel, IRREGULAR'97* (1997), pp. 159–166.

[3] Deering, S. Host extensions for IP multicasting. Published as RFC 1112, Network Working Group, August 1989.

[4] Diekmann, R., Monien, B., and Preis, R. Load balancing strategies for distributed memory machines. In *Multi-Scale Phenomena and their Simulation*, Karsch, Monien, and Satz, Eds. World Scientific, 1997, pp. 255–266.

[5] Garg, V. K. *Principles of Distributed Systems.* Kluwer Academic Publishers, 1996.

[6] Goscinski, A. *Distributed Operating Systems: The Logical Design.* Addison-Wesley Publishing Company, Inc., 1991.

[7] Maheswaran, M., Ali, S., Siegel, H. J., Hensgen, D., and Freund, R. F. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Proceedings of the 8th IEEE Workshop on Heterogeneous Computing Systems (HCW '99)* (San Juan, Puerto Rico, April 1999).

[8] Lüling, R., and Monien, B. A dynamic distributed load balancing algorithm with provable good performance. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures (SPAA '93)* (1993).

[9] Lüling, R., Monien, B., and Ramme, F. Load balancing in large networks: A comparative study. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing (SPDP'91)* (1991), pp. 686–689.

[10] Ridge, D., Becker, D., Merkey, P., and Sterling, T. Beowulf: Harnessing the power of parallelism in a Pile-of-PCs. *IEEE Aerospace* (1997).

[11] Shin, K. G., and Chang, Y.-C. A coordinated location policy for load sharing in hypercube-connected multicomputers. *IEEE Transactions on Computers 44*, 5 (May 1995), 669–682.

[12] Steenkiste, P. Network-based multicomputers: A practical supercomputer architecture. *IEEE Transactions on Parallel and Distributed Systems 7*, 8 (August 1996), 861–875.

[13] Tan, J. Msc thesis: A dynamic, group-based load balancing algorithm with resource matching. De La Salle University, Manila, Philippines, 1994.

[14] Tan, J. Cost-efficient load distribution using multicasting. To appear in: Proc. First IEEE International Workshop on Cluster Computing, Melbourne, Australia, December 1999.

[15] Tanenbaum, A. S. *Distributed Operating Systems.* Prentice-Hall, Inc., 1995.