Dynamic Node Reconfiguration in a Parallel-Distributed Environment

Michael J. Feeley, Brian N. Bershad, Jeffrey S. Chase, and Henry M. Levy

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

Idle workstations in a network represent a significant computing potential. In particular, their processing power can be used by *parallel-distributed* programs that treat the network as a loosely-coupled multiprocessor. But the set of machines free to participate in load sharing changes over time as users come and go from their workstations. To make full use of the available resources, parallel-distributed applications in the network must reconfigure to adapt to these changes as they run.

This paper describes a node reconfiguration facility for Amber, an object-based parallel programming system for networks of multiprocessors. We describe system support that allows a parallel Amber application to adapt to changing network conditions by expanding to make use of new nodes as they become idle, and by contracting as nodes become busy. A key characteristic of Amber's node reconfiguration is that it is handled at the user level in the Amber runtime system; it does not depend on a kernel-level process migration facility. Our experiments with Amber show that node reconfiguration can be implemented easily and efficiently in a runtime library.

1 Introduction

For nearly two decades, networks of workstations have been used as inexpensive multiprocessors with only moderate degrees of success. However, several technological trends promise greater success in the future.

- An order-of-magnitude increase in local area network speed will substantially reduce the cost of inter-node communication, making distributed data sharing more practical.
- With the dramatic increase in processor speeds, the aggregate computational power of workstation networks will exceed that available in supercomputers, and at much lower cost.
- Shared-memory multiprocessors will continue to gain popularity, providing a strong single-node basis for parallel programming.
- Research in parallel-distributed programming has produced systems that provide comfortable programming models and good performance [Li & Hudak 89, Chase et al. 89, Bennett et al. 90a, Arnould et al. 89].

Modern parallel-distributed programming systems restrict applications to run on a fixed set of hosts. Typically, a parallel-distributed program allocates some number of idle workstations at startup and leaves component tasks active on those machines until completion. The program is unable to take advantage of additional power as new processing nodes become idle during its execution. Worse, if the owner of one of the workstations returns, he must compete for resources with the program component running on his machine. The probable results include degraded program performance and an unhappy workstation owner, who may be inclined to abort the program. Such conflicts are likely during a long-running computation, unless the program limits its initial allocation to a small number of nodes with predictable usage patterns: i.e., nodes likely to remain idle. This severely restricts the potential parallelism available to the applications, which tend to be highly decomposable and capable of using a large number of nodes.

The goal of our research is to allow and encourage the execution of parallel-distributed programs that adapt to changes in the set of available processing nodes on the network. We call such changes and the actions that must be taken to accommodate them *dynamic node reconfiguration*. Supporting dynamic node reconfiguration allows programmers to use idle workstations according to the needs of their programs, rather than the working hours of their colleagues.

This work was supported in part by the National Science Foundation (under Grants No. CCR-8619663 and CCR-8907666), the Washington Technology Center, an AT&T Fellowship, and Digital Equipment Corporation (Systems Research Center, External Research Program, and DECwest Engineering).

In this paper we describe the design, implementation, performance, and impact of dynamic node reconfiguration in Amber [Chase et al. 89]. Amber differs from earlier load sharing work [Eager et al. 86, Litzkow et al. 88, Theimer & Lantz 88], which makes use of idle network workstations by redistributing independent sequential processes using kernel-level process migration [Powell & Miller 83, Theimer et al. 85, Zayas 87, Douglis & Ousterhout 87]. Our work extends these ideas to the parallel-distributed case, which involves distributed programs composed of multiple processes cooperating to obtain speedup on a single parallel algorithm. Because control over locality, decomposition, and communication patterns are critical to the performance of a parallel-distributed program, we believe that a user-level approach to reconfiguration is more appropriate than one based exclusively on kernel-level mechanisms. In Amber, programs control the load redistribution policy through reconfiguration mechanisms implemented entirely at the user level within the Amber runtime library.

This paper is organized as follows. In the next section we give an overview of the Amber system. We motivate and explain Amber's user-level approach to reconfiguration in Section 3, and in Section 4 we contrast it with an alternative approach based on kernel-level process migration. In Section 5 we discuss some implementation issues for Amber's reconfiguration facility. In Section 6 we describe the system's performance. We present our conclusions in Section 7.

2 Amber Overview

Amber is an object-oriented parallel-distributed programming system that supports threads, a distributed shared object space, object migration, and object replication [Faust 90], for programs written in a variant of C++. Amber was designed to explore the use of a distributed object model as a tool for structuring parallel programs to run efficiently on a network of DEC Firefly shared-memory multiprocessor workstations [Thacker et al. 88]. Programs are explicitly decomposed into objects and threads distributed among the nodes participating in the application. Threads interact with objects through a uniform object invocation mechanism; remote invocations are handled transparently by migrating the thread, migrating the object, or replicating the object, as determined by the object's type. Amber has been used to experiment with a number of parallel-distributed applications, including iterative point-update (e.g., the Game of Life), jigsaw puzzle [Green & Juels 86], graph search, mesh-based dynamic programming [Anderson et al. 90], simulation, and an n-body gravitational interaction. This section describes those aspects of Amber that are relevant to our approach to node reconfiguration.

Amber can be viewed as a distributed shared memory system that uses language objects rather than pages as the granularity of distribution and coherence. This eliminates the false sharing inherent in page-based distributed memory systems, and it allows coherence and distribution policies to be selected on an object basis according to how objects are used [Bennett et al. 90b]. The maximum parallelism available to an Amber program is determined by its decomposition into objects and threads, and the assignment of objects to nodes determines the distribution of load. The programmer can control object location using explicit object mobility primitives, or by selecting an adaptive "off the shelf" object placement policy implemented with additional runtime support layered on the base system [Koeman 90]. These features are particularly useful for preventing load imbalances over a changing node set; objects can be migrated to balance the load.

An Amber program runs as a collection of processes distributed across a set of network nodes, some of which may be shared-memory multiprocessors. We will refer to these processes as *logical nodes*. A logical node is an address space that encapsulates all of the program code, data objects, and threads for one part of an Amber application. Normally, there is one logical node on each physical node participating in the execution of an Amber application. The Amber runtime system schedules lightweight Amber threads within the address space of each logical node. On multiprocessor nodes, these threads can execute in parallel. Object invocation uses shared memory and procedure calls when an object and thread are on the same machine. Runtime support code maps object moves and remote invocations into RPC calls between the logical nodes. In this way, local communication is efficient and remote communication is transparent.

An important aspect of Amber is its sparse use of virtual memory to simplify and speed up naming of distributed objects. Amber uses direct virtual addresses as the basis for object naming in the network. The address spaces of the logical nodes are arranged in such a way that any object address denotes the same object in any logical node. References to remote objects are detected at the language level and trapped to the runtime system. Amber can globally bind objects to virtual addresses in this fashion because each program executes in a private (but distributed) object space; one Amber program cannot name or use objects created by another program. This sets Amber apart from other location-independent object systems, such as Emerald [Jul et al. 88], which provide a single global object space shared by multiple programs.

3 Reconfiguration in Amber

A running parallel-distributed program can respond to three types of node reconfiguration: the node set can shrink in size, stay the same size but change membership, or grow. Dealing with these changes requires care at two levels. At the system level, all of the cases require system support to transfer program state from one node to another, and to forward subsequent com-

munication as needed. At the application level, the program must resolve load imbalances caused by growth and shrinkage of the node set; additional support is needed to react to changes in the available parallelism and communication patterns.

In Amber, reconfiguration is controlled by the runtime system, but may involve application code. The runtime system provides the mechanism for migrating state and adjusting the communication bindings, but the load balancing policy is left to object placement code at the application level. This section discusses these two levels and their interplay during node reconfiguration.

3.1 The Role of the Runtime System

Node set reconfiguration is handled at the user level without direct involvement from the operating system kernel. The program expands by migrating objects and threads to a newly created logical node running on the new physical node. It contracts by forcing a logical node to terminate after redistributing its objects and threads to the remaining logical nodes. In either case, the transfer of program state is initiated and controlled by the runtime sytem, possibly in cooperation with user code. The operating system kernel is not aware of the reconfiguration; it sees only a flurry of network activity, together with a process creation or death. For simple membership changes (replacing one node with another), this amounts to a user-level process migration facility for Amber logical nodes.

The runtime system is a natural place to support migration of program state. It has responsibility for managing distributed threads and maintaining coherency of the distributed shared memory, so it knows about threads and objects and can relocate them directly. It also knows details about address space usage that allow it to be frugal with network and memory bandwidth. For example, Amber does not copy data that is accessible on demand elsewhere in the network (e.g. replicated objects). It also ignores regions of sparse virtual memory that are unused or reserved for objects residing on other nodes.

3.2 The Role of the Application

The runtime system can independently handle any reconfiguration in a functionally correct way, but it relies on the application (or a higher level object placement package) to redistribute work so as to make the best possible use of the new node set. This must be done by migrating objects using standard object mobility primitives, placing them in a way that best meets the competing goals of balancing load and minimizing network communication. Achieving and maintaining a balanced assignment of work to processing nodes is the primary concern of parallel-distributed programming. The assignment must be adjusted to compensate for load imbalances that develop as the program executes, as well as for changes in the number of nodes. It depends on application-specific knowledge about the amount of computation associated with each object, and the patterns of communication between objects. The extent to which this knowledge is applied will always determine the effectiveness of the decomposition.

Our goal was to create an interface that allows load balancing policies to be tailored to meet the needs of the application. The information to drive the policy is provided by a predefined *NodeSet* object that serves as a channel of communication between the runtime system and the application. Table 1 lists the *NodeSet* operations that are visible to the application. The downcall interface is used by the application to request the current size of the node set, or to traverse the node set when distributing work. The upcall interface consists of procedures *Grow* and *Shrink* that are exported by the application and are called asynchronously by the runtime system when the size of the node set changes. *Shrink* is called before a node leaves the node set, to redistribute its work to the remaining nodes. *Grow* is called after a node is added, to migrate work to the new node.

Upcall Interface	
void	*Shrink*(**Node** *departingNode*)
void	*Grow*(**Node** *newNode*)
Downcall Interface	
int	*NodeCount*()
Node	*NextNode*(**Node** *referenceNode*)

Table 1: NodeSet Operations

4 The Process Migration Alternative

Our approach to node reconfiguration is based on migrating fine-grained program state between Amber logical nodes. Consider now an alternative approach based on a general-purpose process migration facility that migrates and schedules logical node processes without knowledge of their internal state. Simple membership changes are straightforward using kernel-level process migration (i.e., move an address space and its component threads). Expansion of the node set is made possible by decomposing the application into a larger number of operating system processes (logical nodes), allowing the use of as many physical nodes as there are logical nodes in the decomposition. A host could leave an application's node set simply by using process migration to send the local logical node processes to some other machine. Reconfiguration would be fully transparent to the application, assuming that it is sufficiently decomposed.

"Over-decomposing" at the process level is expensive because it involves heavyweight kernel abstractions

(e.g. address spaces). Over-decomposing at the object level is cheaper and yields finer-grained control over the distribution of work. It also serves as a basis for using application knowledge to balance the load at runtime, rather than simply to effect a static decomposition into component processes.

In addition, the process-level approach will tend to perform poorly when several logical nodes are placed on the same physical node. The system can maintain coherency between colocated address spaces using the same mechanisms that are used in the distributed case (transferring data with RPC), but this is much more expensive than using the physically shared memory provided by the hardware.

Overloading a physical node with multiple logical nodes can also cause scheduling problems. For example, computations in one logical node may block awaiting an event in another, and overloading may violate assumptions made by the user-level scheduling system about the number of physical processors available to it. In contrast, Amber's user-level approach preserves the one-to-one mapping of logical to physical nodes, placing all program state on a given physical node within a single address space. This avoids unnecessary interprocess communication, and it allows all Amber thread scheduling and synchronization to be handled using lightweight user-level mechanisms.

A final problem with the kernel-level approach is that it transfers the complete contents of a migrating address space, thereby needlessly moving data that is replicated and can be found elsewhere. Although system-level optimizations such as aggressive pre-copying [Theimer et al. 85], or lazy on-demand copying [Zayas 87], can reduce the latency of migration, the most effective optimization – no copying – can only be implemented with information kept at the user-level.

5 Implementation

This section describes the implementation of node reconfiguration support in the Amber runtime system. Sections 5.2 and 5.3 focus on managing the changing node set and the locations of objects. Section 5.4 describes mechanisms for dealing with internal runtime system state that must be preserved when a node leaves the node set.

5.1 Node Set Changes

Changes to the node set are controlled by the *NodeSet* object, initiated by its functions *RemoveNode*, *ReplaceNode*, and *AddNode*. Changes can be requested by the application itself, by calling one of these functions, or from outside the program, by invoking RPC operations exported by the Amber runtime system in each logical node. In the current implementation this is done manually by means of a command interface, but we plan to implement a server that monitors workstation availability and responds to changes by issuing reconfiguration commands to Amber applications.

Removing a node from the node set involves migrating application state from the departing node to a designated *replacement node*. The replacement node is specified as an argument to *ReplaceNode*, but it is chosen arbitrarily by the runtime system for *RemoveNode*. To remove a node, the runtime system freezes executing threads on both the departing node and the replacement node, and suspends attempts by other nodes to communicate with the departing node. Once both nodes are idle, the runtime system on the departing node uses RPC calls to load its application state into the logical node on the replacement host. When the transfer is complete, the departing logical node terminates and the replacement node resumes execution of user threads. Blocked messages to the departing node are then forwarded to the replacement node. The performance of the running program is degraded during this procedure, but the reconfiguration activity is otherwise transparent to the application.

The node removal procedure is more complicated if the change is a *RemoveNode* rather than a *ReplaceNode*. In this case, the runtime system first notifies the application by upcalling *Shrink* asynchronously and waiting for a selectable time limit. This allows *Shrink* to begin redistributing the departing node's objects to the remaining nodes. If the time limit expires, the thread executing *Shrink* is paused along with the other executing threads, and migrated directly by the runtime system along with the remaining program state. When the replacement node resumes, the *Shrink* thread executes to completion.

5.2 Propagating Node Set Knowledge

The Amber runtime system maintains information about the set of nodes participating in the program, including the size of the node set, the identities of the nodes, and the RPC bindings needed to communicate with them. Each logical node keeps a local copy of the tables with the latest information about the current state of the node set. Keeping perfect knowledge in a dynamic world would require the system to synchronously inform every node of changes to the node set, updating all of their local tables atomically. Instead, the *NodeSet* object maintains a centralized reliable copy of the tables, updating them synchronously with each change and allowing the updates to propagate lazily to the rest of the network. The Amber runtime system manages local tables as a cache over the master tables; it detects and refreshes stale information.

Each logical node is uniquely identified by a *node number* assigned to it when it first joins the application's node set. These are used to name the node in system data structures that store information about the node set and the locations of remote objects. Node

numbers are frequently transmitted between logical nodes as a side effect of the algorithm for locating objects. A node learns of an addition to the node set when it receives a node number for which it has no cached RPC binding. It responds by obtaining the RPC binding for the unknown node from the *NodeSet* object, and caching it locally.

Similarly, nodes discover deletions from the node set when their attempts to communicate with the departed node fail. When a logical node leaves the node set, its node number is inherited by its replacement node. Thus the mapping from node numbers to logical nodes is many-to-one. Reassigning a node number in this fashion may invalidate RPC bindings for that node number cached elsewhere in the network. An attempt to use a stale RPC binding is caught by the Amber runtime system. The runtime system handles the failure by querying the *NodeSet* object for the new binding, updating its local cache of bindings, and retrying the call.

5.3 Locating Remote Objects

Amber's scheme for finding remote objects is based on *forwarding addresses* [Fowler 85]. Each time an object moves off of a node it leaves behind the node number of its destination as a forwarding address. The forwarding address may be out of date if the object moves frequently. In this case the object's location can be determined by following a chain of forwarding addresses, since the object leaves a new address on each node that it visits. If an object is referenced from a node where it is not resident, and where no forwarding address is stored, the search for the object is made by following the forwarding chain beginning at the node where the object was created. Each node caches tables associating regions of memory with node numbers; these can be used to determine the node on which an object was created, given the virtual address of the object.

Node reconfiguration complicates management of forwarding addresses. When a node is removed from the node set, the forwarding addresses stored on the departing node must be preserved by migrating them to the replacement node. But a migrating forwarding address will overwrite forwarding addresses already stored on the replacement node. If the replacement node is a link in the chain from the departing node to the object, then destroying the address stored there creates a cycle in the forwarding graph, causing the object to become unreachable from some nodes. To prevent this, the forwarding addresses associated with each object are timestamped with a counter that is stored with the object and incremented each time the object moves. A forwarding address is never overwritten with an older one; each forwarding address on the departing node is compared with its counterpart on the replacement node, and the older of the pair is discarded.

5.4 Activation Records

The problem for node removal is to transparently preserve application state on the departing node by migrating it to a safe place where it can be found when it is needed by the remaining nodes. The bulk of this state is in the form of application objects and threads, but there are other data structures maintained by the runtime system that also must be dealt with. Other nodes assume that this state exists, and expect to find it by resolving a node number embedded in their local data structures. Amber preserves this state by migrating it to the replacement node, which inherits the node number of the departing node.

Stack activation records are the most difficult example of this kind of state. Activation records are not objects; an Amber thread stack is a contiguous region of virtual memory handled in the standard way by machine instructions for procedure call and return. Due to the way that Amber remote object invocations are handled, a stack may have valid regions on several different nodes, associated with invocations of objects residing on those nodes. The Amber remote invocation code modifies the stack so that a procedure return into a remote activation record causes the thread to return to the node where the actual activation record resides. Of course, activation records residing on a node must be preserved if the node is removed from the node set. Our changes to Amber for node set reconfiguration included code to update a list of valid stack fragments on each remote invocation, and to merge valid activation records into formerly invalid regions on the replacement node when a node set change occurs.

6 Performance

The performance of the node reconfiguration mechanism can be evaluated from two different perspectives. The first is concerned with the amount of time that workstation owners must wait to reclaim their workstations from parallel-distributed applications. The second is concerned with the performance of an application under reconfiguration.

6.1 Reclaiming a Workstation

When an application receives a command to extract itself from a node, application threads on the node are stopped within a fixed period of time specified as an argument to the command. This quickly returns a considerable portion of the workstation resources back to its owner. The time for the subsequent migration of application state from the departing node is dominated by the communication latency of the transfer. The primary factors affecting this time are the number and size of objects resident on the departing node. Table 2 shows the relationship between overall migration time and the number and size of migrated objects. Our current implementation does not include a bulk

object transfer facility capable of packing and shipping multiple objects at once; the cost for moving a large number of objects will be substantially improved once this feature is implemented.

Number of Objects	Object Size (bytes)				
	200	400	800	1600	3200
100	0.84	0.97	1.15	1.86	2.84
200	1.53	1.72	2.09	3.46	5.85
300	2.23	2.47	3.05	5.15	6.68
400	2.90	3.23	3.95	6.77	11.51
500	3.54	3.94	4.87	8.35	13.18

Table 2: Total Migration Time (seconds)

6.2 Application Performance

The performance of an application in a dynamic node set depends on how well the application can adapt to reconfiguration. We built an adaptable version of Red/Black Successive Over-Relaxation (SOR), an iterative point-update algorithm that computes values for points in a grid based on the values of their North, South, East, and West neighbors. This section discusses the steps we took to make SOR adaptable, and presents resulting performance data.

Static node set implementations of SOR divide the grid into *clusters* of adjacent points, distributing one cluster to each node that is available to the application. If SOR is to adapt to changes in the number of available nodes, it must be able to redistribute its work dynamically to balance the load among currently available nodes. The approach we took to achieve this was to over-decompose the problem into 24 clusters. We chose this number so that we could get balanced distributions of clusters to 2, 4, 6 and 8 nodes. Having more than one cluster per node may result in a performance penalty (compared to one cluster per node) due to an increase in scheduling overhead and cross-cluster communication. Running SOR on a 1200*1200 grid, we found that a 24-way partitioning of the grid was 10% slower on two nodes than a minimally decomposed (2-way partitioned) version. The penalty increased to 20% when we decomposed to 64 clusters.

To effectively overlap computation and communication in SOR, threads synchronize at a barrier at key points in each iteration. To achieve the same effect when there is more than one cluster per node and thus more than one barrier, we introduced a node barrier object on which all clusters synchronize. Without this barrier, we pay an additional 15% performance penalty for over-decomposition.

Table 3 shows the results of our experiments with SOR using 24 clusters with a problem size of 1200 by 1200 points. We changed the size of the node set ten times, performing 30 iterations in each of the ten phases; the complete 300 iterations lasted about an hour. The second column of Table 3 shows the number of nodes participating in each phase. Between phases, the application either expanded or contracted as indicated by the table. Column 3 shows the time needed to reconfigure. For node set expansion, this includes a delay of about five seconds to start new logical node processes. Column 4 shows the time taken for the application and runtime system to relocate all of its local objects. Finally, the last two columns show, respectively, the time to execute 10 iterations of the algorithm and the total time spent in that phase of the program. The table shows that the program adapted effectively to changes in the number of nodes. Program phases executing on larger numbers of nodes were able to complete their 30 iterations in less time than phases executing on smaller numbers of nodes. Program phases that were forced to contract back to a smaller number of nodes completed their iterations in about the same amount of time as earlier phases on the same number of nodes.

7 Conclusions

We have described a facility for adapting to changing numbers of nodes in the Amber parallel-distributed programming environment. Amber's fundamental approach to dynamic node reconfiguration involves three aspects: (1) user-level support for a distributed, shared, object memory, (2) programmer-controlled object mobility, and (3) a mechanism for communicating node set changes to the application.

We have compared this approach to the more traditional process migration alternative. In the past, process migration has been used primarily to balance the load given a collection of independent non-cooperating processes. In our environment, however, the objective is to distribute interacting components of a single parallel program. To properly balance such components requires an understanding of both the computational and communication patterns of the various components. We believe that while a parallel application could be structured as a set of processes that rely on kernel-level process migration, a user-level approach is more appropriate because:

- It yields better local performance, due to the use of a shared address space on a node.
- It provides better control of parallelism, because fine-grained scheduling is done at the user level. The kernel does not need to schedule competing threads from different processes that are part of the same application.
- It permits more effective reconfiguration, because the user level knows which data to move and which data to ignore. Also, only the application can decide how to best redistribute objects among remaining nodes in order to optimize processing and communication.

Program Phase	Number of Nodes	Reconf. Time	Distribute Time	10 Iter. Time	Total Time (30 Iter.)
1	2	*na*	*na*	149.3	448.1
2	8	10.6	35.8	48.0	190.6
3	2	29.5	24.0	152.7	511.6
4	4	6.2	23.7	79.0	267.0
5	6	5.3	16.3	60.3	202.6
6	4	12.4	12.7	81.4	269.6
7	8	10.3	23.8	49.5	182.8
8	2	28.2	23.2	149.9	501.3
9	4	5.9	23.6	81.4	274.1
10	2	17.6	15.9	150.6	485.5

Table 3: 24 Cluster Adapting SOR (seconds)

Amber supports node reconfiguration by providing simple mechanisms for state migration, together with an interface that allows the application to adapt to a changing node set. We have shown how an application can use this interface, by overdecomposing at the object level. We are continuing research to determine the performance effects of reconfiguration on other parallel-distributed programs.

8 Acknowledgments

We would like to thank Rik Littlefield, Cathy McCann, Simon Koeman, Tom Anderson, Kathy Faust, and Ed Lazowska for discussing with us the issues raised in this paper.

References

[Anderson et al. 90] Anderson, R. J., Beame, P., and Ruzzo, W. L. Low overhead parallel schedules for task graphs. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 66–75, July 1990.

[Arnould et al. 89] Arnould, E. A., Bitz, F. J., Cooper, E. C., Kung, H. T., Sansom, R. D., and Steenkiste, P. A. The design of Nectar: A network backplane for heterogeneous multicomputers. *3rd Symposium on Architectural Support for Programming Languages and Operating Systems, Computer Architecture News*, 17(2):205–216, April 1989.

[Bennett et al. 90a] Bennett, J. K., Carter, J. B., and Zwaenepoel, W. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 125–134, May 1990.

[Bennett et al. 90b] Bennett, J. K., Carter, J. B., and Zwaenepoel, W. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 168–176, March 1990.

[Chase et al. 89] Chase, J. S., Amador, F. G., Lazowska, E. D., Levy, H. M., and Littlefield, R. J. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[Douglis & Ousterhout 87] Douglis, F. and Ousterhout, J. Process migration in the Sprite operating system. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 18–25, September 1987.

[Eager et al. 86] Eager, D. L., Lazowska, E. D., and Zahorjan, J. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.

[Faust 90] Faust, K. An empirical comparison of object mobility mechanisms. Master's thesis, Department of Computer Science and Engineering, University of Washington, April 1990.

[Fowler 85] Fowler, R. J. *Decentralized Object Finding Using Forwarding Addresses.* PhD dissertation, University of Washington, December 1985. Department of Computer Science technical report 85-12-1.

[Green & Juels 86] Green, P. and Juels, R. The jigsaw puzzle: A distributed performance test.

In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 288–295, May 1986.

[Jul et al. 88] Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Koeman 90] Koeman, S. Dynamic load balancing in a distributed-parallel object-oriented environment. Master's thesis, Department of Computer Science and Engineering, University of Washington, December 1990.

[Li & Hudak 89] Li, K. and Hudak, P. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Litzkow et al. 88] Litzkow, M. J., Livny, M., and Mutka, M. W. Condor - a hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, pages 104–111. IEEE Computer Society, June 1988.

[Powell & Miller 83] Powell, M. L. and Miller, B. P. Process migration in DEMOS/MP. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 110–119. ACM/SIGOPS, October 1983.

[Thacker et al. 88] Thacker, C. P., Stewart, L. C., and Satterthwaite, Jr., E. H. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[Theimer & Lantz 88] Theimer, M. M. and Lantz, K. A. Finding idle machines in a workstation-based distributed system. In *8th International Conference on Distributed Computing Systems*, pages 112–122. IEEE Computer Society, June 1988.

[Theimer et al. 85] Theimer, M. M., Lantz, K. A., and Cheriton, D. R. Preemptable remote execution facilities for the V-system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 2–12. ACM/SIGOPS, December 1985.

[Zayas 87] Zayas, E. R. Attacking the process migration bottleneck. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 13–24, November 1987.