Integration of the LAM/MPI environment and the PBS scheduling system

Brian Barrett[a*], Jeff Squyres[†], Andrew Lumsdaine[†]

[a]Open Systems Laboratory, Indiana University
{brbarret,jsquyres,lums}@lam-mpi.org

The growth of cluster computing as a viable option for high performance computing has lead to the development of a comprehensive software stack for these machines, including cluster scheduling, parallel environments, and scientific libraries. OpenPBS or PBS/Pro is often used for scheduling, with LAM/MPI or MPICH used for parallel communication. This paper details the integration of the PBS scheduling and resource managing infrastructure with the LAM/MPI parallel run-time environment. The integration provides a cluster with several features that, although commonly available on traditional supercomputers, have been conspicuously missing in cluster computing environments: fast job startup, proper resource cleanup, and detailed accounting.

1. Introduction

In recent years, the use of commodity hardware for high performance computing has grown dramatically. Commodity hardware, a fast network, and a small software stack present a complete clustering solution. With the advent of packages like OSCAR [1] and Scyld [2] to assist in installing software on the cluster, clusters have proven to be inexpensive, easy to build, and provide a formidable computational resource.

A common practice for clusters is the use of PBS[1] for batch scheduling of user jobs, combined with either an MPI [5,6] implementation or PVM [7,8] for a parallel run-time environment. Commonly used MPI implementations on clusters are LAM/MPI [9,10] and MPICH [11,12]. Both of these MPI implememntatoins by default use `rsh` or `ssh` to provide remote process startup.

Each of these software components is of high quality, but unlike traditional supercomputer stacks, cluster software tends to be poorly integrated. The MPI implementation and the batch scheduler do not interact, leading to a number of issues for both systems administrators and end users.

This paper presents a solution for integrating PBS and LAM/MPI. LAM/MPI is a freely-available, open source implementation of the MPI standard. The implementation includes all of MPI-1.2 and large portions of MPI-2, including dynamic process control, one-sided communication, C++ bindings, and MPI I/O (through ROMIO [13,14]). The project is currently developed by the Open Systems Laboratory at Indiana University and is available under a BSD-like license.

The remainder of this paper is structured as follows: Section 2 presents the motivations for this project, detailing the shortcomings of existing integration between PBS and MPI implementations. Section 3 presents a high-level overview of the architecture of both PBS and LAM/MPI. The integration architecture is presented in Section 4. Future work and suggested improvements for both LAM/MPI and the PBS interface are suggested in Section 5. Finally, results of the implementation are provided in Section 6 and conclusions are presented in Section 7.

2. Motivation

As discussed in the introduction, commodity clusters running MPI jobs with PBS for batch scheduling face a number of challenges. Current usage requires a parallel application to start as shown in Figure 1 - either `rsh` or `ssh` is used for remote job startup. Specifically, the MPI processes started on remote nodes "escape" PBS's tracking abilities, due to standard Unix process behavior (remote MPI processes are not the children of any PBS control / monitoring process). Because PBS is unaware of processes executing on remote nodes, it is unable to properly free resources at the end of a job's execution. Further, it is unable to provide adequate usage accounting numbers, providing only wall-clock time for a particular job.

A number of solutions have appeared for these problems, mostly as the result of sites sharing

*Work supported by a Department of Energy High Performance Computer Science fellowship
†Work supported by a grant from the Lilly Endowment
[1]PBS, the Portable Batch Scheduler, was commercialized after development for NASA. There are currently two versions of PBS under development: OpenPBS, which is available at no cost, and PBS/Pro, which is a commercial product from Veridian Software. In this paper, we use "PBS" to refer to both OpenPBS [3] and PBS/Pro [4].

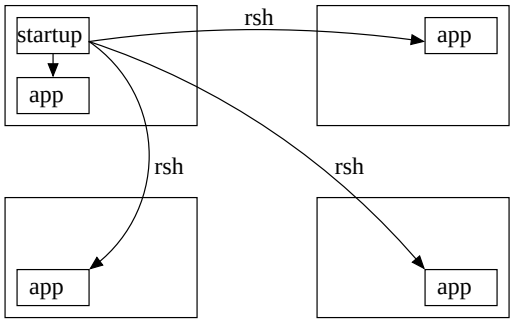

Figure 1. Starting a parallel application using `rsh` or `ssh`. PBS is only able to track resource utilization on the node from which the application is started.

scripts written to solve a problem seen on their cluster. "Clean-up" scripts to kill processes remaining after a job ends are common of many PBS installations. In order to disallow users from bypassing the batch system, shell access is limited by a number of ingenious mechanisms, including modifying `/etc/passwd` on the client node to toggle the user's shell from `/bin/false` to a real shell upon scheduling by PBS. The myriad of possibilities dramatically increases the complexity of installing and maintaining such a cluster. Note, however, that none of these kinds of solutions provides accurate accounting statistics. At best, the wall clock time of the job multiplied by the number of CPUs can determine approximate CPU usage.

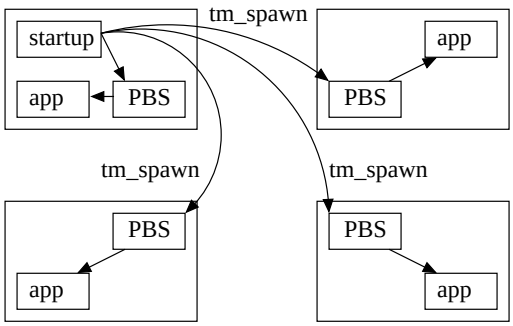

Figure 2. Starting a parallel application using PBS's TM interface. PBS is able to track resource utilization for the entire application.

PBS provides an interface, referred to as the Task Management (TM) interface [15], for job control in the PBS environment. PBS's TM interface is a subset of the PSCHED interface [16]. The PSCHED API aimed to provide a complete interface for parallel job and resource management. Utilizing the TM interface results in an application startup similar to that shown in Figure 2. All processes started by the TM interface remain under PBS control, allowing both resource cleanup and process accounting.

By modifying LAM/MPI to utilize the TM interface, we are able to provide a comprehensive job control facility under PBS. The integration allows LAM/MPI to:

- Start run-time environment without the use of `rsh` or `ssh`
- Provide guaranteed cleanup of resources
- Enable PBS to collect accounting information from all processes in the MPI application

As a side-effect, the startup time of LAM/MPI under PBS was dramatically reduced, even for small numbers of nodes.

3. Systems Overview

This section presents the general architecture of both LAM/MPI (Section 3.1) and PBS (Section 3.2), with a focus on issues that affect integration.

3.1. LAM/MPI

LAM/MPI is designed with two major layers, the LAM layer and the MPI layer, as shown in Figure 3. The LAM layer provides a framework upon which the MPI layer executes. The LAM layer provides a complete message passing service as well as process control, remote file access, and `stdio` forwarding. The MPI layer provides the MPI interface and an infrastructure for direct communication with high-speed networks.

LAM utilizes a user-level daemon to provide many of the services needed for an MPI run-time execution environment. The daemons are started at the beginning of an execution session using the `lamboot` command, a process often referred to as "booting LAM." At the end of an execution session, the daemons are killed using the `lamhalt` command. The network of daemons is referred to as the LAM run-time environment. Figure 4 depicts booting the LAM environment using `rsh`.

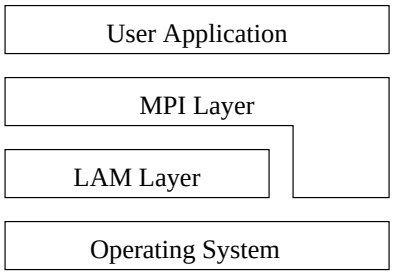

Figure 3. The layered design of LAM/MPI

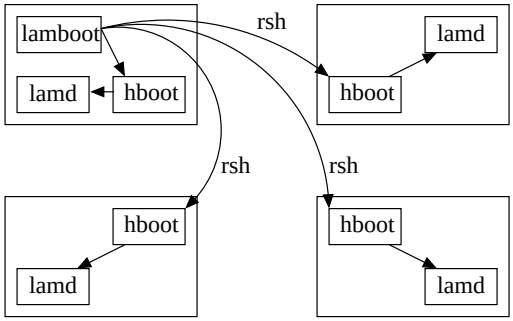

Figure 4. Booting of the LAM environment using `rsh`. The `lamd`s are the user-level daemons used by LAM. `hboot` provides start-up assistance for the LAM daemons.

The LAM daemons provide process control for all MPI jobs executed under LAM/MPI. `mpirun` launches an MPI application by sending a request to the daemons, which then `fork()`/`exec()` the user's application. At the end of the application's life, the daemons are notified of the process deaths through the standard Unix `SIGCHLD` mechanisms. Additionally, the daemons are able to stop execution of an MPI application running under the environment through the use of `kill()`.

3.1.1. System Services Interface

The development version of LAM/MPI has been redesigned to provide a plug-in module framework for various services provided by the LAM infrastructure. This framework, the System Services Interface (SSI), is composed of a number of "kinds" of plug-ins, each of which provide a single service to the MPI implementation. Modules are chosen at run-time, either by the SSI infrastructure or the user, allowing a particular version of LAM/MPI to support multiple underlying infrastructures. Currently, there are SSI interfaces for booting LAM, for the MPI device-dependent communication layer, and for MPI collective communication algorithms.

At present, there are two implementations of the boot SSI interface: RSH and TM. The RSH boot module allows the traditional `rsh`[2] booting mechanism. The TM boot module supports any batch scheduler that conforms to the PSCHED API, although PBS is believed to be the only scheduler with such support. Only one boot SSI module can be used to boot LAM – the run-time environment must either be booted entirely using TM or entirely using RSH. Although the user is free to specify a specific boot module, the SSI infrastructure can generally automatically select the most appropriate module at run-time.

3.2. PBS

PBS is composed of three components: the scheduler, server, and a per-node control daemon (referred to as the Machine Oriented Mini-server, or MOM), as shown in Figure 5. The scheduler provides job-to-node mappings and handles queue management. The server handles communication between components as well as accounting logs. There is generally one scheduler and one server per cluster. The PBS MOM executes on every node in the cluster and provides a number of health and process control features. The MOMs provide a heartbeat mechanism for the server so that jobs are not allocated to unresponsive nodes. The MOMs are also used by the server and TM interface for job startup.

PBS defaults to scheduling jobs on a per-node basis. However, PBS can use the concept of a Virtual CPU (VCPU) for per-CPU scheduling. The mapping between physical CPUs and VPCUs is arbitrary and set by the system administrator. Since PBS can not influence the OS scheduler on a cluster node, the mapping is simply a guideline. The concept is useful, however, for providing hints to the layers above PBS about how to schedule processes within a single job.

3.2.1. The TM Interface

The PBS implementation of the TM interface, provides a number of services for LAM's TM boot module, including remote or local process start-up (`tm_spawn()`), job node assignments

[2] `ssh` or any other remote process command with a similar interface may also be used.

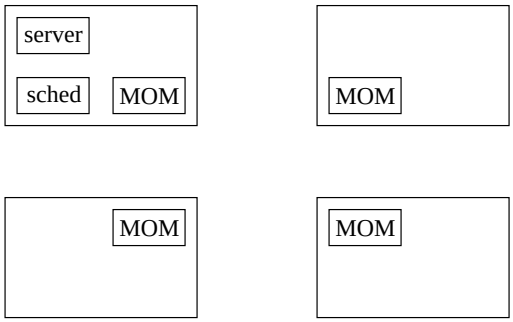

Figure 5. Architecture of a cluster running PBS. The MOMs are responsible for monitoring activity on each node and job startup. The scheduler allocates resources to jobs and communicates with the server, who is responsible for job startup, shutdown, and accounting logging.

(`tm_nodeinfo()`), and resource information for a particular node (`tm_rescinfo()`). `tm_spawn()` and `tm_rescinfo()` are both asynchronous events.

`tm_spawn()` starts a single process on a node specified by a `tm_node_id`. The spawned task is started by the relevant MOM, allowing PBS to track resource utilization for the spawned process and all its children. To allow resource accounting, if the `tm_spawn`ed process exits, all children of the spawned process are immediately killed by the MOM on that node.

In order to examine the execution environment, TM provides `tm_nodeinfo()` and `tm_rescinfo()`. `tm_nodeinfo()` provides a list of `tm_node_id`s – one for each VCPU allocated to the job. The `tm_node_id`s only have meaning within the current job and have no direct mapping to IP addresses or hostnames. Using `tm_rescinfo()`, it is possible to obtain the hostname that corresponds to a particular `tm_node_id`. `tm_rescinfo()` provides all the information normally provided by `uname -a` in a platform-independent format. From this, a hostname that will resolve across the cluster can be obtained.

4. Implementation Architecture

The integration of PBS and LAM/MPI involves three types of changes: implementation of the TM boot module (Section 4.1), improvements in the general LAM infrastructure (Section 4.2), and PBS-specific modifications that fall outside the TM boot module (Section 4.3).

4.1. The TM Boot Module

The traditional RSH booting mechanism, shown in Figure 4, consists of the following steps:

1. Parse hostfile and allocate nodes
2. Use `rsh` to start `hboot` on each node
3. `hboot` starts LAM daemon on each node
4. LAM daemons connect to `lamboot`, providing location information
5. `lamboot` broadcasts location information

`hboot` is responsible for starting the LAM daemons each node. It parses a configuration file to determine which applications to start, closes `stdout` and `stderr`, starts the applications, and exits. The configuration file may specify more than one application to start; using `hboot` allows multiple remote processes to start with only one call to `rsh`. Due to the overhead of `rsh`, this can greatly reduce boot time. Once the applications have launched, `hboot` exits, allowing `rsh` to complete.

The functionality of `hboot` is largely irrelevant in a PBS environment. Remote job startup with TM is significantly faster than with `rsh`, so the overhead of multiple calls to `tm_spawn()` per node is negligible. The configuration file can be parsed on the booting node rather than on each node. More importantly, `hboot`'s early exit causes PBS to kill all children of `hboot` as soon as it exits. Such behavior is obviously not desirable. Since `hboot` is largely rendered irrelevant in a PBS environment, the parsing functionality that it provides was moved into a library call; the `hboot` command is not utilized in the TM boot module.

The booting process for the TM boot module is shown in Figure 6. `lamboot` is run, often with no arguments. A hostfile is not needed, as LAM will obtain a list of target hosts from PBS. `lamboot` initializes the TM boot module, which initializes the TM interface. The TM boot module obtains a list of `node_id`s from PBS and uses `tm_rescinfo()` to find all the unique hosts in the list. If a hostname appears more than once, PBS has scheduled more than one VCPU on that host, and LAM will set its internal CPU count accordingly.

The boot configuration file is then parsed to determine which processes to start in order to form the LAM run-time environment.[3] The TM boot module then calls `tm_spawn()` once per process per node. `tm_spawn()` returns an event handle to allow the starting application to poll for a success /

[3]Normally, this is just one processes: the `lamd`.

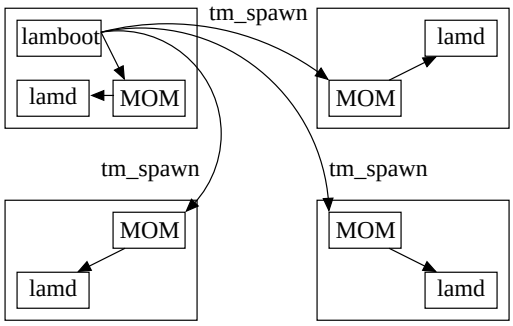

Figure 6. Booting of the LAM/MPI environment using PBS. `lamboot` calls `tm_spawn()` to directly execute the LAM daemons (`lamd`s).

failure in startup. However, the TM boot module ignores the event handle. The call-back in the next phase of booting will ensure that all processes started up correctly, so there is no need collect the events from PBS. The asynchronous nature of `tm_spawn()` means that LAM is able to start multiple processes in a parallel fashion, something that it is not capable of doing with the RSH boot module.

Once all the LAM daemons are started, the TM boot module behaves exactly like the RSH boot module. The LAM daemons each send their location to information to `lamboot`, who then broadcasts this information back out to all LAM daemons. At this point, the boot process is complete and the TM boot module is closed. `lamboot` returns and the user is able to run MPI jobs in the environment.

4.2. LAM Infrastructure Improvements

As a result of work on the TM boot module, some changes were made to LAM that are applicable in environments outside of PBS. The changes are enabled by default, regardless of which boot module is used. The notable changes are automatic resource cleanup if the LAM daemons receive a `SIGTERM` and proper usage of the `$TMPDIR` environment variable.

While PBS provides a guaranteed process cleanup, other resources may still be left in use by MPI applications. In particular, it is possible for temporary files and System V semaphores to remain if TM prematurely kills a LAM/MPI runtime environment. PBS provides a "warning" of impending death by sending a `SIGTERM` to all processes in a job. A short time period later, PBS forcibly kills all remaining processes with `SIGKILL`. By catching the `SIGTERM`, the LAM daemons are each able to launch a small cleanup process (`tkill`) that ensures that all LAM processes are killed, all System V semaphores used by LAM are freed, and all files and directories are deleted. The `SIGTERM` / `SIGKILL` procedure is a common way to "warn" processes they should exit. Therefore, it was felt that such functionality would be useful for cleanup outside of PBS environments.

PBS/Pro 5.0 also provides a `$TMPDIR` environment variable for every job that points to a unique temporary space for that job. The temporary space exists for the duration of the job and is removed during job post-processing. By utilizing the `$TMPDIR` directory, LAM is able to ensure all files are removed, even in the case of a malfunction by the LAM daemons. In addition, the `$TMPDIR` is standard UNIX practice, and therefore applicable outside of PBS environments.

4.3. Other Changes

The only PBS-specific modification to LAM that is not contained in the TM boot module is in the naming of LAM's temporary directory. As PBS defines the `$PBS_ENVIRONMENT` variable in all jobs, if LAM detects this variable, the unique PBS job ID is appended to the directory name. Since the directory name is used throughout LAM (not just in the boot modules), it did not make sense to add the functionality only in the boot module. In addition, there is support for other naming schemes that do not have their own boot module at this time.

5. Improvements

The TM boot module is largely feature complete. A small number of issues remain, mainly related to scalability on large clusters (Section 5.1). There are a number of restrictions placed on the environment, due either to LAM or PBS design decisions (Section 5.2). Finally, there are some issues with the TM interface that unnecessarily complicated the implementation of LAM's TM boot module (Section 5.3).

5.1. Remaining Work

The TM boot module still requires testing on large clusters. While there are no apparent limitations in the TM boot module that do not exist in the RSH boot module, only testing can verify the ability to run on hundreds of nodes.

There is also a problem of node failures during boot – the TM interface is not designed to handle failures during run time. The only practical solutions for LAM in the case of failure are to ei-

ther give up on booting and let the user attempt to restart the job (which PBS may be able to do automatically) – or to start the run-time environment on a subset of the nodes, those that remain alive. This means that the MPI universe would be smaller than requested, which may cause problems for some applications. Abort-on-failure is the is current behavior, as it appears to follow the "law of least astonishment" [17].

5.2. TM Boot Module Restrictions

The TM boot module does place some restrictions on the user of LAM/MPI that do not exist in the RSH boot module. With the RSH boot module, LAM/MPI can be installed in different places on each machine, as long as the LAM executables can be found in the `$PATH`. This scheme allows users to achieve multi-architecture support through the use of intelligent path settings. TM, however, requires that the full path to the executable be given to `tm_spawn()`, so LAM must be installed in the same place on all nodes in the cluster. Multi-architecture support is therefore more difficult on a filesystem shared across all nodes in the cluster. However, the common case for PBS installations is a cluster of the same architecture, so this limitation is not seen as unworkable.

LAM/MPI's RSH boot module allows booting with different usernames on each node. This functionality is not explicitly available in the TM boot module, as `tm_spawn()` does not provide a way to specify a username. As PBS does not appear to be able to create a job using different usernames on different nodes, this is not a limitation in LAM/MPI, but a design decision in PBS.

5.3. TM Improvements

Implementing a run-time environment startup infrastructure using the TM interface presented a number of challenges. Some of the challenges, such as the killing of all children of `tm_spawn()`ed processes, exist for technical reasons and cannot be avoided. Other aspects of the TM interface present issues for implementors of middleware, but do not appear to exist for any technical reason. For example, the TM interface is not well-suited for multiple-CPU execution. In order to map TM `tm_node_id`s to actual nodes, the application must parse a node information string to get the hostname, then do hostname comparisons. A format where this information was returned directly would be beneficial to application writers.

The requirement that `tm_spawn()` be given a full path name also presents problems in some situations. While homogeneous clusters will not experience problems with the requirement that a full path be provided, heterogeneous clusters will require more careful planning. A `tm_spawn_multiple()`, similar in functionality to MPI_SPAWN_MULTIPLE, would also be useful. This was suggested in the PSCHED API, but was not implemented in PBS's TM API.

6. Results

The implementation of the TM boot module enables process accounting and resource cleanup in a PBS environment. As a side-effect of the implementation, boot time was drastically decreased compared to `rsh` or `ssh`

All testing was performed on an OSCAR 2.1 cluster running Red Hat 7.3. Version 3.0.7p8 of the Maui scheduler, combined with OpenPBS 2.3.16 provided batch scheduling services. LAM/MPI was built from a CVS snapshot from January 30, 2003. The server node is a dual 1Ghz PIII machine with 512MB of RDRAM. The 8 compute nodes are single 1.4Ghz P4 machines with 756MB of PC133 SDRAM. The system is connected via 100MB ethernet and MDDS copper Myrinet 200, although the Myrinet was not used in any of the tests. The user account used for testing used the `tcsh` shell, with a `.tcshrc` that contained a single statement setting `$PATH`. SSH keys were setup using the default OSCAR mechanisms.

6.1. Process Accounting

The use of the TM interface enabled process accounting for the entire job. With `rsh` or `ssh`, process resources are only logged on a single node. However, with the TM interface, such control is now possible. Using the accounting logs provided by the PBS server, CPU time and memory usage can be tracked, in addition to wall-clock time. Figure 7 provides resource information for the execution of test program that executes a computational kernel for approximately 90 seconds.

6.2. Resource Cleanup

Through both the use of the PBS MOMs to start applications and the `SIGTERM` handling in the LAM daemons, LAM is able to properly return resources to the system after job execution. Testing was done under a number of scenarios, including PBS ending a job because resources (such as wall-clock time) had expired. In all cases, the environment was properly cleaned by the LAM infrastructure.

Boot	Nodes	Wall Clock	CPU	Mem
RSH	1	120s	111s	4396kb
TM	1	120s	119s	4396kb
RSH	8	100s	93s	4960kb
TM	8	108s	600s	16296kb

Figure 7. Accounting information from PBS, comparing a job run on 1 node and 8 nodes for a process with no run-time communication and requiring approximately 90 seconds of CPU time.

6.3. Improved Boot Performance

The TM boot module is able to quickly and efficiently bring up the LAM run-time environment on a homogeneous cluster of workstations. As shown in Figure 8, the startup time is drastically reduced compared to the RSH boot module, using either `rsh` or `ssh`. There are two reasons for the decrease. First, `rsh` must authenticate every time a remote process is launched, whereas the MOMs only authenticate at startup. Second, the TM boot module is not forced to serialize job startup. The system is capable of sending control messages out serially, but does not wait for the response before sending the next control message.

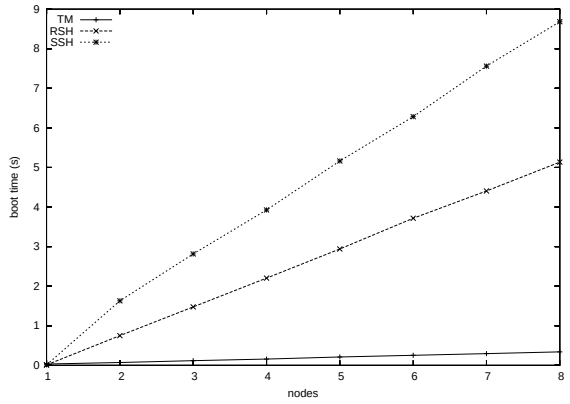

Figure 8. Startup time for LAM/MPI using SSH, RSH, and TM.

7. Conclusions

The ability of LAM/MPI to start jobs using the TM interface of PBS provides a number of important features. It removes the dependency of LAM/MPI on `rsh` or `ssh` in a cluster environment. Job startup time is greatly decreased. More importantly, the system is able to provide guarantees that resources are returned to the system at the end of a PBS job. Accounting is also enabled, proving usage statistics other than simply "wall-time."

Although the modification of LAM/MPI to better integrate with PBS is another step towards clusters providing the same feature set of traditional supercomputers, other parallel environments must still be modified to use the TM interface. Only when parallel environments provide this functionality can systems administrators remove the myriad of kill scripts and login controls from their systems.

The TM boot module and all modifications to the infrastructure that grew out of the project are currently available in LAM's CVS tree. Anonymous, read-only access is available for users wanting to utilize the latest features in LAM/MPI. The TM boot module is also scheduled to be included in the upcoming LAM/MPI 7.0 release. More information on the project can be found on the web:

`http://www.lam-mpi.org/`

REFERENCES

1. Open Cluster Group: OSCAR Working Group. OSCAR: A packaged cluster software for High Performance Computing. http://www.OpenClusterGroup.org/OSCAR/.
2. Scyld Beowulf Scalable Computing, http//www.scyld.com/.
3. Portable Batch System (OpenPBS), http://www.openpbs.org/.
4. Portable Batch System (PBS/Pro), http//www.pbspro.com/.
5. Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proc. of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, November 1993.
6. Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. MPI-2: Extending the Message-Passing Interface. In Luc Bouge, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par*

'96 Parallel Processing, number 1123 in Lecture Notes in Computer Science, pages 128–135. Springer Verlag, 1996.

7. Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: A Parallel Virtual Machine*. Scientific and Engineering Computation Series. MIT Press, 1994.
8. J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated PVM Framework Supports Heterogeneous Network Computing. *Computers in Physics*, 7(2):166–75, April 1993.
9. Greg Burns, Raja Daoud, and James Vaigl. LAM: An open cluster environment for MPI. In John W. Ross, editor, *Proceedings of Supercomputing Symposium '94*, pages 379–386. University of Toronto, 1994.
10. The LAM Team. *Getting Started with LAM/MPI*. University of Notre Dame, Department of Computer Science, `http://www.lam-mpi.org/`, 1998.
11. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.
12. William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.
13. Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, February 1999.
14. Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, May 1999.
15. OpenPBS 2.3.16. *TM(3) (manpage)*, May 1997.
16. The PSCHED API Working Group. PSCHED: An API for Parallel Job/Resource Management, November 1996.
17. Geoffrey James. *The Tao of Programming*. Info Books, Santa Monica, CA, 1987.