PMI: A Scalable Parallel Process-Management Interface for Extreme-Scale Systems *

Pavan Balaji[1], Darius Buntinas[1], David Goodell[1], William Gropp[2], Jayesh Krishna[1], Ewing Lusk[1], and Rajeev Thakur[1]

[1] Argonne National Laboratory, Argonne, IL 60439, USA
[2] University of Illinois, Urbana, IL 61801, USA

Abstract. Parallel programming models on large-scale systems require a scalable system for managing the processes that make up the execution of a parallel program. The process-management system must be able to launch millions of processes quickly when starting a parallel program and must provide mechanisms for the processes to exchange the information needed to enable them communicate with each other. MPICH2 and its derivatives achieve this functionality through a carefully defined interface, called PMI, that allows different process managers to interact with the MPI library in a standardized way. In this paper, we describe the features and capabilities of PMI. We describe both PMI-1, the current generation of PMI used in MPICH2 and all its derivatives, as well as PMI-2, the second-generation of PMI that eliminates various shortcomings in PMI-1. Together with the interface itself, we also describe a reference implementation for both PMI-1 and PMI-2 in a new process-management framework within MPICH2, called Hydra, and compare their performance in running MPI jobs with thousands of processes.

1 Introduction

While process management is an integral part of high-performance computing (HPC) systems, it has historically not received the same level of attention as other aspects of parallel systems software. The scalability of process management is not much of a concern on systems with only a few hundred nodes. As HPC systems get larger, however, systems with thousands of nodes and tens of thousands of processing cores are becoming common; indeed, the largest systems in the world already use hundreds of thousands of processing cores. For such systems, a scalable design of the process-management infrastructure is critical for various aspects such as launching and management of parallel applications, debugging utilities, and management tools. A process-management system must, of course, start and stop processes in a scalable way. In addition, it must provide mechanisms for the processes in a parallel job to exchange the information needed to establish communication among them.

Although the growing scale of HPC systems requires close interaction between the parallel programming library (such as MPI) and the process manager,

* This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract #DE-AC02-06CH11357; by the DOE grant #DE-FG02-08ER25835; and by the National Science Foundation under grant #0702182.

an appropriate separation between these two components is necessary. This separation not only allows for their independent development and improvement but also keeps the parallel programming library generic enough to be used with any process-management framework. At the same time, these two components must share sufficient information so as to allow the parallel programming library to take advantage of specific characteristics of the system on which it is running.

With these requirements in mind, we initially designed PMI, a generic process-management interface for parallel applications. In this paper, we start by describing the first generation of PMI (PMI-1). PMI-1 is widely used in MPICH2 [1] and other MPI implementations derived from it, such as MVAPICH2 [4], Intel MPI [6], SiCortex MPI [12], and Microsoft MPI [7] (for the programming library side) as well as in many process-management frameworks including MPICH2's internal process managers (Hydra, MPD, SMPD, Gforker, Remshell), and other external process managers such as SLURM [15], OSC mpiexec [9], and OSU mpirun [13] (for the process-manager side).

While extremely successful, PMI-1 has several limitations, particularly when applied to modern HPC systems. These limitations include issues related to scalability for large numbers of cores on a single node and efficient interaction with hybrid programming models that combine MPI and threads, amongst others. Building on our experiences with PMI-1, we recently designed a second-generation interface (PMI-2) that overcomes the shortcomings of PMI-1. The second part of the paper describes this new interface and a reference implementation of both PMI-1 and PMI-2 in a new process-management framework within MPICH2, called Hydra [5]. We also present performance results comparing PMI-2's capabilities to that of PMI-1 and other process-management interfaces on system scales of nearly 6,000 processes.

2 Requirements of a Process-Management Interface

In this section we provide a brief overview of what is required of a process-management interface for scalable parallel process management on large systems.

2.1 Decoupling the Process Manager and the Process-Management Interface

In our model, process management comprises three primary components: *(1)* the parallel programming library (such as MPI), *(2)* the PMI library, and *(3)* the process manager. These components are illustrated in Figure 1 with examples of different MPI libraries, PMI libraries, and process managers.

The process manager is a *logically* centralized process (but often a distributed set of processes in practice) that manages *(1)* process launching (including starting/stopping processes, providing the environment information to each process, stdin/out/err forwarding, propagating signals) and *(2)* information exchange between processes in a parallel application (e.g., to set up communication channels). Several process managers are available (e.g., PBS [8], SUN Grid Engine [14], and SSH), that already provide such capabilities.

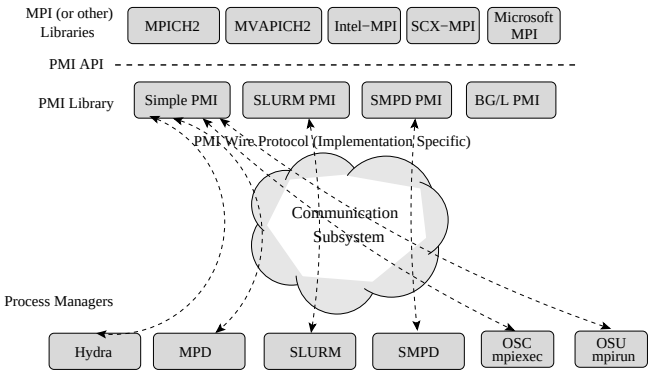

Fig. 1. Interaction of MPI and the process manager through PMI

The PMI library provides the PMI API. The implementation of the PMI library, however, might depend on the system itself. In some cases, such as for IBM Blue Gene/L (BG/L) [3], the library may use system-specific features to provide PMI services. In other cases, such as for processes on a typical commodity cluster, the PMI library can communicate with the process manager over a communication path (e.g. TCP). While the PMI library can be implemented in any way that the particular implementation prefers, in both PMI-1 and PMI-2 there is a predefined "wire protocol" where data is exchanged through the sockets interface. The advantage of using this protocol is that any application that uses the PMI API with the predefined PMI wire protocol is compatible with *any* PMI process manager implementation that accepts the wire protocol.

We note that the PMI API and the PMI wire protocol are separate entities. An implementation may choose to implement both, or just one of them. For example, the PMI library on BG/L provides the PMI API but does not use the sockets-based wire protocol. Thus, the library is compatible with any programming model using the PMI API, but it is not compatible with process managers that accept the sockets-based PMI wire protocol.

2.2 Overview of the First-Generation PMI (PMI-1)

Processes of a parallel application need to communicate with each other. Establishing this communication typically requires publishing a contact address, which may be an IP address, a remotely accessible memory segment, or any other interconnect-specific identifier. Since the process manager knows where all the processes are, and because it is (probably) managing some communication with the processes to handle standard I/O (stdin, stdout, and stderr), it is natural to have the process-management system also provide the basic facilities for information interchange. This is the key feature of our process-management interface, PMI—a recognition that these two features are closely related and can be effectively provided through a single service.

While PMI itself is generic for any parallel programming model and not just MPI, for ease of discussion we consider only the MPI programming model here. In the case of MPI, PMI deals with aspects such as providing each MPI process with

information about itself (such as its rank) as well as about the other processes in the application (such as the size of `MPI_COMM_WORLD`). Furthermore, each PMI process manager that launches parallel applications is expected to maintain a database of all such information. PMI defines a portable interface that allows the MPI process to interact with the process manager by adding information to the database ("put" operations) and querying information added by other processes in the application ("get" operations). The PMI functions are translated into the appropriate wire protocol by the PMI provider library and exchanged with the process manager. Most of the database is exchanged by using "key-value" pairs. Together with "put" and "get" operations, PMI also provides collective "fence" operations that allow efficient, collective data-exchange capabilities (the use of fence is described in more detail in Section 2.3).

As an example interaction between the MPI library, the PMI library, and the process manager, consider a parallel application with two processes, P_0 and P_1, where P_0 wants to send data to P_1. In this example, during MPI initialization, each MPI process adds to the PMI database information about itself that other processes can use to connect to it. When P_0 calls an `MPI_Send` to P_1, the MPI library can look up information about P_1 from the PMI database, connect to P_1 (if needed) by using this information, and send the data.

2.3 PMI Requirements for the Process Manager

In designing the process-management interface, there are two primary requirements for the process manager. First, a careful separation of features is needed to enable layering on a "native" process manager with the lowest possible overhead. This requirement arises because many systems already have some form of a process manager (often integrated with a resource manager) that is tightly tied to the system. A portable PMI must make effective use of these existing systems without requiring extra overhead (e.g., requiring no additional processes beyond what the native system uses). For example, an interface that requires asynchronous processing of data or interrupts to manage data might cause additional overhead for applications even when they are not interacting with the PMI services; this can be a major issue on large-scale systems. Second, a scalable data-interchange approach for the key-value system is needed.

This second requirement has a number of aspects. Consider a system in which each process in a parallel job starts, creates a "contact id," and wishes to make it available to the other processes in the parallel job. A simple way to do this is for the process to provide the data to central server, for example, by adding the data expressed as a (key,value) pair into a simple database. If all p processes do this with a central server, the time complexity is $\mathcal{O}(p)$; the time for all processes to extract just a single value is also $\mathcal{O}(p)$. This approach is clearly not scalable. Using multiple servers instead of a single one helps, but it introduces other problems.

Our solution in PMI is to provide a collective abstraction, permitting the use of efficient collective algorithms to provide more scalable behavior. In this model, processes put data into a key-value space (KVS). They then *collectively* perform a fence operation. Following completion of the fence, all processes can perform

a get operation against the KVS. Such a design permits many implementations. Most important, the fence step, which is collective over all processes, provides an excellent opportunity for the implementation to distribute the data supplied by the put operations in a scalable manner. For example, a distributed process manager implementation with multiple processes can use this opportunity to allow these processes to share their local information with each other.

3 Second-Generation PMI (PMI-2)

While the basic design of PMI-1 was widely adopted by a large number of PMI libraries and process managers, as we move to more advanced functionality of MPI as well as to larger systems, several limitations of PMI-1 have become clear. The second-generation PMI (PMI-2) addresses these limitations.

The complete details of the PMI-2 interface (including function names), and wire protocol are available online [10, 11]. To avoid dilution, we do not explicitly mention them in this paper. Instead, we describe the major areas in which PMI-2 improves on PMI-1.

Lack of Query Functionality: PMI-1 provides a simple key-value database that processes can put values into and get values from. While the process manager is best equipped to understand various system-specific details, PMI-1 does not allow it to share this information with the MPI processes. In other words, the process manager itself cannot add system-specific information to the key-value database; thus MPI processes cannot query such information from the process manager through PMI-1.

An example issue created by this limitation occurs on multicore and multiprocessor systems, where the MPI implementation must determine which processes reside on the same SMP node (e.g., to create shared-memory segments or for hierarchical collectives). Each process gathers this information by fetching the contact information for all other MPI processes and determining which contact addresses are local to itself. While this approach is functional, it is extremely inefficient because it results in $\mathcal{O}(p)$ PMI get operations for each MPI process ($\mathcal{O}(p^2)$ total operations).

PMI-2 introduces the concept of *job attributes*, which are predefined keys provided by the process manager. Using such keys, the process manager can pass system-specific information to the MPI processes; that is, these keys are added into the key-value database directly by the process manager with system layout information, allowing each MPI process to get information about the layout of all MPI processes in a single operation. Further, since the process manager knows that such attributes are read-only, it can optimize their storage by caching copies on local agents, thus allowing the number of PMI requests to be reduced from $\mathcal{O}(p^2)$ (in the case of PMI-1) to nearly zero (in the case of PMI-2)[3].

Database Information Scope: PMI-1 uses a flat key-value database. That is, an MPI process cannot restrict the scope of a key that it puts into the database;

[3] The read-only attributes still need to be fetched, which causes the number of requests with PMI-2 to be non-zero

all information is global. Thus, if some information needs to be local only to a subset of processes, PMI-1 provides no mechanism for the MPI processes to inform the process manager about it. For example, information about shared-memory keys is relevant only to processes on the same node; but the process manager cannot optimize where such information is stored or replicated.

To handle this issue, PMI-2 introduces "scoping" of keys as node-level and global (further restriction of the scope is not supported in PMI-2, in favor of simplicity as opposed to generality). For example, keys corresponding to shared memory segments on a node can be restricted to a node-level scope, thus allowing the process manager to optimize retrieval.

Hybrid MPI+Threads Programs: PMI-1 is not thread safe. Therefore, in the case of multithreaded MPI programs, the MPI implementation must protect calls to PMI-1 by using appropriate locking mechanisms. Such locking is often coarse-grained and serializes communication between the PMI library and the process manager. That is, until the PMI library sends a query to the process manager and gets a response for it (a round-trip communication), no other thread can communicate over the same socket. PMI-2 functions are thread-safe. Thus, multiple threads can communicate with the server in a more fine-grained manner, thereby pipelining requests better and improving performance.

Dynamic Processes: Each process group in PMI-1 maintains a separate database, and processes are not allowed to query for information across databases. For dynamically spawned processes, this is a severe limitation because it requires such processes to manually exchange their database information and load them into their individual databases. This procedure is cumbersome and expensive (with respect to both performance and memory usage). PMI-2 recognizes the concept of a "job" that can contain multiple applications connected to each other or where one is spawned from another. This allows such jobs to share database information without the need to explicitly replicate it.

Fault Tolerance: PMI-1 does not specify any mechanism for respawning processes when a fault occurs. Note that this is different from spawning an MPI-2 dynamic process, since such a process would form its own process group (`MPI_COMM_WORLD`) and not just replace a process in the existing process group. PMI-2 provides a concept of respawning processes, where a new process essentially replaces the original process within the same process group.

PMI-2 has been implemented as part of a new process-management framework in MPICH2, called Hydra [5].

4 Experimental Evaluation and Analysis

In this section, we present the results of several experiments that compare the performance of PMI-2 with that of PMI-1.

4.1 System Information Query Functionality

As described in Section 3, PMI-1 provides only a simple key-value database that processes can put values into and get values from, so a process manager cannot

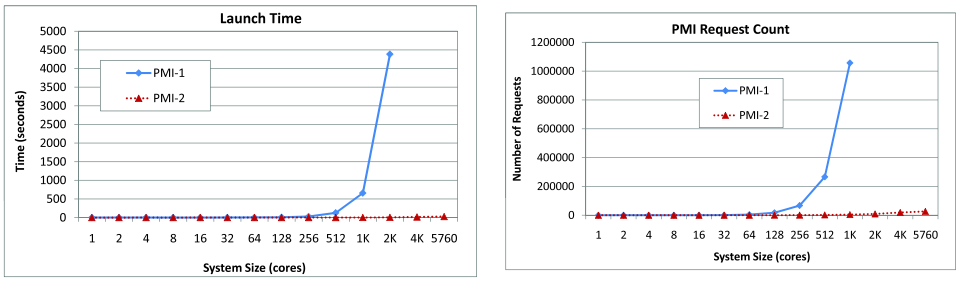

Fig. 2. Process launching on a 5760-core SiCortex system: (left) launch time and (right) number of PMI requests.

provide system specific information to the MPI processes. Thus, in order to determine which processes reside on the same SMP node, $\mathcal{O}(p^2)$ PMI operations are required. With PMI-2's job attributes, this reduces to nearly zero.

This behavior is reflected in the launch time of MPI applications. Figure 2(left) shows this behavior for a simple MPI application (that just calls `MPI_Init` and `MPI_Finalize`) with PMI-1 and PMI-2 on a 5760-core SiCortex system. As shown in the figure, the overall launch time increases rapidly with system size for PMI-1. With PMI-2, on the other hand, the time taken is significantly less. Figure 2(right) shows further analysis of the two PMI implementations with respect to the number of PMI requests observed by the process manager. This figure illustrates the reason for the performance difference between the implementations: PMI-1 has several orders of magnitude more PMI requests than does PMI-2.

4.2 Impact of Added PMI Functionality over the Native Process Manager

As described in Section 2.3, some systems already have a process manager (often integrated with a resource manager) that is tightly tied to the system. While some process managers might natively provide PMI functionality, others do not. An efficient implementation of the PMI interface must make effective use of such "native process managers" without requiring extra overhead (for example, by requiring heartbeat operations that wake up additional processes, thus disturbing the core computation). In this section, we evaluate this "noise impact" on 1600-cores of the SiCortex system using Class C NAS parallel benchmarks in two modes: *(1)* using the native process manager on the system, SLURM, that already provides PMI-1 services, and *(2)* using the Hydra process manager that internally uses SLURM for process launching and management, but separately provides PMI services on top of it using an extra process daemon.

As shown in Figure 3, the impact of having additional PMI services (legend "Hydra") on top of the native process manager (legend "SLURM") on the system does not add any significant overhead. Figure 3(left) shows the impact on runtime, where there is no perceivable overhead. Figure 3(right) shows the percentage difference between the highest and lowest execution times noticed on a large number of runs of the application. Again, in most cases this difference is close to 0%, with a maximum of 0.5% for the EP application.

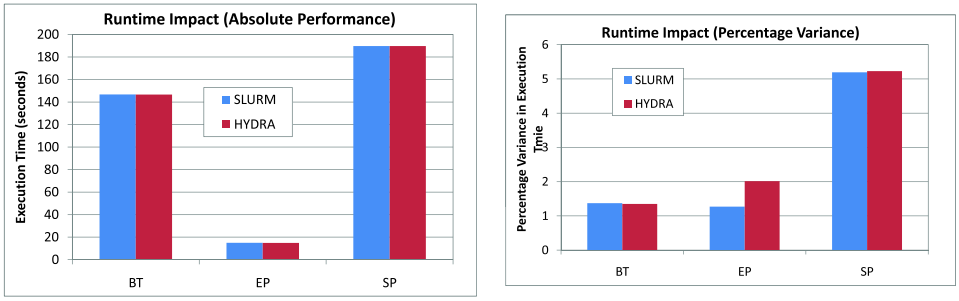

Fig. 3. Runtime impact of separate PMI server daemons: (left) absolute runtime; and (right) percentage variance in runtime.

The primary reason for such lack of overhead is that the PMI design completely relies on synchronous activity, and thus there is no asynchronous waking of PMI service daemons. That is, once the initialization is complete, unless the MPI process sends a PMI request, there is no additional overhead.

4.3 Performance of Multithreaded MPI Applications

With an increasing number of cores on each node, researchers are studying approaches for using MPI in conjunction with threads, in which case MPI functions might be called from multiple threads of a process. Since PMI-1 is not thread safe, all PMI calls must be protected by coarse-grained external locks; thus, only one thread can communicate with a process manager at a given time. PMI-2, on the other hand, is thread-safe, allowing for multiple threads to communicate with the process manager in a fine-grained manner.

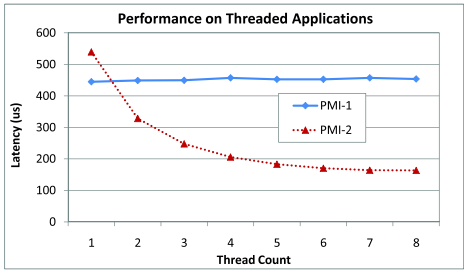

Fig. 4. Multithreading Performance

In this experiment, we measure the concurrency of PMI operations by using a benchmark that continuously publishes and unpublishes services to the process manager[4]. With PMI-1, each thread obtains a lock, sends a publish request and waits for a response from the process manager before releasing the lock. With PMI-2, each PMI request contains a thread ID; so the PMI library can send one request and release the lock (allowing other threads to send requests) even before it gets its response. When the process manager responds to the publish requests, it sends back the original thread ID with the response, allowing it to be forwarded to the appropriate thread.

The impact of such threading capability is illustrated in Figure 4. As shown in the figure, the average time taken by each PMI request for PMI-1 does not reduce with increasing number of threads since all requests are serialized (the

[4] These operations are used in `MPI_Publish_name` and `MPI_Unpublish_name`.

total amount of work is fixed, but shared between all the threads). With PMI-2, however, when multiple threads make concurrent PMI requests, all requests are pipelined in a fine-grained manner, allowing for better concurrency. Thus the average PMI latency perceived by each thread would be lesser. We notice that for the single-threaded case, PMI-2 has additional overhead compared with PMI-1. This result is unexpected and is being investigated.

4.4 Comparison with Alternative Process Management Frameworks

In this section, we compare an implementation of the PMI-2 interface with an alternate process-management framework, OpenRTE. OpenRTE [2] (ORTE) is the process-management system used in Open MPI. It is designed to provide robust and transparent support for parallel process management. Like PMI, it includes a system of (key,value) pairs that are exchanged between the MPI processes and the process-management system. Figure 5 compares launch time of MPI applications for the Hydra implementation of PMI-2 and OpenRTE. As shown in the figure, PMI-2 performs slightly better than ORTE[5] on this 796-core commodity cluster.

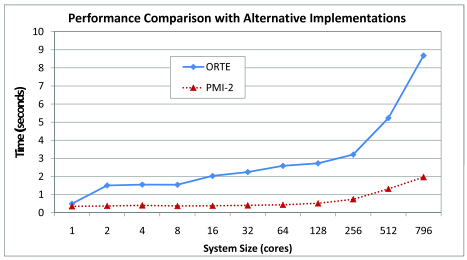

Fig. 5. Job launch time comparison with alternative process managers

5 Related Work

Improvements to the process-management framework for parallel programming models is not a new research topic. However, most efforts have focused on improving the process manager itself with respect to how it launches and manages processes. The OSC mpiexec [9], OSU mpirun (also known as SceLA) [13], and SLURM [15] are examples of such work. OSC mpiexec is a process manager for MPI applications that internally uses PBS [8] for launching and managing jobs. It is a centralized process manager that communicates using multiple process-management wire protocols, including PMI-1. OSU mpirun is based on SSH; it uses a hierarchical approach to launch processes and interacts with PMI-1. SLURM [15] differs from other process managers primarily in that it provides an entire infrastructure that launches and manages processes; it also provides its own PMI-1 implementation to interact with the processes. While all these implementations seek to improve process management on large-scale systems, our work differs in that none of these implementations study the requirements and limitations of the interface between the MPI library and the process manager, which is the PMI API and the wire protocol.

ORTE provides a mechanism for MPI processes to interact with the integrated process manager. However, it does not explicitly decouple these functionalities, as do PMI and its associated wire protocol.

[5] Open MPI v1.4.1 was used for the measurements presented here.

In summary, our work differs from other process-management systems with respect to its capabilities and underlying architecture. At the same time, PMI-2 provides a complementary contribution to those systems in that it can be used with them simultaneously.

6 Concluding Remarks

We presented a generic process-management interface, PMI, that allows different process-management frameworks to interact with parallel libraries such as MPI. We first described PMI-1, which is currently used in MPICH2 and all its derivatives. We then described PMI-2, the second generation of PMI that eliminates various shortcomings in PMI-1 on modern HPC systems, including scalability issues for large multi-core systems and interaction with hybrid MPI-and-threads models. Our performance results demonstrate significant advantages of PMI-2 compared with PMI-1.

References

1. Argonne National Laboratory: MPICH2. http://www.mcs.anl.gov/research/projects/mpich2
2. Castain, R., Woodall, T., Daniel, D., Squyres, J., Barrett, B., Fagg, G.: The Open Run-Time Environment (OpenRTE): A transparent multi-cluster environment for high-performance computing. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. pp. 225–232. Springer (2005)
3. Gara, A., Blumrich, M., Chen, D., Chiu, G., Coteus, P., Giampapa, M., Haring, R., Heidelberger, P., Hoenicke, D., Kopcsay, G., Liebsch, T., Ohmacht, M., SteinmacherBurow, B., Takken, T., Vranas, P.: Overview of the Blue Gene/L system architecture. IBM Journal of Research and Development 49(2/3) (2005)
4. Huang, W., Santhanaraman, G., Jin, H., Gao, Q., , Panda, D.: Design of high performance MVAPICH2: MPI2 over InfiniBand. In: Proceedings of the sixth IEEE International Symposium on Cluster Computing and the Grid. Singapore Management University, Singapore (May 16–19 2006)
5. Hydra process management framework. http://wiki.mcs.anl.gov/mpich2/index.php/Hydra_Process_Management_Framework
6. Intel MPI. http://software.intel.com/en-us/intel-mpi-library/
7. Microsoft MPI. http://msdn.microsoft.com/en-us/library/bb524831(VS.85).aspx
8. PBS: Portable batch system. http://www.openpbs.org
9. OSC Mpiexec. http://www.osc.edu/~djohnson/mpiexec
10. PMI-2 API. http://wiki.mcs.anl.gov/mpich2/index.php/PMI_v2_API
11. PMI-2 Wire Protocol. http://wiki.mcs.anl.gov/mpich2/index.php/PMI_v2_Wire_Protocol
12. SiCortex Inc. http://www.sicortex.com
13. Sridhar, J., Koop, M., Perkins, J., Panda, D.K.: ScELA: Scalable and Extensible Launching Architecture for Clusters. In: International Conference on High Performance Computing (HiPC) (2008)
14. Sun Grid Engine. http://www.sun.com/software/sge/
15. Yoo, A.B., Jette, M.A., Grondona, M.: SLURM: Simple Linux utility for resource management. In: Job Scheduling Strategies for Parallel Processing. Lecture Notes in Computer Science, vol. 2862. Springer (2003)