

Local Area Multicomputer (LAM -MPI)

Athanasios I. Margaris¹

¹ TEI of Larissa, Department of Computer Science and Telecommunications, Larissa, Thessaly, Greece

Correspondence: Athanasios I. Margaris, TEI of Larissa, Department of Computer Science and Telecommunications, Larissa, Thessaly, Greece. E-mail: amarg@uom.gr

Received: November 19, 2012 Accepted: February 4, 2013 Online Published: February 20, 2013

doi:10.5539/cis.v6n2p1

URL: <http://dx.doi.org/10.5539/cis.v6n2p1>

Abstract

The objective of this paper is the short description of the LAM (Local Area Multicomputer) implementation of MPI that can be used for the development of parallel applications based on the message passing interface. The paper describes the main aspects of the LAM environment such as the LAM architecture, configuration and use. A comparison between the LAM and the MPICH implementation (another very popular and commonly used MPI implementation) with respect to their performance is also presented.

Keywords: parallelization, implementation, performance, LAM-MPI, parallel architectures

1. Introduction

The message passing interface (Snir et al., 1995) is an efficient parallel programming model that allows the implementation of a parallel application as a set of processes communicating through properly defined messages. During the last years, many different MPI implementations have been developed - such as the MPICH (Gropp et al., 1999), the CHIMP (Alasdair, 1994), the IBM MPI (IBM, 2000), the Sun MPI (SUN, 2003), the HP MPI (HP, 2007) and the SGI MPI (SGI, 2003) - that allow the use of the model in different and heterogenous computing nodes. In this paper, we focus on another efficient MPI implementation identified by the name LAM-MPI (Squyres & Lumsdaine, 2003), that has been developed by the OpenSystems Laboratory of Indiana University. The current version of the platform is LAM 7.1.4 and it can be downloaded freely from the <http://www.lam-mpi.org> web site. The main feature of this implementation is the SSI system (System Services Interface). This system allows the determination of the basic building blocks used by the platform, at run time, namely, during the execution of the parallel application, that can be build on TCP/IP, InfiniteBand as well as on shared memory systems that use semaphores. LAM supports the implementation of point to point as well as collective operations, it provides the latest version of the ROMIO library for the implementation of high performance parallel I/O operations and it is compatible with the TotalView debugging utility.

2. LAM Architecture

The core of the LAM-MPI implementation is the SSI system defined as a two tiered modular framework capable of enabling multiple instances of interfaces, ensuring their availability at run time. This system allows the dynamic insertion and removal of programming modules (i.e. fundamental units that implement specific operations), in a plug-and-play fashion. This base system can be described as a set of “sockets” that accept the appropriate modules used in any case. In the current version of LAM-MPI there exist four different module types which, in short, are the following:

- **boot SSI modules:** the modules of this category allow the initialization of the LAM environment without the use of LAM daemons and they are used by the lamboot, recon, wipe and lamgrow commands.
- **coll SSI modules:** the modules of this category allow the implementation of collective communications between the processes of the parallel application. The modules lam_basic and smp belong to this category. These two modules allow the application programmers to design and implement collective algorithms without any knowledge of the LAM-MPI internal details.
- **cr SSI modules:** the modules of this type provide support for checkpoint/restart functionality and they are used by the LAM-MPI commands as well as the MPI processes. The most important module of this type is the blrc module.

- **rpi SSI modules:** these modules support the implementation of point to point communications and they are used by the MPI processes only. The modules gm, lamd, tcp, sysv and usysv are members of this family. These modules belong to the lowest layer of the LAM multilayered architecture and they are responsible for the data exchange between the processes of the parallel application.

In a more detailed description, the boot SSI modules are part of the LAM layer. This layer is composed of the LAM runtime environment (LRE) and the associated functions. This environment uses the so-called LAM daemons that operate on user level and provide services of any type such as the message passing operations, the control of the system processes and the remote file access. The startup and the shutdown of the LRE is performed by the commands lamboot and lamhalt. Regarding the MPI layer, it is characterized also by a multilayered structure, in which the upper layers are associated with the MPI functions, and the lower layers implement low level functions. This multilayered LAM architecture is shown in Figure 1.

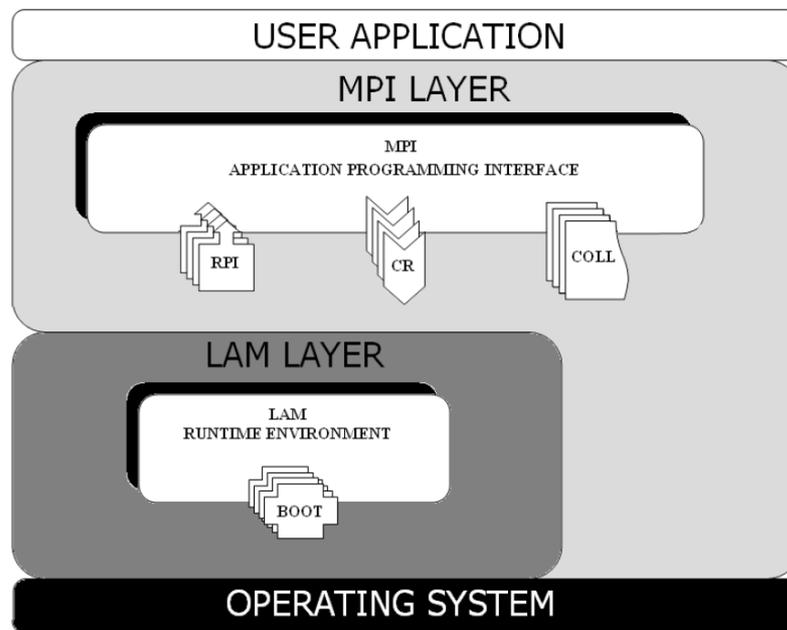


Figure 1. The multilayered structure of LAM-MPI architecture

2.1 Boot SSI Modules

The main functions of the boot SSI modules are the startup of the LAM-RTE and the handling of the parallel application processes in the local as well as in the remote computing nodes. This procedure can be resolved to a sequence of low level stages, such as the identification of the remote nodes, the process execution on these nodes and the passing of the startup protocol parameter to the new nodes registering to the LAM Universe. These stages are performed by the lamboot and lamgrow modules; the first module starts the LAM daemons to the network nodes, while the second module adds a new node to an existing LAM environment. Other modules that belong to the boot SSI module family, are the recon module associated with the control of the status of each computing node, the bproc module that initializes the system using the Bproc library (Beowulf Distributed Process Space) which allows the process startup in remote nodes of Beowulf systems, and the wipe module that terminates the LAM-RTE by issuing the tkill command.

Other interesting modules that belong to this class, is the rsh module which is responsible for the startup of the remote processes based on the rsh or the ssh utility, the globus module that allows the collaboration of heterogenous computing nodes by using the Globus system and a special fork job manager, as well as, the tm module associated with task management. The slurm (Simple Linux Utility for Resource Management) module is a special LAM module that can be used for the management of Linux clusters. Even though there are so many available boot modules, only one of them can be used during the boot process, and the selection of another boot module during the operation of the LAM-RTE is not possible. Regarding the boot process itself, it can be customized by using one of the following boot algorithms:

- **Linear:** this is the simplest boot algorithm and allows the sequential initialization of the processes of the parallel application. However a process starts, only after the completion of its predecessor.
- **Linear-Windowed:** this is a variation of the previous algorithm and supports the execution of the process regardless the status of the previous process. The control of the callback functions associated with the process startup is based on the use of a sliding window of length N. This algorithm is particularly useful if the protocol adopted use TCP and impose restrictions to the number of pending requests associated with a network socket.
- **Tree:** in this algorithm, a tree of auxiliary processes running in the internal tree nodes is constructed. These processes are provided by the LAM itself and not by the boot module and can initialize, in turn, new auxiliary as well as actual processes, thus allowing the generation of all the processes of the parallel application. The process creation in each tree node follows the linear approach.
- **Thread:** this algorithm is similar to the previous one, except the fact that the process initialization can be performed by a set of threads running concurrently.

The use of the modules of the boot family is supported by an API that contains a lot of data structures and functions such as the lamnode structure describing a computing node. The most important fields of that structure, are the node id, the node type (boot node, local node, or original node), the number of CPUs in that node, the TCP port number and the TCP address in binary format, the node host name, as well as a set of (key, value) pairs parsed from the boot schema file that provide an extensible method to obtain module-specific information from that file.

2.2 Coll SSI Modules

The modules of this family implement a set of algorithms used in collective operations in the LAM-RTE. The main task of those modules is the management of the operating environment of each communicator and the selection of the best algorithm for collective operations.

The startup and the initialization of each collective module is performed during the call of the `MPI_Init` MPI function. In this stage, all the available coll modules are probed and only the one that is to be used is being considered for further processing; all the other modules are ignored. In the next step, the pointers to the functions of this module are embedded to the handler of the MPI communicator in order to be used by the parallel application. During the termination stage, the module is shutdown by the `MPI_Finalize` function that calls internally the module shutdown routine.

LAM collective modules provide two different versions of each collective function, one for intracommunicators and one for intercommunicators. The main advantage of this approach is that, each communicator can use different collective modules, thus allowing the implementation of parallel applications in which the collective operations that are performed in different communicators, can use different algorithms. However, in such cases, the collective operations are not allowed to interfere with the possible point to point communications occurring in the same communicator. This restriction can be implemented by using different tags. In this approach, the tags with a positive value denote point to point operations, while, the tags with negative values are associated with collective communications.

The most important modules associated with the Coll family, are briefly described below: (a) the `lam_basic` module provides simple collective communication algorithms based to point to point operations. The algorithms offered by this module are linear or binomial according to the number of processes of the parallel application; (b) the `smp` module, supports the use of SMP nodes and is based to the MagPIe family of algorithms (Kielmann et al., 1999) associated with process locality; (c) the `shmem` module allows collective operations between processes running on the same node. The interprocess communication is performed in a shared memory fashion and requires that all processes must be able to attach the shared memory addresses to their own address space. In a more technical description, the shared memory is composed of two different and disjoint memory spaces, the control section that is responsible for the process synchronization and the message pool associated with the message passing operations. The size of the control section equals to $(2 \times N) + 2 \times C \times S$, where N is the number of the message pool segments, C is the cache line size and S is the number of the processes associated with the current communicator.

2.3 CR SSI Modules

The modules that belong to this class, support checkpoint/restart functionality and allow the storage of the process state at some checkpoints as well as the restart of that process in subsequent time instances on the same or different computing node. In most cases, the stored information includes, among other things, the contents of

the data segment, the stack and the registers for each process, as well as the file descriptors, the network sockets, the signal and timer values and the contents of the shared memory segments. All this information is stored in the secondary memory of the local computing node or a network server, while, the restore of all these values is performed by a set of auxiliary functions and temporary files of the operating system. This mechanism supports the development of fault tolerant applications and the efficient distribution and use of the system resources.

In LAM-MPI applications, the checkpoint/restart functionality can be used only for the mpirun command and the processes of the parallel applications, but not for the LAM runtime environment itself. To use this functionality, a checkpoint request is sent to mpirun command, which, in response, forwards this request to the parallel processes without interrupting their execution. When, at a later time, the user wants to restart the application from the checkpoint, the stored image of mpirun is restored in the memory, and a new instance of that command with an application scheme capable of restarting the parallel processes is issued. In the last step, the processes of the parallel application, in response to this event, restart themselves from their own checkpoint and the control is passed to the parallel application again.

The implementation of the mechanism that controls the checkpoint / restart functionality is supported by the CR SSI modules. These modules are composed of two components, a CRMPI type component that supports the checkpoint functionality and a CRLAM type component used by the mpirun command for the coordination of the procedure and the communication with the corresponding CRMPI unit of each process (this situation is shown in Figure 2). These two components of each CR module share the same name and they are used together as a single unit during the CR module selection stage. Regarding the operation of those units, it must support three different procedures identified by the names Checkpoint, Continue and Restart that describe the type of operation presented above.

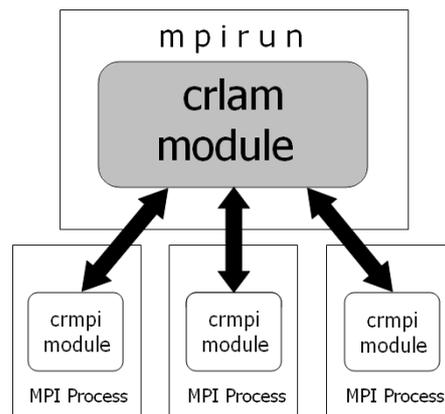


Figure 2. The communication of the CRLAM and CRMPI modules

The Checkpoint/Restart functionality can be used only if all the processes of the parallel application have issued the MPI_Init function. On the other hand, if at least one process has called the MPI_Finalize function, this type of functionality is no longer available. In order to maximize the time duration for this functionality, the MPI_Init function must be called as soon as possible, while, the MPI_Finalize function must be called after the MPI_Barrier function, so that, all the processes of the parallel application reach the same execution point.

The most important modules of the CR family are the blcr module that uses the blcr (Berkeley Lab's Checkpoint/Restart) system in Linux clusters and the self that supports checkpoint/restart functionality based on user defined functions.

2.4 RPI SSI Modules

The RPI (Request Progression Interface) module family of the LAM-MPI, allows the implementation of point to point communications and its main task is the receipt/forward of messages from/to the MPI/LAM level of the LAM architecture. The support of these operations is based on the use of request messages that allow the MPI layer to communicate with the system processes without needing to know anything about the message transfer details. Each of these requests is characterized by the so-called request life cycle that is composed of the following stages: (a) Building: in this stage, the memory space required for the communication is allocated and initialized according to the parameter values of the current message. (b) Starting: in this stage the current request

is placed to the system request list and its state is characterized as START, ACTIVE or DONE. (c) Progression: this stage is associated with the progressive processing of the request until its completion. During the data transfer initialization, the request moves from the START state to the ACTIVE state, while, after its completion and the receipt of the acknowledgment message, the request moves to the DONE state. (d) Completion: in this stage, the process of the request has been completed and the request object is initialized or destroyed, depending on whether it will be used or not.

The most important modules of the RPI family are the crtcp module that allows point to point communications with checkpoint/restart functionality support, the gm and the ib modules that are used in Myrinet and Infiniband networks respectively, and the lamd module that performs interprocess communication by means of lam daemons, thus allowing asynchronous message passing operations by using a lam daemon between the source and the target process. However, the bandwidth available is reduced, while latencies increase. The sysv module performs the message passing between processes running in the same node via shared memory architectures, and by using TCP sockets if the processes run on different machines. Regarding the process synchronization, it is based on the System V semaphores. A variation of this module supported by the usysv module, uses spin locks with back off. Finally, the tcp module uses TCP sockets for the interprocess communication together with the eager or the rendezvous communication protocol according to the size of the transferred messages.

3. LAM Configuration and Use

For security reasons, the LAM environment forbids its usage by the root user; therefore, it has to be used in the name of another user. Regarding the configuration of the LAM host machine it has to support the usage of the rsh or ssh remote services. The parallel application can be executed in a single host or in a set of hosts that can be accessed via a network connection. In the last case, the execution of the parallel application requires the following configuration options:

- The remote host has to be accessible through the network.
- The LAM user must exist in the remote machine.
- The LAM commands have to be accessible in the remote hosts.
- The LAM user must have write access rights to the LAM session directory.
- The system startup scripts must not print messages in the system console.
- The FQDN name of each host has to be resolvable by the remaining hosts.

The names of the computing nodes that are going to execute the processes of the parallel application are stored in an ASCII file, one name per line, and the distribution of the processes to them is performed in a round robin fashion. The LAM RTE startup process is performed by the lamboot command via a three step procedure: (a) The hboot command that allows the initialization of a process set based on a process scheme is executed in each computing node. (b) A communication port is allocated by each node and the lamboot command gets informed about this allocation. (c) The lamboot command broadcasts the collected (machine, port) pairs to all the computing nodes creating thus a fully functional network topology capable of executing parallel applications. If some node is not able to performe anyone of those steps in a predefined time interval, the process is aborted and the wipe command is called to terminate the process and inform the user with the appropriate error message.

The compilation and building of the parallel application can be performed by using a C compiler such as the gcc GNU compiler, while the execution of the application is based on the mpirun command that accepts the name of the executable and the number of processes to be created as arguments. The executable file must be copied to the same location in all computing nodes by using the FTP service or the NFS file system, but in general, there other alternatives available, such as to place the file in a shared network directory visible by all nodes. It is clear that if these nodes are heterogenous machines running different operating systems in different CPU architectures, the source code of the parallel application has to be compiled separately on them, so that the executable file to be compatible with the CPU architecture of each computing node.

4. LAM Commands

The LAM parallel programming environment allows the control of the parallel system via a set of user commands that can be called from the command line at each computing node. We briefly describe the most important of these commands:

- **mpitask:** it prints in the user screen the status of each mpi task such as the task id, the state of the task (running, paused, stop, blocked) as well as the size and the datatype of the current message.

- **mpimsg:** it reports the structure of the stored messages such as the ranks of the source and the target process, the message size, tag and datatype as well as the id of the communicator used for the message passing operation.
- **lamexec:** it allows the execution of non-MPI application in the LAM-RTE computing nodes.
- **lamclean:** it initializes the LAM environment by terminating all the active processes and running messages and freeing the allocated resources in all computing nodes.
- **lamgrow:** it adds a new computing node to an existing LAM-RTE environment.
- **lamshrink:** it removes a computing node from an existing LAM-RTE environment.
- **tping:** it checks the state of each computing node in a LAM-RTE environment by issuing the ping command for each one of them and counting the roundtrip time.
- **recon:** it is called before the lamboot command to check if the current network system is capable of supporting the execution of the LAM-RTE environment.
- **lamhalt:** it terminates the LAM-RTE environment in all computing nodes.
- **lamcheckpoint:** it defines a checkpoint for the current process.
- **lamrestart:** it issues the mpirun command to restart the parallel application from the checkpoint defined via the lamcheckpoint command.

5. LAM Performance

The LAM-MPI programming environment is characterized by very good performance and efficiency compared with the other MPI implementation. A good performance benchmark of the LAM in comparison with the MPICH platform can be found in (Nevin, 1996). In this test, the two platforms were evaluated under many different viewpoints such as the execution time of parallel applications for point to point and collective operations, barrier synchronization and ping requests. Another performance test between LAM, MPICH and MVICH (a VIA based MPI implementation) can be found in (Ong & Farrell, 2000).

Even though both LAM and MPICH use a short/long message protocol for communication, the adopted implementation is quite different. More specifically, in the LAM architecture, a short message is composed of a header and a data section and it is sent to the destination node as a single unit, while, a long message is divided into smaller packets with the first one of them to carry the message header. When this first packet is sent to the destination node, an acknowledgment is transmitted back to the sender in order to initiate the communication of the remaining data. On the other hand, the MPICH implementation supports three different protocols for data exchange: the eager protocol for short messages, and the rendezvous and the get protocol for long messages. In the eager protocol, the message data is sent to the destination node immediately, with the possibility of buffering data at the receiving node, when the node is not expecting the data. In the rendezvous protocol the data is sent to the destination node upon request, while in the get protocol, data is read directly by the receiver.

Since LAM and MPICH use TCP/UDP socket interfaces to communicate messages between nodes, there have been great efforts in reducing the overhead incurred in processing of TCP/IP stacks. An interesting and efficient approach on this direction, is the development of the Virtual Interface Architecture (Dunning, 1998) that defines mechanisms allowing the bypass of the layers of the protocol stack, avoiding thus the generation of the intermediate copies of data during sending and receiving messages. This bypass improves significantly the application performance, not only because eliminates this overhead, but because decreases the processor utilization by the communication subsystem. The only requirement of the VIA architecture is that the control and the setup to go through the kernel of the operating system. VIA supports send/rcv and remote direct memory access read/write types of data movements.

An interesting comparison test regarding the performance of LAM and MPICH can be found in (Ong & Farrell, 2000); in this work, the two implementations were compared by using three different Gigabit Ethernet NICs - namely, a Packet Engine GNIC-II, an Alteon ACEnic and a SysKconnect SK-NET NIC - in two Pentium III PCs running at 450 MHz with a 100 MHz memory bus and 256 MB of PC100 SD-RAM running RedHat Linux version 6.1. The switch-over point from the short to the long message format - this parameter can be configured at compile time - was 128 KB for both LAM and MPICH, while the used MTU value was the default value of the Ethernet standard, namely 1500 bytes. During the test, the maximum attainable throughput and the latency value for both LAM and MPICH were estimated, leading to comparable results (see the Figure 3). However, the two MPI implementations are quite different regarding their behavior with respect to the variation of the MTU parameter. Ong and Farrell found that an increase of the size of MTU, leads to the improvement of the LAM

performance but to the degradation of the performance of the MPICH implementation. This difference is explained by the fact that during initialization, LAM sets the send and receive socket buffers to a size equal to the switch-over point plus the size of the C2C (client to client) protocol envelope data structure (this protocol allows the bypass of the LAM daemon during data exchange operations). On the other hand, in MPICH the buffer size is fixed and equal to 4096 bytes; this is the reason for the fact that the increase of the MTU size does not improve the MPICH performance. Regarding the measured latency value, it is increased slightly in both cases with the increase of the MTU size, except in the case of the test of MPICH using the ACEnik card.

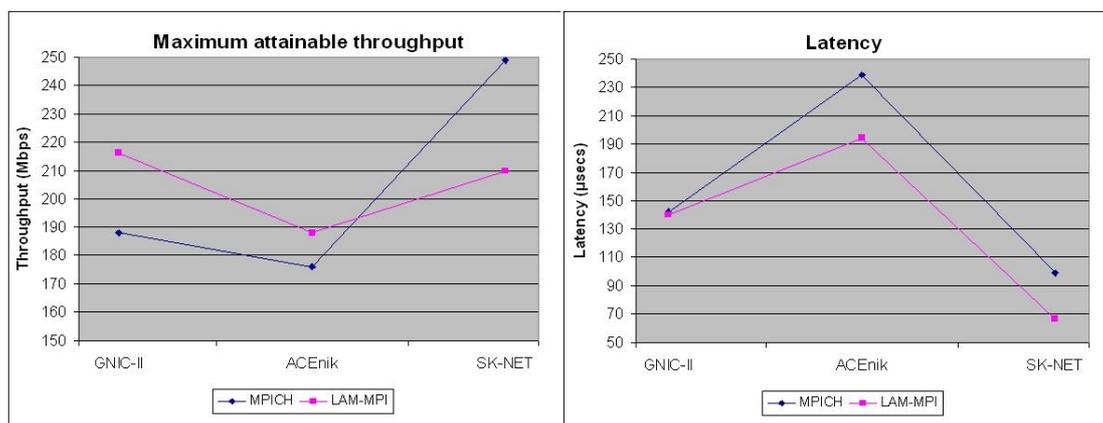


Figure 3. Experimental results for the maximum attainable throughput and the latency for LAM-MPI and MPICH

6. The Future of LAM-MPI

LAM-MPI served as a platform for the development of parallel applications for a long time period leading to new ideas and parallel programming techniques. This platform is no longer supported since 2004, as it has been replaced by the Open MPI project (Squyres, 2012), a more recent implementation that provides more sophisticated characteristics and a more advanced architecture. During the same time period many other MPI implementations as for example CHIMP, SUN MPI and HP MPI stopped to develop and gave their position to more recent efforts such as MVAPICH (Sur et al., 2011) and Intel MPI (Intel, 2012).

Open MPI is characterized by a layered structure similar to the one of LAM-MPI but with a lot of changes. More specifically, the two of the three layers of Figure 1, namely, the operating system layer and the MPI application layer remain the same, but the LAM layer has been substituted by a multi-layered architecture consisting of three layers each one of them is associated with its own programming library (the implementation is coded to the C language) and offers its own services. These layers from down to top are the following:

- Open, Portable Access Layer (OPAL) that is located after the operating system layer. It provides generic support such as linked list and string manipulation, and focuses on the individual processes. Furthermore, it ensures the portability of the Open MPI core between different operating systems, performing functions such as sharing memory between processes and discovering IP interfaces.
- Open MPI Run-Time Environment (ORTE) that provides services to the OMPI layer. Its main task is to launch the individual processes in the computing nodes via the rsh or ssh command, as well as to share computational resources between the users.
- Open MPI (OMPI) that serves as the highest abstraction layer and provides services to the parallel applications. The MPI API is implemented in this layer that supports a wide variety of network types and underlying protocols.

Even though each layer uses the services of its underlying layer, the ORLE and the OMPI layer can bypass the underlying layers and interact directly with the hardware and the operating system when needed. This feature increases the performance of the system, thus leading to higher execution speeds. The modular approach of the LAM-MPI also characterizes the structure of the Open MPI; the version 1.5 of Open MPI implementations is composed of 155 plugins that ensure the software abstraction and allow to third party developers to implement their own plugins outside the core of the Open MPI package. Detailed information about the Open MPI implementation can be found in the literature.

7. Conclusions

The objective of this review paper is the presentation of the LAM-MPI programming environment, an efficient MPI implementation that allows the development of parallel applications based on the message passing interface. The platform is characterized by a modular architecture, while its operation and use can be controlled by the user by means of a lot of tools and utilities. The platform supports the execution of the application in homogenous as well as heterogenous machines that run a variety of operating systems and the applications created are characterized by a good performance compared to applications provided by other platforms. The last years the LAM-MPI has been substituted by the Open MPI, a more sophisticated and advanced MPI implementation that leads to better performances and higher execution speeds.

References

- Alasdair, R. (1994). *CHIMP/MPI user guide*. Technical Report EPCC-KTP-CttIMP-V2-USER 1.2, Edinburgh Parallel Computing Centre.
- Dunning, D., Regnier, G., McAlpine, G., Cameron, D., Shubert, B., Berry, F., ... Dodd, C. (1998). The Virtual Interface Architecture. *IEEE Micro*, 18(2), 66-76. <http://dx.doi.org/10.1109/40.671404>
- Gropp, W., Lusk, E., & Skjellum, A. (1999). *Using MPI: Portable Parallel Programming with the Message Passing Interface* (2nd ed.). The MIT Press.
- HP Corporation. (2007). *HP-MPI User's Guide, HP Part Number*. Retrieved from http://tier3-atlas1.bellarmine.edu/kits/platform-mpi/7.1/1/guides/MPI_User_Guide.pdf
- IBM Corporation. (2000). *IBM Parallel Environment for AIX IBM MPI Programming Guide*. Retrieved from <http://www.serc.iisc.ernet.in/facilities/ComputingFacilities/systems/cluster/poe-3.0/am106004.pdf>
- Intel Corporation. (2012). *Intel MPI Library Release 4.1, Software Documentation*. Retrieved from <http://software.intel.com/en-us/intel-mpi-library>
- Kielmann, T., Hofman, R. F. H., Bal, H. E., Plaat, A., & Bhoedjang, R. A. F. (1999). MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *Proceedings of the 1999 ACM SIGPLAN Symposium on Principled and Practice of Parallel Programming*. pp. 131-140.
- Nevin, N. (1996). *The Performance of LAM 6.0 and MPICH 1.0.12 on a Workstation Cluster*. Ohio Supercomputing Center, Technical Report OSC-TR-1996-4.
- Ong, H., & Farrell, P. (2000). A Performance Comparison of LAM/MPI, MPICH, and MVICH on a Linux Cluster connected by a Gigabit Ethernet Network. *Proceedings of the 4th Annual Linux Showcase and Conference*.
- SGI Corporation. (2003). *SGI MPI Toolkit Programming Manual*. Retrieved from <http://techpubs.sgi.com/library/manuals/3000/007-3687-010/pdf/007-3687-010.pdf>
- Snir, M. (1995). *MPI: The Complete Reference*. The MIT Press.
- Squyres, J., & Lumsdaine, A. (2003). A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting, number 2840 in Lecture Notes in Computer Science*.
- Squyres, J. (2012). *The Architecture of Open Source Applications*. Vol. 2, Chapter 15.
- SUN Corporation. (2003). *SUN MPI 6.0 Programming and Reference Manual, Sun Microsystems*. Retrieved from http://telematica.cicese.mx/computo/super/calafia/manuales/hpc5doc/MPI_Reference.pdf
- Sur, S., Potluri, S., Kandalla, K., Subramoni, H., Tomko, K., & Panda, D. K. (2011). Co-Designing MPI Library and Applications for InfiniBand Clusters. *IEEE Computer*, 11, 31-36. <http://dx.doi.org/10.1109/MC.2011.265>