Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation

Edgar Gabriel[1], Graham E. Fagg[1], George Bosilca[1], Thara Angskun[1], Jack J. Dongarra[1], Jeffrey M. Squyres[2], Vishal Sahay[2], Prabhanjan Kambadur[2], Brian Barrett[2], Andrew Lumsdaine[2], Ralph H. Castain[3], David J. Daniel[3], Richard L. Graham[3], Timothy S. Woodall[3]

[1] Innovative Computing Laboratory, University of Tennessee,
{egabriel, fagg, bosilca, anskun, dongarra}@cs.utk.edu

[2] Open System Laboratory, Indiana University
{jsquyres, vsahay, pkambadu, brbarret, lums}@osl.iu.edu

[3] Advanced Computing Laboratory, Los Alamos National Lab
{rhc, ddd, rlgraham,twoodall}@lanl.gov

Abstract. A large number of MPI implementations are currently available, each of which emphasize different aspects of high-performance computing or are intended to solve a specific research problem. The result is a myriad of incompatible MPI implementations, all of which require separate installation, and the combination of which present significant logistical challenges for end users. Building upon prior research, and influenced by experience gained from the code bases of the LAM/MPI, LA-MPI, and FT-MPI projects, *Open MPI* is an all-new, production-quality MPI-2 implementation that is fundamentally centered around component concepts. Open MPI provides a unique combination of novel features previously unavailable in an open-source, production-quality implementation of MPI. Its component architecture provides both a stable platform for third-party research as well as enabling the run-time composition of independent software add-ons. This paper presents a high-level overview the goals, design, and implementation of Open MPI.

1 Introduction

The evolution of parallel computer architectures has recently created new trends and challenges for both parallel application developers and end users. Systems comprised of tens of thousands of processors are available today; hundred-thousand processor systems are expected within the next few years. Monolithic high-performance computers are steadily being replaced by clusters of PCs and workstations because of their more attractive price/performance ratio. However, such clusters provide a less integrated environment and therefore have different (and often inferior) I/O behavior than the previous architectures. Grid and metacomputing efforts yield a further increase in the number of processors available to parallel applications, as well as an increase in the physical distances between computational elements.

These trends lead to new challenges for MPI implementations. An MPI application utilizing thousands of processors faces many scalability issues that can dramatically impact the overall performance of any parallel application. Such issues include (but are not limited to): process control, resource exhaustion, latency awareness and management, fault tolerance, and optimized collective operations for common communication patterns.

Network layer transmission errors—which have been considered highly improbable for moderate-sized clusters—cannot be ignored when dealing with large-scale computations [4]. Additionally, the probability that a parallel application will encounter a process failure during its run increases with the number of processors that it uses. If the application is to survive a process failure without having to restart from the beginning, it either must regularly write checkpoint files (and restart the application from the last consistent checkpoint [1, 8]) or the application itself must be able to adaptively handle process failures during runtime [3]. All of these issues are current, relevant research topics. Indeed, some have been addressed at various levels by different projects. However, no MPI implementation is currently capable of addressing all of them comprehensively.

This directly implies that a new MPI implementation is necessary: one that is capable of providing a framework to address important issues in emerging networks and architectures. Building upon prior research, and influenced by experience gained from the code bases of the LAM/MPI [9], LA-MPI [4], and FT-MPI [3] projects, *Open MPI* is an all-new, production-quality MPI-2 implementation. Open MPI provides a unique combination of novel features previously unavailable in an open-source, production-quality implementation of MPI. Its component architecture provides both a stable platform for cutting-edge third-party research as well as enabling the run-time composition of independent software add-ons.

1.1 Goals of Open MPI

While all participating institutions have significant experience in implementing MPI, Open MPI represents more than a simple merger of LAM/MPI, LA-MPI and FT-MPI. Although influenced by previous code bases, Open MPI is an all-new implementation of the Message Passing Interface. Focusing on production-quality performance, the software implements the full MPI-1.2 [6] and MPI-2 [7] specifications and fully supports concurrent, multi-threaded applications (i.e., MPI_THREAD_MULTIPLE).

To efficiently support a wide range of parallel machines, high performance "drivers" for all established interconnects are currently being developed. These include TCP/IP, shared memory, Myrinet, Quadrics, and Infiniband. Support for more devices will likely be added based on user, market, and research requirements. For network transmission errors, Open MPI provides optional features for checking data integrity. By utilizing message fragmentation and striping

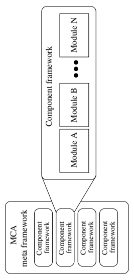

Fig. 1. Three main functional areas of Open MPI: the MCA, its component frameworks, and the modules in each framework.

over multiple (potentially heterogeneous) network devices, Open MPI is capable of both maximizing the achievable bandwidth to applications and providing the ability to dynamically handle the loss of network devices when nodes are equipped with multiple network interfaces. Thus, the handling of network failovers is completely transparent to the application.

The runtime environment of Open MPI will provide basic services to start and manage parallel applications in interactive and non-interactive environments. Where possible, existing run-time environments will be leveraged to provide the necessary services; a portable run-time environment based on user-level daemons will be used where such services are not already available.

2 The Architecture of Open MPI

The Open MPI design is centered around the MPI Component Architecture (MCA). While component programming is widely used in industry, it is only recently gaining acceptance in the high performance computing community [2, 9]. As shown in Fig. 1, Open MPI is comprised of three main functional areas:

– MCA: The backbone component architecture that provides management services for all other layers;
– Component frameworks: Each major functional area in Open MPI has a corresponding back-end component framework, which manages modules;
– Modules: Self-contained software units that export well-defined interfaces that can be deployed and composed with other modules at run-time.

The MCA manages the component frameworks and provides services to them, such as the ability to accept run-time parameters from higher-level abstractions (e.g., `mpirun`) and pass them down through the component framework to individual modules. The MCA also finds components at build-time and invokes their corresponding hooks for configuration, building, and installation.

Each component framework is dedicated to a single task, such as providing parallel job control or performing MPI collective operations. Upon demand, a framework will discover, load, use, and unload modules. Each framework has different policies and usage scenarios; some will only use one module at a time while others will use all available modules simultaneously.

Modules are self-contained software units that can configure, build, and install themselves. Modules adhere to the interface prescribed by the component framework that they belong to, and provide requested services to higher-level tiers and other parts of MPI.

The following is a partial list of component frameworks in Open MPI (MPI functionality is described; run-time environment support components are not covered in this paper):

- Point-to-point Transport Layer (PTL): a PTL module corresponds to a particular network protocol and device. Mainly responsible for the "wire protocols" of moving bytes between MPI processes, PTL modules have no knowledge of MPI semantics. Multiple PTL modules can be used in a single process, allowing the use of multiple (potentially heterogeneous) networks. PTL modules supporting TCP/IP, shared memory, Quadrics elan4, Infiniband and Myrinet will be available in the first Open MPI release.
- Point-to-point Management Layer (PML): the primary function of the PML is to provide message fragmentation, scheduling, and re-assembly service between the MPI layer and all available PTL modules. More details to the PML and the PTL modules can be found at [11].
- Collective Communication (COLL): the back-end of MPI collective operations, supporting both intra- and intercommunicator functionality. Two collective modules are planned at the current stage: a basic module implementing linear and logarithmic algorithms and a module using hierarchical algorithms similar to the ones used in the MagPIe project [5].
- Process Topology (TOPO): Cartesian and graph mapping functionality for intracommunicators. Cluster-based and Grid-based computing may benefit from topology-aware communicators, allowing the MPI to optimize communications based on locality.
- Reduction Operations: the back-end functions for MPI's intrinsic reduction operations (e.g., MPI_SUM). Modules can exploit specialized instruction sets for optimized performance on target platforms.
- Parallel I/O: I/O modules implement parallel file and device access. Many MPI implementations use ROMIO [10], but other packages may be adapted for native use (e.g., cluster- and parallel-based filesystems).

The wide variety of framework types allows third party developers to use Open MPI as a research platform, a deployment vehicle for commercial products, or even a comparison mechanism for different algorithms and techniques.

The component architecture in Open MPI offers several advantages for end-users and library developers. First, it enables the usage of multiple components within a single MPI process. For example, a process can use several network device drivers (PTL modules) simultaneously. Second, it provides a convenient possibility to use third party software, supporting both source code and binary distributions. Third, it provides a fine-grained, run-time, user-controlled component selection mechanism.

2.1 Module Lifecycle

Although every framework is different, the COLL framework provides an illustrative example of the usage and lifecycle of a module in an MPI process:

1. During MPI_INIT, the COLL framework finds all available modules. Modules may have been statically linked into the MPI library or be shared library modules located in well-known locations.
2. All COLL modules are queried to see if they want to run in the process. Modules may choose not to run; for example, an Infiniband-based module may choose not to run if there are no Infiniband NICs available. A list is made of all modules who choose to run – the list of "available" modules.
3. As each communicator is created (including MPI_COMM_WORLD and MPI_COMM_SELF), each available module is queried to see if wants to be used on the new communicator. Modules may decline to be used; e.g., a shared memory module will only allow itself to be used if all processes in the communicator are on the same physical node. The highest priority module that accepted is selected to be used for that communicator.
4. Once a module has been selected, it is initialized. The module typically allocates any resources and potentially pre-computes information that will be used when collective operations are invoked.
5. When an MPI collective function is invoked on that communicator, the module's corresponding back-end function is invoked to perform the operation.
6. The final phase in the COLL module's lifecycle occurs when that communicator is destroyed. This typically entails freeing resources and any pre-computed information associated with the communicator being destroyed.

3 Implementation details

Two aspects of Open MPI's design are discussed: its object-oriented approach and the mechanisms for module management.

3.1 Object Oriented Approach

Open MPI is implemented using a simple C-language object-oriented system with single inheritance and reference counting-based memory management using a retain/release model. An "object" consists of a structure and a singly-instantiated "class" descriptor. The first element of the structure must be a pointer to the parent class's structure.

Macros are used to effect C++-like semantics (e.g., new, construct, destruct, delete). The experience with various software projects based on C++ and the according compilation problems on some platforms has encouraged us to take this approach instead of using C++ directly.

Upon construction, an object's reference count is set to one. When the object is retained, its reference count is incremented; when it is released, its reference count is decreased. When the reference count reaches zero, the class's destructor (and its parents' destructor) is run and the memory is freed.

3.2 Module Discovery and Management

Open MPI offers three different mechanisms for adding a module to the MPI library (and therefore to user applications):

- During the configuration of Open MPI, a script traverses the build tree and generates a list of modules found. These modules will be configured, compiled, and linked statically into the MPI library.
- Similarly, modules discovered during configuration can also be compiled as shared libraries that are installed and then re-discovered at run-time.
- Third party library developers who do not want to provide the source code of their modules can configure and compile their modules independently of Open MPI and distribute the resulting shared library in binary form. Users can install this module into the appropriate directory where Open MPI can discover it at run-time.

At run-time, Open MPI first "discovers" all modules that were statically linked into the MPI library. It then searches several directories (e.g., `$HOME/ompi/`, `${INSTALLDIR}/lib/ompi/`, etc.) to find available modules, and sorts them by framework type. To simplify run-time discovery, shared library modules have a specific file naming scheme indicating both their MCA component framework type and their module name.

Modules are identified by their name and version number. This enables the MCA to manage different versions of the same component, ensuring that the modules used in one MPI process are the same—both in name and version number–as the modules used in a peer MPI process. Given this flexibility, Open MPI provides multiple mechanisms both to choose a given module and to pass run-time parameters to modules: command line arguments to `mpirun`, environment variables, text files, and MPI attributes (e.g., on communicators).

4 Performance Results

A performance comparison of Open MPI's point-to-point methodology to other, public MPI libraries can be found in [11]. As a sample of Open MPI's performance in this paper, a snapshot of the development code was used to run the Pallas benchmarks (v2.2.1) for MPI_Bcast and MPI_Alltoall. The algorithms used for these functions in Open MPI's basic COLL module were derived from their corresponding implementations in LAM/MPI v6.5.9, a monolithic MPI implementation (i.e., not based on components). The collective operations are based on standard linear/logarithmic algorithms using MPI's point-to-point message passing for data movement. Although Open MPI's code is not yet complete, measuring its performance against the same algorithms in monolithic architecture provides a basic comparison to ensure that the design and implementation are sound.

The performance measurements were executed on a cluster of 2.4 GHz dual processor Intel Xeon machines connected via fast Ethernet. The results shown in Fig. 2 indicate that the performance of the collective operations using the Open MPI approach is identical for large message sizes to its LAM/MPI counterpart. For short messages, there is currently a slight overhead for Open MPI compared to LAM/MPI. This is due to point-to-point latency optimizations in LAM/MPI not yet included in Open MPI; these optimizations will be included in the release of Open MPI. The graph shows, however, that the design and overall approach is sound, and simply needs optimization.

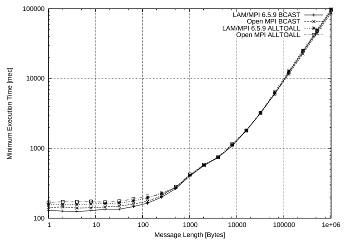

Fig. 2. Performance comparison for MPI_BCAST and MPI_ALLTOALL operations in Open MPI and in LAM/MPI v6.5.9.

5 Summary

Open MPI is a new implementation of the MPI standard. It provides functionality that has not previously been available in any single, production-quality MPI implementation, including support for all of MPI-2, multiple concurrent user threads, and multiple options for handling process and network failures. The Open MPI group is furthermore working on establishing a proper legal framework, which enbales third party developers to contribute source code to the project.

The first full release of Open MPI is planned for the 2004 Supercomputing Conference. An initial beta release supporting most of the described functionality and an initial subset of network device drivers (tcp, shmem, and a loopback device) is planned for release mid-2004. http://www.open-mpi.org/

Acknowledgments

This work was supported by a grant from the Lilly Endowment, National Science Foundation grants 0116050, EIA-0202048, EIA-9972889, and ANI-0330620, and Department of Energy Contract DE-FG02-02ER25536. Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36. Project support was provided through ASCI/PSE and the Los Alamos Computer Science Institute, and the Center for Information Technology Research (CITR) of the University of Tennessee.

References

1. G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *SC'2002 Conference CD*, Baltimore, MD, 2002. IEEE/ACM SIGARCH. pap298,LRI.
2. D. E. Bernholdt et. all. A component architecture for high-performance scientific computing. *Intl. J. High-Performance Computing Applications*, 2004.
3. G. E. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, A. Bukovski, and J. J. Dongarra. Fault tolerant communication library and applications for high perofrmance. In *Los Alamos Computer Science Institute Symposium*, Santa Fee, NM, October 27-29 2003.
4. R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. D. Risinger, and M. W. Sukalksi. A network-failure-tolerant message-passing system for terascale clusters. *International Journal of Parallel Programming*, 31(4):285–303, August 2003.
5. T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, 34(8):131–140, May 1999.
6. Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, June 1995. http://www.mpi-forum.org.

7. Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface*, July 1997. http://www.mpi-forum.org.
8. Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, To appear, 2004.
9. Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, Venice, Italy, Sept. 2003. Springer.
10. 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, Feb 1999.
11. T.S. Woodall, R.L. Graham, R.H. Castain, D.J. Daniel, M.W. Sukalski, G.E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine. TEG: A high-performance, scalable, multi-network point-to-point communications methodology. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004.