Open MPI: A Flexible High Performance MPI

Richard L. Graham[1], Timothy S. Woodall[1], and Jeffrey M. Squyres[2]

[1] Advanced Computing Laboratory, Los Alamos National Lab
{rlgraham,twoodall}@lanl.gov
[2] Open System Laboratory, Indiana University
jsquyres@osl.iu.edu

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, FT-MPI, and PACX-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, as well as performance results for it's point-to-point implementation.

1 Introduction

The face of high-performance computer systems landscape is changing rapidly, with systems comprised of thousands to hundreds of thousands of processors in use today. These systems vary from tightly integrated high end systems, to clusters of PCs and workstations. Grid and meta computing add twists such as a changing computing environment, computing across authentication domains, and non-uniform computing facilities, such as variations in processor type and bandwidths and latencies between processors.

This wide variety of platforms and environments poses many challenges for a production-grade, high performance, general purpose MPI implementation, requires it to provide a high degree of flexibility in many problem axes. One needs to provide tunable support for the traditional high performance, scalable communications algorithms, as well as address a variety of failure scenarios. In addition items such as process control, resource exhaustion, latency awareness and management, fault tolerance, and optimized collective operations for common communication patterns, need to be dealt with.

These types of issues have addressed in one way of another by different projects, but little attention has been given to dealing with various fault scenarios. In particular, network layer transmission errors—which have been considered highly improbable for moderate-sized clusters—cannot be ignored when dealing with large-scale computations [4]. This is particularly true when O/S bypass protocols are used for high performance messaging on systems that do not have end-to-end hardware data integrity. In addition, the probability that a parallel application will encounter a process failure during its run increases with the size of they system used. For an application to survive process failure it either must regularly write checkpoint files (and restart the application from the last consistent checkpoint [1, 10]) or the application itself must be able to adaptively handle process failures during runtime [3] and use an MPI implementation that deals with process failure. These issues are current, relevant research topics. While some have been addressed at various levels by different research efforts, no single MPI implementation is currently capable of addressing all these in a comprehensive manner.

Therefore, a new MPI implementation is required: one that is capable of providing a framework to address important issues in emerging networks and architectures. The Open MPI project was initiated with the express intent of addressing these, and other issues. Building upon prior research, and influenced by experience gained from the code bases of the LAM/MPI [13], LA-MPI [4], FT-MPI [3], and the PACX-MPI [5] project, *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 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 the Open MPI Project

While all participating organizations have significant experience in implementing MPI, Open MPI represents more than a simple merger of the LAM/MPI, LA-MPI, FT-MPI, and PACX-MPI code bases. While influenced by previous implementation experiences, Open MPI uses a new software design to implement of the Message Passing Interface. Focusing on production-quality performance, the software implements the MPI-1.2 [7] and MPI-2 [8] specifications and supports concurrent, multi-threaded applications (i.e., MPI_THREAD_MULTIPLE).

To efficiently support a wide range of parallel machines, high performance "drivers" for established communications protocols have been developed. These include TCP/IP, shared memory, Myrinet (GM and MX), and Infiniband (MVAPI and OpenIB). Support for more devices will likely be added based on user, market, and research requirements. For network transmission errors, ideas first explored in LA-MPI are being extended with optional support is being developed for checking data integrity. In addition, by utilizing message fragmentation and striping over multiple (potentially heterogeneous) network devices, Open MPI is capable of both maximizing the achievable bandwidth to applications and is developing the ability to dynamically handle the loss of network devices when nodes are equipped with multiple network interfaces. Handling of these network failovers is completely transparent to MPI applications.

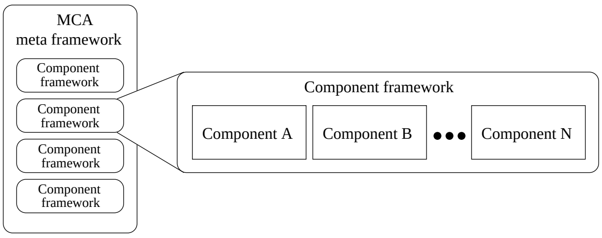

Fig. 1. Three main functional areas of Open MPI: the MCA, its component frameworks, and the components in each framework.

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

2 The Architecture of Open MPI

Open MPI's primary software design motif is a component architecture called the Modular Component Architecture (MCA). The use of components forces the design of well contained library routines and makes extending the implementation convenient. While component programming is widely used, it is only recently gaining acceptance in the high performance computing community [2, 13]. 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;
- Components: Self-contained software units that export well-defined interfaces and can be deployed and composed with other components.

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 components. 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 components. Each framework has different policies and usage scenarios; some will only use one component at a time while others will use all available components simultaneously.

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

The Open MPI software has three classes of components: Open MPI components, Open Run Time Environment (ORTE) components, and Open Portable Access Layer (OPAL) components.

The following is a partial list of MPI component frameworks in Open MPI (only MPI functionality is described; ORTE and OPAL frameworks and components are not covered in this paper):

– Point-to-point Management Layer (PML): this component manages all full message delivery. It implements the the semantics of a given point-to-point communications protocol, such as MPI.
– Byte-Transfer-Layer Layer (BTL): this component is handles point-to-point data delivery over the network, and is unaware of upper-level point-to-point communications protocols, such as MPI.
– BTL Management Layer (BML): this component provides services during job startup and dynamic process creation to discover and maintain the set of BTLs that may be used for point-to-point communications between a given pair of end-points.
– Collective Communication (COLL): the back-end of MPI collective operations, supporting both intra- and intercommunicator functionality.
– 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.
– Parallel I/O: I/O modules implement parallel file and device access. Many MPI implementations use ROMIO [14], 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 networks simultaneously. Second, it provides a convenient possibility to use third party software, supporting both source code and binary distributions of components. Third, it provides a fine-grained, run-time, user-controlled component selection mechanism.

2.1 Module Lifecycle

The Byte Transfer Layer (BTL) framework provides a good illustrative example of the complete usage and lifecycle of a module in an MPI process:

1. During MPI_INIT, the BTL Management Layer (BML) framework (described below) discovers all available BTL components. Components may have been statically linked into the MPI library or can be loaded from shared libraries located in well-known locations.

2. Each BTL component is queried to see if its want to run in the process. Components may choose not to run; for example, an Infiniband-based component may choose not to run if there are no Infiniband NICs available.
3. Components that are selected and successfully initialized will return a set of BTL modules to the BML, each representing distinct network interfaces or ports. Each module may cache resources and addressing information required for communication on the underlying transport.
4. The BML queries each of the BTL modules to determine the set of processes with which they are able to communicate. For each peer, the BML maintains a list of BTLs through which that peer is reachable. These tables are exposed to the upper layers for efficient message delivery and striping.
5. BTL modules exist for the duration of the process. During MPI_FINALIZE, each BTL module is given the opportunity to cleanup any allocated resources prior to closing its corresponding component.

3 Implementation details

Several aspects of Open MPI's design are discussed in this section.

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' structure.

Macros are used to effect C++-like semantics (e.g., new, construct, destruct, delete). 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; destructor (and its parents' destructor) is run and the memory is freed.

The experience with prior software projects based on C++ and the according compatibility and compilation problems on some platforms has encouraged us to take this approach instead of using C++ directly. For example, C++ compilers may layout the same class or structure differently in memory. This can lead to problems when serializing data and sending it across a network if the sender was compiled with a different compiler than the receiver.

3.2 Component Discovery and Management

Open MPI offers three different mechanisms for adding a component 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 components found. These components will be configured, compiled, and linked statically into the MPI library.
- Similarly, components 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 components can configure and compile their components independently of Open MPI and distribute the resulting shared library in binary form. Users can install this component into the appropriate directory where Open MPI can discover it at run-time.

At run-time, Open MPI first "discovers" all components that were statically linked into the MPI library. It then searches several directories to find available components and sorts them by framework type.

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

3.3 Point-To-Point Components

The Open MPI point-to-point (p2p) design and implementation is based on multiple MCA frameworks. These frameworks provide functional isolation with clearly defined interfaces. Fig. 2 illustrates the p2p framework architecture.

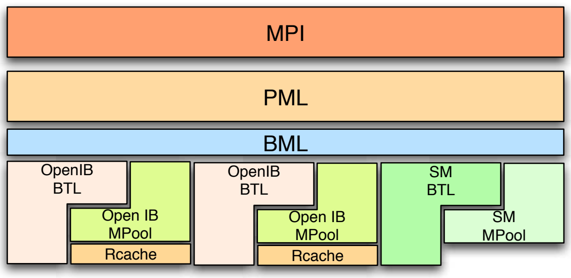

Fig. 2. Open MPI p2p framework

As shown in Fig. 2 the architecture consists of four layers. Working from the bottom up these layers are the Byte Transfer Layer (BTL), BTL Management Layer (BML), Point-to-Point Messaging Layer (PML) and the MPI layer. Each of these layers is implemented as an MCA framework. Other MCA frameworks shown are the Memory Pool (MPool) and the Registration Cache (RCache). While these frameworks are illustrated and defined as layers, performance-critical send/receive paths bypass the BML, as it is used primarily during initialization and BTL component selection.

MPool The memory pool provides memory allocation/deallocation and registration/deregistration services. OS-bypass networks such as Infiniband and Myrinet require memory to be registered (physical pages present and pinned) before send/receive or RDMA operations can use the memory as a source or target. Separating this functionality from other components allows the MPool to be shared among various layers. For example, MPI_ALLOC_MEM uses these MPools to register memory with available interconnects.

RCache The registration cache allows memory pools to cache registered memory for later operations. When initialized, MPI message buffers are registered with the MPool and cached via the RCache. For example, during an MPI_SEND the source buffer is registered with the memory pool and this registration may be then be cached, depending on the protocol in use. During subsequent MPI_SEND operations the source buffer is checked against the RCache, and if the registration exists the PML may RDMA the entire buffer in a single operation without incurring the high cost of registration.

BTL The BTLs expose a set of communication primitives appropriate for both send/receive and RDMA interfaces. The BTL is not aware of any MPI semantics; it simply moves a sequence of bytes (potentially non-contiguous) across the underlying transport. This simplicity will enable early adoption of novel network devices and encourages vendor support.

BML The BML acts as a thin multiplexing layer allowing the BTLs to be shared among multiple upper layers. Discovery of peer resources is coordinated by the BML and cached for multiple consumers of the BTLs. After resource discovery, the BML layer is bypassed by upper layers for performance.

PML The PML implements all logic for p2p MPI semantics including standard, buffered, ready, and synchronous communication modes. MPI message transfers are scheduled by the PML based on a specific policy. This policy incorporates BTL specific attributes to schedule MPI messages. Short and long message protocols are implemented within the PML. All control messages (ACK/NACK/MATCH) are also managed at the PML. The benefit of this structure is a separation of transport protocol from the underlying interconnects. This significantly reduces both code complexity and code redundancy enhancing maintainability.

Althought there are currently three PML components available in the Open MPI, this paper only discusses the `OB1` PML component. `OB1` is Open MPI's latest generation PML, reflecting the most recent communications research. There is currently only one BML component – "`R2`." Finally, there are several BTL modules available, providing support for the following networks: TCP, shared memory, Portals, Myrinet/MX, Myrinet/GM, Mellanox VAPI, and OpenIB VAPI.

These components are used as follows: during startup, a PML component is selected and initialized. The PML component selected defaults to `OB1` but may be overridden by a run-time parameter. Next the BML component `R2` is selected (since there is only one available). `R2` then opens and initializes all available BTL modules. During BTL module initialization, `R2` directs peer resource discovery on a per-BTL component basis. This allows the peers to negotiate which set of interfaces they will use to communicate with each other for MPI communications. This infrastructure allows for heterogeneous networking interconnects within a cluster.

Table 1. Hardware and software setup used to generate results data.

Interconnect	CPU	RAM	Bus	Kernel	Stack
Infiniband	(2) Intel Xeon 3.6 GHZ	6GB	PCIe	2.6.12	IB Gold 1.0
Myrinet	(2) Intel Xeon 2.8 GHz	2GB	PCI-X	2.6.11	GM 2.0.22
TCP/SM	(2) AMD Opteron 2 GHz	8GB	PCI-X	2.6.9	

Table 2. Latency of zero byte ping-pong messages.

Implementation	Latency
Open MPI GM	6.86μs
MPICH-GM	7.10μs
Open MPI Shared Memory	1.23μs
MPICH2 Shared Memory	2.21μs
Open MPI TCP	32.0μs
MPICH2 TCP	29.0μs
Open MPI OpenIB	5.13μs
Open MPI MVAPI	5.64μs
MVAPICH MVAPI (RDMA)	4.19μs
MVAPICH MVAPI (Send/Recv)	6.51μs

4 Performance Results

The data obtained in this section was using Open MPI version 1.0.1rc6 using the OB1 PML with the MVAPI, GM, Shared Memory, and TCP BTL components. These runs are compared with data from three other MPI implementations on the same hardware. GM data was obtained with MPICH-GM version 1.2.6; the shared memory and TCP data with MPICH2 version 1.0.3 [9]; the Infiniband data with MVAPICH version 0.9.5 [6]. Latency is measured as half the round trip time of zero byte MPI messages, using a ping-pong benchmark code, and the bandwidth data is measured using NetPIPE version 3.6.2 [12]. The systems used to make these measurements are described in the Table 1.

4.1 Point-To-Point Latency

Ping-pong latencies are listed in Table 2. As this table shows, Open MPI has extremely competitive latencies. The latency of 6.86 μsec using GM is slightly better than that obtained by MPICH-gm. The shared memory latency of 1.23 μsec is about 1 microsecond better than MPICH2, but the TCP latency of 32.0 μsec, is 3 microseconds higher than MPICH2 (tuning is ongoing). The latency of 5.64 μsec using the MVAPI verbs is about 1.5 higher μsec than MVAPICH's remote queue management scheme. However, because Open MPI uses the Shared Receive Queue support provided by the MVAPI verbs, it scales much better than MVAPICH by using far less memory. MVAPICH's latency increases linearly with the number of processes in MPI_COMM_WORLD; Open MPI's latency remains essentially constant (and is lower than MVAPICH's) [11]. Open MPI's latency is about one microsecond lower than MVAPICH's send/receive semantics.

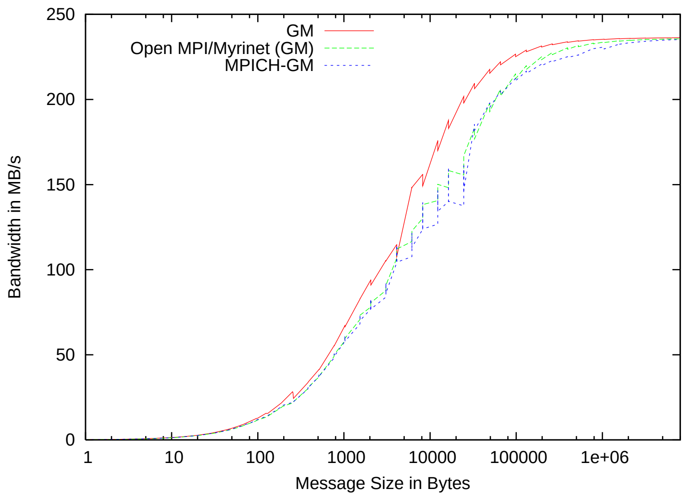

Fig. 3. Myrinet / GM bandwidth of Open MPI vs. MPICH-gm.

4.2 Point-To-Point Bandwidth

Fig. 3 shows Open MPI obtaining slightly higher bandwidths than MPICH-GM, peaking out around 235 MB/sec – very close to the raw GM bandwidth. Similarly, Fig. 4 shows that Open MPI shows better shared memory bandwidths than MPICH2, peaking out at 1020 MB/sec, with a message size of about 1 MB. MPICH2 and Open MPI over TCP (Gigabit Ethernet) exhibit similar bandwidths, peaking out at 112 MB/sec, shown in Fig. 5. Open MPI's MVAPI bandwidth is shown in Fig. 6; it is quite similar to those obtained by MVAPI, peaking out at 915 MB/sec. Finally, Open MPI's Open IB bandwidth is shown in Fig. 7; no other Open IB-native MPI is available to compare to.

5 Summary

Open MPI is a new implementation of the MPI 2.0 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. Open MPI uses a flexible component architecture, and it's point-to-point design is such that it provides excellent point-to-point communications performance for wide variety of interconnects, all within a single library implementation.

Acknowledgments

This work was supported by a grant from the Lilly Endowment, National Science Foundation grants 0116050, EIA-0202048, ANI-0330620, 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 con-

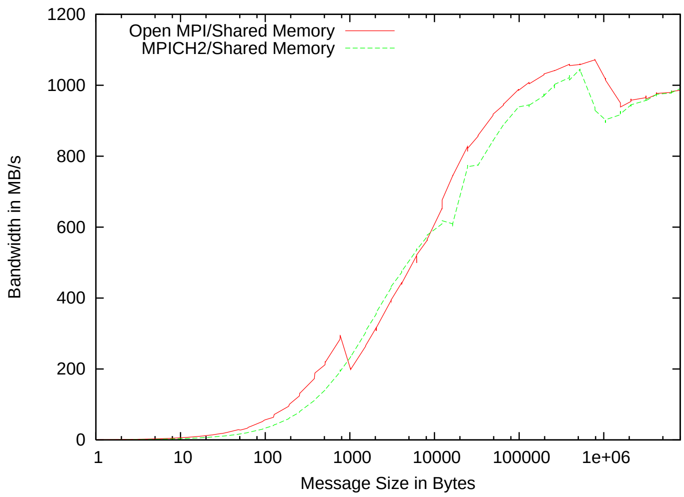

Fig. 4. Shared memory bandwidth of Open MPI vs. MPICH2.

tract W-7405-ENG-36. Project support was provided through ASC/PSE and ASC/S&CS programs. LA-UR-05-9219.

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] Rainer Keller, Edgar Gabriel, Bettina Krammer, Matthias S. Mueller, and Michael M. Resch. Towards efficient execution of parallel applications on the grid: porting and optimization issues. *International Journal of Grid Computing*, 1(2):133–149, 2003.

[6] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. High performance RDMA-based MPI implementation over infiniband. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 295–304, New York, NY, USA, 2003. ACM Press.

[7] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, June 1995. http://www.mpi-forum.org/.

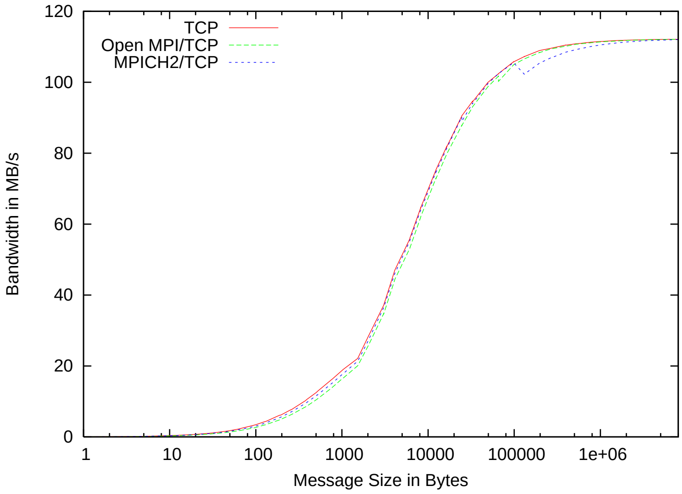

Fig. 5. TCP (Gigabit Ethernet) bandwidth of Open MPI vs. MPICH2.

[8] Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface*, July 1997. http://www.mpi-forum.org/.

[9] MPICH2. http://www.mcs.anl.gov/mpi/mpich2/.

[10] 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.

[11] Galen M. Shipman. Infiniband scalability in Open MPI. Master's thesis, University of New Mexico, December 2005.

[12] Q.O. Snell, A.R. Mikler, and J.L. Gustafson. NetPIPE: A Network Protocol Independent Performace Evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, June 1996.

[13] J.M̃. Squyres and A. 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.

[14] 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.

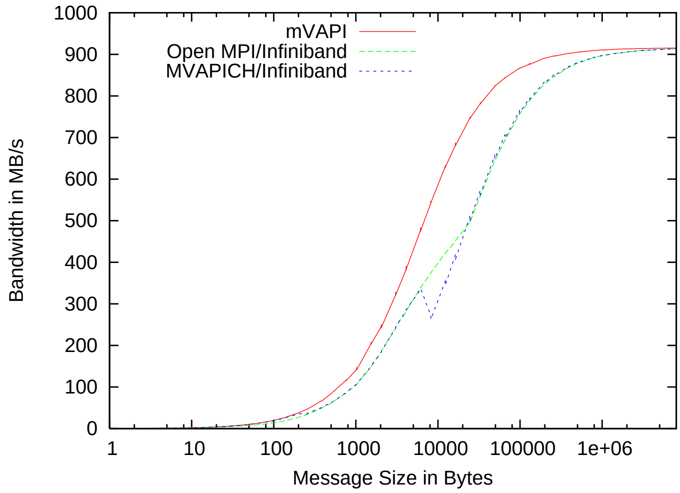

Fig. 6. Mellanox VAPI bandwidth of Open MPI vs. MVAPICH.

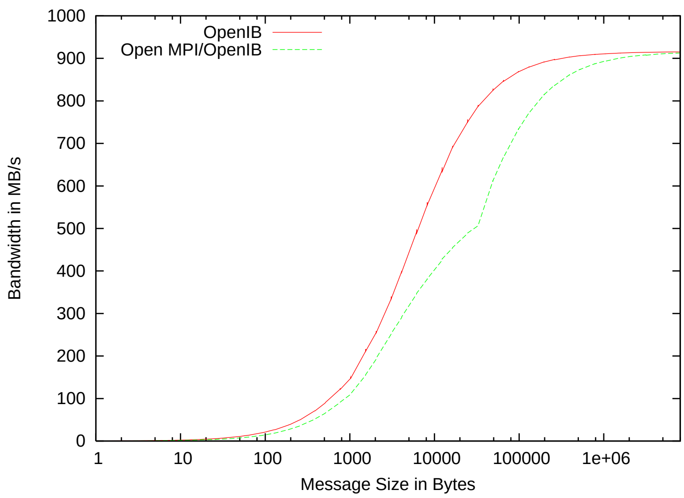

Fig. 7. Open IB bandwidth of Open MPI (no other MPI implementation to compare to).