Open MPI: A High-Performance, Heterogeneous MPI

Richard L. Graham [1], Galen M. Shipman [1], Brian W. Barrett [2],
Ralph H. Castain [1], George Bosilca [3], and Andrew Lumsdaine [2]

[1]Los Alamos National Laboratory
Advanced Computing Laboratory
Los Alamos, NM USA
{rgraham, gshipman, rhc}@lanl.gov

[2]Indiana University
Open Systems Laboratory
Bloomington, IN USA
{brbarret, lums}@osl.iu.edu

[3]University of Tennessee
Dept. of Computer Science
Knoxville, TN USA
{bosilca}@cs.utk.edu

Abstract

The growth in the number of generally available, distributed, heterogeneous computing systems places increasing importance on the development of user-friendly tools that enable application developers to efficiently use these resources. Open MPI provides support for several aspects of heterogeneity within a single, open-source MPI implementation. Through careful abstractions, heterogeneous support maintains efficient use of uniform computational platforms. We describe Open MPI's architecture for heterogeneous network and processor support. A key design features of this implementation is the transparency to the application developer while maintaining very high levels of performance. This is demonstrated with the results of several numerical experiments.

1. Introduction

While heterogeneous, distributed computing has been around for quite some time, much of the focus has been on resolving functional concerns at the subsystem level (e.g., cross-domain authentication and task scheduling). The usability and application-level performance of the resulting environment, however, are system-level issues and have generally received less attention. The Open MPI project is aimed at providing this system-level integration, with the overall goal of addressing both of these issues.

The concept of heterogeneous computing has taken on a number of meanings over the years. For clarity, Open MPI defines a *job* as the execution of either a single application, or multiple communicating applications, each potentially on a separate computing system. Within that context, the project generally breaks the definition of heterogeneity into four broad categories:

Processor heterogeneity Dealing with differences in processor speed, internal architecture (e.g., multi-core), and communication issues caused by the transfer of data between processors with different data representations (different endian, floating point representation, etc.).

Network heterogeneity Using different network protocols between processes in a job. This includes multiple network protocols between two processes and different network protocol to different processes in the job.

Run-Time Environment heterogeneity Executing a job across multiple resources (e.g., multiple clusters) that are locally administered by possibly different scheduling systems, authentication realms, etc.

Binary heterogeneity Coordinating execution of different binaries, either through the coupling of different applications or as part of a complex single application.

While all of these issues are being addressed within the Open MPI system, this paper will focus primarily on dealing with processor and network heterogeneity. The number of potential combinations facing the application developer for processor and network interfaces poses a signif-

1-4244-0328-6/06/$20.00

icant challenge when attempting to build or use a high-performance application. One of the guiding principles in the Open MPI project is to decompose the data exchange and manipulation interfaces in such a way as to hide details from both the application developer and the end user while providing effective and portable application performance. At the same time, those users that want to have very fine control over resource utilization may do so with some extra effort. The design goals of the project fall into two areas that reflect this philosophy:

Transparency Differences in the local environment for the application's processes shall be transparent to the application. The Open MPI library will detect the local environment during initialization of each process, determine the appropriate configuration for maximum performance, and configure the local library appropriately. Differences in the environment can arise from two sources:

- Operation across multiple clusters, grids, or autonomous computers. Open MPI introduces the concept of a *cell* – i.e., a collection of nodes within a common environment – to commonly refer to clusters, grids, and other forms of collective computing resources. Multi-cell operations requires the careful apportioning of the application to each cell, and the coordination of launch operations and subsequent wire-up of the MPI communication system. The problem of automatically apportioning an application across heterogeneous cells is beyond Open MPI's scope – however, once apportioned, Open MPI will launch and execute the application with no further guidance.
- Variations in the configuration of nodes within a single cluster or grid – this includes differences in the amount of available memory, operating system, available network interfaces, and processor architecture.

Performance Optimal communications performance is a design goal. Heterogeneous support should not impact performance of connections between processes running in a homogeneous environment. This mandates the handling of heterogeneous processors at the connection level, rather than mandating a global wire protocol.

This paper will describe the Open MPI collaboration's approach to meeting these goals, and quantify the overall performance of the system. Following a brief review of related work, we present an overview of Open MPI's design with a focus on dealing with processor and network heterogeneity – this will include a detailed description of how the system automatically detects the local environment and configures itself for optimized performance. Finally, the performance of the system in several illustrative scenarios is presented.

2. Related Work

As heterogeneous computing provides an opportunity to maximize available computing resources, there has been a fair amount of work done on inter-process communications processes in MPI applications. Many current production MPI implementations (e.g., LAM/MPI [4], FT-MPI [7], MPICH-G2 [11], StaMPI [10], and PACX-MPI [12]) support the operation of applications within such an environment. Since the Open MPI collaboration involves the developers of several of the more popular implementations, the project has built upon that prior experience to provide an enhanced capability in this area.

The HeteroMPI [13] project provides extensions to the MPI standard for heterogeneous computing. Library assistance is provided to assist in decomposing the problem to best fit the available resources. HeteroMPI still relies on an underlying MPI implementation for process startup and high-performance communication. Our current research in Open MPI seeks to provide the greatest flexibility and performance in precisely the areas where HeteroMPI depends on an underlying MPI implementation for functionality.

Much less work has been done in the area of network heterogeneity, particularly with respect to the optimized use of multi-network interfaces within messages. LA-MPI [9], for example, supports multi-network communications, but with the restriction that any individual message be sent with a single messaging protocol. Similarly, MPICH-VMI2 [15] implements MPI communications over the Grid using a high performance protocol within a cluster, and TCP/IP communications between clusters, but does not support the simultaneous use of multiple network interfaces on a single node.

The Open MPI collaboration seeks to extract the best of these experiences and combine them into a single implementation focused on providing performance in a manner that is both transparent to the user and yet fully configurable when desired. The resulting architecture is described in the following section.

3. Open MPI Design

The Open MPI project is a broad-based collaboration of U.S. national laboratories with industry and universities from both Europe and the United States. It aims to provid-

ing a high performance, robust, parallel execution environment for a wide variety of computing environments, including small clusters, peta-scale computers, and distributed, grid-like, environments. From a communications point-of-view, Open MPI has two functional units: the Open Run-Time Environment (OpenRTE) [5, 6], responsible for bootstrapping operations; and the Open MPI [8] communications library, providing highly efficient communications support, tailored to specific communications context. These two units work together to support transparent execution of applications in a heterogeneous environment.

3.1. Bootstrapping Start-up

OpenRTE provides transparent, scalable support for high-performance applications in heterogeneous environments. OpenRTE originated within the Open MPI project but has since spun-off into its own effort, though the two projects remain closely coordinated. The ability of Open MPI to transparently operate in heterogeneous environments is largely due to the support services provided by OpenRTE, so it is useful to understand the OpenRTE architecture.

The OpenRTE system is comprised of four major functional groups:

General Purpose Registry (GPR) A centralized publish/subscribe data repository used to store administrative information (e.g., communication contact information and process state) supporting OpenRTE and Open MPI operations. The GPR is also available for use by applications to synchronize their internal operations, though it is not intended for the storage or communication of general application data.

Resource Management (RM) group A collection of four subsystems supporting the identification and allocation of resources to an application, mapping of processes to specific nodes, and launch of processes.

Support Services A suite of subsystems that provide general support for OpenRTE operations, including services to generate unique process names, I/O forwarding between application processes and the user, and OpenRTE's communication subsystem. The OpenRTE communication subsystem utilizes a well defined wire protocol with data being packed in network byte order – this ensures accurate communication during the start-up process when processor architecture of the individual elements within the system remains unknown. OpenRTE provides support for sized integer data types, generic integer data types (e.g., `size_t`, `bool`, and `pid_t`), as well as a means for users to define their own (possibly structured) data types.

Error Management (EM) group Several subsystems that collectively monitor the state of processes within an application and respond to changes in those states that indicate errors have occurred.

Implementation of each of these groups and Open MPI itself is based upon the Modular Component Architecture (MCA) [16], used within both OpenRTE and Open MPI. Within this architecture, each of the major subsystems is defined as an MCA *framework* with a well-defined Application Programming Interface (API). In turn, each framework contains one or more *components*, each representing a different implementation of that particular framework that includes support for the entire API. Thus, the behavior of any OpenRTE or Open MPI subsystem can be altered by defining another component within a given framework and requesting that it be selected for use.

Alternatively, the user can allow the system to dynamically sense its environment and automatically select the best components for that situation. Open MPI uses this capability, for example, to identify the available network interfaces on each node and select a strategy for both maximizing throughput and minimizing latency. Similarly, OpenRTE depends upon the dynamic component selection properties within the MCA to identify the necessary interfaces for launching a process, allocating resources, etc.

Open MPI's use of the component architecture to bootstrap the start-up of an MPI application in heterogeneous environments can best be illustrated by considering the launch of such an application on a single cell. Once the user enters the *mpirun* command, OpenRTE begins the launch process by starting a *head node process* (HNP) on the cell. A *head node* is a node within a cell from which processes can be launched – typically the front end machine of a cluster, or a grid controller. The head node on most cells is also where the user will have logged in to execute the *mpirun* command. The HNP serves as the local coordinator for all OpenRTE activities for this user on the cell, including launch and monitoring of processes, routing of messages to/from processes on the cell to those on another cell, etc.

As the HNP starts, it allows each OpenRTE framework to sense the local environment to determine the type of cell upon which it is launching – e.g., the local resource manager (SLURM, BProc, etc.), operating system, and any resource allocations that may already exist for this user. Appropriate components for each framework are selected depending upon either the user's specification or the individual component's judgment that it can best handle the given environment. Once the appropriate components have been selected, the HNP proceeds to launch the processes on the individual nodes within the cell.

Similarly, each application process as it starts executing on a node first passes through the identical framework selection mechanism to identify its appropriate configuration.

This configuration information is made available to all other processes in the job by sending it to the GPR for storage. As these processes are not on the head node, many of the frameworks will select `proxy` components that do little more than relay commands and responses to/from the HNP. For example, requests to dynamically spawn additional processes are passed to the HNP for execution, as are requests to store or retrieve data from the GPR.

During the start-up procedure, each process reports items such as its location, contact information for all available network transports on their local node, and node architecture into the GPR via their connection to the HNP, and subscribes to the contact information from all other processes in the application. This subscription serves two purposes:

- When all processes reach a pre-defined *barrier* within MPI_INIT, the subscriptions are used by OpenRTE to identify the information that is to be sent back to the requesting processes. Such information includes contact information and an architecture description, required for establishing MPI communication.
- Upon change of any process' contact information (e.g., when a process fails and must be restarted), all processes holding a subscription on that data are automatically notified with the updated information. This allows Open MPI to transparently maintain connectivity throughout the application's lifetime.

Thus, OpenRTE provides a bootstrap capability for both initially communicating process contact information, and for establishing a transparent mechanism for communication information updates. Open MPI takes advantage of this mechanism to share information on the availability of high-speed network interfaces, processor architecture, and other configuration parameters to provide a high-performance message passing environment.

3.2. Open MPI Multiple Network Support

As with OpenRTE, the MPI layer utilizes the MCA system to adapt to the system architectures used by an application. Presently, Open MPI supports MPI communication over a wide variety of communication protocols, including shared memory, InfiniBand [2] (mVAPI and OpenIB), Myrinet [14] (GM and MX), TCP/IP, and Portals [3]. These components provide highly optimized and scalable interconnect support and also provide a number of run time parameters which allow for easy tuning for a specific operating environment or application. Open MPI supports two forms of network heterogeneity – it can stripe a single message to a single destination over multiple networks (of either the same or different communication protocols) and it can communicate to different peers using different communication protocols.

The following section will describe the point-to-point architecture of Open MPI, and how this is used to achieve a high performance communication architecture, even in heterogeneous network environments.

3.2.1. Point-to-Point Design

Open MPI provides point-to-point message transfer facilities via multiple MCA frameworks. These frameworks and the overall point-to-point architecture is illustrated in Figure 1.

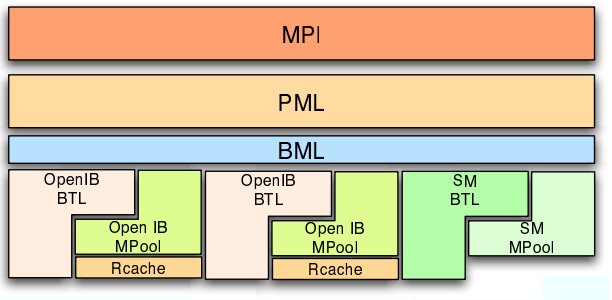

Figure 1. Open MPI point-to-point communication framework

The point-to-point architecture consists of four main layers: the Byte Transport Layer (BTL), BTL Management Layer (BML), Point-to-Point Messaging Layer (PML) and the MPI layer. Two additional frameworks are shown, the Memory Pool (MPool) and the Registration Cache (Rcache).

MPool Many RDMA based interconnects require memory to be registered with the interconnect prior to any send/receive or RDMA operation which uses the memory as the target or source. Registration and deregistration services are provided by the MPool framework which allows other components to use these services. Both PML and BTL components use the MPool in addition to the MPI level where MPI_ALLOC_MEM uses the services of the MPool.

Rcache Registration of memory can be a high cost operation. In fact, the "bandwidth" of memory registration is often less than the bandwidth of the interconnect and may therefore prevent a bottleneck to otherwise high performance. With this in mind, the registration cache provides facilities for caching and later searching for registrations. This allows the high cost of memory registration to be effectively amortized over multiple RDMA operations. Several of the MPool components use the facilities of the Rcache to allow registrations

to be reused over potentially multiple RDMA operations.

BTL The BTL framework provides a uniform method of data transfer for numerous interconnects, both send/receive and RDMA based. The BTL components are MPI agnostic acting as simple byte movers with facilities for both local and remote completion as appropriate to the underlying interconnect. Local completion facilities are provided by a simple callback mechanism for each fragment scheduled on the BTL. Remote completion is accomplished via an Active Message [17] style facility where Active Message callbacks are registered along with an Active Message Tag (AM-Tag) value during BTL initialization. Fragments scheduled on the BTL include an AM-Tag value which provides remote completion callback.

BML In order to allow multiple components to use the BTL components, the BML provides a facilities for BTL initialization and resource discovery (via the BTL). After BTL initialization the BML layer is effectively bypassed via inline functions to the BTL.

PML As the BTL components provide simple byte transfer services, higher level MPI point-to-point semantics are implemented by the PML. Message scheduling and progression is located in the PML as well as MPI specific protocols such as short and long message protocols. This isolation of higher level semantics allows the BTL components to be fairly simple and lightweight which allows easier adoption of new interconnect technologies. This structure also allows for fine grain scheduling of messages across multiple interconnects as well as the ability to change scheduling policies based on interconnect properties. There are currently two PML components under active development `OB1`, a high performance PML optimized for reliable interconnects and `DR` a network fault tolerant PML providing adaptable performance based on the overall reliability of the underlying interconnect(s).

3.3. Resource Discovery and Initialization

During start-up, a PML component is selected and initialized. The PML component selected defaults to `OB1` but may be overridden by a run-time parameter/environment setting. The `OB1` PML component then opens the BML component `R2`. `R2` then opens and initializes all available BTL modules.

During BTL initialization local resources are discovered this may include:

1. Opening Devices
2. Checking for active status
3. Creating a module for each active device
4. publish addressing information to the GPR

After local resource discovery Open MPI initialization begins peer resource discovery. Peer information is queried from the GPR and this information is passed to the PML level. The PML then directs the BML and finally the BTL to parse the peer information. Each BTL is passed the peer information along with a reachability mask. The BTL will then attempt to match the local resource to one of the peer's published resources. If a match is made an `endpoint` is created representing a connection to the peer process.

This infrastructure provides a uniform method of local and remote resource discovery and resource matching. No assumption is made regarding the symmetry of peer resources thereby allowing peers within a single job to have different network interconnects so long as *some* connection exists between each set of peers that will communicate in the job. This infrastructure allows for heterogeneous networking interconnects within a cluster.

3.4. Message Scheduling

To effectively utilize multiple network interconnects Open MPI provides a mechanism to schedule a single message across these network resources. This mechanism is currently isolated at the BTL and BML levels in such a way as to allow other components to implement an effective scheduling policy. The BML also provides a simple round robin scheduling policy which other component may use as appropriate. For point-to-point communication the PML uses both round robin and custom scheduling based on a variety of factors.

Interconnects may exhibit widely different performance characteristics which a scheduling policy should take into account. These performance characteristics are exported by each BTL and include both bandwidth and latency. During BTL initialization the BML prioritizes each BTL based on these characteristics allowing upper level components such as `OB1` to choose the appropriate interconnect(s) to communicate with a peer. In addition to performance characteristics, the BML groups interconnects based on capabilities such as send/receive and RDMA. These groupings are cached on a data structure associated with each peer for efficient access. These groupings include Eager (Low Latency), Send/Receive and RDMA capable BTLs.

When scheduling a message `OB1` will eagerly send a specified amount of data choosing a BTL for the peer from the Eager list created by the BML. The amount of data scheduled is BTL specific allowing resource usage and performance to be balanced on a per BTL basis. Further message fragments may be delivered using Send/Receive based

BTLs where each fragment size is determined by the maximum send size of the BTL and the number of fragments allocated to the BTL is determined by a weighting factor based on BTL bandwidth. The round robin facilities provided by the BML are used to choose the next BTL on which to schedule a fragment. Fragments may also be scheduled using RDMA BTLs in a similar manner.

3.5. Open MPI Derived Datatype Engine

Processor heterogeneity within the MPI layer is handled by the Derived Datatype Engine (DDT), which implements MPI datatype conversion for Open MPI. The DDT engine consists of two parts – the MPI datatype interface and a *datatype convertor*. The convertor handles packing and unpacking messages for transmission, although convertor packing and unpacking does not always require copying into a separate buffer. A number of packing and unpacking algorithms are available, with selection of the optimal algorithm selected for each message transmission.

During MPI_INIT, each process determines the local architecture and creates an architecture description, a 32-bit long bitmask that can be shared among processes. Once computed, the local value is published to the OpenRTE GPR and shared across all processes in the current job, as described in Section 3.1. The architecture description of all peers with which a given process can communicate is readily accessible to the PML/BML and datatype engine. The architecture bitmask includes the information described in Table 1, which encompasses the heterogeneity issues encountered on platforms supported by Open MPI. The size of types is computed at compile time, using the `sizeof()` operator in C. The floating point description is determined by the information provided in the `float.h` header file, and the endian representation and mantissa representation are determined at run time.

The BML and datatype engine use this architecture value (which is a 32 bit integer and therefore easily and quickly comparable) to determine whether the heterogeneous code is needed. The BML level automatically disables RDMA communication between endpoints with different architectures, meaning that communication layers building on the BML/BTL design are generally not aware of processor heterogeneity. One exception is the use of component-specific headers, which are converted to network byte order if different the peers have different endian representations. Both the MPI point-to-point and one-sided communication components are able to build on the BML/BTL design and provide heterogeneous support with little extra work.

Presently, only send/receive semantics are used for communication between processes of different architecture. The BTL receive semantics require a copy of the buffer from BTL memory to user memory. Therefore, a *receiver-makes-right* approach is taken and the unpack routines handle all details of the heterogeneous datatype support. This includes endian correction, floating point format conversion for long doubles, and support for different boolean formats (both Fortran `LOGICAL` and C++ `bool`). Conversion between datatypes of different sizes (such as a 64 bit MPI_LONG to a 32 bit MPI_LONG) is not presently supported, although it is planned for a future release.

Byte	Bits	Description
1	1 - 2	Always 00, allowing recognition of endian encoding
	3 - 4	endian: 00 = little, 01 = big
	5 - 6	reserved: Always 00
	7 - 8	reserved: Always 00
2	1 - 2	length of long: 00 = 32, 01 = 64
	3 - 4	reserved for length of long long: Always 00
	5 - 6	length of C/C++ bool: 00 = 8, 01 = 16, 10 = 32
	7 - 8	length of Fortran LOGICAL: 00 = 8, 01 = 16, 10 = 32
3	1 - 2	length of long double: 00 = 64, 01 = 96, 10 = 128
	3 - 4	number of bits in the exponent of a long double: 00 = 01, 01 = 14
	5 - 7	number of bits of mantissa in a long double: 000 = 53, 001 = 64, 010 = 105, 011 = 106, 100 = 107,101 = 113
	8	Intel or SPARC representation of mantissa: 0 = SPARC, 1 = Intel
4	1 - 2	Always 11, allowing recognition of endian encoding
	3 - 4	reserved: Always 11
	5 - 6	reserved: Always 11
	7 - 8	reserved: Always 11

Table 1. Information contained in 32-bit architecture string. Lengths are given in bits.

4. Results

This section presents performance data for several numerical experiments. They include performance data on network and processor heterogeneity.

4.1. Network Heterogeneity

4.1.1. Experimental Setup

Network heterogeneity benchmarks were performed on a 4 node partition of a 256 node cluster consisting of dual Intel Xeon X86-64 3.4 GHz processors with a minimum 6GB of RAM, Mellanox PCI-Express Lion Cub adapters connected via a Voltair 9288 switch, Myrinet PCI-X LANai 10.0 adapters connected via a Myrinet 2000 switch. The operating system is Red Hat Enterprise Linux 4, kernel version 2.6.9-11. Tests were run with Open MPI pre-release 1.1.1.

4.1.2. Results

The ability to stripe a single message across multiple interconnects can provide enhanced performance depending on the message size and communication pattern. As show in Figure 2, when multiple interconnects are used for medium and large message sizes there is an additive effect on total bandwidth. For smaller messages the lower latency of the Mellanox adapter (when compared to a much older Myrinet adapter using GM) results in increased latency when fragments are striped on both interconnects. For medium size messages, striping the data across the two very different interconnects actually reduces the bandwidth, relative to sending the data only over the InfiniBand interconnect, again due to latency differences. Changes to the simplistic scheduling policy can remedy the negative performance impact for both small and medium sized messages, but this will be investigated methodically in a future publication. For large messages, the current scheduling algorithm is sufficient to produce a near additive increase in the observed bandwidth, when using both the Myrinet and the InfiniBand interconnects, as the time on wire dominates the overall data transmission time.

The effect of the multi-nic scheduling on a user application is briefly explored using a visualization application. For applications which use larger message sizes multi-nic scheduling can provide significant improvements. Table 2 demonstrates the performance impact of multi-nic scheduling on a visualization display benchmark (ParaView [1] simulation). While the Mellanox interconnect provides good performance compared to the older Myrinet interconnect combining the two interconnects provides a marked improvement over the single nic case.

4.2. Processor Heterogeneity

4.2.1. Experimental Setup

Processor heterogeneity experiments were run on two clusters, one Power PC and one Opteron, both connected to the same gigabit Ethernet switch. The Power PC cluster consists of 16 dual 2.3 GHz Apple G5 (IBM Power PC 970) machines with 4GB of memory per node. The operating system is Mac OS X 10.4.6. The Opteron cluster consists of 128 dual 2.0 GHz AMD Opteron machines with 4GB of memory per node. The operating system is Red Hat Enterprise Linux 4, kernel version 2.6.9-34.ELsmp. Open MPI 1.1.1 pre-release was used for the experiments.

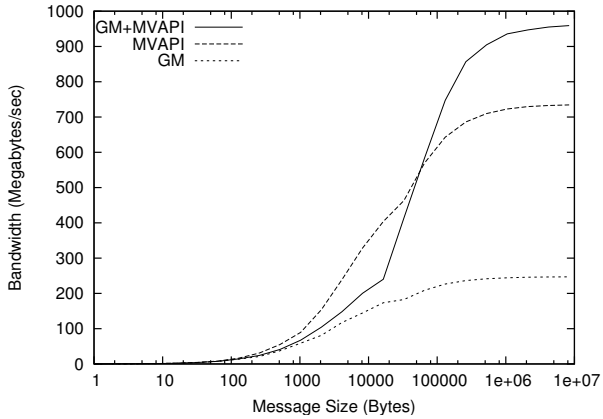

Figure 2. Impact of network heterogeneity on point-to-point bandwidth. Message buffers are allocated via MPI_ALLOC_MEM

Interconnect	Total Time
Myrinet GM Only	24.92 (sec)
Mellanox MVAPI Only	8.53 (sec)
Myrinet GM + Mellanox MVAPI	6.55 (sec)

Table 2. 4 Node (single process per node) Visualization Display Benchmark (Paraview simulation)

4.2.2. Results

Figure 3 demonstrates the impact of endian conversion on bandwidth, using a modified version of NetPIPE to send data as a series of MPI_INTs instead of MPI_BYTEs. MPI_BYTE requires no datatype conversion (indeed, that is the point of MPI_BYTE as an MPI datatype), so there is no penalty for sending a series of MPI_BYTEs in a heterogeneous environment. By using MPI_INTs, datatype conversion is required for heterogeneous data transfer. In the test platform used, the heterogeneous cases requires all data to have it's endianness adjusted by the receiver. As can be seen, there is a small performance hit for performing en-

dian conversion, approximately a 5 megabit / second drop in bandwidth for very large messages.

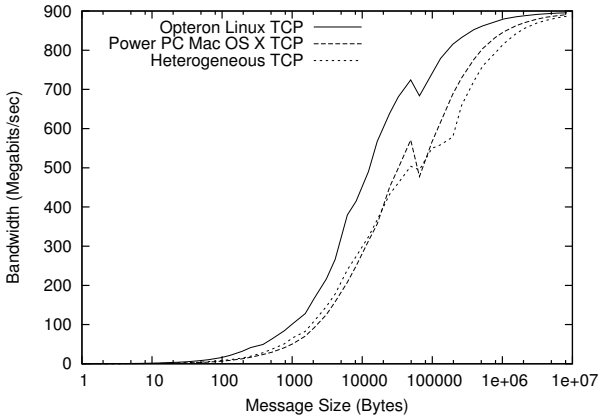

Figure 3. Impact of processor heterogeneity on point-to-point bandwidth.

Figure 4 shows the performance of Open MPI running the IMB Reduce benchmark at 16 nodes. Similar to the NetPIPE results, comparisons are given against homogeneous systems. The heterogeneous tests were run with two different layouts. In one, ranks 0 - 7 were on Opteron processors and ranks 8 - 15 on Power PC processors (block ordering). In the other, even ranks were on Opteron processors and odd ranks on Power PC processors (round-robin ordering). As can be seen in the results, the allocation mechanism does play a factor in performance. On applications that perform communication within groups, this performance hit can be even more noticeable. As expected, the reduction performance for small messages is correlated to the small message latency performance of the nodes in use. For larger messages, performance of the heterogeneous runs is worse than the homogeneous runs, due to the overhead of the extra memory manipulation to "fix up" endian differences.

4.3. Summary

Open MPI is capable of effectively utilizing multiple communication channels between a single pair or peers. This can lead to improved application performance, as seen with the ParaView application. There is little to no cost to the end use for supporting network heterogeneity. Processor heterogeneity, on the other hand, comes at a cost, which can be significant for medium sized messages. The overall impact of this cost is somewhat limited by only performing datatype conversion when absolutely needed (on a peer basis), but is unavoidable in some cases.

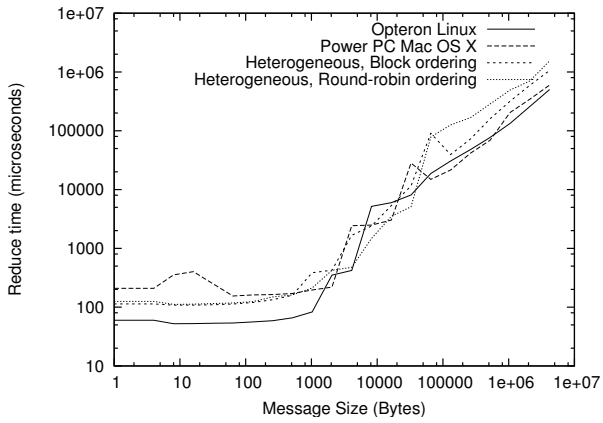

Figure 4. Impact of processor heterogeneity on MPI_REDUCE **performance.**

5. Conclusions

Open MPI provides a high performance platform for parallel applications on both homogeneous and heterogeneous platforms. The application developer is not unduly burdened by the cost of heterogeneous application development as Open MPI transparently handles process start-up, communication, and data conversion. Open MPI also determines all architecture and networking properties on a per-peer basis, and selects the most efficient mode of communication with the given peer, in order to maximize performance. As seen in Section 4, our design to support multiple, disparate networks can offer a performance increase in some situations. The cost for different endianess support, is also incured, only when needes. Further, the architecture allows us to continue research into efficient and user-friendly support for heterogeneous computing, as well as areas such as network fail-over and recovery, without drastic modification to the device-specific code.

6. Acknowledgments

Project support was provided through the United States Department of Energy, National Nuclear Security Administration's ASCI/PSE program and the Los Alamos Computer Science Institute; a grant from the Lilly Endowment and National Science Foundation grants NSF-0116050, EIA-0202048 and ANI-0330620; and the Center for Information Technology Research (CITR) of the University of Tennessee.

Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy un-

der contract W-7405-ENG-36.

The authors would like to thank all those who also contributed to development of Open MPI, particularly Tim Woodall (formerly of Los Alamos National Laboratory, currently at NetQoS). LA-UR-06-3453.

References

[1] J. Ahrens, B. Geveci, and C. Law. *ParaView: An End User Tool for Large Data Visualization*. Academic Press, 2005.

[2] I. T. Association. Infiniband architecture specification vol 1. release 1.2, 2004.

[3] R. Brightwell, T. Hudson, A. B. Maccabe, and R. Riesen. The portals 3.0 message passing interface, November 1999.

[4] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[5] R. Castain and J. Squyres. Creating a transparent, distributed, and resilient computing environment: The Open-RTE project. *Journal of Supercomputing*, To appear, 2007.

[6] R. Castain, T. Woodall, D. Daniel, J. M. Squyres, B. Barrett, and G. Fagg. The Open Run-Time Environment (Open-RTE): A transparent multi-cluster environment for high-performance computing. In *Proceedings, 12th European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.

[7] G. Fagg, E. Gabriel, G. Bosilca, T. Angskun, Z. Chen, J. Pjesivac-Grbovic, K. London, and J. Dongarra. Extending the mpi specification for process fault tolerance on high performance computing systems. In *Proceedings of the 2004 International Supercomputing Conference (ISC2004)*, 2004.

[8] E. Gabriel, G. Fagg, G. Bosilica, T. Angskun, J. J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. Castain, D. Daniel, R. Graham, and T. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 2004.

[9] 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), August 2003.

[10] T. Imamura, Y. Tsujita, H. Koide, and H. Takemiya. An architecture of StaMPI: MPI library on a cluster of parallel computers. In *Proceedings, 7 European PVM/MPI Users' Group Meeting*, September 2000.

[11] N. Karonis, B. Tonnen, and I. Foster. MPICH-G2: a grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 63(5):551–63, May 2003.

[12] R. Keller, E. Gabriel, B. Krammer, M. S. Mueller, and M. M. Resch. Towards efficient execution of MPI applications on the grid: porting and optimization issues. *Journal of Grid Computing*, 1:133–149, 2003.

[13] A. Lastovetsky and R. Reddy. HeteroMPI: Towards a Message-Passing Library for Heterogeneous Networks of Computers. *Journal of Parallel and Distributed Computing*, 66(2):197 – 220, 2006.

[14] Myricom. Myrinet-on-VME protocol specification.

[15] A. Pant and H. Jafri. MPICH-VMI: A high performance MPI implementation of Teragrid. `http://vmi.ncsa.uiuc.edu/presentations/vmi-mpich-arch-perf.ppt`.

[16] J. M. Squyres and A. Lumsdaine. The component architecture of Open MPI: Enabling third-party collective algorithms. In V. Getov and T. Kielmann, editors, *Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, pages 167–185, St. Malo, France, July 2004. Springer.

[17] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.