Implementation of Open MPI on the Cray XT3

Brian W. Barrett, Indiana University,
Ron Brightwell, Sandia National Laboratories,
Jeffrey M. Squyres, Cisco Systems, and
Andrew Lumsdaine, Indiana University

Abstract

The Open MPI implementation provides a high performance MPI-2 implementation for a wide variety of platforms. Open MPI has recently been ported to the Cray XT3 platform. This paper discusses the challenges of porting and describes important implementation decisions. A comparison of performance results between Open MPI and the Cray supported implementation of MPICH2 are also presented.

KEYWORDS: Cray XT3, Open MPI

1 Introduction

Open MPI [9] is a high performance, portable implementation of the Message Passing Interface. Open MPI's performance on commodity Linux clusters with high speed interconnects is well established [18], and similar results are found on other commercial and commodity clustering platforms, including AIX, Mac OS X, and Solaris. The current release includes support for communication over Myrinet (MX and GM), InfiniBand (OpenIB and MVAPI), TCP, Portals, and shared memory, with support for uDAPL and Quadrics under development.

This paper presents our experiences porting Open MPI to the Cray XT3 platform, specifically the Red Storm machine at Sandia National Laboratories. As the XT3 environment is the first "tightly-coupled" system to be supported by Open MPI, some design re-factoring was required, especially within the parallel run-time support library utilized by Open MPI, discussed in Section 3. In Section 4 we describe our implementation of point-to-point messaging over Cray Portals. Performance results are presented in Section 5. Finally, future work is discussed in Section 6.

2 Background

2.1 Open MPI

Open MPI is the result of a collaboration between the authors of the LAM/MPI [5, 17], LA-MPI [2, 11], and FT-MPI [7, 8] implementations of the MPI standard. Open MPI is a full implementation of both the MPI-1 [13, 15] and MPI-2 [10, 12] standards, designed to offer high performance and scalability on a variety of platforms. Open MPI builds on two other projects (Figure 1), which were originally developed as part of the Open MPI project, but are slowly becoming independent projects. OPAL provides system portability code to limit platform-specific workarounds in the general code base, as well as building block code for upper layers. The Open Run-Time Environment (OpenRTE) [6] provides a uniform, portable run-time support system for the MPI messaging layer. The entire system is built around a low overhead component architecture, allowing platform specific code to be implemented within well-defined abstractions [3, 16]. On platforms with shared library support, individual components can be provided as dynamically loaded shared objects, allowing for run-time reconfiguration of Open MPI.

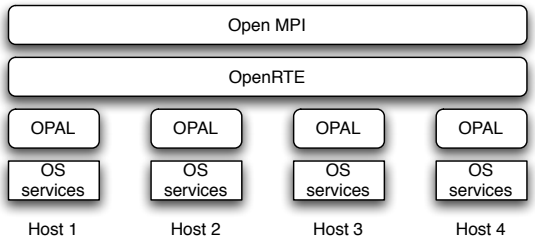

Figure 1: Open MPI high-level design, including logical abstractions of portability code, parallel run-time services, and the MPI implementation.

OPAL provides basic portability and building block features useful for large scale application development, serial or parallel. A number of useful functions provided only on a handful of platforms (such as asprintf, snprintf, and strncpy) are implemented in a portable fashion, so that the rest of the code can assume they are always available. High resolution / low perturbation timers, atomic memory operations, and memory barriers are implemented for a large number of platforms. The core support code for the component architecture, which handles loading components at run-time, is also implemented within OPAL. OPAL also provides a rich reference counted object system to simplify memory management, as well as to implement a number of container classes, such as doubly-linked lists, last-in-first-out queues, and memory pool allocators.

OpenRTE provides a resource manager (RMGR) to provide process control, a global data store (known as the GPR), an out-of-band messaging layer (the RML), and a peer discovery system for parallel start-up (SDS). In addition, OpenRTE provides basic datatype support for heterogeneous network support, process naming, and standard I/O forwarding. Each subsystem is implemented through a component framework, allowing whole-sale replacement of a subsystem for a particular platform. On most platforms, the components implementing each subsystem utilize a number of underlying component frameworks to customize OpenRTE for the specific system configuration. For example, the standard RMGR component utilizes additional component frameworks for resource discovery, process start-up and shutdown, and failure monitoring.

2.2 Cray XT3 / Sandia Red Storm

The Red Storm machine at Sandia National Laboratories in Albuquerque, New Mexico currently consists of 10,368 processors. Each node contains a single 2.0 GHz Opteron CPU with 2 GB of main memory and a Cray SeaStar NIC/router attached via HyperTransport. The network is a 27x16x24 mesh topology, with 2.0 GB/s bidirectional link bandwidth and 1.5 GB/s bidirectional node bandwidth. The Cray XT3 commercial offering is nearly identical to the XT3 machine installed at Sandia. The notable difference is that while the Red Storm communication topology is a 3-D mesh, the XT3 utilizes a 3-D torus configuration. The difference is to allow a significant portion of the Red Storm machine to switch between classified and unclassified operation. The Cray-designed SeaStar [1] communication processor / router is designed to offload network communication from the main processor. The compute nodes run the Catamount lightweight microkernel, allowing for scalable, low-perturbation operations.

The XT3 platform utilizes the Portals 3.3 communication interface [4], originally developed by Sandia National Laboratory and the University of New Mexico for enabling scalable communication in a high performance computing environment. The Portals interface provides true one-sided communication semantics. Unlike traditional one-sided interfaces, the remote memory address for an operation is determined by the target, not the origin. This allows Portals to act as a building block for high performance implementations of both one-sided semantics (Cray SHMEM) and two-sided semantics (MPI-1 send/receive).

3 Run-Time Environment

The OpenRTE system has proved versatile in common cluster computing environments (including moderately coupled systems, like the Los Alamos BProc cluster environment), but was a point of concern for the XT3 environment. The XT3 provides a full-featured process start-up and monitoring system for job control, eliminating the need for the `mpirun` command utilized by OpenRTE on other platforms. `mpirun` acted as a centralized point of contact for the global data store and resource manager subsystems, and the loss of `mpirun` required significant modification to these subsystems.

The resource management subsystem is largely concerned with discovering resources and starting, monitoring, and controlling MPI processes. As the XT3 operating environment provides this functionality, the subsystem is largely irrelevant for this port. The existing component implementing the RMGR component attempts to load a large number of additional component frameworks at initialization, increasing the memory footprint of Open MPI. Many of the components within those frameworks also utilized functionality not available in the compute node environment, such as `fork` and `exec`. Therefore, we implemented a new resource management component for the XT3 compute node environment that returns an error for most requests (such as process start-up or monitoring). In order to implement MPI_ABORT, the resource manager implementation does use the internal `killrank` function to

shut down the proper processes.

The global data store, known as the general purpose registry (GPR), provides database-like semantics for inter-process communication. The MPI layer utilizes the GPR for storing process contact information (the TCP address and port number for the TCP communication device, for example), host information for heterogeneous environments, and contact information necessary for implementing MPI-2 dynamic process features. The GPR is also used extensively by the RMGR subsystem to track job status and resource availability. The MPI layer generally uses the GPR in a callback model, asking for updates on information as it becomes available. For example, a MPI process might express interest in the architecture information of all nodes on which a process in that job is executing. By default, it would assume that the remote architecture is the same as the local architecture and adjust as new information becomes available.

A new General Purpose Registry (GPR) component — NULL — was added in order to handle the lack of a central `mpirun` process that is found in most Open MPI environments. The NULL GPR component acts as an information sink for processes – information updates succeed, but aren't broadcast. Similarly, requests for information updates succeed, but no callbacks are ever triggered. Because the MPI layer is designed to have reasonable default values if no new information is received, this strategy works well on the XT3. The Portals communication driver was designed to bypass the GPR for wire-up information, eliminating the one place the MPI layer used the GPR in a non-callback mode.

The out-of-band messaging (RML) subsystem of OpenRTE provides send/receive point-to-point, barrier, and broadcast communication outside of MPI channels. It is generally used for process wire-up and relaying updates from the GPR as new information on the state of the run-time environment becomes available. The existing implementation utilized TCP/IP for communication, making it impractical for the XT3 environment. A new out-of-band messaging layer that provides no functionality other than calling `cnos_barrier()` to implement an out-of-band barrier. The RML barrier functionality is utilized during MPI_INIT for process synchronization, motivating the XT3-specific RML implementation.

One new framework was required in order to support the XT3. The System Discovery Service (SDS) was implemented to provide a component infrastructure for providing the starting process both its current OpenRTE name[1] and the list of names for the current job. This information is used to determine the contents of MPI_COMM_WORLD. Previously, this information was always specified by the OpenRTE process start-up mechanisms, so there was no need to abstract this functionality into a component framework. Moving to a new component framework allowed for conditional compilation of the components which provide process naming information and, more importantly, the implementation of the CNOS SDS component, which gets job information from the Cray run-time environment.

4 Communication

Open MPI provides a layered architecture for point-to-point communication, shown in Figure 2. The lowest layer, the Byte Transport Layer (BTL) provides active-message-like send and true RDMA put/get semantics. The BTL interface is extremely small, consisting of eleven functions and has no concept of MPI structures or semantics. The BTL Management Layer (BML) provides scheduling and multiplexing of BTL instances, allowing a single BTL to be shared between multiple higher level protocols.[2] The PML implements the point-to-point functions of the MPI interface, with an interface that very closely matches the MPI interface (synchronous, buffered, and ready sends are converted into an argument field, rather than separate interface functions). There are currently two PML implementations: OB1, which provides high performance, RDMA message transfer and DR, which provides message reliability and NIC fail over. Both provide message stripping and fragmenting for multi-NIC environments. As the XT3 provides end-to-end message reliability, we focus on the OB1 PML for the remainder of this paper.

Given the ease of mapping Portals functionality to MPI semantics, it was not clear whether it was better to implement Portals communication support at the PML or BTL level. The decision was made to implement Portals support at the BTL level for

[1] All OpenRTE processes are assigned a unique name during initialization.

[2] Presently, the MPI-2 one-sided interface is also implemented over the BML/BTL interface. It is likely that collectives will (optionally) bypass the PML for some optimizations in the near future.

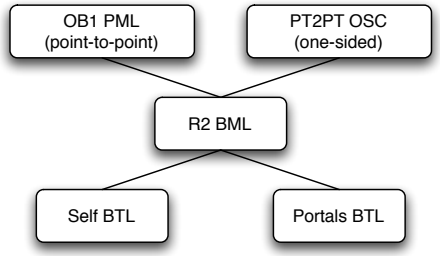

Figure 2: Component design for point-to-point communication.

two reasons: it was believed that performance would be competitive with a PML implementation and the MPI-2 one-sided support utilizes the BTL interface, bypassing the PML interface. We did not wish to implement a Portals-specific one-sided implementation on the first porting effort, so the decision was made to implement a Portals BTL.

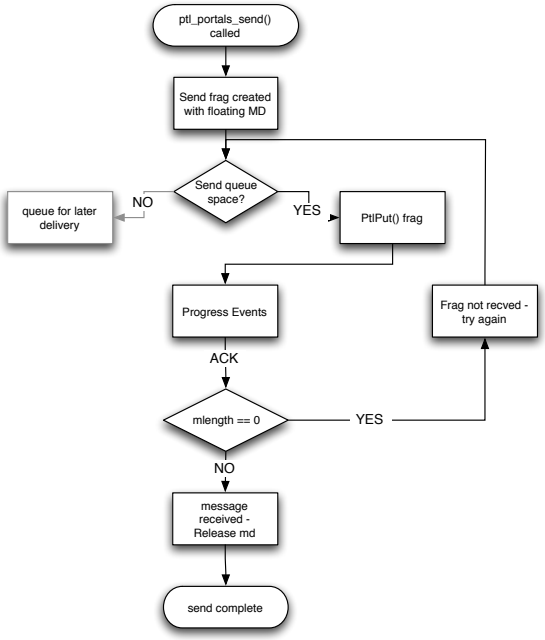

Figure 3: Flow chart of the `btl_send()` logic for Portals.

The BTL interface consists of a number of setup / cleanup functions, and 6 functions required for message transfer. All send/receive communication is in an active-message style – an upper layer expresses interest in a particular message index (in the BTL, this is an integer between 0 and 255) and the send call includes an argument specifying the destination index, in addition to the target process contact information. The communication functions are described below:

alloc Return a region of "btl memory" of a specified size that can be used for `send` communication. On network devices that require per-page pinning, this is generally allocated from a pool of pre-pinned buffers that can be reused to offset pinning costs. The caller is responsible for all data packing, calling `send`, and calling `free` when the send completes.

prepare_src Prepare a specified region of user memory for being the source of a communication call. The region may or may not be contiguous and may require a small area of memory be available in "btl memory" for headers. A datatype convertor is available for determining the memory region to be sent, which may be non-contiguous. The BTL is free to either copy the entire user buffer into "btl memory" or send directly from user memory. `prepare_src` is called for the origin of `send` and `put` calls and the target of `get` calls.

prepare_dst Similar to `prepare_src`, only for the target of `put` calls and the origin of `get` calls. Generally, `prepare_dst` returns an error if the user's memory region can not be sent in a single scatter/gather RDMA operation. A BTL is not required to implement this function if it does not implement `put` or `get`.

free Return resources allocated by `alloc`, `prepare_src`, or `prepare_dst`.

send Send data from a call to `alloc` or `prepare_src` to a specified host. A target index is also provided for dispatch on the receiving side. `send` is a non-blocking call, and a user-specified callback is triggered when the send completes. The BTL can specify the maximum amount of data that can be transferred in any single call to `send`.

put RDMA put operation, called when the origin has been prepared with `prepare_src` and the target with `prepare_dst`. A user-specified callback is triggered on the origin when the transfer completes. No notification is given on the

target. A BTL is not currently required to implement this function.

get RDMA get operation, called when the origin has been prepared with `prepare_dst` and the target with `prepare_src`. A user-specified callback is triggered on the origin when the transfer completes. No notification is given on the target. A BTL is not currently required to implement this function.

The Portals BTL provides both send/receive and RDMA communication. Send communication is limited to 32KB fragments and is implemented utilizing a design similar the method used for unexpected short messages described in [14]. A state diagram of the send logic utilized by the Portals BTL is provided in Figure 3. Unlike most BTL implementations, which require an internal header be transmitted with the user data to send the tag index, the Portals implementation sends the tag index in the `user_data` option of the `PtlPut()` function. The number of outstanding send fragments is limited to ensure there is space in the send message event queue to receive any pending ACK events. As will be discussed later, it is possible that there is no space available on the receiving process for the message. In this case, the sending process will receive an ACK event with an `mlength` of 0, indicating the message was not successfully transmitted (the BTL never sends 0 byte fragments, so `mlength` can never be 0 on a valid transmission). Send completion notification is not given until a valid ACK event is received. The OB1 PML hides the latency of waiting for an ACK in the MPI_SEND case by returning as soon as the `send` call returns, as the data is buffered in the BTL and the user buffer is no longer needed by MPI.

We experimented with the use of iovecs for `send` buffers, allowing the caller to use one buffer for headers and sending the user data directly from user memory. This had an overall negative effect on performance. While the exact cause of the reduced performance is not well understood, we are considering two likely possibilities: performance issues with using iovecs combined with `PtlPut` for short messages or the required delay in completing MPI_SEND calls until the ACK event arrived (as the user buffer is in use by the BTL until the ACK arrives). We intend to investigate this issue further at a later time.

Receiving send messages is done via a set of 1MB memory segments attached to a receive portal table entry. Figure 4 provides an outline of the Portals

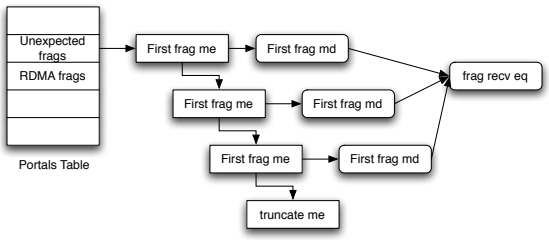

Figure 4: Match list and memory descriptor configuration for receiving message fragments.

structures for receiving `btl_send()` fragments. The `max_data` option is used to ensure the memory descriptor becomes inactive when an entire fragment can not be received. A "reject" match entry / memory descriptor sits as the last entry in the match list, which has no associated event queue and truncates all messages to 0 byte length. This allows the sending process to be notified (via the standard Portals acknowledgment mechanism) that a message was not received and should be retransmitted. When events are pulled from the receive queue, the `hdr_data` field of the event is used for dispatching to the correct receive callback.

The initial implementation limited descriptor activity only with the `max_data` option of the memory descriptor. Because all send fragments are received into one shared receive event queue, it was possible to receive enough fragments to overflow the event queue before the memory descriptors were filled and marked inactive. With the message received but no event to signal the reception, it was nearly impossible to find and deliver the message. The message was delivered successfully into the receive memory descriptor, so the sending process received a ACK event that indicated successful delivery. The receiving BTL was able to detect that the overflow occurred, but at that point lacked a reasonable recovery option. To solve the overflow problem, we now set the `threshold` of each memory descriptor such that the total number of events that may be generated by the receive message descriptors is less than the event queue size. A flood of fragments to one host will cause all receive memory descriptors to go inactive, resulting messages being received to the "reject" descriptor, which truncates the message to 0 bytes. The sender is then notified the message was received but truncated and attempts to retransmit.

RDMA operations are largely a one-to-one map-

ping between the BTL put / get functions and the `PtlPut()` / `PtlGet()` functions. Memory descriptors are created during `btl_prepare_src()` and `btl_prepare_dest()`, and match entries are created and attached to a portals table entry for RDMA operations. The semantics of the BTL descriptor creation means that the target of the RDMA operation can pass a unique 64 bit match key to the origin, allowing the use of Portals matching for RDMA messages. Because there is currently not enough information passed to `btl_prepare_src()` to determine whether the descriptor will be the origin of a put or the target of a get operation, the match entry is always created (the `btl_prepare_dest()` function has the symmetric problem). This causes a slight overhead in creating descriptors for RDMA operations, but that latency is generally masked by the much higher communication latencies required to set up the RDMA protocol used by the OB1 PML.

5 Results

All results presented were run on a Red Storm development test cage at Sandia National Laboratories. Open MPI was built from a Subversion checkout of the development trunk, at revision number 9807. The Cray MPICH2 provided with the currently running software stack was used for comparison.

Figure 5 shows the best case message latency for Open MPI and Cray MPICH2, as well as the best case latency for direct Portals communication. The added overhead of Open MPI's matching header and data copies on both the sender and receiver can be seen. Message latency is still higher than we would expect when compared to other supported networks, so short message latency is still under investigation for possible performance improvements. Some latency-harming debugging code is always on in the current implementation, although we don't believe it explains the large performance difference we are seeing relative to MPICH2.

Implementation	1 Byte Latency
Portals	5.30 μsec
MPICH-2	7.14 μsec
Open MPI	8.50 μsec

Figure 5: Latency for one byte messages using NetPIPE.

Figure 6 shows unidirectional bandwidth from NetPIPE, again comparing Open MPI, Cray MPICH2, and raw Portals communication. Small message performance is slightly lower than MPICH2 and native Portals, for reasons discussed in the previous section. Messages up to approximately 64KB are always copied at both the sender and receiver with Open MPI, which appears to account for the progressively lower relative bandwidth when compared to MPICH2 or raw Portals. At 64 KB, Open MPI begins to use an RDMA get protocol for message transfer. The first portion of the message, along with the source BTL descriptor for the RDMA get is sent. Once the message is matched, the receiving side performs a `PtlGet()` to receive the remainder of the message. A completion acknowledgment is then sent from the receiver to the sender. While peak bandwidth is comparable to MPICH2 and native hardware, the bandwidth for medium sized messages (64 KB - 238 KB) is lower, due to the latencies added by the extra protocol messages.

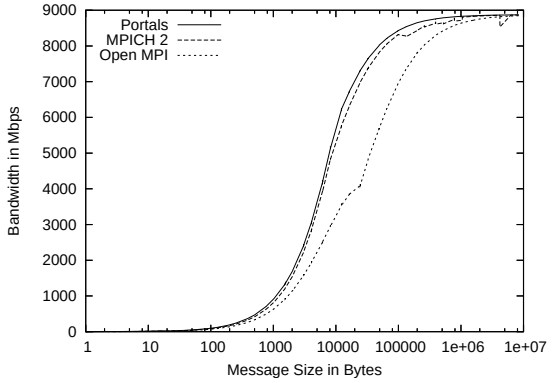

Figure 6: NetPIPE bandwidth on XT3 hardware.

6 Future Work

Open MPI currently provides a complete implementation of the MPI standard for the Cray XT3 environment. However, there are still a number of areas for improvement still under investigation.

6.1 Point-to-point Performance

Currently, all message matching logic is performed in the OB1 PML, at the MPI software layer. The Cray MPICH-2 implementation utilizes the matching capabilities of the Portals interface for message matching. The combination of a lack of data header (instead using the Portals match and ignore bits for

matching information), always sending contiguous data from user memory, and a shorter protocol stack result in lower short message latencies for MPICH-2. Open MPI's OB1 is designed for RDMA networks like InfiniBand or Myrinet/GM, which require a handshake protocol for message transfer setup. The cost of the extra handshake logic is evident in the mid-size message transfer, where Open MPI has lower bandwidth than Cray MPICH-2.

One method of increasing our performance would be to revisit our PML/BTL decision and implement a Portals-specific PML. Initial work has begun on such a configuration, although results are not currently available. However, this approach would still require either maintaining the Portals BTL implementation or developing a one-sided implementation for Portals.

Another option currently under investigation is to extend the BTL interface to allow components that provide match semantics similar to Portals to match receive messages in a fashion similar to that employed by Cray MPICH-2. This approach is currently favored, as it would also be applicable to Myrinet/MX, which has nearly identical performance issues with Open MPI when compared to the vendor-supported MPICH-2 implementation.

6.2 MPI topology component

The Cray XT3 is the first platform Open MPI supports that has an interesting point-to-point network topology. It appears to be possible (at least, on the Red Storm machine) to find the X,Y,Z coordinates on the mesh to each process. Using this information, we should be able to provide intelligent implementations of the MPI topology functions. A TOPO framework exists for allowing such machine-specific implementations, and requires implementing the equivalent of MPI_CART_MAP and MPI_GRAPH_MAP. The TOPO framework will automatically include implementations of the other topology functions based on the two base mapping functions.

6.3 Collective Performance

While not strictly an optimization specific to the XT3 architectures, Open MPI's collective performance is under active research and development. The current implementation provides highly optimized algorithms for a number of collectives, in many cases multiple algorithms for a single collective. Algorithm selection is based on either generic defaults or can be customized to a particular hardware configuration by running a profiling application. Although general algorithm improvements will increase collectives performance on the XT3, it is likely that our collectives routines also need to be optimized for the network topology, which is an area of future research.

7 Conclusions

Open MPI has shown the ability to operate in tightly integrated supercomputing environments, such as the Cray XT3. Performance is indicative of a first attempt at integrating into a new communication paradigm, and can be improved greatly by the addition of hardware matching and the reduction of protocol overhead in the OB1.

Thanks

This work was supported by a grant from the Lilly Endowment, National Science Foundation grants ANI-0330620 and EIA-0202048, and University of California (Los Alamos National Lab) subcontract number 15043-001-05.

References

[1] Robert Alverson. Red storm. In *Invited Talk, Hot Chips 15*, 2003.

[2] Rob T. Aulwes, David J. Daniel, Nehal N. Desai, Richard L. Graham, L. Dean Risinger, Mitchel W. Sukalski, Mark A. Taylor, and Timothy S. Woodall. Architecture of LA-MPI, a network-fault-tolerant mpi. In *Los Alamos report LA-UR-03-0939, Proceedings of IPDPS*, 2004.

[3] B. Barrett, J. M. Squyres, A. Lumsdaine, R. L. Graham, and G. Bosilca. Analysis of the component architecture overhead in open mpi. In *Proceedings, 12th European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.

[4] Ron Brightwell, Tramm Hudson, Arthur B. Maccabe, and Rolf Riesen. The portals 3.0 message passing interface. Technical Report

SAND99-2959, Sandia National Laboratories, 1999.

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

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

[7] G. E. Fagg, A. Bukovsky, and J. J. Dongarra. HARNESS and fault tolerant MPI. *Parallel Computing*, 27:1479–1496, 2001.

[8] Graham E. Fagg, Edgar Gabriel, Zizhong Chen, Thara Angskun, George Bosilca, Antonin Bukovski, and Jack J. Dongarra. Fault tolerant communication library and applications for high perofrmance. In *Los Alamos Computer Science Institute Symposium*, Santa Fe, NM, October 27-29 2003.

[9] E. Garbriel et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 2004.

[10] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir. MPI-2: Extending the Message-Passing Interface. In *Euro-Par '96 Parallel Processing*, pages 128–135. Springer Verlag, 1996.

[11] 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.

[12] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI — The Complete Reference: Volume 2, the MPI-2 Extensions*. MIT Press, 1998.

[13] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proc. of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, November 1993.

[14] Rolf Riesen Ron Brightwell, Arthur B. Maccabe. Design, implementation, and performance of mpi on portals 3.0. *International Journal of High Performance Computing Applications*, 17(1), Spring 2003.

[15] Marc Snir, Steve W. Otto, Steve Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, 1996.

[16] Jeffrey M. Squyres and Andrew Lumsdaine. The component architecture of open MPI: Enabling third-party collective algorithms. In Vladimir Getov and Thilo 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] J.M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, Venice, Italy, September 2003. Springer-Verlag.

[18] T.S. Woodall et al. Open MPI's TEG point-to-point communications methodology : Comparison to existing implementations. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 2004.

A Building Open MPI

Open MPI uses the GNU Autotools for the build process. A number of special arguments are required to properly build Open MPI for the Cray XT3 environment. The required options to successfully build Open MPI are given below.

```
./configure CC=qk-gcc CXX=qk-pgCC \
  F77=qk-pgf77 FC=qk-pgf90 \
  --build=x86_64-unknown-linux-gnu \
  --host=x86_64-cray-linux-gnu \
  --with-platform=redstorm
```

If a Subversion checkout is used to build the source (rather than a tarball), the option --disable-debug should also be specified to turn off internal debugging code that is useful to developers but can have a large impact on performance. Debugging is automatically disabled if built from a tarball.

Open MPI 1.1, when released, will have preliminary support for the XT3 platform. However, numerous enhancements to both the run-time support and the Portals device driver have occurred on the development trunk and will not be part of the Open MPI 1.1 series of releases. Open MPI 1.2 (currently, the development trunk) will provide much better support for the XT3 environment.

About the Authors

Brian Barrett is a PhD Candidate, Department of Computer Science, Indiana University. He is the lead developer of the LAM/MPI project and a developer on the Open MPI project and his research interests include programming paradigms for large scale parallel programming. He can be reached at 215 Lindley Hall, Bloomington, IN, 47405, USA, E-mail: brbarret@osl.iu.edu. Ron Brightwell is Principal Member of Technical Staff, Sandia National Laboratories. His research interests include high-performance, scalable communication interfaces and protocols for system-area networks, operating systems for massively parallel processing machines, and parallel program performance analysis libraries and tools. He can be reached at PO Box 5800, Albuquerque, NM 87185-1110, USA, E-mail: rbbrigh@sandia.gov. Jeff Squyres is Technical Lead for MPI at Cisco Systems. He can be reached at 12910 Shelbyville Rd, Suite 210, Louisville, KY 40243, E-mail: jsquyres@cisco.com. Andrew Lumsdaine is Professor, Computer Science Department and Director of the Open Systems Laboratory at Indiana University. He can be reached at 215 Lindley Hall, Bloomington, IN, 47405, USA, E-mail: lums@osl.iu.edu.