Implementing MPI on Windows: Comparison with Common Approaches on Unix*

Jayesh Krishna[1], Pavan Balaji[1], Ewing Lusk[1], Rajeev Thakur[1], and Fabian Tiller[2]

[1] Argonne National Laboratory, Argonne, IL 60439
[2] Microsoft Corporation, Redmond, WA 98052

Abstract. Commercial HPC applications are often run on clusters that use the Microsoft Windows operating system and need an MPI implementation that runs efficiently in the Windows environment. The MPI developer community, however, is more familiar with the issues involved in implementing MPI in a Unix environment. In this paper, we discuss some of the differences in implementing MPI on Windows and Unix, particularly with respect to issues such as asynchronous progress, process management, shared-memory access, and threads. We describe how we implement MPICH2 on Windows and exploit these Windows-specific features while still maintaining large parts of the code common with the Unix version. We also present performance results comparing the performance of MPICH2 on Unix and Windows on the same hardware. For zero-byte MPI messages, we measured excellent shared-memory latencies of 240 and 275 nanoseconds on Unix and Windows, respectively.

1 Introduction

Historically, Unix (in its various flavors) has been the commonly used operating system (OS) for high-performance computing (HPC) systems of all sizes, from clusters to the largest supercomputers. In the past few years, however, Microsoft Windows has steadily increased its presence as an operating system for running HPC clusters, particularly in the commercial arena. Commercial applications in areas such as computational fluid dynamics, structural analysis, materials, industrial design and simulation, seismic modeling, and finance run on Windows clusters. Windows has also made inroads at the very high end of the spectrum. For example, the Dawning 5000A cluster at the Shanghai Supercomputer Center with 30,720 cores and running Windows HPC Server 2008 achieved more than 200 TF/s on LINPACK and ranked 10th in the November 2008 edition of the Top500 list [16].

Since the vast majority of HPC applications use MPI as the programming model, the use of Windows for HPC clusters requires an efficient MPI implementation. Given the historical prevalence of Unix in the HPC world, however,

* This work was supported in part by a grant from Microsoft Corp. and in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

the MPI developer community tends to have more expertise in implementing and tuning MPI on Unix platforms. In this paper, we discuss some of the issues involved in implementing MPI on Windows and how they differ from commonly used approaches for Unix. We particularly focus on issues such as asynchronous progress, process management, shared-memory access, and threads.

The MPICH implementations of MPI (both the older MPICH-1 and the current MPICH2 implementations) have supported both Unix and Windows for many years. Here we describe how we implement MPICH2 to support both Unix and Windows efficiently, taking advantage of the special features of Windows while still maintaining a largely common code base. We also present performance results on the Abe cluster at the National Center for Supercomputing Applications (NCSA, University of Illinois), where we ran MPICH2 with Unix and Windows on the same hardware.

The rest of this paper is organized as follows. In Section 2, we provide an overview of MPICH2 and its internal architecture. In Section 3, we discuss some of the differences in implementing MPI on Windows and Unix. Performance results are presented in Section 4. We discuss related work in Section 5 and conclude in Section 6 with a brief look at future work.

2 Background

MPICH2 [8] is a high-performance and widely portable implementation of MPI. It supports the latest official version of the MPI standard, MPI 2.2 [7]. MPICH2 has a modular architecture that is designed to make it easy to plug in new network devices, process managers, and other tools (see Figure 1). This design enables anyone to port MPICH2 easily and efficiently to new platforms.

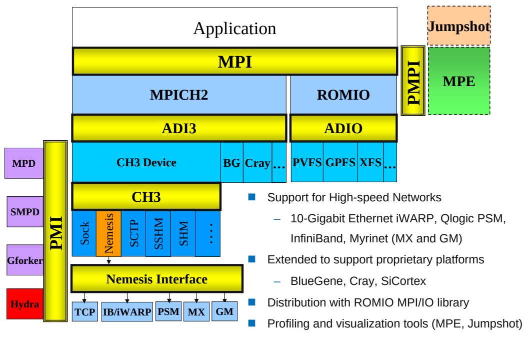

Application
MPI
MPICH2
ROMIO
ADI3
ADIO
CH3 Device
BG
Cray
...
PVFS
GPFS
XFS
...
CH3
Sock
Nemesis
SCTP
SSHM
SHM
...
Nemesis Interface
TCP
IB/iWARP
PSM
MX
GM
PMI
MPD
SMPD
Gforker
Hydra
PMPI
Jumpshot
MPE
Support for High-speed Networks
10-Gigabit Ethernet iWARP, Qlogic PSM, InfiniBand, Myrinet (MX and GM)
Extended to support proprietary platforms
BlueGene, Cray, SiCortex
Distribution with ROMIO MPI/IO library
Profiling and visualization tools (MPE, Jumpshot)

Fig. 1. MPICH2 architecture

A key feature of MPICH2 is a scalable, multinetwork communication subsystem called Nemesis [3]. Nemesis offers very low-latency and high-bandwidth communication by using efficient shared memory operations, lock-free algorithms, and optimized memory-copy routines. As a result, MPICH2 achieves a very low shared-memory latency of around 240–275 ns. We have developed an efficient implementation of Nemesis for Windows.

An MPI library typically requires thread services (e.g., thread creation, mutex locks), shared-memory services (for intranode communication), internode communication services (e.g., TCP/IP sockets), and OS process-management services. The APIs for these features can differ among operating systems and platforms. For portability, MPICH2 uses an internal abstraction layer for these services, which can be implemented selectively on different OS platforms. MPICH2 also includes a portable library for atomic operations, called OPA (Open Portable Atomics) [14], which provides OS-independent atomic primitives, such as fetch-and-increment and compare-and-swap. In addition, we have developed a runtime system that can launch MPI jobs on clusters with any flavor of Unix or Windows.

As a result, MPICH2 can run efficiently on both Windows and Unix operating systems while maintaining a largely common code base.

3 Implementing MPI on Windows versus Unix

From the MPI perspective, Windows and Unix are just different OS flavors, providing similar operating system services. However, a high-performance implementation of MPI on the two OS flavors can differ significantly. These differences can make building a widely portable, high-performance library a huge challenge. In this section, we discuss some of the functionality differences between the two operating systems and the corresponding challenges and benefits.

3.1 Asynchronous Progress

Windows supports an asynchronous model of communication, in which the user initiates an operation and the operating system ensures progress on the operation and notifies the user when the operation is completed. In Nemesis on windows we provide asynchronous internode communication by using an I/O completion object, exposed by the OS as a *completion port*, with TCP/IP sockets. To initiate communication, Nemesis posts a request to the kernel for the operation and waits for a completion event. When the request is completed, the kernel queues a completion packet on the completion port associated with the I/O completion object.

MPI implementations on Unix systems typically use *nonblocking* progress to implement internode communication using TCP/IP sockets. In this case, the library polls for the status of a socket and processes the requested/pending operation. Nonblocking progress differs from asynchronous progress, in which the OS performs the requested operation on the user's behalf. Nonblocking progress is typically implemented on Unix systems by using the POSIX `poll` system call.

Nonblocking progress is generally inefficient compared with asynchronous progress because of deficiencies in `poll`. It also requires the MPI implementation to do more work than with asynchronous progress where some work is offloaded to the operating system. The `poll` system call requires the set of socket descriptors to be polled to be contiguous in memory. This restriction increases bookkeeping and reduces scalability of libraries that allow for dynamic connections or that optimize memory allocated for the socket descriptors by dynamically expanding it. When `poll` returns, indicating the occurrence of an event, the user must search through the entire set of descriptors to find the one with the event. Some operating systems provide event-notification mechanisms similar to completion ports on Windows, for example, `epoll` in Linux and `kqueue` in BSD.

However, these mechanisms are not widely portable across the various flavors of Unix. In Nemesis, we use an asynchronous internode communication module for Windows and a nonblocking internode communication module for Unix systems.

Another MPI feature that can take advantage of asynchronous services is generalized requests, which allow users to define their own nonblocking operations that are represented by MPI request objects. MPI specifies that the user is responsible for causing progress on generalized requests. On Unix, the user may be required to use an external thread. Windows allows users to register callback functions with asynchronous OS calls; this mechanism allows a user library to use generalized requests without needing an external thread to cause progress on the operations.

3.2 Process Management

Process management in MPI typically involves providing a mechanism to launch MPI processes and setting the appropriate runtime environment for the processes to be able to connect to each other.

Launching MPI Jobs On Unix systems that do not have an external job launcher, MPI process managers typically use `fork` to launch processes locally and network protocols such as SSH to launch processes remotely. These network-protocol agents assume the existence of a standalone daemon process on each node that can interact with the remote protocol agent. Since Windows does not natively provide a network-protocol mechanism similar to SSH, we need a distributed process-management framework with standalone manager daemons on each Windows node.

When launching MPI jobs, the remote MPI processes must be launched with the user's credentials. When using a network protocol such as SSH, the protocol provides this service. On Windows, however, the process-manager daemon must do this job. In MPICH2, we have implemented a process manager, called SMPD, that provides process-management functionalities to MPI jobs on both Windows and Unix systems. On Windows, the standalone SMPD daemon impersonates the user launching the job by using the user's credentials. Where available, SMPD can also use technologies such as Active Directory and the job scheduler in Windows HPC Server 2008 to manage user credentials and launch MPI jobs.

Managing MPI Processes Once the MPI processes are launched, the process manager is responsible for managing them. It must provide information to the processes so that they can connect to each other; handle `stdin`, `stdout`, and `stderr`; and handle termination and shutdown. SMPD provides these features using a communication protocol that is independent of the data model used by the individual nodes of a cluster. This allows users to run MPI jobs on a heterogeneous cluster containing both Unix and Windows nodes.

3.3 Intranode Communication

MPI implementations typically use some form of shared-memory communication for communicating between MPI processes running on a single node. Nemesis uses lock-free shared-memory queues for improving scalability and reducing overhead for intranode communication [3]. The use of these queues reduces the intranode communication latency for small messages and is particularly effective when the communicating processes share CPU data caches. When they do not, however, the performance often degrades.

An MPI implementation can also use OS services that allow users to transfer data directly between the memories of two processes. This approach can improve performance for large-message transfers among processes that do not share a cache. A variety of standard and nonstandard methods for doing so are available on Unix [2]. Windows provides an OS service for directly accessing the address space of a specified process, provided the process has appropriate security privileges. For small messages, however, we observed that this service has more overhead than the lock-free shared-memory queues in Nemesis. Therefore, we use the remote-copy method only for large messages in Nemesis on Windows.

3.4 Threads

The MPI standard clearly defines the interaction between user threads and MPI in an MPI program [5]. The user can request a particular level of thread support from the MPI implementation, and the implementation can indicate the level of thread support provided. On both Windows and Unix, MPICH2 supports the `MPI_THREAD_MULTIPLE` level, which allows any user thread to make MPI calls at any time. This feature requires some thread-locking mechanisms in the implementation in order to make it thread safe.

Unix platforms typically use a POSIX threads (Pthreads) library, whereas Windows has its own version of threads. MPICH2 uses an OS-independent thread-abstraction layer that enables it to use different threads libraries and thread-locking mechanisms portably. The default version of MPICH2 uses a global lock to control access to an MPI function from multiple threads. A thread calling an MPI function tries to obtain this global lock and then releases the lock after completing the call or before blocking on an OS request. When a lock is released, a thread waiting for the global lock gets access to the lock and performs progress on its MPI communication. We are also developing a more efficient version of MPICH2 that supports finer-grained locks [1].

4 Experimental Evaluation and Analysis

In this section, we evaluate the different strategies discussed in the paper and compare the results. We ran all tests on the Abe cluster at NCSA, which has both Unix and Windows nodes. Each node consists of 2 quad-core Intel 64 (Clovertown) 2.33 GHz processors with a 2x4 MB L2 cache, 4x32 K L1 cache, and 8 GB

RAM. The Unix nodes ran Linux 2.6.18 and used Intel C/C++ 10.1 compilers. The Windows nodes were installed with Windows Server 2008 HPC Edition SP2 and the Visual Studio 2008 compilers. On both Unix and Windows, we compiled MPICH2 with aggressive optimization and disabled error checking of user arguments. The interconnection network used was gigabit Ethernet.

4.1 Asynchronous Progress

We compared asynchronous and nonblocking progress by calculating the amount of overlap of communication and computation with the two strategies. To measure the performance of nonblocking progress on Windows, we implemented a version of Nemesis that uses the `select` system call since it is more portable across various versions of Windows than `poll`. The default version of Nemesis on Windows uses asynchronous progress with I/O completion ports.

We measured communication latency and bandwidth by using an MPI version of the popular NetPIPE benchmark [11], called NetMPI, which performs a nonblocking receive, a blocking send, and an `MPI_Wait` multiple times in a loop. We modified the benchmark to perform several nonblocking sends and receives at a time and used `MPI_Testall` to test for their completion without blocking. Between calls to `MPI_Testall`, we performed some computation for ≈250 ns. The less time spent by the MPI implementation in `MPI_Testall`, the more time available to the user to perform computation.

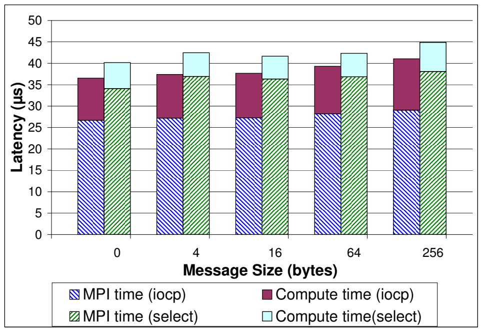


Fig. 2. Time spent in the MPI library (MPI time) and time available for user computation (Compute time) when using asynchronous progress (iocp) versus nonblocking progress (select) on Windows for internode communication

Figure 2 shows the breakdown of the time spent within the MPI library and the time available for the user's computation when sending a message by using asynchronous progress versus nonblocking progress. As expected, asynchronous progress results in less time being spent within the MPI library and more time available for user computation. The reason is that with asynchronous progress the MPI library delegates the reading/writing of data from/to TCP/IP sockets to the OS, whereas with nonblocking progress the library polls for events and then performs the read/write. When I/O is delegated to the OS, the library has little work to do and quickly returns to the application.

4.2 Intranode Communication

In this experiment, we compared the intranode communication performance using lock-free shared-memory queues and direct remote-memory access for large messages. We also compare the intranode performance of MPICH2 on Windows and Unix for small and large messages. We used the Ohio State University (OSU) microbenchmarks [15] to measure latency and bandwidth in all cases.

We have implemented a read remote virtual memory (RRVM) module in Nemesis that performs remote memory access for large messages (≥16 KB) on Windows. Figure 3 shows the intranode communication bandwidth when using lock-free shared-memory queues versus RRVM on Windows. We considered two cases, one where the communicating processes shared a 4 MB L2 cache and another case where the processes were launched on cores that do not share a data cache. We observe that the shared-memory queues perform better than the RRVM module for some message sizes when the processes share a data cache. However the RRVM module delivers a better overall performance because it performs significantly better when the processes don't share a cache.

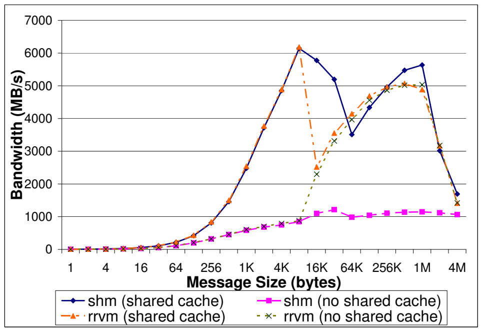


Fig. 3. Intranode communication bandwidth using shared-memory queues (shm) versus direct copy (rrvm) on Windows

The jagged graph in the shared-cache case is because at 16 KB, the double-buffering scheme used for communication runs out of L1 cache. The performance again improves from 128 KB because Nemesis switches to a different protocol that allows pipelining of messages. The bandwidth then drops at 2 MB because the double-buffering scheme runs out of L2 cache. We will investigate whether tuning some parameters in Nemesis can help smoothen the curve.

Figure 4 shows the intranode communication latency for small messages and bandwidth for large messages on Windows and Unix. The latency results on the two operating systems are excellent (240 ns on Unix and 275 ns on Windows for zero-byte MPI messages) and comparable (only ≈35 ns apart for small messages). For small messages, we use lock-free shared-memory queues for intranode communication on both systems. Therefore, we observe a performance degradation on both Unix and Windows for small messages when the communicating processes do not share a cache. For large messages, the bandwidth on Unix degrades substantially when the processes do not share a cache, whereas on Windows the performance is good in both cases because of the use of direct copy.

4.3 Internode Communication

We also studied MPI internode communication performance using TCP on both Windows and Unix. We measured the latency and bandwidth for internode communication by using the OSU microbenchmarks. Figure 5 shows the results.

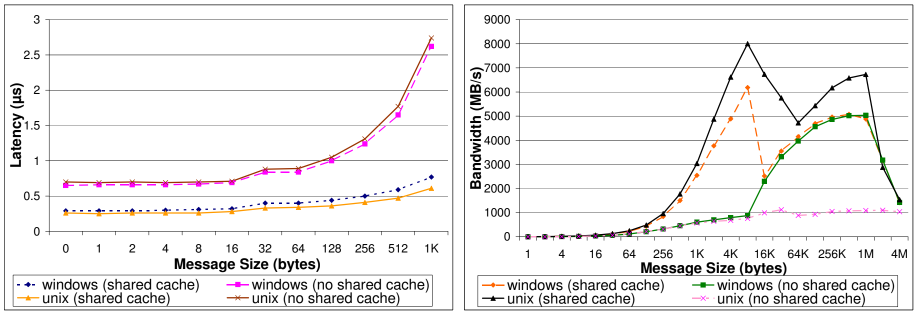


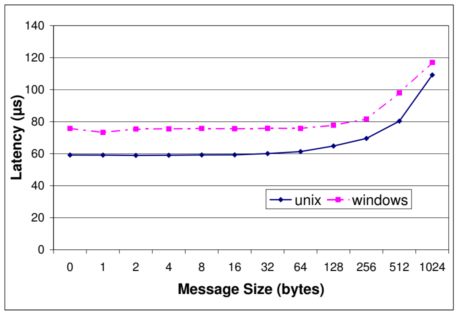
Fig. 4. Intranode communication latency and bandwidth on Windows and Unix

We observe that, in these experiments, MPICH2 on Unix performs better than MPICH2 on Windows, with respect to both latency and bandwidth. We are investigating the cause of the difference, but we expect that further tuning and optimization of the Nemesis TCP module for Windows will eliminate the performance gap. We note that these microbenchmarks measure only point-to-point communication performance, whereas overall application performance depends also on the scalability of the underlying communication subsystem and the ability to overlap communication with computation. We expect the Windows version to have good overall application performance because the asynchronous model with completion ports is scalable and supports overlap. To verify this, we plan to conduct experiments with applications as described in Section 6.

4.4 Cost of Supporting Thread Safety

The MPI library incurs some overhead to support multiple user threads. To measure this overhead, we modified the OSU micro-benchmark to use `MPI_Init_thread` instead of `MPI_Init` and measured the latency for intranode communication with `MPI_THREAD_SINGLE` and `MPI_THREAD_MULTIPLE`. Note that, in both cases, the program has only one thread, but in one case support for multithreading is disabled and in the other it is enabled, requiring the implementation to acquire thread locks in case multiple threads make MPI calls.

Figure 6 shows the overhead as the percentage increase in latency over the `MPI_THREAD_SINGLE` case when multithreading is enabled. We observe that the overhead is significantly lower on Windows than on Unix. The Unix version uses Pthread mutex locks for thread safety; the Windows version uses intraprocess locks (critical sections). We note that, on Windows, we initially used interprocess thread locks (mutexes), but their performance was much worse. By switching to intraprocess locks (critical sections), the performance improved significantly.



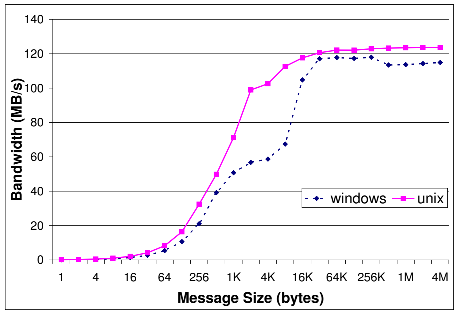


Fig. 5. Internode MPI communication latency and bandwidth on Windows and Unix using Nemesis over TCP/IP

Intraprocess locks are sufficient for Nemesis because it uses lock-free shared-memory queues for interprocess communication.

5 Related Work

Although MPI implementations have traditionally been developed on Unix, several MPI implementations are now available on Windows. Microsoft and Intel have developed MPI implementations for Windows [6, 10], which are both derived from MPICH2. DeinoMPI [4] is another implementation of MPI, derived from MPICH2, for Windows. In addition, Open MPI has recently added support for Windows [13]; and MPI.NET [9] is an implementation that provides C# bindings for Microsoft's .NET environment.

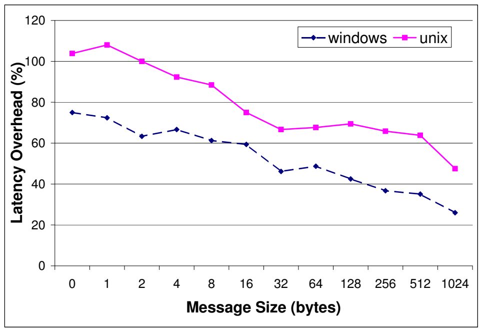


Fig. 6. Overhead in intranode MPI communication latency on Windows and Unix when multithreading is enabled

6 Conclusions and Future Work

We have discussed several issues in implementing MPI on Windows and compared them with approaches on Unix. We have also discussed how we implemented MPICH2 to exploit OS-specific features while still maintaining a largely common code base. Performance results with both Windows and Unix on the same hardware demonstrate that the performance of MPICH2 on both operating systems is comparable. We observed some difference in internode communication performance, which we plan to investigate and optimize on Windows. MPICH2 takes advantage of the asynchronous communication features in Windows, which enable applications to overlap communication with computation.

Windows HPC Server 2008 introduced a new low-latency RDMA network API, called Network Direct [12], that enables applications and libraries to use the advanced capabilities of modern high-speed networks, such as InfiniBand. We plan to implement a Nemesis module for Network Direct and study the performance of MPICH2 with high-speed networks on Windows. We also plan to evaluate application-level performance with MPICH2 on Windows, including commercial MPI applications at large scale.

References

1. Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Thakur, R.: Fine-grained multithreading support for hybrid threaded MPI programming. International Journal of High Performance Computing Applications 24(1), 49–57 (2010)
2. Buntinas, D., Goglin, B., Goodell, D., Mercier, G., Moreaud, S.: Cache-efficient, intranode, large-message MPI communication with MPICH2-Nemesis. In: Proc. of the 2009 International Conference on Parallel Processing. pp. 462–469 (2009)
3. Buntinas, D., Mercier, G., Gropp, W.: Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem. In: Proc. of 6th IEEE/ACM Int'l Symp. on Cluster Computing and the Grid (CCGrid) (May 2006)
4. Deino MPI. `http://mpi.deino.net/`
5. Gropp, W., Thakur, R.: Thread safety in an MPI implementation: Requirements and analysis. Parallel Computing 33(9), 595–604 (September 2007)
6. Intel MPI. `http://software.intel.com/en-us/intel-mpi-library/`
7. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Version 2.2 (September 2009), `http://www.mpi-forum.org`
8. MPICH2 – A high-performance portable implementation of MPI. `http://www.mcs.anl.gov/mpi/mpich2`
9. MPI.NET: A high performance MPI library for .NET applications. `http://osl.iu.edu/research/mpi.net/`
10. Microsoft MPI. `http://msdn.microsoft.com/en-us/library/bb524831(VS.85).aspx`
11. NetPIPE: A network protocol independent performance evaluator. `http://www.scl.ameslab.gov/netpipe/`
12. Network Direct: A low latency RDMA network API for Windows. `http://msdn.microsoft.com/en-us/library/cc904344(v=VS.85).aspx`
13. Open MPI. `http://www.open-mpi.org`
14. Open Portable Atomics library. `https://trac.mcs.anl.gov/projects/openpa/wiki`
15. OSU Micro-Benchmarks (OMB). `http://mvapich.cse.ohio-state.edu/benchmarks/`
16. Top500 list. `http://www.top500.org/lists/2008/11` (November 2008)