The Architecture and Performance of WMPI II

Anders Lyhne Christensen[1], João Brito[1], and João Gabriel Silva[2]

[1] Critical Software SA, {alc,jbrito}@criticalsoftware.com
[2] DEI/CISUC University of Coimbra, Portugal, jgabriel@dei.uc.pt

Abstract. WMPI II is the only commercial implementation of MPI 2.0 that runs on both Windows and Linux clusters. It evolved from the first ever Windows version of MPI, then a port of MPICH, but is now fully built from its own code base. It supports both 32 and 64 bit versions and mixed clusters of Windows and Linux nodes. This paper describes the main design decisions and the multithreaded, non-polling architecture of WMPI II. Experimental results show that, although WMPI II has figures comparable to MPICH and LAM for latency and bandwidth, most application benchmarks perform significantly better when running on top of WMPI II.

1 Introduction

MPI is the de-facto standard for writing high performance applications based on the message-passing paradigm for parallel computers such as clusters and grids, [MPIR1]. The first version of MPI was published in 1994 by the MPI Forum, [MPIS1], a joint effort involving over 80 people from universities, hardware and software vendors and government laboratories. The initial version of the Message Passing Interface specifies functions for point-to-point and collective communication as well as functions to work with data types and structures in heterogeneous environments. The standard has a number of different functions with only slightly different semantics to allow for implementers to take advantage of the special features of a particular platform, like massively parallel computers, clusters of PCs, etc. Naturally, each of these different platforms calls for different communication library designs depending on the underlying hardware architecture. However, even for similar architectures such as the increasingly more popular clusters of PCs, designs can differ considerably, often resulting in radically different performances of real applications. In this paper we take a closer look at the design choices behind WMPI II and how they differ from some other popular implementations, and we present and compare application and micro-benchmark results, for WMPI II, MPICH and LAM [HPI,LAM].

In 1997 the MPI Forum published an ambitious version 2 of the Message Passing Interface Standard (MPI-2), [MPIS2], extending the functionality to include process creation and management, one-sided communication and parallel I/O. These extensions meant that the standard grew from 128 to 287 functions. Whereas a relatively large number of complete MPI-1 implementations exist, only few complete MPI-2 implementations are available, [LAMW]. WMPI II is

one of the few general-purpose MPI-2 implementations for clusters of PCs, and it has by no means been a trivial piece of software to develop. Critical Software has invested more than three years of research and development in the product. We believe that the reason for the relatively limited number of complete MPI-2 implementation is likely due to the complexity of the MPI-2 standard, namely the issues related to extending an existing MPI-1 implementation to support the new MPI-2 features. In MPI-1 each participating process can uniquely be identified in a straightforward manner by its rank in MPI_COMM_WORLD, a static, global communicator valid from startup until shutdown of an application. In this way all processes can have a shared, global view of the MPI environment during the entire run of an application. With the introduction of process creation and management features in MPI-2, this global view assumption is no longer valid since the ability to add and remove processes at runtime has the consequence that each participating process only has a local and partial view of the MPI environment at any given moment.

This article is structured in the following way: In Sect. 2 we give a brief overview of the history of WMPI, in Sect. 3 we explain the main design choices behind WMPI II, and in Sect. 4 we describe the architecture. Sect. 5 discusses our experience with porting from Windows to Linux. Experimental results are presented in Sect. 6 and discussed in Sect. 7. Finally, in Sect. 8 we lay out some lines for the future roadmap for WMPI II.

2 History of WMPI

WMPI started as a research project at the University of Coimbra in 1995. Versions 1.0 to 1.3 of WMPI were based on MPICH and reengineered to run on the Win32 platform. It was the first ever Windows implementation of version 1 of MPI, [MAR].

When the MPI-2 standard was released it became evident that a total rewrite was necessary due to the new functionality related to process creation and management. Version 1.5 consisted of a complete rewrite of WMPI in order to allow for dynamic processes. Virtually all the existing code had to be abandoned, since data structures and associated functions had to be redone to allow for processes to operate with only a partial, local view of the entire application. Although WMPI 1.5 does not have any of the functionality from the MPI 2.0 extensions, its architecture was designed with MPI-2 features in mind.

The first complete implementation of the MPI-2 standard for Windows clusters, WMPI II, was released at the beginning of 2003 and one year later the Linux version of WMPI II followed. WMPI II was based on WMPI 1.5, but extended with all of the functionality of MPI-2, including process creation and management, one-sided communication and parallel I/O. Presently a number of versions exist for various releases of Windows and Linux distributions for 32- and 64 bit systems.

3 Design Choices

With the introduction of MPI-2 it became clear that an entirely new software architecture for WMPI was necessary in order to support the new functionality. This required a significant investment and proved to be a challenging task. It allowed us to rethink a number of fundamental design choices and redesign the core based on our previous experience with WMPI up to version 1.3. We wanted WMPI II to perform well in real applications and be flexible in terms of portability, customization, and extensions. The main goals and consequences of the redesign are listed below:

WMPI II should support multiple, concurrent communication devices: For a cluster of PCs, inter-node communication is done through some type of interconnector, such as Gigabit Ethernet, InfiniBand or Myrinet, whereas processes running on SMP nodes often can benefit from the lower latency and higher bandwidth of shared memory communication. Furthermore, different segments of a cluster can be connected by different types of interconnectors especially as new technologies emerge. Therefore, WMPI II was designed to allow for multiple (two or more) communication devices to be used concurrently. Communication devices are dynamic link libraries (or shared objects in Linux), which are loaded at run-time based on the content of a configuration file. This also enables third parties to implement their own devices thus making WMPI II extendable. In order to allow for multiple communication devices to operate concurrently the Channel Interface from MPICH, [CHA], had to be discarded in favor of a completely new approach with thread-aware callbacks, direct remote memory access and process creation capabilities.

WMPI II should not use polling and be internally multithreaded: Many implementations, such as MPICH, are not internally multithreaded and rely on a singlethreaded and/or polling approach to ensure message progress. This approach requires that the user application frequently calls the MPI implementation in order to ensure timely progress. In micro-benchmarks, e.g. a ping-pong test, where the MPI library is called constantly, this approach works fine as little else is done besides communicating. Keeping computation and communication in the same thread also avoids one or more context switches, which are necessary if these tasks are performed in separate threads, each time a message is sent or received. However, for real applications frequent calls to the message-passing layer are not guaranteed and polling can steal numerous cycles from the application in cases where it attempts to overlap communication and computation. In a multithreaded approach an application can continue doing calculations while an internal thread in the message-passing library takes care of the communication. This is the only approach consistent with the deep design philosophy of MPI of promoting overlapped communication and computation.

A polling/singlethreaded approach is particularly ill-suited for one-sided communication (OSC) applications, where only one party participates explicitly. In these cases the target process does not issue a matching operation and therefore timely progress is not guaranteed.

WMPI II should be thread-compliant: Thread compliance is a feature that is often not supported by open-source implementations. A thread-compliant MPI implementation allows an application to call MPI functions concurrently from multiple threads without restriction, [MPIS2]. Since WMPI II was already required to be internally multithreaded, adding thread compliance was relatively easy. It also gives the user the freedom to develop mixed OpenMP/MPI programs, [OMP], where OpenMP is used for fine-grained parallelism on SMP nodes and MPI is used for coarse-grained parallelism. Furthermore, application-level multithreading is becoming increasingly more important in order to take full advantage of modern dual-/multi-core[3] CPUs capable of running two or more threads concurrently.

WMPI II should be as portable as possible without affecting performance: The initial version of WMPI was exclusively for Windows. Other operating systems, especially Linux, are very popular within the high performance community. Therefore, it was a central design goal to leave the option of porting WMPI II to other operating systems and architectures open, but only to the degree where performance would not be affected. Hence, we did not want to settle for the lowest common denominator at the price of performance, but make it "*as portable as possible*".

WMPI II should be flexible: Naturally portability between operating systems as different as Linux and Microsoft Windows requires flexibility. However, given that WMPI II is a commercial implementation, additional flexibility is needed to allow for custom fitting. For example, some clients have asked for specific functional extensions while others have requested more advanced features such as well-defined fault semantics. The architecture was therefore designed with such extensions in mind.

4 Architecture

The architecture of WMPI II is shown in Fig. 1. At the top sits the language bindings for C, C++ and Fortran, as well as for the profiling interface (PMPI functions). The WML contains the management logic that takes care of implementing complex communication primitives on top of simpler ones, managing datatypes, expected and unexpected messages and requests. Moreover, the WML emulates functionality not directly supported by some devices. For instance the one-sided communication primitives are implemented on top of point-to-point primitives when communicating with another process through TCP/IP or Globus I/O, since this device does not support remote memory access. The WML represents by far the largest part of the code altogether.

The WML operates with two different types of devices: Boot devices, sitting on top of the Boot Engine as shown in the figure, and communication devices located below the Communication Engine. Communication devices implement

[3] Multi-core CPUs: Also called *hyperthreading*, *chip multithreading* and *symmetric multithreading* depending on the hardware vendor.

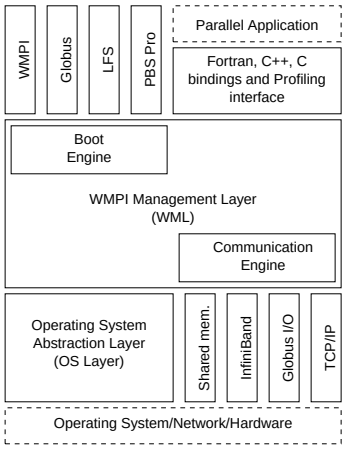

Fig. 1. WMPI II Architecture

a number of primitives such as sends, receives and functions to open and close connections. These devices are used for communication after startup.

Boot devices, on the other hand, are used during startup, when the MPI environment is initialized. In standalone-mode WMPI II takes care of starting processes. This is done through a custom daemon running on each processing node. Information on the environment, e.g. which nodes are participating, who is who, how to contact the different processes etc. is passed through the daemon to new processes. When a new process starts it reads this information and knows what the world looks like. However, WMPI II does not control the starting of processes if applications are run under a scheduler like Platform LSF, PBS Pro or Globusand therefore cannot pass the necessary information to the participating processes. In each of these other modes processes are started outside the control of WMPI II and newly started processes need to rely on the means provided by the scheduler to obtain the necessary information.

In this way the WML was designed to be a component, which is independent of both the underlying communication hardware and software, as well as the surrounding startup environment. Such a design offers a significant level of flexibility since custom startup mechanisms can be added with minimal effort and new boot devices only require local additions. For commercial software vendors, this allows for WMPI II to be embedded in libraries where the startup is either completely or partly controlled by the application software and does not require the application to be started through mpiexec, mpirun or alike.

If a WMPI II application is started through a scheduler, the scheduler controls the environment, such as where processes are started and security issues. If applications are started in native mode, e.g. started by WMPI II, the cluster configuration and security issues are handled by WMPI II. We operate with two different types of configuration: A cluster configuration file and a process group

file. A cluster configuration file lists the nodes available, which communication devices should be used for inter- and intra-node communication, and under which security context processes should run on each node. Whereas the cluster configuration file describes the nodes available, a process group file lists what nodes should run which executables. The two concepts have been separated due to the introduction of process creation and management in MPI-2. With dynamic processes an application might be started as a singleton MPI process running on a single node and then at runtime spawn additional processes on other nodes. In this way, the potential cluster can differ from the active sub-cluster.

Security can be an issue in some cluster environments; moreover in heterogeneous environments security is handled differently on different operating systems. For UNIX-like operating systems a security context consists of a user name and a password, whereas on Windows the security context also comprises a domain. Since the supported operating systems lack common features for starting processes remotely, e.g. rsh, ssh, DCOM, we rely on a custom daemon. In multi-user environments it is often important to enforce permissions in order to avoid unauthorized access to sensitive data and to protect certain parts of a file system against tampering, voluntary or accidentally. Still the WMPI II daemon, responsible for creating new processes, needs to be running as root/administrator, which gives unrestricted access to a node. Therefore, users are required to specify the security context under which they wish to run MPI processes on the cluster nodes. For the same application different contexts can be used on different nodes. When a process is to be started on a node, the daemon on that node switches to the specified security context, provided that it is valid, and starts the process.

5 Porting from Windows to Linux and Interoperability

The first version of WMPI II was exclusively for the Windows platform; however, as mentioned above, it was built with portability in mind. The goal was to make it as portable as possible and to achieve code sharing without compromising performance. Besides from the obvious benefit of having to maintain and extend only one source code tree it also greatly simplifies heterogeneous cluster support.

There are a number of ways to develop portable code, such as settling for the lowest common denominator in terms of features or relying on an operating system abstraction layer, where features present in a certain OS are implemented in the abstraction layer. We chose the latter approach for the WML, while for the devices we largely use OS specific code. An event-model similar to the native Windows event-model was implemented on Linux in the OS abstraction layer, since it proved quite convenient given the multithreaded design. Whereas the OS abstraction layer approach works well for WML, the majority of code for the devices has to be OS-specific in order not to compromise performance. The features used in the devices on Windows and on Linux are quite different, e.g. for the TCP device on Windows a combination of overlapped I/O and completion ports are used, whereas on Linux asynchronous sockets and real-time signals are used. The same is true for the shared memory device: On Windows one process

can read the memory of another running process, whereas on Linux this is not the case. Moreover, cross-process events are used by the shared memory device on Windows whereas real-time signals are used on Linux. Thus, the code for the WML compiles on both operating systems, whereas the devices have been specially optimized for each operating system.

The greatest effort in porting from Windows to Linux was implementing and optimizing the devices for Linux and a large number of tiny compiler and OS differences. All in all it did not require as much effort as anticipated and we have managed to achieve a code sharing of 88%.

6 Experimental Results

In this section we present a number of benchmark results for WMPI II 2.3.0 and compare them with MPICH 1.2.5 on Windows and with MPICH 1.2.5 and LAM 7.0.5 on Linux.

We provide two types of benchmarks: Micro-benchmarks and application benchmarks. Micro-benchmarks attempt to measure the isolated performance of a message-passing library, that is latency and bandwidth, for a number of different MPI functions when nodes do nothing besides communicating. Application benchmarks attempt, as the name suggests, to measure the performance of a message-passing library (and the parallel computer as a whole) under a real application model, e.g. solving a common problem like a dense system of linear equations. For micro-benchmark results we have chosen the PALLAS benchmark, [PAL]. The NAS parallel benchmark version 2.4, [NAS], and HP Linpack, [LIN], have been chosen for application benchmarks given their availability for a large number of operating systems and their popularity in terms of evaluating the performance of parallel computers. Notice that we have used default installations of operating systems and MPI implementations for all benchmarks, which means we did not do any tweaking that could potentially increase and/or decrease the performance of any particular MPI implementation.

6.1 Windows Results

The Windows benchmarks were run on a heterogeneous cluster of 16 Pentium 4 CPUs running at 1.7 – 2.8 GHz and with 512 MB to 1 GB of RAM. The nodes were running different versions of Windows: 2000, XP, and 2003 Advanced Server and they were connected in a 100 Mb/s, switched network. The latency and bandwidth results obtained by running the PALLAS ping-pong benchmark are shown in Fig. 2a and 2b, respectively. The results of running the NAS parallel benchmarks (class B tests) are shown in Fig. 2c, and the Linpack scores are shown in Fig. 2d. The NAS benchmark was compiled with Absoft Pro FORTRAN 7.0 and the Linpack benchmarks were compiled with Microsoft's C/C++ compiler version 13.10.3077.

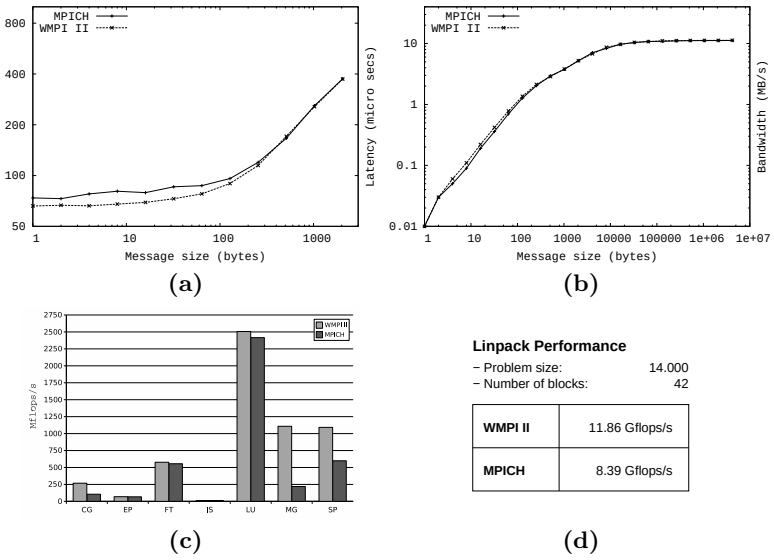

Fig. 2. Windows results for WMPI II and MPICH. **(a)** shows the latency results. **(b)** shows the bandwidth results. **(c)** shows the NAS benchmark results and the table in **(d)** lists Linpack results.

6.2 Linux Results

The Linux benchmarks were run on a cluster of 16 Pentium 4 2.8 GHz nodes, each with 2 GB of memory running Redhat Linux 9.0. The nodes where connected in a 100 Mb/s switched network. As with the Windows benchmark results the latency and bandwidth results on Linux were obtained by running the PALLAS ping-pong benchmark. The results are shown in Fig. 3a and 3b. The application benchmark results are shown in Fig. 3c and 3d for the NAS parallel benchmark class B tests and Linpack, respectively. GCC 3.2 was used to compile all the benchmarks.

7 Discussion

The results for Windows show only little difference in the micro-benchmark results. WMPI II has a latency 10 μs lower for small messages (< 200 bytes) than MPICH (see Fig. 2a), while the differences in terms of bandwidth are only marginal (see Fig. 2b). The raw performance measured by the micro-benchmarks for WMPI II, which relies on a multithreaded approach, and MPICH, which relies on a singlethreaded approach, are thus not significantly dissimilar. The application benchmark results, however, paint a different picture: For some of the NAS parallel benchmarks, EP, FT, and LU (see Fig. 2c), WMPI II and

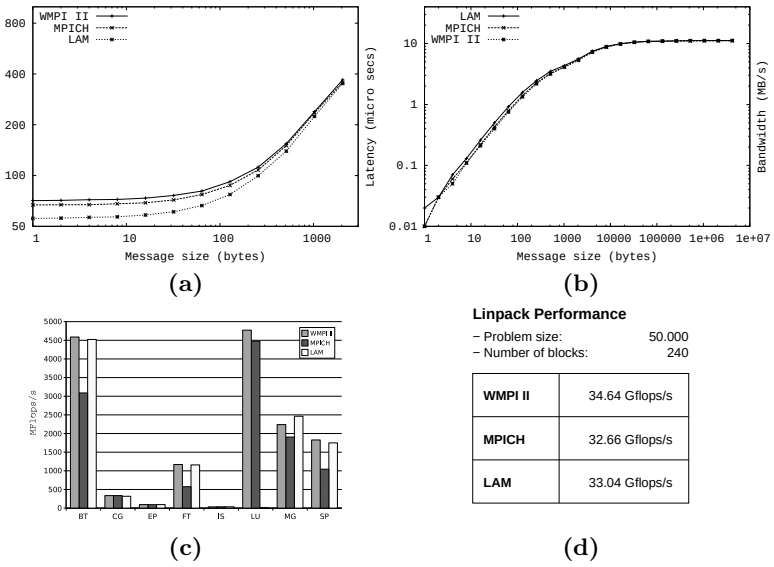

Fig. 3. Linux results for WMPI II, MPICH and LAM. **(a)** shows the latency results. **(b)** shows the bandwidth results. **(c)** shows the NAS benchmark results and the table in **(d)** lists Linpack results.

MPICH have comparable performance, whereas there are major differences in performance for the CG and the MG benchmarks, where using WMPI II yields a performance of 2.5 times and 5.0 times higher than MPICH, respectively. Similarly, the Linpack results show that WMPI II has a performance 41% higher than that of MPICH. This indicates that the ability to overlap computation and communication can be quite advantageous in some applications. Moreover, given the multithreaded, non-polling approach, a WMPI II process will be able to coexist efficiently with other application threads and/or other processes on the same node. We must of course take into consideration that MPICH is relatively old compared to WMPI II; better results for MPICH2 should be expected when stable versions for Windows become available.

The Linux results show that WMPI II has a latency that is slightly worse than MPICH and LAM in particular, while the bandwidth performances for all three implementations are comparable, see Fig. 3a and 3b. For application benchmarks the performance for the three implementations does not vary as much when running on Linux as it is the case for WMPI II and MPICH on Windows. For most of the NAS parallel benchmarks WMPI II and LAM have similar performance while MPICH falls behind on the BT, FT, and SP benchmarks, see Fig. 3c. WMPI II performs 1%-6% better than LAM on the BG, CG, FT and SP benchmarks, while LAM performs 9% better than WMPI II on the MG benchmark. No results are shown in the LU benchmark for LAM, because LAM kept

exiting with an error message when we tried to run this benchmark. The results of the Linpack benchmark on Linux in Table 2 shows that WMPI II is slightly faster than both MPICH and LAM, achieving 34.64 Gflops/s compared to 32.66 Gflops/s and 33.04 Gflops/s, respectively.

It is important to stress that the benchmarks for Linux and Windows should not be directly compared, particularly because the latter were run on a cluster with heterogeneous hardware, where the slower machine tends to set the pace for the whole computation for some problems, such as those solved by Linpack.

Based on our results we can conclude that micro-benchmark results, such as raw latency and bandwidth, are not accurate indicators of the performance of an MPI implementation in real applications. Furthermore, our results show that some applications can benefit significantly from a multithreaded approach in the message-passing layer. Moreover, we believe that it is inherent in the MPI standard that implementations should not rely on polling or on a singlethreaded approach. An added benefit of WMPI II's multithreaded architecture is that it can be used for desktop GRIDs/idle time computing since it can coexist with other applications.

8 Future Work

WMPI II is stable, commercially supported middleware that has a number of new features and improvements on its roadmap. For example we want to extend the native support for new communication devices, like Myrinet. We are also currently working on tighter integration with the GRID infrastructure and fault-tolerance, as well as extended support for more POSIX-like operating systems.

References

[MAR] José Marinho, João Gabriel Silva. "WMPI-Message Passing Interface for Win32 Clusters". Proc. EuroPVM/MPI98, Sept. 1998, Liverpool, Springer Verlag

[MPIR1] Marc Snir et. al., "MPI - The Complete Reference", vol. 1, MIT Press, 1998

[MPIR2] W. Gropp et. al., "MPI - The Complete Reference", vol. 2, MIT Press, 1998

[MPIS1] MPI Forum, "MPI: A Message-Passing Interface Standard", 1994

[MPIS2] MPI Forum, "MPI-2: Extensions to the Message-Passing Interface", 1997

[LAMW] The LAM group maintains a list of available MPI implementations and their features on http://www.lam-mpi.org.

[HPI] W. Group et al. "High-performance, portable implementation of the Message Passing Interface Standard", Journal of Parallel Computing, vol. 22, No 6, 1996

[LAM] Greg Burns et al., "LAM: An Open Cluster Environment for MPI", Proceedings of Supercomputing Symposium, 1994

[CHA] William Gropp and Ewing Lusk, "MPICH working note: Creating a new MPICH device using the channel interface", Technical Report ANL/MCS-TM-213, Argonne National Laboratory, 1995.

[OMP] "OpenMP: An Industry-Standard API for Shared-Memory Programming", IEEE Computational Science & Engineering, Vol. 5, No. 1, Jan/Mar 1998

[PAL] "Pallas MPI Benchmarks - PMB, Part MPI-1", Pallas GmbH, 2000

[NAS] D. H. Bailey et al. "The NAS Parallel Benchmarks", International Journal of Supercomputer Applications, 1991

[LIN] Jack J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software", University of Tennesse, 1995