Analysis of the Component Architecture Overhead in Open MPI

B. Barrett[1], J.M. Squyres[1], A. Lumsdaine[1], R.L. Graham[2], G. Bosilca[3]

Open Systems Laboratory, Indiana University
{brbarret, jsquyres, lums}@osl.iu.edu

Los Alamos National Lab
rlgraham@lanl.gov

Innovative Computing Laboratory,
University of Tennessee,
bosilca@cs.utk.edu

Abstract. Component architectures provide a useful framework for developing an extensible and maintainable code base upon which large-scale software projects can be built. Component methodologies have only recently been incorporated into applications by the High Performance Computing community, in part because of the perception that component architectures necessarily incur an unacceptable performance penalty. The Open MPI project is creating a new implementation of the Message Passing Interface standard, based on a custom component architecture – the Modular Component Architecture (MCA) – to enable straightforward customization of a high-performance MPI implementation. This paper reports on a detailed analysis of the performance overhead in Open MPI introduced by the MCA. We compare the MCA-based implementation of Open MPI with a modified version that bypasses the component infrastructure. The overhead of the MCA is shown to be low, on the order of 1%, for both latency and bandwidth microbenchmarks as well as for the NAS Parallel Benchmark suite.

1 Introduction

MPI implementations are designed around two competing goals: high performance on a single platform and support for a range of platforms. Vendor optimized MPI implementations must support ever evolving hardware offerings, each with unique performance characteristics. Production quality open source implementations, such as Open MPI [6], LAM/MPI [13], and MPICH [9], face an even wider range of platform support. Open MPI is designed to run efficiently on platforms ranging from networks of workstations to custom built supercomputers with hundreds of thousands of processors and high speed interconnects.

Open MPI meets the requirements of high performance and portability with the Modular Component Architecture (MCA), a component system designed for High Performance Computing (HPC) applications. While component based programming is widely used in industry and many research fields, it is only recently gaining acceptance in the HPC community. Most existing component architectures do not provide the low overheads necessary for use in HPC applications. Existing architectures are generally designed to provide features such as language interoperability and to support rapid application development, with performance as a secondary concern.

In this paper, we show that Open MPI's MCA design provides a component architecture with minimal performance implications. Section 2 presents similar work for other component architectures. An overview of the Open MPI architecture is presented in Section 3, focusing on the component architecture. Finally, Section 4 presents performance results from our experiments with Open MPI.

2 Related Work

Component architectures have found a large degree of success in commercial and internet applications. Enterprise JavaBeans, Microsoft COM and DCOM, and CORBA provide a rich environment for quickly developing a component based application. These environments focus on industrial applications, providing reasonable performance for such applications. However, either the languages supported or the overheads involved make them unsuitable for the high performance computing community. Literature on the performance of such component infrastructures is sparse, most likely due to the fact that performance is not a concern for the intended uses of these component architectures.

The Common Component Architecture (CCA) [2] is designed to provide a high performance component architecture for scientific applications. Bernholdt et. al. [3] study the overheads involved in the CCA design and found them to be small, on the order of two extra indirect function calls per invocation. CCA components are designed to be large enough that component boundaries are not crossed for inner loops of a computation. Therefore, the overhead of CCA is negligible for most applications. Much of the overhead is due to inter-language data compatibility, an overhead that is not applicable in Open MPI.

3 Open MPI Architecture

Open MPI is a recently developed MPI implementation, tracing its history to the LAM/MPI [13], LA-MPI [8], FT-MPI [5], and PACX-MPI [10] projects. Open MPI provides support for both MPI-1 and MPI-2 [7,12].[1] Open MPI is designed to be scalable, fault tolerant, and provide high performance in a variety of HPC environments. The use of a component architecture allows for a well architected code base that is both easy to test across multiple configurations and easy to integrate into a new platform.

3.1 Component Architecture

The Modular Component Architecture (MCA) is designed to allow users to build a customized version of Open MPI at runtime using components. The high over-

[1] One-sided support is scheduled to be added to Open MPI shortly after the first public release.

heads generally associated with CORBA and COM are avoided in the MCA by not supporting inter-process object communication or cross-language support – MCA components provide a C interface and interface calls are local to the MPI process. Components are opened and loaded at runtime on demand, using the GNU Libtool `libltdl` software package for portable dynamic shared object (DSO) handling. Components can also be linked into the MPI library for platforms that lack support from `libltdl` or when a static library is desired. Current MPI level component frameworks include point-to-point messaging, collective communication, MPI-2 I/O, and topology support. The runtime infrastructure for Open MPI include component frameworks for resource discovery, process startup, and standard I/O forwarding, among others.

In order to provide a manageable number of measurements while still measuring the overhead of the MCA design, we focus on the components directly responsible for MPI point-to-point communication for the remainder of this paper. Many common MPI benchmarks are based primarily on point-to-point communication, providing the best opportunities for analyzing the performance impact of the MCA on real applications.

3.2 MPI Point-to-Point Design

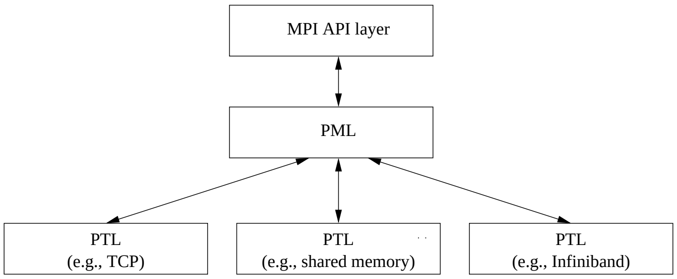


Fig. 1. Open MPI component frameworks for MPI point-to-point messages.

Open MPI implements MPI point-to-point functions on top of the Point-to-point Management Layer (PML) and Point-to-point Transport Layer (PTL) frameworks (Fig. 1). The PML fragments messages, schedules fragments across PTLs, and handles incoming message matching. Currently, there is one PML component, TEG [14]. TEG is designed to support message fault tolerance, recovery from corrupted data, and dropped packets.[2] It can also simultaneously use multiple communication channels (PTLs) for a single message. The PTL provides an interface between the PML and underlying network devices.

[2] These features remain under active development and may not be available in the first release of Open MPI.

4 Component Overhead Analysis

The MCA design's primary source of overhead is the use of indirect calls through function pointers for dispatching into a component. There are two designs for calling into components in the MCA, depending on how many component instances (modules) are active within the framework. For frameworks like the PML, where only one module is active per process, a global structure is used to hold the set of function pointers. The address of the global structure is known at link time. In the case of the PTL frameworks, there are multiple components active, so there is not a single global structure of function pointers. Instead, there are multiple tables stored by the caller of the framework, the PML in this case. The PML must compute the address of the function pointer in a PTL structure, load the value of the function pointer, and make the function call.

To measure the overhead of the component architecture in Open MPI, we added the ability to bypass the indirect function call overhead inherent in the MCA design. Calls from the MPI layer into the PML and from PML into the PTL are made directly rather than using the component architecture. The GM PTL, supporting the Myrinet/GM interconnect, was chosen because it offered a low latency, high bandwidth environment best suited for examining the small overheads involved in the MCA. The ability to "hard code" the PML is available as part of Open MPI as a configure time option. Bypassing the PTL component interface is not part of the Open MPI release, as it greatly limits the functionality of the resulting MPI implementation. In particular, bypassing the PTL component interface disables message striping over multiple devices and the ability to send messages to self. For the majority of the tests discussed in this paper, such limitations were not relevant to examining the overheads of the MCA.

Configuration	Description
MPICH-GM	Myricom MPICH-GM 1.2.6..14a, built using the default build script for Linux with static library
LAM/MPI	LAM/MPI 7.1.1, with GM support and static library
Open MPI shared DSO	Open MPI, `libmpi` shared library. Components dynamically loaded at runtime, using the component interface
Open MPI shared direct	Open MPI, `libmpi` shared library. Point-to-point components part of `libmpi`, bypassing the component interface
Open MPI static DSO	Open MPI, `libmpi` static library. Components part of `libmpi`, using the component interface
Open MPI static direct	Open MPI `libmpi` static library. Point-to-point components part of `libmpi`, bypassing the component interface

Table 1. Build configurations used in performance tests.

Two variables relevant to the overhead of the MCA system for point-to-point communication are how `libmpi` is built and whether the MCA interface is used for point-to-point communication. Table 1 describes the MPI configurations used in testing. Open MPI alpha release r5408 was used for testing Open MPI and was modified to support bypassing the PTL component overhead. MPICH-GM 1.2.6..14a, the latest version of MPICH available for Myrinet,[3] and LAM/MPI 7.1.1 were used to provide a baseline performance reference.

All MPI tests were performed on a cluster of 8 dual processor machines connected using Myrinet. The machines contain 2.8 GHz Intel Xeon processors with 2 GB of RAM. A Myricom PCIX-D NIC is installed in a 64 bit 133 MHz PCI-X slot. The machines run Red Hat 8.0 with a Linux 2.4.26 based kernel and Myricom's GM 2.0.12. Additional CPU overhead tests were performed on a dual 2.0 GHz AMD Opteron machine with 8 GB of RAM running Gentoo Linux and the 2.6.9 kernel and an Apple Power Mac with dual 2.0 GHz IBM PPC 970 processors and 3.5 GB of memory running Mac OS X 10.4.

4.1 Indirect Function Call Overhead

Fig. 2 presents the overhead, measured as the time to make a call to a function with no body, for different call methods. A tight loop is used to make the calls, so all loads should be satisfied from L1 cache, giving a best case performance. The direct call result is the time to call a function in a static library, a baseline for function call overheads on a particular platform. The function pointer result is the cost for calling the same function, but with a load dependency to determine the address of the function. As expected, the cost is approximately the cost of a load from L1 cache plus the cost of a direct function call. Calling a function in a shared library directly (the shared library call result) requires indirect addressing, as the location of a function is unknown until runtime. There is some additional overhead in a shared library call due to global offset table (GOT) computations, so a direct call into a shared library is generally more expensive than an indirect call into a static library [11]. The unusually high overhead for the PPC 970 when making shared library calls is due to the Mach-O ABI used by Mac OS X, and not the PPC 970 hardware itself [1].

Function calls into a DSO are always made through a function pointer, with the address of the function explicitly determined at runtime using `dlsym()` (or similar). In modern DSO loader implementations, GOT computations are not required. The cost of calling a function in a DSO is therefore much closer to the cost of an indirect function call into a static library than a direct function call into a shared library. From this result, it should be expected that the performance impact of the component architecture in Open MPI will be more from the use of shared libraries than from the component architecture itself.

4.2 Component Effect on Latency and Bandwidth

MPI latency for zero byte messages using a ping-pong application and bandwidth using NetPIPE are presented in Fig. 3. All builds of Open MPI exhibit performance differences of less than 2%, with most of the performance difference related to whether Open MPI used shared or static libraries. Bypassing

[3] At the time of writing, Myricom does not provide an MPICH-2 based implementation of MPICH-GM.

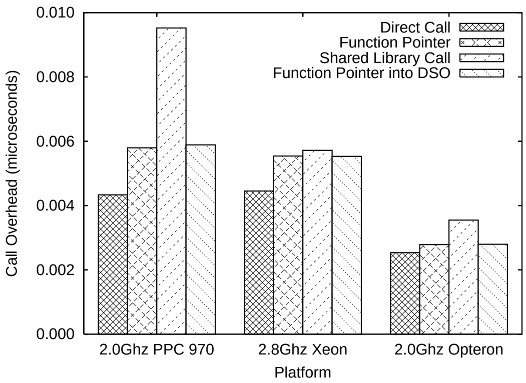
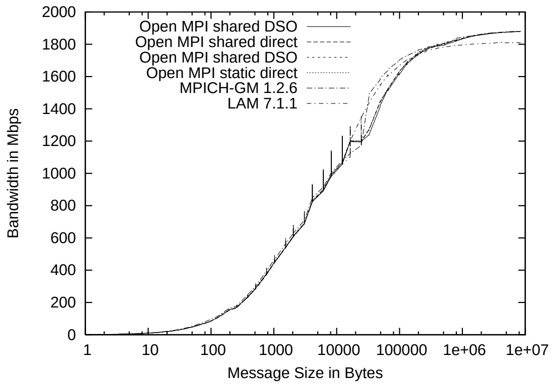


Fig. 2. Time to make a call into an empty function for a number of common architectures

Implementation	Latency
Open MPI shared DSO	7.21us
Open MPI shared direct	7.17us
Open MPI static DSO	7.13us
Open MPI static direct	7.12us
MPICH-GM	6.93us
LAM/MPI	7.55us

(a)



(b)

Fig. 3. Latency of zero byte messages and NetPIPE bandwidth for component and direct call configurations of Open MPI.

the component infrastructure for point-to-point messages shows little impact on either latency or bandwidth. In the worst case, the MCA overhead was .04 microseconds, which is a fraction of the end-to-end latency for the GM software stack. The bandwidth results show Open MPI is comparable to MPICH-GM and LAM/MPI for small messages. For large messages, Open MPI is comparable to MPICH-GM and approximately 70 Mbps faster than LAM/MPI. Open MPI suffers a slight performance drop for messages between 32 KB and 256 KB when compared to LAM/MPI and MPICH-GM. The performance drop appears to be caused by our wire protocol, and should be solved through further tuning.

4.3 Component Effect on NAS Parallel Benchmarks

To approximate real application performance impact from Open MPI's use of components, the NAS Parallel Benchmark suite version 2.4 [4] was run with the build configurations described in Table 1. The results in Table 2 are for four processes, using the B sized benchmarks. Each process was executed on a separate node, to prevent use of the shared memory communication channel by configurations that support multiple interconnects. Each test was run five times, with the lowest time given. Variance between runs of each test was under 2%.

The CG and MG tests invoke MPI communication that requires sending a message to self. Due to the design of Open MPI, this requires multiple PTL components be active, which is disabled in the direct call PTL configuration. Therefore, the CG and MG direct call results are with only the PML component interface bypassed. Performance of the Open MPI builds is generally similar, with variations under 3% in most cases. Similar to Section 4.2, the NAS Parallel Benchmarks show that there is very little measurable overhead in utilizing the MCA in Open MPI. Open MPI performance is comparable to both LAM/MPI and MPICH-GM for the entire benchmark suite.

Implementation	BT	CG	EP	IS	LU	MG	SP
Open MPI shared DSO	471.16s	95.58s	77.20s	4.37s	297.06s	12.12s	422.43s
Open MPI shared direct	475.91s	95.82s	77.33s	4.34s	298.49s	13.54s	422.16s
Open MPI static DSO	472.48s	95.08s	77.17s	4.35s	297.26s	12.96s	416.76s
Open MPI static direct	477.21s	95.15s	77.21s	4.28s	299.35s	13.50s	421.19s
MPICH-GM	475.63s	96.83s	77.14s	4.22s	296.98s	13.74s	421.95s
LAM/MPI	473.93s	99.54s	75.98s	4.01s	298.14s	13.69s	420.70s

Table 2. NAS Parallel Benchmark results for Open MPI, MPICH-GM, and LAM/MPI using 4 processors and the B sized tests.

5 Summary

Open MPI provides a high performance implementation of the MPI standard across a variety of platforms through the use of the Modular Component Architecture. We have shown that the component architecture used in Open MPI provides negligible performance impact for a variety of benchmarks. Further, the Open MPI project provides performance comparable to existing MPI implementations, and has only recently begun optimizing performance. The component architecture allows users to customize their MPI implementation for their hardware at run time. Only features that are needed by the application are included, removing the overhead introduced by unused features.

Acknowledgments

This work was supported by a grant from the Lilly Endowment and National Science Foundation grants NSF-0116050, EIA-0202048 and ANI-0330620. Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36. This paper was reviewed and approved as LA-UR-05-4576. Project support was provided through ASCI/PSE and the Los Alamos Computer Science Institute, and the Center for Information Technology Research (CITR) of the University of Tennessee.

References

[1] Apple Computer, Inc. Mach-O Runtime Architecture for Mac OS X version 10.3. Technical report, August 2004.

[2] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott R. Kohn, Lois McInnes, Steve R. Parker, and Brent A. Smolinski. Toward a common component architecture for high-performance scientific computing. In *HPDC*, 1999.

[3] D. E. Bernholdt et al. A component architecture for high-performance scientific computing. *to appear in Intl. J. High-Performance Computing Applications.*

[4] Rob F. Van der Wijngaart. NAS Parallel Benchmarks version 2.4. Technical Report NAS Technical Report NAS-02-007, NASA Advanced Supercomputing Division, NASA Ames Research Center, October 2002.

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

[6] 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.

[7] 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.

[8] 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.

[9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

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

[11] John R. Levine. *Linkers and Loaders.* Morgan Kaufmann, 2000.

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

[13] 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.

[14] T.S. Woodall et al. TEG: A high-performance, scalable, multi-network point-to-point communications methodology. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004.