Evaluating the performance of MPI-2 dynamic communicators and one-sided communication

Edgar Gabriel, Graham E. Fagg, and Jack J. Dongarra

Innovative Computing Laboratory, Computer Science Departement,
University of Tennessee, Suite 413, 1122 Volunteer Blvd.,
Knoxville, TN-39996, USA
{egabriel, fagg, dongarra}@cs.utk.edu

Abstract. This paper evaluates the performance of several MPI implementations regarding two chapters of the MPI-2 specification. First, we analyze, whether the performance using dynamically created communicators is comparable to the approach presented in MPI-1 using a static communicator for different MPI libraries. We than evaluate, whether the communication performance of one-sided communication on current machines, represents a benefit or a drawback to the end-user compared to the more conventional two-sided communication.

1 Introduction

The MPI-2 specification [3] extends the MPI-1 document [2] by three major chapters (dynamic process management, one-sided communication and parallel File-I/O), several minor ones and some corrections/clarifications for MPI-1 functions. Although it has been published since 1997, up to now only the parallel File-I/O chapter has really been accepted by the end-users, and clearly this is the reason why benchmarks have been mainly developed in this area of the MPI-2 specification[4].

Assuming that the user really wants to use features of the MPI-2 specification, we would like to investigate in this paper, what the performance benefits and drawbacks of different features of MPI-2 are. Two questions are of specific interest in the context of this paper: first, do dynamically created communicators offer the same point-to-point performance on current implementations comparable to the static MPI_COMM_WORLD approach ? And second, what is the achievable performance using one-sided operations compared to two-sided communication?

Since the number of available MPI implementations implementing some parts of the MPI-2 specification is meanwhile quite large, we would like to limit ourselves for the scope of this paper to analyze the performance and implementation of following libraries:

- **MPI/SX**: library version 6.7.2. Tests were executed on an NEC SX-5 consisting of 16 250 MHz processors with 32 GBytes of main memory.

- **Hitachi MPI**: library version 3.07. Tests were executed on a Hitachi SR8000 with 16 nodes, each having 8 250 MHz processors. Each node has 8 GBytes of main memory.
- **SUN MPI**: library version 6. Tests were executed on a SUN Fire 6800, with 24 750 MHz Sparc III processors, and 96 GBytes of main memory.
- **LAM MPI**: library version 6.5.9. Tests were executed on a cluster with 32 nodes, each having two 2.4 GHz Pentium 4 Zeon processors and 2 GBytes of memory. The nodes are connected by Gigabit Ethernet.

The library versions used in the tests are always the most recent versions available from the according vendors/implementors. We would like to emphasize at this point, that our intention is **not** to compare the numbers between the machines. Our goal is to compare the numbers achieved in the tests with the performance measured on the very same machine for the static MPI-1 scenario, and therefore comment on the quality of the implementation of the MPI-library.

2 Performance results with dynamic communicators

The MPI-2 document gives the user three possibilities on how to create a new communicator that includes processes, which have not been part of the previous world-group:

1. Spawn additional processes using MPI_Comm_spawn/multiple
2. Connect two already running (parallel) applications using a socket-like interface, where one application connects using MPI_Comm_connect to another application, which calls MPI_Comm_accept.
3. Connect two already running application processes, which have already a socket connection established by using MPI_Comm_join.

The third method explicitly restricts itself to be used for socket communication. However, the goal of this paper is not to measure the socket performance on each machine, but we would like to measure the performance of methods, where the user might expect to get the same performance with a dynamically created communicator like with the static approach. Therefore, we are just considering the first two approaches in this paper.

The tests performed are a modification of a ping-pong benchmark, which has been adapted to work with variable communicator and partner arguments. The latter one is necessary, since in some of the cases we have to deal with inter-communicators, and therefore the rank of the communication partner might be different than in the static case using MPI_COMM_WORLD.

2.1 Results on the NEC SX-5

The first library which we would like to analyze regarding its performance and usability of this part of the MPI-2 specification, is the implementation of NEC. Starting an application which is using MPI_Comm_spawn, the user has to specify an additional parameter called `max_np`. For example, if the application is started originally with 4 processes and the user wants to spawn later on 4 more processes, the command line has to look like follows:

```
mpirun -np 4 -max_np 8 ./<myapp>
```

While this approach is explicitly allowed by the MPI-2 specification, it also clearly sets certain limits on the dynamic behavior of the application.

When using the connect/accept approach, the user has to set another flag for compiling and starting the application. The `tcpip` flag strongly indicates already, that the connect/accept model has been implemented in MPI/SX using TCP/IP. An interesting question regarding this flag is, whether communication in each of the independent, parallel applications is influenced by this flag, e.g. whether all communication is executed using TCP/IP, or whether just the communication between the two applications connected by the dynamically created inter-communicator is using TCP/IP.

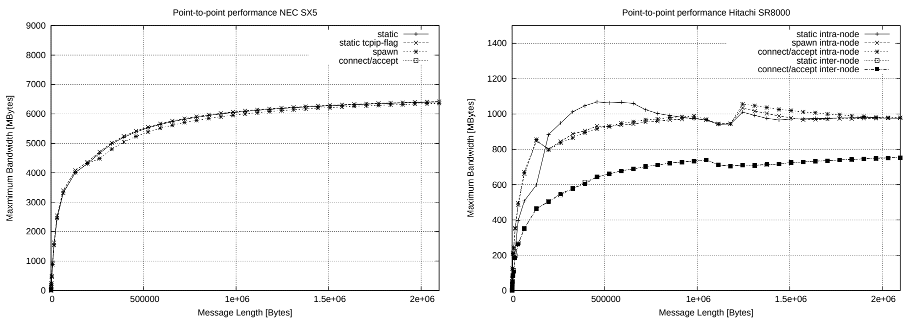


Fig. 1. Point to point performance on the SX-5 (left) and Hitachi SR8000 (right)

Figure 1 shows on the left side the maximum bandwidth achieved with the different communicators. Obviously, the performance achieved with a communicator created by MPI_Comm_spawn is identical to the static approach. However, the line for the MPI_Comm_connect/accept approach is not visible, since the TCP/IP performance of the machine can not compete with the bandwidth achieved through the regular communication device.

At a first glance, the `tcpip` flag does not seem to have an affect on the maximum achievable bandwidth. However, our measurements showed, that the variance were somewhat higher than without the `tcpip` flag. The standard deviation from the average without the flag was usually below 1 %, while using the `tcpip` flag it was in the range of 5-10%. Therefore, the flag does have an influence on the performance of an application, even if it is not dramatic.

2.2 Results on the Hitachi SR8000

On the Hitachi SR8000 we conducted two sets for each experiment: all tests were executed using two processes on the same node, indicated in fig. 1 as intra-node communication, and using two processes on different nodes, referred to as inter-node communication.

Like shown in the right part of fig. 1, for the intra-node tests, the performance achieved with the MPI_Comm_spawn example as well as for the MPI_Comm_connect/accept example is comparable to the static approach. Furthermore, the inter-node performance for the MPI_Comm_connect/accept example was basically identical to the static inter-node performance. However, we did not manage to make the spawn-method work across several nodes. While we do not believe that this is a restriction of the MPI-library, it indicates one of the possible problems for dynamically creating new processes, namely the interaction with the scheduler of the machine.

2.3 Results on the SUN Fire 68000

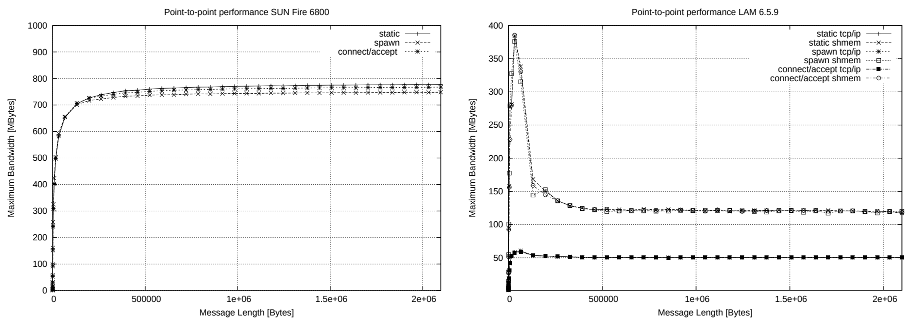


Fig. 2. Point to point performance on the SUN-Fire (left) and the with LAM (right)

The results achieved with SUN-MPI are presented in the left part of figure 2. To summarize these results and the experiences, no additional flags had to be used to make any of the examples work, and the performance achieved in all scenarios tested were always basically identical to the static MPI_COMM_WORLD scenario.

2.4 Results using LAM 6.5.9

Using the most recent version of LAM, all three tests provided basically the same performance, for using both, TCP/IP communication for connecting several processes on several nodes, and a shared-memory interface for intra-node communication (see right part of fig. 2).

We would like to comment on the behavior of LAM when using MPI_Comm_spawn. When booting the lam-hosts, the user has to specify in a hostfile the list of machines, which should be used for the parallel job. A LAM daemon is then started on each of these machines. The processes are started according to their order in the hostfile. When spawning new processes, it appears for the default configuration, the processes are started again using the first machine in the list. Optimally, the user would expect, that the first 'unused' machine (at least unused according to the job which spawns the processes) is chosen, to distribute the load appropriately. With the current scheme, it is probable that for compute intensive application by spawning additional processes on the machines which are running the MPI-job already, the overall job will be slowed down.

3 Performance of one-sided operations

The chapter about one-sided communication is supposed to be the most dramatic supplement to the MPI-1 specification, since it gives the user a completely new paradigm for exchanging data between processes. In contrary to the two-sided communication of MPI-1, a single processes can control the parameters for source and destination processes. However, since the goal was to design a portable interface for one-sided operations, the specification has become rather complex. It can be briefly summarized as follows:

- For moving data from the memory of one process to the memory of another processes, three operations are provided: MPI_Get, MPI_Put and MPI_Accumulate, the latter one combining the data of the processes in a similar fashion to MPI_Reduce.
- For the synchronization between the processes involved, three methods are provided: MPI_Win_fence, MPI_Win_start/post/wait/complete and MPI_Win_lock/unlock. The first two methods are called *active target synchronization*, since the destination processes is also involved in the operation. The last method is called *passive target synchronization*, since the destination process is not participating in any of the MPI-calls.

Another call from the MPI-2 document is of particular interest for the one-sided operation, namely the possibility to allocate some 'fast' memory using MPI_Alloc_mem [3]. On shared-memory architectures this might be for example a shared memory segment which can be directly accessed by a group of processes. Therefore, RMA operations and one-sided communication might be faster, if memory areas are involved, which have been allocated via this function.

3.1 Description of the test code

The ping-pong benchmark used in the last section has been further modified to work with one-sided operations. For this, we are creating first an access and exposure epoch on both processes and putting/getting data in/from the remote memory. After closing the access and exposure epoch on both processes and thus forcing all operations to finish, we create a second exposure and access epoch, transferring the data back. We are timing the overall execution time for both operations thus producing comparable results to the ping-pong benchmark.

For the passive target synchronization, we did not manage to create a reasonable version of this test. For measuring the achievable bandwidth using MPI_Win_lock/unlock, a streaming benchmark should be used instead. For producing comparable results (also with respect to the previous section), we omitted the passive target synchronization in the following tests and focused on communication methods using active target synchronization.

The MPI-2 specification gives the user many possibilities for optimizing the one-sided operations. For example, when creating a window object, the user has to pass an MPI_Info object as an argument, where they can indicate, how the window is used in its application. Another optimization possibility is the assert argument in the synchronization routines. For our tests, we used the default values for both arguments, which are MPI_INFO_NULL and assert=0. An investigation of the effect of each of these parameters on different machines would be very interesting, but it would exceed the length-limit of this paper. Additionally, the usage of these arguments might optimize the communication performance on one platform, while being in the worst case a performance drawback on another one. Therefore, we expect most users to use just the default parameter settings.

3.2 Results on the NEC SX-5

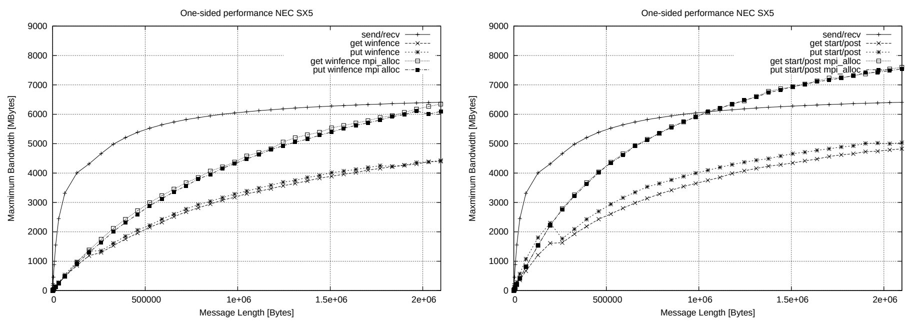


Fig. 3. Performance of one-sided operations using MPI_Win_fence (left) and MPI_Win_start/post for synchronization on the SX-5

The performance of one-sided operations with MPI/SX without using fast memory, is lower than regular point-to-point performance achieved by using MPI_Send and MPI_Recv. The user can still achieve the same maximum bandwidth with one-sided operations like in the MPI-1 scenario, however the message size has to reach up to 4 MBytes for achieving this bandwidth.

Using MPI_Alloc_mem to allocate the memory segments, which are then used in the one-sided operations, the user can improve the performance of one-sided operations for both, the winfence and the start/post test. While in the previous test without the usage of MPI_Alloc_mem, the start/post mechanism was achieving a slightly better performance than the winfence mechanism, with the 'fast' memory, the difference is increasing significantly. For messages larger than 1 MByte, it even outperforms the two-sided communication used as a reference.

3.3 Results on the Hitachi SR8000

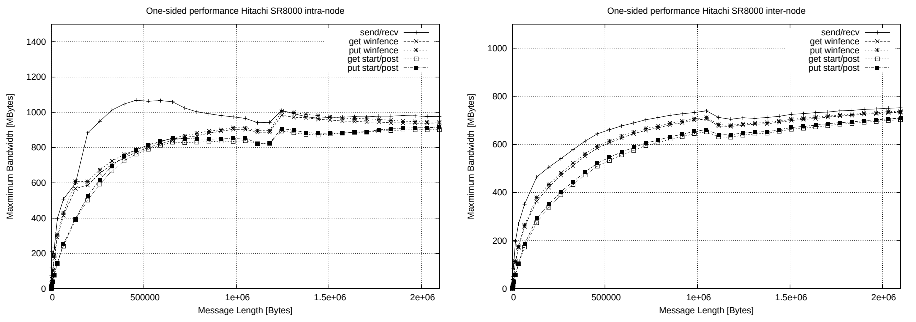


Fig. 4. Performance of one-sided operations for intra node (left) and inter-node communication on the Hitachi SR8000

The results for the Hitachi are shown in figure 4. Up to 1.5 MByte messages, the one-sided operations are partially more than 20 % slower than the two-sided communication. For messages exceeding this message size, the bandwidth achieved using one-sided operations is slowly converging towards the bandwidth of the send/recv test-case. There is also no real difference for the performance whether we are using MPI_Put or MPI_Get. However, the winfence test-case achieves usually a slightly better performance than the start/post mechanism.

The situation is similar for the inter-node case, the implementation of the test-suite using MPI_Win_fence for synchronization achieves a somewhat better performance than the test-case using MPI_Win_Start/ Post. For all tests, the usage of MPI_Alloc_mem did not show any effect on the performance.

3.4 Results on the SUN Fire 6800

The results achieved on with SUN-MPI are presented in figure 5. Two major effects can be observed: first, the usage of MPI_Alloc_mem has a huge influence on the performance achieved. In case where 'fast' memory is allocated using this function, the performance achieved with one-sided operations outperforms the point-to-point performance using send/recv operations. Without this routine however, the achievable bandwidth is roughly half of the bandwidth achieved for two-sided communication.

There is no real performance difference between the two synchronization mechanisms analyzed. However, if we are not allocating memory using the provided MPI-function, the performance using MPI_Get was always higher than the one achieved with MPI_Put.

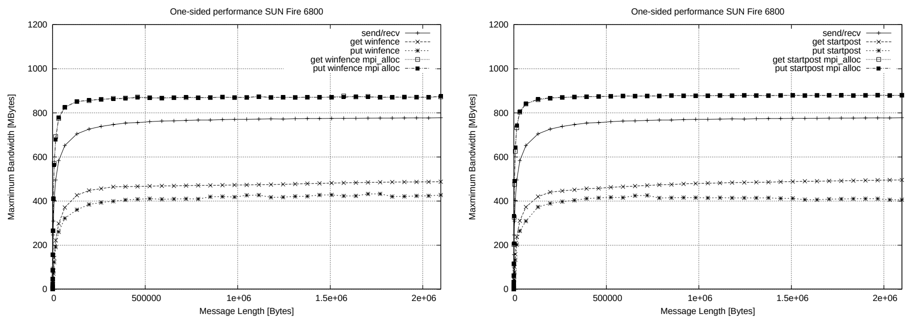


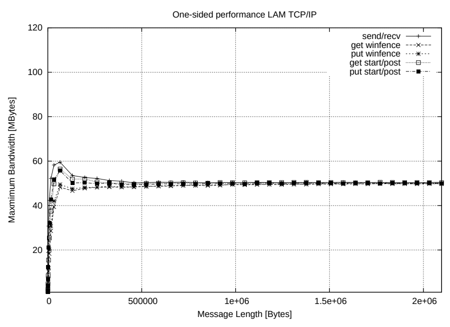
Fig. 5. Performance of one-sided operations using MPI_Win_fence (left) and MPI_Win_start/post for synchronization with SUN-MPI

3.5 Results using LAM 6.5.9

The performance results achieved with LAM are presented in figure 6. For both communication drivers analyzed, the bandwidth achieved with one-sided communication is comparable to the send/recv tests. The only difference is, that the peak observed in both protocols between 32 and 64 Kilobyte messages, is somewhat lower.

3.6 One-Byte Latency

While in the previous chapters we focused on the achievable bandwidth especially for large messages, we would like to summarize the results for small messages on all platforms by presenting the execution time for a data transfer of one byte. Since we did not find any major differences in the performance between MPI_Put and MPI_Get, we present in this section just the results for MPI_Put. A minus in the table does not mean, that the function is not supported by the library, but it just indicates, that the usage of MPI_Alloc_mem did not have any influence on the performance. As shown in table 1, one-sided operations



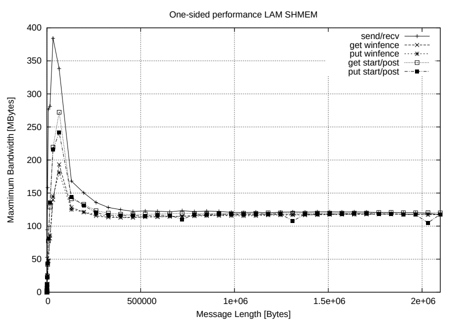


Fig. 6. Performance of one-sided operations for inter-node and intra-node communication with LAM

using active target synchronization have a much higher start-up overhead than two-sided communication. Only on SUN-MPI, when using memory allocated by MPI_Alloc_mem, the user can achieve a reasonable one-byte latency.

	Send/Recv	Win_fence	Win_fence + MPI_Alloc_mem	start/post	start/post + MPI_Alloc_mem
SX5	5.20	117.19	138.03	70.47	81.34
SR8K intra	10.98	64.62	-	182.60	-
SR8K inter	22.26	119.28	-	256.97	-
SUN	2.88	33.9	4.96	27.80	3.38
LAM inter	46.76	260.72	-	133.80	-
LAM intra	16.57	170.88	-	86.90	-

Table 1. Execution time for sending a 1-byte message using different communication methods

4 Summary

In this paper we presented our experiences and the performance of four MPI libraries, with respect to the handling of dynamically created communicators and one-sided communication. Using the MPI-2 features for creating communicators, that include processes which have not been part of the original MPI_COMM_WORLD, has worked basically well on all machines. However, with the exception of SUN-MPI, all have shown certain pitfalls, which might influence the performance and the user-friendliness of these functions.

The usage of one-sided communication offers the application developer a wide variety of possibilities to express the communication pattern of his application. Once familiar with the syntax, the usage of these functions did not impose any major challenges from the usage point of view. The performance achieved with these operations vary partially dramatically, depending on whether the user has allocated the memory for the communication using MPI_Alloc_mem or not. While at the beginning of the 90s, a parallel application had to abstract the communication routines for supporting several communication libraries, it might happen, that end-users have to do this again, if they want to make sure, that they are using the fastest possibility on each machine, even though all communication methods are provided by MPI.

Acknowledgements

The authors would like to thank the Innovative Computing Laboratory, the High Performance Computing Center Stuttgart and the University of Ulm for providing their machines for the tests.

References

1. W. Gropp, E. Lusk, N. Doss, A. Skjellum: *A high-performance, portable implementation of the MPI message-passing interface standard*, Parallel Computing, 22 (1996).
2. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard (version 1.1).* Technical report, June 1995. http://www.mpi-forum.org
3. Message Passing Interface Forum. *MPI-2: Extentions to the Message-Passing Interface* July 18, 1997.
4. Rolf Rabenseifner and Alice E. Koniges *Effective File-I/O Bandwidth Benchmark*, in A. Bode, T. Ludwig, R. Wissmueller (Eds.), 'Proceedings of the Euro-Par 2000, pp. 1273-1283, Springer, 2000.
5. Maciej Golebiewski and Jesper Larsson Traeff *MPI-2 One-Sided Communications in a Giganet SMP Cluster*, Yiannis Cotronis, Jack Dongarra (Eds.), 'Recent Advances in Parallel Virtual Machine and Message Passing Interface', pp. 16-23, Springer, 2001.
6. Glenn Luecke, Wei Hu *Evaluating the Performance of MPI-2 One-Sided Routines on a Cray SV-1*, technical report, December 21, 2002, http://www.public.iastete.edu/ grl/publications.html
7. S. Booth and E. Mourao *Single Sided MPI implementations for SUN MPI* in proceedings of Supercomputing 2000, Dallas, TX, USA.
8. Herrmann Mierendorff, Klaere Cassirer, and Helmut Schwamborn *Working with MPI Benchmark Suites on ccNUMA Architectures* in Jack Dongarra, Peter Kacsuk, Norbert Podhorszki (Eds.), 'Recent Advances in Parallel Virtual Machine and Message Passing Interface', pp. 18-26, Springer, 2000.