Evaluating dynamic communicators and one-sided operations for current MPI libraries *

Edgar Gabriel, Graham E. Fagg, and Jack J. Dongarra
Innovative Computing Laboratory, Computer Science Departement,
University of Tennessee, 1122 Volunteer Blvd., Suite 413,
Knoxville, TN-37996, USA
{egabriel, fagg, dongarra}@cs.utk.edu

November 22, 2004

Abstract

This paper evaluates the current status and 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 then evaluate, whether the communication performance of one-sided communication on current machines, represents a benefit or a disadvantage 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, several minor ones, and some corrections/clarifications for MPI-1 functions. Although it has been published since 1997, currently only the parallel file-I/O chapter has been accepted by the end-users. Clearly this is the reason, that despite of having many benchmarks testing MPI-1 functionality [9, 10, 11], MPI-2 functionality has only been evaluated up to now in this area [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 as the static MPI_COMM_WORLD approach? And second, what is the achievable performance using one-sided operations?

Since the number of available MPI implementations implementing some parts of the MPI-2 specification is meanwhile quite large, this paper can not give a complete overview about all currently available MPI-libraries and the features they are providing. Furthermore, it is not the intention of this paper to compare performance results between the machines, but to compare the numbers achieved using MPI-2 functionality with the performance measured on the very same machine for static MPI-1 scenarios, and therefore to comment on the quality of the implementation of the MPI-2 functionality. As with all benchmark numbers, the reader should be aware, that all results represent only a snapshot.

The structure of this paper is as follows: in section 2 we present briefly the test-suite used throughout the paper. Section 3 presents the results and experiences with handling dynamic communicators. Section 4 presents the performance achieved using one-sided operations with different MPI-libraries and

*This work was supported by the US Department of Energy through contract number DE-FG02-99ER25378.

discusses some related issues. Finally, section 5 summarizes the results achieved and presents the ongoing work in this area.

2 The latency test-suite

This section gives a brief introduction to the latency test-suite, which has been used for the performance evaluation throughout this paper. The latency test-suite is a historically grown collection of tests, which has been used to measure a wide variety of different performance characteristics [12, 13]. Recently, the test-suite has been re-arranged such that it provides building blocks for point-to-point benchmarks, creating an easily extendable benchmarking environment. End-users can thus create their own point-to-point benchmark which incorporates relevant features of their application and thus evaluate the performance of their data exchange routines.

Among the building blocks are:

- variable communicator arguments,
- variable data types, including user defined data types,
- variable communication partners,
- variable data transfer primitives, currently restricted however to ping-pong benchmarks.

The test-suite uses two different methods for determining the next message size to be measured: a multiplicative increase for short message length, and an additive increase of the message size for large messages. This enables a detailed analysis of short message behavior as well as a reasonable number of measurements for large messages.

The latency test-suite reports for each message size an average, maximum and minimum bandwidth achieved and the according execution time. Furthermore, the standard deviation is reported for each message size, indicating the stability of the measurement. The output can either be written to standard output or to a file, using either standard UNIX file operations or relying on MPI-I/O. Sample gnuplot scripts are provided for visualizing the output as well as simple programs which enable comparing several measurements.

For each building block, several reference modules are available. As an example, several constructors for derived datatypes are provided as well as different data transfer primitives or communicator constructors. For the analysis presented in this paper, new communicator constructors using the methods provided in the MPI-2 specification have been added, as shown in section 3 as well as new data transfer primitives using one-sided communication, as shown in section 4.

3 Performance results with dynamic communicators

The MPI-1 specification explicitly restricted itself on a static group of processes, arguing (amongst others), that on most platforms the performance of MPI will be significantly better dealing with a constant process group. The motivation behind the investigations in this chapter therefore is focused around the question, whether this common assumption is justified. More specifically, are the high performance networking devices used on modern architectures capable of dynamically establishing connections to new processes, or are MPI libraries implementing this part of the MPI-2 specification falling back to slower devices (e.g. TCP/IP)?

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. The child processes can retrieve the common communicator by calling MPI_Comm_get_parent.
2. Connect two already running (parallel) applications using a socket-like interface. One application offers a “service” using MPI_Comm_accept, while another application establishes connection to the application offering the service using MPI_Comm_connect.

3. Connect two already running application processes, which have a socket connection established by using MPI_Comm_join. The resulting communicator uses the pre-established socket connection for the data exchange and is restricted to the processes holding the socket-connection.

In the frame of this paper we would like to focus on the first two methods. The executed tests compare the bandwidth and latency achieved using dynamic communicators created by MPI_Comm_spawn, an inter-communicator resulting from the coupling of two independent applications using MPI_Comm_connect and MPI_Comm_accept and a static communicator typical for MPI-1. We furthermore investigate the costs for spawning processes, since many dynamic scenarios would benefit from a flexible handling of child processes.

Not examined in this investigation is how different batch-schedulers could handle applications, which dynamically change the number of processes during runtime. While this is an interesting and important question, it is not a property of the MPI library, and depends strongly on the local installation parameters of the batch-scheduler on the machines.

The MPI libraries examined in this section are:

- **MPI/SX**: library version 6.7.2. Tests were executed on an NEC SX-6 consisting of 8 nodes, each having 8 570 MHz processors with 64 GBytes of main memory per node.
- **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 7.0.4. Tests were executed on a cluster with 32 nodes, each having two 2.4 GHz Pentium 4 Xeon processors and 2 GBytes of memory. The nodes are connected by Gigabit Ethernet.

3.1 Results using MPI/SX

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 from NEC. We analyzed the performance of the library on a single node (intra-node communication) as well as between processes on separate nodes using the NEC IXS switch (inter-node communication).

Starting an application which is going to use MPI_Comm_spawn, the user has to specify an additional parameter called `max_np`, which indicates the maximum number of processes used within the lifetime of the application. For example, if the application is started originally with 4 processes and the user later wants to spawn on four additional processes, the command line would need to be as 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 MPI_Comm_connect and MPI_Comm_accept functions have 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.

Figure 1 shows the maximum bandwidth achieved with the different communicators using a ping-pong benchmark between two processes. Obviously, the performance achieved with a communicator created by MPI_Comm_spawn is identical to the static approach. However, using the communicator created by MPI_Comm_connect/accept approach performs significantly worse, since the TCP/IP performance of the machine can not compete with the bandwidth achieved through the regular communication device.

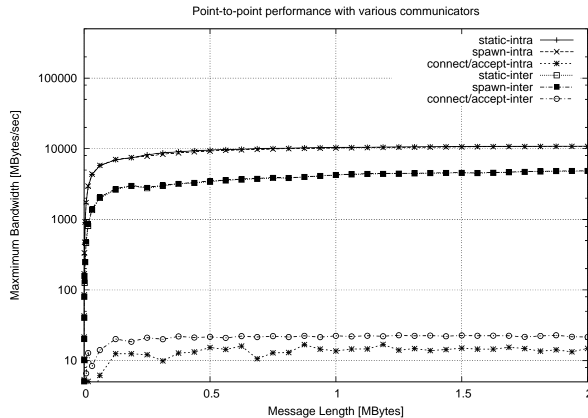

Figure 1: Point to point performance on the SX-6 for various communicators

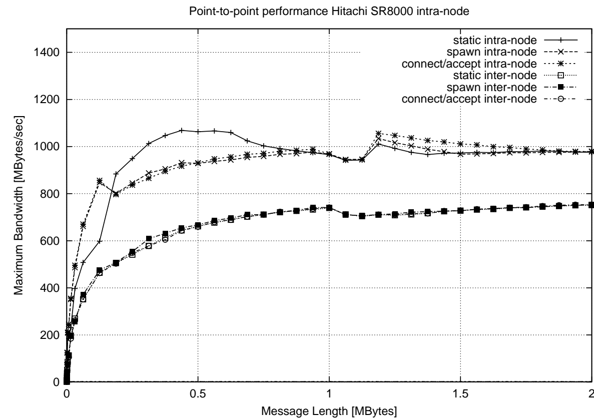

Figure 2: Point to point performance on the Hitachi SR8000 for various communicators

Further investigations showed, that the `tcpip` flag does not seem to influence the maximum achievable bandwidth within each independent application. However, our measurements showed, that the variance was slightly higher than the test case not using the `tcpip` flag. The standard deviation without the `tcpip` flag was usually below 1%, while using the `tcpip` flag it was in the range of 5-10%. This might be the result of occasional polling of TCP/IP sockets.

3.2 Results using Hitachi-MPI

As on the previous machine, we conducted two sets for each experiment on the Hitachi SR8000: all tests were executed using two processes on the same node, indicated in figure 2 as intra-node communication, and using two processes on different nodes, referred to as inter-node communication. As shown in figure 2 the performance achieved with the MPI_Comm_spawn example and with the MPI_Comm_connect/accept example is comparable to the static approach for the inter-node tests. For the intra-node test cases, all tests are achieving the same bandwidth for large messages. For smaller messages, the overall performance is probably identical as well, even if minor variations are observable due to caching effects.

3.3 Results using SUN-MPI

The results achieved with SUN-MPI are presented in figure 3. 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.

3.4 Results using LAM/MPI 7.0.4

Using LAM/MPI v.7.0.4, all three tests provided basically the same performance, as shown in figure 4. We conducted again two sets of tests, one using two processes on separate nodes (inter-node tests) and one using two processes on a single node (intra-node tests). For the intra-node tests, we used the `sysv` shared memory module.

We would like to comment however on the behavior of LAM/MPI 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 of a parallel job are started according to their order in the hostfile. When calling MPI_Comm_spawn, the new processes are started again using the first machine in the list, if no hints are given to the system us-

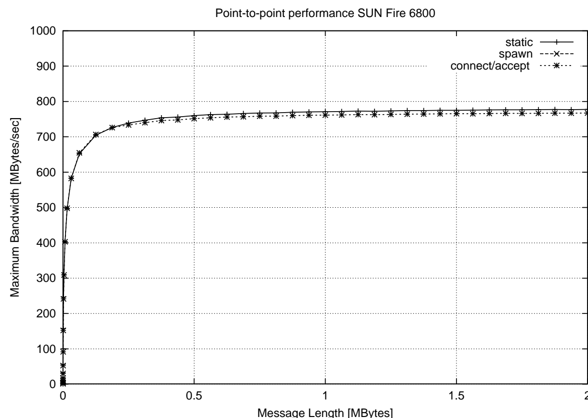

Figure 3: Point to point performance on the SUN-Fire for various communicators

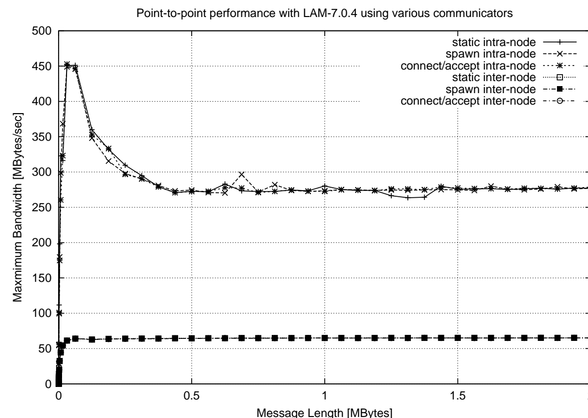

Figure 4: Point to point performance with LAM/MPI for various communicators

ing the according MPI_Info object. Ideally, the user would expect, that the first unused node (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 applications the overall job will slow down by spawning additional processes on the nodes which are already running an MPI-job. The user has the possibility to change this behavior by using specific MPI_Info objects when spawning new process. `lam_spawn_sched_round_robin` introduces a round-robin scheduling of processes giving the application the possibility to specify the first host which would be used. `lam_no_root_node_schedule` excludes the root node of the spawn-command when creating new processes. This second mode is intended for supporting parallel applications using a manager-worker paradigm.

3.5 Costs of MPI_Comm_spawn

Some applications might benefit from a frequently changing number of processes, e.g. some MPI application implementing some type of computational service, various manager-worker scenarios or simulations using adaptive mesh refinement. This section summarizes the costs of spawning one to four processes on each of the analyzed machines. All of these tests have been executed by running the code interactivly, since spawning of processes in a batch environment is on most platforms not supported. Even on machines supporting this functionality, the spawning might then include an arbitrary timeslice for allocating processors through the scheduler.

We could observe on all platforms, that the costs for spawning processes was at least an order of magnitude higher than the point-to-point latency. Most notably, the costs for spawning processes on the Sun Fire are around 40 milliseconds and on the Hitachi SR8000 higher than 400 milliseconds. We are not distinguishing here between the inter- and the intra-node case, since the machine always spawned the first process on the node where the parent process has been running, and the follow-on processes on other nodes. Thus, we always had a mixed intra- and inter-node environment.

For MPI/SX and LAM/MPI, the costs for spawning increases slightly with the number of new processes. However, the overall costs were in a low millisecond area.

No. of procs spawned	1	2	3	4
SX-6 intra	4.1	4.9	5.9	6.7
SX-6 inter	18.7	19.9	19.8	19.9
SR8K	409	419	441	461
SUN	36.3	40.5	43.7	40.4
LAM intra	0.8	1.3	1.8	2.4
LAM inter	0.9	1.3	1.7	2.0

Table 1: Execution time for spawning various number of processes in milliseconds.

4 Performance of one-sided operations

The chapter about one-sided communication was originally planned to be the most dramatic supplement to the MPI-1 specification, since it specifies a completely new paradigm for exchanging data between processes. In contrary to the two-sided communication of MPI-1, a single process controls 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:

- To move 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 target processes in a similar fashion to MPI_Reduce.

- MPI-2 defines furthermore three different methods to synchronize the processes involved in the data exchange: 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 target 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 operations, 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.

Among the analyzed MPI libraries are:

- **MPI/SX**: library version 6.7.2. Tests were executed on an NEC SX-6 consisting of eight nodes with eight 570MHz processors each.

- **Hitachi-MPI**: library version 3.07. Tests were executed on the same machine like in the previous section.

- **IBM-MPI**: ppe.poe version 4.1.0.4. Tests were executed on an IBM p690, each node consisting of 32 1.7GHz Power4+ processors with 128GB memory per node, AIX version 5.2.

- **SUN-MPI**: library version 6. Tests were executed on the same machine as described previously in section 3..

- **LAM/MPI**: library version 7.0.4. Tests were executed on the same cluster as used for the tests in the previous section.

4.1 Benchmarking one-sided operations

One-sided operations are meant to be used in situations, where one process is in control of the sender and receiver side parameters, e.g. the receiver process does not know the precise size or the precise number of incoming messages. Typical usage scenarios might involve for example data exchange between different domains in applications using unstructured meshes.

The basic sequence of functions when using active target synchronization is as follows: processes can allow access to a certain area in their memory to other processes, initiating a so-called exposure epoch. A process must furthermore open an access epoch, if he would like to put data into or get data from the memory of another process. MPI-2 does however not guarantee, that the data transfer is finished after MPI_Put or MPI_Get have returned, the result of the data transfer might only be visible when closing the access and exposure epoch. Because of this property, a code will typically initiate the data transfer using one-sided operations and close the access and exposure epoch at the point, where the data transfer has to be finished from the application point of view. This behavior is similar to non-blocking two-sided communication in MPI-1.

As a result of this property, in each access and exposure epoch an application will execute a limited number of one-sided operations before finishing an access and exposure epoch and launching new ones. The basic performance characteristics of one-sided operations could therefore be measured in determining the synchronization costs, i.e. the costs to open and to close access and exposure epochs, as well as the achievable bandwidth using one-sided operations. These two parameters are the focus of this paper, and will be measured by executing a ping-pong test using one-sided operations.

To implement a ping-pong benchmark using one-sided operations, each process creates first an access and exposure epoch and puts or gets data in/from the remote memory using a single MPI operation. After closing the access and exposure epoch on both processes and thus forcing all operations to finish, both processes create a second exposure and access epoch, transferring the data back. Timing is done including the overall execution time for both operations The following code fragment is indicating the benchmark code for the test-case using MPI_Win_fence for synchronization.

```
if ( rank == 0 ) {
  /* active part in the ping pong */
   MPI_Win_fence ( 0, win );
   MPI_Put ( buf, cnt, datatype,..., win);
   MPI_Win_fence ( 0, win );

   /* passive part in the ping pong */
   MPI_Win_fence ( 0, win );
   MPI_Win_fence ( 0, win );
}
else if ( rank == 1 ) {
   /* passive part in the ping pong */
   MPI_Win_fence ( 0, win );
   MPI_Win_fence ( 0, win );

  /* active part in the ping pong */
   MPI_Win_fence ( 0, win );
   MPI_Put ( buf, cnt, datatype,..., win);
   MPI_Win_fence ( 0, win );
}
```

While a ping-pong benchmark is not necessarily a typical communication pattern when using one-sided operations, we found that it still reveals the relevant communication parameters of one-sided operations. To further judge the MPI implementations, we conducted some additional tests using slight modifications of the ping-pong benchmark described above:

- Analyzed the behavior of one-sided operations using derived datatypes
- Analyzed the performance of one-sided operations for multiple put/get operations between the same pair of processes
- Analyzed whether the processes not actively involved in the one-sided operation can execute calculations, without affecting the performance of the data transfer.

	Send/Recv	Win_fence	Win_fence + MPI_Alloc_mem	start/post	start/post + MPI_Alloc_mem
SX-6 intra	3.0	42.2	53.6	26.9	33.6
SX-6 inter	3.0	42.2	53.6	26.9	33.6
SR8K intra	11.2	64.9	-	182.9	-
SR8K inter	22.8	119.5	-	256.4	-
SUN	2.8	35.3	4.7	29.4	3.3
IBM intra	3.5	63.9	-	69.2	-
IBM inter	11.5	104.0	-	78.4	-
LAM intra	15.4	164.3	-	84.1	-
LAM inter	44.7	257.5	-	127.2	-

Table 2: Execution time for transferring a four-byte message using different communication methods in μs.

MPI-2 allows optimizations of one-sided operations on a group level, which would require additional parameters to characterize them. The analysis of the behavior of one-sided operations on a group level is currently ongoing work and outside of the scope of this paper.

The MPI-2 specification gives users possibilities to optimize one-sided operations by passing hints to the MPI library, which describe certain options on how the windows are used. Some hints can be passed as an MPI_Info object when creating the window, while others can use the `assert` arguments to the synchronization routines. In the following tests, the default values MPI_INFO_NULL and `assert=0` have been used. An investigation into the effects, of each of these parameters on the different systems would be very interesting, it exceeds however the scope of this paper. Additionally, the usage of these arguments might optimize the communication performance on one platform, while degrading the performance on another.

Using passive target synchronization, the parameters characterizing the performance of the operation are different than the ones described above. To measure the achievable bandwidth using MPI_Win_lock/unlock, a streaming benchmark could be used. For producing comparable results with respect to the previous section, we omitted the passive target synchronization in the following tests and focused on operations using active target synchronization only.

4.2 Synchronization costs

To determine the costs of the synchronization operation, the execution time of a single four-byte MPI_Put operation is analyzed. Since the results achieved with MPI_Get are identical, they have been omitted in table 2 for the sake of clarity. Table 2 summarizes the execution time for all analyzed MPI libraries and provides for comparison the time to transfer a four-byte message using MPI_Send and MPI_Recv. All timings are given in μs. A minus in the table indicates, that the usage of MPI_Alloc_mem did not have any influence on the performance for this message size.

The total execution time of the four-byte transfer operation using MPI_Put is for all analyzed MPI libraries significantly higher than the execution time to send four bytes from one process to another using two-sided communication. Since it is very unlikely, that the data transfer costs for the four bytes are the cause of the high execution time for the one-sided operation, we conclude, that the dominating part in these tests are the costs of synchronization operation, respectively the protocol used to implement the synchronization. Only SUN-MPI using memory allocated by MPI_Alloc_mem achieved a reasonable small-message performance close to send/recv communication performance.

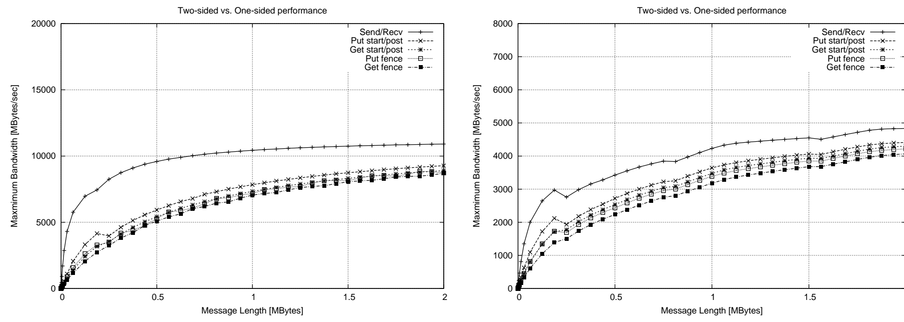

Figure 5: Performance of one-sided operations on the NEC SX-6 for intra node (left) and inter node (right) communication.

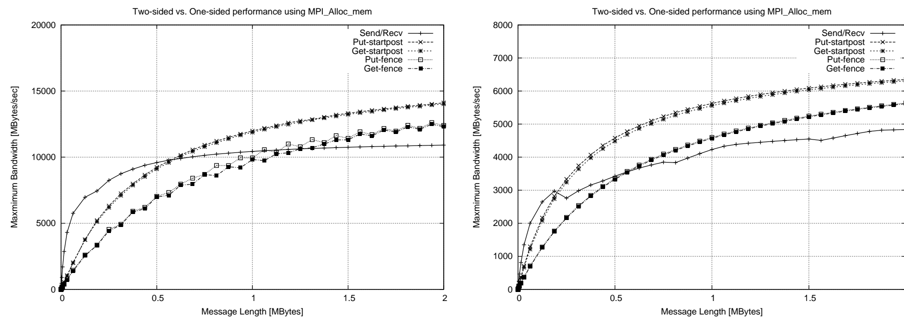

Figure 6: Performance of one-sided operations when using MPI_Alloc_mem on the NEC SX-6 for intra node (left) and inter node (right) communication.

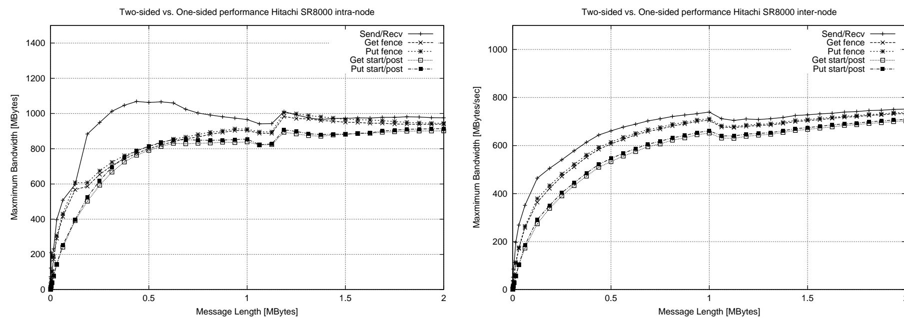

Figure 7: Performance of one-sided operations for intra node (left) and inter node (right) communication on the Hitachi SR8000

4.3 Achieved bandwidth

This section presents the bandwidth achieved with one-sided operations on all analyzed platforms.

4.3.1 Results using MPI/SX

The performance of one-sided operations with MPI/SX without using special memory allocated by MPI_Alloc_mem is lower than regular point-to-point performance achieved using MPI_Send and MPI_Recv. The application can still achieve the same maximum bandwidth with one-sided operations as in the MPI-1 scenario, however the message size has to be significantly larger than for two-sided communication. This is mainly due to the higher synchronization costs as shown in the previous subsection.

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 MPI_Win_fence and the MPI_Win_Start/Post tests. While in the previous test without the usage of MPI_Alloc_mem, the MPI_Win_Start/Post mechanism was achieving a slightly better performance than the MPI_Win_fence mechanism, the difference is increasing significantly when using "fast" memory.

4.3.2 Results using Hitachi-MPI

The results for the Hitachi are shown in figure 7. For messages upto 1.5 Mbytes in length, the one-sided operations are up to 20 % slower than 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 no real difference in the performance whether MPI_Put or MPI_Get is used. However, the performance is usually slightly better when using MPI_Win_fence for synchronizing the participating processes than the performance when using MPI_Win_Start/Post for synchronization.

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.

4.3.3 Results using IBM-MPI

The results achieved on an IBM p690 using an early version of IBM's MPI library for the new High Performance Switch are presented in figure 8. Similarly to other MPI implementations, the cost of the synchronization operation decrease the performance of one-sided operations with IBM-MPI for intra-node communication when passing short messages. For larger message sizes the maximum bandwidth achieved with one and two sided communication become similar. The uneven behavior for large message sizes is reproducible and might be a result of how all the messages fit (or not) into the shared 512MB third level cache on each node during the tests.

IBM-MPI achieves the same inter-node performance for one and two-sided communication. In the tests presented in figure 8 no large pages have been configured on the IBM, which could increase the performance between the nodes for either method.

4.3.4 Results using SUN-MPI

The results achieved with SUN-MPI are presented in figure 9. Two major effects can be observed. First, the usage of MPI_Alloc_mem can dramatically improve the performance of one-sided operations. If 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 optimization, 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 memory is not allocated using the provided MPI-function, the performance using MPI_Get was slightly better than that achieved with MPI_Put.

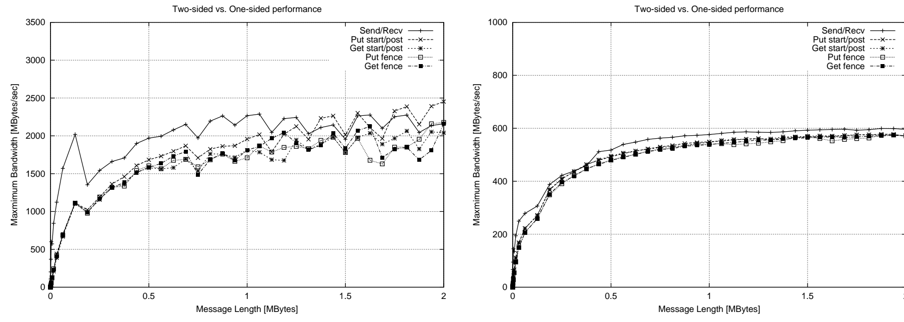

Figure 8: Performance of one-sided operations between processes on the same node (left) and on different nodes (right) with IBM MPI

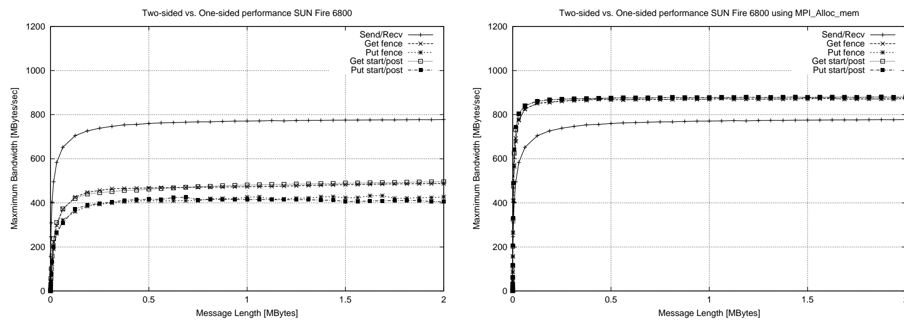

Figure 9: Performance of one-sided operations with SUN-MPI without (left) and with (right) using MPI_Alloc_mem

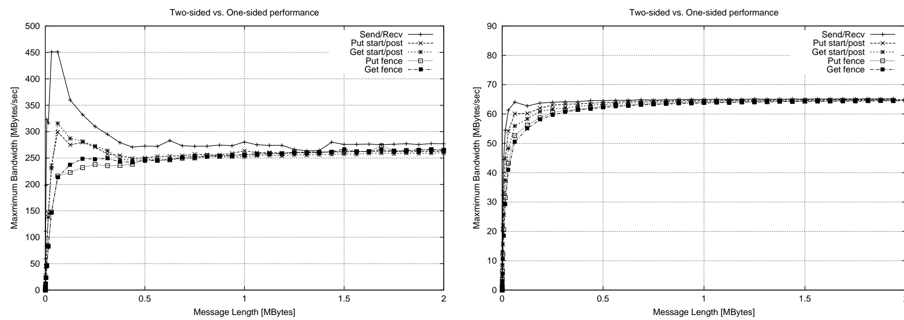

Figure 10: Performance of one-sided operations between processes on the same node (left) and on different nodes (right) with LAM-7.0.4

4.3.5 Results using LAM/MPI 7.0.4

The performance results achieved with LAM are presented in figure 10. For both drivers analyzed, the bandwidth achieved with one-sided communication is comparable to the send/recv performance. The only difference is, that the peak observed in both protocols between 32 and 64 Kilobyte messages, is somewhat lower, which is probably the result of caching effects. The usage of MPI_Alloc_mem does not effect the performance of one-sided operations using these two devices, although according to the documentation the user might profit from this function for certain other devices.

4.4 Derived datatypes in one-sided operations

MPI-2 allows the usage of so-called portable derived datatypes in one-sided operations. Portable datatypes are defined to consist of a single basic datatype, where the displacement between each block can be expressed as a multiple of the extent of the basic datatype.

Since a single process is describing the datatype parameter for the source and destination process, user defined datatypes further increase the complexity of the implementation of one-sided operations, since the MPI library has to find a compact description for derived datatypes, which can be sent and processed in the target window.

With the exception of LAM/MPI, all analyzed implementations fully supported portable datatypes on the analyzed machines. None of the implementations distinguish between portable and non-portable datatypes, which is not an issue as long as the library is used only in a homogeneous environment. The performance of one-sided operations using derived datatypes was on most platforms comparable to the performance of point-to-point operations using the same datatypes. LAM/MPI only supports contiguous derived datatypes. This fact is also mentioned in the documentation and confirmed by our experiments.

4.5 Multiple put/get operations

In contrary to two-sided point-to-point operations, the interface designed for one-sided communication would allow for several inter-message optimizations. Since the data transfer operations just have to be finished after closing the access and exposure windows, an implementation could take advantage of the fact, that it has the complete schedule of all ongoing operations. This would enable for example optimizations on the group level as well as between each pair of processes.

In this subsection we would like to evaluate how the implementations handle the latter scenario. By transferring a single buffer in n Put/Get operations, we compare the execution time of this communication schedule to the case, where the same amount of data is transfered with a single Put/Get operation. A "smart" implementation might recognize, that the operations can be merged into a single one, since the transfered data forms a contiguous buffer on the source as well as on the destination process.

None of the analyzed MPI libraries supported the optimization described in the previous paragraph. The typical behavior of most MPI libraries is similar to the result shown using MPI_Put and the start/post synchronization within a single node on the NEC SX-6 in figure 11. The communication appears to have a protocol switch at certain message lengths, which leads to a temporary drop in the available bandwidth directly at the switching point. For the multiple message scenario analyzed in this subsection, this switching point is shifted towards a larger overall data transfer size with increasing number of transfer operations. This can lead to performance improvements for short Put/Get operations, but also has the effect, that the overall amount of data, which has to be transferred to reach the maximum bandwidth of the system, has to increase.

4.6 Overlapping one-sided operations and computation

This subsection evaluates, how strongly the passive side of a one-sided operation is involved in the data transfer. The test which we conducted is similar to

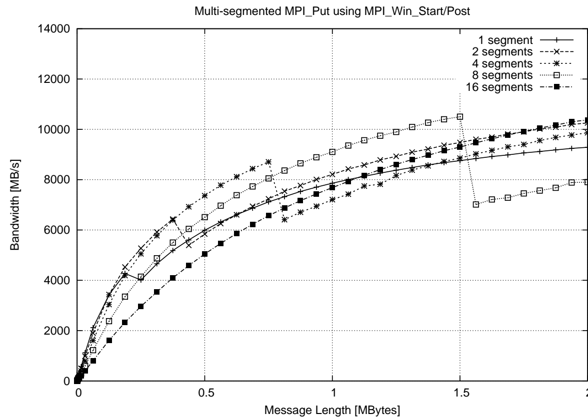

Figure 11: Multiple Put operations using start/post for synchronization on the NEC SX-6

.

the overlap test in the mpptest [9] testsuite. The benchmark measures first the time required for the data transfer of a given message length. In the next step, it determines for a certain calculation the problem size, which takes the same amount of time as the data transfer. The final benchmark executes the data transfer and the computation for the determined problem size simultaneously. If an MPI implementation can not overlap the communication and the computation, the execution time should be approximately twice as high as the pure data transfer, while an MPI library capable of real overlap will execute this benchmark roughly in the same time as the pure data transfer.

Executing this test with one-sided operations was done by introducing a computation in the passive side of the ping-pong. If a library performs well in this test, it can be due to two reasons:

- In case the one-sided operations are modeled on top of two-sided point-to-point communication, the library is capable of doing progress outside of MPI functions (e.g. using a progress thread).
- The library is taking advantage of native support for Put/Get operations on a platform.

From the MPI libraries analyzed, only IBM-MPI handles the overlapping of communication and computation efficiently, for both intra- and inter-node scenarios. This result is shown in the upper part of figure 12. MPI/SX, SUN-MPI, LAM/MPI and Hitachi-MPI all introduced a significant overhead when overlapping communication and computation. As a representative of these MPI libraries, the result of LAM/MPI in the inter-node communication is shown in the lower part of figure 12.

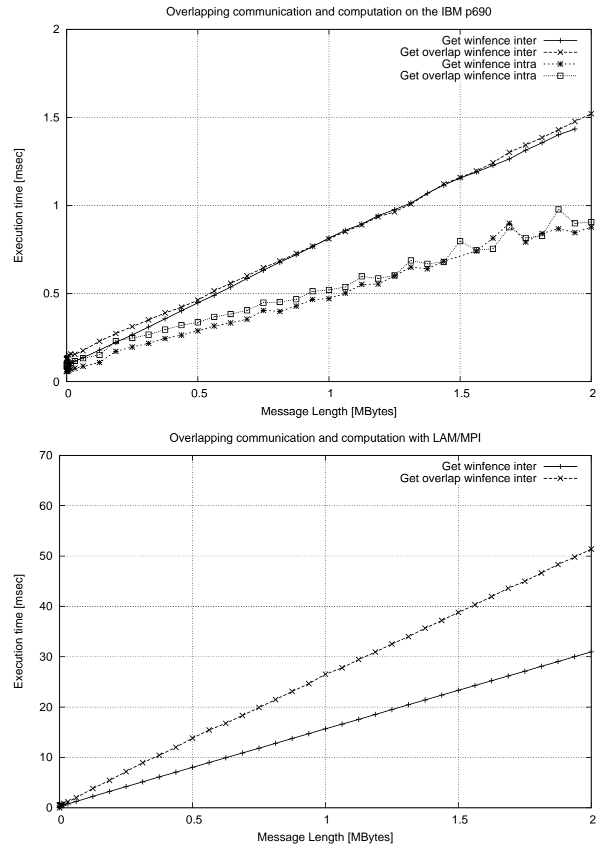

Figure 12: Overlapping computation and one-sided communication with IBM MPI (upper) and LAM/MPI (lower).

5 Summary

In this paper we presented our experiences and the performance of various MPI libraries, with respect to the handling of dynamically created communicators and one-sided communication operations. Handling of dynamic process management is still one of the chapters of the MPI-2 specification, which is implemented by few MPI libraries so far. The results achieved in our analysis show however, that the usage of these functions are not imposing any performance penalties to the end-user per se. If problems are arising with respect to dynamic process management, then according to our experience they are not related to their implementation in MPI. Problems are usually caused by the runtime environment and/or batch queue systems and their restrictive handling of dynamically created processes.

The section covering one-sided communication is supported by many MPI implementations. The analysis of their performance reveals, that the synchronization costs of one-sided operations are on all current implementations fairly high. This leads to the conclusion, that one-sided operations should not be used, if the overall amount of data transferred between two processes is small. The achievable bandwidth with one-sided operations is on all analyzed platforms close to the bandwidth achieved with two-sided communication. The only situation where a performance benefit of one-sided communication over two-sided communication could be observed was, if the memory used in the data transfer had been allocated through a special MPI function. This functionality of providing "fast" memory is however not widely supported by current MPI libraries. The overall conclusion of this analysis therefore has to be, that users should not switch from two-sided communication to one-sided for performance reasons, but only in cases where it matches the communication pattern of their application better than two-sided operations.

The interpretation of the results presented are twofold: on one hand, we see a strong variance with respect to the performance of many MPI-2 functions, which can confuse the application developer and influence their decision whether to use MPI-2 functionality or not. On the other hand, the goal of MPI-libraries cannot be to make everything "equally slow", but to take advantage of as many optimization possibilities as possible. Additionally, one should acknowledge, that the library developers invested huge efforts to optimize MPI-1 functionality, efforts, which might be invested in MPI-2 functions as soon as these sections are more widely used.

Acknowledgements

The authors would like to thank the University of Tennessee Knoxville, the High Performance Computing Center Stuttgart (HLRS), the University of Ulm and the Forschungszentrum Juelich for providing their machines used in our tests. Furthermore, we would like to acknowledge the support of Holger Berger from NEC Europe.

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.

[9] W. Gropp and E. Lusk *Reproducible measurements of MPI performance characteristics* in J.J. Dongarra, E.Luque and T. Margalef (Eds.), 'Recent advances in Parallel Virtual Vachine and Message Passing Interface, pp. 11-18, Springer, 1999.

[10] R. H. Reussner, P. Sanders, L. Prechelt and M. Müller *SKaMPI: A Detailed Accurate MPI Benchmark*, in V. Alexandrov and J. J. Dongarra (Eds.), 'Recent advances in Parallel Virtual Vachine and Message Passing Interface, pp. 52-59, Springer, 1999.

[11] R. Hempel *Basic message passing benchmarks, methodology and pitfalls* Presented at SPEC Workshop, Wuppertal, Germany, Sept. 1999.

[12] E. Gabriel, M. Resch and R. Ruehle *Implementing and Benchmarking Derived Datatypes for Metacomputing* in B. Hertzberger, A. Hoekstra, R. William (Eds.) 'High Performance Computing and Networking', pp. 493-502, Springer, 2001.

[13] E. Gabriel, G. E. Fagg and J. J. Dongarra *Evaluating the Performance of MPI-2 dynamic communicators and one-sided operations* in Jack J. Dongarra, Domenico Laforenza, Salvatore Orlande (Eds.), 'Recent Advances in Parallel Virtual Machine and Message Passin Interface', Lecture Notes in Computer Science vol. 2840, pp.88-97, Springer 2003.

[14] R. Thakur, W. Gropp, and B. Toonen *Minimizing Synchronization Overhead in the Implementation of MPI One-Sided Communication*, in Dieter Kranzlmueller, Peter Kacsuk, Jack J. Dongarra (Eds.), 'Recent Advances in Parallel Virtual Machine and Message Passing Interface', Lecture Notes in Computer Science vol. 3241, pp. 57-67, Springer 2004.