Evaluating Sparse Data Storage Techniques for MPI Groups and Communicators

Mohamad Chaarawi and Edgar Gabriel

Parallel Software Technologies Laboratory,
Department of Computer Science, University of Houston,
{mschaara,gabriel}@cs.uh.edu

Abstract. In this paper we explore various sparse data storage techniques in order to reduce the amount of memory required for MPI groups and communicators. The idea behind the approach is to exploit similarities between the objects and thus store only the difference between the original process group and the resulting one. For each technique, we detail the memory saved compared to the currently used implementations, and present a runtime decision routine capable of choosing dynamically the most efficient technique for each scenario. Furthermore, we evaluate the performance impact of the new structures using point-to-point benchmarks as well as an application scenario over InfiniBand, Myrinet and Gigabit Ethernet networks.

1 Introduction

The memory footprint of a process running the MPI equivalent of 'hello world' can reach tens of Megabytes on today's platforms. Some of the factors contributing to the large memory footprint are related to optimizations within the MPI library code base, such as using statically allocated memory or having many different code paths in order to optimize a particular operation. The larger fraction of the memory utilized by an MPI library is however allocated dynamically at runtime and depends on system parameters such as the network interconnect and application parameters such as the the number of processes.

While reducing the memory footprint of an MPI process was not considered to have a high priority for a while, recent hardware developments force us to rethink some concepts used within communication libraries. Machines such as the IBM Blue Gene/L [4] have the capability to run MPI jobs consisting of more than 100,000 processes. At the same time, each node only has 512 MB of main memory available, leading to 256 MB for each MPI process. A similar problem occurs on commodity clusters due to the increasing number of cores per processor [2] giving the end-users the possibility to run parallel jobs consisting of a large number of MPI processes, while at the same time the main memory per core remains constant at best.

For platforms facing the problem outlined above, an MPI library should avoid internal structures having a high level dependency on the number of MPI processes. In this paper we are exploring various sparse data storage techniques in order to reduce the amount of memory required for MPI groups and communicators. The idea behind the approach is to exploit similarities between the objects. Instead of storing the entire list of processes which are part of a new group or communicator, the approach presented in this paper stores only the difference between the original communicator and the resulting one. These techniques might not only be relevant for very large number of processes, but will also be beneficial for applications having a moderate number of processes, generating however a very large number of communicators [1].

Much of the work on optimizing memory usage within MPI libraries has focused so far on the networking layer. Panda et al. show in [12] the benefits of using the Shared Receive Queue features of InfiniBand in order to reduce the memory usage of large scale applications. Shipman et al. [11] introduce a new pipelining protocol for network fabrics dealing with registered memory, which further reduces the memory utilization for these network interconnects. The work done in [5] focuses on controlling the number of unexpected messages a process can handle in order to limit the memory usage of the MPI library.

The remainder of the paper is organized as follows: Sec. 2 discusses briefly the current implementation of groups and communicators in Open MPI, and presents three different sparse data storage techniques. Sec. 3 evaluates the performance impact of the techniques detailed in the previous section using a point-to-point benchmarks as well as the High Performance Linkpack (HPL) benchmark. Finally, Sec. 4 summarizes the paper and presents the currently ongoing work in this area.

2 Alternative Storage Formats for Groups and Communicator

In most MPI libraries available today, each MPI group contains the list of its member processes. In Open MPI [6], each entry of the list is a pointer to the according process structure, while in MPICH2 [8] the according list contains the ranks of the processes in `MPI_COMM_WORLD`. The position in this array indicates the rank of the process in this group. An MPI communicator contains typically pointers to either one MPI group for intra-communicators, or two groups for inter-communicators. While this approach guarantees the fastest access to the process structure for a communication – and thus minimal communication latencies – the information stored in those arrays is often redundant. For example, in case a numerical library creates a duplicate of `MPI_COMM_WORLD` in order to generate a unique communication context, the communicator structure will contain a process list with redundant information to the original communicator. For a 100,000 process job, this list will take $8 \times 100,000$ bytes of memory, assuming that the size of a pointer is 8 bytes.

Therefore, three alternative storage formats have been evaluated in this study in order to minimize the redundant information between different process groups and thus minimize the memory footprint of the according structures. In order to evaluate benefits and disadvantages of these storage formats, we have implemented all formats in Open MPI. For this, the group structure and the group management functions had to be adapted. The following subsections give some details to each format.

PList Format: The PList storage format is the original storage format containing a list of pointers to the process structures of the group members. This implementation of this format is unchanged compared to the original version.

Range Format: For this format, the included processes in a group are described by ranges of process ranks, e.g. having n consecutive processes starting from rank r. The syntax of this storage format has been derived from the `MPI_Group_range_excl/incl` functions in MPI [9]. Thus, the group structure in Open MPI has been extended by a list holding the required number of < base rank, number of processes> pairs in order to describe the members of the new group. The base ranks stored in the group-range list correspond to the rank of the process in the original group. Thus, the group structure also needs to store a pointer to the original group and increase the reference counter of that object accordingly. While this storage format can be applied to any group/communicator, it will be most memory efficient if the list of ranks included in the new process group can be described by a small number of large blocks.

Strided Format: In some cases, the included processes in a group follow a regular pattern, e.g. a new group/communicator includes every n-th process of the original group. Three integers are required in the group structure in order to support this format: the `grp_offset` contains the rank of the process where the pattern starts; `grp_last_elt` is the rank of the last process in the pattern; the `grp_stride` describes how many processes from the original group have to be skipped between two subsequent members of the new group. The group creation function for strided groups can automatically determine all three parameters. In case no regular pattern could be determined by the routine, the strided group creation function will indicate that it can not be used for this particular process group. Similarly to the range format, a pointer to the original group is required to be able to determine the process structures.

Bitmap Format: The main idea behind this storage format is to use a bit-array of the size of the original communicator/group. The bit at position `i` indicates, whether the process with the rank i in the original group is a member of the resulting group or not. The main restriction of this storage format is that ranks of the included processes in the new group have to be monotonically increasing in order to be able to uniquely map the rank of a processes from one group to another group.

2.1 Group Management Operations

Additionally to the functionality outlined in the previous subsections, each storage format provides a function which estimates the amount of main memory

Table 1. Memory consumption of each storage format.

Storage format	Memory consumption
PList	number of processes × `sizeof(void *)`
Range	number of ranges ×2× `sizeof(int)`
Strided	3× `sizeof(int)`
Bitmap	⌈(number of processes /8)⌉

required by this format, given the list of process members. Whenever a new group or communicator is created, these functions are queried in order to decide which of the storage techniques will be applied. Table 1. summarizes the formula used to estimate the memory consumption of each format. The current runtime decision logic will then pick the storage format requiring the least amount of memory. A flag in the group structure indicates which storage format is being used for a particular group. The groups used by `MPI_COMM_WORLD` and `MPI_COMM_SELF` are always stored using the PList format.

The most performance sensitive functions with respect to group and communicator management are returning a pointer to the process structure given a tuple of <rank, communicator id>, and the ability to translate the rank of a process given in one process group into its rank in another process group. The first functionality is utilized by any data transfer operation, since the process structure contains elements such as the contact information and the communication peers. In the original implementation of MPI groups in Open MPI, access to a process structure is as simple as using the PList array by specifying the rank of the process in the group as the index in the array. Since this array is set to `NULL` whenever a format other than the PList format is being used, this approach is not applicable for multiple storage formats. Thus, a macro, `GROUP_GET_PROC_POINTER`, has been introduced. This macro translates to the original access to the process pointer list in case the support for multiple storage formats has been turned off by the user. In case the support for multiple storage formats has been enabled, the macro would be replaced by a function call determining the according pointer.

To get the pointer to the process structure of a given rank in a group is a two step procedure for any of the alternative storage formats presented so far. First, the function determines the rank of this process in its parent group and second, it will query the original parent group using the rank of the process in that group for the according pointer. In case the parent communicator uses a sparse storage format as well, this procedure will be repeated as long as the algorithm hits a group which uses the PList format. Since `MPI_COMM_WORLD` and `MPI_COMM_SELF` always use the PList format, the algorithm is guaranteed to determine the correct process pointer in any scenario.

Another operation which has to be adapted in order to support multiple storage formats is the group rank translation function. Although it would be easy to generate a universal implementation which works for all storage formats, there are certain special cases where the performance of this operation can be significantly improved by taking into account the relation between the groups, e.g. in case one group has been derived from the other group. Unfortunately, we can not detail the according formulas in this paper due to space limitations, please refer to [3].

The most severe restriction of the current approach is that all storage formats assume as of now that a group is derived from a single parent group. This is not the case for the `MPI_Group_union` and `MPI_Group_difference` operations as well as for `MPI_Intercomm_merge`. For these functions, the implementation will automatically fall back to the PList format. For similar reasons, the current implementation does not support inter-communicators.

3 Performance Evaluation

This section evaluates the performance implications of the various storage formats presented in the previous section. Two separate set of tests have been conducted, namely a point-to-point benchmark in order to quantify the effect on the latency and an application benchmark. The machines used for the tests were the shark cluster at the University of Houston and the IBM BigRed cluster at Indiana University. Shark consists of 24 dual-core 2.2GHz AMD Opteron nodes connected by a 4xInfiniBand and a Gigabit Ethernet network interconnect. BigRed, which was mainly used for the point-to-point benchmarks, consists of 768 IBM JS21 Blades, each having two dual-core PowerPC 970 MP processors, 8GB of memory, and a PCI-X Myrinet 2000 adapter. Within the scope of this analysis we used up to 512 MPI processes on 128 nodes on BigRed.

3.1 Point-to-point Benchmark

In order to evaluate the effect on the point-to-point performance of Open MPI when using the alternative storage formats for groups and communicators, we created in a new test within the *latency* test suite [7]. The basic idea behind the *latency* test suite is to provide building blocks for ping-pong benchmarks, such as different data type constructors, communicator constructors, or data transfer primitives. This allows users to set-up their own point-to-point benchmarks, e.g. by mimicking a particular section of their applications.

The new test case developed within this project creates a hierarchy of communicators. Starting from the processes in `MPI_COMM_WORLD` the test excludes all odd-ranked elements of the communicator. Using the resulting communicator the benchmark keeps creating new communicators excluding odd ranked elements until a communicator consisting of only one or two processes is being created. For each new communicator, a ping-pong benchmark will be executed between the first and second process in one case, and between the first and last process in another case. An additional overhead to the communication latency is expected when executing with the sparse formats, which comes from the fact that getting the actual process pointer for each data transfer operation requires

Table 2. Results of the point-to-point benchmark running 48 processes over InfiniBand.

	PList		Range		Strided		Bitmap	
	0-first	0-last	0-first	0-last	0-first	0-last	0-first	0-last
Level 0	3.7	3.7	3.7	3.7	3.7	3.7	3.7	3.7
Level 1	3.7	3.7	3.74	3.74	3.74	3.71	3.74	3.74
Level 2	3.7	3.7	3.74	3.79	3.74	3.74	3.74	3.79
Level 3	3.74	3.7	3.74	3.85	3.74	3.74	3.79	3.79
Level 4	3.74	3.7	3.85	3.85	3.8	3.8	3.8	3.85
Level 5	3.74	3.7	3.9	3.85	3.8	3.8	3.9	3.9

some additional computation and lookup operations. This effect is supposed to increase with the increase in the hierarchy of groups depending on each other.

For each implementation of the groups (plist - range - sparse - bitmap), the test was executed five times. The results that are provided show the minimum latency of the all the tests executed. This shows the best achievable result on the according cluster. Times are given in μs.

The results on shark over the InfiniBand network interconnect are shown in Table 2. The level of each communicator shown in the first column of the table indicates the number of indirections required to look up the process structure. For level 0 (= `MPI_COMM_WORLD`) the latency is independent of the storage format, since this communicator is always using the PList format. Furthermore, there is no performance difference for level 0 whether the ping-pong benchmark is executed between the first and the second process, or between the first and the last process of the communicator. As expected, the latency is mostly constant for the original PList format, and thus independent of the communicator used. For the other formats, the latency does increase depending on the level of the communicator, i.e. the number of indirections required to lookup the process structure. In order to quantify the overhead, lets consider the highest overhead observed in our measurements, which adds 0.2μs to the original latency. Accessing the process structure for that particular communicator level requires 5 indirections of the algorithm described in section 2.1. Thus, the average overhead per level can be estimated to be up to 0.04μs on this architecture.

For the bitmap and the range formats, we also would have expected to see a slight increase in the latency when executing the ping-pong benchmark between the first and the last process, compared to the first and the second process. The reason for this is that the costs for the rank-translation algorithms for these two formats should increase with the rank being translated, since we have to walk linearly through the list of participating processes. However, due to the fact that our maximum job size is only 48 processes and that the number of processes decreases by a factor of two with each level of communicator, we could not observe in these benchmarks the expected effect. There are slight differences in the performance of the alternative storage formats, with strided being slightly faster than the bitmap and the ranges format. The reason for this is probably the rank-translation algorithm, which only requires applying a simple formula

Table 3. Results of the point-to-point benchmarks running 48 processes over Gigabit Ethernet.

	PList		Range		Strided		Bitmap	
	0-first	0-last	0-first	0-last	0-first	0-last	0-first	0-last
Level 0	51.55	51.84	51.61	51.39	51.11	52.14	51.55	51.2
Level 1	51.61	52.45	51.65	52.34	52.09	52.95	52.05	52.8
Level 2	51.7	52.7	51.59	53.8	51.55	52.64	51.75	52.75
Level 3	51.09	52.3	51.45	53.19	51.4	52.4	51.15	51.86
Level 4	50.8	51.45	51	51.81	51.75	51.8	51.25	52.6
Level 5	51.7	51.5	51.86	51.4	51.55	51.14	51.61	51.7

Table 4. Results of the point-to-point benchmark running 512 processes over Myrinet.

	PList		Range		Strided		Bitmap	
	0-first	0-last	0-first	0-last	0-first	0-last	0-first	0-last
Level 0	6.25	7.34	6.04	7.45	6.25	7.40	6.10	7.25
Level 1	6.25	7.40	6.29	7.65	6.29	7.44	6.29	9.05
Level 2	6.09	7.30	6.25	8.00	6.29	7.45	6.29	9.80
Level 3	6.20	7.30	6.35	8.00	6.29	7.39	6.35	10.14
Level 4	6.21	7.20	6.35	8.06	6.40	7.55	6.60	10.44
Level 5	6.25	7.25	6.45	8.30	6.40	7.55	6.75	10.50
Level 6	6.75	7.25	7.19	8.00	6.85	7.40	7.56	10.19
Level 7	6.75	7.25	7.20	8.05	6.89	7.55	7.95	10.05
Level 8	6.76	7.36	7.40	8.00	7.06	7.55	8.80	9.30

for the strided format, compared to a slightly more complex algorithm for the other two sparse storage formats.

Table 3. summarizes the performance results on shark over Gigabit Ethernet. In our measurements, no performance effects could be observed which could be directly related to the sparse data storage formats used for groups and communicators. The reason for that is, that the perturbation of the measurements using this particular switch was higher than expected overhead, assuming that the overhead due to the different storage formats would be in the same range as for InfiniBand results presented previously.

Table 4. presents the results obtained for a 512 processes run on 128 nodes on Big Red. In order to ensure, that the same network protocol is used for all communicator levels, the `0-first` tests have been modified such that the first MPI processes on the first two nodes are being used for the first three communicators (levels 0, 1, and 2).

First, we would like to analyze the results obtained using the plist format – the default Open MPI approach – on this machine. The results for this storage format are presented in the first two columns of the Table 4. The most fundamental observation with respect to the results obtained on this machine is that the latency shows a fundamentally larger variance depending on the nodes used for the ping-pong benchmark. Furthermore, there is a relevant increase in the communication latency when executing the ping-pong benchmark between the rank 0 and the last process in the communicator, compared to the results obtained in the `0-first` tests. In order to explain this effect, we made several verification runs confirming the results shown above. Furthermore, we verified this behavior for the plist format with support for sparse storage formats being disabled in Open MPI. Since we can positively exclude any effects due to the sparse storage formats for these results, we think that the most probably explanation has to deal with caching effects when accessing the process structures of processes with a higher rank.

With respect to the sparse storage formats, the results indicate a similar behavior as obtained on the shark cluster over InfiniBand. The redirections required to look up the process structure lead to a small performance penalty when using the sparse storage techniques. In order to estimate the overhead introduced by the sparse storage techniques, we compare the latency obtained for a particular communicator level with a sparse storage technique to the latency obtained on the same communicator level with the plist storage format. This overhead is then divided by the number of indirect lookup operations required for that communicator level. Since the results show a larger variance than on the shark cluster, we provide an upper bound for the overhead by reporting only the maximum values achieved in this set of tests and the average obtained over all levels.

In the `0-first` tests, the range format introduces an average overhead of $0.057\mu s$ per level, the maximum overhead found was $0.08\mu s$. The penalty on the latency per level when using the strided format in these tests was $0.04\mu s$, while the highest overhead observed in these tests using this storage format was $0.1\mu s$. The bitmap format shows once again the highest overhead, with an average penalty of $0.118\mu s$ per level, and a maximum overhead of up to $0.255\mu s$. In contrary to the results obtained on the shark cluster, the `0-last` tests show a significant additional overhead for the range and the bitmap format, which are due to the fact, that the rank-translation operation involves a linear parsing of all participating processes in that communicator. The bitmap format has an average overhead of more than $0.8\mu s$ per level in these tests, while the average overhead for the range method increases to $0.197\mu s$. As expected, the strided format does not show any sensitivity to the rank being applied in the rank-translation operation.

3.2 Application Benchmark

In order to determine the impact of the new storage formats on the performance of a real application scenario, we executed multiple test-cases of the HPL benchmark [10]. A major requirement for the application benchmark chosen for this subsection is, that the code has to create sub-communicators which expose a benefit of the sparse storage techniques. HPL organizes processes in a 2-D Cartesian process topology. Three different type of communicators are created by HPL: (I) a duplicate of `MPI_COMM_WORLD`, (II) a row communicator for each row of the 2-D Cartesian topology, and (III) a column communicator for each column of the 2-D Cartesian topology. For (I), the new group creation functions will choose the range format for process numbers larger than 64, the bitmap format otherwise. Similarly, communicator (II) can be represented by a single range of processes, while the communicator (III) is best described by the strided format. Assuming a 90,000 process run of HPL organized in a 300×300 process topology, the default implementation of groups and communicators in Open MPI would take 724,800 bytes to store the list participating processes for all three communicators per process. Using the sparse storage techniques described in this paper, the memory consumption for that scenario can be reduced to 58 bytes per process.

In the following, we present performance results for 48 processes test-cases using shark. Table 5. summarizes the measurements over the InfiniBand network interconnect. Four different test cases have been executed, namely two problem sizes (24,000 and 28,000) each executed with two different block sizes (160 and 240). Since the latency does show some dependence on the storage format used for MPI groups, we would expect to see minor increases in the execution time for highly latency sensitive applications. However, none of the test cases executed in this subsection show a significant performance degradation related to the storage format.

Table 5. Execution time of the HPL benchmark on 48 Processes using InfiniBand in seconds.

Size	Block size	PList	Range	Strided	Bitmap
24000	160	65.81	65.81	65.84	65.87
24000	240	69.05	69.08	69.22	69.14
28000	160	99.78	99.73	99.84	99.81
28000	240	104.25	104.31	104.27	104.29

4 Summary

In this paper, we introduced various storage formats for groups and communicators in order to minimize the memory footprint of the according structures. The main idea behind these formats is to store only the difference between the original group and the newly created one. Three different formats – range, strided and bitmap – have been implemented in Open MPI. Additionally to the memory consumption of each format, the paper also evaluates the performance impact of the sparse storage formats. Using a modified ping-pong benchmark, we could determine the performance overhead due to the new data storage formats to be up to 0.04μs per hierarchy level of the communicator. This overhead is negligible for most scenarios, especially when taking into account, that many applications only derive communicators directly from `MPI_COMM_WORLD`. Our tests using the HPL benchmark did not show any measurable overhead due to the new storage formats. The techniques detailed in this paper will be available with Open MPI version 1.3.

Acknowledgments. This research was funded in part by a gift from the Silicon Valley Community Foundation, on behalf of the Cisco Collaborative Research Initiative of Cisco Systems and was supported in part by the National Science Foundation through TeraGrid resources provided by Indiana University.

References

1. Open MPI users mailing lists `http://www.open-mpi.org/community/lists/users/2007/03/2925.php`, 2007.
2. K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, EECS Department University of California, Berkeley, 2006.
3. Mohamad Chaarawi. Optimizations of group and communicator operations in Open MPI. Master Thesis, Department of Computer Science, University of Houston, 2006.
4. A. Gara et. all. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49(2/3):195–212, 2005.
5. M. Farreras, T. Cortes, J. Labarta, and G. Almasi. Scaling MPI to short-memory MPPs such as BG/L. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 209–218, New York, NY, USA, 2006. ACM.
6. Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.
7. Edgar Gabriel, Graham E. Fagg, and Jack J. Dongarra. Evaluating dynamic communicators and one-sided operations for current MPI libraries. *International Journal of High Performance Computing Applications*, 19(1):67–79, 2005.
8. 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.
9. Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, June 1995. `http://www.mpi-forum.org`.
10. A. Petit, R.C. Whaley, J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. Version 1.0 `http://www.netlib.org/benchmark/hpl/`.
11. Galen M. Shipman, Ron Brightwell, Brian Barrett, Jeffrey M. Squyres, and Gil Bloch. Investigations on InfiniBand: Efficient network buffer utilization at scale. In *Proceedings, Euro PVM/MPI*, Paris, France, October 2007.
12. Sayantan Sur, Matthew J. Koop, and Dhabaleswar K. Panda. High-performance and scalable MPI over InfiniBand with reduced memory usage: an in-depth performance analysis. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 105, New York, NY, USA, 2006. ACM.