Implementing MPI-2 Extended Collective Operations[1]

Pedro Silva and João Gabriel Silva

Dependable Systems Group, Dept. de Engenharia Informática, Universidade de Coimbra, Portugal
ptavares@dsg.dei.uc.pt, jgabriel@dei.uc.pt

Abstract. This paper describes a first approach to implement MPI-2's Extended Collective Operations. We aimed to ascertain the feasibility and effectiveness of such a project based on existing algorithms. The focus is on the intercommunicator operations, as these represent the main extension of the MPI-2 standard on the MPI-1 collective operations. We expose the algorithms, key features and some performance results. The implementation was done on top of WMPI and honors the MPICH layering, therefore the algorithms can be easily incorporated into other MPICH based MPI implementations.

1 Introduction

MPI-2 brings major extensions to MPI: dynamic processes, one-sided communications and I/O [1]. Down the line of the new features come the extensions to the collective operations. These extensions open the doors to collective operations based on intercommunicators, bring a user-option to reduce memory movement and also two new operations: a generic alltoall and an exclusive scan.

The intercommunicator represents a communication domain composed of two disjoint groups, whereas the intracommunicator allows only processes in the same group to communicate [2]. Thus, due to its structure, the intercommunicator supports client-server and pipelined programs naturally. MPI-1 intercommunicators enabled users to perform only point to point operations. MPI-2 brings the generalization of the collective operations to intercommunicators which further empowers users in moving data between disjoint groups. The intercommunicator and the operations it supports are also significant because of the dynamic processes. Solely the intercommunicator enables the user to establish communication between groups of processes that are spawned or joined [3]. It thus expands beyond the bounds of each MPI_COMM_WORLD, unlike the intracommunicators.

This paper embodies a first approach to implement the MPI-2's Extended Collective Operations. We present algorithms for the intercommunicator based collective operations which draw from the existing MPICH [4] intracommunicator algorithms. The focus is on the intercommunicator based operations, as these represent the main extension of the MPI-2 standard on MPI-1's collective operations. We also present some performance results. The objective of this work was to ascertain

[1] This work was partially supported by the Portuguese Ministry of Science and Technology (MCT), under the Programme Praxis XXI.

the feasibility and effectiveness of implementing the operations based on the existing collective algorithms.

There have been many research projects on MPI collective operations. Most of these projects concentrate on specific characteristics of the computation environment. Some focus on the interconnection hardware, such as the LAMP project which set its sights on a Myrinet [5]. Other on the interconnection topology: The collective algorithms for WANs, a project conducted by Kielmann et al. [6], or the MPI-CCL library project that assumed a shared medium [7]. Yet others, like [8] and [9], provide algorithms that attain efficient collectives on heterogeneous NOWs. However, there are no projects that cover the collective operations based on intercommunicators, to the best of our knowledge.

We will show that a simple implementation is feasible using the algorithms from the existing collective operations based on intracommunicators. We also present some performance figures to show that the intercommunicator functions have a good performance and scale as well as the intracommunicator counterparts. This work represents a first step into incorporating the extensions that MPI-2 brings to collective operations on WMPI[10][11]. WMPI is a full implementation of MPI-1 for SMPs or clusters of Win32 platforms. Due to the fact that the top layers of WMPI are based on MPICH, the implementation of the extended collective operations can be easily added to MPICH or others alike.

1 Algorithms

This section lays out the algorithms of the intercommunicator operations. The algorithms are based on existing intracommunicator algorithms because these are simple. First, we present the communication patterns of the rooted operations, which fall into two categories: One-to-all where one process contributes data to all other processes; all-to-one, where all processes contribute data to just one process. Second, we present the algorithms for the all-to-all operations, where all processes contribute data and all processes receive data. Finally we detail the barrier operation.

1.1 Rooted Operations

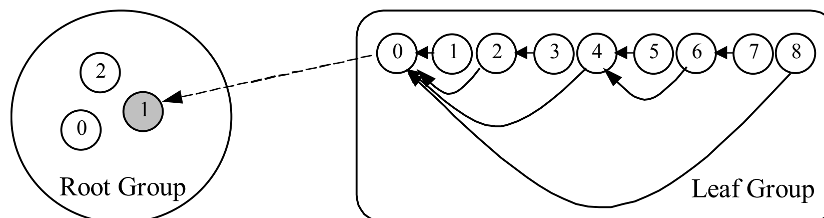

Fig. 1 – Intercommunicator Reduce (12 processes)

MPI_Reduce involves a reduce operation (such as sum, max, logical OR) with data provided by all the members of the leaf group, whose result is stored at the user-

defined root process. The implementation of the MPI_Reduce operation draws heavily from its homonymous intracommunicator operation. The reduction is made up of a local reduce (i.e. with the local group's intracommunicator) in the leaf group and a send from the leaf group's leader to the user defined root process. Figure 1 displays the message passing involved in the intercommunicator reduction operation. The processes to which the user passes MPI_PROC_NULL in the *root* argument (represented in Fig. 1 by the processes ranked 0 and 2 in the root group), call no message passing routines. These are completely passive in the operation and return immediately. This applies to the other rooted operations as well.

MPI_Bcast involves the passing of data from the user defined root to all processes in the leaf group. The MPI_Bcast function is also based on a recursive splitting tree algorithm, as the reduce. The root sends the user-provided data to the remote group's leader.

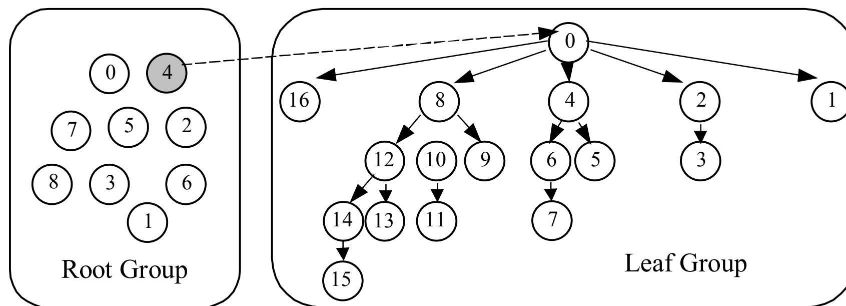

Fig. 2. – Intercommunicator Broadcast (26 processes)

At the leaf group the data is broadcast via a local intracommunicator based broadcast. Figure 2 displays the message passing that each process performs; process with local rank 4, on the left, is the user defined root. The local broadcast, as the local reduce in MPI_Reduce, have O(log N) time complexity, where N is the number of processes.

MPI_Gather has the user defined root collect data from all the processes in the leaf group. The MPI_Gather(v) operations are based on a linear algorithm that is similar to its intracommunicator counterpart. The user defined root process posts immediate receives to obtain the data from each process in the leaf group. In turn, each of these processes sends one message to the root process. This is a direct generalization from the intracommunicator gather operations.

MPI_Scatter has all processes in the leaf group provide data to the user defined root process. The MPI_Scatter(v) operations also have a linear implementation. The root process sends a distinct chunk of data to each of the processes in the leaf group. The root passes the messages via immediate sends to all processes and then waits for these to finish. The recipients simply call a blocking receive. As the aforementioned operations this one draws directly from the existing intracommunicator based counterparts. The scatter and gather operations have O(N) algorithms.

1.1 All-to-All Operations

The MPI-2 standard implies that the MPI_Reduce_scatter should be thought of as an intercommunicator reduce, followed by an intracommunicator scatter. However, at least for a direct implementation and using the current reduce and scatter algorithms, an intra reduce followed by an inter scatterv is generally less costly. The first involves passing the results (obtained from the reduce operation) from each group to the other and only then does the leader process perform the scatter. On the other hand, the latter involves sending the results directly to the final recipient of the remote group. Thus there is one less hop in passing the results through the communication medium. In order for the inter scatterv to take place, the leaders must obtain the *recvcounts* argument from the other side. The fact that the latter algorithm involves this exchange is of no real consequence because it will probably be less heavy than transferring the results and, mainly, because the exchange is overlapped with the reduction operation. The leaders exchange the *recvcounts* and only then do they perform the reduce operation, thus taking advantage of the fact that they are the last to intervene in the local reduction operation. As for the inter scattervs, one for each group, these also lap over because they are based on non-blocking operations.

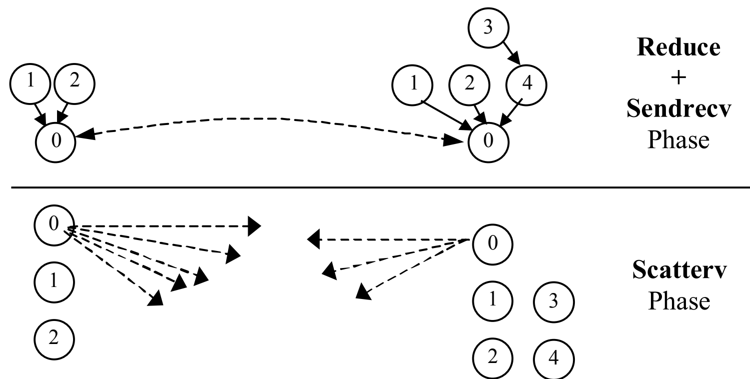

Fig. 3. – Intercommunicator Reduce_scatter (8 processes)

The MPI_Reduce_scatter operation was thus implemented as an intracommunicator reduce, followed by two intercommunicator scattervs. Figure 3 shows the three communication constituents of the operation in two phases – the exchange of *recvcounts* between the leaders, the simultaneous reduce on each group, and finally the scatter of data from the leaders to each member of the leaf group.

MPI_Allgather has each group provide a concatenation of data items from every member process to the other group. The MPI_Allgather and MPI_Allgatherv may be implemented in a straightforward manner which has two phases. First, every process from each group sends the data to the leader of the other group. The leader waits for the corresponding immediate receives to conclude and then broadcasts the data it received to the processes in its local group. The remainder of the processes in each local group join the leader in the broadcast. This function is actually very different

from its intracommunicator counterpart which is based on MPI_Sendrecvs between neighboring processes.

MPI_Alltoall has each process from group A provide a data item to every process in group B, and vice versa. The MPI_Alltoall(v)(w) family of operations are composed of immediate receives and sends between each pair of processes from opposing groups. In an attempt to diminish the contention on the communication infrastructure, one group posts the receives first and then the sends while the other first posts the sends and only then posts the receives. When the intercommunicator is constructed a variable is set that identifies which group is elected to go first.

MPI_Allreduce has the result of the reduction from group A spread to all members of group B, and vice versa. The MPI_Allreduce operation is a three phase operation. First each group participates in an intracommunicator reduce. Second, the first elected root group broadcasts the results to the remote group. Third, the groups perform another inter broadcast where the roles are reversed. This way the reduction operation happens simultaneously on both groups. Also, the broadcast may be overlapped as the processes responsible for the distribution of data (i.e. the roots of each broadcast tree) are also the ones that return earlier from the first inter broadcast.

1.1 MPI_Barrier

MPI_Barrier is a synchronization operation in which all process in group A may exit the barrier when all of the processes in group B have entered the barrier, and vice versa. This operation is based on a hypercube, just as the intracommunicator barrier. To implement the algorithm based on a hypercube, the two groups that make up the intercommunicator are merged; one group keeps its local ranks while the other groups' ranks are shifted (we add the remote's size). In order to use the algorithm based on a hypercube one only has to translate the rank and check whether to use the intracommunicator or the intercommunicator. This way, we kept the algorithm simple through a direct generalization of the intracommunicator barrier.

1 Performance Tests

This section presents test results of some of the intercommunicator operations. To help quantify the results, these are juxtaposed with results obtained with the intracommunicator operations under the same conditions. To better compare the operations, the intercommunicator based operations that are rooted have only one process in the root group. This way the intercommunicator operation is closer to that of the intracommunicator operation; it avoids MPI_PROC_NULL processes in the root group. As for the all-to-all operations, the intercommunicator is divided evenly, each group has four processes. The testbed consisted of eight Windows NT 4.0 workstations (Pentium II based with 128MBytes of RAM), interconnected with a 100Mbps switch. These served as the remote communication testbed. The tests on shared memory were performed on a dual Pentium Pro NT 4.0 machine.

1.1 MPI_Reduce

Most of the intercommunicator collective operations perform as well as their intra counterparts. The tests on the MPI_Reduce, summarized in Figure 4, show that reduce operation is one such case. The inter reduction takes about the same amount of time as the intra one, on average over all processes.

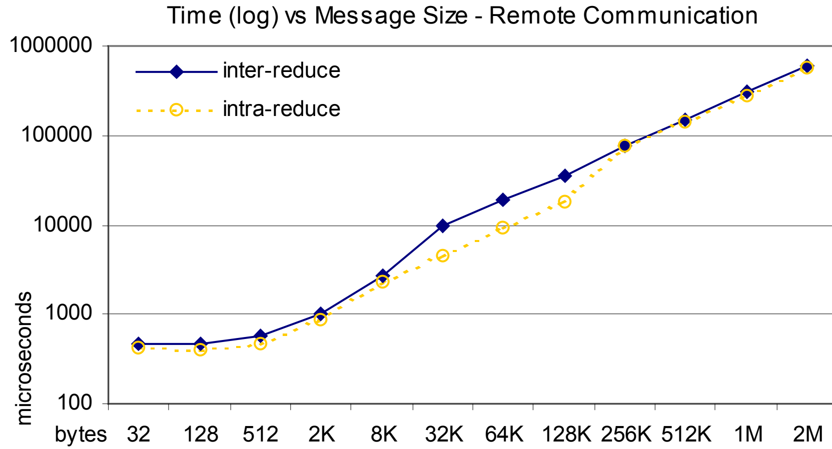

Fig. 4. – Intercommunicator and intracommunicator Reduce on 8 processes

A closer look at the execution time for each process revealed that the leaf processes, which only send data, take about the same amount of time on intra and inter operations. The processes that are truly responsible for the difference, small as it is, are the user defined root and the leaf group's leader which perform the last receive and send, respectively. The same tests in shared memory also resulted in similar execution times between the intra and inter reduce.

1.1 MPI_Reduce_scatter

Table 1 shows the results of MPI_Reduce_scatter via remote communication for a 2MB message size.

Size	**Execution time of each participant process** (microseconds)							
(bytes)	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
2097152	1445076	1435866	1439482	1416597	1471134	1439796	1443290	1391354

Table 1. Intercommunicator Reduce_scatter on 8 processes via remote communication

The results show that the inter scatters overlap well since all processes have a similar execution time. Also, as expected, the leaders (ranked 0 and 4) take longer to finish as these are responsible for scattering the results to the processes of the remote group. The intra reduce scatter execution times are above the ones for the inter, for instance, for a 2MB of data it takes on average, over all processes, 1803691 microseconds. This is due to the fact that all processes participate actively in both the reduction and the scatter of data while in the inter only half perform each operation.

1.1 MPI_Allgather

In MPI_Allgather's case, the inter execution times are well below those of the intra allgather. The allgather results are evidence of just how different the intra operation is from the inter allgather. This is due mainly to the fact that the intra allgather uses blocking send and receives while the inter allgather uses non-blocking routines to take advantage of the overtaking rule in MPI. This way the leader may receive from each process out of order. The intra version cannot use non-blocking sends and receives because each process must first receive the data before forwarding the data to its neighbor. In distributed memory, the execution times were quite further apart as one pays a high price for passing messages over the relatively slow network. The test results for the MPI_Allgather operation are presented in Figure 5.

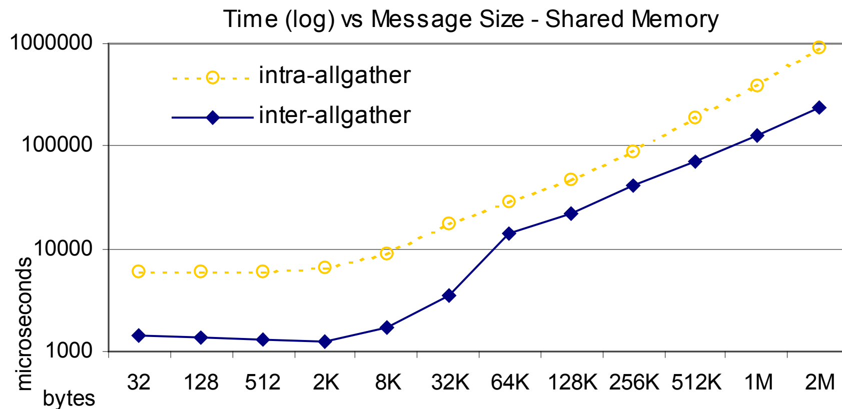

Fig. 5. – Inter and intracommunicator allgather on 8 processes via shared memory

1 Conclusions and Future Work

In this paper we presented a first step towards incorporating the MPI-2 collective operations' extensions into WMPI. We illustrated algorithms for intercommunicators based on the existing intracommunicator collective operations from MPICH. The implementation strives to avoid using the always expensive message passing (especially in networks) and also data movement within each process. We conclude that the operations can easily be implemented and have a good performance in fairly homogeneous environments. Furthermore, the operations have been tested and will provide a useful communication infrastructure for the implementation of the rest of MPI-2 on WMPI.

The next step for the collective operations is to provide algorithms for heterogeneous environments. NOWs, which are commonly heterogeneous, are becoming more and more attractive for cost-effective parallel computing. In order to truly take advantage of such an environment, one needs to incorporate some form of adaptability into the collective algorithms. Specifically, if one considers networked

systems that are composed of SMPs, the bandwidth alone is astronomically different between processes on one machine and processes on distinct machines. ECO [8] and [9], mentioned above, already have shown that adaptability is computationally worthwhile. One way of attaining such adaptability, with low performance cost, is to analyze the computing environment before the computation. ECO uses such a strategy. ECO's library comes with an offline tool which tries to guess the network topology by analyzing the latencies between every pair of computers. In most cases, the tool produces communication patterns that result in efficient collective operations.

A similar approach might work for WMPI, however, due to the fact that MPI-2 brings dynamic processes, this tool cannot be offline. One way to circumvent the problem is to analyze the interconnections between processes every time a communicator is built and include a distribution tree, or another auxiliary structure, in the communicator. With respect to building the communication trees without compromising the performance, [9] proposes two approaches. The Speed-Partitioned Ordered Chain and the Fastest Node First approaches are of low complexity and generate communication patterns for fast and scalable collective operations.

1 Bibliography

1. MPI Forum – http://www.mpi-forum.org
2. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference, Vol. 1 – The MPI Core* (2nd Edition). MIT Press, 1998.
3. W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference, Vol.2 - The MPI Extensions*. MIT Press, 1998.
4. MPICH – http://www-unix.mcs.anl.gov/mpi/mpich/
5. Local Area Multiprocessor at the C&C Research Laboratories, NEC Europe Ltd. http://www.ccrl-nece.technopark.gmd.de/~maciej/LAMP.html
6. T. Kielmann, R. Hofman, H. Bal, A. Plaaat, and R. Bhoedjang. *MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems*. In Symposium on Principles and Practice of Parallel Programming, Atlanta, GA, May 1999.
7. Jehoshua Bruck, Danny Dolev, Ching-Tien Ho, Marcel-Catalin Rosu, and Ray Strong. *Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations*, 7th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA-95 Santa Barbara, California, July 1995.
8. B. Lowekamp and A. Beguelin. *ECO: Efficient Collective Operations for Communication on Heterogeneous Networks*. In International Parallel Processing Symposium, pages 399-405, Honolulu, HI, 1996.
9. M. Banikazemi, V. Moorthy, and D. Panda. *Efficient Collective Communication on Heterogeneous Networks of Workstations*. In International Conference on Parallel Processing, pages 460-467, Minneapolis, MN, August 1998.
10. J. Marinho and J.G. Silva. *WMPI – Message Passing Interface for Win32 Clusters*. In Proc. of the 5th European PVM/MPI Users' Group Meeting, pp. 113-120, September 1998.
11. WMPI – http://dsg.dei.uc.pt/wmpi