SMP-Aware Message Passing Programming

Jesper Larsson Träff

C&C Research Laboratories, NEC Europe Ltd.
Rathausallee 10, D-53757 Sankt Augustin, Germany
`traff@ccrl-nece.de`

Abstract

The Message Passing Interface (MPI) is designed as an architecture independent interface for parallel programming in the shared-nothing, message passing paradigm. We briefly summarize basic requirements to a high-quality implementation of MPI for efficient programming of SMP clusters and related architectures, and discuss possible, mild extensions of the topology functionality of MPI, which, while retaining a high degree of architecture independence, can make MPI more useful and efficient for message-passing programming of SMP clusters. We show that the discussed extensions can all be implemented on top of MPI with very little environmental support.

1 Introduction

Although designed for programming of distributed memory parallel systems in the message-passing paradigm, the *Message Passing Interface* (MPI) [6, 18] is also used for parallel programming on shared memory systems and hybrid shared/distributed memory systems, such as clusters of SMP nodes. Alternatively, clusters of SMP nodes can be programmed in a hybrid style, using OpenMP [4] or a thread-model within the SMP nodes and a message passing interface for the communication between nodes. Arguably, the hybrid style can give better performance, since it reflects more closely the structure of the SMP system. On the other hand, hybrid programming is difficult since it requires mastering both the shared-memory and message passing programming styles. For lack of a simple and broadly accepted model for programming of hybrid systems, a pure message passing paradigm is often preferred. Also, many existing application codes are pure MPI codes, and the extra effort required to rewrite these is considerable. Surprisingly, experiments indicate that pure MPI codes can often be as fast as specialized hybrid codes [2, 9, 15, 17]. However, in order to be efficient on an SMP cluster, the MPI implementation must take the hybrid nature of the system into account. This means that both point-to-point, one-sided and collective communication operations of MPI must be implemented to take advantage of the faster, shared-memory based communication within SMP-nodes. In addition, the *topology functionality* of MPI [18, Chapter 6] can be implemented to perform process reordering based on user-supplied communication patterns to take better advantage of the more powerful intra-node communication. With MPI, however, it is not possible for the user to *explicitly* take the communication structure of an SMP cluster into account. This may be seen as both a strength and a weakness of MPI.

In this paper we first summarize requirements to high-quality MPI implementations for efficient utilization of SMP clusters and other systems with a hierarchical communication structure. In Section 3 we discuss the MPI topology functionality, which can be used to provide an architecture independent means of adapting to SMP-like architectures. We discuss the potential of this functionality, and also some of its shortcomings that could possibly be remedied without compromising or unnecessarily extending the existing MPI standard. In Section 4 we discuss means of incorporating explicit SMP-awareness into an MPI-like programming interface. We stress that the discussed "extensions to MPI" can all be implemented on top of MPI with only minimal environmental support needed.

As ever so often there is a trade-off between precision and efficiency of a proposed mechanism, and its ease of use. The more precise a mechanism, the more knowledge and effort is required for its use. Thus, proposed extensions to a library like MPI must consider questions like: Is it worth the effort? Will the user accept the extended/new functionality? Are the performance benefits large enough? It is worth noticing that the MPI standard as it is, is often criticized for being too large, and many aspects are not used, either because of lack of user knowledge or because of skepticism about the performance (warranted or not). An unfortunate consequence is that MPI implementers sometimes spend too little time on these more exotic parts of MPI (or was it the other way round?).

2 SMP-aware communication

An SMP cluster is a collection of shared-memory processing nodes interconnected by a communication network. SMP clusters range from low-cost, off-the shelf systems with few processors per node (2-way, 4-way clusters) interconnected with a cheap commodity network (Fast Ethernet), through medium- to high-performance clusters with powerful interconnects (Myrinet, SCI, Giganet, Quadrics), to the currently most powerful supercomputers like the multi-way ASCI-machines, the Earth Simulator or the NEC SX6-multi-node systems, all equipped with specialized, high performance interconnects. A common characteristic of these systems is a markedly hierarchical communication structure. Processors on the same shared-memory node can communicate via the shared memory. Typically the shared-memory intra-node bandwidth is higher than the bandwidth achieved by the interconnect. Depending on the power of the memory subsystem, many processor pairs on the same shared-memory node can communicate more or less simultaneously. In contrast, communication between nodes is limited by the (small, fixed) number of network cards (ports) per node, and processors on a node have to share the bandwidth provided by the network. Often only one processor on a node can be involved in inter-node communication at a time. Although less common, SMP-like systems can have more than two hierarchy-levels. Especially lower-end SMP clusters are often heterogeneous both in the sense of having different types of processors on the different nodes, and in the sense of having different numbers of processors per node.

For the purpose of this paper we make the following simple assumptions for general, multilevel SMP clusters:

1. Processors are grouped hierarchically into a tree of processing nodes. A single node on level 0 represents the complete SMP cluster. Single processors form singleton nodes at the bottom (deepest level) of the hierarchy.

2. Processing nodes on level i are assumed to be fully connected and communication between nodes is uniform (same communication costs for any pair of level i nodes).

3. Communication between nodes on level $i-1$ is more expensive (higher latency, lower bandwidth, port restrictions) than communication between nodes on level i

4. Communication between pairs of processors take place via the cheapest communication medium connecting them, that is on the deepest level i such that both processors belong to the same level i processing node.

Thus a multilevel SMP system can be thought of as a tree of processing nodes. The tree need not be balanced, i.e. some processors can sit deeper in the hierarchy than others. In this model a “standard” SMP cluster has three levels. The intermediate level consists of the shared memory nodes, and Assumption 2 states that the communication between shared memory nodes is uniform. Systems where this assumption does not hold because of the interconnect, can sometimes be modeled as systems with more than two levels. For instance, fat-tree networks have a hierarchical structure that fit into the model. Note, however, that networks like meshes do not have a hierarchical structure in this sense. More formal models needed for the detailed design and analysis of communication algorithms for multilevel SMP systems can be found in e.g. [1, 3].

Assumption 4 must be guaranteed by the programming interface. For instance, an MPI implementation for an SMP cluster should use the shared memory for communication between processors on the same shared-memory node. For point-to-point communications, many (most?) MPI implementations ensure this. The MPICH implementation [7] for instance has a (lock-free) shared memory device for intra-node communication, although many other (MPICH-derived) implementations are better suited to Linux-clusters.

The MPI collectives, which are often implemented on top of point-to-point communication immediately benefit from “SMP-aware” point-to-point communications. However, to deal with restrictions on inter-node communication like the fact that only a small number of processors per node can do inter-node communication at the same time, different algorithms than algorithms designed under the assumption of a flat system are needed to support efficient collective communication. Also the possibility that SMP clusters can be heterogeneous need to be taken into account. At the least algorithms for collective operations must be able to deal with the fact that different processing nodes can have different numbers of processors; this is so either by design, or because the MPI communicator spans only part of the system.

Hierarchical algorithms for collective operations like barrier synchronization and broadcast are easy [10, 11, 12, 13], and are incorporated in many MPI implementations. For example, broadcast from a root processor on level 0 can be done by the root broadcasting to chosen root processors on the level 1 nodes, all of which do a broadcast recursively [3].

This recursive decomposition is more difficult or not possible at all for other collectives. An explicitly hierarchical algorithm for the `MPI_Alltoall` collective is discussed and implemented in [16, 20]. Hierarchical algorithms for `MPI_Allgather` and `MPI_Allgatherv` are currently being implemented by the author [21].

3 The MPI topology functionality

The MPI topology mechanism provides a portable means of adapting an application to the underlying communication system. The mechanism allows the user to specify a communication pattern (*virtual topology*) as a graph over processes in which edges represent potential communication between pairs of processes. By a call to a topology creation function, the MPI implementation is allowed to perform a process remapping which brings processes that will presumably communicate closer to each other. An MPI implementation for SMP clusters could attempt to map processes that are direct neighbors in the virtual topology to the same shared-memory node. MPI allows specification of virtual topologies explicitly as communication graphs, or implicitly as meshes/tori/hypercubes (*Cartesian topologies*) where communication is assumed to be along the dimensions of the mesh.

Implementations of the MPI topology mechanism for multilevel SMP systems are described in [8, 19]. Both are based on graph-partitioning, and worthwhile, sometimes considerable improvements in communication performance for synthetic benchmarks are reported. We note here that exact graph partitioning is an NP-hard problem [5], and finding an exact partition even for medium sized graphs is prohibitively expensive. We also note that for a remapping to have any effect on application performance, the implementation of point-to-point communication must be SMP-aware (Assumption 4).

The MPI topology functionality is a weak mechanism in that it allows only a very rough specification of communication patterns. We now discuss some of its shortcomings and possible remedies. The remarks are mostly directed at the functionality for creating graph topologies, but (except for the scalability issue) are also relevant for Cartesian topologies. A graph topology is created by the collective MPI call

```
MPI_Graph_create(basecomm,
                 nnodes,index,edges,
                 reorder,
                 &graphcomm);
```

which returns a new communicator `graphcomm` spanning `nnodes` processes. The processes in this communicator may have been reordered relative to their order (that is, mapping to processors) in `basecomm` to better support the given communication pattern. This in turn is described as an undirected (symmetric) graph and given by the arguments `nnodes`, `index` and `edges` (see [18, Chapter 6]). The boolean `reorder` argument determines whether the MPI implementation should attempt a process remapping.

Vagueness: The topology creation calls `MPI_Graph_create` and `MPI_Cart_create` are collective, and as for other collective MPI calls the arguments provided by different processes must "match". For the graph creation call all processes must supply *the* full communication graph (although not said so explicitly in the standard), presumably with identical values for the `index` and `edges` arrays. The MPI standard is (deliberately?) vague about matching arguments, and no explicit (coercion) rules are given. Likewise, we assume that all processes in both graph and Cartesian topology creation calls must give the same value for the `reorder` argument.

Lack of precision:

- What is the communication volume between neighboring processes? Are some edges more heavily loaded than other?
- What is the frequency of communication? Are some edges used more frequently than others?
- When do communications happen? Are communications "simultaneous", or separate in time?
- for Cartesian topologies: is communication along dimensions, or along diagonals or both?

Such information, which, if at all, is known only by the application program, can clearly influence what the best possible remapping is for the given communication pattern.
Possible solution: allow weighted graphs, or multigraphs. Multiple edges or weighted edges between processes can be used to indicate heavier load (volume and/or frequency). The ordering of the edges could be used to indicate the timing relations, although it seems complicated to give a consistent and useful definition of such a functionality. The creation call for Cartesian topologies could be extended with a `diagonal` flag, indicating whether communication is along dimensions only, or also along diagonals in the grid.

Lack of accuracy:

- What is the optimization criterion that should be applied for the process reordering? Is minimizing the total amount of communication between processing nodes important? Or should rather the maximum number of communication edges out of any one processing node be minimized?
- Are there special requirements, e.g. that certain processes not be remapped, because they are bound to processors with special features, e.g. to a node with especially large memory, or with special I/O capabilities?

Possible solutions: An `MPI_Info` object could be used in the topology creation call to give hints to the MPI implementation. A directive argument could be used to assert that the calling process not be remapped. Alternatively, the `reorder` argument could be used locally by each process to indicate whether the calling process may be remapped or not.

Difficulty of use/lack of scalability:

- Each calling process must give the complete communication graph. This is error-prone, and can be tedious for the application programmer. For applications with irregular communication patters each process probably knows its immediate communication neighborhood, but will (most?) often not know the whole communication graph. In such cases extra communication is needed in the application program to build the communication graph. The requirement that the same (isomorphic or identical) graph is given by each process also takes care to ensure. The construct is also non-scalable (graphs may grow as the square of the number of processes) but this is probably the less significant drawback.

- How much time should be be invested in finding a good remapping (see NP-completeness remark above)?

- What is the performance benefit?

Possible solutions: Put the burden of building the full communication graph on the MPI implementation: each process should only supply its list of neighbors. This cannot lead to errors since rules for how MPI should construct a consistent communication graph can easily be specified. This solution is furthermore scalable. An `MPI_Info` object could again be used to instruct the MPI implementation on how much time should be invested in the remapping. Alternatively, runtime option or environment variable could be used. The performance benefit issue can partially be handled by being able to query communicators: do processes i and j reside on the same shared-memory node? A performance benefit can be expected if the new communicator maps many neighboring processes (in the virtual topology) to the same shared memory node. A concrete proposal in this direction is given in Section 4.

3.1 Alternative topology interfaces

In this section we describe alternative interfaces for graph and Cartesian topology creation functions that address the issues raised above. The interfaces can trivially be implemented on top of the MPI topology functionality, but a serious implementation, producing a better remapping than an already existing MPI implementation of the topology functionality (which may be possible because more information is supplied) takes a serious effort, and will require environmental support, possibly in the form outlined in Section 4.

The following graph creation function allows a higher degree of precision, allows more accuracy, is more scalable and might be more convenient to use for some applications:

```
int Graph_create(MPI_Comm basecomm,
                 int degree,
                 int neighbors[],
                 int reorder,
                 MPI_Info mapinfo,
                 MPI_Comm *graphcomm)
```

Instead of all processes supplying the full graph, each process gives only its own list of processes with which it wants to communicate. The number of neighbors is given as `degree`, and we allow the same neighbor to appear multiple times in `neighbors`. Neighbor multiplicity is used indicate a higher communication load between the process and that neighbor. A process that should not appear in the resulting `graphcomm` gives -1 as `degree` argument (0 indicates a process with no neighbors that will appear in the resulting `graphcomm` communicator). There is no symmetry requirement for the neighbor lists: a consistent, symmetric (multi-)graph is built by the implementation. The `reorder` argument has local semantics; a `reorder` value of 0 means that the calling process should remain fixed (that is, bound to the same processor as in `basecomm`). An `MPI_Info` argument can be used to provide further hints to the MPI implementation, for example on the optimization criteria to be used.

A trivial implementation of this interface that already might be useful in cases where the user does not want to bother with building the whole communication graph is as follows: The neighborlists are gathered by all processes (by an `MPI_Allgather` and an `MPI_Allgatherv` call). Each process constructs the full communication graph, and after a minimum all-reduction over the `reorder` arguments, the existing `MPI_Graph_create` function creates the required `graphcomm` communicator. More ambitious implementations will of course want to do more.

The complete communication graph implicitly constructed by the `Graph_create` call can be queried by the already present topology query functions of MPI.

Similarly for Cartesian topologies:

```
int Cart_create(MPI_Comm basecomm,
                int ndims, int dims[],
                int periods[],
                int diagonal,
                int multiplicity[],
                int reorder,
                    MPI_Info mapinfo,
                    MPI_Comm *cartcomm)
```

The new `diagonal` argument indicates to the underlying implementation whether communication may be also along diagonals or is solely along the dimensions. The `multiplicity` argument is used to specify the load of the communication along each dimension for the calling process; if `diagonal` is set to 1, multiplicities must also be given to the diagonals (an ordering of the diagonals must of course be defined). A `NULL` value can be given, indicating that all communication edges have the same load. All processes in `basecomm` must give the same values for `ndims`, `dims` and `periods`, while the remaining input parameters are local, and may differ.

To decide whether such changes to the MPI topology mechanism are worthwhile to pursue further, collaboration with users is indispensable. As mentioned, a non-trivial implementation of a topology interface along the lines discussed above is possible on top of MPI if information about the SMP system can be made available. Needed is essentially the distribution of the processes in `basecomm` over the processing nodes. In Section 4 it is shown how this information can be provided.

3.2 Additional functionality

Whenever processes are remapped as a result of a call to a topology creation function (or other communicator creation function), data redistribution from old to new communicator may become necessary, especially if new communicators are created repeatedly during the execution of the application program. For instance, an `MPI_Graph_create(basecomm,...,&graphcomm)` call returns a new communicator which is a *sub-communicator* of `basecomm` in the sense that all processes in the new communicator are also in the old communicator. In the MPI setting, data redistribution means that the process with rank 0 in `basecomm` sends its data to the process which has rank 0 in `graphcomm`, the process with rank 1 in the `basecomm` sends its data to the process with rank 1 in `graphcomm`, and so on.

In general, processes in a super-communicator have to send data to processes with the same rank (if they are there - the sub-communicator may have fewer processes than the super-communicator) in a sub-communicator. Thus, it would be convenient to be able to compare two communicators with the purpose of determining whether one is a sub-communicator of the other. MPI already has a comparison function for communicators [18, Chapter 5]. A call

```
MPI_Comm_compare(comm1,comm2,&result);
```

with communicators `comm1` and `comm2` returns in `result` either `MPI_IDENT`, `MPI_CONGRUENT`, `MPI_SIMILAR`, or `MPI_UNEUQAL`. It is easy to extend this with further result values; `MPI_SUBCOMM` if `comm1` is a sub-communicator of `comm2`, `MPI_SUBCOMM_STRICT` if `comm1` is a sub-communicator of `comm2` and furthermore the ordering of the processes in `comm1` is as in `comm2`, `MPI_SUPERCOMM` if `comm1` is a super-communicator of `comm2`, and finally `MPI_SUPERCOMM_STRICT` if `comm1` is a strict super-communicator of `comm2`. A comparison function with this extended functionality can easily be implemented using the group manipulation functions `MPI_Group_intersection` and `MPI_Group_compare`. A full implementation of such a function, called `MPI_Comm_relate`, is given in Appendix A.

For the data redistribution let `basecomm` and `subcomm` be the given communicators. Since not all processes of `basecomm` may be in `subcomm`, redistribution has to take place in `basecomm`. The process with rank i in `basecomm` has to send data to the process with rank i in `subcomm`, provided that i is smaller than the size of `subcomm`. Let this process have rank i' in `basecomm`. We call i' the *to-rank* of process i (in `basecomm`). The *from-rank* of process i' (which has rank i in `subcomm`) is i (in `basecomm`), and process i' has to receive data from process i (both in `basecomm`). Processes in `basecomm` that are not also in `subcomm` have no from-rank, and processes in `basecomm` with rank greater than the number of processes in `subcomm` have no to-rank.

A convenient utility function for data redistribution from super- to sub-communicator computes the `torank` and `fromrank` of the calling process. This is the task of a new collective function:

```
int MPI_Comm_map(MPI_Comm basecomm,
                 MPI_Comm subcomm,
                 int *torank,
                 int *fromrank)
```

If the calling process is not in `subcomm` it does not have a handle to this communicator, and gives `MPI_COMM_NULL` as parameter. Non-existing to- or from-ranks are returned as `MPI_PROC_NULL`. An implementation of `MPI_Comm_map` is given in Appendix B, which also explains why the function cannot be implemented with local semantics.

A new collective operation

```
int MPI_Permute(void *inbuf, int incount,
                MPI_Datatype intype,
                int fromrank,
                void *outbuf,int outcount,
                MPI_Datatype intype,
                int torank,
                MPI_Comm comm)
```

can now be used to perform the redistribution. As in MPI other collectives type signatures between sending and receiving processes must match. The implementation is trivial:

```
MPI_Sendrecv(output,outcount,outtype,
             torank,MPI_PERMUTE_TAG,
             inbuf,incount,intype,
             fromrank,MPI_PERMUTE_TAG,
             comm,&status);
```

However, for special networks better, specialized algorithms for permutation routing exist, and could be exploited in the implementation. Therefore, and for sheer convenience, it might be useful to have the redistribution collective be part of the communications library.

4 Explicitly SMP-aware MPI programming

The topology mechanism makes it possible for the MPI implementation to adapt a virtual communication topology to fit better with the SMP system. Experiments indicate that it can be implemented to give significant performance benefits for applications with communication patterns with some degree of locality [19]. The mechanism, however, gives no solution to the orthogonal situation in which the user wants to *explicitly* take the SMP-structure into account *when* setting up his communication pattern. In this situation, it would be desirable to be able to query the system as to how many nodes are available, which processes in a given communicator are on a given shared-memory node etc. For this situation some amount of SMP-topology information could be incorporated into the communications library. An easy and useful way of doing this is to provide a hierarchy of communicators reflecting the hierarchical communication structure of the SMP cluster. At the bottom of this hierarchy, each process belong to a `MPI_COMM_SELF` communicator. An attribute `MPI_DEPTH` of this communicator could give the depth of the process in the hierarchy. At the top of the hierarchy `MPI_COMM_WORLD` represents the whole SMP cluster. `MPI_COMM_WORLD` might have an attribute `MPI_MAX_DEPTH` giving the maximum depth of a processor in the system (as determined by `MPI_COMM_WORLD`). For each intermediate processing node, there is a communicator `MPI_COMM_NODE[i]`, where `i` is a level between 0 and $\texttt{MPI\_MAX\_DEPTH} - 1$. Any given process thus belongs to a hierarchy of communicators, like this:

$$\begin{array}{c} \texttt{MPI\_COMM\_WORLD} = \texttt{MPI\_COMM\_NODE[0]} \\ \supseteq \\ \ldots \\ \supseteq \\ \texttt{MPI\_COMM\_NODE[i]} \\ \supseteq \\ \ldots \\ \supseteq \\ \texttt{MPI\_COMM\_NODE[MPI\_DEPTH]} = \texttt{MPI\_COMM\_SELF} \end{array}$$

For SMP like systems, the processes on the same shared-memory node as any given process are contained in that process' `MPI_COMM_NODE[MPI_DEPTH-1]` communicator (which always exist; for non-hierarchical systems simply as `MPI_COMM_WORLD`). This communicator identifies the processes on the node, as well as the number of processes on the node. Thus, this functionality is sufficient for providing the information needed for a non-trivial implementation of the extended topology remapping functions `Graph_create` and `Cart_create` on top of MPI.

A useful hierarchy inquiry function returns for any two processes i and j in a communicator `comm` the level on which i and j can communicate most cheaply. The interface could take the form

```
int MPI_Comm_level(MPI_Comm comm,
                   int i, int j,
                   int *level)
```

and would compute the deepest `level` such that processes `i` and `j` both belong to `MPI_COMM_NODE[level]`.

This inquiry function could be used together with the topology functionality to get an idea of the quality of the remapping produced. Let `graphcomm` be the communicator returned by a call to `MPI_Graph_create`. If processes i and j have the same depth, and the call to `MPI_Comm_level` returns a `level` equal to $\texttt{MPI\_DEPTH} - 1$, then the two processes are on the same shared memory node. If a smaller level is returned, i and j are on more remote nodes, and communication between these processes is more expensive. A `graphcomm` where many neighboring processes are mapped to the same shared-memory node, will probably be better than communicators where this is not the case.

A communicator hierarchy as sketched here can be made available using existing MPI operations with some environmental support/knowledge provided by the user. The MPI function `MPI_Get_processor_name` returns a processor name/identification for the calling process. Assuming that this identifies the shared-memory node to which the calling process belongs, the communicators in the hierarchy can be created by a sequence of

```
MPI_Comm_split(MPI_COMM_NODE[i-1],
               shared_node_id[i],rank,
               &MPI_COMM_NODE[i]);
```

calls, where `shared_node_id[i]` is an integer id of the level i processing node of the calling process, extracted from the processor name. Note that the use of processor names as returned by `MPI_Get_processor_name` makes this solution non-portable, since MPI does not prescribe a naming convention which allows to extract the desired id's in a portable fashion.

A different proposal of how to incorporate SMP awareness explicitly into MPI communicators is described in [14]. This proposal is orthogonal to the one presented here, and describes the hierarchy implicitly by having cluster attributes associated with each communicator which identifies the assignment of processes to SMP nodes. It is not immediately clear that this approach suffices for multi-level hierarchical systems.

Explicit incorporation of communication hierarchy into a communication library like MPI to some extent compromises the architecture independence of the standard. Consideration must be applied.

5 Conclusion

We discussed requirements to high-quality MPI implementations for the efficient utilization of SMP clusters, and suggested further functionality for more convenient and efficient programming of SMP clusters. The extensions could presumably be made to fit with the existing MPI standard, but can also be implemented as a separate library on top of MPI. The suggestions of this paper are meant to generate discussion, and should not be taken as actual proposals for extending the MPI standard. Whether they are worth pursuing further must be decided in close cooperation with actual SMP cluster users with serious message passing applications.

Acknowledgments

The author thanks Bill Gropp, Rolf Hempel, Rolf Rabenseifner and Joachim Worringen for clarifying discussions and constructive suggestions.

References

[1] B. Alpern, L. Carter, and J. Ferrante. Modelling parallel computers as memory hierarchies. In *IEEE Conference on Massively Parallel Programming Models*, pages 116–123, 1993.

[2] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks. In *Supercomputing*, 2000. See `http://www.sc2000.org/proceedings/techpapr/index.htm#04`.

[3] F. Cappello, P. Fraigniaud, B. Mans, and A. L. Rosenberg. HiHCoHP: Toward a realistic communication model for hierarchical HyperClusters of heterogeneous processors. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS01)*, pages 42–47, 2001.

[4] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2001.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979. With an addendum, 1991.

[6] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI – The Complete Reference*, volume 2, The MPI Extensions. MIT Press, 1998.

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

[8] T. Hatazaki. Rank reordering strategy for MPI topology creation functions. In *5th European PVM/MPI User's Group Meeting*, volume 1497 of *Lecture Notes in Computer Science*, pages 188–195, 1998.

[9] D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Supercomputing*, 2000. See `http://www.sc2000.org/proceedings/techpapr/index.htm#04`.

[10] N. T. Karonis, B. R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS 2000)*, pages 377–386, 2000.

[11] T. Kielmann, H. E. Bal, and S. Gorlatch. Bandwidth-efficient collective communication for clustered wide area systems. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS 2000)*, pages 492–499, 2000.

[12] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. In *Proceedings of the 1999 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, volume 34(8) of *ACM Sigplan Notices*, pages 131–140, 1999.

[13] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MPI's reduction operations in clustered wide area systems. In *Third Message Passing Interface Developer's and User's Conference (MPIDC'99)*, pages 43–52, 1999.

[14] Message Passing Forum. *MPI-2 Journal of Development*, 1997. `http://www.mpi-forum.org/docs/mpi-20-jod.ps`.

[15] R. Rabenseifner. Communication and optimization aspects on hybrid architectures. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 9th European PVM/MPI Users' Group Meeting*, volume 2474 of *Lecture Notes in Computer Science*, pages 410–420, 2002.

[16] P. Sanders and J. L. Träff. The hierarchical factor algorithm for all-to-all communication. In *Euro-Par 2002 Parallel Processing*, volume 2400 of *Lecture Notes in Computer Science*, pages 799–803, 2002.

[17] H. Shan, J. P. Singh, L. Oliker, and R. Biswas. A comparison of three programming models for adaptive applications on the Origin2000. *Journal of Parallel and Distributed Computing*, 62(2):241–266, 2002.

[18] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI – The Complete Reference*, volume 1, The MPI Core. MIT Press, second edition, 1998.

[19] J. L. Träff. Implementing the MPI process topology mechanism. In *Supercomputing*, 2002. `http://www.sc2002.org/paperpdfs/pap.pap122.pdf`.

[20] J. L. Träff. Improved MPI all-to-all communication on a Giganet SMP cluster. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 9th European PVM/MPI Users' Group Meeting*, volume 2474 of *Lecture Notes in Computer Science*, pages 392–400, 2002.

[21] J. L. Träff. Efficient all-gather communication on parallel systems with hierarchical communication structure. In preparation, 2003.

A Extended communicator comparison function

An extended communicator comparison function `MPI_Comm_relate` is given in Figure 1. In addition to the existing result values provided by the MPI standard [18, Chapter 5] four new values are introduced for identifying (strict) sub/super-communicator relationships. A communicator `comm1` is a sub-communicator of `comm2` if all processes in `comm1` are also in `comm2`; the relationship is *strict* if furthermore the ordering of the processes in `comm1` is the same in `comm2`. The implementation relies on process group intersections: a set A is a subset of B iff $A = A \cap B$. Because MPI process groups are ordered sets, and because of the semantics of `MPI_Group_intersection`, two calls to `MPI_Group_intersection` are necessary to determine whether the sub/super-communicator relationship is strict. As for `MPI_Comm_compare` the function cannot be called with `MPI_COMM_NULL` as either of the communicator arguments.

B An implementation of the computation of communicator mapping information

In this appendix we give an implementation of a collective operation for computing a mapping between ranks in a base communicator and a sub-communicator. The function computes `torank` and `fromrank` as required by the redistribution operation described in the main text. The code is shown in Figure 2.

Recall that `torank` is only defined for processes in `basecomm` whose rank is smaller than the size of `subcomm`. Not all processes, however, have a handle to the latter communicator, so the size of `subcomm` has to be distributed among all `basecomm` processes. We use the non-rooted (symmetric) `MPI_Allreduce` collective for this. This partly explains why `MPI_Comm_map` cannot be implemented with local semantics. Processes that are both in `subcomm` and `basecomm` can compute both their `torank` and `fromrank` locally, using group translation functions to translate among processes in the two communicators. For instance, the `torank` of process `rank` (in `basecomm`) is the rank of the process in `basecomm` that has rank `rank` in `subcomm`. Processes that are not in `subcomm`, and whose rank (in `basecomm`) is smaller than the size of `subcomm` determine their `torank` by receiving with `MPI_ANY_SOURCE` from the corresponding process. The `torank` of these processes cannot otherwise be determined without communication, thus the map-computation needs collective participation by all processes in `basecomm`. In order to make `MPI_Comm_map` a safe library function, a `Comm_dup` is performed on the `basecomm` used for communication.

```
#define MPI_SUBCOMM          MPI_UNEQUAL+1
#define MPI_SUBCOMM_STRICT   MPI_UNEQUAL+2
#define MPI_SUPERCOMM        MPI_UNEQUAL+3
#define MPI_SUPERCOMM_STRICT MPI_UNEQUAL+4

int MPI_Comm_relate(MPI_Comm comm1, MPI_Comm comm2, int *result)
{
  MPI_Group group1, group2, inter;
  int groupresult;

  MPI_Comm_compare(comm1,comm2,result);
  if ((*result)==MPI_UNEQUAL) {
    MPI_Comm_group(comm1,&group1);
    MPI_Comm_group(comm2,&group2);

    MPI_Group_intersection(group1,group2,&inter);
    MPI_Group_compare(inter,group2,&groupresult);
    if (groupresult==MPI_SIMILAR)    *result = MPI_SUPERCOMM;
    else if (groupresult==MPI_IDENT) *result = MPI_SUPERCOMM_STRICT;
    else {
      MPI_Group_free(&inter);
      MPI_Group_intersection(group2,group1,&inter);
      MPI_Group_compare(inter,group1,&groupresult);
      if (groupresult==MPI_SIMILAR)    *result = MPI_SUBCOMM;
      else if (groupresult==MPI_IDENT) *result = MPI_SUBCOMM_STRICT;
      else                             *result = MPI_UNEQUAL;
    }
    MPI_Group_free(&inter);
    MPI_Group_free(&group1);
    MPI_Group_free(&group2);
  }

  return MPI_SUCCESS;
}
```

Figure 1. Implementation of the `MPI_Comm_relate` operation.

```
#define COMM_MAP_TAG 999

int MPI_Comm_map(MPI_Comm basecomm, MPI_Comm subcomm, int *torank, int *fromrank)
{
  MPI_Comm mapcomm; /* for map-internal communication */
  MPI_Group basegroup, subgroup;
  int rank, subrank, subfrom, subsize;
  int *subinbase;
  MPI_Status status;

  MPI_Comm_rank(basecomm,&rank);

  if (subcomm!=MPI_COMM_NULL) {
    MPI_Comm_group(basecomm,&basegroup);

    MPI_Comm_group(subcomm,&subgroup);
    MPI_Group_rank(subgroup,&subrank);
    MPI_Group_size(subgroup,&subsize);

    *fromrank = subrank;
  } else {
    subsize = 0;
    *fromrank = MPI_PROC_NULL;
  }

  MPI_Allreduce(MPI_IN_PLACE,&subsize,1,MPI_INT,MPI_MAX,basecomm);

  MPI_Comm_dup(basecomm,&mapcomm);
  if (subcomm==MPI_COMM_NULL) {
    if (rank<subsize) {
      MPI_Recv(torank,1,MPI_INT,MPI_ANY_SOURCE,COMM_MAP_TAG,mapcomm,&status);
      if (*torank!=status.MPI_SOURCE)
        fprintf(stderr,"Rank %d error in source, expected %d actual %d\n",
                rank,*torank,status.MPI_SOURCE);
    } else *torank = MPI_PROC_NULL;
  } else {
    if (rank<subsize) {
      MPI_Group_translate_ranks(subgroup,1,&rank,basegroup,torank);
    } else *torank = MPI_PROC_NULL;

    MPI_Group_translate_ranks(basegroup,1,&subrank,subgroup,&subfrom);
    if (subfrom==MPI_UNDEFINED) {
      MPI_Send(&rank,1,MPI_INT,subrank,COMM_MAP_TAG,mapcomm);
    }
    MPI_Group_free(&subgroup);
    MPI_Group_free(&basegroup);
  }
  MPI_Comm_free(&mapcomm);

  return MPI_SUCCESS;
}
```

Figure 2. Implementation of the `MPI_Comm_map` **collective.**