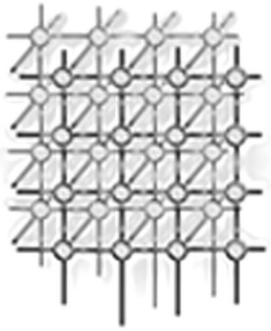
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; :1–21 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*



The Scalable Process Topology Interface of MPI 2.2

Torsten Hoefler[1], Rolf Rabenseifner[2], Hubert Ritzdorf[3], Bronis R. de Supinski[4], Rajeev Thakur[5], and Jesper Larsson Träff[6]

[1] *University of Illinois at Urbana-Champaign, 1205 W. Clark St. Urbana, IL 61801, USA*[†]
[2] *HLRS, University of Stuttgart, D-70550 Stuttgart, Germany*[‡]
[3] *HPCE, NEC Deutschland GmbH, Hansaallee 101, 40549 Düsseldorf, Germany*[§]
[4] *Lawrence Livermore National Laboratory, Box 808, L-557, Livermore, CA 94551-0808*[¶]
[5] *Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439*[‖]
[6] *Department of Scientific Computing, University of Vienna, Nordbergstrasse 15/3C, A-1090 Vienna, Austria*[**]

SUMMARY

The *Message-Passing Interface* (MPI) standard provides basic means for adaptations of the mapping of MPI process ranks to processing elements to better match the communication characteristics of applications to the capabilities of the underlying systems. The MPI *process topology mechanism* enables the MPI implementation to rerank processes by creating a new communicator that reflects user-supplied information about the application communication pattern. With the newly released MPI 2.2 version of the MPI standard, the process topology mechanism has been enhanced with new interfaces

*Correspondence to: Department of Scientific Computing, University of Vienna, Nordbergstrasse 15/3C, A-1090 Vienna, Austria
[†]htor@cs.indiana.edu. The work of this author was done mostly at Indiana University, IN, USA.
[‡]rabenseifner@hlrs.de
[§]hritzdorf@hpce.nec.com. The work of this author was done mostly at NEC Laboratories Europe, St. Augustin, Germany.
[¶]bronis@llnl.gov
[‖]thakur@mcs.anl.gov
[**]traff@par.univie.ac.at. The work of this author was done mostly at NEC Laboratories Europe, St. Augustin, Germany.

for *scalable* and *informative* user-specification of communication patterns. Applications with relatively static communication patterns are encouraged to take advantage of the mechanism whenever convenient by specifying their communication pattern to the MPI library. Reference implementations of the new mechanism can be expected to be readily available (and come at essentially no cost), but nontrivial implementations pose challenging problems for the MPI implementer.

This paper is first and foremost addressed to application programmers wanting to use the new process topology interfaces. It explains the use and the motivation for the enhanced interfaces and the advantages gained even with a straightforward implementation. For the MPI implementer, the paper summarizes main issues in the efficient implementation of the interface and explains the optimization problems that need to be (approximately) solved by a good MPI library.

KEY WORDS: Message Passing Interface, MPI, MPI 2.2, MPI Forum, process topologies, process mapping, communication patterns, reordering

1. Motivation

The Message Passing Interface (MPI) Forum recently released the MPI 2.1 (September 2008) and MPI 2.2 (September 2009) standards. These standards address and fix many smaller issues and inaccuracies in the MPI 1.0 and MPI 2.0 specifications and introduce a few significant enhancements. The most significant addition in MPI 2.2 [1] is a highly scalable and informative *graph topology* interface as an alternative to the non-scalable graph topology interface of MPI 1.0 [2].

The previously existing MPI process topology mechanism was controversial from the outset. It is not strictly a message-passing feature, but provides a portable way for the MPI library to adapt application communication patterns to the underlying hardware without exposing its details. It was based on and designed to resemble the PARMACS interface [3] and to support PARMACS applications that already used this facility. The basic abstraction of the interface is to represent application communication patterns by directed, unweighted graphs, either explicitly or implicitly (as in the Cartesian topologies). However, the original design in MPI 1.0 is not scalable (in fact, it is one of the most non-scalable constructs in MPI [4]) and is insufficient and problematic in many other ways [5, 6]. In short, the lack of scalability, precision, and information, the unreasonable burden on the programmer to collect the full application communication graph at each calling process, and other issues render the interface useless at scale, exactly where it should provide the most benefit.

Using static information about application communication patterns to adapt to the underlying communication system remains attractive, also for interfaces other than MPI [7]. Large networks use components with a fixed degree and thus exhibit sparse connectivity. Prominent examples are k-ary n-cubes with n-dimensional torus topologies (n-ary 2-cube) as special cases [8], folded Clos (fat-tree) or Butterfly networks, and even directed networks based on Kautz graphs [9]. For example, the Cray XT-4/5 series and the IBM Blue Gene L/P systems use three-dimensional tori as communication networks, the SiCortex systems were based on Kautz graphs, while most InfiniBand or Myrinet installations use folded Clos

topologies. Unfortunately, tori have low bisection bandwidth, and InfiniBand is restricted to static oblivious routing schemes [10]. The prominence of hierarchical communication structures in SMP-clusters, which will be exacerbated with upcoming multi- and many-core based systems, further adds to network complexity. Performance improvements achievable by matching application communication patterns to the hierarchical structure of multi-core nodes have recently been demonstrated [11,12], illustrating again the need for graph topology features with weighted communication edges.

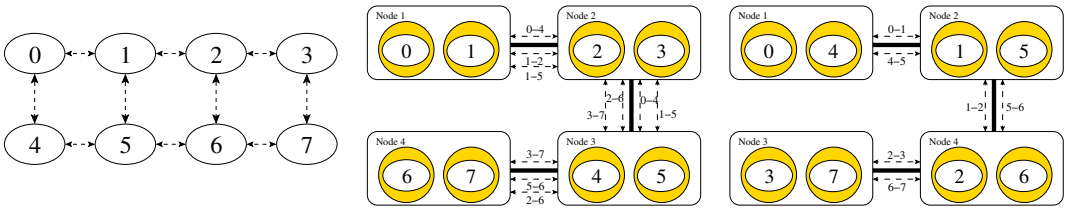
The MPI 1.0 standard [2, Chapter 6] provides two ways to specify application communication patterns: implicit, Cartesian constructors and explicit, general (graph) constructors. The topology functions return proper MPI communicators that support all MPI communication models (point-to-point, collective, and one-sided), and thus permits communication between any of the included processes. The topology specification merely provides hints on expected communication that the library can use to improve performance.

Cartesian constructors specify d-dimensional mesh or torus topologies. This interface is scalable since it only requires each process to provide the sizes of all dimensions of the mesh or torus. However, it only captures simple mesh or torus communication patterns, thus limiting the optimization opportunities available to the MPI library. For example, users cannot indicate that the application will communicate along diagonals or with larger neighborhoods than the immediate $2d$ neighbors. Furthermore, the interface provides no mechanism to specify varying communication or message volumes on the different dimensions.

General graph constructors can specify arbitrary communication topologies as unweighted, directed graphs, with nodes representing processes and edges representing communication relationships among processes. Each process must call the constructor with the full communication graph, which results in a non-scalable mechanism. The memory consumption per process is linear in the number of processes even for sparse communication patterns, and quadratic in the worst case. Perhaps worse, the user must build the communication graph on all processes, which frequently requires additional, global, irregular communication prior to calling the constructor. Finally, the interface only supports unweighted graphs that specify binary communication relationships in which each edge indicates substantial expected communication between the two processes.

An MPI library may return a communicator with the same process-to-processor mapping as in the input communicator, which enables a trivial implementation of these constructors. For better performance, the MPI library can generate a new communicator that maps processes in a way that the expected communication as expressed with the communication graph is better matched to the capabilities of the underlying hardware. Although MPI libraries unfortunately rarely provide optimized implementations of the topology mechanism, such can provide significant benefits [13–15].

Figure 1 illustrates how the process topology interface supports optimization of the mapping of a simple application with nearest-neighbor communication. In this case, the target system consists of four dual-CPU machines that are connected in a chain from node 1 to node 4. MPI supports communication among all processes so the underlying network needs to forward messages through intermediate nodes, even with the simple two-dimensional 2×4 mesh topology that Figure 1(a) shows. Although each process has at most three neighbors, the default linear mapping of MPI ranks to processors $\pi_1 = (0, 1, 2, 3, 4, 5, 6, 7)$ results in



(a) Communication Pattern (b) Mapping A (c) Mapping B

Figure 1. An example for different mappings of a two-dimensional mesh on a chain topology.

substantial unnecessary forwarding as shown in Figure 1(b). For example, four connections use the link between nodes 2 and 3, whereas the improved mapping $\pi_2 = (0, 4, 1, 5, 2, 6, 3, 7)$ has a maximum link congestion of two (Figure 1(c)).

The rest of the paper is organized as follows: Section 2 describes the new MPI 2.2 distributed graph topology interface that alleviates the most glaring problems with the MPI 1.0 constructors and supports richer communication pattern optimizations. It also details a path for migration of legacy code written using the MPI 1.0 graph topology functionality to the MPI 2.2 distributed graph topology interface. Section 3 presents several use cases with real applications that demonstrate that the new interfaces can substantially reduce memory usage and provide better optimization oportunities. In Section 4 optimization problems that have to be solved to provide better process reorderings based on the more informative user specifications are outlined, and it is discussed how an implementation can exploit the additional hints provided through the new interface. Finally, Section 5 discusses some of the deliberations by the MPI Forum[†] that lead to the new interface and the continuing efforts to improve the process topology mechanisms. The paper should be read as an invitation for applications to use the MPI 2.2 interface (possibly by migrating from the old MPI 1.0 interface) and an incentive to collect experience for possible extensions to the process topology mechanism for future MPI versions.

2. The Distributed Graph Topology Interface Specification

The new MPI 2.2 graph topology interface addresses the three primary problems with the topology interface of MPI 1.0 and provides a *scalable*, more *informative*, and more *user-friendly* interface. It does not change the basic abstraction of representing communication patterns as directed graphs. However, the new interface allows distributed specification the communication graph which provides scalability and relieves the user of the burden of constructing the full

[†]See `http://www.mpi-forum.org`

graph at all processes. Next, the user can provide relative weights for the communication edges to capture their expected utilization. Finally, the interface also provides a handle to influence the interpretation of the weights as well as to convey other information to guide the adaptation of the specified communication pattern.

The MPI 2.2 standard has two variants of the *distributed graph constructor interface*:

1. MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights, outdegree, destinations, destweights, info, reorder, comm_dist_graph) is the *adjacent specification*. Each process specifies its (incoming and outgoing) neighbors with which it will communicate (its sources and destinations) and the associated weights, reflecting in some sense the amount or frequency of communication with these neighbors. For the call to be correct each source of process i must list i as a destination, and conversely each destination of process i must list i as a source. The call returns a new communicator in which processes may have been remapped to other processors so as to match the specified communication pattern better to the hardware communication system. If the correctness condition on sources and destinations is not respected, the behavior and result of the call to this function is undefined.
2. MPI_Dist_graph_create(comm_old, n, sources, degrees, destinations, weights, info, reorder, comm_dist_graph) is the *general specification*. Each process can arbitrarily specify edges between communicating processes of the application; that is, a process need not even be a source or destination of the edges it specifies. Each process specifies edges by a source node i in the sources array (of size n), the number of outgoing edges of i in the same position in the degrees array, and lists the targets of i in a consecutive segment of the destinations array. The communication graph effectively provided to the MPI library by a call to this function is the union of all such locally specified edges.

Both functions are collective, meaning that all processes in the comm_old communicator must perform the call. The created graph communicator has the same size as comm_old. For both interfaces, the reorder flag determines whether a reordering of the processes should be attempted. Reordering means that the MPI library is allowed to change the rank of the calling process in the new communicator, which is equivalent to applying a permutation π (cf. Section 1) to the original rank mapping. This enables communication optimization by the MPI library through a process reordering that maps the specified communication pattern to the hardware network topology. Such optimizations must not be attempted if reorder is set to false, but even in this case the interface provides other optimization possibilities (cf. Section 4.1). *All* calling processes must give the same value for reorder (either all true or all false); in case this pre-condition is not respected, the behavior (deadlock or run-time error) and result (wrong or inconsistent comm_dist_graph) of the calls is undefined.

Multiplicity of edges is allowed in the communication graph but is not required to be interpreted by the MPI library. It is recommended that the user specifies each edge in the application communication topology only once as far as possible. However, the laxity of the specification may make it easier for some applications to use the interface: the user does not need to check for and eliminate duplicate edges explicitly. Edges can be given integer weights that can further influence the reordering optimization. A larger weight is intended to indicate heavier communication. The info argument can influence the exact meaning of the edge weights.

All processes must give the same set of info (key,value) pairs. Section 4.3 describes possible info hints. The explicit edge weights can be omitted, which can be useful if the application has a purely binary relationship (communication or no communication) between pairs of processes. In this case, *all* processes supply MPI_UNWEIGHTED as the argument instead of the array of weights. This capability simplifies porting applications from the MPI 1.0 unweighted interface to either of the new MPI 2.2 interfaces. As for the reorder argument, failure on behalf of the user to respect the pre-conditions on info and MPI_UNWEIGHTED will lead to undefined behavior.

The MPI 1.0 interface provides query functions (with local, non-collective semantics) through which each process in the new communicator can query the neighborhood of *any* other process. This construct is also not scalable and has been replaced with more restricted query functions in the MPI 2.2 specification. The scalable specification allows each process to query information about its immediate neighbors in the graph topology. The interface defines two functions to retrieve the size and the structure of the neighborhood:

1. MPI_Dist_graph_neighbors_count(comm_dist_graph, indegree, outdegree, weighted)
2. MPI_Dist_graph_neighbors(comm_dist_graph, maxindegree, sources, sourceweights, maxoutdegree, destinations, destweights)

The latter call returns lists of source and destination neighbors and their associated weights (or MPI_UNWEIGHTED).

In order to support the query functions, the MPI library must store the neighbors of each process in the graph locally. The *adjacent specification* can support the query specifications without communication, unless of course a reordering is performed, in which case the neighbor arrays will have to be transferred to the processor to which the calling process has been mapped. A possible disadvantage of the adjacent specification is that each edge must be supplied twice, namely by the source *and* the destination process. For the *general specification*, the MPI library must determine, in a distributed way, the incoming and outgoing edges of each process in the new communicator and store these locally in order to support the query functions. This check and possible redistribution can be done in $\mathcal{O}(\log P+K)$ time on P (fully connected) processors, with the $K \leq P$ term dependent on the actual amount of redistribution to be done.

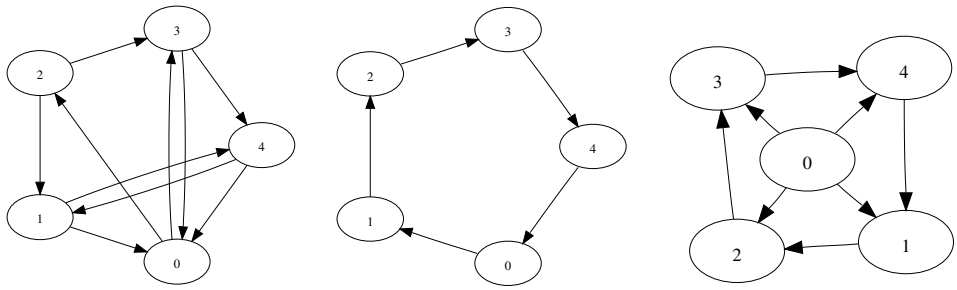
2.1. Example Graph Specifications

We present some example topologies and their specification in order to illustrate the use of the new interface for specifying communication patterns. Figure 2 shows different communication patterns as graph topologies for a communicator with five processes.

All lists of edges and weights are given as integer arrays. For the *adjacent constructor*, the graph is represented as lists of incoming edges given by the parameters indegree, sources, sourceweights and lists of outgoing edges given by the parameters outdegree, destinations, destweights of the calling process, as Table I shows for the example topologies of Figure 2. Note that each directed (s, t) edge is listed as outgoing edge by its source process s and incoming edge by its target process t.

Table I. Parameters for the adjacent constructor for three communication graphs.

Topology/Process	**indegree**	**sources**	**outdegree**	**destinations**
random/0	3	1,3,4	2	2,3
random/1	2	2,4	2	0,4
random/2	1	0	2	1,3
random/3	2	0,2	2	0,4
random/4	2	1,3	2	0,1
ring/i ($0 \leq i \leq 4$)	1	$i-1 \bmod 5$	1	$i+1 \bmod 5$
wheel/0	0	-	4	1,2,3,4
wheel/1	2	0,4	1	2
wheel/2	2	0,1	1	3
wheel/3	2	0,2	1	4
wheel/4	2	0,3	1	1



(a) Random (b) Ring (c) Wheel

Figure 2. Three possible communication topologies for a communicator of size 5.

The *general constructor* has more flexibility in representing the graph. Any (s, t) edge needs only be specified once, and can be supplied by any of the calling processes. Table II shows a possible combination of input arguments to generate the graphs in Figure 2.

2.2. Migration of Legacy MPI 1.0 Graph Topology Applications

Applications that already use the MPI 1.0 graph topology interface can easily (and are urged to) migrate to the new distributed graph interface. We suggest the following steps. Legacy applications already compute or collect the full communication graph on each process, as the MPI 1.0 MPI_Graph_create constructor requires. These applications can instead use

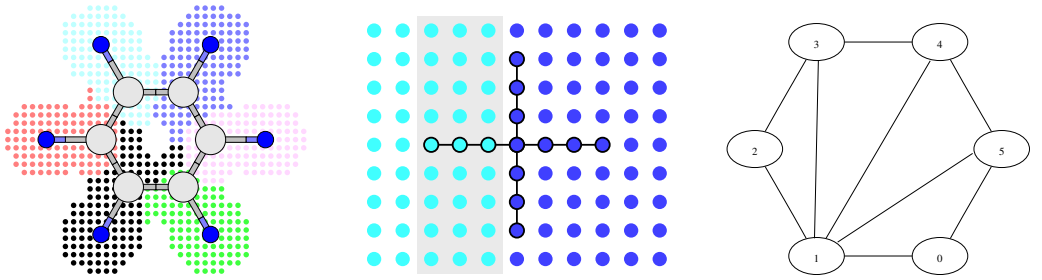
Table II. Possible parameters for the general constructor for three communication graphs.

Topology/Process	**n**	**sources**	**degrees**	**destinations**
random/0	1	2	1	1
random/1	1	1	2	0,4
random/2	2	0,4	2,1	2,3,1
random/3	2	4,3	1,1	0,0
random/4	2	2,3	1,1	3,4
ring/i ($0 \leq i \leq 4$)	1	$i - 1 \bmod 5$	1	i
wheel/0	1	2	1	3
wheel/1	1	3	1	4
wheel/2	1	4	1	1
wheel/3	1	1	1	2
wheel/4	1	0	4	1,2,3,4

either of the new functions MPI_Dist_graph_create_adjacent or MPI_Dist_graph_create. The graph representation must be converted to either the adjacent representation, in which each process specifies its incoming and outgoing neighbors, or the more flexible distributed representation. For the latter, there are several possibilities. At one extreme, only one process (a "root") supplies the communication graph; at another extreme, each process supplies only its outgoing edges. In both cases, as a first approach, the user can omit edge weights by giving MPI_UNWEIGHTED as the value for the weight parameters.

As a next step, the application programmer can gradually eliminate computation and/or communication that was needed to build the full graph on all processes as input for the legacy MPI 1.0 interface. These changes result in a genuine advantage with the new interface since the full graph often may not easily be computed locally, while most edges are likely to be readily available to some process. The adjacency interface is straightforward to use if the processes naturally know their incoming and outgoing communication edges. The distributed interface is the right choice if knowledge of the communication pattern is distributed over the processes.

The reorder argument has the same semantics in the new and old specifications, and for the info argument, the value MPI_INFO_NULL can be used as a default. Useful values for this argument are MPI implementation dependent; users should consult their local installation to determine which types of info may be relevant to their application cases. In general, MPI requires that info values that have no effect are simply ignored; thus, there is no harm to application portability by supplying nontrivial auxiliary information.



(a) Decomposition of a Benzene ring into six partitions.

(b) 13-point stencil at a process boundary.

(c) Resulting communication graph with six processes.

Figure 3. Domain decomposition and nearest neighbor communication in Octopus.

3. Application Examples

This section surveys typical, real-world applications that can benefit from the MPI 2.2 graph topology functionality.

3.1. TDDFT/Octopus

The first example, Octopus [16], is a part of the TDDFT package in which a quantum-mechanical solver uses iterative diagonalization to solve large eigenproblems. Octopus uses finite-difference grids to represent electronic orbitals and distributes the grid by dividing the real-space mesh. Different parts of the grid are assigned to different processes.

Figure 3 illustrates the domain decomposition in Octopus. First, the input domain is decomposed with the METIS library [17] to obtain a good layout with a similar number of points per process [18], as Figure 3(a) shows schematically for the decomposition of a Benzene ring among six processes. The size of the boundary region depends on the computation; Figure 3(b) shows a 13-point stencil that reaches three points into the domain of the neighbor process and hence requires communication. Figure 3(c) shows the final MPI 2.2 graph layout, which uses rank 1 for the black domain in Figure 3(a). This rank is connected to all other domains as one can see in the inner area of Figure 3(a). Although the drawn graph is undirected (information is exchanged in both directions), the MPI interface requires a directed definition, so each edge (u, v) must also be added as edge (v, u).

3.2. Block-Structured Grids

Three-dimensional block-structured grids are used, for example, for the computation of flow fields around complex objects, such as aircrafts or cars, in order to optimize their design. The industrial simulation codes FLOWer (DLR Braunschweig) and NSFLEX (Daimler Benz

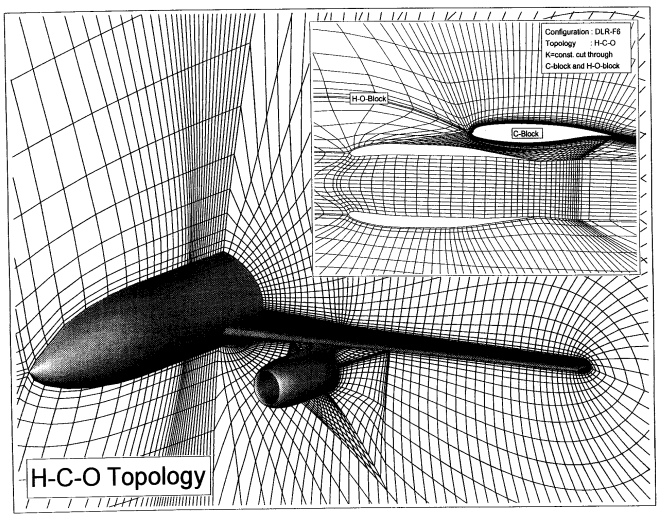


Figure 4. Block structure; source DLR.

Aerospace AG) [19] used block-structured grids for aircraft design on parallel systems as far back as the late 1990s.

The definition of a block-structured grid is fairly general. Each grid block is a logically cubic grid; the six faces (sides) of each block are assumed to be subdivided into $2d$-rectangular segments. Each segment is either a part of the physical (geometrical) boundary or an interface segment; that is, it coincides with a segment of a neighboring block. In general, the computational domain is split into non-overlapping blocks; that is, each mesh line ending at a block boundary must be continued by a mesh line in the neighboring block. Formally, this allows for any kind of computational domain, and such block-structured grids are general enough to cover important realistic flow regions and regular enough to allow for efficient parallelization and vectorization. Since the definition is quite general and the grid generation for complex domains is not simple, grid-topological singularities (see Figure 5) are often included, which makes the correct update of overlap regions a complex task.

The parallelization is typically performed on a block basis; that is, all blocks of the block structure are treated simultaneously. The number of grid points per block may vary widely (depending on domain complexity or mesh sizes) and, therefore, the blocks are mapped in a load-balanced way to the MPI processes available. Within an update of the halo regions, an MPI process has to transfer different amounts of data to the MPI processes working on neighboring blocks. The amount of data depends on the logical connection to the neighboring block and, for nontrivial discretization stencils, the size corresponds to:

- $2d$-region $\times$ overlap width for interface segments,

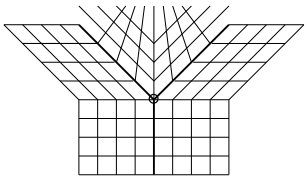


Figure 5. A $2d$ split point in a block structured grid.

- $1d$-region $\times$ square of overlap width for block edges, or
- cube of overlap width for block corners.

The MPI 2.2 distributed graph creation functions allow to specify the MPI processes to which neighboring blocks belong and also the amount of data to be transferred. This capability is critical since the MPI processes may host different numbers of blocks and the amount of data depends on the type of neighbor connection. The significance of data volume for an optimized mapping becomes even more important for self-adaptive refinements using multigrid approaches and Chimera techniques [20] on top of block-structured grids. In both cases, $3d$ volume data needs to be transferred in addition to data transferred in the halo update.

Within a multigrid approach, self-adaptive refinements create new block-structured grids in domain regions where a finer mesh size is required. Since these new block structures must be distributed over all available MPI processes in order to load balance the work, $3d$ volume data must be transferred in the refined domain regions. In the Chimera technique, significantly overlapping block-structured grids are introduced to simplify the grid generation process or to improve the discretization accuracy in critical regions. For this technique, $3d$ volume data must be transferred in the intersection of overlapping grids.

3.3. Multiphysics Applications

Large scale multiphysics applications are becoming common and, thus, the importance of mechanisms that support their optimization is growing significantly. For example, important industrial applications couple fluid dynamic simulations with structure simulations to simulate the interaction of the flow around an aircraft and for airfoil design. Climate research (cf. the PRISM [21–23] project in Europe and the ESMF [24, 25] project in the United States) and weather prediction are other important examples where simulation codes for atmosphere, ocean, ice, chemistry reactions, and other physics work together to simulate a complex scenario. It is common in both application areas that different teams develop the simulation codes, and the coupling is performed by interpolation (at the surface) since the physics codes use different grids. Parallelization and load balance (for example, distribution of the grid to the MPI processes) also depend on the simulation code. Thus, the MPI processes must transfer different amounts of data (corresponding to the size of joint intersections at surfaces) to the

corresponding MPI processes of other applications. This communication can be improved through an appropriate topological layout of the processes along the boundary between them.

Coupled applications can benefit even more from the MPI 2.2 graph mapping functions if coupling data must be exchanged for $3d$ intersections. $3d$ intersection transfers are required when local and global modeling is used in climate research. Simulating atmosphere and chemistry in the atmosphere in different codes is another example for $3d$ exchange of coupling data and the requirement of a good mapping of MPI processes.

3.4. Graph Partitioning Libraries

Graph partitioning is often used for load balancing of computational effort and communication [26,27]. In such applications, the problem domain is typically partitioned by a graph partitioning library, such as METIS, ParMETIS [17], SCOTCH [28], PT-SCOTCH [29], or others [30]. Each edge in the domain graph typically reflects some aspect of communication, and the vertices model computation. A good partitioning over the computational nodes will typically assign each vertex to a process such that the computation is balanced and the communication is minimized. One can naturally combine the partition with the MPI topology interface to map the communication between the partitions to the hardware communication system. However, this two-stage approach is not optimal, even if both the graph partition and the process-to-processor mapping stage would be optimal, because the domain decomposition assumes an underlying fully connected network graph [6]. Nonetheless, it can be expected to yield improvements. Here, it is indicated how to use a domain graph partition computed by METIS or ParMETIS[‡] as input to the MPI 2.2 topology constructors.

METIS is often applied to problems where the data are replicated among all processes and only the computation is performed in parallel. This suits (that is, is scalable for) only instances with small domains and comparatively long computations, as is often the case for combinatorial search problems. If a deterministic partitioning algorithm is used, the graph partitioning can be done sequentially on all processes in parallel and the adjacent topology creation interface be used to specify the graph topology. The communication pattern consists of the edges between the partitions computed by METIS. The input for the MPI 2.2 adjacent topology interface MPI_Dist_graph_create_adjacent can be computed by examining all edges from or to the partition that is assigned to the local process. Listing 1 shows this strategy. Here, the edge weights are ignored for simplicity but can be easily added.

This simple solution does not scale because it replicates and scans the whole domain graph at each process. A better strategy would limit the scan to the edges that originate in the local partition although large parts of the domain graph may then have to be redistributed afterwards if the reorder argument is set to true.

To unleash the full potential of the scalable graph topology interface, a distributed partitioner such as ParMETIS must be used. ParMETIS inputs a distributed graph in distributed CSR format, in which each process specifies the edges that originate at a block of

[‡]See `http://glaros.dtc.umn.edu/gkhome/views/metis` (2010)

Listing 1. Creating a distributed communication graph description from METIS output.

```
/* create input for METIS, fill xadj and adjncy arrays */

/* call METIS, e.g., k-way partitioning; n must be equal
   to communicator size */
METIS_PartGraphKway(n, xadj, adjncy, ..., part);

/* create MPI 2.2 adjacency lists sources and targets */
int srcind=0, tgtind=0;
for(int src=0; src<n; src++) /* loop over vertices */
  for(int tgt=xadj[src]; tgt<xadj[src+1]; tgt++) {
    if(part[src] != part[tgt]) { /* found a remote edge */
      if(part[tgt] == myrank) sources[srcind++] = tgt;
      if(part[src] == myrank) targets[tgtind++] = src;
  } }

/* create MPI 2.2 graph topology */
MPI_Dist_graph_create_adjacent(comm, srcind, sources, MPI_UNWEIGHTED,
  tgtind, targets, MPI_UNWEIGHTED, MPI_INFO_NULL, 1, new_comm);
```

vertices. ParMETIS returns the partitioning in a block-distributed array. Thus, communication is only required when the distribution of the permutation (the partition number of a target process of a given edge) is not known locally to check if a given edge in the partitioned graph crosses a partition (process) boundary.

A simple solution is to distribute the permutation to all processes (for instance, with MPI_Allgather) and apply an algorithm as outlined in Listing 1. However, in large-scale systems individual processes may typically not have enough memory to store the full array (or graph). A request-response protocol is a scalable scheme for the topology creation for a directed graph. When a process encounters a remote edge, it requests the partition number for the target vertex from the owner. Alternatively, this protocol could be implemented collectively as a request and response round with two global, sparse data exchanges. The design of such protocols depends on the architecture and approximate structure of the graph (for an overview of scalable data exchange protocols is provided, see [31]). However, for ParMETIS the symmetry of the graph can be exploited to optimize the protocol. In symmetric graphs, each process knows which processes need permutation information from it. Thus, each process can simply push the permutation information along each remote edge in the graph. A scalable redistribution algorithm could, for example, be based on the Personalized Exchange ($\mathcal{PEX}$) algorithm [31]. This algorithm fully distributes partitioning, graph creation, and data redistribution and does not require global information to be collected at any process. The fully distributed handling comes at a cost of complexity. The library MPIParMETIS[§] hides this complexity and creates a topological communicator from ParMETIS parameters and output (graph specification and

[§]See `http://www.unixer.de/MPIParMETIS/` (2010)

the local permutations, respectively). This library can significantly aid the development of highly scalable applications.

3.5. Dealing with Special Resources and Nodes

Some applications require that distinguished processes run on special, dedicated processor nodes, e.g. nodes with I/O capabilities or hardware accelerators. Processes on such nodes should remain fixed under any topology remapping performed by the MPI library. This cannot be explicitly specified in the MPI 2.2 topology interfaces (cf. Section 5.1). One can circumvent the problem by first remapping the processes and then announcing the ranks of the node(s) with the special capabilities to all other ranks by an MPI_Allreduce or MPI_Allgather operation. However, this straightforward solution does not optimally exploit the provided topology information as discussed in Section 5.1.

4. Implementation Hints for MPI Developers

Both of the new graph topology interfaces can trivially be implemented on top of the MPI 1.0 topology functionality. From a distributed communication graph given in either the MPI 2.2 adjacent or general format a full communication graph must be collected by all processes in the MPI 1.0 graph format, after which the MPI 1.0 MPI_Graph_create function can be called. While this solution is not optimal and defies the purpose of the MPI 2.2 specification, it may be acceptable for small systems and can be used as a start for MPI libraries aiming to support the MPI 2.2 standard. A reference implementation of this is available online[¶]. Of course, a scalable implementation interface cannot consume the quadratic memory that is implied in the worst case by this solution for storing the entire communication graph at each individual process. Therefore, scalable and distributed implementations of the new interface are needed.

In the following implementations that can support the distributed neighborhood query functions are outlined, and approaches to nontrivial reordering optimizations are discussed.

The MPI standard mandates that neighbor queries can be satisfied locally at each process. Thus, in a minimal implementation each process must cache its direct (incoming and outgoing) neighbors in the graph topology. The caching is trivial for the adjacent specification since the user must locally specify the complete neighborhood of each process, and only this information needs to be stored. Thus, the functions do not require any communication with the adjacent specification, which was a major reason for including it in the MPI 2.2 standard. In case reordering has been performed (by reorder set to true by all processes) the neighborhood information may have to be redistributed.

The general distributed graph constructor, however, necessitates a communication step in order to construct the neighborhood information for each process. Each process must send every locally specified edge to the source and target processes of the edge. This exchange can

[¶]See `http://www.unixer.de/research/mpitopo/` (2010)

be implemented with the sparse data exchange protocols mentioned in [31]. Listing 2 in the Appendix gives a scalable implementation based on the Personalized Census Exchange ($\mathcal{PCX}$) algorithm [31].

4.1. Optimization for Connection Management

For some communication systems, such as TCP or InfiniBand, a connection between any two (communicating) MPI processes in a communicator eventually must be set up and maintained. Doing this eagerly and explicitly requires $\Theta(P^2)$ resources (time and space) for communicators with P processes which is prohibitive for large systems. To circumvent this problem many MPI implementations use lazy connection management in which connections are set up on first use [32]. However, this approach might have unfavorable side effects, for example, load imbalance and unpredictable wait times during connection establishment. The graph topology interface can guide eager connection management based on the user-provided communication pattern: connections are set up eagerly for those processes for which a communication edge is specified. The MPI library must still offer all P^2 connections and would need to implement lazy connection establishment for the remaining channels. This type of optimization is independent of whether reorder is set to true or false.

4.2. Reordering and Embedding Optimization Problems

A general formalism for the optimization problem presented by the topology interface is the *(weighted) graph embedding problem*. A weighted, directed guest graph that describes the application communication pattern must be embedded into a (directed) weighted host graph that describes the hardware communication system [33–35]. For the MPI specification, both guest and host graphs have the same number of nodes, and the embedding is a one-to-one mapping of nodes (MPI processes) in the guest graph to nodes (processors) of the host graph.

Different optimization criteria could be relevant:

- minimize total load (*congestion*) of the host graph edges (number of times any host graph edge is used in a path that is an image of a guest graph edge, see Figure 1) to improve the utilization of total system bandwidth;
- minimize the maximum load on any host graph edge to minimize bottlenecks;
- minimize the maximum *dilation* (distance in host graph between neighbors in guest graph) to improve latency
- minimize average dilation

and others, in weighted or unweighted formulations. Suitable info values could be used to convey the optimization objective to be pursued to the MPI library. Edge weights in the host graph could for instance model a communication system capacity that can only be exceeded at a cost.

These kinds of problems are typically NP-hard [36], although polynomial solutions (may) exist for relevant, special cases of host and/or guest graphs. For special systems, variants of the optimization problems reduce to simpler problems. For SMP clusters, for instance, the

embedding can be solved by weighted graph partitioning [15], for which efficient, parallel heuristics exist [37].

In general, it cannot be expected that an MPI library will find an optimal solution. However, this limitation may not be too significant since the abstraction of modeling communication patterns by graphs is in any case approximate and imprecise (e.g., it does not capture any temporal properties of the communication). Furthermore, good heuristics or approximate solutions to the relevant optimization problem will generally be sufficient. Nonetheless, a major research challenge is to devise efficient, distributed algorithms for these problems. To our knowledge, little is known regarding the embedding problems.

Reordering may imply that all P processes must store the mapping information which requires $\Theta(P)$ space for each process or $\Theta(P^2)$ in total (across all processes), which does not scale on a large number of processes [4]. Thus, besides the optimization criteria mentioned above, an additional criterion maybe the minimization of the space needed to store such a mapping in a compressed way. If the number of edges is sparse (that is, each process has only a limited number of neighbors in the graph topology), the storage required for the rank mapping can be reduced. For example, on architectures that offer remote direct memory access, the mapping in both directions need be stored once only in a distributed way on a subset of the processes, and on each process a local cache can be used for frequently used ranks. This cache can be implemented as a hash table and initialized with the neighbors defined in the graph topology.

4.3. Possible Optimization Hints

In MPI 2.2 no (key,value) pairs are standardized for the info argument. Some relevant info argument (key,value) pairs to control the mapping performed by MPI implementations based on the different optimization criteria for the graph embedding problem could be:

- (optimize_latency, average): Useful for latency-bound applications that can tolerate some variance in latency across the processes (e.g., loosely synchronous or asynchronous algorithms with adaptive load balancing);
- (optimize_latency, maximum): Useful for synchronous latency-bound applications for which the slowest process determines overall performance (e.g., iterative solvers that perform a blocking MPI_Allreduce during each iteration)
- (optimize_bandwidth,average): Useful for loosely synchronous or asynchronous bandwidth-bound applications;
- (optimize_bandwidth,maximum): Useful for synchronous bandwidth-bound applications;
- (edge_weights,coalesce): Useful for multigraphs (graphs with multiple parallel edges) – for example, coalesce (add) weights of parallel edges into a single edge or use different edge reduction functions to guide topology decisions for multigraphs.

Experience from applications and implementations will guide future developments of the MPI standard, and may result in a set of standardized (key,value) pairs for the info argument. As for other uses of MPI_Info objects throughout the MPI standard, implementations are permitted to define their own values but should document such additional values. A particular

implementation should ignore (key,value) pairs that is does not know. Thus, such practice does not compromise application portability.

5. The MPI Standardization Effort

This section discusses the background and rationale for the decisions that were made by the MPI Forum[‖] while standardizing the new MPI 2.2 topology interface. Potential future enhancements and additions to the topology mechanism are also briefly outlined.

5.1. Discussion of the MPI 2.2 Interface

The new MPI 2.2 topology interface alleviates the obvious shortcomings of the MPI 1.0 specification, but retains the basic abstraction of communication patterns as directed graphs. The interface thus cannot assist with communication optimization when communication along different edges takes place at different times, communication is interleaved with different amounts of computations, or similar, more complex situations. The interface can only convey a total amount of communication (volume or frequency) to the MPI library, thus allowing the library to allocate heavy communication to parts of the hardware network with correspondingly high capacity and/or seek to avoid congestion. Unlike the MPI 1.0 interface, the MPI 2.2 interfaces both return a communicator with the same size as the calling communicator; that is, all calling processes are implicitly nodes of the communication graph.

The info object can provide more information to the MPI implementation about the relevant optimization criteria, such as time-quality trade-offs or possible interpretations of the edge weights (frequency or volume, for instance). During the MPI 2.2 standardization process, the MPI Forum decided not to prescribe any specific info values. Thus, each MPI library implementation is free to define sensible values for the info argument. The MPI Forum is actively soliciting feedback from applications and implementations on standardizable values for this argument. All calling processes must provide the same value for the info argument, but an MPI implementation need not follow specific hints, and MPI_INFO_NULL is, in any case, always a legal (default) value.

The calling processes must also *all* provide the same value for the reorder argument. The MPI Forum considered letting this argument have local semantics, that is, giving each calling process in the graph the option of indicating whether or not the implementation may remap that particular process. This additional freedom would allow an implementation to accommodate special resources (i.e., processors that have special capabilities such as I/O, special co-processors, or extended memory) by keeping these fixed in the resulting communicator. That is, process i having such special capabilities in the calling comm_old by virtue of residing on some processor I and being connected to a number of other processes in the communication graph would remain mapped to processor I in the resulting comm_dist_graph. An application

[‖]See `http://www.mpi-forum.org`

could specify this restriction by calling with reorder set to false for this process. The MPI Forum decided against this (natural) feature, finding it a too complex deviation from the practice established with the MPI 1.0 interface.

It should be noted that it is not possible to mimic the potential to optimize wrt. fixed processes with global (all true or all false) reorder semantics. The processors with special capabilities that possibly have a different rank in the resulting communicator could, of course, announce their new rank to the other processes in the communicator (by means of a collective call), but the processes now physically close to them may no longer be the ranks that the application needed to have close in the communication graph. The only way to ensure that special processes remain mapped to the special processors is by removing them entirely from the communicator used in the call to the topology constructor. Not only is this solution tedious because it requires the user to restrict the remaining communication graph to the smaller communicator, but it loses all information about communication with the special processors, which eliminates the potential to optimize the placement of their neighborhoods.

In both the adjacent and general MPI 2.2 topology interfaces only the communication *edges* can be weighted. Process, that is *node* weights, could have been used to capture relative computation requirements and would provide for even more optimization potential (computational load balancing) for processor-heterogeneous systems by trying to fit communication (or memory) requirements to the resources of the different types of processors in the system. The MPI Forum did not discuss this possibility.

5.2. Potential Future Extensions

The MPI 2.2 distributed topology interface, although complete and more expressive than the MPI 1.0 interface, is open towards extensions and enhancements. Future versions of the MPI standard may introduce additional, even more expressive interfaces to better support adaption of applications to hardware and system communication capabilities. Feedback from MPI user and implementation communities will influence these directions. Since the MPI Forum is committed to maintain backward compatibility, all information and hints in this article will remain valid in the future.

The MPI Forum may consider additional support and helper functionality for manipulating and constructing communication graphs, for instance mechanisms to set up the (sometimes tedious) distributed graph descriptions from other descriptions or implicitly represented communication patterns. An immediate extension along these lines is functionality to extract a distributed graph description from a Cartesian MPI communicator with functionality to add or remove edges to such graphs. This could be used to easily capture mesh-like applications with more complex neighborhoods as discussed in Section 3.2. Another issue is to determine a set of standard values that the info argument may take. A somewhat different issue is socalled *sparse* collective communication operations that would be defined on process topologies, such as nearest-neighbor gather, alltoall, or reduce [38]. Such operations are more difficult to implement efficiently on top of existing MPI functionality, and it may therefore make sense to include them in a future standard.

Other directions could be the definition of functionality to assists with data redistribution, although existing point-to-point and collective communication operations can handle these

aspects. Similarly, special libraries that build on MPI could encapsulate much of the helper functionality discussed above. In Section 3 such an example library that implements an interface between ParMETIS and the MPI 2.2 topology functionality was discussed.

Finally, the MPI Forum might decide to deprecate parts of the MPI 1.0 functionality, first and foremost the non-scalable graph constructor and its non-scalable query functionality.

6. Summary

This paper described the new MPI 2.2 distributed graph topology interface which is a scalable, informative, and easy to use mechanism for conveying information on communication patterns from the application to the MPI library. This information in turn can be used by the MPI library to improve the mapping of processes to processing elements to better fit the application communication requirements to the hardware capabilities. The paper complements the MPI 2.2 standard document itself by including discussions on applications that can benefit from the distributed topology interfaces, outlining optimization problems that must be addressed by MPI implementers, and providing extended background information.

Hopefully application programmers will embrace the new interface and actually start using it, possibly as a good replacement for the legacy MPI 1.0 interface. In turn, this will put pressure on MPI implementers (much more so than was the case for MPI 1.0) to provide efficient implementations that can fulfill the promise of this of functionality. In the future the MPI Forum may discuss further enhancements and additional helper functionality. Feedback from application programmers and MPI implementers will be most valuable to guide such enhancements.

Acknowledgments

We thank all other members of the MPI Forum that actively participated in the standardization process for the scalable graph topology interface. We thank Bill Gropp (UIUC) for chairing the MPI 2.2 standardization effort. We thank Florian Lorenzen (FU Berlin) for support with the TDDFT/Octopus application.

This work was supported in part by the Lilly Endowment, DOE FASTOS II (LAB 07-23), and the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under contract DE-AC02-06CH11357.

REFERENCES

1. MPI Forum. *MPI: A Message-Passing Interface Standard. Version 2.2* June 23rd 2009. `www.mpi-forum.org`.
2. Snir M, Otto SW, Huss-Lederman S, Walker DW, Dongarra J. *MPI: The Complete Reference*. MIT Press, 1996.
3. Calkin R, Hempel R, Hoppe HC, Wypior P. Portable programming with the PARMACS message-passing library. *Parallel Computing* 1994; **20**(4):615–632.

4. Balaji P, Buntinas D, Goodell D, Gropp W, Kumar S, Lusk E, Thakur R, Träff JL. MPI on a million processors. *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 16th European PVM/MPI Users' Group Meeting*, *Lecture Notes in Computer Science*, vol. 5759, Springer, 2009; 20–30.
5. Träff JL. SMP-aware message passing programming. *Eighth International Workshop on High-level Parallel Programming Models and Supportive Environments (HIPS03), International Parallel and Distributed Processing Symposium (IPDPS 2003)*, 2003; 56–65.
6. Berti G, Träff JL. What MPI could (and cannot) do for mesh-partitioning on non-homogeneous networks. *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI Users' Group Meeting*, *Lecture Notes in Computer Science*, vol. 4192, Springer, 2006; 293–302.
7. Bhatelée A, Kalé LV. Benefits of topology aware mapping for mesh interconnects. *Parallel Processing Letters* 2008; **18**(4):549–566.
8. Dally WJ. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers* 1990; **39**(6):775–785.
9. Tvrdík P, Harbane R, Heydemann MC. Uniform homomorphisms of de Bruijn and Kautz networks. *Discrete Applied Mathematics* 1998; **83**(1–3):279–301.
10. Hoefler T, Schneider T, Lumsdaine A. Multistage Switches are not Crossbars: Effects of Static Routing in High-Performance Networks. *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, IEEE Computer Society, 2008.
11. Mercier G, Clet-Ortega J. Towards an efficient process placement policy for MPI applications in multicore environments. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 16th European PVM/MPI Users' Group Meeting*, *Lecture Notes in Computer Science*, vol. 5759, 2009; 104–115.
12. Jeannot E, Mercier G. Near-optimal placement of MPI processes on hierarchical NUMA architectures. *Euro-Par 2010 Parallel Processing*, Lecture Notes in Computer Science, 2010. To appear.
13. Hatazaki T. Rank reordering strategy for MPI topology creation functions. *5th European PVM/MPI User's Group Meeting*, *Lecture Notes in Computer Science*, vol. 1497, Springer, 1998; 188–195.
14. Yu H, Chung IH, Moreira JE. Topology mapping for Blue Gene/L supercomputer. *ACM/IEEE Supercomputing*, 2006; 116.
15. Träff JL. Implementing the MPI process topology mechanism. *Supercomputing*, 2002. `http://www.sc-2002.org/paperpdfs/pap.pap122.pdf`.
16. Castro A, *et al.*. `octopus`: a tool for the application of time-dependent density functional theory. *Phys. stat. sol. (b)* 2006; **243**(11):2465–2488.
17. Karypis G, Kumar V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.* 1998; **20**(1):359–392.
18. Hoefler T, Lorenzen F, Lumsdaine A. Sparse non-blocking collectives in quantum mechanical calculations. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting*, *Lecture Notes in Computer Science*, vol. 5205, Springer, 2008; 55–63.
19. Schüller A ((ed.)). *Portable Parallelization of Indsutrial Aerodynamic Applications (POPINDA)*, *Notes on Numerical Fluid Mechanics*, vol. 71. Vieweg, 1999.
20. Meakin RL. *Composite Overset Structured Grids*, chap. 11. CRC Press, 1999.
21. Redler R, Valcke S, Ritzdorf H. OASIS4 – a coupling software for next generation earth system modelling. *Geoscientific Model Development* 2010; **3**(1):87–104. URL `http://www.geosci-model-dev.net/3/87/2010/`.
22. Valcke S, Guilyardi E, Larsson C. PRISM and ENES: a european approach to earth system modelling. *Concurrency and Computation: Practice and Experience* 2006; **18**(2):247–262.
23. Valcke S, Budich R, Carter M, Guilyardi E, Foujols MA, Lautenschlager M, Redler R, Steenman-Clark L, Wedi N. A european network for earth system modelling. *EOS, Transactions American Geophysical Union* 2007; **88**(12):143 pages, doi:10.1029/2007EO120003.
24. Hill C, DeLuca C, Balaji, Suarez M, da Silva A. The architecture of the Earth System Modeling Framework. *Computing in Science and Engineering* 2004; **6**(1):18–28.
25. Zhou S. Coupling climate models with the earth system modeling framework and the common component architecture. *Concurrency and Computation: Practice and Experience* 2006; **18**(2):203–213.
26. Walshaw C, Cross M. Mesh partitioning: a multilevel balancing and refinement algorithm. *SIAM Journal on Scientific Computing* 2000; **22**(1):63–80.
27. Walshaw C, Cross M. Multilevel mesh partitioning for heterogeneous communication networks. *Future Generation Computer Systems* 2001; **17**(5):601–623.
28. Pellegrini F, Roman J. SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. *High-Performance Computing and Networking, International Conference*

and Exhibition, HPCN'96, Brussels, Belgium, April 15-19, *Lecture Notes in Computer Science*, vol. 1067, Springer, 1996; 493–498.
29. Chevalier C, Pellegrini F. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Computing* 2008; **34**(6–8):318–331.
30. Pellegrini F. Graph partitioning based methods and tools for scientific computing. *Parallel Computing* 1997; **23**(1–2):153–164.
31. Hoefler T, Siebert C, Lumsdaine A. Scalable Communication Protocols for Dynamic Sparse Data Exchange. *Proceedings of the 2010 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ACM, 2010; 159–168.
32. Shipman GM, Woodall TS, Graham RL, Maccabe AB, Bridges PG. InfiniBand scalability in Open MPI. *Proceedings of IEEE Parallel and Distributed Processing Symposium*, 2006.
33. Hong JW, Mehlhorn K, Rosenberg AL. Cost trade-offs in graph embeddings, with applications. *Journal of the ACM* 1983; **30**(4):709–728.
34. Monien B, Sudborough H. Embedding one interconnection network into another. *Computational Graph Theory*, *Computing Suppl.*, vol. 7, Tinhofer G, Mayr E, Noltemeier H, Syslo MM (eds.). Springer, 1990; 257–282.
35. Rosenberg AL. Issues in the study of graph embeddings. *Graphtheoretic Concepts in Computer Science. International Workshop (WG)*, *Lecture Notes in Computer Science*, vol. 100, Springer, 1981; 150–176.
36. Garey MR, Johnson DS. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979. With an addendum, 1991.
37. Holtgrewe M, Sanders P, Schulz C. Engineering a scalable high quality graph partitioner. *International Parallel and Distributed Processing Symposium (IPDPS 2010)*, 2010. To appear.
38. Hoefler T, Träff JL. Sparse Collective Operations for MPI. *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium, HIPS'09 Workshop*, 2009.

Appendix: Pseudo-Code for the neighborhood construction algorithm

This appendix gives the C-like pseudocode mentioned in Section 4 for the construction of the neighborhood information from a distributed graph as needed to support the neighborhood query functions. The code is for illustration purposes only, details for a full-fledged implementation have to be filled in as needed.

The input to the algorithm in Listing 2 are the parameters of `MPI_Dist_graph_create`. The arrays of neighbors to be stored locally arrive in the array `edgebuf` at the end of the algorithm. The lists of edge weights are constructed analogously, but the obvious details have been omitted.

Listing 2. Scalable gathering of neighborhood information during topology creation.

```
/* distribute input edges into lists of outgoing and incoming edges */
int index = 0;
for(int i=0; i<n; ++i) {
  outsize[i] = 0; insize[i] = 0;
}
/* scan through sources and destinations */
for(int i=0; i<n; ++i) {
  for(int j=0; j<degrees[i]; ++j) {
    outlist[sources[i]][outsize[sources[i]]++] = destinations[index]
    inlist[destinations[index]][insize[destinations[index]]++] =
      sources[i];
    index++;
  }
```

```
}

/* set count for processes for which this process has incoming or
   outgoing edges */
int localcount[2*comm_size];
int totalcount[2];
for(int i=0; i<comm_size; ++i) {
  if(insize[i]>0) localcount[2*i]=1; else localcount[2*i]=0;
  if(outsize[i]>0) localcount[2*i+1]=1; else localcount[2*i+1]=0;
}

/* compute total number of processes with incoming or outgoing edges
   for each process. Use scalable MPI 2.2 function for this */
MPI_Reduce_scatter_block(&localcount[0], &totalcount[0], 2, MPI_INT,
                         MPI_SUM, comm_old);

/* post nonblocking sends to transfer non-empty edge lists to
   respective destination processes */
sends=0;
for(int i=0; i<p; ++i) {
  if(insize[i]>0)
    MPI_Isend(&inlist[i], insize[i], MPI_INT, i, INEDGETAG,
              comm_old, &request[sends++]);
  if(outsize[i]>0)
    MPI_Isend(&outlist[i], outsize[i], MPI_INT, i, OUTEDGETAG,
              comm_old, &request[sends++]);
}

/* receive all non-local incoming and outgoing edges */
for(int j=0; j<2; ++j) {
  for(int i=0; i<totalcount[j]; ++i) {
    MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm_old, &status);
    int source = status.MPI_SOURCE;
    int tag = status.MPI_TAG;
    MPI_Get_count(status, MPI_INT, &edges);
    /* allocate edgebuf */
    MPI_Recv(edgebuf, edges, MPI_INT, source, tag, comm_old,
      MPI_STATUS_IGNORE);
    /* add received edges to local neighborhood array as either
       source or as destination, depending on tag */
  }
}
```