Topological Collectives for MPI-2

Torsten Hoefler[1], Florian Lorenzen[2], Douglas Gregor[1] and Andrew Lumsdaine[1]

[1]Open Systems Lab, Indiana University
[2]Freie Universtät Berlin

February 11, 2008

1 Introduction

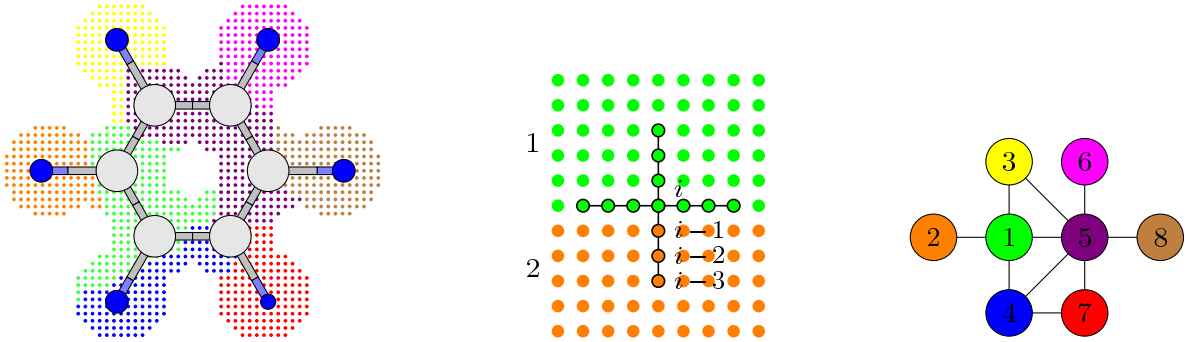
MPI topologies currently provide the MPI implementation with information about the typical communication behavior of the processes in a new communicator, illustrating the structure of communication via a Cartesian grid or a general graph. Topologies contain important application- and data-specific information that can be used for optimized collective implementations and improved mapping of processes to hardware. However, from the application perspective, topologies provide little more than a convenient naming mechanism for processes.

This proposal introduces new collective operations that operate on communicators with process topologies. These topological collectives express common communication patterns for applications that use process topologies, such as nearest-neighbor data exchange and shifted Cartesian data exchange. These collectives are usually implemented by the application programmer. However, a performant implementation of those operations is not trivial and programmers frequently face problems with deadlocks.

Nearest neighbor exchanges are, e. g., required in real space electronic structure codes that represent physical quantities on a discrete mesh (like [1]). The mesh's points are distributed among the nodes to increase computational throughput. Figure 1(a) shows the geometry of a benzene molecule on top of a real space grid. The grid points are divided into eight partitions.

Communication between adjacent partitions is necessary to calculate derivatives by finite-difference formulas, or, more generally, any nonlocal operator. Figure 1(b) shows the situation for a third order discretization: to calculate the derivative at point i, the values at points $i-1, \ldots, i-3$ have to be communicated from partition 2 to partition 1.

The communication structure can be represented as a graph with each vertex representing one computational node and edges representing neighboring partitions (cf. Figure 1(c)). Using this abstraction, the data exchange prior to the calculation of a derivative can be mapped onto a general nearest neighbor communication pattern.



(a) The benzene ring distributed on eight nodes.

(b) A third order stencil leaking into a neighboring partition.

(c) The communication pattern for the benzene ring.

Figure 1: Process topologies in real space electronic structure codes.

All proposed collective operations have been implemented in LibNBC [3] (as non-blocking operations, but they can easily be used in a blocking way). An experimental version of the software package Octopus [1] uses those operations and shows good results.

2 Proposed Extensions

2.1 Nearest Neighbor Communication

We propose to add a new collective function named MPI_NEIGHBOR_XCHG that performs nearest-neighbor communication on all processes in a communicator with a topology. Each process transmits data to and receives data from each of its neighbors.

```
int MPI_Neighbor_xchg(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                      void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

MPI_NEIGHBOR_XCHG(SENDBUF, SENDCOUNT, SENDTYPE,
                  RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR

void Topocomm::Neighbor_xchg(const void* sendbuf, int sendcount, const Datatype& sendtype,
                             void* recvbuf, int recvcount, const Datatype& recvtype) const
```

IN sendbuf starting address of send buffer
IN sendcount number of elements sent to each neighbor
IN sendtype data type of the send buffer elements
OUT recvbuf address of receive buffer
IN recvcount number of elements received from any neighbor
IN recvtype data type of receive buffer elements
IN comm communicator

The neighbors of a process in the communicator can be determined by MPI_COMM_NEIGHBORS. As part of this collective, each process sends data to and receives information from each of its neighbors. The type signature associated with sendcount, sendtype at a process must be equal to the type signature associated with recvcount, recvtype at any other process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of neighboring processes. As usual, however, the type maps may be different.

The outcome is as if each process executed a send to each of its neighbors with a call to,

MPI_Send(sendbuf + i∗sendcount∗extent(sendtype), sendcount, sendtype, neighbor[i], ...),

and a receive from every neighbor with a call to,

MPI_Recv(recvbuf + i∗recvcount∗extent(recvtype), recvcount, recvtype, neighbor[i], ...).

All arguments on all processes are significant. The argument comm must have identical values on all processes. If provided with a communicator that does not have a topology, returns MPI_ERR_TOPOLOGY.

int MPI_Neighbor_xchgv(**void**∗ sendbuf, **int** ∗sendcounts, **int** ∗sdispls, MPI_Datatype sendtype,
void∗ recvbuf, **int** ∗recvcounts, **int** ∗rdispls, MPI_Datatype recvtype,
MPI_Comm comm)

MPI_NEIGHBOR_XCHGV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE,
RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)
<type> SENDBUF(∗), RECVBUF(∗)
INTEGER SENDCOUNTS(∗), SDISPLS(∗), SENDTYPE, RECVCOUNTS(∗),
RDISPLS(∗), RECVTYPE, COMM, IERROR

void Topocomm::Neighbor_xchgv(**const void**∗ sendbuf, **const int** sendcounts[], **const int** sdispls[],
const Datatype& sendtype, **void**∗ recvbuf, **const int** recvcounts[],
const int rdispls[], **const** Datatype& recvtype) **const**

IN sendbuf starting address of send buffer (choice)
IN sendcounts integer array (of length number-of-neighbors) specifying the number of elements to send to each neighbor
IN sdispls integer array (of length number-of-neighbors). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for neighbor j
IN sendtype data type of send buffer elements (handle)
OUT recvbuf address of receive buffer (choice)
IN recvcounts integer array (of length number-of-neighbors) specifying the number of elements that can be received from each neighbor
IN rdispls integer array (of length number-of-neighbors). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i
IN recvtype data type of receive buffer elements (handle)
IN comm communicator (handle)

MPI_Neighbor_xchgv adds flexibility to MPI_Neighbor_xchg by making it possible for different neighbors to receive a different amount of data.

The j^{th} block sent from process i is received by its j^{th} neighbor and is placed in the k^{th} block of recvbuf, where i is the k^{th} neighbor of its j^{th} neighbor. These blocks need not all have the same size.

The type signature associated with sendcount[j], sendtype at process i must be equal to the type signature associated with recvcount[k], recvtype at the j^{th} neighbor of process i. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. Distinct type maps between sender and receiver are still allowed.

The outcome is as if each process sent a message to every other process with,

MPI_Send(sendbuf + displs[i]∗extent(sendtype),sendcounts[i],sendtype,neighbor[i], ...),

and received a message from every other process with a call to

MPI_Recv(recvbuf + displs[i]∗extent(recvtype),recvcounts[i],recvtype,neighbor[i], ...).

All arguments on all processes are significant. The argument comm must have identical values on all processes. If provided with a communicator that does not have a topology, returns MPI_ERR_TOPOLOGY. Figure 2 shows an example of the communication operation.

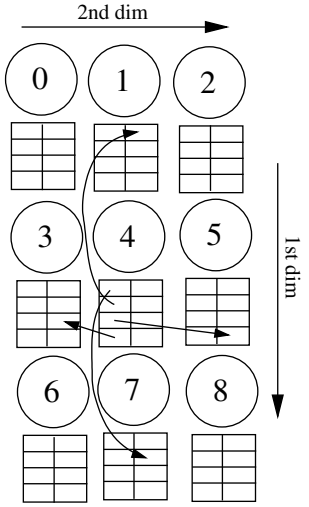


Figure 2: MPI_Neighbor_exchange, illustrating the communication operations originating at rank 4 in a 2-dimensional cartesian communicator. The left side of the buffer represents the send memory and the right side the receive memory.

2.2 Neighbor Query

We propose two new functions that allow one to determine the neighbors of any communicator with a topology. These functions are modeled after MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS, but they work for both communicators with graph topology and for communicators with Cartesian topology.

int MPI_Comm_neighbors_count(MPI_Comm comm, **int** rank, **int** *nneighbors)

MPI_COMM_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
INTEGER COMM, RANK, NNEIGHBORS, IERROR

int Topocomm::Get_neighbors_count(**int** rank) **const**

IN comm communicator with graph or cartesian topology (handle)
IN rank rank of process in group of comm (integer)
OUT nneighbors number of neighbors of specified process (integer)

For a graph communicator, nneighbors will receive the number of neighbors in the graph (i.e., the same result that MPI_Graph_neighbors_count provides). For a cartesian communicator, nneighbors will receive the number of processes whose distance from the given rank is 1. If provided with a communicator that does not have a topology, returns MPI_ERR_TOPOLOGY.

int MPI_Comm_neighbors(MPI_Comm comm, **int** rank, **int** maxneighbors, **int** *neighbors)

MPI_COMM_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR

void Topocomm::Get_neighbors(**int** rank, **int** maxneighbors, **int** neighbors[]) **const**

IN comm communicator with graph or cartesian topology (handle)
IN rank rank of process in group of comm (integer)
IN maxneighbors size of array neighbors (integer)
OUT neighbors ranks of processes that are neighbors to specified process (array of integer)

For a graph communicator, neighbors receives the ranks of each of the neighbors of process rank (i.e., the same result that MPI_Graph_neighbors provides). The order of the ranks in neighbors is unspecified, but successive calls to MPI_Comm_neighbors for the same communicator will return ranks in the same order.

For a cartesian communicator, neighbors receives the ranks of each of the neighbors of process rank. The order of the ranks in neighbors is first along the first dimension in displacement +1 and -1 (either of which may wrap around, if the topology is periodic in that dimension, or will be omitted, if not periodic in that dimension), then along the second, third and higher dimensions if applicable.

If provided with a communicator without a topology, returns MPI_ERR_TOPOLOGY.

Remark: MPI_Graph_neighbors_count and MPI_Graph_neighbors could be deprecated with this proposal.

2.3 Cartesian Shift Communication

We propose to add a new collective function named MPI_CART_SHIFT_XCHG that performs a shift operation along a certain dimension of a cartesian communicator. The operation acts

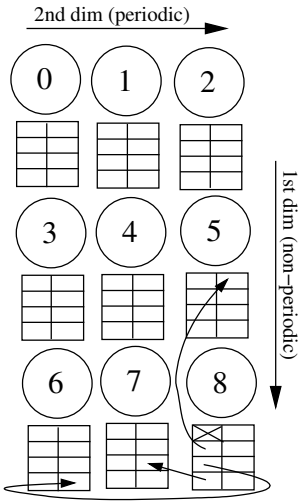


Figure 3: MPI_Neighbor_exchange, ilustrating the communication originating at node 8 for a non-periodic and a periodic dimension. The crossed buffer will be ignored by the collective (but has to be allocated)

on a single buffer with *count* elements. The dimension is selected by the direction parameter $(0..ndims-1)$ and the displacement is selected by disp.

int MPI_Cart_shift_xchg(**void** ∗sbuf, **int** scount, MPI_Datatype stype,
void ∗rbuf, **int** rcount, MPI_Datatype rtype, **int** direction,
int disp, MPI_Comm comm)
MPI_CART_SHIFT_XCHG(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
RECVTYPE, DIRECTION, DISP, COMM)
<type> SENDBUF(∗), RECVBUF(∗)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, DIRECTION, DISP, COMM

void Cartcomm::Shift_xchg(**const void**∗ sbuf, **const int** scount, **const** Datatype& stype,
const void ∗rbuf, **const int** rcount, **const** Datatype& rtype, **const int** direction,
const int disp)

The communication performed is the same is if the communicator would be queried with MPI_Cart_shift and the appropriate sends and receives are issued. However, the collective operation allows for careful message scheduling and simplify the user's task. If provided with a communicator that does not have a cartesian topology, the call returns MPI_ERR_TOPOLOGY.

Application example: A Cartesian shift operation simplifies the parallel implementation of Cannon's matrix multiplication [2] and related algorithms.

Cannon's algorithm perfoms the multiplication of two matrices A and B by subdividing each matrix in blocks A_{ij}, B_{ij} with $i, j = 1, \ldots, M$ and assigning each block to one of $P = M^2$ nodes, i. e. the matrices are mapped onto a $M \times M$ cartesian grid. To calculate the matrix product AB each node first computes the subproduct of its A_{ij}, B_{ij}, before rotating the rows of A upwards and the columns of B leftwards along the respective cartesian dimensions. The

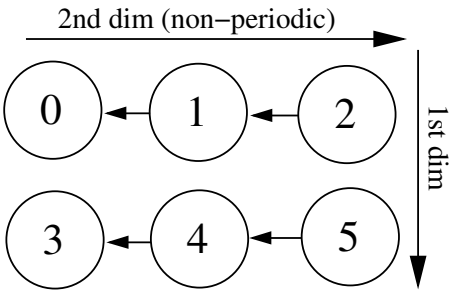


Figure 4: MPI_Cart_shift_xchg on a 2-dimensional communicator in the second dimension direction -1.

next subproduct can be calculated and added to the previous result.

2.4 New C++ Class

We propose to add a new C++ class, Topocomm, which contains functionality available to communicators with any topology. This includes the neighbor query functions and the nearest-neighbor exchange collectives. We have added the Topocomm class and made both Graphcomm and Cartcomm derive from this new class.

```
namespace MPI {
  class Comm {...};
  class Intracomm : public Comm {...};
  class Topocomm : public Intracomm {...};
  class Graphcomm : public Topocomm {...};
  class Cartcomm : public Topocomm {...};
  class Intercomm : public Comm {...};
  // ...
}
```

References

[1] A. Castro, H. Appel, M. Oliveira, C. A. Rozzi, X. Andrade, F. Lorenzen, M. A. L. Marques, E. K. U. Groß, A. Rubio. *phys. stat. sol (b)*, 243(11):2465–2488, 2006.

[2] L. E. Cannon. PhD thesis, Montana State Univ., 1969.

[3] T. Hoefler, A. Lumsdaine and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI *In proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*