Sparse Collective Operations for MPI

Torsten Hoefler
Open Systems Lab
Indiana University
Bloomington, IN, 47405
Email: htor@cs.indiana.edu

Jesper Larsson Träff
NEC Laboratories Europe, NEC Europe Ltd.
Rathausallee 10
D-53225 Sankt Augustin, Germany
Email: traff@it.neclab.eu

***Abstract*—We discuss issues in designing *sparse (nearest neighbor) collective operations* for communication and reduction operations in small neighborhoods for the *Message Passing Interface* (MPI). We propose three such operations, namely a sparse gather operation, a sparse all-to-all, and a sparse reduction operation in both regular and irregular (vector) variants. By two simple experiments we show a) that a collective handle for message scheduling and communication optimization is necessary for any such interface, b) that the possibly different amount of communication between neighbors need to be taken into account by the optimization, and c) illustrate the improvements that are possible by schedules that posses global information compared to implementations that can rely on only local information. We discuss different forms the interface and optimization handles could take. The paper is inspired by current discussion in the MPI Forum.**

I. Introduction

The Message Passing Interface Standard (MPI) [1] provides different ways to realize communication patterns between processes in a communication universe (communicator). One way is to implement each pattern with individual send/receive operations between pairs of processes. This can be tedious and error-prone because the user has the full responsibility of optimizing the communication pattern to particular network topologies, ensuring correctness, and preventing deadlocks. The use of such point-to-point schemes therefore often results in suboptimal algorithms for common communication patterns (such as a linear reduction instead of a tree-based reduction).

To alleviate these drawbacks and support parallel programming at a higher, more portable level, MPI offers a set of (16) collective operations. These embody common patterns and operations such as broadcast, all-to-all personalized exchange, scatter/gather and several parallel data reduction operations. Most of these operations come in both *regular* and *irregular* (vector) variants, the latter enabling communication of different amounts of data between processes in the collective pattern. The collective operations all require participation of all processes in the given communicator.

This set of predefined, *dense* communication patterns covers a wide range of practical applications. However, parallel scientific applications often communicate in a localized neighborhood, e.g., with their neighbors in a regular or irregular mesh. In MPI, such *sparse* communication with only a limited number of processes is currently not supported by collective operations, and must instead be implemented by the application (or by application-specific libraries such as PETSc [2] or ScaLAPACK [3]) by means of point-to-point communication operations. However, the operations often have enough common structure that could conveniently be captured by a set of additional *sparse collective operations*. Some sparse collective operations, e.g., sparse, personalized all-to-all exchange where all processes exchange data with each of their neighbors, can be expressed by existing MPI collectives, e.g., a dense, irregular (vector), personalized all-to-all exchange, in which no data are exchanged between processes that are not neighbors (in the neighborhood of the application). This has several disadvantages. First, it is not a natural abstraction for applications with a very small local neighborhood (e.g., 9-neighbor stencil in a 2-dimensional mesh) compared to the total number of processes. This will tend to detract users from collective operations that might otherwise have given them a performance or portability benefit. A second drawback is that such a solution is not scalable, e.g., the major part of the arguments to the MPI_Alltoallv call will be zeroes that will nevertheless have to be read an interpreted by the underlying MPI implementation. Furthermore, because MPI_Alltoallv is one of the (most) general collective operations, the optimizations (if any!) that are applied by the MPI library may be weaker (or more time-consuming) than those that would be possible *given* the knowledge that communication takes place only in small neighborhoods. Finally, as we will see, not all natural, sparse collective operations are easily and efficiently expressible with the existing MPI collectives.

To address these issues, and in order to reflect current application needs, we define three new sparse collective operations, in both regular and irregular (vector) variants. We discuss several design possibilities to make the proposal compatible with the current MPI standard. More concretely, we propose a sparse gather operation, a sparse personalized all-to-all operation, and a sparse reduction operation (the latter being an example of an operation that is not efficiently expressible in terms of existing MPI collectives), and we discuss alternatives for specifying the neighborhoods of processes. By means of two simple experiments, we first show that scheduling of communication can indeed have an effect on the overall performance (time to completion) of sparse collectives, and thereby argue that a global, collective handle is required for performing such scheduling optimizations. Second, we show that for the irregular sparse collectives, the actual amount of

data communication between different neighbors needs to be taken into account in this optimization. These observations set some natural constraints on the design of the interface, which we elaborate on in this paper.

For the discussion of optimization/schedules we make the simplifying, sometimes problematic assumption that processes arrive at the sparse collective more or less at the same time. If the processes arrive in a very unbalanced fashion other algorithms/approaches than discussed here must be employed. A preliminary discussion of such algorithms for the (dense) MPI_Bcast collective can be found in [4]. Another alternative is to define the (sparse) collectives to be *nonblocking*, which is touched briefly upon below.

A. Background

Many algorithms, for example weather prediction [5] and computational fluid dynamics [6], naturally exhibit sparse communications among neighboring elements or zones. Additionally, many practical domain decompositions [7] result in nearest neighbor communication patterns. Such sparse communication patterns are very important for applications such as Qbox [8], TDDFT/Octopus [9], QCD codes [10], POP [11] and many more. Also libraries such as PETSc [12] or ScaLAPACK [3], that use MPI to implement several numerical methods and solvers, often exhibit sparse communication patterns.

Efficient implementation of sparse communication patterns is most important for large-scale applications. Most large-scale parallel computers support only sparse communication efficiently. For example, the 3d-torus Blue Gene/L [13] and Cray XT 4 communication networks supports direct communication with six neighboring nodes most efficiently. The QCDOC supercomputer [14] also optimizes for nearest neighbor communication. For such architectures, which exhibit a very low effective bisection bandwidth, the mapping from logical communication paths to physical channels is most important. It is also rather clear that future massively parallel systems can not support high bisection bandwidth for every communication pattern [15]. Thus, the use of sparse communication techniques and proper mapping will gain more importance in the near future.

We assume that many such algorithms and applications would benefit from a higher-level description of their communication operations. A previous study with Octopus [16] shows that the implementation complexity of the application can be reduced by applying collective semantics to sparse communication patterns.

II. Sparse collective operations

We propose three different collective operations that are defined on process neighborhoods. Process neighborhoods are attached to the communicator on which the collectives are called. How neighborhoods can get associated with communicators will be discussed in Section IV. A *process neighborhood* is a set of local neighborhoods that for each process i consists of a list of k *target processes* $[t_0, t_1, \ldots, t_{k-1}]$ and a list of ℓ not necessarily different *source processes* $[s_0, s_1, \ldots, s_{\ell-1}]$. In each of the sparse collective operations, process i sends some data to its target processes, and receives some data from its source processes. Note that a process can be both a target and a source neighbor of itself. It is also not required that processes are unique in the source and target lists as long as all send/receive pairs match (see Section II-D).

```
MPI_Neighbor_gather(sendbuf,sendcount,sendtype,
                    recvbuf,recvcount,recvtype,
                    comm)
```

Listing 1. Sparse gather collective function description.

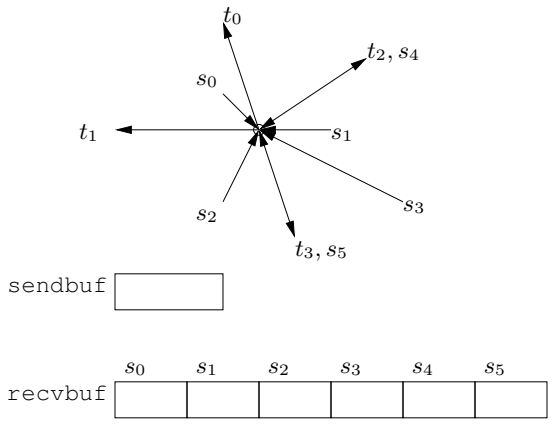

Fig. 1. The sparse gather operation on a local neighborhood with 6 source and 4 target processes. As indicated the size of the block sent to the target processes *may* be different from the size of the blocks received from the source processes. In the regular MPI_Neighbor_gather operation the received blocks all have the same size and type signature.

As for the standard MPI collectives, data to be sent and received are stored in send and receive buffers relative to some start address. The order of data is given by the order of the neighbors in the target and source lists. Data are described by a datatype and a repetition count. The usual semantic constraints shall apply, i.e., the datatype signature between neighbors where data are sent and/or received must match.

a) Gather: In the sparse gather operation each process i receives a block of data from each of its source processes $[s_0, s_1, \ldots, s_{\ell-1}]$, and stores the block in that order relative to the `recvbuf` address. All received blocks are of the *same* size. Process i sends the *same* block of data from its `sendbuf` to its target processes $[t_0, t_1, \ldots, t_{k-1}]$. This is depicted in Figure 1. The function description is given in Listing 1. Note that the size of the data sent must equal the size of the data received by each of the target processes. If a process is source and target of itself, this implies that the size of the data sent and received must be equal. More generally, all processes lying on a cycle in the process neighborhood must send and receive the same amount of data.

b) All-to-all: The sparse all-to-all personalized exchange operation extends the sparse gather operation in that personalized data is sent to each target process. Thus, instead of just a single block the `sendbuf` holds k blocks of data (of the same size). The function description is shown in Listing 2.

```
MPI_Neighbor_alltoall(sendbuf,sendcount,sendtype,
                      recvbuf,recvcount,recvtype,
                      comm)
```

Listing 2. Sparse all-to-all collective function description.

```
MPI_Neighbor_reduce(sendbuf,sendcount,sendtype,
                    recvbuf,recvcount,recvtype,
                    op,comm)
```

Listing 3. Sparse reduction collective function description.

c) Reduce: In the sparse reduction operation each process gathers data from each of its source processes, and reduces these data into a single block using an MPI binary reduction operator (either pre- or user-defined). Each process contributes the *same* data block to each of its target processes. A description of this function is shown in Listing 3.

The sparse reduction operation cannot easily be expressed in terms of existing MPI collectives. The current reduction collectives all perform a reduction on contributions from *all* processes of the communicator, whereas the sparse operation produces a value for each process computed from contributions from the local neighborhoods. An alternative is to revert to a general MPI_Alltoallv operation, followed by local, sequential reductions by the processes of the data from their neighborhoods.

A. Irregular (vector) variants

For user convenience, in analogy with the existing MPI collectives, we propose an *irregular* (vector) variant for each of the three sparse collectives.

For the all-to-all operation this is straightforward. Each process can send a different amount of data to each of its target processes, now addressed by a `senddispls` displacement relative to the `sendbuf`, and receive a different amount of data from each of its source processes. We also propose an MPI_Alltoallw-like operation where each block to be sent and received may have its own datatype. As will be seen in Section II-D this is useful when the blocks have different memory layouts.

The irregular, sparse gather operation allows that a block of different size can be received from each source process, whereas the same block is still sent to each target process. Likewise, the irregular, sparse reduction operation relaxes the constraint that the same block (of the same size) is sent for reduction to the target processes. Instead, an individual block can be specified for each neighbor. The proposed function interfaces are shown in Listing 4.

```
MPI_Neighbor_gatherv(sendbuf,sendcount,sendtype,
                     recvbuf,
                     recvcounts,recvdispls,recvtype,
                     comm)
MPI_Neighbor_alltoallv(sendbuf,
                       sendcounts,senddispls,sendtype,
                       recvbuf,
                       recvcounts,recvdispls,recvtype,
                       comm)
MPI_Neighbor_alltoallw(sendbuf,
                       sendcounts,senddispls,sendtypes,
                       recvbuf,
                       recvcounts,recvdispls,recvtypes,
                       comm)
MPI_Neighbor_reducev(sendbuf,
                     sendcounts,senddispls,sendtype,
                     recvbuf,recvcount,recvtype,
                     op,comm)
```

Listing 4. Function descriptions for the irregular (vector), sparse collectives.

B. Semantics

The proposed operations are collective and need involvement of the processes of the communicator. Whereas the existing MPI collectives require that all processes in the communicator call the collective at the same time (i.e., with no other collective calls in-between), it is possible to define a more relaxed semantics for the sparse collectives. For correctness we will always require that if process j is in the target list of process i (with multiplicity r), then process i is in the source list of process j (with multiplicity r).

For each sparse collective, we propose/require that if process i is calling sparse collective A, then each process $j \in [t_0, t_1, \ldots, t_{k-1}] \cup [s_0, s_1, \ldots, s_{\ell-1}]$ must eventually call A with no other collective call (on the same communicator) in-between. This means that if processes i and j are not connected by a path in the communication neighborhood (see below) their calls to sparse collectives can be independent of each other. From a user's perspective, it might not always be easy to verify that all necessary conditions, e.g., matching block sizes, are met. Thus, it is worth noting that they can be automatically verified (by similar collective operations), and thus incorporated into a verification interface as described in [17], [18].

The relaxed calling conditions can give more flexibility to applications with disconnected neighborhoods. Processes in one neighborhood need not ensure that sparse collective calls follow in the same order or with the same frequency as sparse collective calls in another disconnected neighborhood. The number of iterations in loops with sparse collective calls need thus not be the same in all neighborhoods of the communicator.

We note that this definition excludes optimizations in which processes not actually in the neighborhood of some process helps with routing or computation in the collective operation. It can be shown that this can sometimes reduce the number of communication rounds, e.g., for irregular all-to-all communication [19].

C. Nonblocking variants

Nonblocking versions of the proposed sparse collective operations are possible. This would enable overlap of communication and computation during the communication phase. Such benefits of nonblocking nearest neighbor communication on a graph communicator were investigated in [16]. Another, more wide-ranging possibility that is not discussed further here is to introduce a partial completion function to indicate

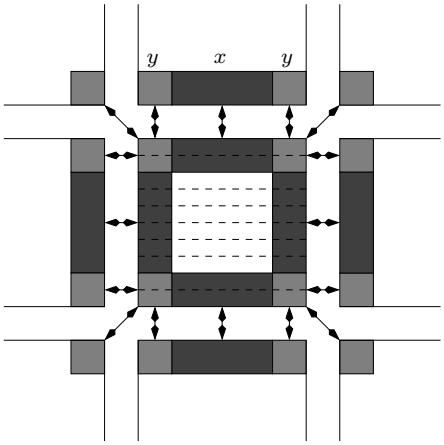

Fig. 2. Logical nearest neighbor communication needed for a typical ghost cell (shaded regions) update operation. The common optimization for eliminating diagonal communication of the y regions by piggy backing on the horizontal/vertical x communication can be done transparently to the user by a good implementation of MPI_Neighbor_alltoallw.

neighbors for which data have already arrived. This would allow finer control of the communication while retaining the higher abstraction level and message scheduling possibilities.

D. An example: ghost cell updates

Applications that use ghost cell regions to introduce more communication slackness update these by operations that are captured by the sparse MPI_Neighbor_alltoallw operation. In Figure 2 each processor in a mesh maintains a border of ghost cells (shaded) that have to be exchanged with its 8 neighbors. This pattern can be expressed as an MPI_Neighbor_alltoallw sparse collective communication operation. Horizontal and vertical x blocks are exchanged with one neighbor only, but may typically have different layout in memory and thus different MPI datatypes. Each of the smaller y blocks is sent to three neighbors (horizontally, vertically and diagonally), thus the MPI_Neighbor_alltoallw operation must allow overlapping displacements.

The common optimization for saving diagonal communication, sending each y block first in the horizontal direction, combining it with the y block from this neighbor, and then sending both blocks in the vertical direction (see e.g., [20]) could be automatically taken care of by a good implementation of MPI_Neighbor_alltoallw.

III. Implementation and performance

We now consider one of the simplest process communication topologies in practical use [21], [22], namely a n-dimensional mesh or torus (we chose two dimensions for simplicity). Each process has a local neighborhood consisting of 4 neighbors (excluding itself) as shown in Figure 3, and sends and receives (the same amount of) data to and from all neighbors. For this simple communication topology we give two implementations of the MPI_Neighbor_alltoall collective.

A straightforward implementation queries the communicator for source and target lists and posts the corresponding

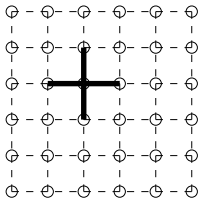

Fig. 3. Two dimensional mesh or torus (wrap-around edges not shown). The local neighborhood of each process indicated with heavy lines consists of the same 4 source and target processes.

```
int MPI_Neighbor_alltoall(sendbuf,sendcount,sendtype,
                          recvbuf,recvcount,recvtype,
                          comm)
{
  // Get source and target lists from communicator
  Get_alltoall_targets(comm,&no_of_targets,target);
  Get_alltoall_sources(comm,&no_of_sources,source);

  for (i=0; i<no_of_sources; i++) {
   MPI_Irecv(recvbuf+i*recvcount*recvextent,...,
             source[i],...,comm,
             request[2*i]);
  }
  for (i=0; i<no_of_targets; i++) {
   MPI_Isend(sendbuf+i*sendcount*sendextent,...,
             target[i],...,
             comm,request[2*i+1]);
  }
  MPI_Waitall(no_of_sources+no_of_targets,request,
              MPI_STATUSES_IGNORE);
}
```

Listing 5. Generic, naive sparse all-to-all implementation.

nonblocking send and receive calls for all source and target processes. This is the way that many applications currently implement the sparse exchange pattern (cf. Section I-A). The implementation is shown in Listing 5.

Since there is no control of the order in which the send and receive operations are started by the underlying MPI implementation, this implementation could suffer from contention by several source processes trying to send data to the same target process at the same time. Furthermore, nonblocking operations typically incur a certain overhead that could grow considerably for larger neighborhoods than in the simple mesh case.

By exploiting the knowledge of the global mesh topology, a possibly more efficient implementation of the sparse all-to-all operation would schedule communication in dimension order, and use combined, blocking send-receive operations. If processes can be assumed to arrive at the collective more or less at the same time, this implementation will not suffer from contention, and can possibly more efficiently take advantage of bidirectional communication capabilities of the underlying network (even in the case where the size of a dimension is odd). This implementation is sketched in Listing 6.

Both implementations send and receive the same amount of data without any combining or fancy rerouting, and therefore have the same message complexity. The first, generic, implementation may suffer from contention, and possibly has

```
int MPI_Neighbor_alltoall(sendbuf,sendcount,sendtype,
                          recvbuf,recvcount,recvtype,
                          comm)
{
  i = 0;
  for (d=0; d<dim; d++) {
    MPI_Cart_shift(comm,d,1,&down,&up);

    MPI_Sendrecv(sendbuf+i*sendcount*sendextent,...,
                 up,...,
                 recvbuf+i*recvcount*recvextent,...,
                 down,...,comm,MPI_STATUS_IGNORE);
    i++; // next pair of neighbors
    MPI_Sendrecv(sendbuf+i*sendcount*sendextent,...,
                 down,...,
                 recvbuf+i*recvcount*recvextent,...,
                 up,...,comm,MPI_STATUS_IGNORE);
    i++; // next pair of neighbors
  }
}
```

Listing 6. Scheduled sparse all-to-all implementation for Cartesian meshes and tori.

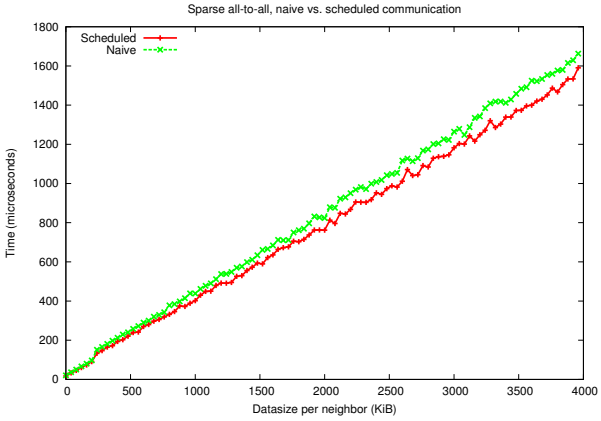

Fig. 4. Running times for the two sparse all-to-all implementations on an NEC SX-8 node with 8 processes.

a higher overhead due to the use of nonblocking operations, but does not rely on global knowledge of the communication topology. The second implementation works strictly for meshes or similar, regular topologies. Posting blocking send-receive operations on a local neighborhood without any knowledge of the order in which the other processes perform the send-receive operations will deadlock in most cases. Only for this reason, global knowledge is necessary for scheduling blocking communication for general sparse collective operations, unless one is satisfied with the generic solution of Listing 5.

We use a simple benchmark for measuring the time to completion of the two alternatives. Results on a single shared-memory node are given in Figure 4 and a results for a hybrid x86 cluster with InfiniBand interconnect are shown in Figure 5. We see that the generic, naive implementation (Listing 5) can be about 10% slower than the more carefully scheduled implementation (Listing 6).

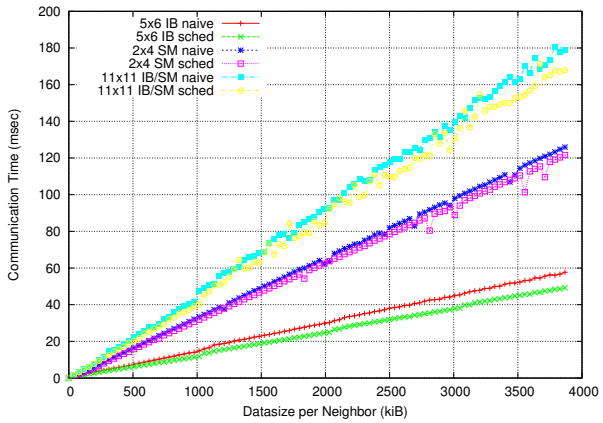

Fig. 5. Running times for the naive and scheduled sparse 2-d exchange on an x86 cluster system with InfiniBand (IB) and Shared Memory (SM). We used 32 nodes that either ran with 1 process per node on a 5x6 process grid or 4 processes per node on a 11x11 grid. Shared memory measurements were performed on a single machine.

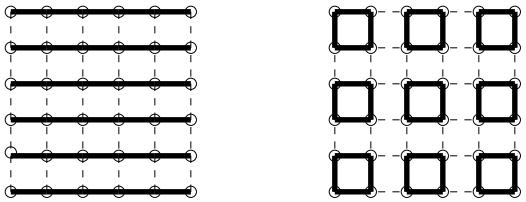

Fig. 6. The two exchange patterns for the irregular all-to-all benchmark. Left: horizontal. Right: circular.

A. *Irregular*

We now use the dimension-scheduled implementation of MPI_Neighbor_alltoall also for solving the *irregular* variant of the problem. For the benchmark, we distinguish between communication *light* and *heavy* edges and analyze the two different exchange patterns shown in Figure 6. We refer to these as *horizontal* (left) and *circular* (right) exchanges, respectively.

The two patterns obviously have the same message complexity (each process sends and receives two heavy and two light messages), but whereas the horizontal pattern has all communication in only one dimension, and can thus complete in only two *heavy* rounds (right-to-left followed by left-to-right), the circular pattern requires four heavy rounds (right-to-left sends or receives on a heavy edge, and likewise left-to-right sends or receives on a heavy edge. Also the two up-down communication rounds are both heavy). It is obvious that a different schedule could easily be used to solve the circular problem also in two heavy rounds.

The results from the benchmark show the expected difference in performance of about a factor two on an NEC SX-8, see Figure 7. Figure 8 shows the same benchmark for an x86 cluster with InfiniBand. We see that the horizontal communication is faster for all homogeneous communication systems, however, when we mixed InfiniBand and shared memory, we see that our scheduling performs worse. This shows that message scheduling strategies perform differently

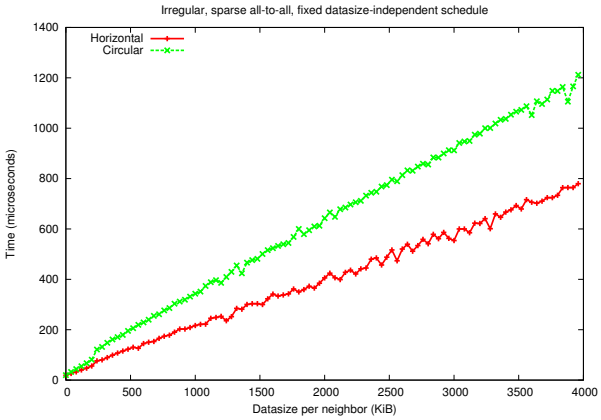

Fig. 7. Running times for horizontal and circular patterns on an NEC SX-8 node with 8 processes.

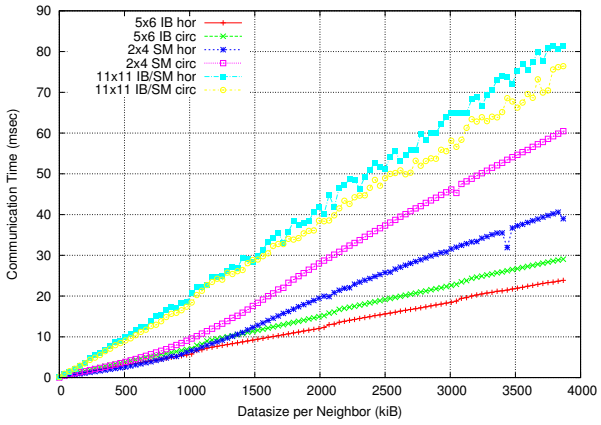

Fig. 8. Communication times for different mapping options for an x86 cluster system with InfiniBand. Processes are either mapped in a 5x6 grid on 32 nodes with 1 process per node using InfiniBand (IB) to communicate or an 11x11 grid on 32 nodes using IB and Shared Memory (SM) with 4 processes per node. The 2x4 example uses only shared memory.

on the investigated systems. Thus, an abstract interface could not only provide higher performance but also performance portability among different systems. This simple example shows that scheduling optimizations that only consider the communication topology but ignore the actual amount of data that is transferred can perform far from optimal.

IV. Specifying Neighborhoods

So far we have assumed that the local neighborhoods, i.e., the lists of source and target processes for each process are somehow associated with the communicator. We now have to discuss how this can be done. It is clear that this requires some collective call in which the processes define their local neighborhoods and communication volumes. Such a handle can then be used to schedule correct, deadlock-free and efficient communication among the neighbors. This collective handle would be responsible for computing and distributing knowledge about the global communication topology to each process, thereby allowing it to make efficient and correct scheduling decisions. Although the handle needs to be collective, this does not necessarily imply that a global communication graph has to be explicitly constructed in full by any one process. Distributed algorithms (e.g., for graph coloring [23]) could be used to obtain enough information for each process to make the correct scheduling and optimization decisions.

```
MPI_Neighbor_alltoall_set(sources,sourceweights,
                          targets,targetweights,
                          info,comm)
```

Listing 7. Collective handle for attaching local neighborhoods to communicator (for MPI_Neighbor_alltoall).

```
MPI_Neighborhood_set(operation,
                     sources,sourceweights,
                     targets,targetweights,
                     info,comm)
```

Listing 8. Combined collective handle for attaching local neighborhoods to communicator.

In the following we discuss two solutions to the problem.

A. Handles taking local information

The straightforward alternative would be to introduce a collective operation in which each process contributes its local neighborhood in the form of source and target lists. These lists would get associated with the communicator used in the call, and can later be queried locally. At this point, schedules and other optimizations can be decided (locally) for the processes. However, since the three sparse collectives described in Section II may use different local neighborhoods in the application, and since the scheduling and other optimization criteria may be different for the three collectives, it seems that a specific handle for each type of collective would be necessary. Thus, either three handles, as shown in Listing 7, would have to be introduced or a composite handle, as in Listing 8, with source and target lists for each collective as the first argument.

The `operation` argument for the second possibility could be a bit-vector of values MPI_NEIGHBOR_GATHER, MPI_NEIGHBOR_ALLTOALL, and MPI_NEIGHBOR_REDUCE that could thus be given in any combination.

For both possibilities an MPI `info` object is introduced to convey information of optimization criteria, process arrival patterns and other information that could be relevant to the MPI implementation. The weight arguments should reflect (approximately) the amount of communication with neighbors, and can be used by the MPI library to find a good schedule for the underlying hardware architecture. Note that this is decoupled from the *actual* communication amounts (given by the send and receive counts and datatype arguments in the actual, sparse collective calls), and it is not required that the two are identical. If a user makes a sparse collective call with completely different communication amounts from what was specified in the neighborhood handle call, suboptimal performance may be expected. Correctness must, however, not be compromised. Thus, communication weights and `info` object are only *hints* to the library as to what can be expected.

In MPI, lists (ordered sets) of processes are represented by *process groups*. The source and target list arguments should thus be given as process groups, although many users consider this construct tedious. An alternative might be simply to supply simple lists of process ranks.

B. Using the MPI virtual topology functionality

The set of all local neighborhoods, that is the source and target lists introduced in Section II, together comprise a directed communication graph. In MPI there is already a mechanism for using such communication graphs to improve communication performance. This is the virtual topology functionality [1, Chapter 7]. It is therefore a natural alternative to use this functionality also for fixing the process neighborhoods for the sparse collective operations.

The Cartesian topology mechanism allows implicit specification of simple meshes, tori (and hypercubes). Each process has neighbors along the dimensions. This neighborhood communication pattern is probably too restricted for many actual applications (that use e.g., 9-neighbor stencils in the 2-dimensional case), and it further limits the sparse collectives to only symmetric exchange patters. For using this, otherwise convenient functionality, extensions that give more flexibility would need to be considered.

The general, graph topology interface gives full flexibility in describing neighborhoods, since communication graphs are not required by the MPI standard [1] to be symmetric. Unfortunately, in the current interface all processes are required to supply the full communication graph. This may not be known, and it would therefore entail global, collective communication by the user to build this knowledge. Furthermore, the existing functionality has room neither for communication edge weights, nor for an `info` object for providing information to the MPI library. This is, however, likely to be extended with new, more flexible functionality in upcoming versions of the MPI standard. In order to get the list of local neighbors that is needed to pack and access data in the communication buffers, additional convenience functionality is needed for the Cartesian topology functionality.

The virtual topology functionality creates a new communicator with possibly a new process to processor mapping that can have been improved for the communication pattern implied by the user supplied communication graph. If used as a handle for sparse collective operations, this would also have the effect of optimizing for these patterns. However, as discussed above, the same neighborhood would be fixed for all three collective operations. Furthermore, any change in neighborhood would force the user to create a new virtual topology communicator, and this might be rather expensive. The solution discussed in the previous section is more flexible in this respect.

In addition to the advantage of reusing existing (but extended) MPI functionality, the major advantage of this solution is that process reordering (for improved point-to-point communication) and optimization of sparse collective patterns is *not* separated. The first alternative suffers from this drawback.

C. Programmability

Both approaches, the virtual topology and the creation of separate handles, are mostly equivalent from a user's perspective. The user has to create a (local) neighborhood at each process and either create a topological communicator or attach it to a neighbor operation. It was already shown in [16] that such abstractly defined collective operations can simplify parallel programs significantly.

D. Scheduling communication and other optimizations

For the regular, sparse all-to-all collective, the experiments showed exchange in dimension-order to be beneficial, and this was an example of a more general scheduling optimization.

For homogeneous, single-ported systems both the regular and the irregular all-to-all problem can be solved by graph edge-coloring. A communication multi-graph is constructed where the number of edges between any pair of nodes is proportional to the communication amount between the corresponding processes. An edge coloring with colors $0 \leq i < c$ where $c - 1$ is the largest color used gives a deadlock-free schedule for the communication in c communication rounds. In round i processes sends and receives on edges with color i. This is a common solution, see e.g., [19], with many variations. We were deliberately vague about how to handle directed graphs and uni- or bidirectional communication capabilities. The general edge-coloring problem is NP-complete, but good approximation schemes exist, so that this is a viable course for finding good schedules. Also, depending on the communication model sometimes the problem to be solved is a bipartite edge coloring problem (or other variant), that can be solved in polynomial time. For non-homogeneous systems, like SMP-clusters, this approach is not likely to produce good results. For such cases, optimizations that combine messages to save on communication start-ups over weak connections are likely to pay off. Such are often implemented by users, e.g., sending the data to a vertical and diagonal neighbor that reside on a process on a different SMP node in one message. Here it is important that the interface definitions of Section II do not preclude such optimizations.

Figure 9 finally illustrates the sensitivity of the communication schedules to process mapping on an SMP system. The benchmark of Figure 6 was used on a 4-node SX-8 system with 8 processes per node. We compared the cases of horizontal heavy edges, vertical heavy edges, and the circular pattern on the `MPI_COMM_WORLD` communicator. With MPI's row major ordering, the horizontal pattern has the most heavy edges crossing nodes, and thus performs the worst, whereas the vertical pattern performs best, with circular in between. We note that process reordering (which is done by NEC's MPI/SX implementation [24]) cannot improve the communication in any of the cases, because the (current) MPI topology functionality does not permit specification of communication amounts between neighbors.

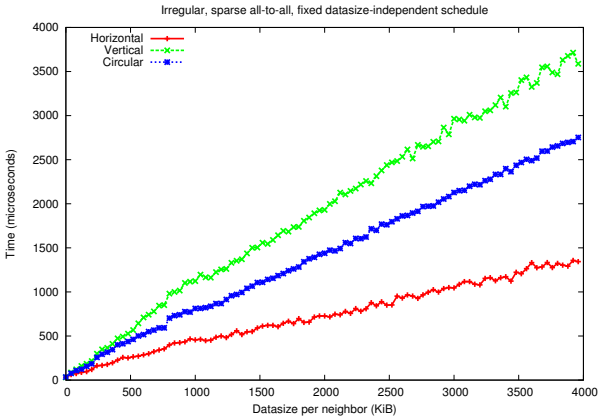

Fig. 9. Performance of horizontal, vertical, and circular patterns on a 4x8 process grid a 4-node NEC SX-8.

V. Conclusions

Sparse collective operations are (together with other collective enhancements) currently under discussion in MPI Forum for future MPI versions. In this paper we proposed three such sparse collective communication and reductions operations, each in a regular and an irregular (vector) variant. We proposed two alternative ways of informing the MPI library of the local neighborhoods of the processes.

Collective operations and collective handles for specifying neighborhoods were designed to capture patterns in actual applications, and should not exclude any of the optimizations that are typically carried out by users, e.g., combination of messages and scheduling of communication. One exception was made: The relaxed collective semantics do not require that all processes in the communicator always take part in a call, and this excludes certain types of rerouting via otherwise idle processes.

By experiments with one of the simplest sparse communication topologies it was shown that messages scheduling is necessary for performance, especially for the irregular operations and that global knowledge is necessary also for correctness (deadlock-freedom). We think that these are the side-constraints that any design for sparse collectives for MPI must obey.

Acknowledgments

The authors thank members of the MPI Forum, in particular the Collectives Working Group for discussions that have influenced this paper. This work was partially supported by a grant from the Lilly Endowment, National Science Foundation grant EIA-0202048 and a gift from the Silicon Valley Community Foundation on behalf of the Cisco Collaborative Research Initiative.

References

[1] MPI Forum, "MPI: A message-passing interface standard. version 2.1," September 4th 2008, www.mpi-forum.org.

[2] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, "PETSc Web page," 2001, http://www.mcs.anl.gov/petsc.

[3] J. Choi, J. J. Dongarra, S. Ostrouchov, A. Petitet, D. W. Walker, and R. Whaley, "Lapack working note 80: The design and implementation of the scalapack lu, qr, and cholesky factorization routines," Knoxville, TN, USA, Tech. Rep., 1994.

[4] P. Patarasuk and X. Yuan, "Efficient MPI_Bcast across different process arrival patterns," in *22nd International Parallel and Distributed Processing Symposium (IPDPS)*, 2008, p. 32.

[5] D. Singh and A. Ganju, "Improvement in nearest neighbour weather forecast model performance while considering the previous day's forecast for drawing forecast for the following day," *Current science (Bangalore)*, vol. 91, no. 12, pp. 1686–1691, 2006.

[6] J. Ferziger and M. Perić, *Computational methods for fluid dynamics*. Springer New York, 2002.

[7] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[8] F. Gygi, "Large-scale first-principles molecular dynamics: moving from terascale to petascale computing," *Journal of Physics Conference Series*, vol. 46, pp. 268–277, Sep. 2006.

[9] A. Castro, H. Appel, M. Oliveira, C. A. Rozzi, X. Andrade, F. Lorenzen, M. A. L. Marques, E. K. U. Gross, and A. Rubio, "Octopus: a tool for the application of time-dependent density functional theory," *PsiK Newsletter*, vol. 73, pp. 145–173, 2006.

[10] A. Krasnitz, "Lattice field theory in a parallel environment," in *PARA '95: Proceedings of the Second International Workshop on Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science*. London, UK: Springer-Verlag, 1996, pp. 352–360.

[11] P. W. Jones, P. H. Worley, Y. Yoshida, I. J. B. White, and J. Levesque, "Practical performance portability in the parallel ocean program (pop): Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 17, no. 10, pp. 1317–1327, 2005.

[12] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds., 1997, pp. 163–202.

[13] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, M. E. G. P. Coteus, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas, "Overview of the blue gene/l system architecture," *IBM Journal of Research and Development*, vol. 49, no. 2, pp. 195–213, 2005.

[14] P. Boyle, C. Jung, and T. Wettig, "The QCDOC supercomputer: hardware, software, and performance," *Arxiv preprint hep-lat/0306023*, 2003.

[15] T. Hoefler, T. Schneider, and A. Lumsdaine, "Multistage Switches are not Crossbars: Effects of Static Routing in High-Performance Networks," in *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Oct. 2008.

[16] T. Hoefler, F. Lorenzen, and A. Lumsdaine, "Sparse Non-Blocking Collectives in Quantum Mechanical Calculations," in *Proceedings of the 15th European PVM/MPI Users' Group Meeting*, ser. LNCS, vol. 5205. Springer, Sep. 2008, pp. 55–63.

[17] J. L. Träff and J. Worringen, "Verifying collective MPI calls," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science, vol. 3241. Springer, 2004, pp. 18–27.

[18] C. Falzone, A. Chan, E. Lusk, and W. Gropp, "Collective error detection for MPI collective operations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 12th European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science, vol. 3666, 2005, pp. 138–147.

[19] P. Sanders and R. Solis-Oba, "How helpers hasten h-relations," *Journal of Algorithms*, vol. 41, no. 1, pp. 86–98, 2001.

[20] B. Palmer and J. Nieplocha, "Efficient algorithms for ghost cell updates on two classes of MPP architectures," in *IASTED International Conference on Parallel and Distributed Computing Systems (PDCS)*, 2002, pp. 192–197.

[21] L. Giraud, R. Guivarch, and J. Stein, "A parallel distributed fast 3D poisson solver for Méso-NH," in *Euro-Par '99: Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, ser. LNCS, vol. 1685. London, UK: Springer-Verlag, 1999, pp. 1431–1434.

[22] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm, "Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations," *Elsevier Journal of Parallel Computing (PARCO)*, vol. 33, no. 9, pp. 624–633, Sep. 2007.

[23] M. V. Marathe, A. Panconesi, and L. D. R. Jr., "An experimental study of a simple, distributed edge-coloring algorithm," *ACM Journal of Experimental Algorithms*, vol. 9, 2004.

[24] J. L. Träff, "Implementing the MPI process topology mechanism," in *Supercomputing*, 2002, http://www.sc-2002.org/paperpdfs/pap.pap122.pdf.