A proposal for persistent, sparse, and non-blocking collectives

May 30, 2008

1 Contributors

This is an open proposal and contributors are listed here (alphabetically):

- Torsten Hoefler
- Andrew Lumsdaine
- Christian Siebert
- Jesper Larsson Traeff

2 Persistent, sparse collectives

A number of issues cannot be addressed well by the interface provided by the standard set of blocking collective operations of MPI. First, the irregular variants of the communication collectives (MPI_Gatherv etc.) are not scalable (to very large systems) because of the lists of counts (and displacements) that have to be supplied, most of which can in many applications be expected to be zero. Second, for these collectives, optimal implementations may require computations of elaborate schedules. Such computations can only be amortized if a number of calls are made using the same schedule, and thus requires a handle for when to perform such precomputation and where to store the resulting schedule. Third, the currently defined collective operations do not provide a non-blocking interface and make it thus practically impossible to overlap communication and computation.

All those issues can be addressed by an interface for *persistent* collective operations. These are defined in this section, and (even if it for some cases makes less sense) there is a persistent counterpart to each of the blocking MPI collectives.

The first issue is handled by providing each operation with an extra group argument, and semantically each operation is carried out only over the processes of the communicator also belonging to the supplied group. In order to allow for optimizations (like routing through intermediate processes not in the group) the calls are collective (see Section 2.1 below) over all processes of the communicator. The operations are defined in analogy with the persistent point-to-point operations, and the initialization calls are per default *local*. The order in which data are received or sent, and handled locally at the processes is determined by the order of the participating processes in group.

The second issue is handled by allowing the initialization calls to be collective. At this point information can be exchanged, schedules computed, and cached for later reuse. Whether such optimization should be performed, with what objective, and whether reuse should be attempted is controlled by the info argument.

The third issue is resolved with non-blocking start and startall functions for persistent requests (as they already exist for persistent point-to-point functions).

Sparse collectives, with efficient support of rapidly changing sparsity patterns is a sometimes desired feature, since it creation of new communicators may for such cases be too expensive (and precomputing a large set of communicators infeasible). Such functionality may be supported through the interface functions provided in this chapter, and can be efficiently supported by implementations through the "hints" provided through an info object. To cope with sparse, collective operations occuring typically in "grid" structured computations, three new collectives (both in regular and irregular variants) are defined in the same vein. To interact better with the topology functionality, an improved interface for creating virtual topologies, and extracting information from topologies, that can be used as input for the new persisten collectives, is needed. This in addition solves many of the known (scalability) problems with the existing topology functionality.

2.1 Syntax and Semantics

MPI_Barrier_init(group,info,comm,request)
IN group
IN info
IN comm
OUT request

Persistent barrier. The call is collective over all processes in the group of comm, but the barrier semantics are guaranteed only for the processes in group. All processes must call with the same group argument or MPI_GROUP_EMPTY .

MPI_Bcast_init(buffer, count, datatype, root, group, info, comm, request)
INOUT buffer
IN count
IN datatype
IN root
IN group
IN info
IN comm
OUT request

Persistent broadcast. Data are broadcast from the root process to the other processes in group. The root process must be a member of group. Same rules as for blocking collectives apply to datatype matching.

MPI_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, group, info, comm, request)
IN sendbuf
IN sendcount
IN sendtype
OUT recvbuf
IN recvcount
IN recvtype
IN root
IN group
IN info
IN comm
OUT request

Persistent gather.

MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, recvdisp, recvtype, root, group, info, comm, request)
IN sendbuf
IN sendcount
IN sendtype
OUT recvbuf
IN recvcounts
IN recvdisp
IN recvtype
IN root

IN group
IN info
IN comm
OUT request

Persistent, irregular gather.

MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, group, info, comm, request)
IN sendbuf
IN sendcount
IN sendtype
OUT recvbuf
IN recvcount
IN recvtype
IN root
IN group
IN info
IN comm
OUT request

Persistent scatter.

MPI_Scatterv_init(sendbuf, sendcounts, senddisp, sendtype, recvbuf, recvcount, recvtype, root, group, info, comm, request)
IN sendbuf
IN sendcounts
IN senddisp
IN sendtype
OUT recvbuf
IN recvcount
IN recvtype
IN root
IN group
IN info
IN comm
OUT request

Persistent, irregular scatter.

MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, group, info, comm, request)
IN sendbuf
IN sendcount

IN sendtype
OUT recvbuf
IN recvcount
IN recvtype
IN group
IN info
IN comm
OUT request

Persistent allgather.

MPI_Allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, recvdisp, recvtype, group, info, comm, request)
IN sendbuf
IN sendcount
IN sendtype
OUT recvbuf
IN recvcounts
IN recvdisp
IN recvtype
IN group
IN info
IN comm
OUT request

Persistent, irregular allgather.

MPI_Alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, group, info, comm, request)
IN sendbuf
IN sendcount
IN sendtype
OUT recvbuf
IN recvcount
IN recvtype
IN group
IN info
IN comm
OUT request

Persistent alltoall.

MPI_Alltoallv_init(sendbuf, sendcounts, senddisp, sendtype, recvbuf, recvcounts, recvdisp, recvtype, root, group, info, comm, request)

IN sendbuf
IN sendcounts
IN senddisp
IN sendtype
OUT recvbuf
IN recvcounts
IN recvdisp
IN recvtype
IN group
IN info
IN comm
OUT request

Persistent, irregular alltoall.

MPI_Alltoallw_init(sendbuf, sendcounts, senddisp, sendtypes, recvbuf, recvcounts, recvdisp, recvtypes, group, info, comm, request)
IN sendbuf
IN sendcounts
IN senddisp
IN sendtypes
OUT recvbuf
IN recvcounts
IN recvdisp
IN recvtypes
IN group
IN info
IN comm
OUT request

Persistent, irregular alltoall (with possibly different types).

MPI_Reduce_init(sendbuf, recvbuf, recvcount, recvtype, root, group, info, comm, request)
IN sendbuf
OUT recvbuf
IN recvcount
IN recvtype
IN group
IN info
IN comm
OUT request

Persistent reduce.

MPI_Allreduce_init(sendbuf, recvbuf, recvcount, recvtype, group, info, comm, request)
IN sendbuf
OUT recvbuf
IN recvcount
IN recvtype
IN group
IN info
IN comm
OUT request

Persistent allreduce.

MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, recvtype, group, info, comm, request)
IN sendbuf
OUT recvbuf
IN recvcounts
IN recvtype
IN group
IN info
IN comm
OUT request

Persistent reduce-scatter.

MPI_Scan_init(sendbuf, recvbuf, recvcount, recvtype, group, info, comm, request)
IN sendbuf
OUT recvbuf
IN recvcount
IN recvtype
IN group
IN info
IN comm
OUT request

Persistent, inclusive scan.

MPI_Exscan_init(sendbuf, recvbuf, recvcount, recvtype, group, info, comm, request)
IN sendbuf
OUT recvbuf
IN recvcount
IN recvtype
IN group
IN info
IN comm

OUT request

Persistent, exclusive scan.

MPI_Exchange_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, sendgroup, recvgroup, info, comm, request)
IN sendbuf
IN sendcount
IN sendtype
OUT recvbuf
IN recvcount
IN recvtype
IN sendgroup
IN recvgroup
IN info
IN comm
OUT request

Persistent exchange. Each process sends data to the processes in its sendgroup (in the order of the processes in that group), and receives data from the processes in its recvgroup (in the order of the processes in that group). Different processes may supply different groups, but if process j is in the sendgroup of process i, then i must likewise be in the recvgroup of process j. The result of the operation is as if, for each process i an MPI_Isend(sendbuf+j*sendcount*extent(sendtype),sendcount,sendtype,rank(j),...) for each j in sendgroup where rank(j) is the rank of j in comm together with corresponding MPI_Irecv(recvbuf+j*recvcount*extent(recvbuf),recvcount,recvtype,rank(j),...) for each j in recvgroup. Note that different processes will typically give different values for sendgroup and recvgroup. In the limit where both sendgroup and recvgroup are the same as the group of comm the function is equivalent to MPI_Alltoall.

MPI_Exchangev_init(sendbuf, sendcounts, senddisp, sendtype, recvbuf, recvcounts, recvdisp, recvtype, sendgroup, recvgroup, info, comm, request)
IN sendbuf
IN sendcounts
IN senddisp
IN sendtype
OUT recvbuf
IN recvcounts
IN recvdisp
IN recvtype
IN sendgroup
IN recvgroup
IN info

IN comm
OUT request

Irregular, persistent neighbor exchange.

MPI_Neighbor_Reduce_init(sendbuf, recvbuf, count, datatype, sendgroup, recvgroup, info, comm, request)
IN sendbuf
OUT recvbuf
IN recvcount
IN recvtype
IN sendgroup
IN recvgroup
IN info
IN comm
OUT request

Persistent neighbor reduction. Each process performs a reduction over data supplied by the processes in the recvgroup. It contributes the data (stored in sendbuf+j*count*extent(datatype), count, datatype) to all processes in sendgroup (which may or may not include itself). All processes in the union over all sendgroup and recvgroup must supply data of the same signature.

MPI_Neighbor_Reducev_init(sendbuf, sendcounts, senddisp, sendtype, recvbuf, count, datatype, sendgroup, recvgroup, info, comm, request)
IN sendbuf
IN sendcounts
IN senddisp
IN sendtype
OUT recvbuf
IN recvcount
IN recvtype
IN sendgroup
IN recvgroup
IN info
IN comm
OUT request

MPI_Neighbor_Bcast_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, sendgroup, recvgroup, info, comm, request)
IN sendbuf
IN sendcount
IN sendtype

OUT recvbuf
IN recvcount
IN recvtype
IN sendgroup
IN recvgroup
IN info
IN comm
OUT request

Persistent neighbor broadcast (or allgather). Each process performs a broadcast of data stored in sendbuf to the processes in sendgroup. It receives data from the neighbors in recvgroup. All processes in the union over all sendgroup and recvgroup must supply data of the same signature.

MPI_Neighbor_Bcastv_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, sendgroup, recvgroup, info, comm, request)
IN sendbuf
IN sendcounts
IN senddisp
IN sendtype
OUT recvbuf
IN recvcounts
IN recvdisp
IN recvtype
IN sendgroup
IN recvgroup
IN info
IN comm
OUT request

2.2 Semantic issues

The persistent initalization calls all return a request argument which is later used to start the collective operation. The operation is completed by a wait or test call. An MPI_Start call of a persistent request is *collective* in the sense that eventually all other processes in the group of the communicator over which the persistent call was initialized *must* perform a start call for the same opertation. [More explanation needed]

Persistent collective operations do *not* match the blocking collectives.

Examples:

```
proc 1                        proc 2
MPI_Gather_init(&req1)        MPI_Gather_init(&req2)
...

MPI_Start(&req1)              MPI_Start(&req2)
...                           MPI_Wait(&req2)
MPI_Wait(&req1)

Legal!

proc 1                        proc 2
MPI_Gather_init(&req[0])      MPI_Gather_init(&req[1])
MPI_Scatter_init(&req[1])     MPI_Scatter_init(&req[0])

MPI_Start(&req[0])            MPI_Start(&req[1])
MPI_Start(&req[1])            MPI_Start(&req[0])
...                           MPI_Wait(&req[1])
MPI_Waitall(2,req)            ...
                              MPI_Wait(&req[0])

Legal!

proc 1                        proc 2
MPI_Gather_init(&req[0])      MPI_Gather_init(&req[1])
MPI_Scatter_init(&req[1])     MPI_Scatter_init(&req[0])

MPI_Start(&req[0])            MPI_Start(&req[0])
MPI_Start(&req[1])            MPI_Start(&req[1])
...                           MPI_Wait(&req[1])
MPI_Waitall(2,req)            ...
                              MPI_Wait(&req[0])

Illegal!
```

Startall: Request matching with similar tags and sources is defined like in the point-to-point case (undefined?).

Cancelation is not allowed, and a call to MPI_Cancel with a persistent collective request is illegal.

2.3 Hints

Provided through the info argument.

The following information is predifined:

enforce make the initialization call collective and enforce optimization of schedule...

nonblocking optimized for non-blocking/overlap behavior

blocking Blocking behavior (at wait call) expected, no optimization for overlap

reuse some arguments of this persistent operation will be reused by a later persistent init (forward hint to cache information and algorithm).

previous try to reuse arguments from a previous persistent init operation (backward hint to look in cache)

2.4 Efficiency

This functionality gives more flexibility to the application programmer, and may allow implementations to allow for more overlap (non-blocking), non-balanced applications, applications with localized, rapidly changing collective patters, and can therefore not be expected to perform as efficiently as the blocking collective operations.

3 New topology functionality

Achieves several things: permits for reordering, permits for precomputation of routing tables, allocation of queue-pairs... Therefore the topology information is not necessarily identical to the neighborhood later used in a collective exchange operations, and the two concerns are kept separate.

MPI_Cart_create()

MPI_Cart_neighbor_group(selected_dims,distance,comm,group)
IN selected_dims
IN distance (nonnegative integer)
IN comm
OUT group

Returns the group of neighbors of the selected_dims (1 if dimension should be included, 0 if not) that are distance hops away from the calling process. The call is local. The array dims must have the size of the Cartesian topology associated with comm. The call is erroneous if comm is not a Cartesian communicator. The order in which the neighbors are returned is not fixed

Example:

MPI_Graph_create(comm,outgroup,,reorder,info,graphcomm)
IN comm
IN outgroup
IN reorder (integer)
IN info
OUT graphcomm

outgroup is the (ordered) set of neighbors. reorder determines whether the calling process may be reordered (if 0 the process should no be reordered). Values of info: "latency" (optimize for latency, count can be taken to roughly bound the number of communication calls), "bandwidth" (optimize for bandwidth, count can be taken to bound the total data volume), "maxcut" (if partitioning is used, minmize the maximum cut...), "totalcut" (...)

MPI_Graph_neighbor_groups(graphcomm,outgroup,ingroup)
IN graphcomm
OUT outgroup
OUT ingroup

Note that the order of processes in the outgroup returned MPI_Graph_neighbor_groups need not be the same as the order supplied in the MPI_Graph_create call.