PERSISTENCE WORKING GROUP DOCUMENT
UPDATE JUNE 3, 2014

A. SKJELLUM, COORDINATOR (skjellum@auburn.edu);
P. V. BANGALORE, CO-COORDINATOR (puri@uab.edu)

PURPOSE: MAKE POINT-TO-POINT AND COLLECTIVE OPERATIONS WITH REUSE FASTER THAN POSSIBLE IN MPI-1-3.

HISTORY: WE'VE BEEN AT THIS SINCE MPI-3 … RAMPING UP SO WE GET ACTION IN MPI-4.

I. POINT-TO-POINT.

BACKGROUND FUNCTIONALITY USED / NOTATION:

int MPI_Send_init(void *sbuf, int scount, MPI_Datatype sdatatype, int dest, int tag, MPI_Comm comm, MPI_Request *sreq1) [called in process with rank in comm source]

int MPI_Recv_init(void *rbuf, int rcount, MPI_Datatype rdatatype, int source, int tag, MPI_Comm comm, MPI_Request *rreq1) [called in process with rank in comm dest]

Provide examples of how to create the initial send/receive requests used in the following proposals. Here, the communicator *comm* is the same intra-communicator in both calls; should also work fine with inter-communicators.

The requests sreq1, and rreq1 are normal MPI-1 style send/receive persistent requests.

NEW (SINCE JUNE 2013)

1) These standard APIs are the same as before, but we OVERLOAD some of the meanings.
2) These form the master, or persistent requests. The per-message requests will have specific ways they get formed and started. The above syntax is STANDARD MPI. We will use auxiliary calls to help achieve the added behaviors. The goal is to minimize the amount of new API.
3) Tag space is overloaded for persistent, channelized operations. We use the tag space as specified below to identify slack and other properties.
4) Wildcard receives on source are not valid for this. Wildcard receives on tag in the tag range for messages **may** be supported, depending on the channel types.
5) The communicator is not necessarily a vanilla communicator. We use differentiated service on the communicator to get the added restrictions and with it the added performance on operations.
6) Binding happens because of the communicator semantics, not through explicit bind.

NOTE 1. Using point-to-point channels.

PROPOSAL 1 from June 2013 and before is eliminated.

Standard MPI_Start() and MPI_Wait() functions are used. Matching is restricted through the properties of the specific communicator.

Slack is achieved by creating multiple tags. The specialized communicator creation specifies the maximum tags, and their range for high performance. MPI_Send_init() and MPI_Recv_init() outside those tag ranges are erroneous.

Slack is achieved by using multiple persistent requests.

PROPOSAL 2. Creating, Managing, Using, Freeing persistent collective operations communication.

NO CHANGE SINCE JUNE 2013.

We still want to have persistent collective operations, simply because they inherently improve the ability to let MPI know to allocate resources statically, and optionally to do expensive decision making one time.

Rule for creating a persistent collective communication:

Take a non-blocking collective communication name, and augment it with persistent naming, such as: MPI_IBCAST -> MPI_BCAST_INIT

Generically, the operation MPI_COLLECTIVE_INIT(original_blocking_arguments, request, info) derived from the respective MPI_ICOLLECTIVE(*original_blocking_arguments, nb_request)* augments the meaning of nb_request. nb_request is now persistent. The info argument allows users to give hints to implementers about how hard to work in doing this instantiation.

Properties of the MPI_COLLECTIVE_INIT() operations:

a) The operation is collective over the communicator defining the collective operation
b) Normal communicator ordering requirements for the operation apply
c) No user data is passed by the init()
d) The request created is a persistent non-blocking operation request.
e) Use the persistent collective operations with MPI_Start()
f) Wait on their completion with Wait-type operations. Status information is undefined.

Advice to users

Use persistent collective operations to get the fastest collective communication, because the implementation can take the time to optimize the operation defined, since reuse is expected.

End advice to users

Advice to implementers

A set of defined info arguments concerning reuse and how hard to work will be defined, and these info arguments will be implementation independent. [TBD]

End advice to everyone

The way to free non-blocking, persistent collective operations is to use MPI_Request_free().

Advice to forum

There should be no pending collective operation underway in a safe program, but this is a separate discussion as to whether that makes the program unsafe or not.

At this point, we can stop, we have revealed the temporal locality of the collectives fully by allowing persistent creation and use, and these otherwise function like their original counterparts, but with obvious runtime optimization structure revealed to implementations.

However, it is possible to do more with persistent collectives. Binding them into their own communication space, separate from their original communicator is also possible.

For orthogonality/symmetry, seems interesting, but for practicality, we are still thinking about it, especially with regards to the limits on persistent point-to-point channels.

So there is no MPI_COLLECTIVE_INIT_AND_BIND() or bind() proposal yet after the creation of a persistent collective operation.

The meaning of such an operation would chiefly be to separate it from its parent communicator space, but other interpretations are possible. This is just not well developed. Extra optimizations plausible/possible from this are TBD, and we don't have a use case yet.

End advice to forum

PROPOSAL 3. Managing systematically changing base addresses of transfers with persistence for point-to-point and collective.

NEW: This is a use case that can be implemented with a series of related persistent requests and differentiated communicator. No special syntax needed here.

The concept of a "bundle" of related point-to-point persistent channels has been discussed previously in the working group. These involve the same pairs of endpoints, the same tags, and memory that is somehow algorithmically related:

Here is a plausible set of send-side transfers for a 5-slot circular buffer:

```
(SAddr   + 0*sizeof(MPI_Double),  MPI_Double,   1024)    : Sreq[0]
(SAddr+1024*sizeof(MPI_Double),  MPI_Double,   1024)    : Sreq[1]
(SAddr+2048*sizeof(MPI_Double),  MPI_Double,   1024)    : Sreq[2]
(SAddr+3072*sizeof(MPI_Double),  MPI_Double,   1024)    : Sreq[3]
(SAddr+4096*sizeof(MPI_Double),  MPI_Double,   1024)    : Sreq[4]
(SAddr+     0*sizeof(MPI_Double),  MPI_Double,   1024)    : Sreq[0] …
```

And here is a corresponding receive side circular buffer:

```
(RAddr   + 0*sizeof(MPI_Double),  MPI_Double,   1024)    : Rreq[0]
(RAddr+1024*sizeof(MPI_Double),  MPI_Double,   1024)    : Rreq[1]
(RAddr+2048*sizeof(MPI_Double),  MPI_Double,   1024)    : Rreq[2]
(RAddr+3072*sizeof(MPI_Double),  MPI_Double,   1024)    : Rreq[3]
(RAddr+4096*sizeof(MPI_Double),  MPI_Double,   1024)    : Rreq[4]
(RAddr+     0*sizeof(MPI_Double),  MPI_Double,   1024)    : Rreq[0] …
```

In certain low-level implementations of RMA write/put, you could easily have semantics that buffers in the DMA requests auto increment, modulo a limit (in the example above that would be a count of 5*1024). This type of state machine does not exist nor is needed at the MPI level with late binding, but it is needed with the early binding since all arguments are *frozen* with point-to-point Send_init().

The way to implement this is to create 5 channel-pairs between the two end points, each with a unique memory base. That means a series of Send_init() and Recv_init()s on each end.

Everything that follows is therefore to make programming easier and more reliable, but we are also looking for places to eliminate up-calls. In particular, if a low-level DMA driver supports such auto-incrementing addresses modulo a limit, then further optimization of transfers would be possible. To create such a scenario, a multi-slack channel will be needed, however, as the point is to have multiple outstanding overlapping transfers in principle. We don't address that now; we allow the user to specify which requests he/she wants, round robin, rather than trying to push slack into new syntax.

Proposal 4. Bundling Channels [WAS INCOMPLETE]

NEW FOR JUNE 2013: Deleted. Not needed.

Proposal 5. Differentiated Service per Communicator

NEW. We build all the properties we want by defining communicators with restricted properties in return for added performance.

We use the INFO arguments with MPI_COMM_DUP() wherever possible to reduce the need for extra syntax in the standard.

1. N-slack channel FIFO point-to-point communicators.

MPI_CHANNEL_MODE is specified in MPI_Comm_dup_differentiated() info arguments. [LETS DISCUSS HOW TO ANNOTATE]; DUP plus info args.

MPI_Comm_dup_differentiated(MPI_Comm in, MPI_Comm *out, MPI_Info *info);

MPI_CHANNEL_SLACK is specified (if omitted, slack is 1).
MPI_CHANNEL_MINTAG is specified (if omitted, only valid tag is 0)
Note: MPI_CHANNEL_MAXTAG is not needed, MINTAG+SLACK-1 = MAXTAG.
MPI_CHANNEL_ALLOW_WILDCARD_TAG – specifies that there is a more general receive – otherwise tags must match at both ends

In MPI_CHANNEL_MODE there is FIFO-only ordering, no wildcard receives on source.

2. Topologies other than virtual all-to-all

MPI_TOPOLOGY_SPARSE is specified in MPI_Comm_dup()
MPI_TOPOLOGY_GROUP is specified as a reference to a group of the input comm

Only communications between the given rank and those specified in the group are guaranteed on the output communicator. It will be different in each rank of input communicator in principle.

May be used in conjunction with MPI_CHANNEL_MODE

3. Limited or no collective operations; limited or no point-to-point

MPI_COLLECTIVE_NONE
MPI_POINT_TO_POINT_NONE

Cannot both be specified.

Limited types of collective
MPI_COLLECTIVE_ONLY_SPECIFIED – says only support those requested
MPI_COLLECTIVE_BCAST – support BCAST
MPI_COLLECTIVE_REDUCE – support REDUCE
Etc.

Limited types of point-to-point
MPI_POINT_TO_POINT_READ_ONLY (all F-mode)
MPI_POINT_TO_POINT_SYNCH_ONLY (all S-mode)
MPI_POINT_TO_POINT_NON_BLOCKING_ONLY (no non-I-mode)
MPI_POINT_TO_POINT_BLOCKING_ONLY (no I-mode)
MPI_POINT_TO_POINT_NO_BUFFERED (no B-mode)

Other types are possible too – to be studied.

1) Allowing sparse communicators as part of fault tolerance approaches
2) Utilizing sparse-to-sparse communicators with inter-communication options.
3) Providing finer-grain control of exactly what operations are available on a communicator – treat the communicator as an object with methods, and allow for specific subsets on a given communicator (subsetting) without subsetting the standard itself.

Proposal 6. Differentiated Service for MPI_COMM_WORLD

With a single additional function (one per kind of MPI_Init(), we can greatly update MPI to get away from the "all to all virtual fabric" even in the initial communicator. This would also support Fault tolerant explorations, such as discussed with "Reinit" and elsewhere.

Simply, MPI_Init_differentiated(int *argv, char ***argv, MPI_Info *info);
[one version of this needed for all the threaded options too, but this conveys the ideas]. Then you can apply the above Info arguments mentioned for MPI_Comm_dup even on the original communicator. What this would allow is to create non-all-to-all worlds, and reduce the set of functions that have to operate on MPI_COMM_WORLD. Reasons for doing this

1) scalability
2) fault tolerance
3) Performance if a whole program uses only restricted set of functions or semantics.

When applied with Spawn, worlds could also be created that have limited functionality required.

This approach overcomes the problem that the system is by default "totally connected" virtually speaking, and that is often much more than needed. Although lazy approaches to resource allocation are possible, letting the program declare what it actually needs is useful.

The ability to describe an Info argument that might contain a group reference implies that MPI_Group operations would have to be legal before MPI_Init() operations. That would be a change to the standard, admittedly, and also require that groups so created (which would only be ranks with a known maximum size) would have to be potentially updated after Init. That might cause some heartburn. It might mean that where we have specified the use of an MPI_Group for differentiated service, that people might prefer a list of the ranks of the underlying group itself.

Relating to Reinit() / Shrink():
Although we are not specifically advocating for MPI_Reinit(), the idea of MPI_Reinit() is describable, at least in spirit, with this kind of approach; Reinit() it is an MPI_Init_differentiated() using a sparse info object, if the goal is to work with a subset of surviving ranks. A set of Reinit operations could be defined as Info arguments. This provides another possible tie to fault tolerance strategies.

A form of the MPI_Comm_Shrink() could also be achieved using MPI_Dup_differentiated() with the right info arguments too.

OTHER NOTES

We **may** need a "fence" or "barrier" to allow the communicator to synch up after all the inits. Right now, we don't say how they link up… the assumption is that this is handled internally without a specific barrier.