PERSISTENCE WORKING GROUP DOCUMENT – MAY 9, 2011 (REVISED MAY 11, 2011 BASED ON PLENARY FEEDBACK ON MAY 10 @ MPI FORUM MTG.)

A. SKJELLUM, COORDINATOR

PURPOSES:

- MAKE POINT-TO-POINT AND COLLECTIVE OPERATIONS WITH REUSE FASTER THAN POSSIBLE IN MPI-1 AND MPI-2.
- REDUCE FLOW CONTROL OVERHEADS WHERE SEMANTICS SUPPORT. ENHANCE OVERLAPPING OF COMMUNICATION, COMMUNICATION, AND PIPELINING.
- REVEAL MEANS FOR IMPLEMENTATIONS TO REDUCE THE CRITICAL PATH OF INSTRUCTIONS NEEDED TO MOVE DATA
- USE CONCEPTS OF EARLY BINDING (AKA PLANNED TRANSFER)
- OFFER COMPLETELY EARLY BINDING, WITH SIMPLE EXTENSIONS TO SUPPORT EXTREMELY FAST TRANSFERS (E.G., CIRCULAR BUFFERS).
- MINIMIZE THE CHANGES (AND HENCE SOFTWARE ENGINEERING COSTS) NEEDED IN CURRENT MESSAGE PASSING CODES THAT HAVE THE RIGHT SEMANTICS TO TAKE ADVANTAGE OF THE PROPERTIES (REGULARITY, DATA PARALLEL, SUFFICIENT INTERNAL FLOW CONTROL).

HISTORY: LAST TWO WORKING GROUP MEETINGS IN DECEMBER AND FEBRUARY, NO PROGRESS IN MARCH MEETING.

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 intracommunicator in both calls; should also work fine with intercommunicators.

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

PROPOSAL 1. Creating, Managing, Using, Freeing persistent point to point communication channels.

Given a send-request *sreq1* created using MPI_Send_init() or any variant thereof, and a receive-request *rreq1,* created using MPI_Recv_init() or any variant thereof, with the following properties:

a) The requests are created using the same intra communicator or intercommunicator, as noted above
b) MPI_Start(sreq1) and MPI_Start(rreq1) would match under current MPI matching rules.

The following new functionality is defined

```
MPI_BIND_CHANNEL(Request_In, Request_Out, Info)
MPI_Request Request_In      [IN]
MPI_Request Request_Out     [OUT]
MPI_Info    Info            [IN]
```

MPI_Bind_channel(MPI_Request Request_in, MPI_Request *Request_out, MPI_Info info)

Then, a persistent point-to-point communication *channel* is created between the pair of processes as follows:

Send side: MPI_Bind_channel(sreq1, &sreq2, sinfo, comm);

Recv side: MPI_Bind_channel(rreq1, &rreq2, rinfo, comm);

This is a pairwise collective operation on the pair of processes. It is blocking. On return, if there is no error, then

- sreq1, rreq1 are unchanged
- sreq2, rreq2 are bound endpoints that now are in a separate communication space w.r.t. ordering and completion compared to comm.
- Any special properties to be defined, come from info arguments [TBD]

This additional variant is defined:

```
MPI_IBIND_CHANNEL(Request_In, Request_Out, Info)
MPI_Request Request_In      [IN]
MPI_Request Request_Out     [OUT]
MPI_Info    Info            [IN]
```

MPI_Ibind_channel(MPI_Request Request_in, MPI_Request *Request_out, MPI_Info info);

This form is non-blocking, and a Wait, Test, or variant must be used on the in request before the out request becomes valid. The in request is not destroyed, because it is itself persistent.

Multiple blocking and non-blocking binds are also defined:

```
MPI_BIND_CHANNELS(Requests_In, Requests_Out, int Nrequests,
Info_Array)
MPI_Request Requests_In[]   [IN]
MPI_Request Requests_Out[]  [OUT]
INTEGER     Nrequests       [IN]
MPI_Info    Info_Array[]    [IN]
```

MPI_Bind_channels(MPI_Request Requests_in[], MPI_Request Requests_out[], int Nrequests, MPI_Info Info_Array[])

```
MPI_IBIND_CHANNELS(Requests_In, Requests_Out, int
Nrequests, Info_Array)
MPI_Request Requests_In[]   [IN]
MPI_Request Requests_Out[]  [OUT]
INTEGER     Nrequests       [IN]
MPI_Info    Info_Array[]    [IN]
```

MPI_Ibind_channels(MPI_Request Request_in[], MPI_Request Request_out[], int Nrequests, MPI_Info info_array[])

When using multiple channel binds, the pairwise channels can use different communicators, MPI will order completion so as to avoid deadlock.

Completion is determined by waiting/testing/probing the Request_in[] array elements.

The multiple channel form of this operation must be called in all processes involving either end point of a concerned channel, or else the program is erroneous. The operation is not collective over the whole communicators involved. Uninvolved processes in the communicator do not need to this operation.

Advice to users:

In using multiple binds, it us up to the user to ensure that the right end points bind to one another. A given persistent request can only appear once in a single call to MPI_BIND_CHANNELS or MPI_IBIND_CHANNELS.

If the same (comm, tag, dest) is used in more than one send request, or the same (comm, tag, src) is used in more than one receive request, the order of matching between such endpoints is arbitrary and MPI is not required to order them based on their order in the arrays. The same is true when using wildcards. This behavior could be used intentionally. Such programs are not erroneous.

Note that, as proposed, there are no order guarantees between pairs of channels created between the same two processes, even though they were created with the same communicator.

The status produced by MPI_Wait type functions using persistent binds provides information on the actual binding partner, which may be useful in terms of wildcard use cases, but provides no other information as currently proposed. Use of MPI_STATUS_IGNORE is legal.

End advice to users

Note to Forum

The properties of the Info arguments will be defined later in this proposal. For now, assume empty info arguments.

End Note to Forum

Persistent channels may be freed using MPI_Request_unbind(). These comprise a collective operation over the process pair involved:

```
MPI_UNBIND_CHANNEL(Request_In, Info)
MPI_Request Request_In      [IN]
MPI_Info    Info            [IN]
```

MPI_Unbind_channel(MPI_Request Request_in)

This is blocking over the process pair of the implied channel, and is erroneous when used on a non-bound channel or normal request.

These additional variants are proposed:

```
MPI_IUNBIND_CHANNEL(Request_In)
MPI_Request Request_In      [IN]
```

MPI_Iunbind_channel(MPI_Request Request_in, MPI_Info info);

This form is non-blocking, and a Wait, Test, or variant must be used on the request before operation is deemed completed. The in request is no longer useful after a successful WAIT-type operation.

Multiple blocking and non-blocking binds are proposed:

```
MPI_UNBIND_CHANNELS(Requests_In, int Nrequests)
MPI_Request Requests_In[]   [IN]
INTEGER     Nrequests       [IN]
```

MPI_Unbind_channels(MPI_Request Requests_in, int Nrequests)

```
MPI_IUNBIND_CHANNELS(Requests_In, int Nrequests,
Info_Array)
MPI_Request Requests_In[]    [IN]
INTEGER     Nrequests        [IN]
```

MPI_Iunbind_channels(MPI_Request Request_in[], int Nrequests)

Advice to users:

Using MPI_Request_free() on a persistent channel type request is erroneous.

The same group of channels created together can be unbound independently. The fact that they were created together places no constraint on the ordering or timing of their destruction.

End advice to users

Performing a Point-to-point transfer using a persistent channel

Properties of the bound communication channels

a) there can be only one outstanding
b) they automatically have the ready property

Sender initiates [T1]: MPI_Start(sreq2);
…
MPI_Wait(sreq2, MPI_STATUS_IGNORE);

Receiver initiates [T2 < T1-epsilon]: MPI_Start(rreq2);
…
MPI_Wait(rreq2, MPI_STATUS_IGNORE)

These requests can also be used in MPI_Start_all() groups, provided the ready-rules are observed for each pair.

Advice to Implementers / Users

Each start wait sequence must occur in lock step. This is a slack-1 channel, analogous to the LLC sublayer (ISO OSI RM seven layer model) stop-and-wait protocol.

The means for achieving pipelining is to have a set of similar channels in an array, that are started in turn, such as to create a circular buffer between senders and receivers.

Applications that can't use persistent ready semantics are less likely to want this than those that do.

End advice to Implementers / Users

Advice to implementers

The purpose of this function is to be as fast as possible, not be the same speed or slower than persistent send/receive, or normal send-receive.

The goal is that this could be a much reduced critical instruction path compared to normal send/receive. Ideally, an MPI_Start() on a send request could just be a DMA kickoff instruction, and the MPI_Test() could simply be to check that the DMA is completed.

Because channels automatically have the ready property, there are redundant MPI_Send_init() variants. The S-variants are meaningful, but they will require the implementation to be slower. This could actually be useful, in terms of building user-level ready and synchronous streams of communication.

End advice to implementers

Comments related to other parts of the Forum work.

1) Persistent channels with piggy-back information could be optimized.
2) There is absolutely nothing helpful that happens in a fault tolerant scenario with persistent channels, since they are working hard to work at the bare metal level of protocol for highest performance
3) One-sided communication may overlap with this concept. The purpose of this set of functionality is to allow minimal message-passing program modification to MPI-1 type programs that have simple, efficient semantics and can then get significant improvements without major rewrites. This effort does not mean to interfere with RMA.

End Comments related to other parts of the Forum work.

Relationship to other work

MPI/RT has point-to-point channels with QoS, similar to this concept, although with admission control and a lot more complexity including event handling, and time-based scheduling of transfers (zero-sided mode).

End relationship to other work

This concludes proposal 1.

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

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

Advice to everyone

There are only non-blocking variants.

All collective operations in MPI **must** get non-blocking variants. This proposal extends automatically, if approved, to all operations that are given non-blocking variants.

End advice to everyone

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]

If there is a non-blocking collective communication option for this kind of INIT operation, it should be provided as a non-blocking collective. However, this is not contemplated here. There is no equivalent operation for point to point.

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 as of 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.

An example of where alternative COLLECTIVE_INITs are merited would involve the slow evolution of the properties of the collective, such as growing sizes of an ALL-TO-ALL variant. This has to be scoped out in future.

End advice to forum

Comments related to other parts of the Forum work.

This clearly relies on the use and adoption of non-blocking collectives as has been proposed for MPI-2.x and MPI-3.

End Comments related to other parts of the Forum work.

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

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 are 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 use case given Proposal 1, is to create 5 channel-pairs between the two end points, each with a unique memory base.

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 autoincrementing 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.

Given that the goal is to have the lightest weight here, defining data types together with MPI_BOTTOM, and then having the state machine increment from MPI_BOTTOM modulo a maximum count is plausible.

Options A) We add more forms of MPI_Bind or MPI_Send_init(), MPI_Recv_init to support this.

Option B) We use required Info arguments to MPI_Bind, where we have added info arguments, to support this kind of state machine optimization.

In either event, as long as there is only a single persistent channel, there is only slack 1, so this remains incomplete.

Therefore, we propose to provide a form of Bind that has K-slackness and also supports the use of simple state machine transformations on the base addresses. In that way, DMA devices or other optimized transfer processors can have a push down of the procedure needed to queue the transfer without upcalls.

```
MPI_BIND_SLACK_CHANNEL(Request_In, Request_Out, int
Slackness, Info)
MPI_Request Request_In      [IN]
MPI_Request Request_Out     [OUT]
INTEGER     Slackness       [IN]
MPI_Info    Info            [IN]

MPI_IBIND_SLACK_CHANNEL(Request_In, Request_Out, Info)
MPI_Request Request_In      [IN]
MPI_Request Request_Out     [OUT]
MPI_Info    Info            [IN]
```

MPI_Bind_slack_channel(MPI_Request Request_in, MPI_Request *Request_out, int slackness, MPI_Info info)

MPI_Ibind_slack_channel(MPI_Request Request_in, MPI_Request *Request_out, int slackness, MPI_Info info);

For Slackness = 1, this produces the same channel as previously described in Proposal 1.

The multiple forms of binding are also proposed. The unbinding operation is the same as in Proposal 1, a slack variant is not needed.

Multiple blocking and non-blocking binds are also defined:

```
MPI_BIND_SLACK_CHANNELS(Requests_In, Requests_Out, int
Nrequests,  int Slackness_Array[], Info_Array)
MPI_Request Requests_In[]   [IN]
MPI_Request Requests_Out[]  [OUT]
INTEGER     Nrequests       [IN]
INTEGER     Slackness_Array[] [IN]
MPI_Info    Info_Array[]    [IN]
```

MPI_Bind_slack_channels(MPI_Request Requests_in[], MPI_Request Requests_out[], int Nrequests, int *Slackness_Array MPI_Info Info_Array[])

```
MPI_IBIND_CHANNELS(Requests_In, Requests_Out, int
Nrequests, Slackness_Array, Info_Array)
MPI_Request Requests_In[]   [IN]
MPI_Request Requests_Out[]  [OUT]
INTEGER     Nrequests       [IN]
INTEGER     Slackness_Array[] [IN]
MPI_Info    Info_Array[]    [IN]
```

MPI_Ibind_slack_channels(MPI_Request Request_in[], MPI_Request Request_out[], int Nrequests, MPI_Info Info_Array[]);

When using multiple channel binds, the pairwise channels can use different communicators, MPI will order completion so as to avoid deadlock.

Advice to users

Slackness must be the same on both ends of the channel created, or the program is erroneous.

There is no such thing (currently) as cutting and pasting buffers in channels, so a sender sends 1K doubles, and a receiver receives twice as many 512 double buffers. Why not? Should be reviewed. Can definitely be supported to have data types that are commensurable, given the early binding nature of the protocols. This is already implied in slack-1 channels, because the send and received data/type counts don't have to be the same, so this definitely needs to be reviewed, in terms of # of transfers. Split and merging of messages is a new idea not otherwise in MPI .

End advice to users

The following info arguments are defined. If present, they must be interpreted, and used correctly:

ADDRESS_BASE_INCREMENT = count by which to increment the address (can be negative)

The modulus is computed automatically via the slackness and the count information in the original data types.

Advice to WG

This could simply be an extra argument instead of an info argument.

End advice to WG

Utilizing a slack channel:

Given a K-slack send channel Ksreq1, and its corresponding Krreq1, then a series of starts and waits is allowed.

Advice to WG

Without autoincrementing addresses, a K-slack channel is useless.

End advice to WG

Rules

Sender may start as many as K times before first Wait.

Receiver may start as many as K times before first Wait.

No more than K outstanding send or receive starts on either end is legal.

The ready semantics must be maintained.

In this form, specifying either Synchronous or Ready is possible. Synchronous means flow control every K times. Ready means no flow control, but you have to ensure ordering to make the program correct.

Proposal 4. Bundling Channels [INCOMPLETE]

In order to allow arbitrary communication patterns that are both persistent and collective, channel bundling provides a simple means to support such operations.

The WG will make a more formal proposal on this concept, but it amounts to this:

A means to make a persistent generalized request.

This allows very general new collective operations to be offered, with different communicators, different data motion, early binding, etc, without focusing on a graph topology or other specific predefined scheme, including of course that different layers of the program contribute to the pattern.

Miscellaneous Notes

1) We haven't proposed rebind operations in this form of the proposal; we could, but they don't appear faster than a bind
2) The simple autoincrementing addresses go a long way to supporting very fast DMA reuse without upcalls, but other state machines are possible too, some of which are optimizable, such as strides potentially. We should investigate

3) The implications of using state machines for addresses in collectives requires a bit of further thought, but use of INFO arguments already is the way to do this.
4) More general rebinding of data, such as for growing the size of an ALL-TO-ALL periodically, is an important use case, but beyond current proposal. It will be in the next proposal version if this proposal survives.

ADDENDUM: KEY OUTCOMES OF MAY 10, 2011 DISCUSSION

1) The WG will explore non-blocking collective operations for I/O with Quincy K, and propose non-blocking collective persistent operations.
2) The WG will explore and propose persistent one-sided communication based on MPI-3 RMA, and compared and contrasted with the current proposals.
3) The WG will create Wiki presence for this and previous presentations.
4) This document will be put in LaTex format for July meeting.
5) Show how to manage collectives that change over time, but are still somewhat persistent; also for point-to-point (intermediate binding, not completely early, not completely late).
6) Consider proposing persistent generalized requests.