MPI HYBRID & ACCELERATOR WORKING GROUP UPDATE

James Dinan, NVIDIA
HACC WG Chair
SC '22 MPI BoF

HYBRID & ACCELERATOR WORKING GROUP

Mission: Improve interoperability of MPI with other programming models

Active topics:

1. Continuations proposal [Joseph Schuchart, UTK] #6
2. Clarification of thread ordering rules [Daniel Holmes, Intel] #117
3. Integration with accelerator programming models:
    1. Accelerator info keys [Jim Dinan, NVIDIA] #3 #714
    2. Accelerator Synchronous MPI Operations [Several Proposals] #11
    3. Accelerator bindings for partitioned communication [Jim D., NVIDIA + Maria Garzaran, Intel] #4
    4. Partitioned communication buffer preparation [Ryan Grant, Queen’s U.] #264
4. Asynchronous operations [Quincey Koziol, Amazon] #585

More information: https://github.com/mpiwg-hybrid/hybrid-issues/wiki

COMPLETION CONTINUATIONS

Treat the completion of an MPI operation as continuation of some activity

- Interoperability with asynchronous and multithreaded programming models
- Register callbacks that continue the activity upon completion of an MPI operation

```c
MPI_Request cont_req;
MPIX_Continue_init(&cont_req);

omp_event_handle_t event;
int value;
#pragma omp task depend(out:value) detach(event)
{
  MPI_Request req;
  MPI_Irecv(&value, ...., &req);
  MPIX_Continue(&req, &release_event, event, MPI_STATUS_NULL, cont_req);
}

#pragma omp task depend(in: value)
{
  // process value
}
```

```c
void release_event(MPI_Status status, void *data)
{
  omp_event_handle_t event = (omp_event_handle_t)(uintptr_t)data;
  omp_fulfill_event(event);
}
```

"Callback-based completion notification using MPI Continuations,"
Joseph Schuchart, Christoph Niethammer, José Gracia, George Bosilca, Parallel Computing, 2021.

"MPI Detach - Asynchronous Local Completion,"
Joachim Protze, Marc-André Hermanns, Ali Demiralp, Matthias S. Müller, Torsten Kuhlen. EuroMPI '20.

CLARIFICATION OF THREAD ORDERING

MPI-3.1 Section 3.5: If a process has a single thread of execution, then any two communications executed by this process are ordered. On the other hand, if the process is multithreaded, then the semantics of thread execution may not define a relative order between two send operations executed by two distinct threads. The operations are logically concurrent, even if one physically precedes the other. In such a case, the two messages sent can be received in any order. Similarly, if two receive operations that are logically concurrent receive two successively sent messages, then the two messages can match the two receives in either order.

Option 1: Operations from different threads are unordered, even if app. enforces ordering

- Pro: Can enable MPI libraries to optimize performance for multithreaded applications
- Con: Hard to get ordering, and MPI doesn't know what the user considers to be a thread
  - E.g., A user-level thread can be migrated to a different shepherd thread. Does MPI see it?

Option 2: MPI must respect an order across threads if the user enforces one

- Pro: Probably what users expect, can relax ordering with info assertions and per-thread comms
- Con: Removes a (questionable) performance optimization opportunity

Option 3: Clarify that thread ordering is currently ambiguous

- May be addressed in a future version of the standard, tied to other threading interoperability efforts in HACC WG
- Portable applications should assume the most relaxed semantic (option 1)

ALLOCATOR KIND INFO

Use MPI info to provide users with a portable solution to:

1. Detect whether accelerator memory is supported by the MPI library
2. Request support for accelerator memory from the MPI library (when using Sessions)
3. Constrain usage of accelerator memory to specific communicators, windows, etc.

mpi_request_memory_alloc_kind

- Request support for memory allocator kind from the MPI library

mpi_assert_memory_alloc_kind

- Assert memory kinds used by the application on the given MPI object

mpi_memory_alloc_kind

- Memory kinds supported by the MPI library

REQUEST SUPPORT FOR CUDA ALLOCATED MEMORY

```
bool cuda_aware = false;
int len = MPI_MAX_INFO_VAL, flag = 0;
char *val = malloc(MPI_MAX_INFO_VAL);
MPI_Info info;

MPI_Info_create(&info);
MPI_Info_set(info, “mpi_request_memory_alloc_kind”, “cuda:device”);
MPI_Session_init(info, MPI_ERRORS_ARE_FATAL, &session);
MPI_Info_free(&info);

MPI_Session_get_info(session, &info);
MPI_Info_get_string(info, “mpi_memory_alloc_kind”, &len, val, &flag);

// Check mpi_memory_alloc_kind for “cuda” or “cuda:device”
while (flag && (kind = strsep(&val, ",")) != NULL) {
  if (strcasecmp(kind, "cuda") == 0 || strcasecmp(kind, "cuda:device") == 0) {
    cuda_aware = true;
    break;
  }
}
```

ACCELERATOR SYNCHRONOUS MPI OPERATIONS

Integrate MPI operations as tasks in a deferred work scheduling model

Four approaches under discussion:

1. Stream and Graph Based MPI Operations (Jim Dinan, NVIDIA) #5
2. MPIX Stream (Hui Zhou, Argonne National Lab.) #12
3. MPIX_Queue (Naveen Ravichandrasekaran, HPE)
4. Project Delorean (Quincey Koziol, Amazon) #585

Key differences:

- Batching of operations to optimize overheads of coordination across models
- Exposure of external work scheduler through an MPI object
- Task scheduling control, internal (e.g. MPI) or external (e.g. CUDA, HIP, etc.)

PROPOSAL 1: STREAM-BASED MPI OPERATIONS

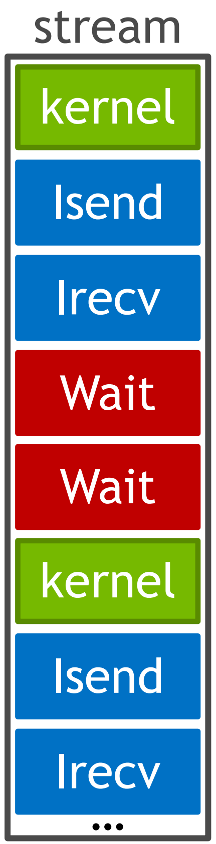
Simple Ring Exchange Using a CUDA Stream

```
MPI_Request send_req, recv_req;
MPI_Status  sstatus,  rstatus;

for (i = 0; i < NITER; i++) {
  if (i > 0) {
    MPI_Wait_enqueue(recv_req, &rstatus, MPI_CUDA_STREAM, stream);
    MPI_Wait_enqueue(send_req, &sstatus, MPI_CUDA_STREAM, stream);
  }

  kernel<<<..., stream>>>(send_buf, recv_buf, …);

  if (i < NITER - 1) {
    MPI_Irecv_enqueue(&recv_buf, …, &recv_req, MPI_CUDA_STREAM, stream);
    MPI_Isend_enqueue(&send_buf, …, &send_req, MPI_CUDA_STREAM, stream);
  }
}
cudaStreamSynchronize(stream);
```



Prototype available: https://github.com/NVIDIA/mpi-acx

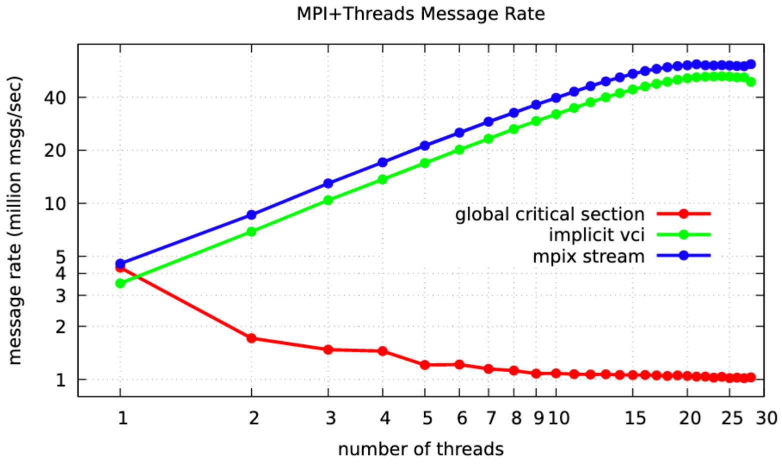
Proposal 2: Introduce MPIX_Stream Object

- MPIX Stream object represents execution context, can be mapped to network endpoints at both sender *and* receiver

```
int MPIX_Stream_create(MPI_Info info, MPIX_Stream *stream)
```

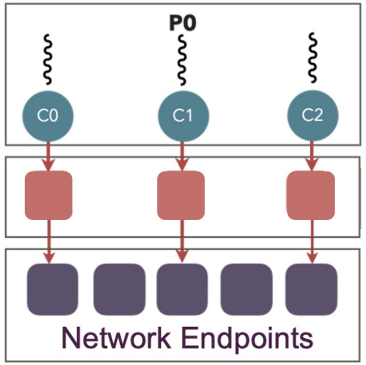
- MPIX Stream communicator as context, provide backward compatibility

```
int MPIX_Stream_comm_create(MPI_Comm parent_comm,
                MPIX_Stream stream, MPI_Comm *stream_comm)
```



- Enqueue APIs for async launching operations onto GPU context

```
int MPIX_Send_enqueue(buf, count, datatype, dest, tag, comm)
int MPIX_Recv_enqueue(buf, count, datatype, source, tag, comm, status)
int MPIX_Isend_enqueue(buf, count, datatype, dest, tag, comm, request)
int MPIX_Irecv_enqueue(buf, count, datatype, source, tag, comm, request)
int MPIX_Wait_enqueue(request, status)
int MPIX_Waitall_enqueue(count, array_of_requests, array_of_statuses)
```



Paper: https://arxiv.org/abs/2208.13707

PROPOSAL 3: QUEUE-MEDIATED STREAM TRIGGERED INTEGRATION

- New ***MPIX_Queue*** object
  - Users can tie an MPI Queue to a GPU Stream
  - Implementation can batch up MPI operations
- Users enqueue operations on the Queue
  - Stream-aware data movement operations
  - Enqueueing is nonblocking
- Enqueued operations are started as a batch
  - Inserts a control operation into GPU stream
  - Triggers the batch of MPI ops in stream order
- Advantage of MPIX_Queue with MPI P2P
  - Offload operations to NIC for execution
  - Batching provides support for efficient NIC resource management

int **MPIX_Create_queue** (IN void * stream, OUT MPIX_Queue * queue)
int **MPIX_Free_queue** (IN MPIX_Queue queue)

int **MPIX_Enqueue_send** (const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPIX_Queue queue, MPI_Request *req)

int **MPIX_Enqueue_recv** (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPIX_Queue queue, MPI_Request *req)

int **MPIX_Enqueue_start** (const MPIX_Queue queue)
int **MPIX_Enqueue_wait** (const MPIX_Queue queue);

Link to paper: https://arxiv.org/pdf/2208.04817.pdf

PROPOSAL 4: PROJECT DELOREAN

Composable Asynchronous Communication Graphs and Streams in MPI

Asynchronous data movement orchestration to overlap compute, communication, and I/O

Extend MPI with graphs and streams

Contain both deferred MPI operations and user-defined operations

MPI executes graph/stream and guarantees forward progress

Use the whole machine simultaneously!

```
// Info keys control graph execution behavior
MPIX_Graph_create(&graph, info);
...
// Define deferred operations and add to graph
MPIX_File_open_def(..., &token);
MPIX_Graph_add(graph, token, <dependency info>);
...
MPIX_File_read_def(..., &token);
MPIX_Graph_add(graph, token, <dependency info>);
...
MPIX_File_close_def(..., &token);
MPIX_Graph_add(graph, token, <dependency info>);
...
MPIX_Bcast_def(..., &token);
MPIX_Graph_add(graph, token, <dependency info>);
...
// Execute operations in graph
MPIX_Graph_execute(graph);
```

ACCELERATOR BINDINGS FOR MPI PARTITIONED APIS

CUDA and SYCL Language Bindings Under Exploration

```
int MPI_Psend_init(const void *buf, int partitions, MPI_Count count,
                   MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Info info,
                   MPI_Request *request)

int MPI_Precv_init(void *buf, int partitions, MPI_Count count,
                   MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Info info,
                   MPI_Request *request)

int MPI_[start,wait][_all](...)
```

Keep host only

Add device bindings

```
__device__ int MPI_Pready(int partition, MPI_Request request)

__device__ int MPI_Pready_range(int partition_low, int partition_high, MPI_Request request)

__device__ int MPI_Pready_list(int length, const int array_of_partitions[], MPI_Request request)

__device__ int MPI_Parrived(MPI_Request request, int partition, int *flag)
```

KERNEL TRIGGERED COMMUNICATION USAGE

Partitioned Neighbor Exchange

Host Code

```
MPI_Request req[2];
MPI_Psend_init(..., &req[0]);
MPI_Precv_init(..., &req[1]);
while (...) {
  MPI_Startall(2, req);

  kernel<<<..., s>>>(..., req);

  MPI_Waitall(2, req);
}
MPI_Request_free(&req[0]);
MPI_Request_free(&req[1]);
```

Device Code

```
__device__
void MPI_Pready(int idx, MPI_Request req);

__global__ kernel(..., MPI_Request *req) {
  int i = my_partition(...);
  // Compute and fill partition i
  // then mark i as ready
  MPI_Pready(i, req[0]);
}
```

Prototype available: https://github.com/NVIDIA/mpi-acx

KERNEL TRIGGERED COMMUNICATION USAGE

Partitioned Neighbor Exchange

Host Code

```
MPI_Request req[2];
MPI_Psend_init(..., &req[0]);
MPI_Precv_init(..., &req[1]);
while (...) {
  MPI_Startall(2, req);
  MPI_Prepare_all(2, req);
  kernel<<<..., s>>>(..., req);

  MPI_Waitall(2, req);
}
MPI_Request_free(&req[0]);
MPI_Request_free(&req[1]);
```

Device Code

```
__device__
void MPI_Pready(int idx, MPI_Request req);

__global__ kernel(..., MPI_Request *req) {
  int i = my_partition(...);
  // Compute and fill partition i
  // then mark i as ready
  MPI_Pready(i, req[0]);
}
```

Prepare allows the sender to wait until receiver is ready, so Pready is a copy or RDMA write

Thank you!

Wednesdays 10-11am US Eastern Time

https://github.com/mpiwg-hybrid/hybrid-issues/wiki