MPI HYBRID & ACCELERATOR WORKING GROUP DISCUSSION

MPI Forum Meeting
May 23, 2022

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
    2. Stream/Graph Based MPI Operations [Jim Dinan, NVIDIA] #5
    3. Accelerator bindings for partitioned communication [Jim D., NVIDIA + Maria Garzaran, Intel] #4
    4. Partitioned communication buffer preparation [Ryan Grant, Queen's U.] #264

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

```
11  MPI_Request cont_req;
12  MPIX_Continue_init(&cont_req);
13
14  omp_event_handle_t event;
15  int value;
16  #pragma omp task depend(out:value) detach(event)
17  {
18    MPI_Request req;
19    MPI_Irecv(&value, ..., &req);
20    MPIX_Continue(&req, &release_event, event, MPI_STATUS_NULL, cont_req);
21  }
22
23  #pragma omp task depend(in: value)
24  {
25    // process value
26  }
```

```
31  void release_event(MPI_Status status, void *data)
32  {
33    omp_event_handle_t event = (omp_event_handle_t)(uintptr_t)data;
34    omp_fulfill_event(event);
35  }
```


"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.

Camp B: Operations from different threads are unordered

- 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?

Camp A: MPI must respect an order across threads

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

ACCELERATOR INFO

Is the MPI library CUDA/HIP/L0/ETC-Aware?

“mpi_memory_kind” (string, default: implementation defined)

- This info key contains a comma separated list of the memory kinds that the MPI library supports as buffer arguments to MPI routines. Possible values include:
  - “sysmem” - Memory allocated by standard operating system allocators
  - “cuda” - Memory allocated by CUDA memory allocators
  - “hip” - Memory allocated by HIP memory allocators
  - “l0” - Memory allocated by One API L0 memory allocators
- This info key should appear in MPI_INFO_ENV and info returned by MPI_SESSION_GET_INFO
- This info key can be passed in the info argument to MPI_SESSION_INIT to request support for a memory kind
  - Users must check info on the session to determine what is actually supported

ACCELERATOR INFO

Pointer Attribute Checking Overhead

“mpi_assert_memory_kind” (string, default: none)

- This info key contains a comma separated list of the memory kinds that will be used by operations involving the given MPI object. Possible values include:
  - “sysmem” - Memory allocated by standard operating system allocators
  - “cuda” - Memory allocated by CUDA memory allocators
  - “hip” - Memory allocated by HIP memory allocators
  - “l0” - Memory allocated by One API L0 memory allocators

- Can be applied to MPI communicators and datatypes

MPI DATATYPE INFO

New Functions

```
int MPI_Type_commit_with_info(MPI_Datatype *datatype, MPI_Info info);
int MPI_Type_get_info(MPI_Datatype datatype, MPI_Info *info_used);
```

- Info does not automatically propagate (e.g. when making another datatype using a datatype that has info on it)

STREAM TRIGGERED NEIGHBOR EXCHANGE

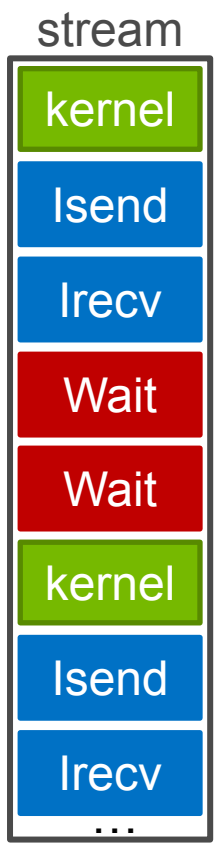
Simple Ring Exchange Using a CUDA Stream

```
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);
```



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);
  MPI_Pbuf_prepare_all(2, req);
  kernel<<<..., s>>>(..., req);
  cudaStreamSynchronize(s);
  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]);
}
```

KERNEL & STREAM TRIGGERED COMMUNICATION USAGE

Partitioned Neighbor Exchange

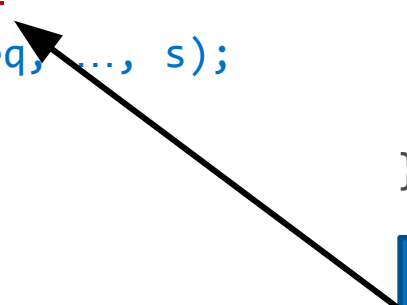
Host Code

```
MPI_Request req[2];
MPI_Psend_init(..., &req[0]);
MPI_Precv_init(..., &req[1]);
while (...) {
  MPI_Startall_enqueue(2, req, …, s);
  MPI_Pbuf_prepare_all_enqueue(2,req,…,s);
  kernel<<<..., s>>>(..., req);
  cudaStreamSynchronize(s);
  MPI_Waitall_enqueue(2, req, …, s);
}
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]);
}
```



Moving control ops to stream eliminates stream synchronization overhead

KERNEL & STREAM 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_enqueue(2, req, …, s);
  MPI_Pbuf_prepare_all_enqueue(2,req,…,s);
  kernel<<<..., s>>>(..., req);
  cudaStreamSynchronize(s);
  MPI_Waitall_enqueue(2, req, …, s);
}
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]);
}
```

Allows the sender to wait until receiver is ready, so Pready becomes RDMA write

Thank you!

Wednesdays 10-11am US Eastern Time

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