EXPLICIT MPI+X PROPOSAL – MPIX STREAM

Hybrid & Accelerator WG Meeting

Hui Zhou /ANL

Wednesday, June 22, 2022

BACKGROUND AND MOTIVATION

- Need a GPU stream enqueue API for MPI communications
- MPI dose not have a concept for GPU stream
- MPI also does not have a concept for thread execution context
- MPI standard only specifies the compatibility for MPI+Thread
- GPU-aware implementation satisfies the compatibility of MPI+GPU
  (A gpu support level interface would be nice)
- Compatibility is only half of the story, the other half is performance
- Thread synchronization and GPU synchronization are critical to performance

BACKGROUND AND MOTIVATION

Serial execution context

- Both thread and GPU stream are serial execution context
- MPI does not control either thread or GPU stream, by design
- Identification of serial execution context is crucial to avoid unnecessary synchronizations
- Sufficient identification and promise of serial execution context may relieve MPI the burden of (thread and GPU) synchronizations

CONSIDERATIONS OF API DESIGN

A. Successful story: hiding the *concept* from user
   - When user only concerned with the concept as means to an end
   - Example: MPI ranks and conncetions between ranks
   - Reduces complexity, improves performance

B. Not so successful: hiding the *control* from user
   - When user already works with the concept directly
   - Example: thread synchronization in MPI_THREAD_MULTIPLE
   - It hides the control of thread synchronization
   - It increases complexity, hampers performance

MPIX STREAM

- `MPIX_Stream` identifies a serial execution context

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

- `info` can be `MPI_INFO_NULL`, identifies a generic thread context
- For special stream, e.g. for `cudaStream_t`

```
MPI_Info_create(&info);
MPI_Info_set(info, "type", "cudaStream_t");
MPIX_Info_set_hex(info, "value", &stream, sizeof(stream));

MPIX_Stream_create(info, &mpi_stream);
```

STREAM COMMUNICATOR

- Stream communicator is a communicator with local streams attached.

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

- MPIX streams are local, but communications are between pairs of them
- Otherwise, synchronizations are unavoidable at receiver or sender.
- It okay for `stream` to be `MPIX_STREAM_NULL`.
- Conventional communicators are the same as stream communicators with `MPIX_STREAM_NULL` on every process.

USING STREAM COMMUNICATOR

Operations with thread context

- Use stream communicators the same way as using normal communicators, but remember –
- It is illegal to issue operations on the same MPIX_Stream concurrently
- In particular, do not MPI_Waitall on mixed requests

- In a sense, we are shifting the burden of thread synchronization from implementation to application
- It is less complex and more effective on the application side

GPU ENQUEUE OPERATIONS

Point-to-Point Communications

- Same interface as conventional API, but explicit names may be preferred

```
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)
```

GPU ENQUEUE OPERATIONS

Collective Communications

- Same interface as conventional API, but with explicit names, e.g. –

```
int MPIX_Allreduce_enqueue(sendbuf, recvbuf,
                           count, datatype, op, comm)
```

- Compare to NCCL API -

```
ncclResult_t ncclAllReduce(sendbuff, recvbuff, count,
        ncclDataType_t datatype, ncclRedOp_t op,
        ncclComm_t comm, cudaStream_t stream)
```

GPU ENQUEUE OPERATIONS

One-sided communications

- Implementations can identify local GPU stream context from the stream communicator that is used for creating windows
- The semantics of one-sided communications are often tied to window synchronization
- E.g. MPIX_Put_enqueue and MPI_Win_fence does not work unless it is MPIX_Win_fence_enqueue
- Making all one-sided operations on a stream communicator with local GPU stream attached implicit enqueuing, may not be a bad idea.

SIDETRACK: ASYNC PROGRESS THREAD

- The implementation of stream enqueue operations may require an asynchrounous progress thread, at least for generic fallbacks
- [wish] GPU runtime events need support CPU/GPU 2-way synchronization
- [alternative] cudaLaunchHostFunc, where CUDA runtime manages the progress thread.
- It may be less complex and more effective to let application manage progress thread(s).
- They'll need a genric progress function --

```
int MPIX_Stream_progress(MPIX_Stream stream)
```

ONE-TO-MANY AND MANY-TO-MANY

Multiplex Stream Communicators

- Stream communicator binds one local stream to each process, good for one-to-one communication patterns.
- Many HPC applications observe one-to-one patterns, e.g. stencil computations
- Others need one-to-many or many-to-many communication patterns.
  E.g. task-based applications
- It may be convenient, and sometime necessary, to allow addressing multiple streams using a single communicator --

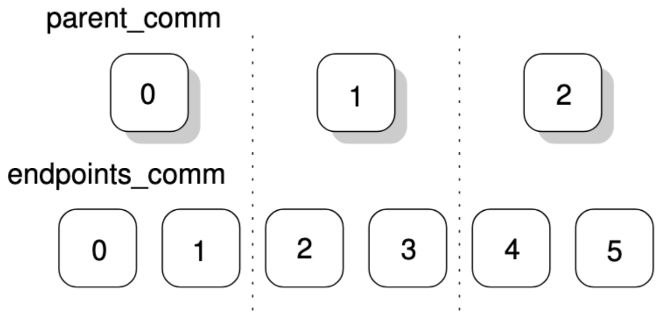
```
int MPIX_Stream_comm_create_multiple(MPI_Comm parent_comm,
                                     int count, MPIX_Stream local_streams[],
                                     MPI_Comm *stream_comm)
```

USING MULTIPLEX STREAM COMMUNICATOR

- A set of point-to-point APIs that use rank+index to address local streams

```
int MPIX_Stream_send(buf, count, datatype, dest, tag, comm,
                     int src_idx, int dst_idx)
int MPIX_Stream_recv(buf, count, datatype, source, tag, comm,
                     int src_idx, int dst_idx, status)
int MPIX_Stream_isend(buf, count, datatype, dest, tag, comm,
                     int src_idx, int dst_idx, request)
int MPIX_Stream_irecv(buf, count, datatype, source, tag, comm,
                     int src_idx, int dst_idx, request)
```

COMPARING TO THE ENDPOINTS PROPOSAL



- Both achieves the same technical goal of explicit thread addressing
- Static attaching vs. semantic agreement
- With the endpoints proposal --
  - Incompatible communicators
  - Inflating thread to virtual process, which does not really address process+thread
- With MPIX Stream
  - Does not cover the inter-thread communication within a process

PERSISTENT COMMUNICATION ON STREAMS

- Complexity
  - Send
  - Isend + Wait
  - Send_init + Start + Wait + Free

- Missing formal semantics for the stream parameter

- MPIX stream can be used for persistent operations (if needed)

```
MPI_Send_init(send_buf, ..., &send_req);
MPI_Recv_init(recv_buf, ..., &recv_req);

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_Start_enqueue(recv_req, MPI_CUDA_STREAM, stream);
    MPI_Start_enqueue(send_req, MPI_CUDA_STREAM, stream);
  }
}
cudaStreamSynchronize(stream);
```

SIDETRACK: GPU TRIGGERED OPERATION

- Solution 1: MPI functions in GPU kernels
  - Which need be in kernel? Where do we draw the line? Seems *ad hoc*

- Alternative: What if MPI define an MPIX Event that can interoperate with X
  - Just as MPIX Stream is the missing execution context
  - Event may be the missing element for synchronization

COMPARE TO PRIOR ARTS

MPI Session

- Both MPI Session and MPIX Stream are local objects that can be customized or extended with arbitrary info hints
- If we attach MPI_THREAD_SERIALIZED to an MPI Session, techinically we can replace MPIX Stream with MPI Session …
- But
  - Good luck explain the API
  - Session is also good for other purpose, such as for isolating library contexts
  - Session has its complexity – init/finalize/bootstraping

“LOGICAL CONCURRENCY”

When a thread is executing one of these (MPI) routines, if another concurrently running thread also makes an MPI call, the outcome will be as if the calls executed in some order

- On the surface, MPI will serialize all MPI routines.
- It is a hammer definition that is effective for addressing compatibility
- But, it is also desirable to achieve performance, thus we argue for its interpretation…
- With MPIX stream, performance is achievable, so the all-serializing interpretation of the default stream may be acceptable