Flexible Communication Endpoints

MPI Forum Hybrid Working Group

March 13, 2013

Flexible Communication Endpoints Overview

- Allow threads/tasks to acquire MPI ranks
- Benefit: progress for threads
  - Make progress on endpoint rather than single shared rank

- Difference from previous proposed approaches
  - New: Spawn new communicators with additional endpoints
  - Old: special MPI_COMM_ENDPOINTS, Init_endpoints(), attach/detach()

- Rough sketch of the interface:
  1. Generate a communicator with additional endpoints
  2. Threads/tasks attach to endpoints
  3. … (awesomeness) …
  4. Free communicator

Creation of Communicator with Endpoints

- MPI_Comm_create_endpoints(
  - MPI_Comm parent_comm,
  - int my_num_ep,
  - MPI_Info info,
  - MPI_Comm *output_comm)

- Create a new intracommunicator where my_num_ep ranks will be available at each process
  - The operation is collective over parent_comm
  - No cached info propagates
  - All processes/threads are initially detached

- my_num_ep semantics
  - Each process can provide a different value
  - If value is zero, the process will be left out of new communicator

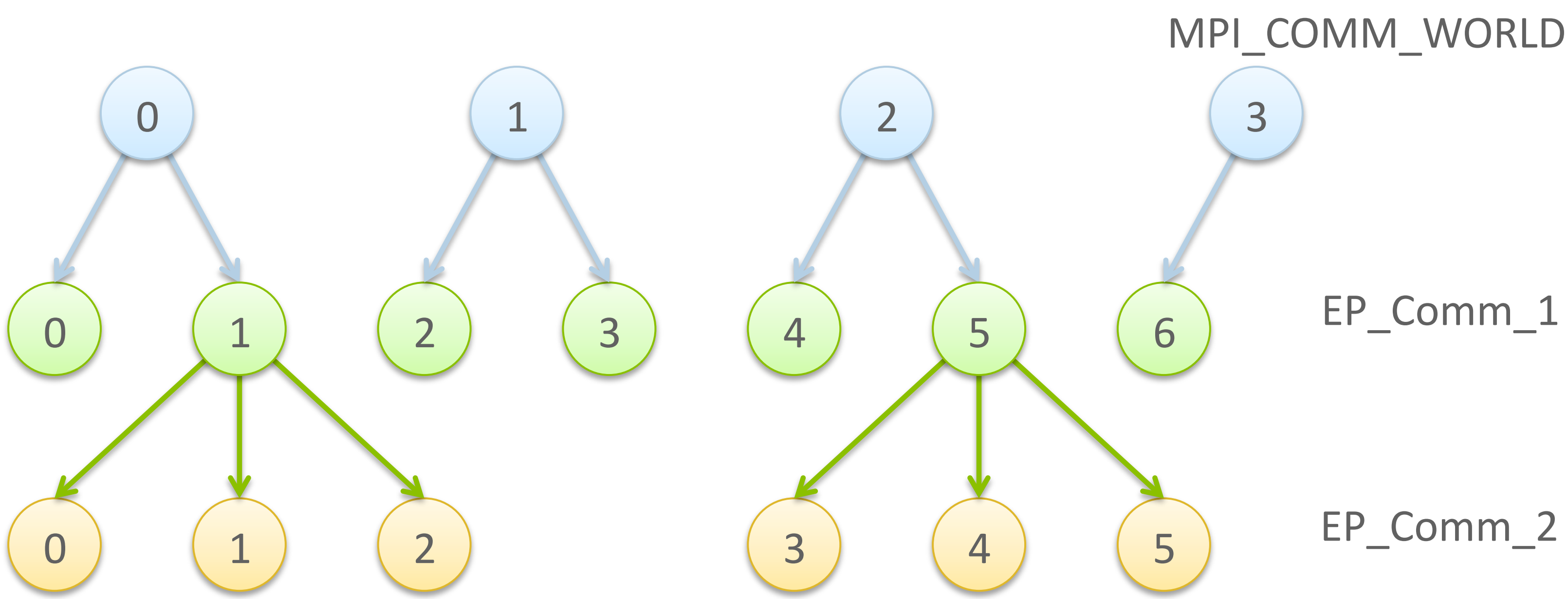
- Endpoint ranks are assigned in <process, index> order

Attaching threads/tasks to EP Communicator

- MPI_Comm_attach(
  - MPI_Comm comm,
  - int parent_rank,
  - int index)

- Initializes threads to make MPI calls on EP communicator
- Threads are attached to one of parent_rank's endpoints
  - Selected endpoint indicated by index argument
  - Multiple threads may attach to each communicator endpoint
  - A thread can attach to a communicator only once
- Default thread rank
  - Conventional communicators: parent process rank
  - Endpoint communicators: undefined, cannot call MPI until attached
- This call is not collective

Motivation for parent_rank/index



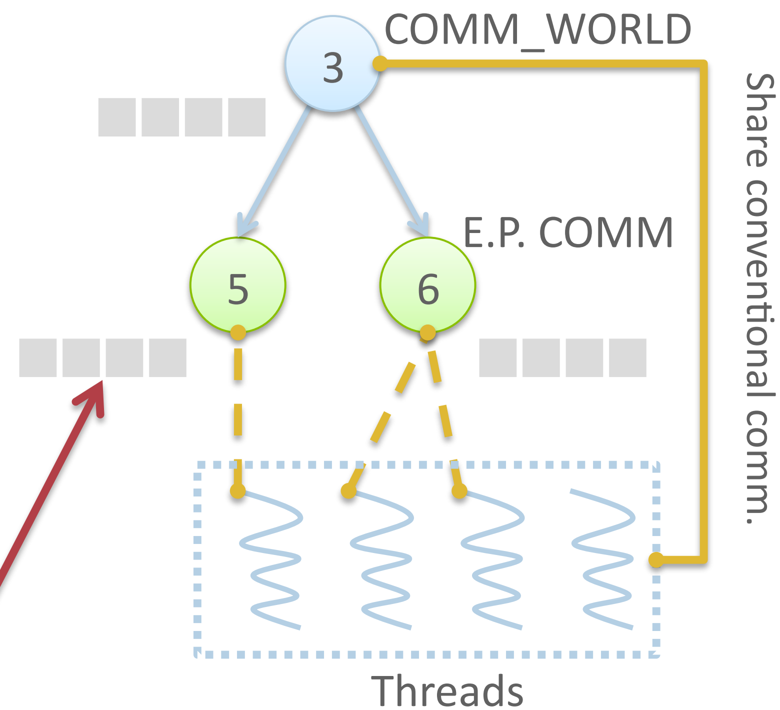
- Parent rank allows app to identify endpoint rank without knowing number of endpoints requested by other processes

Freeing a Communicator with Endpoints

- MPI_Comm_free(…)
  - Must be called once per endpoint

Progress on Communicator Processes/Endpoints

- Currently, each process makes individual progress on communications involving that process
- Threads make progress on all of their MPI ranks
  - Each thread makes progress on its "process"
  - Each thread makes progress on its attached endpoints
- Enables per-communicator progress for E.P. communicators
  - Individual message queues



Interoperability of EP communicators with …

- Endpoints are treated as “processes” or “ranks” in existing MPI calls
  - Existing calls should work as expected
- MPI_Comm_dup, split, create, etc.
  - Results in another communicator with endpoints
  - All endpoints from the parent communicator are attached to their rank in the output communicator
  - Additional threads are unattached by default and can call attach
- Request objects
  - Associated with a rank, must be completed by the rank that created them
- Collectives
  - Must be called once per rank in the EP communicator
- RMA
  - Communicators with endpoints can be used to create RMA windows
  - Standard local access semantics apply:
    - Only the rank that owns the window buffer is allowed to perform load/store
- I/O
  - Communicators with endpoints can be used for I/O

OpenMP Example - Global Communicator

Threads in the parallel region acquire MPI ranks

```
int main(int argc, char **argv) {
    int world_rank, tl;    MPI_Comm omp_comm;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &tl);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

#pragma omp parallel
    {
#pragma omp master
        {
            MPI_Comm_create_endpoints(MPI_COMM_WORLD, omp_get_num_threads(),
                                      MPI_INFO_NULL, &omp_comm);
        }
        MPI_Comm_attach(omp_comm, world_rank, omp_get_thread_num());
#pragma omp for
        for (...) {
            ...
        }

        MPI_Comm_free(&omp_comm);
    }
    MPI_Finalize(); return 0;
}
```

OpenMP Ex. - Hierarchical Node Communicator

Threads in the parallel region acquire MPI ranks

```
int main(int argc, char **argv) {
    int world_rank, tl;    MPI_Comm omp_comm;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &tl);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

#pragma omp parallel
    {
#pragma omp master
        {
            MPI_Comm_create_endpoints(MPI_COMM_SELF, omp_get_num_threads(),
                                      MPI_INFO_NULL, &omp_comm);
        }
        MPI_Comm_attach(omp_comm, 0, omp_get_thread_num());
#pragma omp for
        for (...) {
            ...
        }

        MPI_Comm_free(&omp_comm);
    }
    MPI_Finalize(); return 0;
}
```

UPC Example Notes

- In this example, UPC threads are implemented as threads
  - Not O.S. processes
- This UPC implementation utilizes a 1:1 mapping between UPC threads and MPI processes
  - Generate a new "endpoints" communicator where UPC threads are assigned new MPI ranks

UPC Example Code - Generated Code

```
/* This is C code, generated by the UPC compiler */
int main(int argc, char **argv) {
    int world_rank, tl;
    MPI_Comm upc_comm;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &tl);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    MPI_Comm_create_endpoints(MPI_COMM_WORLD, THREADS_PER_NODE,
                              MPI_INFO_NULL, &upc_comm);

    /* Calls upc_thread_init(), which calls user's upc_main() */
    UPCR_Spawn_threads(THREADS_PER_NODE, upc_thread_init, upc_comm);

    MPI_Finalize();
}


upc_thread_init(int argc, char **argv, MPI_Comm upc_comm) {
    MPI_Comm_attach(upc_comm, world_rank, MYTHREAD);
    upc_main(argc, argv); /* User's main function */
    MPI_Comm_free(upc_comm);
}
```

UPC Example Code - User's Code

```
shared [*] double data[100*THREADS];

int main(int argc, char **argv) {
    int rank, i;
    double err;
    MPI_Comm upc_comm;

    UPCMPI_World_comm_query(&upc_comm);

    do {
        upc_forall(i = 0; i < 100*THREADS; i++; i) {
            data[i] = …;
            err += …;
        }
        MPI_Allreduce(&err, …, upc_comm);
    } while (err > TOL);

    return 0;
}
```

Discussion

- Straw vote in support of continuing work on this proposal:
  - Yes: 20, No: 0, Abstain: 3

- An alternative interface was proposed, that consists of one function:

- MPI_Comm_create_endpoints(
  - MPI_Comm parent_comm,
  - int my_num_ep,
  - MPI_Info info,
  - MPI_Comm output_comms[])

- Collective on parent_comm, produces an array of communicator handles, one per endpoint. No attach/detach. Threads just start using one of the comms.

- Advantages:
  - Does not require THREAD_MULTIPLE (attach does)
  - Places fewer dependencies on threading model
  - Data encapsulated in MPI_Comm, removes dependency on thread-local storage

Additional Semantics

- MPI_Comm_create_endpoints info key
  - max_attached_per_ep = integer value
- Requests are associated with an endpoint
  - Must be completed by the endpoint that generated them

Things We Still Need to Figure Out…

- Query functions to find that a communicator is of "endpoint type", to find that different ranks are in the same address space, etc.
  - Similar to RMA interface, we could use: MPI_Comm_get_attr(…, MPI_COMM_FLAVOR, …)