MPI-3 Hybrid Working Group Status

MPI interoperability with Shared Memory

- Motivation: sharing data between processes on a node without using threads
- Sandia applications motivation:
  - Dump data into a "common" memory region
  - Synchronize
  - All processes just use this memory region in read-only mode
  - And they want this in a portable manner

MPI interoperability with Shared Memory: Plan A

- The original plan was to provide
  - Routines to allocate/deallocate shared memory
  - Routines to synchronize operations, but not to operate on memory (similar to MPI_Win_sync)
    - Operations on shared memory are done using load/store operations (unlike RMA)
    - Synchronization would be done with something similar to MPI_Win_sync
    - There was a suggestion to provide operations to operate on the data as well, but there was no consensus on that in the working group
  - A routine to create a communicator on which shared memory can be created
- The Forum's feedback that this was not possible to do in MPI unless it knows the compilers capabilities

```
Process 0                      Process 1
store(X)
MPI_Shm_sync()
MPI_Barrier() ----------------- MPI_Barrier()
                                MPI_Shm_sync()
                                load(X)
```

MPI interoperability with Shared Memory: Plan B

- The second plan was to remove shared memory synchronization routines; we still have:
  - Routines to allocate/deallocate shared memory
  - A routine to create a communicator on which shared memory can be created
- The Forum's feedback was that allocation/deallocation might not be useful without operations to synchronize data
  - The use-case for only creating shared memory and expect the user to handle memory barriers was fairly minimal
  - Also, another feedback was to do this as an external library

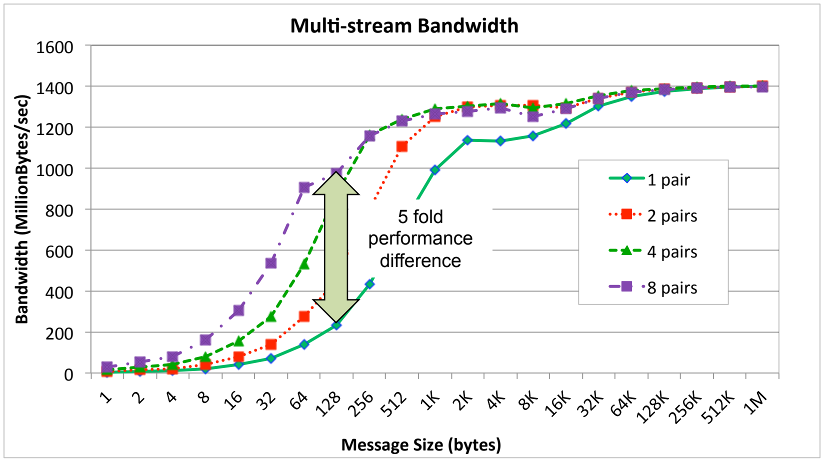
MPI interoperability with Shared Memory: Plan C

- The third plan is to remove shared memory allocation routines; we still have:
  - A routine to create a communicator on which shared memory can be created
- MPI_Shm_comm_create(old_comm, info, &new_comm)
  - The info argument can provide implementation-specific information such as, within the socket, shared by a subset of processes, whatever else
    - No predefined info keys
  - There has been a suggestion to generalize this to provide a communicator creator function that provides “locality” information
    - Will create an array of communicators, where the lower index communicators “might contain” closer processes (best effort from the MPI implementation)
    - An attribute on the communicator would tell if shared memory can be created on it

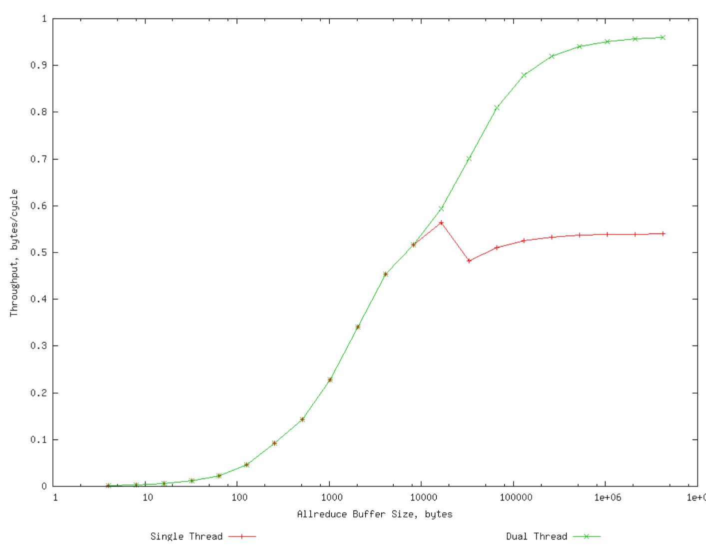
MPI Interoperability with Thread Teams

- Motivation: Allow coordination with the application to get access to threads, instead of the application maintaining a separate pool of threads than the MPI implementation
- Proposal in a nutshell
  - Thread teams are created
  - The user can provide the threads in the team to the MPI implementation to help out the MPI implementation
  - A user-defined team allows the user to have control on locality and blocking semantics for threads (i.e., which threads block waiting to help)

Point-to-point Communication on InfiniBand



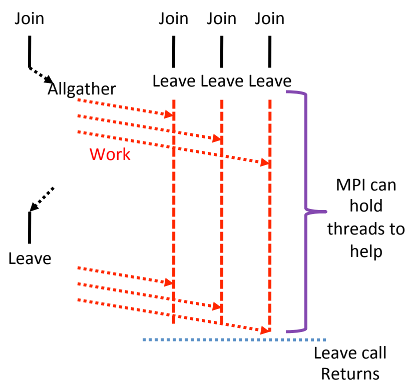
Allreduce on BG/P



Thread Teams Models

- Two models proposed
  - Synchronous model: all threads in the team synchronously join and leave the team – more restrictive for the application but has more optimization opportunity for MPI
  - Asynchronous model: threads join the team asynchronously; processes can leave synchronously or asynchronously break out
- Both models are essentially performance hints
  - Optimized MPI implementations can use this information

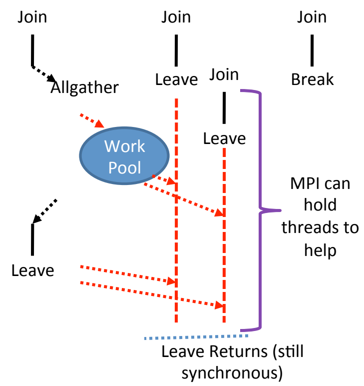
Thread Teams: Synchronous Model



Possible Implementation

- In the synchronous model, the Join and Leave calls can be synchronizing between the threads
  - The MPI implementation can assume that all threads will be available to help
  - The programmer should keep the threads synchronized for good performance
- The MPI implementation knows how many threads are going to help, so it can statically partition the available work, for example
- This model does not allow for threads to “escape” without doing a synchronous leave

Thread Teams: Asynchronous Model



Possible Implementation

- In the asynchronous model, the Join call is not synchronizing between the threads
- The leave call is still synchronizing, but threads are allowed to either help using "leave" or "break out" without helping
- The MPI implementation does not know how many threads are going to help
- This model allows for threads to "break out" without doing a synchronous leave

Thread Teams Proposed API

- Team creation/freeing
  - MPI_Team_create(team_size, info, &team)
    - One predefined info key for "synchronous"; default is "asynchronous"
    - Similar to the MPI RMA chapter in that the info arguments are true assertions; if the user says "synchronous" and tries to break out, that's an erroneous program
  - MPI_Team_free(team)
- Team join/leave functionality
  - MPI_Team_join(team)
    - A thread can only join one team at a time
  - MPI_Team_leave(team)
  - MPI_Team_break(team)

Hartree-Fock Example

```
do {
        One_Electron_Contrib(Density, Fock)
        while (task = next_task()) {
                {i, j, k} = task.dims
                X = Get(Density, {i,j,k} .. {i+C,j+C,k+C})
#pragma omp parallel {
                Y = Work({i,j,k}, X)                    ; <------ compute intensive
                omp_barrier()
                MPI_Team_join(team)
                if (omp_master) {
                        Accumulate(SUM, Y, Fock, {i,j,k}, {i+C,j+C,k+C}) ;    <----- communication intensive
                } ; OMP master
                MPI_Team_leave(team)
} ; OMP – end parallel block
        }
#pragma omp parallel {
        Update_Density(Density, Fock)           ; <----- communication intensive
        my_energy = omp_gather()
        MPI_Team_join(team)
        if (omp_master) {
                energy = MPI_Allgather(my_energy)       ; <----- moderately communication intensive
        } ; OMP master
        MPI_Team_leave(team)
} ; OMP – end parallel block
} while (abs(new_energy - energy) > tolerance)
```