Proposal to "MPI3: Hybrid Programming" for support of horizontal parallelism
(a.k.a. Helper Threads)
V0.7 July 8+, 2010
mpi3-hybridpm@lists.mpi-forum.org

Douglas Miller, dougmill@us.ibm.com

INTRODUCTION

One of the challenges with modern high-performance computing hardware is how to utilize larger numbers of Processing Elements (cores or threads) per node. Modern desktop computers may have 8 or more cores, while HPC systems are being planned for even larger numbers.

One area where multiple cores may improve performance is in communications. The simplest method is to load-balance communications across multiple hardware resources, using multiple PEs to parallelize the work. This "vertical parallelism" can be as simple as round-robin assignment of communications to resources, using some software mechanism to distribute among threads. This can also include message striping, if the application divides the data into separate communications. Another need is for "horizontal parallelism" where multiple PEs participate in the same communication, perhaps performing different roles. This can include striping, if the striping is done by the communications layer without direct involvement of the application.

The problem comes down to getting PEs applied in sufficient numbers to parallelize the work. One way is for the communications layer to spawn and manage SW threads onto PEs. But this may interfere with the application's use of PEs (if it results in over-subscription of the PEs). Since the application often goes through phases of computation and communication, it makes sense for the computation threads, now idle while waiting for communications, to apply their PEs to the communications. This can also include such paradigms as OpenMP, where a section of computation may be parallelized but the communications is done single-thread. This proposal is for a method for applications to "lend" their PEs to the communications layer during such times.

The idea is that these "available" threads join a "team", and any members of that team that initiate communication may spread out that communication across any/all members of the team. This results in communications being performed (or assisted) by multiple threads. As a trivial example, if communications involved non-contiguous datatypes, one thread might perform the packing of the data, another the communications, and a third the unpacking, all in a pipelined fashion. Such an example case would benefit from three threads (team members) even though only one communication resource existed.

Any prior association of a thread with communication resources (by an implementation), either implicit or explicit, would result in that communications resource being a part of the team. For example, if a platform had 4 network adapters and 4 threads joined a team, then the adapters might be assigned one-each to the threads internally. Or if an implementation had more-explicit association between threads and resources, those associations would be carried along into the team (e.g. MPI3 Endpoints).

In other words, a team represents both the thread(s) and the communication resource(s) the thread would use if it made MPI calls (e.g. MPI3 Endpoints). This is equivalent to the MPI3 Hybrid Programing proposal term "Agents" (threads attached to endpoints). In such a scheme, a Helper Team would be a collection of Agents. In MPI2, a team would be the threads and communication resources, if any, associated with the threads. If an MPI2 implementation had only one communication resource per process, then the team would consist of that one communication resource plus all member threads. An implementation might have some sort of work list associated with each team member such that blocking MPI calls would execute the work on the list while waiting for the completion of the particular MPI call.

In order for a thread (and it's resources) to be used in a communication, that thread must have called JOIN (as well as having done a TEAM_CREATE). By calling JOIN, a thread is making a commitment to also call LEAVE. If a team member blocks in any MPI call after JOIN (up to and including LEAVE), the thread becomes available to accelerate communications initiated by other members in the team. In some cases a member would call LEAVE immediately after JOIN (perhaps with a barrier between, if needed for the application usage model).

PROPOSED API

MPI_HELPER_TEAM - Opaque handle for a team of helper threads.

C binding:	typedef int MPI_Helper_team
FORTRAN:	INTEGER
C++:	class MPI::Helper::Team

Note: MPI_HELPER_* calls are a separate class of functions and do not adhere to restrictions from the MPI_THREAD_* model. This means that multiple threads in the same process may make these calls regardless of the threading model used to initialize MPI (e.g. MPI_INIT_THREAD). Communications calls (all other MPI calls) made within a JOIN/LEAVE block must still adhere to the threading model established during MPI_INIT (MPI_INIT_THREAD, etc). In other words, all MPI_HELPER_* functions are implicitly thread-safe.

MPI_HELPER_TEAM_CREATE(team_id, team_size, team)

IN	team_id	locally unique identifier of team to join
IN	team_size	total number of members in team
OUT	team	handle describing team

```
int MPI_Helper_team_create(int team_id, int team_size, MPI_Helper_team *team)
```

```
MPI_HELPER_TEAM(TEAM_ID, TEAM_SIZE, TEAM, IERROR)
        INTEGER TEAM_ID, TEAM_SIZE, IERROR
        INTEGER TEAM
```

```
static MPI::Helper::Team MPI::Helper::Team::Create(int team_id, int team_size)
```

The function creates a team to be used with subsequent JOIN calls. The call is made by all members of the team, but is non-blocking. *team_id* is an integer that is unique among all currently created teams, but may be re-used as long as the previous team using that ID was destroyed. [if the function is blocking and collective over members of the team, the need for *team_id* could be eliminated] An implementation may associate communications resources with the team at this time, or at JOIN.

For the purpose of error handlers, MPI_HELPER_TEAM_CREATE is associated with MPI_COMM_WORLD.

MPI_HELPER_TEAM_FREE(team)
    IN,OUT   team   handle describing team

int MPI_Helper_team_free(MPI_Helper_team *team)

MPI_HELPER_TEAM_FREE(TEAM, IERROR)
    INTEGER TEAM, IERROR

static void MPI::Helper::Team::Free(MPI::Helper::Team &team)

The function destroys the team. This function is called by all entities that created the team. It is non-blocking. It is an error to destroy a team while joined.

For the purpose of error handlers, MPI_HELPER_TEAM_FREE is associated with MPI_COMM_WORLD.

MPI_HELPER_JOIN(team)
    IN   team   handle describing team

int MPI_Helper_join(MPI_Helper_team team)

MPI_HELPER_JOIN(TEAM, IERROR)
    INTEGER TEAM, IERROR

static void MPI::Helper::Join(MPI::Helper::Team team)

The function registers the calling thread (and any associated resources) as an active participant in the team. The caller's resources may now be used by communications started by other members of the team. A thread may only be active in one team at a time.

Since JOIN is non-synchronizing it is possible for a thread to call JOIN and begin performing communications before other threads have called JOIN. In this case the communication(s) that were initiated before other members called JOIN may not be able to use all team members.

> *Advice to implementers:* An implementation may choose to defer the execution of a communication until after all (necessary) members call JOIN, in which case the communication may not actually complete until the rest of the members call LEAVE or make some other blocking MPI call. *(End of advice to implementers)*
>
> *Advice to users:* Because JOIN is non-synchronizing and a given communication may start before all members join the team, programmers may choose to perform a team-wide barrier after calling JOIN to ensure that all threads are available when communications are initiated. *(End of advice to users)*

For the purpose of error handlers, MPI_HELPER_JOIN is associated with MPI_COMM_WORLD.

MPI_HELPER_LEAVE(TEAM)

int MPI_Helper_leave(MPI_Helper_team team)

MPI_HELPER_LEAVE(TEAM, IERROR)
    INTEGER TEAM, IERROR

static void MPI::Helper::Leave(MPI::Helper::Team team)

The function completes all outstanding communications for the team, then dissolves the team association with the resources. It is blocking and collective over the members of the team. In some cases, on some implementations, this call may behave like a barrier.

Team members that initiate communications must handle completion of those communications normally, before calling LEAVE. [text removed] The presence or absence of JOIN-LEAVE should not affect the correctness of the communications performed between JOIN-LEAVE.

> *Advice to implementers:* If an implementation can determine conclusively that all communications have been started and have completed, then a given call to LEAVE may pass-through without blocking or synchronizing. In cases where it is known that a given team member will not be given any work, that team member's call to LEAVE may be a NO-OP. *(End of advice to implementers)*

For the purpose of error handlers, MPI_HELPER_LEAVE is associated with MPI_COMM_WORLD. Note, however, that communications being performed by a team member, whether in LEAVE or some other blocking MPI call, will report errors to the originating thread and be associated with the communicator used for that communication.

[The following is a potential convenience/optimization but needs advantages to be shown]
MPI_HELPER_FENCE(TEAM)

int MPI_Helper_fence(MPI_Helper_team team)

MPI_HELPER_FENCE(TEAM, IERROR)
    INTEGER TEAM, IERROR

static void MPI::Helper::Fence(MPI::Helper::Team team)

The function completes all outstanding communications for the team, similar to MPI_HELPER_LEAVE, but remains joined to the team. It is blocking and collective over the members of the team. In some cases, on some implementations, this call may behave like a barrier. The same rules for explicit completion apply to FENCE as to LEAVE, each thread must separately ensure completion of any communication it starts. FENCE is conceptually identical to calling LEAVE followed by JOIN on the same team. FENCE is intended as a convenience and possible optimization over calling LEAVE-JOIN.

Usage and Examples

These functions denote a boundary for a region of horizontal parallelism. All communications performed within this region (by the members) may use all member resources (and threads) to perform the communication(s). The thread on which a communications is started (the actual communications call occurs) is the primary thread for that operation. It is expected that the destination, or remote members of the communicator, are also the primary threads for their respective remote nodes.

Any communication performed outside (without) JOIN/LEAVE will utilize only the calling thread and resources. It is an error if all participants are not similarly involved, i.e. all must be in a JOIN/LEAVE or none are in JOIN/LEAVE.

Example 1: OpenMP program with distinct compute/communicate phases

A simple example where one thread performs an allreduce but the rest of the threads lend themselves as helpers. The omp barrier is to ensure that all endpoints are available when the allreduce begins.

```
#pragma omp parallel num_threads(N) {
      /* ... other thread setup */
      t = omp_get_thread_num();
      MPI_Helper_team team;
      MPI_Helper_team_create(0, omp_get_num_threads(), &team);
      /*
       * some computation may occur here...
       *
       * then a communications phase begins:
       */
      MPI_Helper_join(team);
      #pragma omp barrier
      if (t == 0) {
            MPI_Allreduce(...);
      }
      MPI_Helper_leave(team);
      /*
       * more computation and/or communication
       */
      MPI_Helper_team_free(&team);
      /* ... other thread tear-down */
}
```

Example 2: Pthreads program with distinct compute/communicate phases

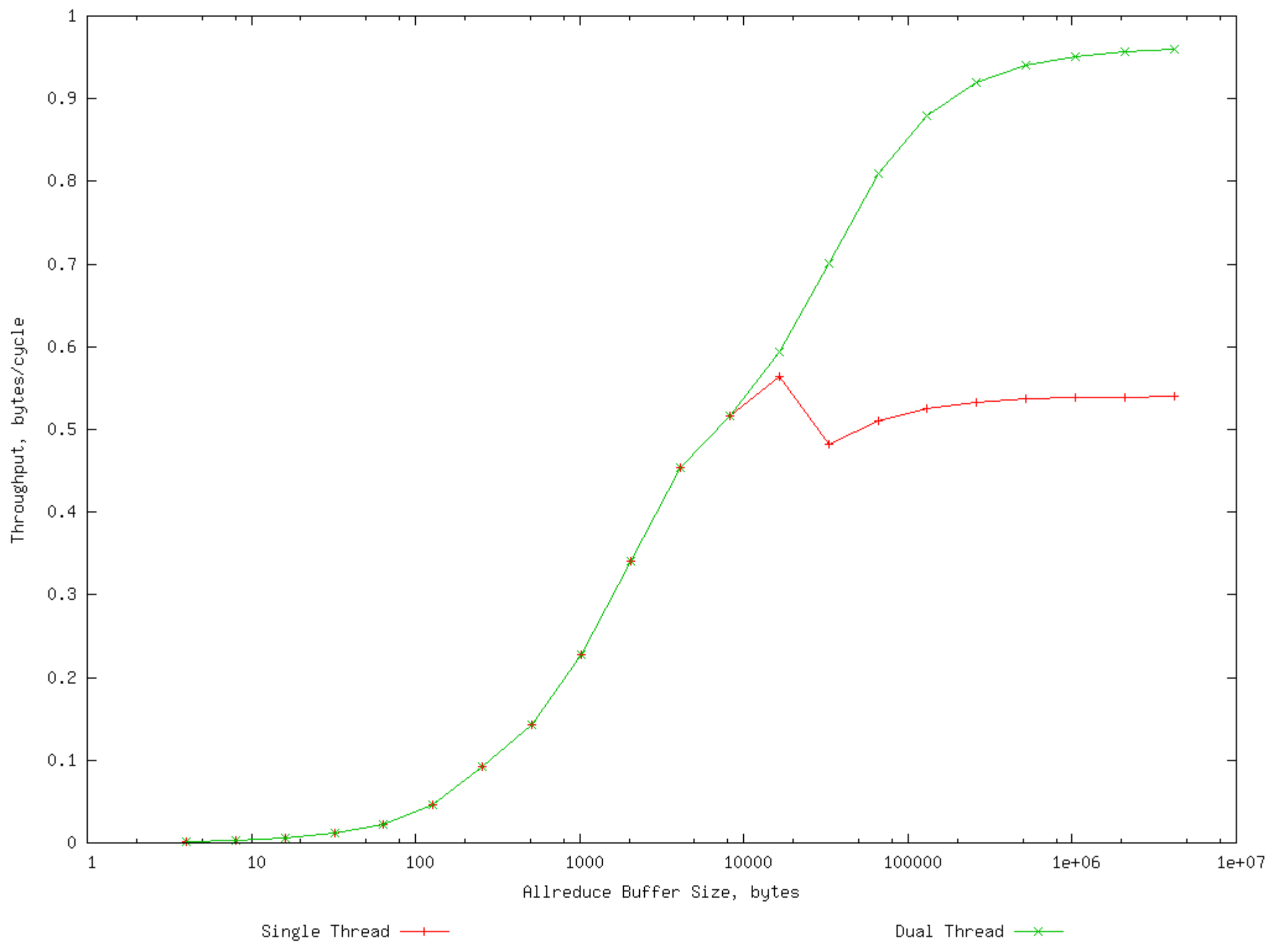
```
main(...) {
      ...
      /* N is the total number of threads to participate */
      for (x = 0; x < N - 1; ++x) {
            pthread_create(..., compute_only, ...);
      }
      compute_comm(N);
      ...
}
```

```
compute_only(...) {
      MPI_Helper_team team;
      MPI_Helper_team_create(0, N, &team);
      while (!done) {
            /*
             * perform computation phase here...
             *
             * then a communications phase begins:
             */
            MPI_Helper_join(team);
            pthread_barrier_wait(...);
            MPI_Helper_leave(team);
      }
      MPI_Helper_team_free(&team);
}

compute_comm(...) {
      MPI_Helper_team team;
      MPI_Helper_team_create(0, N, &team);
      while (!done) {
            /*
             * perform computation phase here...
             */
            MPI_Helper_join(team);
            pthread_barrier_wait(...);
            MPI_Allreduce(...);
            /* other communications... */
            MPI_Helper_leave(team);
      }
      MPI_Helper_team_free(&team);
}
```

Appendix A: Performance justification for using multiple threads in communications

As an example, the Collective Network on BlueGene BG/P systems is software driven. Maximum throughput cannot be achieved with less than 2 threads. The following graph shows single-thread and dual-thread performance of MPI_INT/MPI_SUM MPI_Allreduce on the BG/P Collective Network. The graph was produced by making two runs of a simple Allreduce performance test, one run as the default (no environment variables, which results in two comm-threads being automatically spawned for large messages), and the other with the environment variable DCMF_THREADED_TREE=0 to disable all use of comm-threads – which results in only the main program thread performing all transfers.



Graph 1: BG/P Coll Net comm-threads vs. main-only

Note, the sudden drop-off at 16k-32k bytes for single-thread is due to resource-contention affects of trying to handle both injection and reception FIFOs at the same time, in the same core. That is the point, timing wise, where the reception packets start arriving and result in both reception and injection activities going on together, dividing the core's resources between the two.