Motivation

- Advances in computing power are currently being achieved by adding more Processing Elements (cores and/or hardware threads) per chip.
  - This necessitates a programming paradigm shift towards multi-threading, both for users of MPI as well as implementers of MPI.
- Impacting this is the need in HPC to avoid over-subscribing Processing Elements (PEs) in order to eliminate OS overheads for context-switching.
  - This makes it objectionable for libraries (including MPI) to spawn and use their own threads without the knowledge, or consent, of the user.

Proposed Solution

- This proposal is an attempt to increase the cooperation between users and implementers of MPI with respect to use of threads.
  - The idea being that the application writer knows best how, and when, threads should be used.
  - If the application writer is given a mechanism to communicate it's use of threads (or rather, the availability of existing, idle, threads) to MPI then an MPI implementation can make use of those threads with minimal impact to the application.

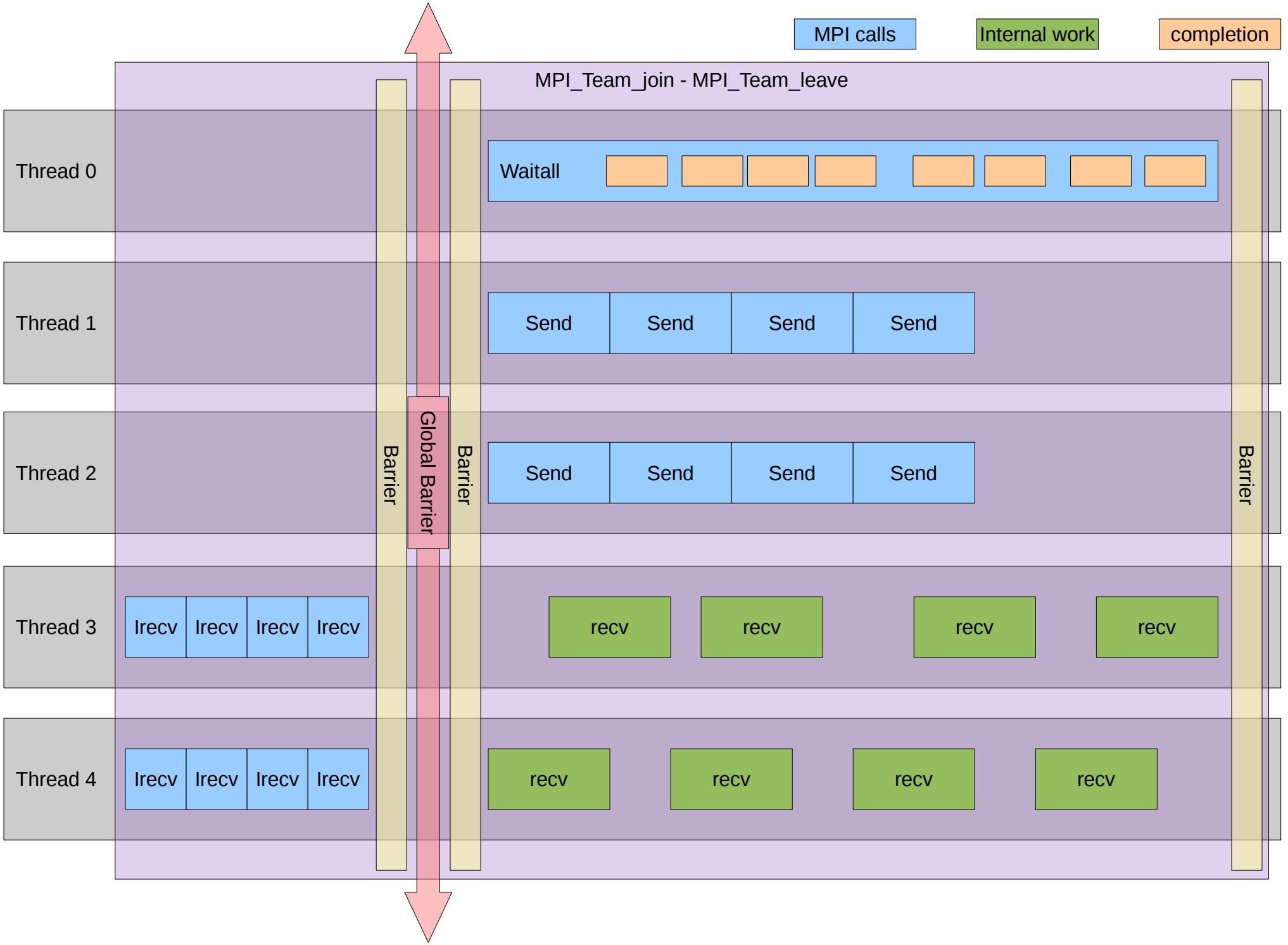
Summary

Note: the MPI_TEAM functions do not provide any functionality to a program, they simply provide information to an MPI implementation about threads (or threads attached to MPI Endpoints) that may be used to accelerate MPI operations.

This draws a parallel to OpenMP, where the OpenMP constructs do not alter the functionality of a program they simply allow an OpenMP implementation to (optionally) apply threads to work on certain sections of code. The MPI_TEAM functions provide the same sorts of hints to an MPI implementation, to be used or ignored as decided by the implementation.

Backup Slides

- An Example for Endpoints and Helper Threads
- Teams and Endpoints work well together, although they do not require each other to be useful.
  - However, each does enhance the other.


MPI calls
Internal work
completion
MPI_Team_join - MPI_Team_leave
Thread 0
Waitall
Thread 1
Send
Send
Send
Send
Thread 2
Send
Send
Send
Send
Thread 3
Irecv
Irecv
Irecv
Irecv
recv
recv
recv
recv
Thread 4
Irecv
Irecv
Irecv
Irecv
recv
recv
recv
recv
Barrier
Global Barrier
Barrier
Barrier

```
MPI_Team team;
MPI_Info info;
MPI_Info_create(&info);
MPI_Info_set(info, "balanced", "true");
MPI_Team_create(omp_get_thread_limit(), info, &team);
MPI_Request reqs[NITER * NNBOR];
MPI_Status stss[NITER * NNBOR];
MPI_Comm_size(MPI_COMM_CLIQUE, &nep);
assert(nep >= NNBOR * 2);
#pragma omp parallel num_threads(NNBOR * 2 + 1), shared(team,reqs,stss)
{
        t = omp_get_thread_num();
        if (t > 0) MPI_Thread_attach(t – 1, MPI_COMM_CLIQUE);
        MPI_Team_join(omp_get_num_threads(), team);
        if (t == 0) {
#               pragma omp barrier
                MPI_Team_sync(team);
                MPI_Waitall(NITER * NNBOR, &reqs, &stss);
        } else if (t <= NNBOR) {
#               pragma omp barrier
                if (t == 2) MPI_Barrier(MPI_COMM_WORLD);
                MPI_Team_sync(team);
                for (i = 0; i < NITER; ++i) {
                        MPI_Send(...);
                }
        } else if (t <= NNBOR * 2) {
                for (i = 0; i < NITER; ++i) {
                        n = (t – NNBOR - 1) * NITER + i;
                        MPI_Irecv(..., &reqs[n]);
                }
#               pragma omp barrier
                MPI_Team_sync(team);
        } else {
                /* should never happen - either abort or just cope... */
#               pragma omp barrier
                MPI_Team_sync(team);
        }
        MPI_Team_leave(team);
}
MPI_Team_free(team);
```

Notes on Example

- Access to handles is by the Team.
  - A request created by a thread in the team may be waited-on, and completed, by different thread(s) (attached to a different endpoint).
  - The team controls which threads perform which tasks related to completion of the request.
- Both “omp barrier” and MPI_Team_sync/_leave provide synchronization of the threads in the team.
  - In the diagram, “barrier” does not specify which is used.

Example for TBB

```
class ThreadedAllreduce {
    const MPI_Team _team;
    const int _size;
public:
    int result;

    void operator() (const blocked_range<int> &r) {
        int t = r.begin();
        // assert(r.end() - r.begin() == 1) ?
        MPI_Thread_attach(t, MPI_COMM_CLIQUE);
        MPI_Team_join(_size, _team);
        if (t == 0) {
            MPI_Allreduce(sendbuf, recvbuf,
                          count, datatype, op,
                          MPI_COMM_WORLD);
        }
        else {
            // The remaining threads go directly
            // to MPI_Team_leave
        }
        MPI_Team_leave(_team);
        MPI_Thread_attach(0, MPI_COMM_CLIQUE);
        result = 0;
    }

    ThreadedAllreduce(ThreadedAllreduce &x, split) :
        _team(x._team),
        _size(x._size),
        result(1)
    {}

    void join(ThreadedAllreduce &y) {
        result |= y.result;
    }

    ThreadedAllreduce(MPI_Team team, int size) :
        _team(team),
        _size(size),
        result(1)
    {}
}
```

```
int main(int argc, char **argv) {
    int provided, N;
    MPI_Team team;
    MPI_Info info;

    MPI_Init_thread(&argc, &argv,
                    MPI_THREAD_MULTIPLE, &provided);
    MPI_Comm_size(MPI_COMM_CLIQUE, &N);

    MPI_Info_create(&info);
    MPI_Info_set(info, "balanced", "true");
    MPI_Team_create(N, info, &team);
    MPI_Info_free(&info);

    ThreadedAllreduce thar(team,N);
    parallel_for(blocked_range<int>(0,N,1), thar,
                                simple_partitioner());

    MPI_Team_free(&team);
    return thar.result;
}
```