12.4.4 Helper Team Functionality

Motivation

With the end of Moore's Law regarding processor speed, advances in processing power are currently being achieved by adding more “Processing Elements” (cores and/or hardware threads) per chip. This necessitates a programming paradigm shift towards multithreading, both for users of MPI as well as implementers. Impacting this is the common practice in HPC to avoid over-subscribing Processing Elements (PEs) in order to eliminate OS overheads for context-switching, making it objectionable for libraries, including MPI, to spawn threads without the knowledge, or consent, of the user and possibly even preempting critical user threads.

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.

Helper Teams

The following functions provide mechanisms to allow the MPI implementation to parallelize its internal processing. These functions may be used for an application to temporarily hand-over control of its threads for the MPI implementation to use. These functions allow the application to create teams of threads, and use these teams to perform the processing required by the MPI operations initiated by one or more of the threads in the team.

MPI_TEAM_CREATE(team_size, info, team)
IN team_size total number of members in team (integer)
IN info info (handle)
OUT team handle describing team (handle)

int MPI_Team_create(int team_size, MPI_Info info, MPI_Team *team)

MPI_TEAM_CREATE(TEAM_SIZE, INFO, TEAM, IERROR)
INTEGER TEAM_SIZE, INFO, TEAM, IERROR

This call creates a team of helper threads to be used with subsequent JOIN-LEAVE or JOIN-BREAK calls. This call must only be made by one thread. It is not required for the thread creating a team to join the team.

The info argument provides optimization hints to the runtime about the expected usage pattern of the threads team. The following info keys are predefined:

balanced — if set to true, then all threads in the team will be calling MPI_TEAM_JOIN and MPI_TEAM_LEAVE.

The MPI implementation may synchronize between all the threads in the team in an MPI_TEAM_JOIN or MPI_TEAM_LEAVE call, and assume that no thread in the team would call MPI_TEAM_BREAK on that team. The MPI implementation

may thus expect all threads to be available to participate in communications.

MPI_TEAM_FREE(team)
INOUT team handle describing team (handle)

int MPI_Team_free(MPI_Team *team)

MPI_TEAM_FREE(TEAM, IERROR)
INTEGER TEAM, IERROR

This call frees the team object team and sets the team handle to MPI_TEAM_NULL.

This call must be made by only one thread. It is not required for the same thread that created this team to free it. MPI_TEAM_FREE(team) can be invoked by a thread only after it has completed its involvement in MPI communications initiated while it had joined the team team: i.e., the thread has called MPI_TEAM_LEAVE or MPI_TEAM_BREAK on the team, before it can free the team.

Advice to users. Users should be careful to not free a team handle while other threads are operating on it (e.g., using MPI_TEAM_JOIN). *(End of advice to users.)*

Advice to implementors. Implementors should be careful not to free team resources before all the threads in the team have either called MPI_TEAM_LEAVE or MPI_TEAM_BREAK, even if one of the threads calls MPI_TEAM_FREE on the handle, possibly by internally reference counting on the handle. *(End of advice to implementors.)*

MPI_TEAM_JOIN(team)
IN team handle describing team (handle)

int MPI_Team_join(MPI_Team team)

MPI_TEAM_JOIN(TEAM, IERROR)
INTEGER TEAM, IERROR

This call registers the calling thread as a participant in the team, indicating that the thread will eventually call a BREAK or LEAVE. A thread may only participate in one team at a time.

Advice to users. If the team is created with the info argument set to balanced, the MPI implementation might treat the MPI_TEAM_JOIN call as a \contract" that this thread will be available to help MPI operations initiated by other members of the team (including itself), while maintaining the local/non-local semantics of the MPI operations. *(End of advice to users.)*

MPI_TEAM_LEAVE(team)
IN team handle describing team (handle)

int MPI_Team_leave(MPI_Team team)

MPI_TEAM_LEAVE(TEAM, IERROR)
INTEGER TEAM, IERROR

This call deregisters the calling thread from being a participant in the team. A thread can exit from the MPI_TEAM_LEAVE call only after all threads in the team have either called

MPI_TEAM_LEAVE or MPI_TEAM_BREAK.

Advice to users. The MPI implementation may choose to synchronize all threads in the team that have not called MPI_TEAM_BREAK during the MPI_TEAM_LEAVE call, to effectively utilize all resources for MPI operations initiated by the team members. *(End of advice to users.)*

MPI_TEAM_BREAK(team)
IN team handle describing team (handle)

int MPI_Team_break(MPI_Team team)

MPI_TEAM_BREAK(TEAM, IERROR)
INTEGER TEAM, IERROR

This call allows a thread to deregister itself from being a participant in the team, without synchronizing with other threads in the team. If the balanced info key is set, the user is not allowed to deregister using the MPI_TEAM_BREAK call.

12.4.5 Examples

Example 12.3 The following example shows OpenMP code that uses multiple threads to help MPI communication using MPI_ALLREDUCE initiated by one thread.

```
...
MPI_Team team;
MPI_Team_create(0, N, MPI_INFO_NULL, &team);
#pragma omp parallel num_threads(N) {
      ...
      t = omp_get_thread_num();
      /* some computation and/or communication */
      MPI_Team_join(team);

      if (t == 0) {
            MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm);
      }
      else {
            /* The remaining threads directly go to MPI_Helper_team_leave */
      }
      MPI_Team_leave(team);
      /* more computation and/or communication */
}
MPI_Team_free(&team);
```

Example 12.4 The following example shows OpenMP code that uses multiple threads to help MPI communication initiated by some threads.

```
...
MPI_Team team;
MPI_Team_create(0, N, MPI_INFO_NULL, &team);
#pragma omp parallel num_threads(N) {
      ...
      t = omp_get_thread_num();
      /* some computation and/or communication */
      MPI_Team_join(team);
      if (t == 0) {
            MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm1);
      }
      else if (t == 1) {
            MPI_Bcast(buffer, count, datatype, root, comm2);
      }
      else if (t == 2) {
            MPI_Send(buf, count, datatype, dest, tag, comm3);
      }
      else {
            /* The remaining threads directly go to MPI_Helper_team_leave */
      }
      MPI_Team_leave();
      /* more computation and/or communication */
}
MPI_Team_free(&team);
```

We need examples with: (1) info argument set, and (2) using TEAM_BREAK.

Tutorial: Parallelizing existing codes

Consider the following pseudo-code:

```
energy = 0.0
do {
      ; Calculate one-electron contributions to Fock
      One_Electron_Contrib(Density, Fock)
      ; Calculate two-electron contributions to Fock
      while (task = next_task()) {
            {i, j, k} = task.dims
            X = Get(Density, {i,j,k} .. {i+C,j+C,k+C})
            ; Cost of O(N^2) to O(N^4)
            Y = Work({i,j,k}, X) ; <------ compute intensive
            Accumulate(SUM, Y, Fock, {i,j,k}, {i+C,j+C,k+C})
      }
      ; Update the Density matrix for next iteration
      Update_Density(Density, Fock)  ; <----- communication intensive
      energy = Gather_Energy()       ; <----- communication intensive
} while (abs(new_energy - energy) > tolerance)
```

The first step, and the most obvious to an applications developer, is to parallelize the computation:

```
energy = 0.0
do {
      ; Calculate one-electron contributions to Fock
      One_Electron_Contrib(Density, Fock)
      ; Calculate two-electron contributions to Fock
      while (task = next_task()) {
            {i, j, k} = task.dims
            X = Get(Density, {i,j,k} .. {i+C,j+C,k+C})
            ; Cost of O(N^2) to O(N^4)
#pragma omp parallel {
            Y = Work({i,j,k}, X) ; <------ compute intensive
} ; OMP – end parallel block
            Accumulate(SUM, Y, Fock, {i,j,k}, {i+C,j+C,k+C})
      }
      ; Update the Density matrix for next iteration
      Update_Density(Density, Fock)  ; <----- communication intensive
      energy = Gather_Energy()       ; <----- communication intensive
} while (abs(new_energy - energy) > tolerance)
```

However, in the above code there are threads that are idle during the communications part of the code. Since the computation phase requires all data to be updated before beginning, there is implied fork and join (synchronization – which OMP actually provides), and the communications phase is single-threaded with all other available PEs idle. Note that, for the code outside of the OMP #pragma, all additional threads (PEs) are idle and simply waiting for the master thread to reach the OMP #pragma again.

This code could be modified to inform the MPI layer that these threads are otherwise idle and may be used if needed:

```
MPI_Team team;
MPI_Info info;
MPI_Info_create(&info);
MPI_Info_set(info, "balanced", "true");
MPI_Team_create(&team, &info, nthr);
energy = 0.0
do {
      ; Calculate one-electron contributions to Fock
      One_Electron_Contrib(Density, Fock)
      ; Calculate two-electron contributions to Fock
      while (task = next_task()) {
            {i, j, k} = task.dims
            X = Get(Density, {i,j,k} .. {i+C,j+C,k+C})
            ; Cost of O(N^2) to O(N^4)
#pragma omp parallel num_threads(nthr) {
            Y = Work({i,j,k}, X) ; <------ compute intensive
} ; OMP – end parallel block
            Accumulate(SUM, Y, Fock, {i,j,k}, {i+C,j+C,k+C})
      }
      ; Update the Density matrix for next iteration
#pragma omp parallel num_threads(nthr) {
      MPI_Team_join();
      if (omp_master) {
            Update_Density(Density, Fock)  ; <----- communication intensive
            energy = Gather_Energy()       ; <----- communication intensive
      }
      MPI_Team_leave(team);
} ; OMP – end parallel block
} while (abs(new_energy - energy) > tolerance)
MPI_Team_free(&team);
```

In the above code, The calls to Update_Density() and Gather_Energy() are performed by the master thread only, and would initiate (and complete) all necessary communications. The non-master threads would all go directly to the MPI_TEAM_LEAVE call and would wait there for the master thread to also leave. While waiting the non-master threads are executing in the MPI library and are available to help the master thread in MPI operations.

The code could be written to have a single OMP #pragma at the beginning and handle all single-threaded activities appropriately, but that is beyond the scope of this tutorial.