MPI3: The Behavior of MPI Processes Sharing Memory

Hybrid Programming Working Group

October 17, 2011

0.0.1 Introduction

Rationale

It is often desirable to have more than one MPI process per node. For example, message matching time tends to increase linearly with the number of pending requests; the problem is inherent to the matching semantics of MPI. Multiple MPI processes will usually support a higher message rate. The problem becomes more acute as the number of concurrent threads per node increases. Similarly, a large node may have multiple network interfaces; using them for one MPI process will increase MPI overheads and may result in superfluous communication in a NUMA system.

On the other hand, it is often desirable to use shared memory for intra-node communication, as this reduces communication overheads and avoids the replication of shared data. This will lead to a situation where multiple MPI processes have partially or totally overlapping address spaces.

This hybrid model (multiple MPI processes with partially or totally overlapping address spaces) also facilitates the efficient support of hybrid programming models such as MPI+OpenMP or MPI+PGAS: One may desire one OpenMP program with multiple MPI processes, or one UPC program where each UPC thread is an MPI process, but multiple UPC threads run in the same address space.

The MPI standard does not preclude such a situation. It says [3, §2.7]:

> An MPI program consists of autonomous processes, executing their own code, in a MIMD style. The codes executed by each process need not be identical. The processes communicate via calls to MPI communication primitives. Typically, each process executes in its own address space, although shared-memory implementations of MPI are possible.

In fact, there are MPI implementations where an MPI process is mapped onto an OS thread [1, 5], or even a task that can be dynamically scheduled by a run-time on different threads, for load balancing [2].

In such a hybrid model, threads may be scheduled to run on different cores, and cores may have access to the physical communication resources (e.g., network interface registers) of different MPI processes. One could have a symmetric system, where a thread running on any core could use any MPI resource; on the other hand the system architecture may preclude or penalize some mappings, in an asymmetric configuration. We intend to support both cases.

The design of MPI precludes simultaneous use by a thread of multiple ranks within a communicator: A thread can belong to only one MPI process at a time. Thus, a mechanism is needed to associate a thread with a specific MPI process. A computation may start with only one thread running (e.g., in the case of OpenMP); thread spawning and scheduling will not be controlled by MPI. Thus, it is necessary to provide a mechanism for including an existing thread into an MPI process. We provide such a mechanism in this section.

Once a mechanism is provided to associate a thread with an MPI process, it is a small step to allow for this association to change during execution. This will facilitate load balancing and facilitate the support of a model such as an OpenMP program with multiple MPI processes. We support dynamic association of threads with MPI processes/

Definitions

We use the following terminology

An MPI endpoint consists of a set of resources that enable MPI calls. Such an endpoint is identified in MPI by a rank in a communicator.

An MPI clique is a set of endpoints that can be accessed by the same collection of threads. I.e., if a thread can access an endpoint in a clique, then it can access any of the other endpoints in the clique.

An MPI process consists of an MPI endpoint and a set of threads that can perform MPI calls using that endpoint. All references to a "process" in other parts of the MPI standard should be understood as referring to an "MPI process".

There is no necessary connection between MPI processes and any other notion of process that the system may support; thus, an "MPI process" could actually be associated with a OS thread and an OS process could contain multiple OS threads. Similarly, no assumptions are made about threads, except that (i) a thread that executes a blocking MPI call is "descheduled", i.e., cannot prevent the progress of other threads; and (ii) when the blocking MPI call is complete the blocked thread is eventually "rescheduled", i.e., resumes running the code following the call.

The mechanisms described in this section define the initial clique that a thread "belongs" to, and provide mechanisms for the thread to attach to any of the endpoints in the clique.

> *Rationale.* The main case we envisage is where each clique is associated with a distinct address space. However, the proposal does not require this to be the case. For example, one could have MPI endpoints mapped onto an address segment common to several address spaces. (*End of rationale.*)

0.0.2 Endpoints and Their Use

MPI Endpoints

An *MPI endpoint* is a (handle to a) set of resources that supports the independent execution of MPI communications. These can be physical resources (e.g., registers mapped into the address space of the process), or logical resources. An endpoint corresponds to a rank in an MPI communicator. A thread can be *attached* to an endpoint, at which point it can make MPI calls using the resources of the endpoint. A thread can be attached to at most one endpoint; the association is dynamic.

In MPI with no endpoints, there is a fixed one-to-one correspondence between MPI processes and ranks in MPI_COMM_WORLD; when a thread executes an MPI call with argument MPI_COMM_WORLD then the rank of the caller is the rank of the MPI process containing this thread.

Similarly, in our proposal, there is a one-to-one correspondence between MPI endpoints and ranks in MPI_COMM_ENDPOINTS; when a thread that is attached to an MPI endpoint executes an MPI call with argument MPI_COMM_ENDPOINTS then the caller's rank is the rank of the attached endpoint. Thus, if a thread is attached to endpoint 5, then a call by that thread to MPI_SEND(..., MPI_COMM_UNIVERSE) will appear as a send by the MPI process with rank 5 in MPI_COMM_ENDPOINTS. Similarly, if a thread executes

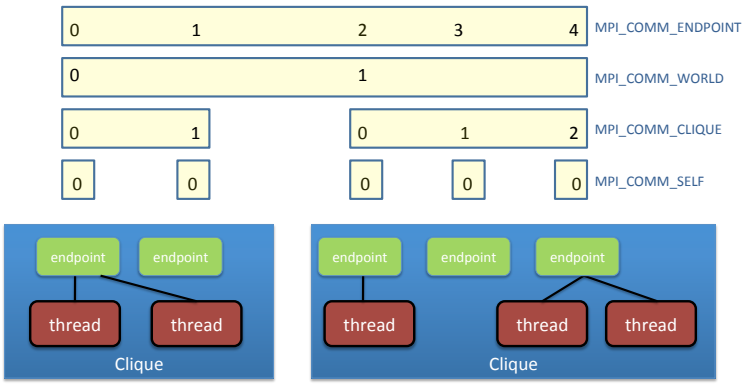

Figure 1: Communicators created st MPI initialization

```
MPI_Comm_Dup(MPI_COMM_ENDPOINTS, newcomm);
MPI_Send(..., newcomm);
```

then the send appears to be executed by the MPI process with rank 5 in `newcomm`.

> *Rationale.* We introduce a new initial communicator MPI_COMM_ENDPOINTS, rather than reusing MPI_COMM_WORLD in order to avoid the need for changes in current MPI codes. This is explained in more detail later in the section. (*End of rationale.*)

Initialization

The total number of available endpoints and their location are determined by the MPI job startup command, e.g., mpiexec; see Section 0.0.4.

A call to an MPI initialization routine MPI_INIT or MPI_THREAD_INIT generates four communicators:

MPI_COMM_ENDPOINTS communicator that includes all endpoints

MPI_COMM_WORLD communicator that includes one endpoint from each clique

MPI_COMM_CLIQUE separate communicator for the endpoints of each clique

MPI_COMM_SELF communicator that contains exactly one endpoint

Endpoints are ordered in the same order within MPI_COMM_ENDPOINTS ans MPI_COMM_CLIQUE; endpoints that belong to the same clique have consecutive ranks in MPI_COMM_ENDPOINTS. The MPI_COMM_WORLD communicator has one endpoint within each clique, the first endpoint in that clique. This is illustrated in Figure 1.

> *Rationale.* When each clique corresponds to a separate address space, then codes using MPI_COMM_WORLD will behave as they behave now in MPI libraries where each MPI process runs in a distinct address space. (*End of rationale.*)

An MPI program starts execution with at least one thread of control within each clique; the initialization call should be executed by all threads that run when MPI_INIT or MPI_THREAD_INIT are called.

> *Advice to users.* MPI_COMM_WORLD and derived communicators should be used by code that is unaware of multiple endpoints per process; MPI_COMM_ENDPOINTS and derived communicators should be used by code that is aware of multiple endpoints per process. (*End of advice to users.*)
>
> *Discussion.* The new communicator MPI_COMM_ENDPOINTS is required; the communicators MPI_COMM_WORLD and MPI_COMM_SELF are provided for compatibility. One could avoid creating MPI_COMM_CLIQUE at initialization, using, instead, a call to MPI_COMM_SPLIT_TYPE. (*End of discussion.*)

0.0.3 Thread Attachment

All threads that already run when MPI is initialized are attached to an endpoint in MPI_COMM_WORLD; each endpoint in MPI_COMM_WORLD must have an attached thread after MPI initialization. Any newly spawned thread is attached, by default, to the endpoint its parent is attached to.

MPI_THREAD_ATTACH(rank, comm)

IN	rank	rank of endpoint (integer)
IN	comm	communicator containing endpoint (handle)

```
int MPI_Thread_attach(int rank, MPI_Comm comm)
```

```
MPI_THREAD_ATTACH (RANK, COMM, IERROR)
    INTEGER RANK, COMM, IERROR
```

The function detaches the invoking thread from the endpoint it is currently attached and attaches it to the endpoint specified by the call arguments. The call is erroneous if the attached endpoint is not in the same clique as the detached endpoint.

A thread may attach to at most one endpoint (i.e., it can belong to at most one MPI process). The call outcome is undefined if the level of thread support is MPI_THREAD_FUNNELED and a thread is already attached to the endpoint.

The MPI_THREAD_ATTACH call is local. We discuss progress when an endpoint has no attached thread in Section 0.0.4.

An error of class MPI_ERR_ENDPOINTS is returned if the attach call fails.

> *Advice to users.* Note that a thread attaches to an endpoint, i.e., migrates to an MPI process. The endpoint can have different ranks in different communicators. Once attached to endpoint, the thread can communicate using any of these communicators. Thus, in the example of Figure 1, a thread in the second clique can execute the following code:

```
1 ...
2 MPI_Thread_attach(2, MPI_COMM_CLIQUE);
```

```
MPI_Send(..., MPI_COMM_CLIQUE);  // sender rank is 2
MPI_Send(..., MPI_COMM_ENDPOINTS);  // sender rank is 4
...
```

(*End of advice to users.*)

0.0.4 Communication With Endpoints

Unless said otherwise, whenever "process" is mentioned in the MPI standard, it should be understood to mean "MPI process", i.e., an endpoint and all the threads attached to that endpoint. The rules listed below follow from this interpretation.

A thread must be attached to an endpoint (i.e. must be part of an MPI process) in order to make MPI calls other than MPI_INIT, MPI_INIT_THREAD, MPI_GET_VERSION, MPI_INITIALIZED, MPI_FINALIZED and MPI_THREAD_ATTACH. A nonblocking call started on an endpoint must be completed on the same endpoint. The rules and restrictions specified by the MPI standard [3, §12.4] for threads continue to apply. In particular, a blocking MPI call will block the thread that executes the call, but will not affect other threads. The blocked thread will continue execution when the call completes. Since each thread can be attached to only one endpoint, deadlock situations do not arise. Two distinct threads should not block on the same request.

> *Advice to implementors.* The support of multiple MPI processes within one address space requires that the MPI library maintain, for each thread, the identity of the endpoint the thread is attached to, if any. No other changes are needed. Communication using the MPI_THREAD_FUNNELED model, with k endpoints in one clique, should be performing as well or better than communication with k single-threadedMPI processes, each with one endpoint each, using the same endpoints. (*End of advice to implementors.*)

Progress

The MPI standard specifies situations where progress on an MPI call at an agent might depend on the execution of matching MPI calls at other agents. Thus, a blocking send operation might not return until a matching receive is executed; a blocking call to a collective operation call might not return until the call is invoked by all other processes in the communicator; and so on. On the other hand, a non-blocking send or non-blocking collective will return irrespective of activities at other MPI processes.

These rules are extended to the situation where an OS process may have multiple endpoints: a blocking send on an endpoint might not return until a matching receive has occurred at the destination endpoint; and a blocking call to a collective operation might not return until the operation is invoked at all endpoints of the communicator (including endpoints at the same OS process). On the other hand, a non-blocking send or a non-blocking call to a collective operation will return, irrespective of the activities of threads attached to other endpoints (including threads in the same address space).

The same rules dictate progress when an endpoint has no attached thread. An endpoint with no attached thread might prevent progress of an MPI call if the progress of that call depends on the execution of a matching MPI call at that endpoint, but will not prevent progress of other MPI calls. Thus, a blocking send might not return if no thread is attached to the destination endpoint; a blocking call to a collective operation call might not return if

one of the endpoints in the communicator has no attached thread. However, a non-blocking send will return even if there is no thread attached to the destination endpoint; and a non-blocking call to a collective operation will return even if one of the endpoints in the communicator has no attached thread.

Thread Support

If the level of thread support is MPI_THREAD_FUNNELED then only one thread can execute MPI calls on each endpoint; different threads within the same OS process can execute concurrently MPI calls on different endpoints. If the level of thread support is MPI_THREAD_SERIALIZED then no two threads can make concurrent MPI calls on the same endpoint.

Finalize

MPI_FINALIZE must be invoked once at each OS process. The call should be invoked only after all pending MPI calls at that process have completed. (This is an exception to the rule – we do not require a separate call for each MPI process.)

Memory Allocation

Memory allocated by MPI_ALLOC_MEM [3, §6.2] can be used only for communication with the endpoint the calling thread is attached to.

> *Rationale.* Endpoints may be supported by distinct adapters, each requiring different memory areas for efficient communication. (*End of rationale.*)

Error Handling

Error handlers are attached to endpoints. A communicator may have different error handlers attached to the different endpoints of that communicator within the same address space.

The same rule applies to error handlers attached to windows or files.

Process Manager Interface

The functionMPI_COMM_SPAWN function can be used to spawn OS processes with multiple endpoints, with the same number of endpoints at each OS process. The number of spawned OS processes is specified by the argument maxprocs. The number of endpoints per OS process is specified by the value of the reserved key num_endpoints in the info argument. If the key num_endpoints is not provided in the info argument, then the new communicator has one endpoint per OS process. The call returns an error of class MPI_ERR_SPAWN if it cannot generate the required number of endpoints per process. If it succeeds, then it returns an intercommunicator that contains the parent endpoints in the local group and the first endpoint of each spawned process in the remote group. This intercommunicator has, as local group, the endpoints of the comm argument of MPI_COMM_SPAWN, and, as a remote group, the endpoints of newly created MPI_COMM_WORLD. This intercommunicator will be returned at a newly spawned process by the call to MPI_GET_PARENT (for threads attached to the first endpoint of the OS process). One error code per OS process is returned in array_of_errcodes argument. The MPI initialization call at the newly spawned OS processes return four communicators, as specified in Section 0.0.2.

The function MPI_COMM_SPAWN_MULTIPLE is extended in a similar manner. The function is passed multiple array arguments. The values associated with the key num_endpoints in the i-th entry of the info array argument specifies the number of endpoints to generate in each of the processes that execute the i-th command.

Portable Process Startup

> *Advice to implementors.* If the implementation provides an mpiexec function then it is advised that the number of endpoints per OS process be indicated by the parameter -endpoints ¡ ¿. (*End of advice to implementors.*)

Windows

An invocation to MPI_WIN_CREATE(base, size, disp_unit, info, comm, win) may return a different window for different endpoints in the same address space. Windows associated with different endpoints in the same address space may overlap. However, the outcome of a code where conflicting accesses occur to a location that appears in two windows is undefined.

I/O

The invocation to MPI_FILE_OPEN returns a distinct file handle at each endpoint. Note that the function is collective and must be invoked at all endpoints of the communicator with the same file name and access mode arguments.

An invocation to MPI_FILE_SET_VIEW can set a different view of the file for each file handle argument (passing different disp, filetype or info arguments) – hence a different view at each endpoint within the same address space.

Data access calls that use individual file pointers (such as MPI_FILE_READ) maintain a distinct file pointer for each file handle; hence different endpoints within the same address space are associated with distinct individual file pointers.

0.0.5 Interoperability

This section discusses how the endpoint constructs can interact with other programming models, such as Posix threads, OpenMP and PGAS languages.

Posix Binding

An MPI library is compatible with a POSIX thread library if the behavior described in the previous section holds for threads spawned by the POSIX thread library.

example

We present below a simple "hello world" program. The program creates multiple endpoints at each process, and attaches one thread to each endpoint. The thread attached to endpoint zero gathers the tread ids of all the spawned threads and prints them.

Listing 1: Simple MPI hybrid program

```
1 #include <mpi.h>
2 #include <posix.h>
```

```
#include <stdio.h>
#define max_endpoints 32

pthread_t thread[max_endpoints];

/* code executed by each thread */
void *foo(void *id)
{
  int i = (int)id;
  long tid = (long)thread[i];
  int rank, size;
  long *recvbuf;

  MPI_Thread_attach(i, MPI_COMM_CLIQUE);
  MPI_Comm_size(MPI_COMM_ENDPOINTS, size);
  MPI_Comm_rank(MPI_COMM_ENDPOINTS, rank);
  if (rank == 0)
    recvbuf = (long *)malloc(size*sizeof(long));
  MPI_Gather(&tid, 1, MPI_LONG, &recvbuf, 1, MPI_LONG, 0,
                                        MPI_ENDPOINTS);
  if (rank==0)
  {
    printf("number of endpoints is %d; their thread ids are: ",
                                     size);
    for (int j=0; j<size; j++)
      printf(" %d ", *(recvbuf+j));
  }
  pthread_exit(NULL);
}

int main()
 {
  MPI_Init_endpoint(NULL, NULL, max_endpoints, max_thread_level,
  size, rank);

  /* create a thread for each endpoint */
  for (int i=0; i < max_endpoints; i++)
     pthread_create(&thread[i], NULL, foo, (void *)i);
  pthread_exit(NULL);
}
```

OpenMP

An MPI library is thread-compatible with OpenMP if the behavior described in the previous section holds for threads as defined in the OpenMP standard.

An MPI library is task-compatible with OpenMP if the behavior described in the previous section holds for threads corresponding to tasks as defined in the MPI standard.

0.0.6 PGAS languages

A UPC or Fortran implementation that is compatible with MPI will provide a mechanism for identifying which threads (in UPC) or images (in Fortran) are within the same address space. The MPI library should support the creation of at least one MPI endpoint per thread (in UPC) or image (in Fortran).

0.0.7 Porting Codes to Hybrid MPI

Current MPI codes usually assume that each MPI process runs in a distinct address space, so that, for example, static variables of code executing in different processes are distinct. Such a code ports without change, if it use one endpoint per clique and cliques are in distinct address spaces. The transition from current MPI codes to codes using multiple endpoints per clique can entail one or more of the following scenarios:

1. Code is written to utilize multiple endpoints per clique and leverage shared memory communication within the clique.

2. Libraries are written so that they can be invoked in parallel by one thread per endpoint; they compute correctly irrespective of the number of endpoints per clique. Such portability is important for MPI libraries invoked from UPC or Fortran: The library can have one MPI process per UPC thread (or Fortran image), and its behavior will not change if multiple UPC threads run within the same address space.

3. Code written to use a single endpoint per clique invokes a library written to use multiple endpoints per clique. This will enable recoding only critical kernels, with no changes to the overall program logic.

MPI code can be written so that it has the same outcome, whether each MPI process is in a distinct address space, or multiple MPI processes run within the same address space. The only part of the code that has to be aware of the system configuration (i.e., the number of MPI processes per address space) is the initialization code that creates the endpoints and attaches threads to endpoints. To achieve this portability one needs to ensure that threads belonging to different MPI processes do not interfere with each other (i.e., do not communicate inadvertently). Consider the simple case of a code with single-threaded MPI processes written using C, C++ or Fortran. Problems arise if the code uses mutable static variables: When the threads are in distinct address spaces then the threads have distinct instances of these variables; but when they run in the same address space, they would share the same instance.

In C and C++, static heap variables can be made thread-private by declaring them with the storage class keyword `__thread`. This storage specifier ensures that there will be one separate instance of the declared variable for each thread. While not standard, this extension is widely supported [4, §5.54]. This extension is not currently supported in Fortran – we hope this will change. Alternatively, a preprocessor can be used for replacing each static heap variable with a dynamic one that is allocated separately by each thread – see, e.g., [2]. Additional care must be taken to ensure that the library code invokes only thread-safe libraries.

Bibliography

[1] E.D. Demaine. A threads-only MPI implementation for the development of parallel programs. In *Proceedings of the 11th International Symposium on High Performance Computing Systems*, pages 153–163. Citeseer, 1997. 0.0.1

[2] C. Huang, O. Lawlor, and L.V. Kale. Adaptive mpi. *Lecture notes in computer science*, pages 306–322, 2003. 0.0.1, 0.0.7

[3] MPI Forum. MPI: A Message-Passing Interface Standard V2.2, 2009. 0.0.1, 0.0.4, 0.0.4

[4] Richard M. Stallman and the GCC Developer Community. Using the GNU Compiler Collection (for GCC version 4.4.2). 0.0.7

[5] H. Tang, K. Shen, and T. Yang. Program transformation and runtime support for threaded MPI execution on shared-memory machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(4):673–700, 2000. 0.0.1