MPI3: Supporting Multiple Endpoints per Process

Hybrid Programming Working Group

August 15, 2011

0.1 Introduction

0.1.1 Rationale

Many applications run best when shared memory is used for communication inside a node and MPI is used between nodes, *with more than one MPI process per node*. We expect this model to become increasingly common as the number of cores per node increases: A larger number of concurrent threads per node will often lead to a higher message rate that is hard to achieve within one MPI process, for a variety of reasons. For example, message matching time tends to increase linearily with the number of pending requests; the problem is inherent to the matching semantics of MPI. The use of of multiple MPI processes can also reduce intra-node communication, especially on NUMA nodes.

This model can be supported in two ways:

1. Run multiple OS processes at each node, each with one MPI rank, and use (Unix V like) shared memory segments to communicate with shared memory between the processes.

2. Run one OS process per node, but provide support for multiple "MPI processes" within one address space.

.

The use of Unix V like shared memory segments introduces various overheads and complexities, and is not supported by higher level shared memory programming models, such as OpenMP, and by debuggers and performance tools. This document describes an MPI extension for supporting the second approach.

Most MPI libraries, but not all, equate an MPI process with an OS process. This equivalence is not an MPI requirement: The MPI standard 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.

The standard does not define "process" and does not equate it with an OS process. In fact, there are MPI implementations where an MPI agent is 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].

0.1.2 Proposed Approach

We use the following terminology

An OS process consists of an address space, a program, one or more threads and, possibly, additional system resources, such as MPI endpoints.

A thread consists of a stack, program counter and other state associated with an execution.

An MPI endpoint consists of a set of resources that enable MPI calls. Such an endpoint is identified in MPI by a rank in MPI_COMM_WORLD or a rank in derived communicators.

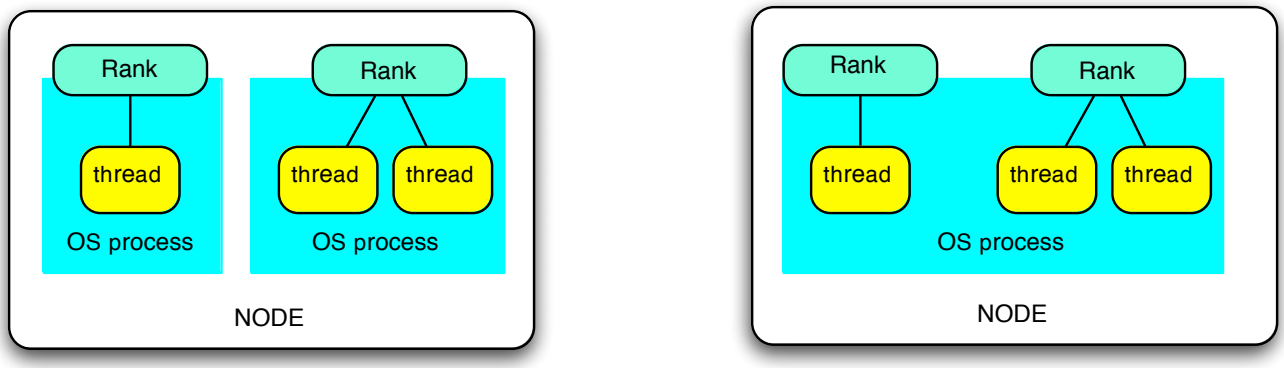

Figure 1: Current and new design for MPI

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".

Most MPI implementations can support multiple MPI processes within one OS image, each running in a distinct address space, as shown in Figure 1, left. The same mechanisms can be used to create multiple endpoints within one address space, as shown in Figure 1, right.

Changing the nature of an MPI process does not change in any way the semantics of MPI. It also requires very few changes in the MPI software stack – the changes will mostly be in the initialization code that allocates communication resources to an MPI process.

Currently, an MPI library creates an MPI endpoint for each OS process at initialization time; all threads running in this process are implicitly associated with this endpoint.

In the new design, multiple MPI endpoints may be bound to the same OS process. The binding of MPI endpoints to processes will occur when MPI is initialized. On the other hand, the binding of threads to endpoints can change during execution. At any point in time, a thread will be attached to at most one endpoint; an MPI send executed by such a thread will use the (sender) rank of that endpoint.

0.1.3 Outline

The remainder of the document is organized as follows:

Proposed extensions to MPI to enable the association of multiple MPI endpoints with distinct threads within one process are described in Section 0.2. We describe this proposal assuming the existence of threads, but without making assumptions on the properties of thread, other that they run within a shared address space.

Section 0.3 describes additional functions that can ease the use of endpoints

Section 0.4 describes how the MPI endpoints constructs interoperate with shared memory programming models such as Posix thread libraries, OpenMP and PGAS languages.

0.2 MPI Support of Multiple Endpoints per Process

0.2.1 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. Each process owns one or more endpoints; the association is static. Each thread running within this process 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_ENDPOINTS) will appear as a send by the MPI process with rank 5 in MPI_COMM_ENDPOINTS. Similarly, if a thread executes

```
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`.

0.2.2 Initialization

A new initialization call is provided to generate endpoints.

MPI_INIT_ENDPOINT(count, thread_level, endpoints)

IN	count	number of endpoints created at local process (integer)
IN	thread_level	thread level of support (integer)
OUT	endpoints	array of endpoints (array of handles)

```
int int MPI_Init_endpoint_end(int *argc, char **argv, int count, int*
              thread_level, MPI_Endpoint* endpoints)

MPI_INIT_ENDPOINT(COUNT, THREAD_LEVEL, ENDPOINTS, IERROR)
    INTEGER COUNT, THREAD_LEVEL, ENDPOINTS(*), IERROR
```

The first two arguments in the C/C++ function MPI_Init_endpoint_begin are either the arguments of the main() function, or NULL; see the description of MPI_INIT in [3, §8.7] for details.

The other arguments of MPI_ENDPOINT_INIT specify:

count: the number of endpoints created at the local OS process. This must be less or equal to the maximum value supported by the system. It can be different at different

processes.

thread_level: the level of thread support for the created endpoints. This must be less or equal to the value supported by the system. If more than one endpoint is created within one process, then the level of thread support must be at least MPI_THREAD_FUNNELED.

The argument endpoints is an array of length count. The call returns an array of handles to (opaque) endpoint objects.

The initialization code must be aware of system parameters, such as the maximum number of endpoints supported per process or the maximum level of thread support. These can be passed through argc/argv or other such mechanism, or set by arguments to mpiexec.

> *Advice to implementors.* If level of thread support and maximum number of endpoints is set by mpiexec, then it is recommended to use the arguments -threadlevel ¡ ¿ and -endpoints ¡ ¿, with thread level from 0 (not supported) to 3 (dully supported).
>
> Good quality implementations will support a large number of endpoints; the selection of the "best" number of endpoints will be system and application dependent, similarly to the selection of the "best" number of threads. (*End of advice to implementors.*)

The call to MPI_INIT_ENDPOINT generates four communicators:

MPI_COMM_ENDPOINTS communicator that includes all endpoints

MPI_COMM_WORLD Communicator that includes the first endpoint of each OS process

MPI_COMM_PROCESS Communicator that includes all endpoints within the local OS process

MPI_COMM_SELF Communicator that contains exactly one endpoint

Endpoints within each process are ordered by endpoint number and have consecutive ranks in MPI_COMM_ENDPOINTS and MPI_COMM_PROCESS. This is illustrated in Figure 2.

An error of class MPI_ERR_ENDPOINT is returned if the initialization call fails.

All MPI programs must contain at least one call per OS process to an MPI initialization routine. If only one endpoint is created at the OS process, the call can be one of MPI_INIT, MPI_INIT_THREAD or MPI_INIT_ENDPOINT. If more than one endpoint is created, then the initialization must use MPI_INIT_ENDPOINT. Additional calls to initialization routines are erroneous. The only MPI functions that can be invoked before MPI_INIT_ENDPOINT are MPI_GET_VERSION, MPI_INITIALIZED, and MPI_FINALIZED.

An initialization with MPI_INIT creates the same MPI environment as an initialization with MPI_ENDPOINT_INIT that creates one endpoint, with thread level MPI_THREAD_SINGLE.

An initialization with MPI_INIT_THREAD creates the same MPI environment as an initialization with MPI_INIT_ENDPOINT that creates one endpoint, with thread support at the same level as returned by the provided argument of MPI_INIT_THREAD.

> *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.

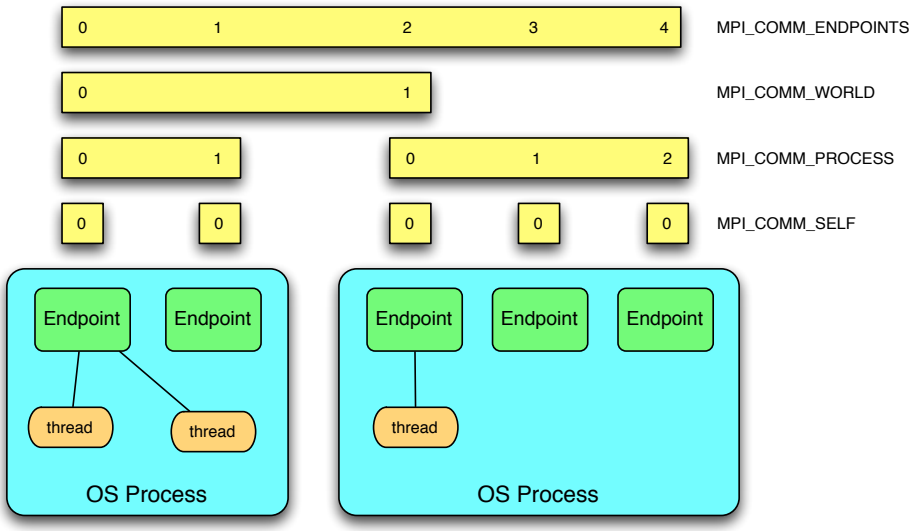

Figure 2: Communicators created by a call to MPI_INIT_ENDPOINT

Once MPI is initialized, the code can query the level of thread support using MPI_QUERY_THREAD and can find the number of endpoints per process by querying the size of MPI_COMM_PROCESS. (*End of advice to users.*)

Advice to implementors. The four communicators should be always created when MPI is initialized, even if it is initialized with MPI_INIT or MPI_INIT_THREAD. If a process initializes with MPI_INIT or MPI_INIT_THREAD, then MPI_COMM_PROCESS at that process is a duplicate of MPI_COMM_SELF. If all processes initialize with MPI_INIT or MPI_INIT_THREAD, then MPI_COMM_ENDPOINTS is a duplicate of MPI_COMM_WORLD. (*End of advice to implementors.*)

Discussion. The current design assumes that the same level of thread support is provided at all endpoints at the same OS process, but different OS processes can provide different levels of thread support. (*End of discussion.*)

0.2.3 Thread Attachment

Any newly spawned thread is attached, by default, to the first endpoint of the process. Thus, code that uses only one endpoint per OS process requires no modification. Threads can be detached and re-attached to any endpoint.

MPI_THREAD_DETACH()

```
int MPI_Thread_detach()

MPI_THREAD_DETACH(IERROR)
    INTEGER IERROR
```

This call detaches the calling thread from the endpoint it is currently attached to. The call outcome is undefined if the invoking thread is not attached to an endpoint or if there is a pending local MPI call on that endpoint (i.e., a non-blocking MPI call that is not complete) and no other thread is attached to it.

MPI_THREAD_ATTACH(endpoint)

INOUT	endpoint	endpoint (handle)

```
int int MPI_Thread_attach(MPI_Endpoint endpoint)
```

```
MPI_THREAD_ATTACH (ENDPOINT, IERROR)
    INTEGER ENDPOINT, IERROR
```

The function attaches the invoking thread to the specified endpoint. A thread may attach to at most one endpoint. The call outcome is undefined if the invoking thread is already attached, or if the level of thread support is MPI_THREAD_FUNNELED and a thread is already attached to the endpoint.

The MPI_THREAD_ATTACH and MPI_THREAD_DETACH calls are local. We discuss progress when an endpoint has no attached thread in Section 0.2.4.

An error of class MPI_ERR_ENDPOINTS is returned if the attach or detach calls fail.

0.2.4 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_INIT_ENDPOINT, MPI_GET_VERSION, MPI_INITIALIZED, MPI_FINALIZED and MPI_THREAD_ATTACH.

MPI handles are local to an MPI process and cannot be communicated between MPI processes, even within the same OS process. Thus, a handle returned by an MPI call of a thread attached to an endpoint can be used only by threads attached to that 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 should not be different than the support of multiple MPI processes, each within a distinct address space, within one OS image. In particular, communication using the MPI_THREAD_FUNNELED model, with k endpoints in one OS process within an OS image, should be performing as well or better than communication with k single-threaded OS processes, each with one endpoint, in that OS image. (*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.

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.

A newly spawned OS process has to initialize MPI with MPI_INIT_ENDPOINTS if it contains more than one endpoint.

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.

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.

0.3 Extensions

The functions described in this section facilitate the use of endpoints

0.3.1 Endpoint Attributes

Attributes can be associated with endpoints. Endpoint attributes are manipulated using the following functions:

MPI_ENDPOINT_CREATE_KEYVAL(endpoint_keyval)

OUT	endpoint_keyval	key value for future access (integer)

```
int MPI_Endpoint_create_keyval(int *endpoint_keyval)
```

```
MPI_ENDPOINT_CREATE_KEYVAL(ENDPOINT_KEYVAL, IERROR)
    INTEGER ENDPOINT_KEYVAL, IERROR
```

Rationale. Endpoints cannot be duplicated or freed; they do not change after initialization. Hence, no call-back functions are associated with endpoint key values. (*End of rationale.*)

MPI_ENDPOINT_FREE_KEYVAL(endpoint_keyval)

INOUT	endpoint_keyval	key value (integer)

```
int MPI_Endpoint_free_keyval(int *endpoint_keyval)
```

```
MPI_ENDPOINT_FREE_KEYVAL(ENDPOINT_KEYVAL, IERROR)
    INTEGER ENDPOINT_KEYVAL, IERROR
```

MPI_ENDPOINT_SET_ATTR(endpoint, endpoint_keyval, attribute_val)

INOUT	endpoint	endpoint to which attribute will be attached (handle)
IN	endpoint_keyval	key value
IN	attribute_val	attribute value

```
int MPI_Endpoint_set_attr(MPI_Endpoint endpoint, int endpoint_keyval, void
            *attribute_val)
```

```
MPI_ENDPOINT_SET_ATTR(ENDPOINT, ENDPOINT_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER ENDPOINT, ENDPOINT_KEYVAL, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
```

MPI_ENDPOINT_GET_ATTR(endpoint, endpoint_keyval, attribute_val, flag)

INOUT	endpoint	endpoint to which attribute is attached (handle)
IN	endpoint_keyval	key value
OUT	attribute_val	attribute value, unless flag = false
OUT	flag	false if no attribute is associated with the key (logical)

```
int MPI_Endpoint_get_attr(MPI_Endpoint endpoint, int endpoint_keyval, void
            *attribute_val, int *flag)
```

```
MPI_ENDPOINT_GET_ATTR(ENDPOINT, ENDPOINT_KEYVAL, ATTRIBUTE_VAL, FLAG,
             IERROR)
    INTEGER ENDPOINT, ENDPOINT_KEYVAL, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
            LOGICAL FLAG
```

MPI_ENDPOINT_DELETE_ATTR(endpoint, endpoint_keyval)

INOUT	endpoint	endpoint to which attribute is attached (handle)
IN	endpoint_keyval	key value

```
int MPI_Endpoint_get_attr(MPI_Endpoint endpoint, int endpoint_keyval)
```

```
MPI_ENDPOINT_GET_ATTR(ENDPOINT, ENDPOINT_KEYVAL, IERROR)
    INTEGER ENDPOINT, ENDPOINT_KEYVAL, IERROR
```

Advice to users. The following attributes may facilitate the design of libraries that use endpoints:

A mutex reference, to synchronize threads attached to the same endpoint.

A count of the number of threads attached to the endpoint.

A list of the ids of the threads attached to an endpoint (*End of advice to users.*)

0.3.2 Merge of Communicators

The following function is used to merge partially overlapping communicator groups.

MPI_COMM_MERGE(comm1, comm2, newcomm)

IN	comm1	first merged communicator (handle)
IN	comm2	second merged communicator (handle)
OUT	newcomm	new communicator (handle)

```
int MPI_Comm_merge(MPI_Comm comm1, MPI_Comm comm2, MPI_Comm* newcomm)
```

```
MPI_COMM_MERGE(COMM1, COMM2, NEWCOMM, IERROR)
    INTEGER COMM1, COMM2, NEWCOMM, IERROR
```

Either comm1 or comm2 can be null, but not both. If they are both non-null, then the groups of comm1 and comm2 are merged together in the new communicator. As a result the communicator returned in newcomm contains all endpoints that are in the same connected component as the invoking endpoint. The ranks in the returned communicators are arbitrary.

The effect of this function is illustrated in Figure 3.

The call is invoked by all 13 endpoints shown in the figure. 4 of the calls provide two non-null communicator arguments, the other provide one non-null argument. The result is

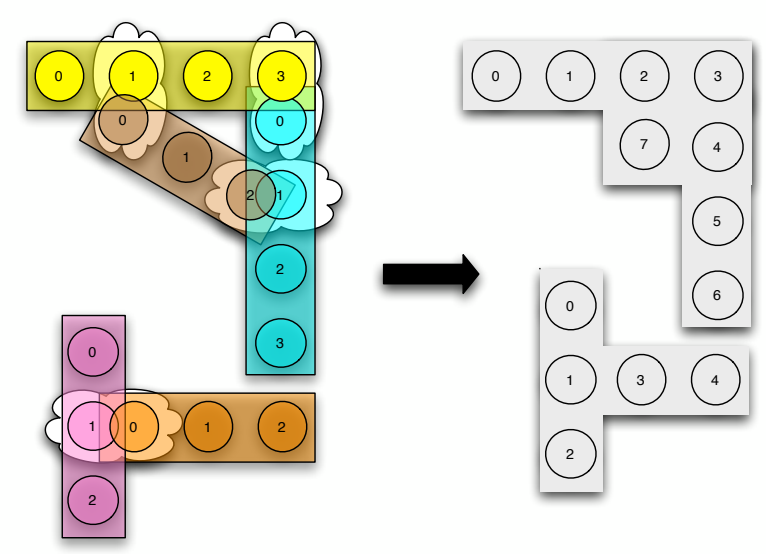

Figure 3: Communicator created by the merge of several communicators

two disjoint communicators, one for each connected component of the input communicators.

MPI_COMM_MERGE is collective over each connected component of the input communicators.

> *Advice to users.* This call is useful in a situation where a library is invoked collectively by one endpoint per OS process (in the old MPI model).The library can invoke MPI_COMM_MERGE with the communicator argument passed to the library and with MPI_COMM_PROCESS in order to create a communicator containing all the endpoints at these OS processes. (*End of advice to users.*)

0.4 Interoperability

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

0.4.1 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.

0.4.2 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>
```

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

MPI_Endpoints endpoint[max_endpoints];
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_detach();
  MPI_Thread_attach(&endpoint[i]);
  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);
}
```

0.4.3 OpenMP

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

0.4.4 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).

0.4.5 Porting Codes to Hybrid MPI

Codes written with the current MPI interfaces port without change, and use one endpoint per OS process. The transition from current MPI codes to codes using multiple endpoints per OS process can entail one or more of the following scenarios:

1. Code is written to utilize multiple endpoints per OS process and leverage shared memory communication within OS processes.

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 OS process. 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 depend on the number of UPC threads per OS process.

3. Code written to use a single endpoint per OS process invokes a library written to use multiple endpoints per OS process. 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 a distinct OS process, 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 OS process) 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. 0.1.1

[2] C. Huang, O. Lawlor, and L.V. Kale. Adaptive mpi. *Lecture notes in computer science*, pages 306–322, 2003. 0.1.1, 0.4.5

[3] MPI Forum. MPI: A Message-Passing Interface Standard V2.2, 2009. 0.1.1, 0.2.2, 0.2.4, 0.2.4

[4] Richard M. Stallman and the GCC Developer Community. Using the GNU Compiler Collection (for GCC version 4.4.2). 0.4.5

[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. 0.1.1