MPI3: Supporting Multiple Endpoints per Process

Hybrid Programming Working Group

May 1, 2010

0.1 Introduction

0.1.1 Current State

In the next few years, supercomputers will be built of nodes with an increasingly large number of cores. Indeed, most of the increase in performance will come from an increase in the number of cores per node, while the number of nodes will increase at a more modest rate. This increases the interest in good support for hybrid programming models that take advantage of shared memory inside shared memory nodes, while using message passing across nodes.

Experiments with current systems that have a large number of cores per node indicate that performance is often improved by using shared memory communication withing one OS process inside nodes: irregular applications can benefit from dynamic load balancing within nodes; communication using shared memory is more efficient as it avoids one or two memory-to-memory copies; and the replication of read-only data structures is avoided. On the other hand, careless use of shared memory can lead to excessive memory traffic due to false sharing and to improper memory placement in NUMA systems; these problems can be avoided with proper programming practices. See [12, 15, 3], and references therein.

The predominant hybrid model used so far is that of one multi-threaded process per node; MPI is used for inter-node communication while shared memory parallelism, such as provided by OpenMP, is used inside the node. The node corresponds to one MPI process – i.e., one MPI rank in a communicator. This model has deficiencies, as pointed out in the previously referenced papers: if only one thread makes MPI calls, then the thread may not be able to inject messages at a high enough rate; MPI calls will typically be executed within the sequential part of an OpenMP code, imposing unnecessary serialization, and reducing communication/computation overhead. If, on the other hand, multiple threads access MPI, then one needs to use a thread-safe MPI implementation, which may impose a performance penalty [21]; call-backs that service out-of-order messages may still serialize. Systems with large SMP nodes will often have multiple message-passing adapters; performance may be improved by having each adapter used by a subset of the threads – thus reducing synchronization overheads and possibly improving locality on a large NUMA node. The hierarchical model of one MPI process with multiple threads may not be the best way of organizing a hybrid parallel computation: Figure 1, taken from [15] illustrates this for simple halo-swaps: One achieves best performance (on a system with no jitter) by allocating to each thread a patch of the matrix, and having each thread communicate with its neighbors. The communication uses shared memory (with no halo layer) with neighbors at the same node and MPI for neighbors at other nodes. The code is simplified (and performance is improved) if each thread that has to communicate with a neighbor at another node has its own MPI rank. The analysis in [7] leads to the same conclusion. Finally, PGAS languages such as UPC and Fortran 2008 hide from the user the process boundaries: UPC threads can be in the same address space or in distinct address space. The implementation chhoses how many processes to ruin per node and how many threads to run per process; this choice in transparent to the user. It is desirable to be able to write MPI libraries that match this model.

This leads us to the goal of supporting multiple "MPI endpoints" within one multi-threaded process, and enabling the association of specific threads with specific endpoints. We want to support a hybrid programming model with the following properties:

1. Intranode communication uses shared memory

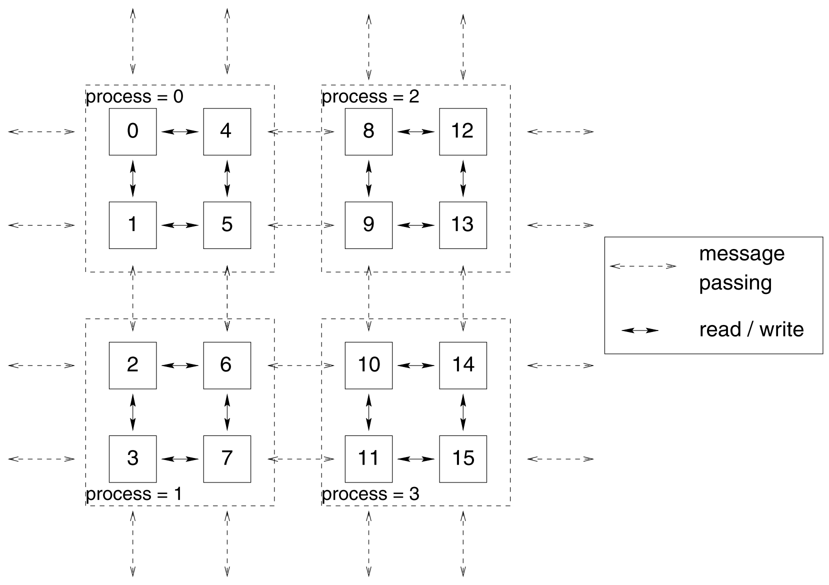

Figure 1: Halo swaps within the mixed SPMD OpenMP / MPI code

2. Internode communication uses message passing
3. The number of MPI endpoints per process can be larger than one
4. The maximum number of endpoints per process is configuration-dependent and can be chosen to optimize performance
5. MPI endpoints can be dedicated to one thread, to avoid the overhead of a thread-safe MPI library. More generally, the programmer can control the association of threads with endpoints
6. The programming model can interface with the standard, commonly used and emerging shared memory programming models. This includes
   - C programs using the pthread library
   - C++0x programs using the C++0x thread library
   - Programs written using OpenMP [11]
   - Programs written using PGAS languages, such as UPC [22] or Fortran 2008 [13]. Users may choose to exploit shared memory using PGAS languages, in order to ensure good locality on NUMA systems
   - Programs written using emerging shared memory parallel models, such as TBB [14]
7. Programs can be migrated from the current model to a model where multiple "MPI processes" run in the same address space with few modifications

In addition, we consider the issue of interoperability between PGAS languages and MPI, when PGAS programs run globally, across multiple nodes.

0.1.2 Proposed Approach

To remove confusion we shall use the term *MPI agent* as a synonym for "MPI process"; while *process* will refer to an OS process, i.e., an address space, one or more threads, and a set of system resources.

The fundamental insight of the proposed design is that there are no compelling reasons to equate an MPI agent with an OS process. This is not an MPI requirement: The MPI standard says [10, §2.7]:

> An MPI program consists of autonomous processes, executing their own code, in an 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 [4, 20], or even a task that can be dynamically scheduled by a run-time on different threads, for load balancing [6]. These implementations attempt to hide from the user the association of MPI agents with threads or tasks, relying on a thread or task scheduler to schedule MPI agents in an appropriate manner. Our approach is motivated by different concerns. We want to expose the association of MPI processes to threads in order to improve resource management and standardize the interoperability with other APIs.

Changing the nature of an MPI agent 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 agent.

An *MPI agent* consists of

- A set of communication resources allocated by the OS which we call an *MPI endpoint*. These resources are represented, in MPI, by a rank in a communicator.
- A set of one or more threads that can communicate using the endpoint

Most MPI implementations can support multiple MPI agents within one OS image, each running in a distinct address space, as shown in Figure 2. In a single-threaded implementation, we have only one thread associated with the MPI endpoint (Figure 2, left); in a multi-threaded implementation, we can have multiple threads associated with the same endpoint (Figure 2, right).

To support this, the communication hardware and software enable the creation of multiple logically independent endpoints at a node; these endpoints can by supported by physically distinct communication resources (e.g., distinct Infiniband adapters); or they can be multiplexed onto shared communication resources; the sharing is hidden from the user, except for possible performance interference. The association of threads with communication endpoints is implicit: threads running in an address space can only access the endpoint associated with this address space. This association is managed by the OS: It can map distinct command registers into the distinct address spaces, thus creating distinct endpoints for the different processes that can be accessed in user mode, when multiplexing is done by the communication adapter; or it can associate each process with a different file descriptor (socket, pipe), when communication multiplexing is done by the kernel.

The same mechanisms can be used to create multiple endpoints within one address space, as shown in Figure 3. The design is same as before, except that the multiple MPI

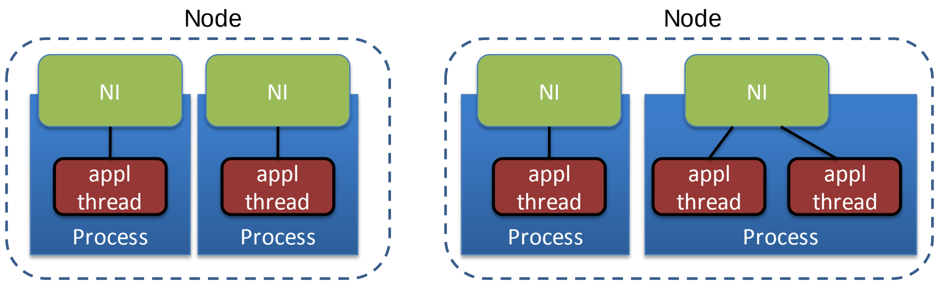

Figure 2: MPI Current Design

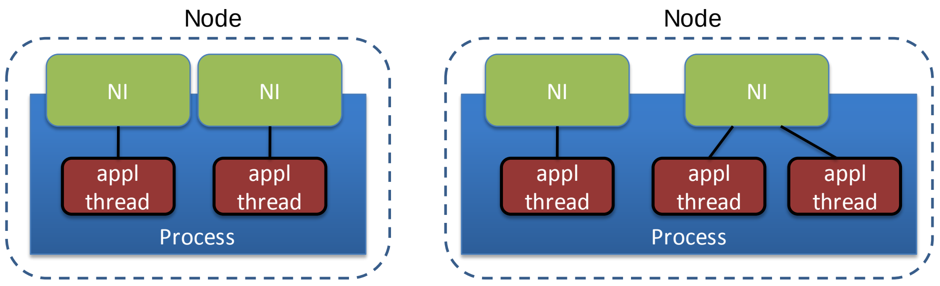

Figure 3: MPI Proposed Design

agents at a node can all run within a shared address space. We shall have the same choice: one thread per endpoint (Figure 3, left), or multiple threads per endpoint (Figure 3, right). This should not necessitate any major changes in the MPI library or the underlying device drivers.

In order to support the new model, we need to explicitly partition threads into groups, where each group of threads is associated with one endpoint. The separation of endpoints is not enforced anymore by the OS, but by the application code.

This design implies two binding times for:

1. binding of MPI endpoints to OS processes; and

2. binding of threads to MPI endpoints

The binding of MPI endpoints to processes occurs when MPI is initialized, in the preamble code. This is consistent with current MPI libraries that allocate communication resources at initialization time. On the other hand, the binding of threads to endpoints is dynamic and can be changed during execution. This allows support for environments (such as OpenMP) where the number and identity of threads can change during a computation.

A thread can be attached to at most one endpoint, so that when it executes an MPI call, it is clear which MPI endpoint is used by the call; i.e., what the implied rank of the caller is. If affinity scheduling is used for threads, then one can ensure in a NUMA system that MPI calls on a particular endpoint are done on a specific core or set of cores that can access this endpoint with lower overhead.

A process could have multiple threads, each attached to one endpoint; it could have, in addition, threads that are not attached to endpoints and cannot execute MPI calls.

Alternatively, multiple threads could be attached to each endpoint, as in the MPI_THREAD_MULTIPLE model.

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. This proposal modifies and expands the proposal presented at the MPI3 forum by Alexander Supalov [19] – with one key pragmatic difference: we do not focus on supporting an arbitrary number of threads, each acting as an MPI process, within one OS process; but, rather, supporting a number of MPI agents that optimizes communication performance – and that is lower than a limit set by the system. Therefore, the proposed approach can reuse current MPI machinery with few changes, and does not require an additional level of multiplexing and resource management. Such an additional level, if desired, could be provided by a library implemented on top of MPI. 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.

An actual implementation of hybrid MPI requires an MPI library and a thread package; the MPI library should be aware of the thread package used, as it needs to interact with it. Each specific binding of hybrid MPI may require a distinct version of the MPI library, and will require distinct include file or module to identify it. In practice, we expect that the amount of MPI code that is thread package specific to be very small and isolated, so that different bindings will be defined by a small set of functions provided atop the core MPI library; furthermore, most bindings are likely to use POSIX threads.

We present in Section 0.3 a general discussion of collective libraries in a hybrid model. MPI provides a clear model for the collective invocation of a library from single-threaded processes; this model is used by libraries such as Scalapack [2] and PetsC [1]. No similar model has been defined for parallel library invocation from multi-threaded processes. We propose such models and discuss their extension to MPI code that uses endpoints.

A binding of the proposed model with POSIX thread is described in Section 0.4.

A general discussion, in Section 0.5, describes how MPI can bind to shared memory programming languages and frameworks.

Section 0.6 introduces the proposed bindings for OpenMP.

Section 0.7 discusses possible bindings to task-oriented environments, such as TBB or Cilk.

Section 0.8 describes the general principles for binding to PGAS languages and defines the bindings for UPC and Fortran 2008.

Finally, Section 0.9 provide several examples of hybrid code.

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 can change, dynamically.

In current (static) MPI, there is a fixed one-to-one correspondence between MPI processes and ranks in MPI_COMM_WORLD; when a process executes an MPI call with argument MPI_COMM_WORLD then the local rank is not specified as an argument to the call, but is implied by that correspondence. Similarly, in our proposal, there is a one-to-one correspondence between MPI endpoints and ranks in MPI_COMM_ENDPOINT; when a thread that is attached to an MPI endpoint executes an MPI call with argument MPI_COMM_ENDPOINT then the local rank implied by the call 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_ENDPOINT) will appear as a send by "MPI process" with rank 5 in MPI_COMM_PROCESS. Similarly, if a thread executes

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

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

0.2.2 Initialization

Endpoint Initialization

Initialization is done in two phases. The first phase is a call to MPI_INIT_ENDPOINT that returns information that may be needed in order to determine how many endpoints to create at the local process. The second phase is a call to MPI_ENDPOINT_CREATE that creates local endpoints and initializes communicators.

Rationale. The examples in Section 0.9 illustrate the need for information on the total number of processes, or the process rank, in order to determine how many endpoints to create at the local process. However, this information is normally available only after initialization. The two phase initialization is used to resolve this conundrum. (*End of rationale.*)

Advice to implementors. Some implementations use out of band communication to gather the information on number of processes and local rank, next initialize MPI and allocate physical communication resources. Those implementation can use out of band communication to provide the information returned by MPI_INIT_ENDPOINT, namely number of processes and local process rank; and allocate physical communication resources when MPI_ENDPOINT_CREATE is called and the number of required endpoints is known. Other implementations do not require a priori knowledge of the communication topology, do not use out of band communication, and can allocate communication resources dynamically. Such implementations can initialize MPI when MPI_INIT_ENDPOINT is invoked and change later the communication topology, as needed. (*End of advice to implementors.*)

MPI_INIT_ENDPOINT(max_endpoints, size, rank)

OUT	max_endpoints	maximum number of endpoints that can be created at local process (integer)
OUT	size	total number of processes (integer)
OUT	rank	rank of local process (integer)

```
int int MPI_Init_endpoint(int *argc, char **argv, int *max_endpoints, int*
             size, int *rank)
```

```
MPI_INIT_ENDPOINT(MAX_ ENDPOINTS, SIZE, RANK, IERROR)
    INTEGER MAX_ENDPOINTS, SIZE, RANK, IERROR
```

MPI_ENDPOINT_CREATE(count, endpoints)

IN	count	number of endpoints created (integer)
OUT	endpoints	array of endpoints (array of handles)

```
int int MPI_Endpoint_create(int count, MPI_Endpoint* endpoints)
```

```
MPI_ENDPOINT_CREATE(COUNT, ENDPOINTS, IERROR)
    INTEGER COUNT, ENDPOINTS, IERROR
```

All MPI programs must contain at least one call per process to an MPI initialization routine. If only one endpoint per process is created, 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, followed by MPI_ENDPOINT_CREATE. Additional calls to initialization routines are erroneous. The only MPI functions that can be invoked before MPI_ENDPOINT_CREATE are MPI_GET_VERSION, MPI_INITIALIZED and MPI_FINALIZED. We do not require that all processes use the same initialization function. An MPI program

Alternatives: All processes must use the same initialization functions. **Discussion:**

Implementers should say whether they care.

The first two arguments in the C/C++ function MPI_Init_endpoint are either the arguments of the main() function, or NULL. The values returned by the call to MPI_INIT_ENDPOINT are:

max_endpoints – the maximum number of endpoints that can be created on the local process

size – the size of the communicator MPI_COMM_WORLD that will be created by the call to MPI_ENDPOINT_CREATE; and

rank – the rank of the local process in that communicator.

Advice to users. The user can use the information returned by MPI_INIT_ENDPOINT in order to decide how many endpoints to create in the subsequent call to MPI_ENDPOINT_CREATE. (*End of advice to users.*)

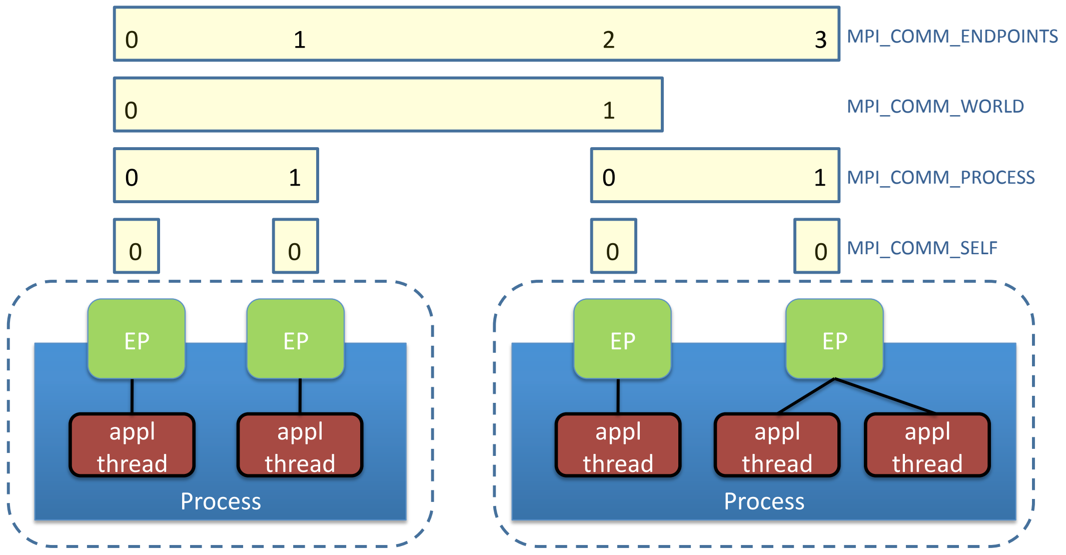

Figure 4: Communicators created by a call to MPI_CREATE_ENDPOINT

Advice to implementors. The value returned by max_endpoints could be set externally by the user using an argument to the parallel job creation facility. System default should be chosen to be the value that provide most performance benefit, for well-written codes. This will typically be smaller or equal to the number of cores in the OS image. (*End of advice to implementors.*)

The argument count in the call to MPI_CREATE_ENDPOINT must be smaller or equal to the value returned by MPI_INIT_ENDPOINT in max_endpoints. It can have a different value at different processes. The argument endpoints is an array of length count. The call returns an array of handles to (opaque) endpoint objects.

The call to MPI_CREATE_ENDPOINT generates four communicators:

MPI_COMM_ENDPOINTS communicator that includes all endpoints

MPI_COMM_WORLD Communicator that includes the first endpoint of each process

MPI_COMM_PROCESS Communicator that includes all endpoints with the local 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 4.

Rationale. It would be sufficient to create only one communicator, namely MPI_COMM_ENDPOINTS, as the others can be split from this one; the creation of all four communicators is for convenience. (*End of rationale.*)

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: Current proposal sets the provided level of thread support when a thread is attached. If this is a problem for some implementations, then we can set it when the endpoint is created.

Missing: Need to add error codes

Endpoint Attributes

The following attributes are cached with each endpoint, when the endpoint is created:

MPI_MAX_THREAD_LEVEL	Highest available level of thread support
MPI_CURRENT_THREAD_LEVEL	Currently provided level of thread support

The value of MPI_CURRENT_THREAD_LEVEL is undefined when no thread is attached to the endpoint.

If MPI_INIT_ENDPOINT initiated more than one endpoint, then the level of thread support must be at least MPI_THREAD_FUNNELED. Different endpoints at the same process may have different levels of available thread support

Additional, implementation dependent attributes may be used to provide information on the endpoint; e.g., to indicate its location and type in an heterogeneous architecture.

Endpoint attributes are manipulated using the following functions:

MPI_ENDPOINT_CREATE_KEYVAL(endpoint_keyval)

OUT	endpoint_keyval	key value for future access (integer)

```
int 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 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 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 will be 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 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 will be attached (handle)
IN	endpoint_keyval	key value

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

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

Missing: Need to add error codes

Thread Attachment

MPI_THREAD_ATTACH(endpoint, required)

INOUT	endpoint	endpoint (handle)
IN	required	required level of thread support (integer)

```
int int MPI_Thread_attach(MPI_Endpoint endpoint, int required)
```

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

The function attaches the invoking thread to the endpoint. The required level of thread support can be one of MPI_THREAD_SINGLE, MPI_THREAD_FUNNELED, MPI_THREAD_SERIALIZED, or MPI_THREAD_MULTIPLE. The call is erroneous if the required level of thread support is higher than the maximum available level of thread support at that endpoint. The required level should be at least MPI_THREAD_FUNNELED if more than one thread will be active; it should be at least MPI_THREAD_SERIALIZED if more than one thread will attach to that same endpoint – the result is implementation dependent, otherwise.

A thread may attach to at most one endpoint. If an endpoint thread support level is MPI_THREAD_SINGLE or MPI_THREAD_FUNNELED then the endpoint can be attached by at most one thread. If multiple threads are simultaneously attached to the same end-point, then they must have attached with the same required level of thread support. The provided level of thread support at an endpoint may change during the computation, but it can be changed only when a thread attaches to an endpoint that has no other threads attached to it.

A thread must be attached to an endpoint in order to execute any MPI call, other than MPI_INIT_ENDPOINT, MPI_GET_VERSION, MPI_INITIALIZED and MPI_FINALIZED . The call to MPI_ENDPOINT_CREATE must precede the call to MPI_THREAD_ATTACH.

Discussion. We could make MPI_THREAD_ATTACH to look more like MPI_INIT_THREAD – by having an input parameter with required thread support level and an output parameter with provided thread support level. But this would not work very well: The function MPI_INIT_THREAD is invoked by one thread per process, while the function MPI_THREAD_ATTACH is invoked by each thread; these threads need to coordinate before attaching, to ensure that more than one thread can attach to the endpoint and to attach with the same level of thread support. The current design enables them to coordinate before invoking MPI_THREAD_ATTACH – without any communication. The alternative design would require one thread to attach and broadcast the provided level of thread support to the other threads; next have those threads attach.

The current choice assumes that the number of threads attached to an endpoint can change at different phases of a computation and that the implementation can take advantage of phases where endpoints are single-thread to improve performance. The MPICH team believes that the later is true. The former sounds reasonable. (*End of discussion.*)

Advice to users. Programmers can ensure portability of their code by checking the value of the MPI_THREAD_LEVEL attribute of an endpoint before attaching multiple threads to that endpoint. (*End of advice to users.*)

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 is erroneous if the invoking thread is not attached to an endpoint.

This function should be invoked only when there are no pending local MPI calls on the specified endpoint; it is erroneous, otherwise.

Discussion: Do we really need the restriction in the last paragraph? We could have, instead:

No restrictions: thread can execute a nonblocking call, detach and the call will be completed by a thread that attaches later.

The endpoint is required to be quiescent only when the last thread detaches – when the number of attached threads goes to zero.

Do implementers care?

The MPI_THREAD_ATTACH and MPI_THREAD_DETACH calls are local. We discuss progress when an endpoint has no attached thread in Section 0.2.3.

Missing: Add error codes.

0.2.3 Communication With Endpoints

Whenever "MPI process" is mentioned in the MPI standard, it should now be understood to mean "MPI agent" – i.e., the set of threads currently attached to an endpoint. This rules listed below follow from this interpretation.

A thread must be attached to an endpoint (i.e. must be part of an MPI agent) in order to make MPI calls other than MPI_INIT_ENDPOINT, MPI_GET_VERSION, MPI_INITIALIZED and MPI_FINALIZED. An MPI call by a thread uses the endpoint the thread is attached to.

The rank of an MPI agent in a communicator comm is determined by the original rank of the endpoint in the initial communicators (MPI_COMM_WORLD, MPI_COMM_ENDPOINTS, MPI_COMM_PROCESS and MPI_COM_SELF, and the sequence of operation used to derive comm from these initial communicators. This determines the sender id of a message sent by a thread attached to that endpoint.

MPI handles are local to an MPI agent and cannot be communicated between agents. Thus, a handle returned by an MPI call of a thread attached to an endpoint can be used only by threads attached to the same endpoint.

Rationale. The sharing of MPI handles across agents results in the same overheads as in multi-threaded MPI, since the implementation has to guarantee atomics updates to MPI objects, and to their reference counts. (*End of rationale.*)

The rules and restrictions specified by the MPI standard [10, §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 agents at a process should not be different than the support of multiple processes at an SMP node. In particular, communication using the MPI_THREAD_FUNNELED model, with k endpoints in one process at a node, should be performing as well or better than communication with k single-threaded MPI processes at the node. (*End of advice to implementors.*)

Progress

MPI 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 complete until a matching receive is executed; a blocking collective operation might not complete 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 complete irrespective of activities at other processes.

These rules are extended to the situation where a process may have multiple endpoints: a blocking send on an endpoint might not complete until a matching receive has occurred at the destination endpoint; and a collective operation might not complete until it is invoked at all endpoints of the communicator. On the other hand, a non-blocking send or a non-blocking collective will complete, 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 complete if no thread is attached to the destination endpoint; a collective operation might not complete if one of the endpoints in the communicator has no attached thread. However, a non-blocking send will complete even if there is no thread attached to the destination endpoint; and a non-blocking collective will complete even if one of the endpoints in the communicator has no attached thread.

> *Advice to implementors.* Since endpoint creation is collective, no message may arrive at a process before the local endpoints have been initialized. However, a message may arrive before a thread is attached to the receiving endpoint. When endpoints are initialized, the MPI library should create the structures needed to handle "early arrivals": This situation is not much different from the situation obtaining when an eager send arrives before a matching receive is posted. Therefore, we do not expect major implementation changes. (*End of advice to implementors.*)

Caching

Each endpoint can be associated with different attribute values.

MPI_FINALIZE()

MPI_FINALIZE must be invoked once at each 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 agent.)

Advice to users. The finalize call will usually be invoked in a sequential postamble after all threads, but the master thread, have completed execution. (*End of advice to users.*)

Memory Allocation

Memory allocated by MPI_ALLOC_MEM [10, §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 handles are attached to endpoints. A communicator may have different error handlers attached to the different endpoints of that communicator with the same address space.

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

Process Manager Interface

The MPI_COMM_SPAWN function can be used to spawn processes with multiple endpoints, with the same number of endpoints at each process. Therequired number of endpoints per process is specified by the value of the reserved key num_endpoints in the info argument. The call returns argument returns an intercommunicator containing the endpoints of the old communicator and the new endpoints.

Alternatives: Could, instead, have a *soft* num_endpoint argument that specifies the requested number of endpoints – will need then a mechanism to return the actual number of endpoints created.

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 all endpoints must supply 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, filtype 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.

0.2.4 Porting Codes to Hybrid MPI

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

1. Code is written to utilize multiple endpoints per process and leverage shared memory communication within 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 precess. Such portability is important for MPI libraries invoked from UPC or Fortran 2008: The library can have one MPI agent per UPC thread (or Fortran 2008 image), and its behavior will not depend on the number of UPC threads per address space.

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

We discuss in subsequent sections how the first scenario is supported with different shared memory programming models (Posix threads, OpenMP, etc.).

MPI code can be written so that it has the same outcome, whether each MPI agents is a distinct OS process, or multiple MPI agents run within the same address space. The only part of the code that has to be aware of the system configuration (i.e., then number of endpoints per address space) is the initialization code that creates the endpoints and attadches threads to endpoints. To achieve such portability one needs to ensure that threads belonging to different MPI agents do not interfere with each other and communicate only using MPI. Consider the simple case of a code with single-threaded MPI processes written using C, C++ oor Fortran. Then one needs to ensure that threads running in the same address space do not communicate with each other indavertendly. 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 implies that there will be one separate instance of the declared variable for each thread. While not standard, this extension is widely supported [18, §5.54]. This extension is not currently supported in Fortran – we hope this will change. This (or similar) transformation can be automated with a preprocessor – see, e.g., [6]. Additional care must be taken to ensure that the library code invokes only thread-safe libraries.

The third scenario is discussed in detail in the next section.

0.3 Interface to MPI Libraries

0.3.1 Interface for Single Endpoint Processes

An important feature of MPI is its support for loosely synchronous invocations of parallel libraries. This is briefly described in the standard in [10, §6.9], and in more detail in [16, §5.5]. The model described in these references is that of one invocation per process. One of the arguments in the invocation is a communicator that provides information on the set of processes involved in the invocation and a mechanism for their communication; additional needed information can be cached with this communicator. Rather than an explicit communicator argument, libraries often use an opaque, library specific argument that identifies the participating processes and provides a handle to the communicator(s) used by the library. A common scheme is to have a library initialization call that is collective over all participating processes and returns a descriptor of the group of involved processes; and then to pass this descriptor in each library invocation; see, e.g., the Scalapack library [2]. Following Scalapack, we call such an argument a *context*.

Scientific libraries are being extended to use multiple threads per process; see, e.g., [17], but no convention yet exists about the interface to such libraries. In a hybrid environment one can have two possible library interfaces:

Process-Collective: The library is invoked by one thread on each involved process. We have, at the collective invocation interface, a one-level hierarchy of single-threaded processes that communicate via MPI.

Thread-Collective: The library is invoked by multiple threads (possibly all threads) on each involved process. We have, at the collective invocation interface, a two-level hierarchy of processes that communicate via MPI and, within each process, threads that communicate via shared-memory.

Both designs can be upward compatible with current designs. In the first design, a library that uses multiple threads will spawn (or awake) the threads inside the library – adding some overhead. The second design reduces such overheads.

If a thread-collective interface is used, the context has to enable communication across distinct processes, and distinct threads within each process. It should contain:

- A communicator that enables communication with all other MPI agents that invoke the library.
- A shared synchronization object that enables coordination with all other threads in the local MPI agent that invoke the library.

The nature of the synchronization object depends on the thread library. E.g., with POSIX threads, it could be a structure containing a count of the number of synchronizing threads, a pointer and an initialized mutex – this is sufficient to bootstap any synchronization; or, it could be a shared task queue.

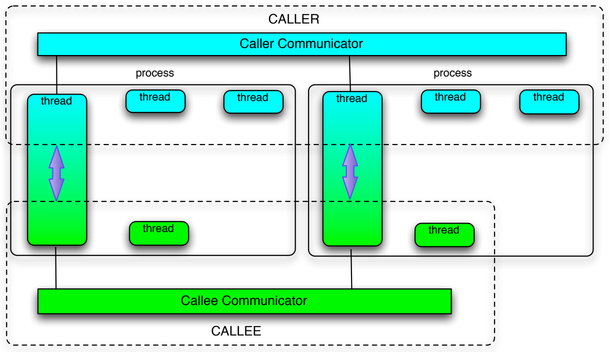

Figure 5: Process-Collective Model of Parallel Library Invocation

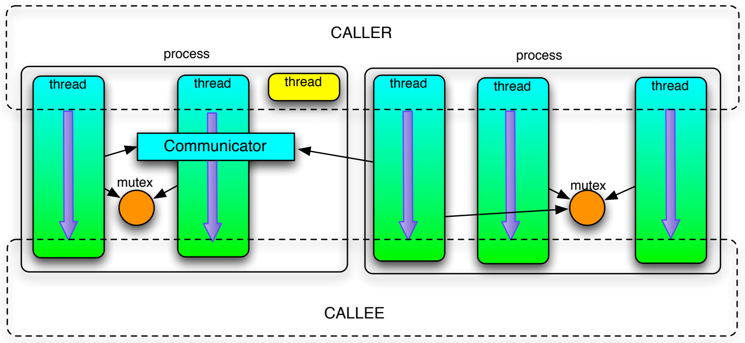

Figure 6: Thread-Collective Model of Parallel Library Invocation

In the process-collective model, the library invocation can be asynchronous (one thread per process invokes the library, other threads continue with another computation); or, it can be synchronous (one thread per process invokes the library, the other threads block and wait for the library call to complete). The calling thread can spawn or activate additional threads within the library body, to share the work; these threads terminate before the library invocation completes. This model is illustrated in Figure 5: The computation includes two processes, each with three threads. One thread on each process calls the library; the library spawns additional threads.

In the thread-collective model, all threads can invoke the library synchronously. The library could also be invoked by a subset of the threads, while the other threads continue to compute, asynchronously. This is illustrated in Figure 6. The computation includes two processes and five threads are invoking the library. One thread, possibly working with another communicator, is not invoking the library. One thread on each process passes as argument a handle to a communicator that is shared by all invoking processes; threads within the same process pass as argument a handle to a synchronizing object that is shared by all invoking threads with the process.

Both models can be useful and need to be supported.

0.3.2 Interface with Multiple Endpoints per Process

When we have multiple endpoints per process, there are three possible models:

Process-Collective The library is invoked by one thread per process. The library can spawn more threads and use more endpoints.

Endpoint-Collective The library is invoked by one thread per endpoint. The library can spawn more threads.

Thread-Collective The library is invoked by multiple threads per endpoint.

In the endpoint-collective model, the context argument provides a handle to a communicator that identifies all involved endpoints. in addition, in the thread-collective model, it provides a handle to a synchronization object that enables coordination among invoking threads in each MPI agent. This can also be used in the endpoint-collective model, in order to enable shared memory communication between MPI agents running in the same address space.

"Legacy" MPI applications that were written for one endpoint per process will use the process-collective invocation model. It is desirable to have libraries that are invoked by one thread on each process to be able to use more than one endpoint per process. To do so, the context argument will need to carry handles to the local endpoints and to a communicator that spawns all the endpoints involved in the library computation. In all these cases, the required information can be associated with the context in a library initialization code.

The proposed approach assumes that the objects involved in the library invocation (invoking threads, endpoints, communicators) are known at library initialization time. This library execution context may change during proigram execution: Foe example, one may have a multi-physics code where different modules execute concurrently using distinct communicators; furthermore, these communicators may be redefined during execution, to accomodate the migration of processes from one module to another for load balancing. In such a scenario, the library will need to be reinitialized whenever the context changes.

0.4 POSIX Binding

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

An MPI program that needs to interoperate with a POSIX thread library must include (in C/C++) a header file hybridmpi.h or (in Fortran) a module hybridmpi. This replaces both the mpi.h file (C/C++) or mpi module (Fortran) and the pthread.h header file (C/C++) or pthreads module (Fortran).

Advice to implementors. The hybridmpi.h file will often include mpi.h, pthread.h and a small number of additional declarations. (*End of advice to implementors.*)

Missing: Should check whether there is a standard Fortran pthread module

0.4.1 example

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

Listing 1: Simple MPI hybrid program

```
#include <mpihybrid.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_attach(&endpoint[i], MPI_THREAD_FUNNELED);
  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()
 {
  int max, size, rank;
  /* initialize endpoints */
  MPI_Init_endpoint(NULL, NULL, max, size, rank);
  if (max > max_endpoints) max = max_endpoints;
  mpi_create_endpoints(max, endpoint);

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

0.5 Bindings for Shared Memory Languages and Libraries

Shared-memory parallel programming languages such as OpenMP [11], and frameworks such as TBB [14] , .NET Task Parallel Library [9], Java fork-join framework [8] and Cilk [5] provide a *task model*: A task is defined by OpenMP [11, §1.2.3] as "a specific instance of executable code and its data environment" and by TBB [14, §8] as "a quantum of execution". Tasks are generated dynamically during execution by parallel control constructs, and are scheduled dynamically to the executing threads. Tasks in OpenMP can be suspended at various scheduling points and resumed later, possibly on another thread. Other environments, such as TBB, provide non-preemptive tasks. In addition OpenMP and other frameworks support *work-sharing constructs*, such as parallel loops; those define units of work (iterates) that are allocated to the threads sharing the work; the allocation can be dynamic and system dependent.

The endpoint API defined in Section 0.2 speaks of threads: Threads are attached to or detached from endpoints and execute MPI calls. We have two possible approaches for reconciling these two models:

Task Model: An MPI *thread* is equated with an OpenMP or TBB *task*.

Thread Model: An MPI *thread* is an OpenMP (or TBB) *thread*.

The task model is more convenient for (casual) programmers, since the programmer does not need to understand the mapping of tasks to threads; this is the responsibility of the run-time. On the other hand, it requires more work from implementers, as they have to ensure that MPI state is being migrated with a task, when that task is migrated from one thread to another. In particular, close interaction between MPI implementers and OpenMP or TBB implementers will be required to support the first model. Also, it is not clear that such a model will satisfy power programmers who want more control on locality and task scheduling.

The thread model is much easier to implement. If the various shared memory programming environments, such as OpenMP or TBB are implemented atop a POSIX thread package, then it is very likely that the same implementation, with few changes, will work for shared memory environments. It provides more control for power users, but it requires a more limited or stylized programming model.

We describe in the document support for the thread model and postpone work on a task model to a later stage.

Discussion. The use of MPI from shared memory languages would be facilitated if those languages provided a mechanism for binding a task to a particular thread, or set of threads. Such mechanism will also help in the handling of heterogeneous architectures and better handling of locality: We may want to control where a particular computation will executed; affinity scheduling of threads provide such a control for threads, but we do not have now affinity scheduling for tasks.

With such a mechanism, one would be able to dynamically schedule tasks on a thread that is attached to a particular endpoint. (*End of discussion.*)

0.6 OpenMP Binding

0.6.1 OpenMP Scheduling

We briefly review the scheduling mechanism of OpenMP (references are to Version 3.0 of the OpenMP standard [11]):

An OpenMP program begins as a single thread of execution. When a thread encounters a parallel construct [11, §2.4] , it creates a team consisting of itself (as the master thread of the team) and possibly other threads to execute the construct. The master task that reached the parallel construct is suspended and resumes on the master thread when the parallel construct has completed. Each task in the parallel construct is tied to one thread in the team that executes the task to completion. The exact number of threads allocated to a team is determined by a complex formula and depends on various environmental variables, the depth of the parallel construct, the number of available threads, and clauses of the parallel construct [11, §2.4.1]. Once a team is created, the team's threads do not change. Parallel constructs can be nested. A thread is associated with one active team at a time (in a nested parallel construct, it can be associated with teams at different level of nesting).

When a *work-sharing* construct [11, §2.5] is encountered within a parallel section then the iterates within the work-sharing construct are distributed among the the team's threads. The distribution may be dynamic and schedule and data dependent; or it can be fixed – depending on the clauses in the work-sharing construct.

When a *task* construct [11, §2.7] is encountered, then a new task is explicitly created. This task can be scheduled on any of the threads of the relevant team. Such tasks can be descheduled at any *scheduling point* [11, §2.7.1]. Tasks are, by default, *tied*, and resume execution on the same thread that started their execution. *Untied* tasks can resume execution on another thread of the team.

OpenMP provides three levels of data sharing [11, §2.9]:

- private variables have a different instance on each task.
- threadprivate variables have a different instance on each thread; tasks executing on the same thread share the same instance.
- shared variables have one global instance that is shared by all tasks.

The level of sharing of a variable within each parallel construct is specified by clauses in the construct and by default rules. The execution of a parallel construct may change the level of sharing, in which case the clauses also specify how the variable value(s) immediately before the change relate(s) to the value(s) immediately after the change.

threadprivate instances of variables are preserved within parallel regions, OpenMP does not specify the correspondence between threadprivate variables across different parallel constructs, with two exceptions [11, §2.9.2]:

- Within a parallel region, reference by the master thread to threadprivate variables are to the instance on that thread before entering the parallel region; this instance persists after exiting the parallel region. (The master thread has number 0 within its current team; thread number can be queried using the library routine omp_get_thread_num.)

- The values of threadprivate variables of non-master threads are guaranteed to persist across two consecutive active parallel regions only if the following conditions hold:
  - Neither parallel region is nested inside another explicitparallel region.
  - Both parallel regions use the same number of threads.
  - The value of the *dyn-var* internal control variable is false on entry to both parallel regions. (When *dyn-var* is true then OpenMP run-time can determine on its own the number of threads it allocates to a team; when it is false, this number is determined by the user. The value of *dyn-var* can be set using the library routine omp_set_dynamic).

0.6.2 OpenMP Interoperability with MPI

OpenMP C/C++ programs that invoke MPI must have an include file ompi.h; OpenMP Fortran programs that invoke MPI must have a module ompi. This replaces the mpi.h header file or mpi module normally used in MPI programs.

The OpenMP binding to MPI is defined by the following rule:

> Assume that an OpenMP task invokes MPI_THREAD_ATTACH. Then an MPI call to a function MPI_XX using the attached endpoint is valid at another point in the program if
>
> - The call to MPI_XX occurs after the call to MPI_THREAD_ATTACH.
> - Any threadprivate variable that was set when the MPI_THREAD_ATTACH call occurred and was not updated afterward is guaranteed to have the same value when the call to MPI_XX occurs.

Since threadprivate variables are guaranteed to persist only within parallel sections (with the two exceptions listed above), normally, threads will attach to endpoints at the start of a parallel section, use the endpoints within the parallel section, and detach at the end of the parallel section. The attach and detach calls can be avoided for consecutive parallel sections that fulfill the conditions listed above. Within a parallel section, the user has to ensure that calls pertaining to an endpoint occur on a thread that attached that endpoint. It also has to ensure that handles to MPI objects are preserved through the parallel section; this is best done by using threadprivate variables for these handles.

We illustrate with several examples. The examples are used to illustrate corner cases in our definitions – not to indicate recommended programming practices.

Missing: Need to compile the example to check for correctness

Listing 2: Correct Use

```
#include <omp.h>
#include <stdio.h>
#include <ompi.h>

int main() {

  int max_endpoints, size, rank, max, max_threads;
```

```
 8
 9   omp_set_dynamic(0);
10   MPI_Init_endpoint(NULL, NULL, &max_endpoints, &size, &rank);
11   max_threads = omp_get_max_threads();
12  max = max_endpoints < max_threads? max_endpoints : max_threads;
13   MPI_Endpoint endpoints[max];
14   int thread_ranks[max];
15   MPI_Endpoint_create(max, endpoints);
16
17  #pragma omp parallel num_threads(max)
18      {
19      int my_thread_num = omp_get_thread_num();
20      MPI_Thread_attach(endpoints[my_thread_num],
21                                         MPI_THREAD_FUNNELED);
22
23      #pragma omp barrier
24      MPI_Gather(*my_thread_num, 1, MPI_INT, thread_ranks, max,
25                               MPI_INT, 0, MPI_COMM_PROCESS);
26      }
27   for (int j=0; j<max; j++) printf("%d, ", thread_ranks[j]);
28   printf''\n'');
29
30   MPI_Finalize();
31  }
```

Listing 2 demonstrates correct usage of MPI with OpenMP.

In line 7, OpenMP is set to use a fixed number of threads; in line 12, `max` is set to the lesser of the number of available threads and the number of endpoints. Thus, the parallel section that starts in line 17 is executed by exactly max threads, each executing one instance of the section body.

In the parallel section, threads within each process communicate using MPI. Each thread is associated with one endpoint. Master thread 0 gathers the ranks of all threads in the process, next print these ranks, after the parallel section has terminated.

It follows that the output of the process consists of the ordered list 0, 1, ..., max-1.

The OpenMP barrier call at line 22 has no effect. We inserted it to demonstrate that, within a parallel construct, tasks are tied to threads: Each task will resume, after the barrier, on the same thread it executed before the barrier, and the value of threadprivate variables is preserved across the barrier.

Note that the receive buffer argument (thread_ranks) is significant only at the root: we do not have conflicting writes into that buffer.

If we replace the text on line 9 with omp_set_dynamic(1), then OpenMP will use dynamic scheduling and the team associated with the parallel section could have less than max threads. The program will have created superfluous endpoints and, more importantly, the Gather function has a wrong receive count argument. We show haw to handle dynamic scheduling in Listing 3.

Listing 3: Handling a Dynamic Number of Threads

```
1 #include <omp.h>
```

```
 2 #include <stdio.h>
 3 #include <ompi.h>
 4
 5 int main()
 6   {
 7   int max_endpoints, size, rank, max_threads;
 8
 9   omp_set_dynamic(1);
10   max_threads = omp_get_maxthreads();
11   MPI_Init_endpoint(NULL, NULL, &max_endpoints, &size, &rank);
12   MPI_Endpoint endpoints[max_threads];
13   int indices[max_threads];
14
15   #pragma omp parallel num_threads(max_endpoints)
16       {
17       int my_thread_num = omp_get_thread_num();
18       int num_threads = omp_get_num_threads();
19
20     #pragma omp master
21         {
22         MPI_Endpoint_create(num_threads, endpoints);
23       }
24
25     MPI_Thread_attach(endpoints[my_thread_num],
26                                                  MPI_THREAD_FUNNELED);
27     MPI_Gather(&my_thread_num, 1, MPI_INT, indices, num_threads,
28                MPI_INT, 0, MPI_COMM_PROCESS);
29   }
30   for (int j=0; j<max_all; j++) printf("'%d, ", thread_ranks[j]);
31   printf("\n");
32
33   MPI_Finalize();
34   }
```

The endpoints are initialized within the parallel section, at which point the number of threads in the executing team is known and fixed. The initialization is enclosed within a master section, which executes only on the master thread. The num_threads clause on line 15 ensures that the parallel section is executed by at most max_endpoints threads, so that each thread can attach to a distinct endpoint.

We next illustrate the use of parallel loops.

Listing 4: Incorrect Parallel For Loop

```
1 #include <omp.h>
2 #include <stdio.h>
3 #include <ompi.h>
4
5 int main()
6   {
```

```
7    int max_endpoints, max_threads, max;
8    omp_set_dynamic(0);
9    MPI_Get_max_endpoints(&max_endpoints);
10   max_threads = omp_get_max_threads();
11   max = max_threads < max_endpoints ? max_threads : max_endpoints;
12   MPI_Endpoint endpoints[max];
13   int indices[max];
14   MPI_Endpoint_init(NULL, NULL, max, endpoints);
15
16   #pragma omp parallel for num_threads(max)
17   for(int i=0; i< max; i++)
18     {
19     MPI_Thread_attach(endpoints[i], MPI_THREAD_FUNNELED);
20     MPI_Gather(&i, 1, MPI_INT, indices, max,
21         MPI_INT, 0, MPI_COMM_PROCESS);
22     }
23   for (int j=0; j<max_all; j++) printf("%d, ", indices[j]);
24   printf("\n");
25   MPI_Finalize();
26   }
```

The code in Listing 4 is identical to the one listed in 2, except that we used on line 16 a parallel loop construct, rather than a parallel section. While it is quite likely that the OpenMP runtime will allocate one iteration to each thread, there is no guarantee this will happen, since the user did not specify which scheduling policy is to be used. In particular, the scheduler could allocate more than one iteration to the same thread, possibly causing a deadlock at the collective MPI_Gather call.

This problem can be alleviated by specifying which schedule should be used. We replace the statement on line 16 with

```
#pragma omp parallel for num_threads(max), schedule(static,1)
```

then iterations are scheduled statically in chunks of one iteration each. Therefore, each of the max threads in the team will execute one iteration, where thread i executes the i-th iteration.

Suppose we replace line 16 with

```
#pragma omp parallel for num_threads(max), schedule(dynamic,1)
```

Then each thread repeatedly requests chunks of size one and execute them, until no work is left. If the call to MPI_GATHER blocks, then each thread will pick one iterate. The program will complete, and the process will print the ranks of all threads, possibly out of order

However, the call to MPI_GATHER does not necessarily block until all endpoints have invoked the function. It is possible that a thread will invoke MPI_GATHER twice, for two different loop index values, but the same endpoint, causing a deadlock.

Advice to implementors. Implementation will be simplified if the MPI library uses the same mechanism for managing thread private variables as the OpenMP runtime. This can be achieved by using the ompi.h header file to list each MPI internal data structure that should be thread private in a threadprivate clause. (*End of advice to implementors.*)

0.7 TBB Binding

The TBB library [14] adds to C++ classes that implement generic parallel algorithms, such as pipelines or divide-and-conquer. The methods provided enable to decompose it a problem into subproblems, and handle the interaction between the subproblems. The user will typically provide a routine that is invoked to solve a subproblem sequentially, when it is not possible or desirable to further decompose it. The TBB run-time uses work-stealing for task scheduling. Task scheduling is nonpreemptive, so that, once scheduled on a thread, a sequential solver will run to completion on that thread. Execution starts with one sequential thread.

This suggests the following approach for interoperability with MPI:

- Initialization (calling MPI_INIT_ENDPOINT) should occur in the initial sequential part of the code, before TBB methods are called.
- A task may attach to an endpoint when executing a sequential task that does not further split; it should detach from that endpoint before it completes.

We now detail how this approach works for the main TBB constructs:

parallel_for: An endpoint can be attached to within the body operator method (Body::operator()) – the method applied to a range that is not divisible) – provided this operator does not invoke any parallel method. The endpoint should be detached before the method exits. The same applies to the body operator method of parallel_reduce, the two body operator methods of parallel_scan, the body operator method of parallel_do and the operator method of the filter class (that implements pipelines).

Missing: Need to discuss containers.
Also, a TBB maven needs to check above text
May need a special header file.

0.8 PGAS Binding

0.8.1 Introduction

PGAS languages such as UPC and Fortran 2008 provide a model of a fixed number of "locales" (*thread* in UPC, *image* in Fortran 2008) each with one executing thread. All threads execute the same program. The language supports private variables that are accessible only at one locale; and partitioned global arrays that can be accessed by all threads. Access to a private variable is as efficient as a regular memory access; access to a global array may be more expensive – especially so if the variable accessed is not in the local partition.

In addition, UPC has a work-sharing construct, upc_forall. The arguments of the construct determine which thread executes each iteration.

The mechanisms for specifying the number of locales in an execution are external to the language: the number is specified at compile time or load time. However, both languages provide functions for querying the number of locales.

Implementations may use either a separate single-threaded process for each locale; or, they may support multiple locales within each address space, with one executing thread for each locale. The optimal choice of the number of locales per process is system, implementation and application dependent.

PGAS languages can be used in HPC in two ways:

Local mode, where a PGAS program is used to control a shared-memory node, while message-passing (MPI) is used across nodes. The use of a PGAS language provides control of locality, hence possibly improved performance on NUMA systems; a good PGAS implementation can optimize for the case where all locales are in the same shared address space, so that all accesses – be it to local variables or to global variables – are implemented as regular loads and stores. Interoperability with MPI is needed in order to develop hybrid programs (PGAS intranode, MPI internode).

Global mode, where a PGAS program controls execution on a distributed memory system, replacing MPI and providing a possibly more convenient or more performing communication model. Interoperability with MPI is needed in order to invoke MPI libraries from the PGAS program, or vice-versa.

0.8.2 Execution Model

Motivation

Our goal is to provide a model for UPC (Fortran 2008) interaction with MPI that

- supports both the local and global usage modes
- works in the same manner for UPC and Fortran 2008.
- enables programs to be written so that their outcome does not depend on the number of locales per address space.

We illustrate below several possible configurations of UPC+MPI. Figure 7 shows a UPC program with 16 threads that invokes MPI on each thread. The 16 threads are on two nodes, where each node has two processes, each with four threads. However, the correctness of the program should not depend on the configuration of the UPC program: The program should produce the same results, whether the sixteen UPC threads are on one address space in one node, two address spaces with eight threads each on two nodes, or the configuration illustrated.

Figure 8 illustrates another possible configuration: Each node executes one UPC program; internode communication is provided by MPI, while intranode communication can use either UPC or MPI. MPI can be invoked at each thread. Again, the internal setup of each UPC program should not impact the program semantics.

We may not want an MPI endpoint at each UPC thread. In particular, we may prefer a model of one MPI endpoint per UPC program, as illustrated in Figure 9. Again, the program should be written for one MPI endpoint per UPC program, and should yield the same answers whether the eight threads of each UPC program run in one address space, multiple address spaces, one node or multiple nodes. (Of course, performance may vary.)

Finally, we may want a compromise of more than one endpoint per UPC program, but fewer endpoints than threads. This is shown in Figure 10, where we have one endpoint per four threads.

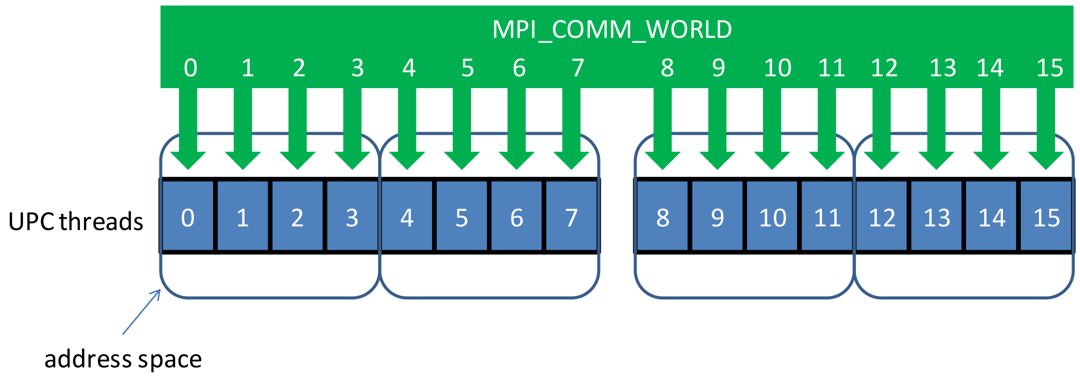

Figure 7: Global UPC Program Invokes MPI at Each Thread

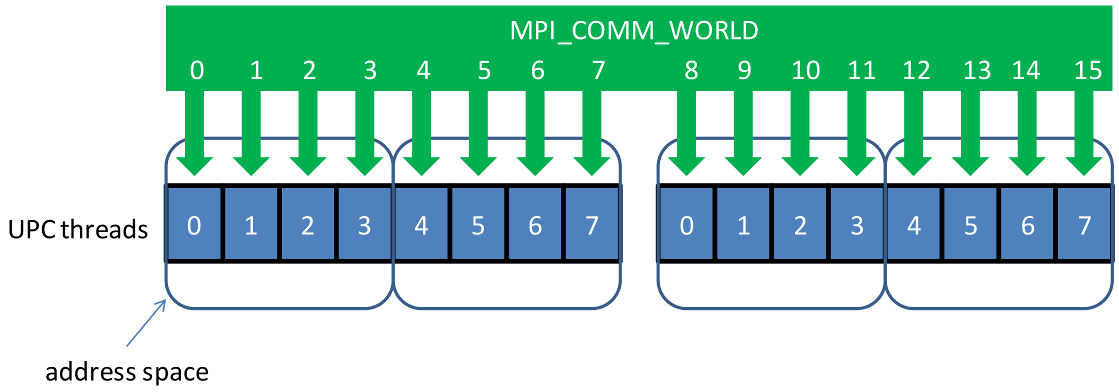

Figure 8: MPI Program with Two UPC Programs that Invoke MPI on Each Thread

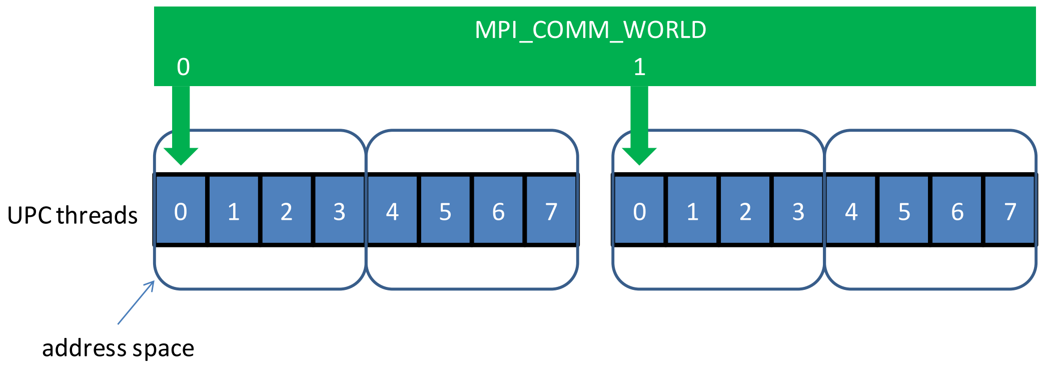

Figure 9: MPI Program with Two UPC Programs; Each UPC Program Has One Endpoint

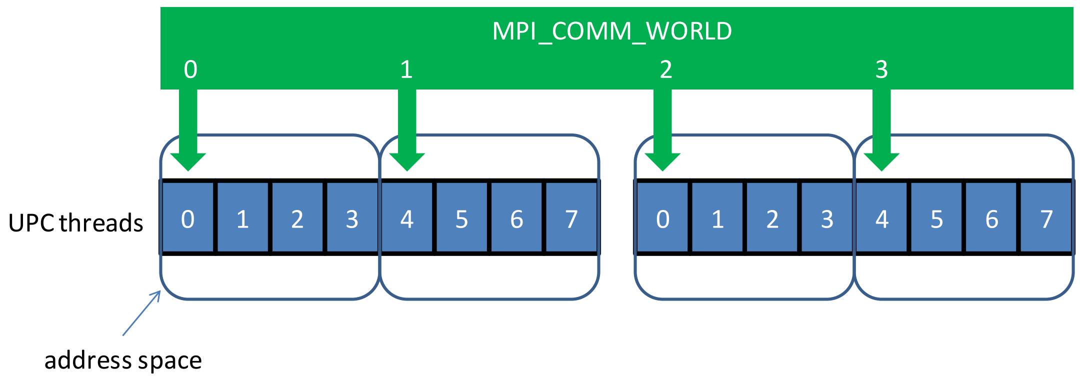

Figure 10: MPI Program with Two UPC Programs; Each UPC Program Has Two Endpoints

The last configuration happens to have one MPI endpoint per address space, in the particular UPC implementation that is illustrated in Figure 10. This may be desirable, from a performance view-point. However, the outcome of the execution should be the same if we had fewer or more address spaces.

PGAS+MPI

The required portability can be provided by using endpoints and ensuring that the behavior of the MPI code does not not depend on the number of endpoints (and locales) that share each address space:

- When the program executes UPC (or Fortran 2008) code, then each program executes independently, according to the semantics of UPC (or Fortran 2008). Thus, in Figure 7 we have one UPC execution; in Figure 8 we have two independent UPC executions.
- Only locales bound to MPI endpoints can execute MPI calls. Thus, in the examples of Figures 7 and 8, a collective call on MPI_COMM_ENDPOINTS will involve all sixteen executing threads; in the example of Figure 9 it will involve two threads – thread zero of each of the two programs; and in the example of Figure 10 it will involve four threads, two from each UPC program.
- MPI handles cannot be shared across locales. The outcome of a program that does not fulfill this restriction is implementation dependent.
- All arguments in an MPI call must be local to the invoking thread/image (UPC: the access expressions for the arguments is not shared-qualified; Fortran 2008: the access expression has no square brackets). E.g., in UPC, the send or receive buffer in an MPI call should have affinity to the thread executing the call; the buffer argument should be a private pointer to private.

  Advice to users. Programmers can make sure that MPI calls will occur only on locales attached to endpoints by predicating the execution on the value of MYTHREAD, in UPC, or the value of THIS_IMAGE() in Fortran 2008. (*End of advice to users.*)

  Rationale. We want to ensure that buffer arguments are addresses in local memory, rather than global references. (*End of rationale.*)

0.8.3 Initialization

UPC programs that invoke MPI must include the header file upcmpi.h, instead of mpi.h ; Fortran 2008 programs that use more than one image and invoke MPI must include the module cafmpi, instead of mpi.

The configuration of a hybrid PGAS/MPI execution is set out externally to the program, at compile or load time. The mechanisms for doing so are implementation-specific. The configuration includes

- the number of independent UPC (Fortran 2008) programs that are part of the execution
- the binary executed by each independent program
- the number of locales in each program; different programs may use a different number of locales.

The functions MPI_GET_UPC_NUMPROGS and MPI_GET_UPC_CONFIG (respectively MPI_GET_CAF_NUMPROGS and MPI_GET_CAF_CONFIG can be used to query the initial configuration in a UPC (respectively Fortran 2008) hybrid program. They can be invoked before MPI is initialized and can be called on each locale.

MPI_GET_UPC_NUMPROGS(num_progs)

OUT	num_progs	number of distinct UPC programs (integer)

```
int int MPI_Get_UPC_numprogs(int* num_progs)
```

MPI_GET_UPC_CONFIG(num_progs, my_prog, num_threads)

IN	num_progs	number of distinct programs (integer)
OUT	my_prog	index of program of the calling thread (integer)
OUT	num_threads	number of threads in each program (array of integers)

```
int int MPI_Get_UPC_config(int num_progs, int* my_prog, int* num_threads)
```

The argumentnum_threads is an array of length num_progs. The same values are returned in num_threads at all calling threads; the same value is returned in my_prog at all calling threads that belong to the same program.

Example : Assume that MPI_GET_UPC_NUMPROGS returns num_progs = 4, and MPI_GET_UPC_CONFIG returns my_prog=2, num_threads =(16,8,16,8). The computation consists of 4 independent UPC programs;the first and the third have 16 threads, while the second and the fourths have eiight threads. The calling thread belongs to the second program.

Advice to users. The calling thread can use MYTHREAD to find its rank within its program. (*End of advice to users.*)

UPC programs that use MPI must invoke (at each thread) the function MPI_INIT_UPCMPI, before calling any MPI function, other than MPI_GET_VERSION, MPI_INITIALIZED, and MPI_FINALIZED . This function initializes MPI (instead of MPI_INIT, MPI_INIT_THREAD or MPI_INIT_ENDPOINT).

MPI_INIT_UPCMPI(FLAG)

IN	flag	if flag=true then an endpoint is created and the calling thread is attached to it (boolean)

```
int int MPI_Init_UPCMPI(int *argc, char **argv, int flag)
```

MPI_INIT_UPCMPI is called by each thread of each UPC program in the execution. An endpoint is created for each thread that provides a value of flag=1. After the call to MPI_INIT_UPCMPI the threads with endpoints attached to them can execute MPI calls.

The equivalent functions for Fortran 2008 are listed below.

MPI_GET_CAF_NUMPROGS(num_progs)

OUT	num_progs	number of distinct programs (integer)

```
MPI_GET_CAF_NUMPROGS(NUM_PROGS, IERROR)
    INTEGER NUM_PROGS, IERROR
```

MPI_GET_CAF_CONFIG(num_progs, my_prog, num_images)

IN	num_progs	number of distinct programs (integer)
OUT	my_prog	index of program of the calling image(integer)
OUT	num_images	number of images in each program (array of integers)

```
MPI_GET_CAF_CONFIG(NUM_PROGS, MY_PROG, NUM_IMAGES, IERROR)
    INTEGER COUNT, NUM_PROGS, MY_PROG, NUM_IMAGES(*), IERROR
```

MPI_INIT_CAFMPI(flag)

IN	flag	if flag=true then an endpoint is created and the calling thread is attached to it (boolean)

```
MPI_INIT_CAFMPI(FLAG, IERROR)
    LOGICAL FLAG, INTEGER IERROR
```

Discussion. This design assumes that the implementation can support an endpoint for each UPC thread. This is not a strong constraint, since the implementation can chhose how many UPC threads share each address space and create more processess with fewer endpoints per process, if needed.

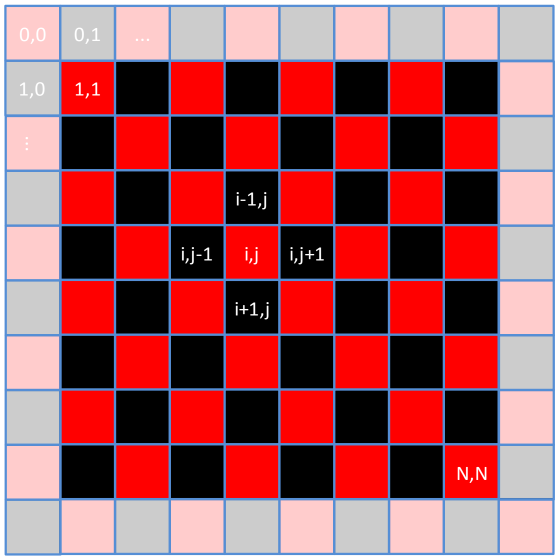

Figure 11: Red-Black SOR

We could provide a more complex design where the maximum number of endpoints in a program may be smaller than the number of threads in that program. But such a design is messy, since the true limitation is unlikely to be the number of endpoints per program, but the number of endpoints for each process in this program. We shall need to specify the numer of threads per process and the maximum number of endpoints for each such process. (*End of discussion.*)

Discussion. The Berkeley implementation of UPC supports the use of MPI from UPC programs. It seems to support only the global mode, where all threads belong to one UPC program. It also requires that UPC communication and MPI communication not occur simultaneously. We should investigate this last constraint. (*End of discussion.*)

0.8.4 PGAS + C/Fortran + MPI

Missing: It would be convenient to be able to invoke, from a UPC thread or from a Fortran 2008 image, a C or Fortran sequential program which, in turn, could invoke MPI. Fortran 2008 specifies how to invoke C; the other combinations are not supported in a standard manner, although implementations such as the Berkeley one do provide such interoperability.

0.9 Examples

We illustrate below the use of MPI with OpenMP with a schematic red-black parallel SOR code, illustrated in Figure 0.9.

At odd iterations red values are updated using the neighboring black values, and at even iterations black values are updated using the neighboring red values.

The sequential code is shown in Listing 5

Listing 5: Simplified Sequential Red-Black Code

```

#define N   10000   /* array  size  */
double  a[N+2][N+2];     /* array*/
enum COLOR {RED,  BLACK}  color;
int  i,j;
double  w;

int  main()
{

   init(a);
   while(!converged())
   {
     w = new_coefficient();
     w1 = (1.0-w)*0.25;
     for(i = 1; i <= N; i++)
       for(j = 1+(i%2)^color; j <= N; j +=2)
       a[i][j] = w*a[i][j]
                         +w1*(a[i-1][j]+a[i+1][j]
                              +a[i][j-1]+a[i][j+1]);
     color = 1-color;
   }
}
```

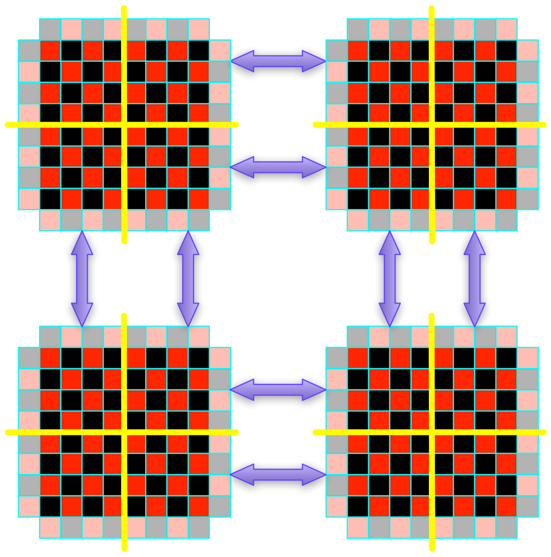

Figure 12: Red-Black Parallel SOR with Hybrid Decomposition

To simplify the parallel code, we assume that the matrix has size $N \times N$, where $N = P \times T \times m$, and m is even. The computation uses $P \times P$ processes, each working on a submesh of size $Tm \times Tm$; the process computation is split into $T \times T$ tasks, each working on an $m \times m$ submesh.This is shown in Figure 0.9, for $\mathsf{N} = 16$, and $\mathsf{P} = \mathsf{T} = 2$ and $\mathsf{m} = 4$.

We allocate four threads to handle MPI communication, each using a distinct endpoint (the number four is arbitrary), the remaining threads are allocated dynamically to execute the $T \times T$ computation tasks (we assume there are more than four threads).

OpenMP does not directly support producer-consumer synchronization. For such synchronization, it is convenient to use the following synchronization calls:

`omp_init_counter(counter)` initializes a synchronization counter.

`omp_increment_counter(counter)` atomically increments counter.

`omp_wait_counter(counter, threshold)` blocks until the counter has reached the specified threshold.

Only one thread can block on a counter. The counter is reset to zero when this thread is woken up.

These routines can be implemented using OpenMP locks, but such an implementation will result in superfluous context switches.

Listing 6: Red-Black SOR Dynamic Hybrid Code

```

#include <omp.h>
#include <stdio.h>
#include <ompi.h>

typedef enum {RED, BLACK} Color_t;
typedef enum {UP, RIGHT, DOWN, LEFT} Dir_t;




/* function to perform an iteration on a submatrix */
void compute(int len, double a[len][len], int first_row,
       int lastp1_row,  int first_col, int lastp1_col, double w,
             Color_t color)
{
  int skip = (first_row+first_column+color)%2;
  double w1 = 0.25*(1.0-w);
  for (int i=first_row; i<lastp1_row; i++) {
    for (int j=first_col+skip; j<lastp1_col; j += 2)
       a[i][j] = w*a[i][j]
                +w1*(a[i-1][j]+a[i+1][j]
                  + a[i][j-1]+a[i][j+1]);
       skip = 1-skip;
  }
}

int main() {
  /* per process sequential code */

  int max, p_size, p_rank;
  MPI_Init_endpoint(NULL, NULL, &max, &p_size, &p_rank)

  int P, T, m;  /* see problem definition */
  init1(P, T, m);

  double a[m*T+2][m*T+2];
  init2(a);

 /* compute process coordinates */
  int p_row, p_col;
  p_row = p_rank/P;
  p_col = p_rank%P;


  /* communication structures */
  /* per computation thread */
  omp_counter_t  tcounter[T][T][4][2];
  /* per communication thread */
  omp_counter_t   pcounter[4][2];


  /* initialize endpoints */
  int neighbors = 4;
  if (p_row == 0) neighbors--;
  if (p_col == P-1) neighbors--;
  if (p_row == P-1) neighbors--;
  if (p_col == 0) neighbors--;
  MPI_Endpoints endpoints[neighbors];
  MPI_Endpoint_create(neighbors, endpoints);

  /* thread parallel code */
  #pragma omp parallel sections
    {

    /* code for the four message-passing threads */
    #pragma omp section
      #pragma omp parallel for numthreads(4)
         for (Direction_t dir = UP ; i <= LEFT; dir++) {

            /* initialize */
           MPI_Comm comm, h_comm, v_comm;
           int dest;
           double *sendbuf;
           double *recvbuf;
           MPI_Datatype type;
           MPI_Request request[2][2];
           MPI_Status status[2][2];
           int i;

```

```
        /* pointers to counters of neighbor compute threads */
        omp_counter_t *neighbor_counter[T];

        /* variables used to create a communicator
        for each pair of adjacent process rows/columns */
        int hgroup, vgroup, hrank;

        switch dir {
          UP: {
              vgroup = MPI_UNDEFINED;
              if (p_row > 0) {
                  MPI_Thread_attach(endpoint[0]);
                  hgroup = p_row-1;
                  rank = p_col + P;
                  dest = p_col;
                  sendbuf = &a[1][1];
                  recvbuf = &a[0][1];
                  MPI_Type_vector(m/2, 1, 2, MPI_DOUBLE, &type);
              }
              else hgroup = MPI_UNDEFINED;
              for (i = 0; i<T; i++)
                  ncounter[i] = tcounter[0][i][UP];
              break;
           }
          RIGHT: {
              hgroup = MPI_UNDEFINED;
              if (p_col < P-1) {
                 i = (p_row != 0);
                 MPI_Thread_attach(endpoint[i]);
                 vgroup = p_col;
                 rank = p_row;
                 dest = p_row + P;
                 sendbuf = &a[T][1];
                 recvbuf = &a[T+1][1];
                 MPI_Type_vector(m/2, 1, 2*m*T, MPI_DOUBLE,
                                                    &type);
              }
              else vgroup = MPI_UNDEFINED;
              for (i = 0; i<T; i++)
                  ncounter[i] = tcounter[i][T-1][RIGHT];
              break;
            }
          DOWN: {
             vgroup = MPI_UNDEFINED;
             if (p_row < P-1) {
               i = (p_row != 0) + (p_col != P-1);
               MPI_Thread_attach(endpoint[i]);
               hgroup = p_row;
```

```
                  rank = p_col;
                  dest = p_col+P;
                  sendbuf = &a[T][1];
                  recvbuf = &a[T+1][1];
                  MPI_Type_vector(m/2, 1, 2, MPI_DOUBLE, &type);
                }
                else hgroup = MPI_UNDEFINED;
                for (i = 0; i<T; i++)
                   ncounter[i][j] = tcounter[T-1][i][DOWN];
                break;
              }
              LEFT: {
                 hgroup = MPI_UNDEFINED;
                 if (p_col > 1) {
                    i = (p_row != 0) + (p_col != P-1)
                           + (p_row != P-1);
                    MPI_Thread_attach(endpoint[i]);
                    vgroup = p_col-1;
                    rank = p_row;
                    dest = p_row - P;
                    sendbuf = &a[1][1];
                    recvbuf = &a[0][1];
                    MPI_Type_vector(m/2, 1, 2*m*T, MPI_DOUBLE,
                                                  &type);
                 }
                 else vgroup = MPI_UNDEFINED;
                   for (i = 0; i<T; i++)
                      ncounter[i][j] = tcounter[i][0][LEFT];
          }

         /* create communicator for each pair
              of adjacent process rows/columns */
         MPI_Comm_split(MPI_COMM_ENDPOINT, hgroup, rank, &h_comm);
         MPI_Comm_split(MPI_COMM_ENDPOINT, vgroup, rank, &v_comm);
         switch (dir) {
            case UP:
            case DOWN: {
               comm = h_comm;
               break;
            }
           case RIGHT:
           case LEFT: {
              comm = v_comm;
              break;
           }
         }

         for (j = 0; j < 2; j++)
```

```
            omp_counter_init( pcounter[dir][j]);


        /* computation */
       Color_t color = RED;
        while (!converged()) {
            /* wait for neighbor computing threads */
            omp_counter_wait(pcounter[dir][color], T);

             /* communicate withg neioghbor process */
             MPI_Isend(sendbuf+color, 1, *type, dest, 0,
                                 comm, &request[color][0]);
             MPI_IRecv(recvbuf+1-color, 1, *type, dest, 0,
                                 comm, &request[color][1]);
             MPI_Wait(2, request[color], status[color]);

             /* signal neighbor computing theads */
              for (i=0; i<T; i++)
                  omp_counter_signal(*(ncounter[i]+color));
           color = 1-color;
       }



/* code for computing threads */
#pragma omp section
   #pragma omp parallel for collapse(2)
   for (int t_row=0; t_row<T; t_row++)
       for (int t_col=0; t_col<T; t_col++) {

           /* initialize */

           Color_t color = RED;

           / * initial coordinates of submatrix */
           int i = t_row*m+1;
           int j = t_col*m+1;

           int k, l;
           Dir_t dir;

            /* pointers to counters of 4 neighborts */
            omp_counter_t* ncounter[4];

           for (l = 0; l < 4; l++)
               for (k=0; k<2; k++)
                  omp_counter_init( tcounter[t_row][t_col][l][k]);

            /* connect to neighbor counters */
            /* UP */
           if (t_row > 0)
              ncounter[UP] = tcounter[t_row-1][t_col][DOWN];
           else
               ncounter[UP] = pcounter[UP];

           /* RIGHT */
           if (t_col < T-1)
               ncounter[RIGHT] = tcounter[t_row][t_col+1][LEFT];
           else
               ncounter[RIGHT] = pcounter[RIGHT];

           /* DOWN */
           if (t_row < T-1)
               ncounter[DOWN] = tcounter[t_row+1][t_col][UP];
           else
               ncounter[DOWN] = pcounter[DOWN];

           /* LEFT */
           if (t_col > 0)
              ncounter[LEFT] = tcounter[t_row][t_col -1][RIGHT];
           else
               ncounter[LEFT] = pcounter[LEFT];

           /* computation */
           double w;
           while (!converged()) {
                w = new_coefficient();
                 /* UP */
                  compute(T*m+2, a, i, i+1, j, j+m, w, color);
                  omp_signal_counter(*(ncounter[UP]+color));
                  /* RIGHT */
                  compute(T*m+2, a, i+m-1, i+m, j+m-1, j+m,
                                  w, color);
                  omp_signal_counter(*(ncounter[RIGHT]+color));
                  /* DOWN */
                  compute(T*m+2, a, i+m-1, i+m, j, j+m, w, color);
                  omp_signal_counter(*(ncounter[DOWN]+color));
                  /* LEFT */
                  compute(T*m+2, a, i, i+m, j, j+1, w, color);
                  omp_signal_counter(*(ncounter[LEFT]+color));
                  /* INTERIOR */
                  compute(T*M+2, a, i+1, i+m-1, j+1, j+m-1,
                                  w, color);
                  omp_wait_counter(tcounter[t_row][t_col], 4);
                  color = 1-color;
              }
```

```
            }
        }
}
```

Missing:
Example with static thread schedulking
Example with Fortran 2008

Bibliography

[1] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.0.0, Argonne National Laboratory, 2008. 0.1.3

[2] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997. 0.1.3, 0.3.1

[3] F. Cappello, O. Richard, and D. Etiemble. Understanding performance of SMP clusters running MPI programs. *Future Generation Computer Systems*, 17(6):711–720, 2001. 0.1.1

[4] 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. 0.1.2

[5] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, 1998. 0.5

[6] C. Huang, O. Lawlor, and L.V. Kale. Adaptive mpi. *Lecture notes in computer science*, pages 306–322, 2003. 0.1.2, 0.2.4

[7] Timothy H. Kaiser and Scott B. Baden. Overlapping communication and computation with OpenMP and MPI. *Scientific Programming*, 9(2/3):73, 2001. 0.1.1

[8] Doug Lea. A java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, New York, NY, USA, 2000. ACM. 0.5

[9] Microsoft. Parallel Programming in the .NET Framework. 0.5

[10] MPI Forum. MPI: A Message-Passing Interface Standard V2.2, 2009. 0.1.2, 0.2.3, 0.2.3, 0.3.1

[11] OpenMP Architecture Review Board. OpenMP C and C++ Application Program Interface, Version 3.0, 2008. 6, 0.5, 0.6.1

[12] R. Rabenseifner, G. Hager, G. Jost, T.A.C. Center, and TX Austin. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *Proc. of 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2009)*, pages 427–236, 2009. 0.1.1

[13] J. Reid. The new features of Fortran 2008. *ACM SIGPLAN Fortran Forum*, 27(2):8–21, 2008. 6

[14] James Reinders. *Intel Threading Building Blocks Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007. 6, 0.5, 0.7

[15] Lorna Smith and Mark Bull. Development of mixed mode MPI / OpenMP applications. *Scientific Programming*, 9(2/3):83, 2001. 0.1.1

[16] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference (Vol. 1)*. MIT Press, second edition, 1998. 0.3.1

[17] Fengguang Song, Asim YarKhan, and Jack Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM. 0.3.1

[18] Richard M. Stallman and the GCC Developer Community. Using the GNU Compiler Collection (for GCC version 4.4.2). 0.2.4

[19] A. Supalov. Treating threads as MPI processes thru registration/deregistration, 2008. 0.1.3

[20] 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. 0.1.2

[21] R. Thakur and W. Gropp. Test suite for evaluating performance ofMPI implementations that support MPI_THREAD_MULTIPLE. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 14th European Pvm/Mpi User's Group Meeting, Paris France, Sept 30-October 3, 2007, Proceedings*, page 46. Springer-Verlag New York Inc, 2007. 0.1.1

[22] UPC Consortium. UPC language specifications v 1.2. 6