MPI3: Hybrid Programming

Marc Snir

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 [11, 14, 1], 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 defficiencies, as pointed 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 unessessary serialization, and reducing communication/computation overhead. If, on the other hand mutple threads access MPI, then one needs to use a thread-safe MPI implementation, which imposes a performance penalty [18]; 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. Finally, the hierarchical model of one MPI process with multiple threads may not be the best way of organizing a hybrid parallel computation: Figure 0.1.1, taken from [14] illustrates this for a 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 performane is improved) if each thread that has to communicate with a neighbor at another node has its own MPI rank. The analysis in [6] leads to the same conclusion.

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
2. Internode communication uses message passing
3. The number of MPI endpoints per process can be larger than one

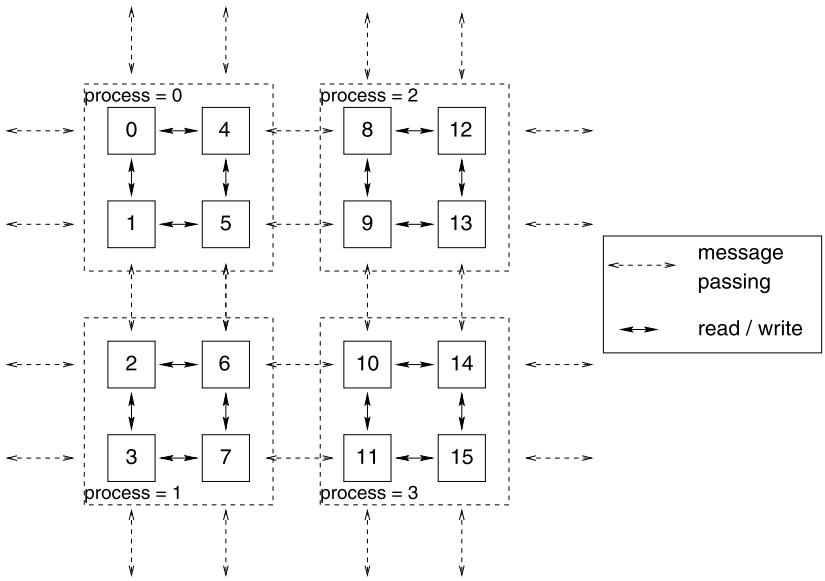

Figure 1: Halo swaps within the mixed SPMD OpenMP / MPI code

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 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 [10]
   - Programs written using PGAS languages, such as UPC [19] or CAF [12]. (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 [13]

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 interoperbility 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 match an MPI agent with an OS process. This is not an MPI requitement: The MPI standard says [9, §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 [2, 17], or even a task that can be dynamically scheduled by a run-time on different threads, for load balancing. [5]. 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 so as to provide users or libraries ways to better control resource management and so as to 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 allocate communication resources to an MPI agent.

An *MPI agent* consists of

- A set of one or more threads
- A set of communication resources allocated by the OS which we call an *MPI endpoint*.

Most (all?) MPI implementations can support multiple MPI agents within one OS image. each running in a distinct address space, as shown in Figure 0.1.2. In a single-threaded implementation, we have only one thread associated with the MPI endpoint (Figure 0.1.2, left); in a multi-threaded implementation, we can have multiple threads associated with the same endpoint (Figure 0.1.2, right).

To support this, the communication hardware and software enable the creation of multiple logically independent endpoints at a node; these endpoints can by mapped onto 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 enpoint asdsociated 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 (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 0.1.2. The design is same as before, except that the multiple MPI agents at a node can all run within a shared address space. We shall have the same choice: one thread per endpoint (Figure 0.1.2, left), or multiple threads per endpoint (Figure 0.1.2, right). This should not necessitate any major changes in the MPI library or the underlying device drivers.

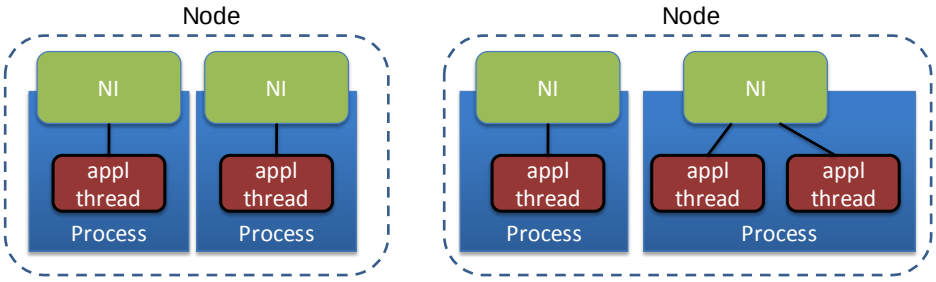

Figure 2: MPI Current Design

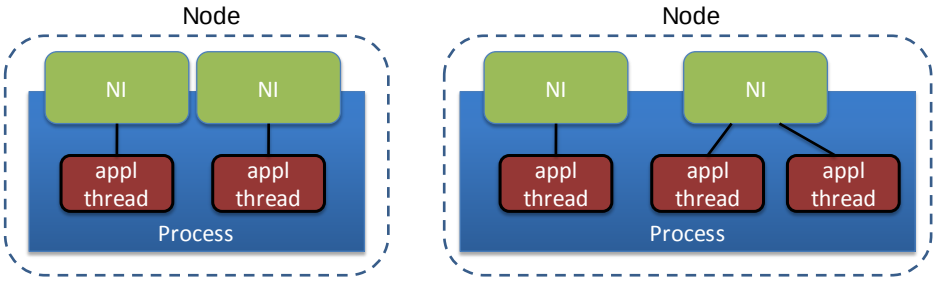

Figure 3: MPI Proposed Design

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

In the current proposal, 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.

Alternatives: The binding of threads to MPI endpoints is static and done at initialization time. This simplifies initialization but provides a more restricted shared memory model.

A thread can be attached to at most one endpoint, so that when it executes an MPI call, it is clear what 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 that MPI calls on a particular endpoint are done on a specific core, or set of cores. 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 [16] – 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.

A binding of the proposed model with POSIX thread is described in Section 0.3.

A general discussion, in Section 0.4, describes how MPI can bind to shared memory programming languages and frameworks.

Section 0.5 introduces the proposed bindings for OpenMP.

Section 0.6 discusses possible bindings to task-oriented environments, such as TBB or Cilk.

Section 0.7 describes the general principles for binding to PGAS languages and defines the bindings for UPC and Fortran 2008.

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_WORLD; when a thread that is attached to an MPI endpoint executes an MPI call with argument MPI_COMM_WORLD 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_WORLD) will appear as a send by "MPI process" with rank 5 in MPI_COMM_WORLD. Similarly, if a thread executes

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

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

0.2.2 Initialization

Discussion. The current design makes several choices that need attention (i.e. **STRAW VOTES**) by the entire forum.

- We make MPI_COMM_WORLD to be a communicator that contains all endpoints (no MPI_COMM_EWORLD). The user may compute, if it desires so, a communicator with only one end point per process, using MPI_COMM_SPLIT. This choice seems to simplify the conversion of codes that use the current MPI design to code using endpoints.
- We initialize in two phases: first phase creates endpoints and second phase attaches threads to endpoints. We could do everything in one call, but then
  - We give up on the ability to attach annd detach threads to endpoints, as threads come and go in a dynamic computation;
  - We must have all threads that will call MPI up and running at initialization time. It is more natural to do as much of the initialization as possible in the sequential preamble on a "master thread" for OpenMP or other multithreaded environments.

(*End of discussion.*)

Information Available Before Initialization

MPI_GET_MAX_ENDPOINTS(count)

OUT	count	maximum number of endpoints at local process (integer)

```
int MPI_Get_max_endpoints(int count)
```

```
MPI_GET_MAX_ENDPOINT(COUNT, IERROR)
    INTEGER COUNT, IERROR
```

The function returns the maximum supported number of endpoitns that can be created at the local process. The function can be invoked before MPI has been initialized. The function may return different values at different processes.

Alternatives: We may prefer a constant MPI_MAX_ENDPOINTS.

Discussion: The examples in Section **??** show that it can be useful for a process to know the totasl number of processes and the local process rank, in order to determin the number of endpoints it initiates. This works OK with the alternative design where initialization first creates a communicator with one endpoint per process. With the current design, we could add a function that can be invoked before MPI_INIT and that returns total number of processes and local rank.

MPI_INIT_ENDPOINT(count, endpoints)

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

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

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

The argument count must be smaller or equal to the maximum supported number of endpoints at the local process. 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 first two arguments in the C/C++ function are either the arguments of the main() function, or NULL.

> *Advice to users.* Users can query what is the maximum number of endpoitns before calling MPI_INIT_ENDPOINT to ensure that the count argument is valid. (*End of advice to users.*)

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 there is more than one endpoint at the process, 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, MPI_FINALIZED and MPI_GET_MAX_ENDPOINTS.

> **Discussion:** Should we require that all processes use the same initialization function?

The initialization call generates three communicators:

MPI_COMM_WORLD Communicator that includes all endpoints

MPI_COMM_PROCESS ommunicator that includes all process-local endpoints

MPI_COMM_SELF Communicator that contains exactly one endpoint

Endpoints within each process have consecutive ranks in MPI_COMM_WORLD and ordered in the same order as in MPI_COMM_PROCESS. This is illustrated in Figure 0.2.2.

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

MPI_THREAD_LEVEL Available level of thread support

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

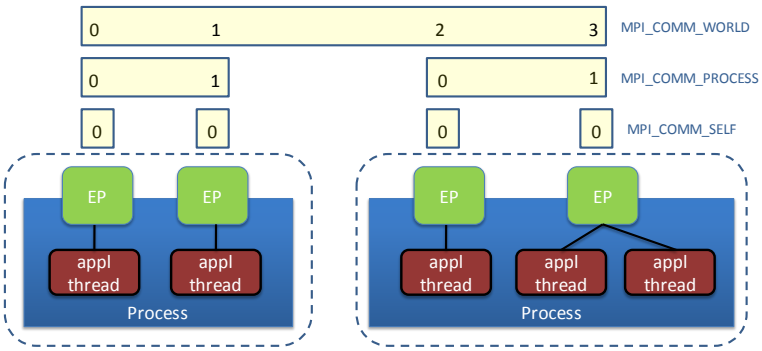

Figure 4: Communicators created by a call to MPI_INIT_ENDPOINT

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.

Alternatives: The current call does not allow the user to specify a requested level of thread support for each endpoint. To do so, would require an additional array argument.

Missing: Need to add error codes

Thread Attachment

MPI_THREAD_ATTACH(endpoint)

IN	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 endpoint. 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, MPI_FINALIZED and MPI_GET_MAX_ENDPOINTS. The call to MPI_INIT_ENDPOINT must precede the call to MPI_THREAD_ATTACH.

A thread may attach to at most one endpoint. If an endpoint thread support level is MPI_THREAD_FUNELLED then the endpoint can be attached by at most one thread.

Advice to users. Users 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.*)

Alternatives: We can have the requested level of thread support as an argument in the attach call: MPI_THREAD_ATTACH(endpoint, requested_level). Requested level must be $\leq$ the available level of thread support at that endpoint. If multiple threads attach to the same endpoint then they must all provide the same requested level argument, and must request at least level MPI_THREAD_SERIALIZED.

Discussion. We have three choices:

1. Do not provide a thread support level argument.
2. Provide a thread support level argument when the endpoint is created.
3. provide a thread support level argument when a thread is attached to the endpoint.

The choice should reflect what implementations do (or are likely to do):

1. Code behavior does not depend on level of thread support; or code behavior is determined before initialization: No use for thread support arguments.
2. Code behavior is specialized for different endpoints, according to the level of multithreading used at these endpoints; the behavior at an endpoint does not change during execution. We shall want a thread support level argument in MPI_ENDPOINT_INIT.
3. Code behavior can dynamically change during execution; e.g., if we have a phase where an endpoint is single-threaded, followed by a phase where the enppoint is multi-threaded, then the MPI library could execute differently in the two phases. We shall want a thread support level argument in MPI_THREAD_ATTACH.

Need input from implementors. (*End of discussion.*)

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.

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.

Alternatives: We can replace the two calls (to generate endpoints and to attach threads) with one call.

MPI_INIT_ENDPOINT(number_of_endpoints, rank)

IN	number_of_endpoints	number of endpoints created (integer)
IN	rank	local rank of endpoint the trhead attaches to (integer)
IN	thread_rank	rank of invoking thread (integer)

The initialization call is (as usual) collective and completes when at least one thread has attached to each created endpoint. This design does not (easily) allow to have different thread support level at different endpoints.

If we want to support the attach and detach functionality (which is quite important for environmentss with dynamic thread management). Then the initialization call will need to return an endpoint:

MPI_INIT_ENDPOINT(number_of_endpoints, rank, endpoint)

IN	number_of_endpoints	number of endpoints created (integer)
IN	rank	local rank of endpoint the trhead attaches to (integer)
IN	thread_rank	rank of invoking thread (integer)
OUT	endpoint	endpoint the calling thead attaches to (handle)

We believe that the first design is cleaner (each call deals with different objects and different operations) and easier to understand.

0.2.3 Communicating With Endpoints

A thread must be attached to an endpoint in order to make MPI calls other than MPI_INIT_ENDPOINT, MPI_GET_VERSION, MPI_INITIALIZED, MPI_FINALIZED and MPI_GET_MAX_ENDPOINTS. An MPI call by a thread uses the endpoint the thread is attached to. This endpoint determines its rank in MPI_COMM_WORLD and MPI_COMM_PROCESS and, by recursion, the rank in any communicator derived from those initial communicators.

Unless said otherwise, MPI handles returned by an MPI call of a thread attached to an endpoint can be used only by threads attached to the same endpoint. E.g., the (handle to the) communicator returned by a call to MPI_COMM_DUP should be used only by a thread that is attached to the same endpoint as the thread that executed the call. Section 0.2.4 lists exceptions to this rule: *Process-global* handles can be shared by all threads in the same address space. These typically are handles that are not associated with a communicator or an object derived from a communicator (such as window or file).

The rules and restrictions specified by the MPI standard [9, §12.4] for threads continue to apply. In particular, when a thread executes a blocking MPI call, then the calling thread may be descheduled, but other threads are not affected; two distinct threads should not block on the same request, as the MPI runtime will wake up only one thread when a request is satisfied.

A blocking MPI call will block the thread that executes the call. That thread will be rescheduled when the call completes. Since each thread can be attached to only one endpoint, deadlock situations do not arise.

> *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 processes at the node. (*End of advice to implementors.*)

Progress

MPI specifies situations where progress on an MPI call at a process might depend on the execution of matching MPI calls at other processes. 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 nonblocking send or nonblocking collective will complete irrespective of activites 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 occured at the desitnation endpoint; and a collective operation might not complete until it is invoked at all endpoints of the communicator. On the other hand, a nonblocking send or a nonblocking 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 not 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 nonblocking send will complete even if there is no thread attached to the destination endpoint; and a nonblocking collective will complete even if one of the endpoints in the communicator has no attached thread.

> *Advice to implementors.* Once an endpoint is initialized, the implementation must have created the environment needed to handle "early arrivals": An eager send may arrive before any receiving thread is attached to the destination endpoint. Note, however, that the communicator used by the eager send must have been initialized at the destination before the send occurs. The situation is not much differen from the situation obtaining when an eager send arrives bedfore a matching receive is posted. Therefore, we do not expect major implementation changes. (*End of advice to implementors.*)
>
> *Discussion.* One some systems it may be desirable to be able to attach to an endpoint a helper thread that is used as a progress engine by the MPI library. This could be achieved as follows:
>
> - A call MPI_THREAD_ATTACH_HELPER is added (or an additional argument is added to MPI_THREAD_ATTACH). This call is blocking, and the attached thread is controlled/used by MPI until the call returns.

- A call MPI_THREAD_DETACH_HELPER(endpoint) is added (or an additional argument is added to MPI_THREAD_DETACH). This call can be invoked to detach all helper threads that are attached to endpoint. The call is invoked by a thread that is not attached as a helper thread and returns immediately. The helper threads are eventually detached and return from their call to MPI_THREAD_ATTACH_HELPER.

Should discuss whether such extension is warranted. (*End of discussion.*)

0.2.4 Process-Global Objects

The following objects can be shared by all threads running within the same address space:

- Datatype
- Error handler
- Group
- Info
- Operation (Op)
- Request

Discussion: We should discuss whether sharing request objects can impact performance: this requires that accesses to request objects to always be atomic. Sharing requests make generalized requests truly useful.

As an aside: the Rationale on [9, page 373] is obsolete – it probably predates MPI with threads.

0.2.5 Interaction with Other MPI Features

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 nonblocking MPI calls at that process have completed.

> *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 [9, §6.2] can be used by all threads within a shared address space. (I.e., the value returned in the argument baseptr can be shared.)

Error Handling

When a communicator is used in a communication call, then all error handlers attached to the communicator at endpoints within the same address space must be identical. I.e., MPI_COMM_GET_ERRHANDLER(comm, errhandler) must return the same value at all endpoints that belong to the communicator and are within the same process. Thus, if a thread invokes MPI_COMM_SET_ERRHANDLER then MPI_COMM_SET_ERRHANDLER must be invoked for all other endpoints in the same address space, with the same communicator and error handler arguments.

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

Discussion: Not sure this is needed. Question to implementors: How easy/difficult would it be to have different error handlers at different endpoints within the same address space?

Process Manager Interface

The MPI_COMM_SPAWN function can be used to spawn endpoints. The argument maxprocs is interpreted to indicate the maximum number of new endpoints to create. The intercomm argument returns an intercommunicator containing the endpoints of the old communicator and the new endpoints. Information on the desired endpoint configuration is passed in the info argument. The key endpoints is reserved and can be used to indicate the desired number of endpoints per spawned process.

Alternatives: Could, instead, have a *soft* endpoint field that would specify a set of possible values.

The function MPI_COMM_SPAWN_MULTIPLE is extended in a similar manner.

Windows

All threads within a process that invoke MPI_WIN_CREATE(base, size, disp_unit, info, comm, win) as part of the same collective operation must provide the same values for base, size, disp_unit and info. Each provides its private handles for comm and win; each is returned in win a private handle to a window object. However, all these handles are pointing to the same memory window in the local address space, using the same local displacement unit. I.e., the values of the attributes MPI_WIN_BASE, MPI_WIN_SIZE and MPI_WIN_DISP_UNIT are the same for all win handles within the same process returned by a collective call to MPI_WIN_CREATE.

Generalized Requests

Requests used in generalized request functions ([9, §12.2]) can be shared by all threads running in the same address space. I.e., calls to MPI_GREQUEST_START, MPI_GREQUEST_COMPLETE and MPI_WAIT or MPI_TEST, with the same request argument can be executed on any thread.

> *Advice to implementors.* Mutual exclusion must be ensured between code that marks a request as complete and code that tests a request. Since the conflicting accesses both

occur within MPI functions, it is possible to use lock-free coordination, for enhanced performance. (*End of advice to implementors.*)

I/O

A new file access mode is defined: The mode MPI_MODE_THREAD_UNIQUE_OPEN indicates that the file is opened by one thread only. On the other hand, a file opened with MPI_MODE_UNIQUE_OPEN can be opened by mutiple threads – but by one process only.

Discussion: We may want to deprecate MPI_MODE_UNIQUE_OPEN and replace it by MPI_MODE_PROCESS_UNIQUE_OPEN.

Missing: Should decide which file hints have to be identical at all threads within a process.

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.

Discussion: Not sure this is the right design. Question to implementors: how easy/hard is it to have a file pointer per endpoint, rather than per process?

Alternative is to require that only one endpoint per process call MPI_FILE_OPEN.

0.3 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.

Advice to users. MPI program written for the regular "enpoint = process" model can be converted to use multiple endpoints per address space by

- Adding a preamble that creates the endpoints; spawns a thread for each endpoint; attaches the thread to the endpoint; and starts executing the main function of the original program.
- Replacing shared heap variables with thread-private heap variables, as needed.

In C and C++ programs, 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 [15, §5.54]. This extension is not currently supported in Fortran – we expect this to change. This (or similar) transformation can be automated with a preprocessor – see, e.g., [5]. (*End of advice to users.*)

0.4 Bindings for Shared Memory Languages and Libraries

Shared-memory parallel programming languages such as OpenMP [10], and frameworks such asTBB [13] , .NET Task Parallel Library [8], Java fork-join framework [7] and Cilk [3] provide a *task model*: A task is defined by OpenMP [10, §1.2.3] as "a specific instance of executable code and its data environment" and by TBB [13, §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, support 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.

Our definition of endpoints are based on a *thread model*: endpoints are attached to threads. In order to use endpoints in OpenMP or TBB, users have to ensure that successive invocations that use the same endpoint occur on the same thread. To do so, one need to leverage information on task scheduling, and detach and reattach threads to endpoints whenever the association of tasks to threads may change. We describe below how this is done for several shared-memory programming models.

> *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.*)

0.5 OpenMP Binding

0.5.1 OpenMP Scheduling

We briefly review the scheduling mechanism of OpenMP (references are to Version 3.0 of the OpenMP standard [10]) :

An OpenMP program begins as a single thread of execution. When a thread encounters a parallel construct [10, §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 [10, §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 [10, §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 [10, §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* [10, §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 [10, §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 level of sharing may change, 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 [10, §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 parasllel 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).

0.5.2 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 an include file ompif.h. This replaces the mpi.h or mpif.h header files normally used in MPI programs.

Alternatives: Use the same mpi.h and mpif.h include files as for regular MPI.

Discussion: Up to implementors to say whether they need different include header files.

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 occured and was not updated afterwards 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 consective 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 ednpoint. 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 1: Correct Use

```
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
9    omp_set_dynamic(0);
10   MPI_Get_max_endpoints(&max_endpoints);
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_init(NULL, NULL, max, endpoints);
16
17   #pragma omp parallel num_threads(max)
18     {
19     int my_thread_num;
20
21     my_thread_num = omp_get_thread_num();
22     MPI_Thread_attach(endpoints[my_thread_num]);
23
24     #pragma omp barrier
25     MPI_Gather(*my_thread_num, 1, MPI_INT, thread_ranks, max,
```

```
26          MPI_INT, 0, MPI_COMM_PROCESS);
27      }
28    for (int j=0; j<max_all; j++) printf("%d, ", thread_ranks[j]);
29  printf("\n");
30
31  MPI_Finalize();
32  }
```

Listing 1 demonstrates correct usage of MPI with OpenMP. The team created when the parallel construct at line 18 is executed has exactly max threads. This, because the number of threads requested (the numthread clause on line 18 is less than the number of threads available (computed at line 12); and dynamic thread allocation has been disabled on line 9. Therefore, the number of threads equals the number of endpoints, and each thread attaches to a distinct endpoint (line 23); the nonblocking collective call (at line 26) within MPI_PROCESS is performed by all threads, each with its own endpoint. The thread number was used to select which endpoint the thread attaches to; therefore it is identical to the rank within MPI_PROCESS. It follows that the output consists of the ordered list 0, 1, ..., max-1.

The OpenMP barrier call at line 25 has no effect. We inserted it to demontrate that, within a parallel construct, tasks ate 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 would have on line 9 omp_set_dynamic(1) then the team associated with the parallel section could have less than max threads.

Listing 2: 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, max_threads, num_threads, max;
8
9   omp_set_dynamic(1);
10  max_threads = omp_get_max_threads()
11  MPI_Endpoint endpoints[max_threads];
12  int indices[max_threads];
13
14  #pragma omp parallel
15    {
16    #pragma omp master
17      {
18      num_threads = om_get_num_threads();
19      MPI_Get_max_endpoints(&max_endpoints);
20      max = num_threads < max_endpoints ? num_threads : max_endpoints;
```

```
21        MPI_Endpoint_init(NULL, NULL, max, endpoints);
22        }
23
24      my_thread_num = omp_get_thread_num();
25      MPI_Thread_attach(endpoints[my_thread_num]);
26      MPI_Gather(&my_thread_num, 1, MPI_INT, thread_ranks, max,
27          MPI_INT, 0, MPI_COMM_PROCESS);
28      }
29    for (int j=0; j<max_all; j++) printf("%d, ", thread_ranks[j]);
30  printf("\n");
31
32  MPI_Finalize();
33  }
```

We list the modified code in Listing 2. 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.

We next illustrate the use of parallel loops.

Listing 3: Incorrect Parallel For Loop

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <mpi.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]);
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 3 is identical to the one listed in 1, except that we used on line 17 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 trhead, 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 17 with

```
1 #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 17 with

```
1 #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 will print the same output. (Note, however, that endpoint i may be attached by a thread $j \neq i$.)

However, the call to MPI_GATHER does not necessarily block until all enpoints have invoked the function. It is possible that a thread will invoke MPI_GATHER twice, for two different values of i, 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 openmpi.h header file to list each MPI internal data structure that should be thread private in a threadprivate clause. (*End of advice to implementors.*)

0.5.3 Example

We illustrate below the use of MPI with OpenMP with a schematic red-black parallel SOR code, illustrated in Figure 0.5.3.

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 4

Listing 4: Simplified Sequential Red-Black Code

```
1
2 #define N  10000  /*array size */
3 double a[N+2][N+2];    /*array*/
4 enum COLOR {RED, BLACK} color;
5 int i,j;
6 double w;
7
8 int main()
9 {
```

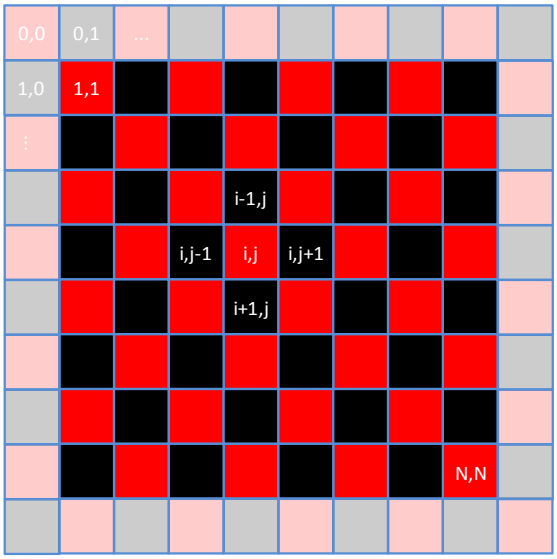

Figure 5: Red-Black SOR

```
10
11    init(a);
12    while(!converged())
13    {
14      w = new_coefficient();
15      w1 = (1.0-w)*0.25;
16      for(i = 1; i <= N; i++)
17        for(j = 1+(i%2)^color; j <= N; j +=2)
18        a[i][j] = w*a[i][j]
19                        +w1*(a[i-1][j]+a[i+1][j]
20                             +a[i][j-1]+a[i][j+1]);
21      color = !color;
22    }
23  }
```

The first parallel hybrid code shown uses the scheme shown in Figure 0.1.1: The mesh is subdivided into rectangular submeshes, each allocated to one process; each submesh is further divided in subsubmeshes, each allocated to one thread. A halo is used to buffer communication betwen threads at distinct processes; communication between threads on the same process uses shared memory, with no halo cells.

To simplify the code we assume that we have P^2 processes, each with T^2 threads, where $\mathsf{m} \times \mathsf{T} \times \mathsf{P} = \mathsf{N}$, $\mathsf{T} > 1$ and $\mathsf{P} > 1$, and m is even. Each thread computes on a subsubmesh of size $\mathsf{m} \times \mathsf{m}$, Each thread that needs to communicate with a thread on another process has an endpoint. The scheme is illustrated in Figure 0.5.3, for $\mathsf{N} = 8$ and $\mathsf{P} = \mathsf{T} = \mathsf{m} = 2$. For this case, each process needs 3 endpoints. In general, a process needs either $2\mathsf{T} - 1$ endpoints (corner); $3\mathsf{T} - 2$ endpoints (side); or $4\mathsf{T} - 4$ endpoints (middle); the total number of endpoints needed is $4(\mathsf{P} - 1)(\mathsf{PT} - \mathsf{P} + 1)$.

It would be convenient, in this case, to first generate a communicator with one endpoint

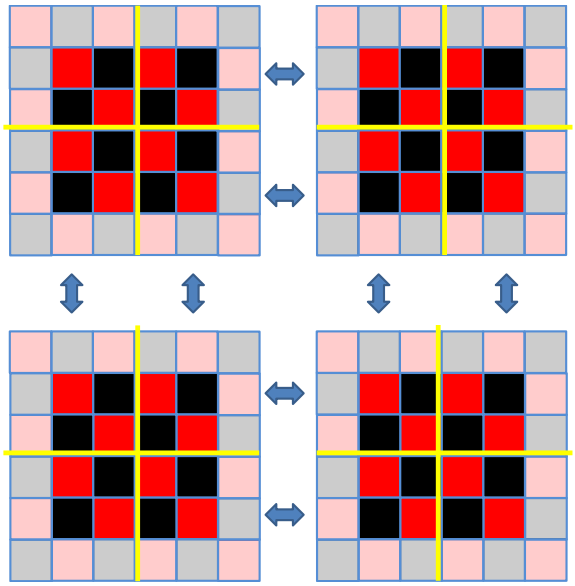

Figure 6: Red-Black Parallel SOR with Hybrid Decomposition

per process, next compute the location of each process on the 2D mesh, and use this information to generate on each process the exact number of required endpoints. Since our design requires that we generate all endpoints upfront, we shall, instead, generate $4\mathsf{T}-4$ endpoints at each process, for a total of $4(\mathsf{T}-1)\mathsf{P}^2$, but not use all.

We are using gneralized requests so as to have the same way of indicating completion of message passing and completion of shared memory communication.

The (simplified) code is shown in Listing 5

Listing 5: Red-Black SOR Static Hybrid Code

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

typedef enum Color {RED, BLACK} ColorType;

int rowlen;    /* length of submesh row */
double w, w1; /* over-relaxation coefficients */
double* a;     /* process-local submesh */
int P, T, m;   /* see problem definition */

/* function to perform an iteration on a submatrix */
void compute(double a[T*m][T*m], int first_row, int num_row,
             int first_col, int num_cols,
             ColorType color)
{
  int i, j;

  for (i=first_row; i<first_row+num_row+1; i++)
```

```
20    {
21      for (j=first_col+color; j<first_col+num_cols+1; j += 2)
22         a[i][j] = w*a[i][j]
23                   +w1*(a[i-1][j]+a[i+1][j]
24                     + a[i][j-1]+a[i][j+1]);
25      color = !color;
26    }
27  }
28
29  /* functions used by the generalized requests */
30  int gquery(void *x, MPI_Status *s) {}
31  int gfree(void *x) {}
32  int gcancel(void *x, int c) {}
33
34  int main()
35  {
36    { compute P, T, m }
37
38    double a[m*T][m*T];
39
40    { initialize a}
41
42    MPI_Init_endpoint(4*(T-1), endpoints);
43
44    /* compute datatypes for a same color row/column */
45    MPI_Datatype rowtype, coltype;
46    MPI_Type_vector(m/2, 1, 2, MPI_DOUBLE, &rowtype);
47    MPI_Type_vector(m/2, 1, 2*m*T, MPI_DOUBLE, &coltype);
48
49    /* compute process coordinates */
50    int proc_row, proc_col, proc_rank;
51    MPI_Comm_Rank(MPI_COMM_WORLD, &proc_rank);
52    proc_row = proc_rank/P;
53    proc_col = proc_rank%P;
54
55
56    /* requests for nearest neighbor communication
57       per thread, color and direction. Directions are
58       numbered clockwise, starting with UP */
59    MPI_Request sendreq[T*T][4][2], recvreq[T*T][2][4]
60
61    #pragma omp parallel num_threads(T*T)
62      {
63
64    /* color of currently updated squares*/
65    ColorType color = RED;
66
67      /* compute thread coordinates */
```

```
68      int thread_row, thread_col, thread_rank;
69      thread_rank = omp_get_thread_num();
70      thread_row = thread_rank/T;
71      thread_col = thread_rank%T;
72
73      /* compute origin of subsubmesh */
74      int first_row = thread_row*m+1;
75      int first_col = thread_col*m+1;
76
77      /* initialize requests; we only wait for receives
78         we are starting the pipeline in the correct state */
79
80      for (int dir=0; dir<4; dir++)
81      {
82        /* black received */
83        MPI_Grequest_start(&gquery, &gfree, &gcancel,
84              NULL, &recvreq[thread_rank][BLACK][dir]);
85        MPI_Grequest_complete(&recvreq[thread_rank][BLACK][dir];
86       }
87
88       /* set up red receives */
89       /* top */
90       if (thread_row == 0)
91         if (proc_row == 0)
92         {
93           /* requests for boundaries are always complete */
94           MPI_Grequest_start(&gquery, &gfree, &gcancel,
95               NULL, &recvreq[thread_rank][RED][0]);
96           MPI_Grequest_complete(&recvreq[thread_rank][RED][0]);
97         }
98         else /*proc_row > 0 */
99           /* ready to receive RED in halo */
100          MPI_Irecv( &a[0][first_col+1], 1, rowtype,
101           my_rank-P*T, 0, MPI_COMM_WORLD, &recreq[RED][0]);
102      else /* thread_row > 0 */
103          /* waiting for neighbor */
104        MPI_Grequest_start(&gquery, &gfree, &gcancel,
105              NULL, &recvreq[thread_rank][RED][0]);
106
107      /* right */
108      if (thread_col == (T-1))
109        if (proc_col == (P-1))
110        {
111          MPI_Grequest_start(&gquery, &gfree, &gcancel,
112              NULL, &recvreq[thread_rank][RED][1]);
113          MPI_Grequest_complete(&recvreq[thread_rank][RED][1]);
114        }
115        else /*proc_col < T-1 */
```

```
116           MPI_Irecv( &a[first_row+m][T*m+1], 1, coltype,
117            my_rank+T, 0, MPI_COMM_WORLD, &recreq[RED][1]);
118       else /* thread_col < T-1 */
119         MPI_Grequest_start(&gquery, &gfree, &gcancel,
120               NULL, &recvreq[thread_rank][RED][1]);
121
122        /* bottom */
123       if (thread_row == (T-1))
124         if (proc_row == (P-1))
125         {
126            MPI_Grequest_start(&gquery, &gfree, &gcancel,
127               NULL, &recvreq[thread_rank][RED][2]);
128           MPI_Grequest_complete(&recvreq[thread_rank][RED][2]);
129         }
130         else /*proc_row < P-1 */
131           MPI_Irecv( &a[T*m+1][first_col], 1, rowtype,
132            my_rank-P*T, 0, MPI_COMM_WORLD, &recreq[RED][2]);
133       else /* thread_row < T-1 */
134         MPI_Grequest_start(&gquery, &gfree, &gcancel,
135               NULL, &recvreq[thread_rank][RED][2]);
136
137       /* right */
138       if (thread_col == 0)
139         if (proc_col == 0)
140         {
141           MPI_Grequest_start(&gquery, &gfree, &gcancel,
142               NULL, &recvreq[thread_rank][RED][3]);
143           MPI_Grequest_complete(&recvreq[thread_rank][RED][3]);
144         }
145         else /*proc_col > 0 */
146           MPI_Irecv( &a[first_row][0], 1, coltype,
147            my_rank-T, 0, MPI_COMM_WORLD, &recreq[RED][3]);
148       else /* thread_col < T-1 */
149         MPI_Grequest_start(&gquery, &gfree, &gcancel,
150               NULL, &recvreq[thread_rank][RED][3]);
151
152
153     /* main body */
154     while(!converged())
155     {
156       w = new_coefficient();
157       w1 = (1.0-w)*0.25;
158       /* wait for all neighbors */
159      MPI_Waitall(4, recvreq[thread_rank][color]);
160
161       /* compute top */
162       compute(a, first_row, 1, first_col, m, color);
163       if (thread_row > 0)
```

```
164         {
165           MPI_Grequest_complete(&recvreq[thread_rank-T][color][2]);
166           MPI_Grequest_start(&gquery, &gfree, &gcancel,
167                NULL, &recvreq[thread_rank][!color][0]);
168         }
169         else if (proc_row > 0)
170         {
171           MPI_Isend(&a[1][first_col+color], 1, rowtype,
172             my_rank-P*T, 0, MPI_COMM_WORLD, &sendreq[color][0]);
173           MPI_Irecv( &a[0][first_col+color], 1, rowtype,
174             my_rank-P*T, 0, MPI_COMM_WORLD, &recvreq[!color][0]);
175         }
176
177         /* compute right */
178         compute(a, first_row, m, first_col+m-1, 1, color);
179         if (thread_col < (T-1))
180         {
181           MPI_Grequest_complete(&recvreq[thread_rank+1][color][3]);
182           MPI_Grequest_start(&gquery, &gfree, &gcancel,
183                NULL, &recvreq[thread_rank][!color][1]);
184         }
185         else if (proc_col < P-1)
186         {
187           MPI_Isend(&a[first_row+!color][T*m], 1, coltype,
188             my_rank+P*T, 0, MPI_COMM_WORLD, &sendreq[color][1]);
189           MPI_Irecv(&a[first_row+!color][T*m+1], 1, coltype,
190             my_rank+P*T, 0, MPI_COMM_WORLD, &recvreq[!color][1]);
191         }
192
193         /* compute bottom */
194         compute(a, first_row+m-1,1, first_col, m, color);
195         if (thread_row < T-1 )
196         {
197           MPI_Grequest_complete(&recvreq[thread_rank+T][color][0]);
198           MPI_Grequest_start(&gquery, &gfree, &gcancel,
199                NULL, &recvreq[thread_rank][!color][3]);
200         }
201         else if (proc_row < P-1)
202         {
203           MPI_Isend(&a[T*m][first_col+!color], 1, rowtype,
204             my_rank+P*T, 0, MPI_COMM_WORLD, &sendreq[color][2]);
205           MPI_Irecv(&a[T*m+1][first_col+color], 1, rowtype,
206             my_rank+P*T, 0, MPI_COMM_WORLD, &recvreq[!color][2]);
207         }
208
209
210         /* compute left */
211         compute(a, first_row, m, first_col, 1, color);
```

```
212         if (thread_col > 0)
213         {
214           MPI_Grequest_complete(&recvreq[thread_rank-1][color][1]);
215           MPI_Grequest_start(&gquery, &gfree, &gcancel,
216                NULL, &recvreq[thread_rank][color][3]);
217         }
218         else if (proc_row > 0)
219         {
220           MPI_Isend(&a[first_row+color][1], 1, coltype,
221             my_rank-T, 0, MPI_COMM_WORLD, request[num_req]);
222            else if (proc_row > 0)
223           MPI_Irecv(&a[first_row-1+color][0], 1, coltype,
224             my_rank-T, 0, MPI_COMM_WORLD, request[num_req++]);
225         }
226
227
228         /* compute interior */
229         compute(a, first_row+1, m-2, first_col+1, m-2, color);
230       }
231     }
232   }
```

0.6 TBB Binding

The TBB library [13] 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_ENDPOINT_INIT) 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, need to check above text – probably need help from a TBB guru.

May need a special header file.

0.7 PGAS Binding

0.7.1 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 proviate 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 querrying 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 for the number of locales per process is system, implementation and application dependent.

As vendors often use the same run-time for UPC and Fortran 2008, it will be convenient to have the same solution for both.

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.

0.7.2 Execution Model

Motivation

Our goal is to provide a logical model for UPC (Fortran 2008) interaction with MPI that supports the different models of interactions of MPI with PGAS programs. The correctness of programs should not depend on the choice of UPC (Fortran 2008) implementors of using one or multiple address spaces for one UPC (Fortran 2008) program.

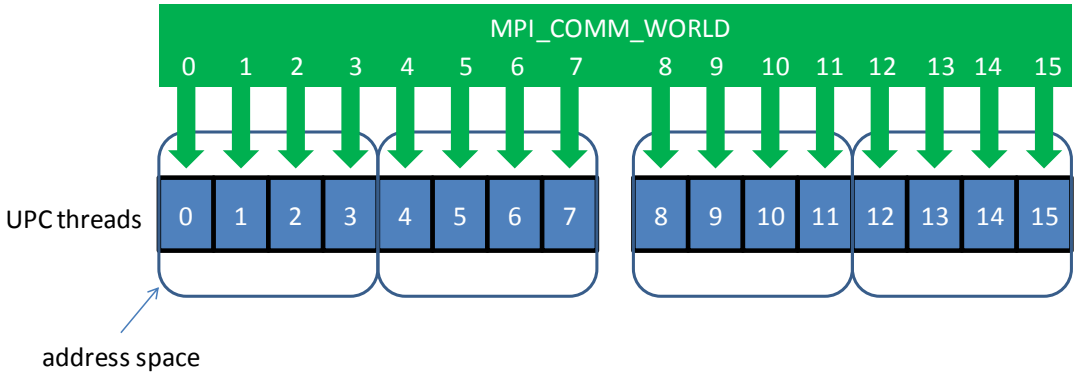

Figure 7: Global UPC Program Invokes MPI at Each Thread

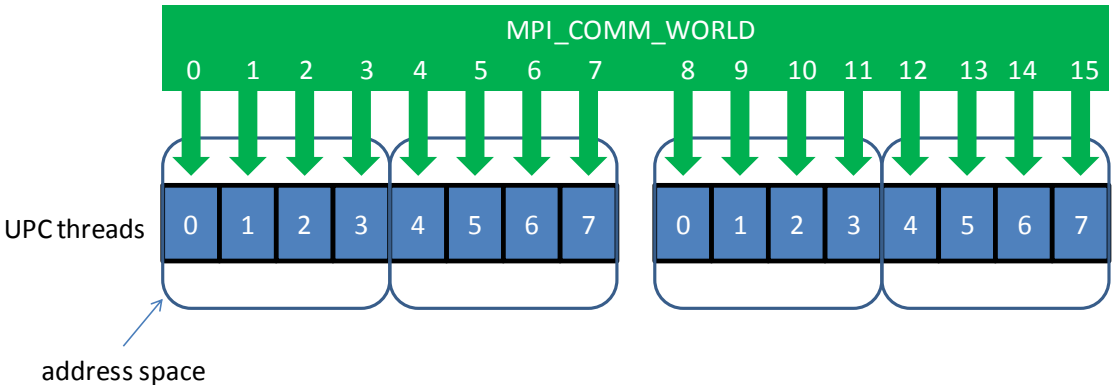

Figure 8: MPI Program with Two UPC Programs that Invoke MPI on Each Thread

We illustrate below several possible configurations of UPC+MPI. Figure 0.7.2 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 **??** 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 0.7.2. Again, the program should be written for one MPI endpoint per UPC program, and should yield the same answers wether 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 0.7.2, where we have one endpoint

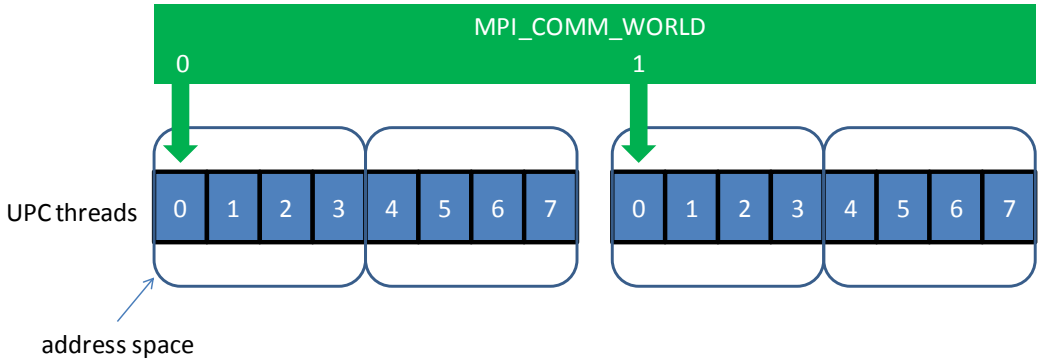

Figure 9: MPI Program with Two UPC Programs; Each UPC Program Has One Endpoint

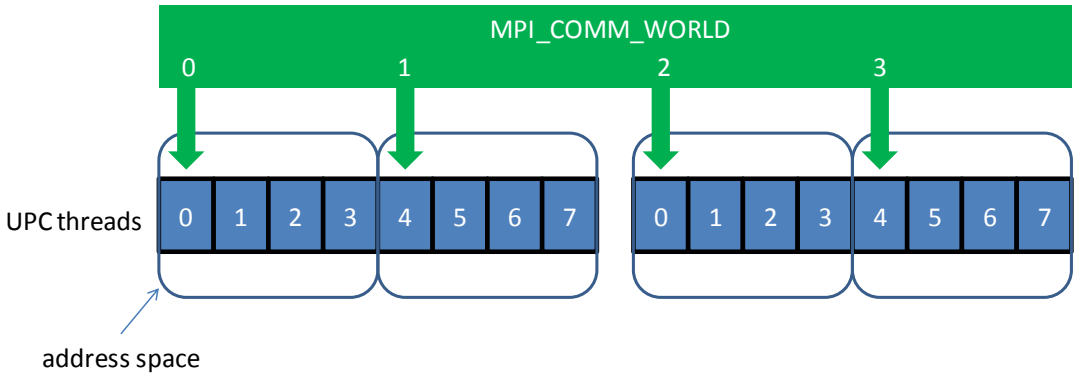

Figure 10: MPI Program with Two UPC Programs; Each UPC Program Has Two Endpoints

per four threads.

The last configuration happens to have one MPI endpoint per address space, in the particular UPC implementation that is illustrated in Figure 0.7.2. 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.

Model

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 0.7.2 we have one UPC execution; in Figure 0.7.2 we have two independent UPC executions.

Only threads (images) bound to MPI endpoints can execute MPI calls. When the program executes MPI calls then, in effect, we have a global MPI program with one execution thread per endpoint. Thus, in the examples of Figures 0.7.2 and 0.7.2, a collective call on MPI_COMM_WORLD will involve all sixteen executing threads; in the example of Figure 0.7.2 it will involve two threads – thread zero of each of the two programs; and in the example of Figure!0.7.2 it will involve four threads, two from each UPC program.

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.*)

MPI object cannot be shared across locales, unless they are one of the global MPI objects listed in Section 0.2.4. The outcome of a program that does not fulfill this restriction is implementation depdent.

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.

Rationale. We want to ensure that buffer arguments are addresses in local memory, rather than global references.

In some cases, the user may be aware that a global reference is actually implemented as a regular local memory address; e.g., when multiple UPC threads are known to be running in the same addres space. Users may want to take advantage of such a situation – but the output of such a code is implementation dependent. (*End of rationale.*)

0.7.3 Initialization

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

The function MPI_GET_UPC_CONFIG (respectively MPI_GET_CAF_CONFIG can be used to query the initial configuration in a UPC (respectively Fortran 2008) program. It can be invoked before MPI is initialized.

MPI_GET_UPC_CONFIG(programs, spaces)

OUT	programs	number of independent UPC programs in the computation (integer)
OUT	spaces	number of address spaces managed by the lcoal UPC program)

```
int int MPI_Get_UPC_config(int* programs, int* spaces)
```

The parallel computation consists of programs independent UPC executions (that can communicate using MPI). The local UPC program controls spaces address spaces; THREADS/spaces threads execute within each address space.

MPI_GET_CAF_CONFIG(programs, spaces)

OUT	programs	number of independent UPC programs in the computation (integer)
OUT	spaces	number of address spaces managed by the lcoal UPC program)

```
MPI_GET_CAF_CONFIG(PROGRAMS, SPACES, IERROR)
    INTEGER PROGRAMS, SPACES, IERROR
```

The parallel computation consists of programs independent Fortran 2008 executions (that can communicate using MPI). The local Fortrran 2008 program controls spaces address spaces; NUM_IMAGES()/spaces images execute within each address space.

Discussion: We assume that each address space within a PGAS program has the same number of locales. This is true in current implementations but may change in the future – in which case, we shall need to return arrays.

UPC programs that invoke MPI must invoke (at each thread) the function MPI_INIT_UPC, before calling any MPI function, other than MPI_GET_VERSION, MPI_INITIALIZED, MPI_FINALIZED and MPI_GET_MAX_ENDPOINTS. This function initializes MPI (insterad of MPI_INIT, MPI_THREAD_INIT or MPI_ENDPOINT_INIT).

MPI_INIT_UPC(count, stride)

IN	count	number of endpoints created (integer)
IN	stride	stride between successive endpoints (integer)

```
int int MPI_Init_UPC(int *argc, char **argv, int count, int stride)
```

This initialization call creates count endpoints that are attached in order to the count threads $0, \mathsf{stride}, 2\mathsf{stride}, \ldots, (\mathsf{count}-1)\mathsf{stride}$. The call is erroneous if any of the arguments is not positive, if the required number of endpoints cannot be generated, or if $(\mathsf{count}-1)\mathsf{stride}+1$ is larger than the number of UPC threads.

Fortran 2008 programs with more than one image that invoke MPI. must invoke (at each image) the function MPI_INIT_CAF, bfore calling any MPI function, other than MPI_GET_VERSION, MPI_INITIALIZED, MPI_FINALIZED and MPI_GET_MAX_ENDPOINTS. This function initializes MPI (insterad of MPI_INIT, MPI_THREAD_INIT or MPI_ENDPOINT_INIT).

After the call to MPI_INIT_UPC the threads with endpoints attached to them can execute MPI calls.

MPI_INIT_CAF(count, stride)

IN	count	number of endpoints created (integer)
IN	stride	stride between successive endpoints (integer)

```
MPI_INIT_CAF(COUNT, STRIDE, IERROR)
```

```
INTEGER COUNT, STRIDE , IERROR
```

This initialization call creates count endpoints that are attached in order to the count images $1, \mathsf{stride}+1, 2\mathsf{stride}+1, \ldots, (\mathsf{count}-1)\mathsf{stride}+1$. The call is erroneous if any of the arguments is not positive, if the required number of endpoints cannot be generated, or if $(\mathsf{count}-1)\mathsf{stride}+1$ is larger than the number of Fortran 2008 images.

After the call to MPI_INIT_CAF the images with endpoints attached to them can execute MPI calls.

Examples

MPI_INIT_UPC(1,1) attaches one endpoint to thread zero: The (node-local) UPC code can correspond to other nodes using MPI calls on thread zero; other threads cannot execute MPI calls. (Code can be executed only on thread zero by conditioning its execution on the value of MYTHREAD).

MPI_INIT_UPC(THREADS, 1) attaches one endpoint to each thread in the UPC program. The rank of the endpoint within MPI_COMM_PROCESS equals to MYTHREAD. All threads can execute MPI calls.

MPI_INIT_UPC(THREADS/4,4) attaches one endpoint to each fourth thread in the UPC program.

Missing: Need to add a full application example.

Alternatives: We may prefer to use the same initialization calls both for UPC and for Fortran 2008. This will work assuming consistent (and interoperable) implementations for these two languages.

0.7.4 PGAS code invoked from an MPI program.

The presious sections provide mechanisms for using MPI in order to connect multiple UPC or Fortran 2008 programs; and invoking MPI libraries form a UPC or Fortran 2008 program. They do not provide a mechanism for invoking a UPC or Fortran 2008 library from a regular MPI program. To do so, we need to pass array arguments from the MPI code to the PGAS code. I.e., we need to be able to generate, in a distributed memory MPI code, multiple local arrays that will be interpreted within the PGAS code as one global, distributed array. This would be similar (but reverse) to the extrinsic interface of High Performance Fortran [4, [§6]. Such interface, while very useful, is beyond the scope of this document – as its implementation requires support in the PGAS language run-time, but no MPI extensions.

0.8 Acknowledgements

I wish to thank J P Hoeflinger for his careful reading of a previous version of this proposal.

Bibliography

[1] 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

[2] 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

[3] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, 1998. 0.4

[4] High Performance Fortran Forum. High performance fortran language specification. *Scientific Programming*, 2:1 – 170, 1993. 0.7.4

[5] C. Huang, O. Lawlor, and L.V. Kale. Adaptive mpi. *Lecture notes in computer science*, pages 306–322, 2003. 0.1.2, 0.3

[6] 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

[7] 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. 0.4

[8] Microsoft. Parallel Programming in the .NET Framework. 0.4

[9] MPI Forum. MPI: A Message-Passing Interface Standard V2.2, 2009. 0.1.2, 0.2.3, 0.2.4, 0.2.5, 0.2.5

[10] OpenMP Architecture Review Board. OpenMP C and C++ Application Program Interface, Version 3.0, 2008. 6, 0.4, 0.5.1

[11] 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

[12] J. Reid. The new features of Fortran 2008. In *ACM SIGPLAN Fortran Forum*, volume 27, pages 8–21. ACM, 2008. 6

[13] James Reinders. *Intel Threading Building Blocks Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007. 6, 0.4, 0.6

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

[15] Richard M. Stallman and the GCC Developer Community. Using the GNU Compiler Collection (for GCC version 4.4.2). 0.3

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

[17] 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

[18] 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

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