

# A PROPOSAL FOR HYBRID PROGRAMMING SUPPORT ON HPC PLATFORMS

MARC SNIR

8/5/2009

## 1 Introduction

### 1.1 Current State

In the next few years, supercomputers will be built of nodes with a increasingly large number of cores. Indeed, much 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.

We assume that such future systems will have hardware support for multiple, concurrent communication channels at a node: they may have multiple network interfaces, and each network interface may support multiple independent communication channels. This has already been the case on existing systems.

The two most prevalent approaches to handle large SMP nodes are:

**MPI only:** One associates a process with each core; processes communicate using MPI, both between nodes and within a node.

**MPI+OpenMP:** One associates one multithreaded process with each node (each OS image) – typically with one (kernel) thread running on each core. MPI is used for communication across nodes, while OpenMP is used for handling threads within a node; the threads communicate using shared memory.

The first model leads to an inefficient use of shared memory communication, as MPI forces an additional copy, going from address space to address space. The second model may lead to other inefficiencies: If only one thread makes MPI calls (the `MPI_THREAD_FUNNELLED` model) then the single thread making these calls becomes a bottleneck. If, on the other hand, we allow all threads to make MPI calls, then we have to use the `MPI_THREAD_MULTIPLE` model that requires a thread safe MPI implementation; such implementation is often less performing than a lock-free implementation [11].

In addition, OpenMP encourages a programming style with heavy use of fine-grain dynamic resource allocation (e.g., dynamic scheduling of the iterates of a parallel loop). It is easy to write code that has high scheduling overheads, and bad locality [3].

## 1.2 Goals

We seek a way of supporting 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 end-points per node can be larger than one, but need not be equal to the number of cores / executing threads
4. MPI end-points can be dedicated, to avoid the overhead of a thread-safe MPI library

## 1.3 Possible Approaches

One possible solution is to use memory segments shared across multiple processes. Some, or all of these processes would be MPI processes. The UNIX V IPC shared memory segment calls `shmget`, `shmat`, `shmdt`, and `shmatctl` provide mechanisms for allocating memory segments that are shared by multiple processes running on the same node [12, Section 2.1]; Windows provide similar capabilities. A current proposal [2] suggests extensions to MPI that will provide a portable syntax for the creation of memory segments shared by multiple MPI processes on one node.

Process parallelism was used quite frequently several decades ago, since thread support was rudimentary. As operating system support for threads improved, process parallelism has been discarded in favor of thread parallelism. Thread parallelism provides more flexibility, as all variable can be shared; it facilitates resource management, since each node is associated with one user process. (Shared memory segments persist even if the attached processes have died, and have to be reclaimed separately.) Shared memory languages such as OpenMP [4], or frameworks such as TBB [9], provide a higher-level task abstraction, and handle the mapping of tasks to threads in the run-time. It is not clear that there are compelling reasons to reverse this evolution. True, task parallel constructs in OpenMP or TBB are focused on load balancing and often encourage a programming style that leads to poor locality. We believe this is better handled by defining an OpenMP programming style that promotes locality, rather than by reverting to lower level, less flexible sharing mechanisms.

Another approach is to improve multithreading support on MPI implementations, so that the overhead of sharing communication resources becomes insignificant. Note, however, that as the number of cores per node increases, systems are becoming more “NUMA-ish”: there is a significant difference in access time to local and remote memory; and cache-to-cache transfer time also varies. We may want to have more algorithmic control on the association of threads to MPI endpoints to computational threads.

The approach we propose in this document is to provide support to a model where kernel threads, rather than processes, serve as *MPI endpoints*. In this approach, each core (or a subset of the cores) could function as “MPI processes”, while communicating with other cores on the same node using shared memory, within the same address space.

The proposed solution consists of three parts:

1. Proposed extensions to MPI to enable the association of multiple MPI end-points with distinct threads within one process. These are described in Section 2. This proposal modifies and expands the proposal presented at the MPI-3 forum by Alexander Supalov [10] – 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 processes” equal to the number of distinct communication channels that the hardware supports: *The MPI endpoints will be created at initialization time, and will not change during execution.* The proposed approach does not require changes on how “MPI processes” are identified, and does not even require a thread-safe MPI library. Therefore, we expect that changes to current implementations will be modest.
2. A proposed binding of these MPI extensions to OpenMP. This is described in Section **Error! Reference source not found.**
3. A “manual of style” for the development of OpenMP codes with good locality. This is outlined in Section **Error! Reference source not found.**

We also discuss in Section 5 extensions to this proposal.

## 2 MPI Support of Multiple Endpoints per Process

### 2.1 Proposed Model

To avoid confusion, we shall use the following definitions:

*Process:* OS process – i.e., an entity consisting of an address space, an executable program, and other OS resources (such as file descriptors).

*Thread:* Kernel thread – i.e., a unit of execution consisting of a set of registers, a stack, etc. Each process has at least one thread; if it has multiple threads, then they share the same address space. Threads are preemptable and scheduled by the operating system.

*Fiber:* User thread – i.e., a unit of execution that is scheduled on a thread by the run-time. A fiber is non preemptable (the OS can preempt the thread that executes the task, but cannot preempt the task and allocate another task to the thread). Fiber scheduling is cooperative, and fibers have to yield in order to enable another task to be scheduled on the executing thread.

*Endpoint:* 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. The “endpoint” concept replaces the “MPI process” concept, to avoid confusion with OS processes. Each process is associated with one or more endpoints.

The fundamental insight of the proposed design is that we are separating the concept of an endpoint from the concepts of process or thread: A process may be associated with multiple endpoints; the association of threads within this process with endpoints can be dynamic.

In a static MPI model, there is a one-to-one correspondence between endpoints and ranks in

`MPI_COMM_TWORLD`<sup>1</sup>; we can index endpoints using the corresponding rank in `MPI_COMM_TWORLD`. When new communicators are created, each rank in a newly created communicator corresponds to a distinct rank in the input communicator. (In the old terminology, when a new communicator is created, each “MPI process” is associated with a unique new rank in the newly created communicator that contains it). Thus, by induction, each rank in a communicator is associated with an endpoint – and distinct ranks within a communicator correspond to distinct endpoints.

At any point in time a thread is associated with at most one endpoint; when this thread executes MPI calls, it executes calls directed to that endpoint. Thus, if a thread is associated with endpoint 5, then a call to `MPI_SEND(..., MPI_COMM_TWORLD)` will appear as a send by the “MPI process” with rank 5 in `MPI_COMM_TWORLD`. Similarly, if a thread executes

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

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

**Discussion:** If we redesigned MPI from scratch then we could add to each MPI call an explicit caller rank – thus explicitly identifying which endpoint is used by the call; the association of threads to endpoints could change dynamically at each MPI call. But this would require a syntax change for the large majority of MPI calls. To avoid this, we separate between the operation that associates a thread with an endpoint and the subsequent MPI calls that use the endpoint; the “endpoint” or “caller rank” argument is implicit in the MPI calls. We expect that the most important use of the new design will be one where the association of threads to endpoints is not changing during computation, or is changing rarely.

## 2.2 Initialization

**Discussion:** We have two possible designs. (a) Have a new `MPI_INIT` call that creates upfront an “`MPI_COMM_WORLD`” with more than one port per process; or (b) start with one port per process in `MPI_COMM_WORLD`, and add new ports afterward. We choose the second design, in order to facilitate support for heterogeneous systems, where one might want to create a different number of ports at different processes. To do so, we need to have a process id and/or information on the processor name; these, in MPI, are associated with `MPI_COMM_WORLD`.

We add a new predefined attribute `MPI_ENDPOINTS` that has integer type and holds the maximum number of MPI endpoints that can be created at the local process. This attribute is handled as other predefined attributes in MPI – see [7, §16.3.7]

---

<sup>1</sup> We are not handling, for the time being, the dynamic process model, but the extension of the current proposal to support dynamic endpoint creation is not conceptually hard. Note that the interaction between thread support model and dynamic process creation is not clearly defined by the MPI standard: Do newly created processes inherit the same level of thread support as the processes in the original communicator? This will need to be clarified, before we can discuss dynamic endpoint creation.

A program that uses multiple endpoints per process must initialize by calling `MPI_INIT_ENDPOINT`, next calling `MPI_ENDPOINT_CREATE`.

```
MPI_INIT_ENDPOINT(required, provided)
IN required      desired level of thread support (integer)
OUT provided     provided level of thread support (integer)

int MPI_Init_endpoint(int *argc, char *((*argv)[]), int required,
                     int *provided)

MPI_INIT_ENDPOINT(REQUIRED, PROVIDED, IERROR)
INTEGER REQUIRED, PROVIDED, IERROR

int MPI::Init_endpoint(int& argc, char**& argv, int required)
int MPI::Init_endpoint(int required)
```

This first call has the same arguments as `MPI_INIT_THREAD` [7, §12.4.3]. This call sets the thread support mode provided and creates `MPI_COMM_WORLD` with one endpoint at each process. After this call, `MPI_COMM_WORLD` can be queried, e.g. to find the processor name, or to find `MPI_ENDPOINT`, the number of supported endpoints.

As for `MPI_THREAD_INIT()`, the argument `required` specifies the required level of thread support, while the argument `provided` returns the actual level of thread support provided. The meaning of the possible values is as follows:

**`MPI_THREAD_SINGLE`:** Only one thread can be associated with each endpoint; each thread is associated with an endpoint.

**`MPI_THREAD_FUNNELLED`:** Only one thread can be associated with each endpoint; the process may have additional threads that are not associated with any endpoint.

**`MPI_THREAD_SERIALIZED`:** Multiple threads may be associated with the same endpoint. However, MPI calls using the same endpoint cannot be made concurrently by two distinct threads.

**`MPI_THREAD_MULTIPLE`:** Multiple threads may be associated with the same endpoint and make concurrent MPI calls.

Implementations can support only some of these modes.

The second call has the following syntax:

```
MPI_ENDPOINT_CREATE(num_endpoints,array_of_endpoints)
  IN  num_endpoints      number of endpoints (integer)
  OUT array_of_endpoints  array of endpoint handles (array of handles)

int MPI_Endpoint_create( int num_endpoints, MPI_Endpoint
                        *array_of_endpoints)

MPI_ENDPOINT_CREATE(NUM_ENDPOINTS, ARRAY_OF_ENDPOINTS, IERROR)
  INTEGER NUM_ENDPOINTS
  INTEGER ARRAY_OF_ENDPOINTS(MPI_ENDPOINT_SIZE,*)
  INTEGER IERROR

int MPI::Endpoint_create(int num_endpoints,
                        MPI::Endpoint array_of_endpoints[])
```

The second routine must be called before any communication occurs; it should be called at most once on each process. The call generates `num_endpoints` MPI endpoints at the calling process. It returns an array of handles to these endpoints in `array_of_endpoints`. This argument should be an array of length at least `num_endpoints`. The call is erroneous if `num_endpoints > MPI_ENDPOINTS`.

The call is collective; it will generate a communicator `MPI_COMM_WORLD` that includes all processes and has `num_endpoints` endpoints at each calling process. It will also create at each process a communicator `MPI_COMM_PROCESS` that is local to the process and contains all the endpoints at that process. At each process, the endpoint with rank 0 in `MPI_COMM_PROCESS` is also an endpoint in `MPI_COMM_WORLD` (i.e. `MPI_COMM_WORLD` contains the first endpoint of each process).

#### Implementation notes:

1. Communication using the `MPI_THREAD_SINGLE` 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.
2. We chose to define a new function `MPI_INIT_ENDPOINTS`, rather than reuse `MPI_INIT_THREAD`, to facilitate implementation on systems where it is inconvenient to change the number of ports dynamically.

#### Discussion:

If the current design (two initialization functions) is considered ugly, then the alternative is to have one combined initialization function, but have some predefined “load-time constants” that provide information on the id and type of each process and the number of endpoints it supports.

## 2.3 Registration

Thread registration functions associate and disassociate a thread with an endpoint.

```
MPI_THREAD_REGISTER(endpoints, i)
  IN endpoints      array of endpoint handles (array of handles)
  IN index          index of endpoint to be used by thread
```

```
int MPI_Thread_Register(MPI_Endpoints *endpoints, int index)
```

```
MPI_THREAD_REGISTER (ENDPOINTS, INDEX, IERROR)
INTEGER ENDPOINTS(MPI_ENDPOINT_SIZE,*)
INTEGER INDEX, ENDPOINT
```

```
int MPI::Endpoint::Register(MPI::Endpoint endpoints[], int index)
```

The invocation of this call by a thread associates the invoking thread with the corresponding endpoint. Each thread can be registered with at most one endpoint at any point in time. If the thread support level is `MPI_THREAD_SINGLE` or `MPI_THREAD_FUNELED` then each endpoint can be registered by only one thread. If the thread support level is `MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE` then multiple threads can be registered with the same endpoint at the same time.

```
MPI_THREAD_UNREGISTER(endpoints, i)
  IN endpoints      array of endpoint handles (array of handles)
  IN index          index of endpoint to be disassociated from thread (integer)
```

```
int MPI_Thread_Unregister(MPI_Endpoints *endpoints, int index)
```

```
MPI_THREAD_UNREGISTER (ENDPOINTS, INDEX, IERROR)
INTEGER ENDPOINTS(MPI_ENDPOINT_SIZE,*)
INTEGER INDEX, ENDPOINT
```

```
int MPI::Endpoint::Unregister(MPI::Endpoint endpoints[], int
index)
```

This call disassociates the calling thread from the specified endpoint. This function should be invoked only when there are no pending local MPI calls on the specified endpoint.

The rules and restrictions specified by the MPI standard [8, §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 thread must be registered with an endpoint before making MPI calls (other than queries on the predefined attributes of `MPI_COMM_WORLD`); the MPI calls will use the corresponding

endpoint. If an endpoint is the target of a communication (e.g., the receiver of a send, or a party to a collective communication) then the communication operation (i.e. the send or the collective operation) may not complete until a thread has registered with that endpoint.

A blocking collective call will block all the threads that execute the call. Note that since each thread can be associated only with one endpoint, and cannot change its association while there are pending calls, then each thread executes the call only once, so that deadlock situations do not arise.

## 2.4 Example

```

-
... /* omit declarations */
MPI_Init_endpoint(argc, argv, MPI_THREAD_SINGLE, &provided);
MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_ENDPOINTS, &pnum, &flag);
#pragma omp parallel private(myid)
{
    #pragma omp master
    {
        /* find number of threads in current team */
        Nthreads = omp_get_num_threads();
        if (Nthreads != *pnum) abort();
        /* create endpoints */
        MPI_Endpoint_create(Nthreads, *endpoints);
    }

    /* associate each thread with an endpoint */
    myid = omp_get_thread_num();
    MPI_Endpoint_register(myid, *endpoints);

    /* MPI communication involving all threads */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_PROCESS, &mythreadid);
    if (myid != mythreadid) abort();
    if (myrank > 0)
        MPI_Isend(buff, count, MPI_INT, myrank-1, 3,
                  MPI_COMM_WORLD, &req[mythreadid]);
    if (myrank < size)
        MPI_Recv(buff1, count, MPI_INT, myrank+1, 3,
                 MPI_COMM_WORLD);
    MPI_Wait(&req[mythreadid], &status[mythreadid]);
}
...

```

## 2.5 Implementation Issues

The support of multiple MPI endpoints at a process should not be different than the support of multiple processes at an SMP node. The only additional overhead is that, whenever a thread



executes an MPI call, it needs to access the data structures that implement the endpoint the thread is currently associated with. This requires an additional level of indirection for each MPI call (to retrieve from the thread private data a pointer to the endpoint the thread is currently associated with) and tests to check that the pointer is valid.

The missing parts of this proposal can be extended by keeping in mind this analogy. Thus, `MPI_FINALIZE()` should be invoked by a thread attached to an endpoint exactly once for each endpoint. Once the invocation occurred, no further MPI calls on this endpoint are allowed.

Of course, there are many possible optimizations. In particular, communication between endpoints in the same address space should require only one memory copy.

## 2.6 Missing items

A complete proposal needs additional items, including:

- `MPI_FINALIZE()` (obvious)
- `mpiexec` (easy)
- Dynamic processes (reasonably easy)
- Formalize relation of endpoint to (communicator, rank) pair (easy)
- One-sided (windows are per process – per address space, not per endpoint – will probably want windows to be associated with processes – not endpoints).
- I/O (file descriptors are per process, not per endpoint – will probably want file manipulation to be per process, not per end-point).
- New error codes
- A discussion of progress – in relation to thread scheduling
- Whatever constraints we need to impose on the use of thread synchronization operations
- ...

## 3 OpenMP Binding

### 3.1 OpenMP Scheduling

We briefly review the scheduling mechanism of OpenMP (references are to the V 3.0 standard [8]):

The execution model of OpenMP is a fork-join model: The program starts in a single thread; a `parallel` construct forks a team of threads that execute in parallel (the team includes the master thread that reached the parallel construct); they join back at the exit from the parallel region. Parallel constructs can be nested. 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 arguments of the parallel construct [8, Section 2.4.1]. Once a team is created, the team's threads do not change. A thread is associated with only one team at a time.

On top of this thread model, OpenMP also has a fiber model. Work-sharing constructs, such as parallel loops, define fibers that can be dynamically allocated to the threads of the team associated with the innermost containing parallel construct. OpenMP has much flexibility in chunking shared work into fibers and scheduling fibers to threads. For example OpenMP can introduce arbitrary scheduling point in untied OpenMP tasks – i.e., have these tasks yield at arbitrary points during their execution; such tasks can resume on any other thread in the team [8, Section 2.7.1]. Therefore, it is safe to assume that code in a work-sharing construct is executed by a thread in the team associated with the innermost containing parallel construct, but unsafe to make any assumptions on the identify of that thread, or make assumptions that pieces of code will execute on the same thread.

### 3.2 Binding

The preceding discussion informs the restrictions listed below:

#### 3.2.1 *MPI\_THREAD\_SINGLE Model*

- OpenMP has to use a fixed number of threads – The Internal Control Variable (ICV) *dyn-var* should be set to *false*, either externally, or using `omp_set_dynamic()`.
- Each thread has to be bound to one endpoint.
- No MPI calls can occur within work sharing constructs.

#### 3.2.2 *MPI\_THREAD\_FUNELLED Model*

- No MPI calls can occur within work sharing constructs

#### 3.2.3 *MPI\_THREAD\_SERIALIZED*

- MPI calls occurring within work-sharing constructs must be within a `critical` or `master` construct.

#### 3.2.4 *MPI\_THREAD\_MULTIPLE*

No restriction (beyond those specified by MPI for threads).

Note that if a blocking MPI call is made then the call will block the calling thread; the thread cannot be used to run other shared work.

## 4 OpenMP Manual of Style

### 4.1 Static Model

A simple MPI\_OpenMP model that is broadly consistent with current MPI programming models is obtained by

- Using a fixed number of threads in OpenMP
- Having a fixed one-to-one mapping of MPI endpoints with OpenMP threads, using the `MPI_THREAD_SINGLE` model.
- Refraining from using work sharing constructs (no parallel loops, sections, workshare or tasks)

We further assume that the OS can bind threads to cores and ensure that the computing threads are not preempted. With these assumptions, then each core will be associated with one thread, and each thread will be associated with one MPI endpoint. From the MPI viewpoint, this provides the same model (and should provide the same performance) as when one attaches an MPI process to each core. However, the threads within one process can communicate using shared memory. This model does not provide automatic load-balancing at the nodes – the programmer manages resources directly.

In some cases it may be appropriate to have a number of endpoints that is smaller than the number of cores. This, as the number of physical communication endpoints can be smaller than the number of threads, and the addition of a larger number of virtual endpoints may harm communication performance and stretch MPI scalability. For example, on cores that support a large number of simultaneous threads, we could have one communication thread and multiple computation threads; or we could have some computation cores and some communication cores. In such a case, we shall obey the same constraints listed above, except that we shall use the `MPI_THREAD_FUNNELLED` mode and have only a subset of the threads associated with endpoints.

### 4.2 Dynamic Model

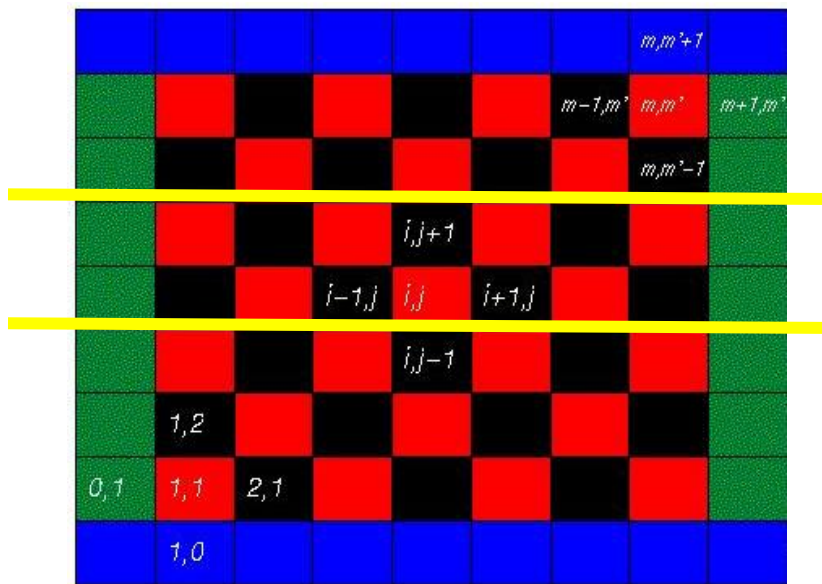
Work sharing constructs do simplify programming, by taking care of load balancing, but can lead to higher overheads and loss of locality. If one desires to use such constructs, then the usual rules about tuning OpenMP codes should be obeyed:

- Make sure that the number of fibers is quite larger than the number of threads (a factor of 2--5).
- Make sure that fibers are quite large (1000's of instructions)
- Avoid the use of constructs that force serializations, such as `critical`, `master` `ordered` and `single`.

### 4.3 Examples

*The examples have not (yet) been run – there are unlikely to be correct.*

We illustrate the design with a schematic red-black parallel SOR code, illustrated in the figure below: 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. We assume that the array is partitioned into horizontal stripes – to simplify the example; a more communication efficient algorithm would partition into subsquares. We show codes that do not use work sharing and perform no load balancing.



#### 4.3.1 Sequential code

```
#define N 10000          /*array size */
float a[N+2][N+2];      /*array*/
enum Color {RED, BLACK};
enum Color color = RED;
int i,j;

int main()
{

    init(a);
    while(!converged())
    {
        for(i = 1; i <= N; i++)
        {
            for(j = 1+(i%2)^color; j <= N; j +=2)
                a[i][j] = (a[i][j]+a[i-1][j]+a[i+1][j]+a[i][j-1]+a[i][j+1])*0.2;
            color = !color;
        }
    }
}
```

#### 4.3.2 OpenMP code with no load balancing

This code uses the threaded OpenMP model, with no work sharing.

```

#include <omp.h>

#define N 10000          /*array size */
float a[N+2][N+2];      /*array*/

enum Color {RED, BLACK};
enum Color color = RED;
int i, j, ibegin, iend, mythreadid, nthreads;

#pragma omp threadprivate(i,j,ibegin,iend,mythreadid,nthreads,color)

void set_thread_stripe(int mythreadid, int numthreads, int *ibegin, int *iend)
/* compute thread stripe boundaries */
{
    *ibegin = 1+mythreadid*N/numthreads;
    *iend = (mythreadid == numthreads-1) ? N : (mythreadid+1)*N/numthreads;
}

int main()
{
    init(a);

    #pragma omp parallel
    {
        nthreads = omp_get_num_threads();
        mythreadid = omp_get_thread_num();
        set_thread_stripe(mythreadid, nthreads, &ibegin, &iend);

        while (!converged())
        {
            for(j = (i%2)^color; j <= N; j +=2)
                a[i][j] = (a[i][j]+a[i-1][j]+a[i+1][j]+a[i][j-1]+a[i][j+1])*0.2;
            color = !color;
            #pragma omp barrier
        }
    }
}

```

### 4.3.3 Single threaded MPI code

```
#include <mpi.h>
#include <stdlib.h>

#define N 10000          /* array size */
int stripe_size;        /* local stripe size */
float a[][N+2];          /* process stripe, with a size one ghost row
                          on each side */

enum Color {RED, BLACK};
enum Color color = RED;
int numprocs, myrank, k;

int boundary;

/* first red cell in first and last row */
char fred, lred;

int lastrow;

MPI_Datatype dtype[2]; /* datatype for row of single color cells */
MPI_Request req[4];
MPI_Status status[4];

void compute(int ibegin, int iend, int firstcell)
{
    /* perform iterations on rows ibeginÉiend */
    int i,j;

    for(i = ibegin; i < iend; i++)
    {
        for(j = firstcell; j <= N; j += 2)
            a[i][j] = (a[i][j]+a[i-1][j]+a[i+1][j]+a[i][j-1]+a[i][j+1])*0.2;
        firstcell = !firstcell;
    }
}

int main()
{
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    boundary = (myrank == 0) || (myrank == numprocs -1);

    /* datatypes for red and for black squares */
    MPI_Type_vector((N+1)/2, 1, 2, MPI_FLOAT, &dtype[0]);
    MPI_Type_vector(N/2, 1, 2, MPI_FLOAT, &dtype[1]);

    /* compute process stripe size, First and last stripes should be slightly
       larger as they communicate less */
```

```

set_process_stripe(myrank, numprocs, &stripe_size);
float a[stripe_size+2][N+2];
init(a);

/* compute location of first cell of red color */
MPI_Scan ( &stripe_size, &lastrow, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
fred = (lastrow-stripe_size)%2;
lred = lastrow%2;

while (!converged())
{
    k=0;
    /* compute 1st row */
    compute(1,1,fred^color);
    if (myrank > 0)
    {
        MPI_Isend(&a[1][fred^color], 1, dtype[fred^color], myrank-1, 0,
                  MPI_COMM_WORLD, &req[k++]);
        MPI_Irecv(&a[0][!fred^color], 1, dtype[!fred^color], myrank-1, 0,
                  MPI_COMM_WORLD, &req[k++]);
    }

    /* compute last row */
    compute(stripe_size,stripe_size,lred^color);
    if (myrank < numprocs-1)
    {
        MPI_Isend(&a[stripe_size][lred^color], 1, dtype[lred^color], myrank+1, 0,
                  MPI_COMM_WORLD, &req[k++]);
        MPI_Irecv(&a[stripe_size+1][!lred^color], 1, dtype[!lred^color], myrank+1, 0,
                  MPI_COMM_WORLD, &req[k]);
    }

    /* compute middle rows */
    compute(2, stripe_size-1, !fred^color);

    MPI_Waitall(4-2*boundary, req, status);
    color = !color;
}
}

```

#### 4.3.4 OpenMP code with no load balancing and no barrier

A thread proceeds to compute the red (resp. black) iteration on its stripe if the one or two neighbor threads finished the black (resp. red) iteration. The avoidance of global synchronization makes code more resilient to jitter. This code has the same logic as the MPI code above: Point-to-point communications are replaced by thread-to-thread synchronizations.



```

#include <stdlib.h>
#include <omp.h>

#define N 10000          /*array size */
int numthreads;          /* number of threads used */
int stripe_size;
float a[N+2][N+2];       /*array*/

extern int done[][2];     /* used to count number of ready predecessors. Alternative
                           sets are used to avoid races. */

enum Color {RED, BLACK};
enum Color color = RED;
int ibegin,iend,mythreadid;

#pragma omp threadprivate(ibegin, iend, mythreadid)

void mysignal(int id, enum Color color)
{
    /* signals a predecessor is ready */
    if ((id >= 0) && (id < numthreads))
    {
        #pragma omp atomic
        done[id][color]++;
    }
}

void mywait(int id, enum Color color)
{
    /* busy waits until dependencies are satisfied */

    int boundary = ((id == 0) || (id == numthreads-1));

    while (done[id][color] + boundary < 2)
    {
        #pragma omp flush (done[id][color])
    }
}

void compute(int ibegin, int iend, enum Color color)
{
    /* perform iterations on rows ibegin...iend */
    int i,j;

    for (i = ibegin; i < iend; i++)
        for (j = 1+(i%2)^color; j <= N; j += 2)
            a[i][j] = (a[i][j]+a[i-1][j]+a[i+1][j]
                        +a[i][j-1]+a[i][j+1])*0.2;
}

```

```

void set_thread_stripe(int mythreadid, int numthreads, int *ibegin, int *iend)
/* compute thread stripe boundaries */
{
    *ibegin = 1+mythreadid*N/numthreads;
    *iend = (mythreadid == numthreads-1) ? N : (mythreadid+1)*N/numthreads;
}

int main()
{
    int i, j;

    init(a);

    #pragma omp parallel
    {
        /* initialization */
        mythreadid = omp_get_thread_num();
        numthreads = omp_get_num_threads();
        set_thread_stripe(mythreadid, numthreads, &ibegin, &iend);
    }

    int done[numthreads][2];
    for (i=0; i<numthreads; i++)
        for (j=0; j<2; j++)
            done[i][j] = 0;

    #pragma omp parallel
    {
        while (!converged()) {

            /* compute first row */
            compute(ibegin,ibegin,color);
            mysignal(color, mythreadid-1);

            /* compute last row */
            compute(iend,iend,color);
            mysignal(mythreadid+1, color);

            /* compute middle */
            compute(ibegin+1, iend-1, color);

            mywait(mythreadid,color);
            done[mythreadid][color] = 0;
            color = !color;
        }
    }
}

```

Note that the OpenMP code is similar in size to the MPI code: The simple barrier synchronization has been replaced with detailed point-to-point synchronization.

#### 4.3.5 Hybrid OpenMP+MPI code with no load balancing

We combine the logic of the two previous codes: Some communications use shared memory, while others use message passing.

```
#include <stdlib.h>
#include <mpi.h>
#include <omp.h>

typedef int MPI_Endpoint;

#define N 10000          /* array size */

float a[][N+2]; /* process array stripe */

enum Direction {UP, DOWN};
enum Color {RED, BLACK};

int stripe_size; /* node local stripe size. Should be larger for first and last node
*/
int myrank, numthreads, mythreadid, ibegin, iend, numports, numprocs, myprocid, *pnum,
flag, lastrow, firstrowparity;
enum Color color = RED;
int done[][2];
int fred, lred;

MPI_Datatype dtype[2];
MPI_Request req[2][2];
MPI_Status status[2];

MPI_Endpoint endpoints[2];
int provided;

#pragma omp threadprivate(ibegin, iend, fred, lred, numthreads, mythreadid, req, status)
;

void compute(int ibegin, int iend, int firstcell)
{
/* perform iterations on rows ibegin...iend */

int i,j;

for(i = ibegin; i < iend; i++) {
for(j = firstcell; j <= N; j += 2)
a[i][j] = (a[i][j]+a[i-1][j]+a[i+1][j]
+a[i][j-1]+a[i][j+1])*0.2;
firstcell = !firstcell;
}
}

void mysignal(int firstcell, enum Direction dir, enum Color color)
{
if (dir == DOWN)
```

```

{
    if (mythreadid == 0)
    {
        if (myrank > 0) {
            /* communicate down via message passing */
            MPI_Isend(&a[1][firstcell], 1, dtype[firstcell], myrank-1, 0, MPI_COMM_WORLD, &req
[color][0]);
            MPI_Irecv(&a[0][!firstcell], 1, dtype[!firstcell], myrank-1, 0, MPI_COMM_WORLD, &req
[color][1]);
        }
    }

    else /* mythreadid > 0 */
    {
        /* shared memory communication */
        #pragma atomic
        done[mythreadid-1][color]++;
    }
}
else /* dir == UP */
if (mythreadid == numthreads-1)
{
    if (myrank < numports)
    {
        /* communicate up via message passing */
        MPI_Isend(&a[stripe_size][firstcell], 1, dtype[firstcell], myrank+1, 0,
MPI_COMM_WORLD, &req[color][0]);
        MPI_Irecv(&a[0][!firstcell], 1, dtype[!firstcell], myrank-1, 0, MPI_COMM_WORLD, &req
[color][1]);
    }
}
else /* mythreadid < numthreads-1 */
{
    /* shared memory communication */
    #pragma atomic
    done[color][mythreadid+1]++;
}
}

void mywait(enum Color color)
{
    if (((mythreadid == 0) && (myrank == 0)) || ((mythreadid == numthreads-1) && (myrank =
numports-1)))
        /* boundary stripe */
        done[color][mythreadid]++;
    if (((mythreadid == 0) && (myrank > 0)) || ((mythreadid == numthreads-1) && (myrank <
numports-1)))
    {
        /* need to complete send & receive */

```

```

        MPI_Waitall(2, req[color], status);
        done[color][mythreadid]++;
    }
    while (done[color][mythreadid] < 2)
    {
        #pragma omp flush (done[color][mythreadid])
    }
}

int main(int argc, char **argv)
{
    /* process initialization */
    MPI_Init_endpoint(argc, argv, MPI_THREAD_SINGLE, &provided);
    MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_ENDPOINTS, &pnum, &flag);
    if (*pnum < 2) abort();
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myprocid);

    /* compute size of process stride; first and last processes should
       have larger stripes */
    set_process_stripe(N, numprocs, &stripe_size);
    float a[stripe_size+2][N+2];
    init(a);

    /* create one or two endpoints at each process */
    if ((myprocid == 0) || (myprocid == numprocs-1))
        MPI_Endpoint_create(1, endpoints);
    else
        MPI_Endpoint_create(2, endpoints);

    /* datatypes for red and for black squares */
    MPI_Type_vector((N+1)/2, 1, 2, MPI_FLOAT, &dtype[0]);
    MPI_Type_vector(N/2, 1, 2, MPI_FLOAT, &dtype[1]);

    /* compute parity of each process stripe */
    MPI_Scan (&stripe_size, &lastrow, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    firstrowparity = (lastrow-stripe_size)%2;

    /* start thread parallel computation */
    #pragma omp parallel
    /* thread initialization */
    {
        numthreads = omp_get_threads_num();
        mythreadid = omp_get_num_thread();
    }

    int done[numthreads][color];

```

```

#pragma omp parallel
{
    /* register endpoints with first or last thread */
    if ((mythreadid == 0) && (myprocid > 0))
        MPI_Thread_register(0, endpoints);
    else if (mythreadid == numthreads-1)
    {
        if (myprocid == 0)
            MPI_Thread_register(0, endpoints);
        else if (myprocid < numprocs-1)
            MPI_Thread_register(1, endpoints);
    }

    /* associate thread stripe with each thread; message passing threads have smaller
    stripes */
    set_thread_stripe(&ibegin, &iend);

    /* compute location of first cell of red color */
    fred = (firstrowparity+ibegin)%2;
    lred = (firstrowparity+iend)%2;

    while (1)
    {

        /* compute first row */
        compute(ibegin, ibegin, fred^color);
        mysignal(fred^color, UP, color);

        /* compute last row */
        compute(iend, iend, lred^color);
        mysignal(lred^color, DOWN, color);

        /* compute middle */
        compute(ibegin+1, iend-1, !fred^color);
        mywait(color);
        color = !color;
    }
}

```

The last code is much longer than the sequential code and twice as long as the MPI code. It would be nice to have such a code automatically or semi-automatically generated by a compiler. A good compiler for UPC or CAF should be able to do so.

## 5 Extensions

The proposal outlined in the paper extends the MPI model from a “process model” to a “thread model”. OpenMP is a hybrid model, with support both for threads and for fibers; the current proposal essentially matches MPI to the OpenMP thread support.

Increasingly, shared memory parallel programming languages and framework use a “fiber model”; this is true of TBB [9], .NET Task Parallel Library [6], Java fork-join framework [5], Cilk [1], etc. future parallel shared memory languages are likely to hide threads from the user and provide only a view of fibers – with no control on the number of threads or the scheduling of fibers to threads. This is because one does not need the protection and fairness provided by the OS scheduler – at the cost of some overhead; and the fiber scheduler can implement scheduling policies that are more appropriate to tightly coupled, cooperating fibers.

A more elegant design would be to fully integrate MPI with a fiber model. By this, we mean that when a fiber executes a blocking MPI call, then the fiber yields and is descheduled, but the thread that was executing the fiber is not descheduled, and can pick another fiber for execution. This would require coordination between the MPI library and the language runtime:

- When a fiber executes a blocking MPI call, then the MPI library will call the fiber scheduler to indicate that the fiber yielded.
- When the call completes, the MPI library will mark the fiber as runnable.
- Progress can be ensured by the fiber scheduler by periodically scheduling an MPI progress fiber.

Such a scheme would integrate MPI with any of the languages above mentioned, and could lead to a lighter MPI infrastructure.

It is likely that such a scheme could be of more general use – to support “blocking user calls” for a variety of purposes. In particular, such interface could be used to interface parallel codes that use distinct fiber run-times (and possibly run on distinct sets of threads). The basic functions required are:

- A callback to mark a fiber as blocked and associate it with an event.
- A call to mark an event as complete.
- A mechanism to split resources (statically or dynamically) among distinct subsystems.

Such a design would require changes both on the MPI side and on the language runtime side, hence is beyond the scope of the MPI 3 forum – but should be considered as a research direction.

## References

- [1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall and Y. Zhou, *Cilk: An efficient multithreaded runtime system*, ACM SigPlan Notices, 30 (1995), pp. 207-216.
- [2] R. Brightwell, *MPI Runtime and Affinity Enhancements for Multi-Core Processors*,
- [3] F. Cappello and D. Etiemble, *MPI versus MPI+ OpenMP on the IBM SP for the NAS Benchmarks*, 2000, pp. 12-12,
- [4] B. Chapman, G. Jost and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*, The MIT Press, 2007.
- [5] D. Lea, *A Java fork/join framework*, ACM New York, NY, USA, 2000, pp. 36-43,
- [6] D. Leijen and J. Hall, *Optimize Managed Code For Multi-Core Machines*, 2007, <http://msdn.microsoft.com/en-us/magazine/cc163340.aspx>
- [7] MPI Forum, *MPI: A Message-Passing Interface Standard V2.1*, 2008, <http://www.mpi-forum.org/docs/mpi21-report.pdf>
- [8] OpenMP Architecture Review Board, *OpenMP Application Program Interface Version 3.0*, 2008, <http://www.openmp.org/mp-documents/spec30.pdf>
- [9] J. Reinders, *Intel Threading Building Blocks Outfitting C++ for Multi-core Processor Parallelism*, O'Reilly, 2007.
- [10] A. Supalov, *Treating threads as MPI processes thru registration/deregistration*.
- [11] R. Thakur and W. Gropp, *Test suite for evaluating performance of MPI implementations that support MPI\_THREAD\_MULTIPLE*, Springer, 2007, pp. 46,
- [12] The Open Group, *The Single Unix Specification Version 3*, [http://www.unix.org/single\\_unix\\_specification/](http://www.unix.org/single_unix_specification/)