Enabling MPI Interoperability Through Flexible Communication Endpoints

James Dinan,* Pavan Balaji,* David Goodell,* Douglas Miller,†
Marc Snir,* and Rajeev Thakur*

*Mathematics and Computer Science Division
Argonne National Laboratory
{dinan,balaji,goodell,snir,thakur}@mcs.anl.gov

†International Business Machines Corp.
dougmill@us.ibm.com

ABSTRACT

The current MPI model defines a one-to-one relationship between MPI processes and MPI ranks. This model captures many use cases effectively, such as one MPI process per core and one MPI process per node. However, this semantic has limited interoperability between MPI and other programming models that use threads within a node. In this paper, we describe an extension to MPI that introduces communication endpoints as a means to relax the one-to-one relationship between processes and threads. Endpoints enable a greater degree interoperability between MPI and other programming models, and we illustrate their potential for additional performance and computation management benefits through the decoupling of ranks from processes.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*

General Terms

Design, Performance

Keywords

MPI, Interoperability, Endpoints, Hybrid Parallel Programming

1. INTRODUCTION

Hybrid parallel programming in the "MPI+X" model has become the norm in high-performance computing. This approach to parallel programming mirrors the hierarchy of parallelism in current high-performance systems, in which a high-speed interconnect joins many highly parallel nodes. While MPI is effective at managing internode parallelism, alternative data-parallel, fork-join, and offload models are needed to utilize current and future highly parallel nodes effectively.

EuroMPI '13, Sept. 15–18, 2013, Madrid, Spain.

Interoperability between MPI and other parallel programming systems has long been a productivity and composability goal within the parallel programming community. The widespread adoption of "MPI+X" parallel programming has put additional pressure on the community to produce a solution that enables full interoperability between MPI and system-level programming models, such as X10, Chapel, Charm++, UPC, and Coarray Fortran, as well as node-level programming models such as OpenMP, threads, and TBB. A key challenge to interoperability is the ability to generate additional MPI ranks that can be assigned to threads used in the execution of such models.

The MPI 3.0 standard resolved some issues affecting interoperability of threads with MPI, but it did not provide any mechanism that allows additional MPI ranks to be generated and assigned to threads. In the current MPI interface, the programmer must use either tags or communicators to distinguish communication operations between individual threads. However, both approaches have significant limitations. When tags are used, it is not possible for multiple threads sharing the rank of an MPI process to participate in collectives. In addition, when multiple threads perform wildcard receive operations, matching is nondeterministic. Using multiple communicators can sidestep some of these restrictions, but at the expense of partitioning threads into individual communicators where only one thread per parent process can be present in each new communicator.

In this paper, we present an MPI extension, called MPI endpoints, that enables the programmer to generate additional ranks at each process. We explore the design space of MPI endpoints and how endpoints can impact the MPI implementation and application performance. In addition, we demonstrate the utility of endpoints in addressing the interoperability of MPI with system-level and node-level parallel programming systems. Notably, endpoint ranks can be distributed to threads in system-level programming models, such as X10 or UPC, enabling these threads to act as MPI processes and interoperate with MPI directly and with MPI libraries. Endpoints can also be distributed among threads in node-level parallel programming systems, enabling these threads to participate fully in internode MPI operations, as well as to perform intranode MPI operations. Finally, we demonstrate that breaking the one-to-one relationship between ranks and processes opens the possibility for a variety of computation-management strategies that balance the workload through the reassignment of processes to different nodes.

2. BACKGROUND

Led by the authors, members of the MPI Forum first began discussion of an MPI endpoints interface as a part of the MPI 3.0 effort. Several interfaces were explored, resulting in the static in-

terface that we present in Section 3.1. We proposed the dynamic interface, presented in Section 3.2, as a more flexible alternative to the static interface, and it is under consideration for inclusion in a future version of the standard.

MPI interoperability has been investigated extensively in the context of a variety of parallel programming models. Interoperability between MPI and Unified Parallel C was defined in terms of one-to-one and many-to-one mappings of UPC threads to MPI ranks [3]. Support for the one-to-one mapping cannot be provided in MPI 3.0 when the UPC implementation utilizes operating system threads, rather than processes, to implement UPC threads. However, this mode of operation can be supported using endpoints.

Hybrid parallel programming with MPI and a node-level parallel programming model has become commonplace. MPI is often combined with multithreaded parallel programming models, such as MPI+OpenMP [16]. MPI 2.0 [13] defined MPI's interaction with threads in terms of several levels of threading support that can be provided by the MPI library. MPI 3.0 further refined the definition of MPI's interaction with threads, including new features such as matched-probe operations to enable the use of MPI_Probe when multiple threads share an MPI rank. In addition, MPI 3.0 added support for interprocess shared memory through the Remote Memory Access (RMA) interface [7, 8].

Recently, researchers have endeavored to integrate MPI and accelerator programming models. This effort has focused on supporting the use of CUDA or OpenCL buffers directly in MPI operations [9, 19]. Other efforts have focused on enabling accelerator cores to perform MPI calls directly [17].

Numerous efforts have been made to integrate node-level parallelism with MPI. FG-MPI (Fine-Grain MPI) [10] implements MPI processes as lightweight coroutines instead of operating-system processes, enabling each coroutine to have its own MPI rank. Programs have been run with as many as 100 million MPI ranks using FG-MPI [15]. Li et al. recently demonstrated significant performance improvements in collective communication when the node is not partitioned into individual MPI processes [12], i.e., endpoints are used instead of one process per core. HMPI transmits ownership for shared buffers to improve the performance of intranode communication [5].

3. COMMUNICATION ENDPOINTS

We define an MPI endpoint as 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. One or more threads can be attached to an endpoint, after which they can make MPI calls using the resources of the endpoint. In this context, an MPI process consists of an MPI endpoint and a set of threads that can perform MPI calls using that endpoint.

The current MPI standard has a one-to-one mapping between endpoints and threads. Communication endpoints breaks this restriction as shown in Figure 1. A variety of mechanisms could be used to incorporate endpoints into MPI. We break down the design space into static versus dynamic approaches and further discuss how and when additional ranks are generated, how ranks are associated with endpoints, and how threads are mapped to ranks.

3.1 Static Interface

A static endpoints interface allows the programmer to request additional endpoints at each MPI process once during the execution of their program, typically during launching or initialization of an MPI execution. In one approach to supporting static endpoints,

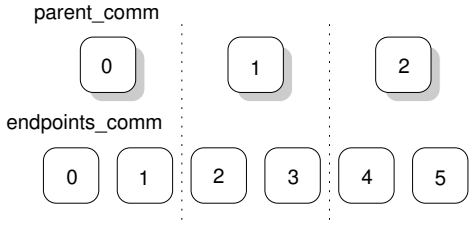

Figure 1: Flexible communication endpoints extend MPI with a many-to-one mapping between ranks and processes

the programmer requests additional endpoints as an argument to mpiexec, as in [3]. This approach requires MPI_Init to be called multiple times at each process and results in an MPI_COMM_WORLD that incorporates all endpoints. As an alternative to this approach, a new MPI initialization routine can be added that allows the programmer to indicate the desired number of endpoints.

```
int MPI_Init_endpoints(int *argc, char *argv[], int count,
        int tl_requested, int *tl_provided)
```

where count indicates the desired number of endpoints at this process, tl_requested is the level of thread support requested by the user, and tl_provided is an output parameter indicating the level of thread support provided by the MPI library.

This approach preserves an MPI_COMM_WORLD communicator that contains only the number of processes that the user requested when their program was launched. An additional communicator, MPI_COMM_ENDPOINTS contains all processes in MPI_COMM_WORLD, as well as additional endpoints at each process that were requested in the call to MPI_Init_endpoints.

In order to use an endpoint, a thread must first attach to a rank in the endpoints communicator. Identifying the specific endpoint can be accomplished in a variety of ways. One approach is to introduce an explicit MPI_Endpoint object to represent each endpoint; an array of such objects would be returned by MPI_Init_endpoints or they could be returned individually by a query routine. We present below a simpler approach where the endpoint index, between 0 and count-1, is used.

```
int MPI_Comm_endpoint_attach(MPI_Comm comm, int index)
```

An advantage of the static endpoints scheme is that it can better align MPI implementations with systems where endpoint resources are reserved or created when MPI is initialized, for example, in the case of implementations based on IBM's PAMI low-level communication library [11]. However, statically specifying the number of endpoints restricts the ability of MPI to interoperate fully with dynamic threading models and limits opportunities for libraries to use endpoints. To better support such use cases, we describe an alternative approach that allows endpoints to be created dynamically.

3.2 Dynamic Interface

MPI communicators provide a unique communication context that is used to perform matching between operations and to distinguish communication operations that arise from different components in a program or library. In addition to providing encapsulation, communicators contain a group that defines the processes that are members of the communicator and the mapping of ranks to these processes. When a communicator is created, the group of the new communicator is derived from the group of an existing *parent* communicator. In conventional communicator creation routines, the group of the new communicator is a subset of the group of

its parent. (We ignore here the dynamic processes interface, which can be used to create new processes but not to generate additional ranks for the purpose of interoperability with threaded programming models.)

The dynamic endpoints interface uses the grouping property of communicators to generate multiple ranks that are associated with each process in the parent communicator. An example of this interface is as follows.

```
int MPI_Comm_create_endpoints(MPI_Comm parent_comm, int
      my_num_ep, MPI_Info info, MPI_Comm out_comm_hdls[])
```

In this collective call, a single output communicator is created, and an array of my_num_ep handles to this new communicator are returned, where the i^{th} handle corresponds to the i^{th} rank requested by the caller of MPI_Comm_create_endpoints. Ranks in the output communicator are ordered sequentially and in the same order as the parent communicator.

After it has been created, the output communicator behaves as a normal communicator, and MPI calls on each endpoint (i.e., communicator handle) behave as though they originated from a separate MPI process. In particular, collective calls (including MPI_Comm_free) must be made once per endpoint.

An additional parameter of the endpoints design is the association of threads with endpoints. From the point of view of the endpoints interface, the parent MPI process is considered to be another thread. A simple and flexible strategy is to allow any thread to communicate on any of its endpoints as soon as they have been created. Alternatively, threads can be associated with specific endpoints by attaching to them, as follows.

```
int MPI_Comm_attach(MPI_Comm ep_comm)
```

An advantage of this approach is that it allows the MPI implementation to associate threads with specific endpoints, potentially enabling the MPI implementation to manage resources more effectively. Functionality to detach from an endpoint can also be added, enabling a more dynamic programming interface but potentially limiting optimizations.

Freeing an endpoints communicator requires a collective call to MPI_Comm_free, where the free function is called once per endpoint. As currently defined, this could require MPI_Comm_free to be called concurrently on all endpoints, which in turn requires that the MPI implementation be initialized with MPI_THREAD_MULTIPLE support. To avoid this restriction, we propose that MPI_Comm_free be defined in a way that enables a single thread to call MPI_Comm_free separately for each endpoint. Naive implementations of this semantic could lead to deadlock when the thread level is not MPI_THREAD_MULTIPLE. A more sophisticated implementation may be needed, where the MPI implementation internally aggregates endpoint free requests to parent MPI processes and performs any needed communication among parent processes when the last endpoint rooted at that process calls MPI_Comm_free.

3.3 Progress Semantics

The MPI standard specifies a minimal progress requirement: Communication operations whose completion requirements are met (e.g., matching send and receive calls have been made, test or wait has been called) must eventually complete regardless of other actions in the system. Endpoints add an additional dimension to the progress semantic, as one could define a semantic where they are treated as independent processes or as part of a single parent process.

The single parent process model defines the simplest semantic, and it would require that the MPI implementation ensure progress is made on all endpoints of a process. This semantic is easy for users to reason about and is compatible with an endpoints interface that does not require threads to attach to endpoints. However, this approach may limit concurrency in some MPI implementations by requiring any thread entering the MPI progress engine to access the state for all endpoints.

Treating endpoints as individual processes in the progress semantic is a weaker guarantee that is more consistent with the MPI standard and introduces additional opportunities for concurrency within the MPI library. In this model, it can be helpful to the MPI implementation if users explicitly associate threads with endpoints through attach operations. This feature can be used to limit the number of endpoints with which a thread is associated. When such a restriction is not made, a thread can be associated with multiple endpoints by issuing communication operations on each endpoint.

When the progress semantics treat endpoints as individual processes, the MPI implementation can partition progress across endpoints. For example, a thread that calls a blocking operation must make progress on its associated endpoints and on any shared endpoints with which it is associated (e.g., its rank in MPI_COMM_WORLD). This approach could enable the MPI library to allow multiple threads to drive communication on different endpoints concurrently. As an additional optimization, if the user knows that each endpoint will be used by only one thread, an MPI_info hint could be provided that would enable the MPI implementation to avoid locking when accessing that endpoint's state.

4. IMPACT ON MPI IMPLEMENTATIONS

In this section we discuss the impact of endpoints on existing MPI implementations. We focus on the MPICH implementation of MPI [14], but the discussion should be generally applicable to other MPI implementations.

4.1 Internal Communicator Representation

In MPICH, communicators are internally represented as a structure at each process that contains several important components: the process's rank in the communicator, the size of the communicator, and mapping from rank to *virtual connection* (VC). This mapping structure is currently a dense array indexed by communicator rank, though other implementation choices are possible [6]. Each VC represents a *logical* connection to another MPI process, even though the underlying network may not have a concept of connections. VCs must be unique within a communicator, since each process may hold only a single rank in any given communicator. The addition of endpoints requires us to relax the uniqueness restriction, breaking an otherwise injective relationship. While rank-to-VC translation is important, the reverse lookup is not performed; thus the loss of uniqueness does not present any new challenges.

The communicator structure is referred to in MPI applications by an integer handle value, which provides the semantics required by the MPI standard and simplifies Fortran binding issues. Thus, a handle value can most easily be thought of as an obfuscated pointer to the corresponding underlying communicator structure. Calling MPI_Comm_create_endpoints will create one communicator handle for each endpoint on a given process, and the handles are returned in the out_comm_hdls array. The handles may share one or more underlying structures. For example, just as normal communicators that are duplicated by MPI_Comm_dup may easily share a mapping data structure, endpoint communicators within a process may also easily share the mapping structure within an MPI process among all the endpoint communicators derived from the same call to MPI_Comm_create_endpoints.

MPI communicators are logically composed of an MPI group

and a communication context. MPI groups are opaque data structures representing ordered sets of MPI processes that are local to an MPI process. Groups are immutable once created and are derived from existing communicators or other groups. In order to extend MPI groups to endpoint communicators, groups must be generalized to incorporate endpoints as well as conventional MPI processes. This problem is traditionally viewed as a mapping problem [18] from the dense group space to a *global process ID* of some type (though the same process need not be represented by the same value on two different processes, leading to the LPID concept of previous work [6]). Endpoints must be assigned unique IDs at their creation time to serve as an element in the codomain of the group rank mapping function. This ID must not be recycled/reused as long as any communicator or group contains a reference to the endpoint associated with that ID. In this sense, endpoints can be treated in a fashion similar to MPI-2 dynamic processes.

4.2 Associating Endpoints with Connections

Prior to the introduction of endpoints, communication from multiple communicators could potentially multiplex over the set of VCs in an MPI process. MPI message matching in the receive queues (posted and unexpected) is performed using $\langle ContextID, CommRank, Tag \rangle$ triples and is already independent of the VC structures. Therefore the introduction of endpoints should not induce an additional multiplexing/demultiplexing step at this layer.

The dynamic endpoints interface introduces complications for interconnects that must create network endpoints when MPI_Init is called. Such networks may be unable to create create additional endpoints later, such as an implementation of MPI over PAMI [11]. In this case, the implementation may create all endpoints when MPI_Init is called, possibly directed by runtime parameters or environment variables. The implementation would need to multiplex MPI endpoints over network endpoints, and the user can improve performance through hints indicating that the number of endpoints created in calls to MPI_Comm_create_endpoints will not exceed the number of network endpoints.

4.3 Interaction with Threading Models

An endpoints interface that returns separate communicator handles for each endpoint has implementation advantages relative to interfaces that share a communicator handle for all endpoints within a single MPI process. Models that share communicator handles maintain an implicit association between threads and endpoints, which has two major implications for the MPI implementation's interaction with the application's threading model.

Any explicit binding between threads and specific endpoints necessitates the usage of *thread-local storage* (TLS) in order to determine the thread's rank in the communicator as well as other endpoint-specific information. Using TLS within the MPI library obligates the MPI implementers to be aware of all possible threading models and middleware that a user might use. This coupling could limit an MPI library's interoperability with different threading models.

A larger problem than tightly coupling the threading model and the MPI library is that such TLS lookups would be on the critical path for any communication operation. Though some threading models and architectures have support for fast TLS access [4], many do not. A design utilizing separate communicator handles per endpoint is advantageous because it keeps TLS off the communication critical path. The design trade-off to using separate handles in the endpoints interface is that it slightly burdens the user, who must distribute these handles among the application threads in order to call MPI routines.

4.4 Optimizing Network Performance

Modern interconnection networks are often capable of greater speeds than can be generated by transmitting a single message. In order to saturate the network, multiple streams of communication must be performed in parallel, and these communications must use distinct resources in order to avoid being serialized.

Endpoints can be used to separate the communication resources used by threads, enabling multiple threads to drive communication concurrently and achieve higher network efficiency. For example, endpoint ranks may be implemented through distinct network resources, such as NICs, DMA FIFOs, or PAMI Contexts [11]. An MPI implementation might have an optimal number of endpoints based on the number of available hardware resources; users could query this optimal number and use it to influence the number of endpoints that are created.

In this regard, endpoints differ from the use of multiple communicators because additional communicators replicate the current rank into a new context, typically using the same hardware resource. Thus, it is challenging to drive multiple communication channels/resources using multiple threads within a single MPI process. The endpoints creation operation generates new ranks that can be associated with separate hardware resources and with the specific intent of enabling parallel, multithreaded communications within a single context.

5. IMPACT ON MPI APPLICATIONS

Endpoints introduce a new capability within the MPI standard in that they allow the programmer a greater degree of freedom in the mapping of ranks to processes. This capability can have broad impact on interoperability as well as mapping of the computation. In this section, we demonstrate these capabilities by using the dynamic endpoints interface. First, we use an OpenMP example to highlight the impact of endpoints on interoperability with node-level parallel programming models. Next, we demonstrate the impact of endpoints on interoperability with system-level parallel programming models through a UPC example. Although we use OpenMP and UPC as examples, the techniques shown are applicable to a variety of parallel programming models that may use threads within a node, including Charm++, Co-array Fortran (CAF), X10, and Chapel. Finally, we demonstrate that the relaxation of process-rank mapping enables new approaches to computation management through a dynamic load balancing example.

5.1 Node-Level Hybrid Programs

In Listing 1, we show an example hybrid MPI+OpenMP program where endpoints have been used to enable all OpenMP threads to participate in MPI calls. In particular, all threads at all nodes are able to participate in a call to MPI_Allreduce. This example highlights one possible use of endpoints. Multiple use cases are possible, including an alternative scheme where MPI_COMM_SELF is used as the basis for the endpoints communicator, allowing the programmer to construct a multilevel parallel structure where MPI can be used within the node and between nodes.

In Listing 1, it is assumed that the maximum number of threads allowed in the OpenMP implementation is compatible with the number of endpoints allowed in the MPI implementation. This is often the case, as the number of cores on a node typically drives the number of threads allowed as well as the available network resources. This example uses exactly one thread per endpoint. Each thread joins an MPI_Allreduce with a different communicator as

```
int main(int argc, char **argv) {
    int world_rank, tl;
    int max_threads = omp_get_max_threads();
    MPI_Comm ep_comm[max_threads];

    MPI_Init_thread(&argc, &argv, MULTIPLE, &tl);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

#pragma omp parallel
    {
        int nt = omp_get_num_threads();
        int tn = omp_get_thread_num();
        int ep_rank;
#pragma omp master
        {
            MPI_Comm_create_endpoints(MPI_COMM_WORLD,
                nt, MPI_INFO_NULL, ep_comm);
        }
#pragma omp barrier
        MPI_Comm_attach(ep_comm[tn]);
        MPI_Comm_rank(ep_comm[tn], &ep_rank);
        ... // divide up work based on 'ep_rank'
        MPI_Allreduce(..., ep_comm[tn]);

        MPI_Comm_free(&ep_comm[tn]);
    }
    MPI_Finalize();
}
```

Listing 1: Example hybrid MPI+OpenMP program where endpoints are used to enable all OpenMP threads to participate in a collective MPI allreduce.

if they were separate MPI ranks in separate processes. Since the threads share a process and address space, the MPI implementation can optimize the combining of data between those threads. In addition, the MPI implementation can utilize the threads and network resources in the endpoints to maximize network throughput.

5.2 System-Wide Hybrid Programs

Endpoints can be used to enable interoperability with system-level programming models that use threads for on-node execution, e.g., UPC, CAF, Charm++, X10, and Chapel. In this setting, endpoints are used to enable flexible mappings between MPI ranks and execution units in the other model.

To illustrate this capability, we show a hybrid MPI+UPC program in Listing 2. This program uses a flat (one-to-one) mapping between MPI ranks and UPC threads [3]. The UPC specification allows UPC threads to be implemented using processes or threads; however, implementations commonly use threads as the execution unit for UPC threads. In order to support the flat execution model, a mechanism is needed to acquire multiple ranks per unit of MPI execution. In [3], the authors extended the MPI launcher with a `-ranks-per-proc` argument that would allow each spawned process to call MPI_Init multiple times, once per UPC thread. This is one approach to enabling a static endpoints model. However, it results in all endpoints being within MPI_COMM_WORLD, which may not be desired.

In order to support the UPC code in Listing 2, the UPC compiler must intercept usages of MPI_COMM_WORLD and substitute the upc_comm_world communicator. Alternatively, the MPI profiling interface (PMPI) can be used to intercept MPI calls and provide communicator translation. This approach provides the best compatibility with MPI libraries that are not compiled by the UPC compiler.

In Listing 3, we show the code that a UPC compiler could generate to enable this hybrid execution model. In this example, MPI is used to bootstrap the UPC execution, which is the approach used by several popular UPC implementations [1]. Once the execution

```
shared [*] double data[100*THREADS];

int main(int argc, char **argv) {
    int rank, i; double err;

    do {
        upc_forall(i = 0; i < 100*THREADS; i++; i) {
            data[i] = ...; err += ...;
        }
        MPI_Allreduce(&err, ..., MPI_COMM_WORLD);
    } while (err > TOL);
}
```

Listing 2: Example hybrid MPI+UPC user code

```
int main(int argc, char **argv) {
    int world_rank, tl, i;
    MPI_Comm upc_comm_world[NUM_THREADS];

    MPI_Init_thread(&argc, &argv, MULTIPLE, &tl);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Comm_create_endpoints(MPI_COMM_WORLD,
        THREADS_PER_NODE, MPI_INFO_NULL, upc_comm_world);

    /* Calls upc_thread_init(), which calls upc_main() */
    for (i = 0; i < NUM_THREADS; i++)
        UPCR_Spawn(upc_thread_init, upc_comm_world[i]);

    MPI_Finalize();
}

upc_thread_init(int argc, char **argv,
                MPI_Comm upc_comm_world) {
    MPI_Comm_attach(upc_comm_world);
    upc_main(argc, argv); /* User's main function */
    MPI_Comm_free(&upc_comm_world);
}
```

Listing 3: Example hybrid MPI+UPC bootstrapping code generated by the UPC compiler

has been bootstrapped, a "flat" endpoints communicator is created, UPC threads are spawned, threads attach to their endpoints, register the endpoints communicator with an interoperability library, and finally run the user's main function (shown in Listing 2).

5.3 Impact on Computation Management

Endpoints introduce powerful, new flexibility in the mapping of ranks to processes. In the current MPI specification, ranks can be shuffled, but the number of ranks assigned to each process must remain fixed. Dynamic endpoints allow ranks to be shuffled and also the number of ranks assigned to each process to be adjusted. This capability can be used to perform dynamic load balancing by treating endpoints as "virtual processes" and repartitioning endpoints across nodes. This enables an application behavior that is similar to Adaptive MPI [2], where MPI processes are implemented as Charm++ objects that can be migrated to perform load balancing.

A schematic example of this approach to load balancing is shown in Figure 2. In this example, individual components of the computation are associated with each endpoint rather than particular threads of execution. This enables a programming convention where per-iteration data exchange can be performed with respect to neighbor ranks in the endpoints communicator (e.g., halo exchange). Thus, when endpoints are migrated, the virtual communication pattern between endpoints is preserved. Such a communication-preserving approach to dynamic load balancing can provide an effective solution for adaptive mesh computations.

While this model for load balancing is powerful and useful, it requires the programmer to manually communicate computational state from the previous thread or process responsible for an end-

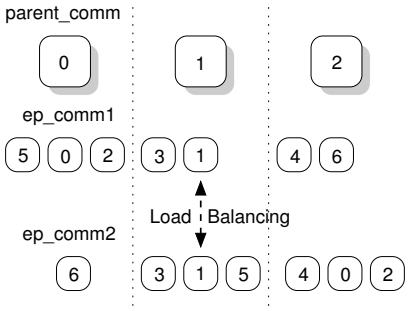

Figure 2: Dynamic load balancing via endpoints migration

point to the endpoint's new owner. This introduces an interesting opportunity for parallel programming researchers to develop library solutions that can address this issue. For example, a library could interface with existing load balancing or work partitioning tools to generate the remapped communicator. A library interface could then be used to migrate endpoints state from locations in the old communicator to locations in the new communicator.

6. SUMMARY

Endpoints provide a natural evolution of MPI that relaxes the one-to-one mapping of ranks to processes. This change can enable greater interoperability between MPI and parallel programming models that use threads or other flexible execution strategies. In addition, the decoupling of ranks from processes provides additional flexibility in mapping the computation to individual nodes in the system that can be useful for dynamic load balancing.

We believe that the proposed dynamic endpoints extension cleanly integrates with the MPI specification. Endpoints make concrete the existing implicit association of threads with processes by treating a group of threads and an associated endpoint as an MPI process. These refinements to the MPI standard establish a self-consistent interaction between endpoints and the existing MPI interface, and provide important flexibility that is needed to fully harness the performance potential of future systems.

Acknowledgments

We thank the members of the MPI Forum, the MPI Forum hybrid working group, and the MPI community for discussions related to this work. This work was supported by the U.S. Department of Energy under contract DE-AC02-06CH11307.

7. REFERENCES

[1] Berkeley UPC. Berkeley UPC user's guide version 2.16.0. Technical report, U.C. Berkeley and LBNL, 2013.

[2] Milind Bhandarkar, L. V. Kale, Eric de Sturler, and Jay Hoeflinger. Object-Based Adaptive Load Balancing for MPI Programs. In *Proc. Intl. Conf. on Computational Science*, volume LNCS 2074, pages 108–117, May 2001.

[3] James Dinan, Pavan Balaji, Ewing Lusk, P. Sadayappan, and Rajeev Thakur. Hybrid parallel programming with MPI and unified parallel C. In *Proc. 7th ACM international conference on Computing frontiers*, CF '10, 2010.

[4] Ulrich Drepper. ELF handling for thread-local storage. Technical report, Red Hat, Inc., December 2005.

[5] Andrew Friedley, Torsten Hoefler, Greg Bronevetsky, Andrew Lumsdaine, and Ching-Chen Ma. Ownership passing: Efficient distributed memory programming on multi-core systems. In *Proc. 18th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2013.

[6] David Goodell, William Gropp, Xin Zhao, and Rajeev Thakur. Scalable memory use in MPI. In *Proc. Recent Adv. in MPI - 18th European MPI Users' Group Meeting*, EuroMPI 2011, September 2011.

[7] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. Leveraging MPI's one-sided communication interface for shared-memory programming. In *Proc. Recent Adv. in MPI - 19th European MPI Users' Group Meeting*, EuroMPI '12, September 2012.

[8] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. MPI+MPI: A new, hybrid approach to parallel programming with MPI plus shared memory. *J. Computing (to appear)*, 2013.

[9] Feng Ji, Ashwin M. Aji, James Dinan, Darius Buntinas, Pavan Balaji, Rajeev Thakur, Wu-Chun Feng, and Xiaosong Ma. MPI-ACC: An integrated and extensible approach to data movement in accelerator-based systems. In *Proc. 14th IEEE Intl. Conf. on High Performance Computing and Communications*, HPCC '12, June 2012.

[10] Humaira Kamal and Alan Wagner. FG-MPI: Fine-grain MPI for multicore and clusters. In *11th IEEE Intl. Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC)*, pages 1–8, 2010.

[11] S. Kumar, A.R. Mamidala, D.A. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, Dong Chen, and B. Steinmacher-Burrow. PAMI: A parallel active message interface for the Blue Gene/Q supercomputer. In *Proc. 26th Intl. IEEE Parallel & Distributed Processing Symposium (IPDPS)*, May 2012.

[12] S. Li, T. Hoefler, , and M. Snir. NUMA-Aware shared memory collective communication for MPI. In *Proc. 22nd Intl. ACM Symp. on High-Performance Parallel and Distributed Computing (HPDC)*, Jun. 2013.

[13] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. `http://www.mpi-forum.org/docs/docs.html`.

[14] MPICH – A portable implementation of MPI. `http://www.mpich.org`.

[15] Over 100 Million MPI Processes with MPICH, January 2013. `http://www.mpich.org/2013/01/15/over-100-million-processes-with-mpich/`.

[16] Lorna Smith and Mark Bull. Development of mixed mode MPI / OpenMP applications. *Scientific Programming*, 9(2,3):83–98, 2001.

[17] Jeff A. Stuart, Pavan Balaji, and John D. Owens. Extending MPI to accelerators. In *ASBD 2011: First Workshop on Architectures and Systems for Big Data*, October 2011.

[18] Jesper Larsson Träff. Compact and efficient implementation of the MPI group operations. In *Proc. Recent Adv. in MPI - 17th European MPI Users' Group Meeting*, EuroMPI 2010, September 2010.

[19] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Singh, Sayantan Sur, and Dhabaleswar Panda. MVAPICH2-GPU: Optimized GPU to GPU communication for InfiniBand clusters. In *Proc. Intl. Supercomputing Conf.*, ISC, 2011.