Special Issue Article

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
2014, Vol. 28(4) 390–405

DOI: 10.1177/1094342014548772
hpc.sagepub.com
SAGE

Enabling communication concurrency through flexible MPI endpoints

James Dinan[1], Ryan E Grant[2], Pavan Balaji[3], David Goodell[4], Douglas Miller[5], Marc Snir[3] and Rajeev Thakur[3]

Abstract

MPI defines a one-to-one relationship between MPI processes and ranks. This model captures many use cases effectively; however, it also limits communication concurrency and interoperability between MPI and programming models that utilize threads. This paper describes the MPI endpoints extension, which relaxes the longstanding one-to-one relationship between MPI processes and ranks. Using endpoints, an MPI implementation can map separate communication contexts to threads, allowing them to drive communication independently. Endpoints also enable threads to be addressable in MPI operations, enhancing interoperability between MPI and other programming models. These characteristics are illustrated through several examples and an empirical study that contrasts current multithreaded communication performance with the need for high degrees of communication concurrency to achieve peak communication performance.

Keywords

MPI, endpoints, hybrid parallel programming, interoperability, communication concurrency

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.

In the MPI+X programming model, multiple cores are utilized by a single MPI process with a shared MPI rank. As a result, communication for all cores in the MPI process is effectively funneled through a single MPI address and its corresponding communication context. As processor core counts have increased, high-speed interconnects have evolved to provide greater resources to support concurrent communications with multiple cores. For such networks, a growing number of cores must be used in order to realize peak performance (Underwood et al., 2007; Blagojević et al., 2010; Dózsa et al., 2010; Barrett et al., 2013). This situation is at odds with conventional hybrid programming techniques, where the node is partitioned by several MPI processes and communications are funneled through a small fraction of the cores.

Interoperability between MPI and other parallel programming systems has long been a productivity and composability goal in 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 Intel® 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 several issues affecting hybrid parallel programming with MPI and threads, but it did not include any new mechanisms to address these foundational communication

[1]Intel Corporation, Hudson, MA, USA
[2]Sandia National Laboratories, Albuquerque, NM, USA
[3]Argonne National Laboratory, Lemont, IL, USA
[4]Cisco Systems Incorporated, San Jose, CA, USA
[5]International Business Machines Corporation, Rochester, MN, USA

Corresponding author:
James Dinan, Intel Corporation, 75 Reed Road, Hudson, MA 01749, USA.
Email: james.dinan@intel.com

concurrency and interoperability challenges. In current multithreaded MPI programs, 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.

Several solutions to the communication concurrency challenge have been explored, including internally parallelizing MPI processing (Kumar et al., 2012; Tanase et al., 2012) and modifying the networking layer to enable greater concurrency for threads using the current MPI interface (Luo et al., 2011). However, these approaches can require customized thread-management techniques to avoid overheads and noise from oversubscribing cores, and they are not able to address the programmability challenges that arise when MPI threads share a rank.

In this paper we present an MPI extension, called "MPI endpoints", that enables the programmer to create additional ranks at existing MPI processes. We explore the design space of MPI endpoints and the impact of endpoints on MPI implementations. We illustrate through several examples that endpoints address the problem of interoperability between MPI and parallel programming systems that utilize threads. Additionally, we explore how endpoints enrich the MPI model by relaxing the one-to-one relationship between ranks and MPI processes. We explore this new potential through a brief example that utilizes endpoints to achieve communication-preserving load balancing by mapping work (e.g. mesh tiles) to ranks and reassigning multiple ranks to processes. We also conduct an empirical study of communication concurrency in a modern many-core system. Results confirm that multiple cores are required to drive the interconnect to its full performance. They further suggest that private, rather than shared, communication endpoints may be necessary to achieve high levels of communication efficiency.

2. Background

The design of MPI communicators has been vigorously debated within the MPI community, and several approaches to increasing the generality of MPI communicators have been suggested (Geist et al., 1996; Demaine et al., 2001; Graham and Keller, 2009). Led by the authors, members of the MPI Forum have rekindled this discussion in the context of the MPI endpoints extension that is proposed for MPI 4.0.[1] Initially, static interfaces were explored, and we present these designs in Section 3.1. The dynamic interface, presented in Section 3.2, is preferred as a more flexible alternative to the static interface, and it was refined as described in Section 3.5 into a proposal that is currently under consideration for inclusion in version 4.0 of the MPI standard.

MPI interoperability has been investigated extensively in the context of a variety of parallel programming models (Jose et al., 2010, 2012; Yang et al., 2014). 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 (Dinan et al., 2010). Support for the one-to-one mapping cannot be provided in MPI 3.0 when the unified parallel C (UPC) implementation utilizes operating system threads, rather than processes, to implement UPC threads. However, this mode of operation can be supported through the endpoints interface by assigning endpoint ranks to threads.

Hybrid parallel programming, referred to as "MPI+X", which combines MPI and a node-level parallel programming model, has become commonplace. MPI is often combined with multithreaded parallel programming models, such as MPI+OpenMP (Smith and Bull, 2001; Rabenseifner et al., 2009). The 1997 MPI 2.0 Forum (Message Passing Interface Forum, 1997) 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 clarified the interaction between several MPI constructs and threads. For example, matched-probe operations were added to enable deterministic 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 (Hoefler et al., 2012, 2013).

Recently, researchers have endeavored to integrate MPI and accelerator programming models. This effort has focused on the impact of separate accelerator memory on communication operations. Several approaches to supporting the use of NVIDIA* CUDA* or OpenCL* buffers directly in MPI operations (Wang et al., 2011; Ji et al., 2012) have been developed. Other efforts have focused on enabling accelerator cores to perform MPI calls directly (Stuart et al., 2011).

Numerous efforts have been made to integrate node-level parallelism with MPI. Fine-Grain MPI (FG-MPI) (Kamal and Wagner, 2010) implements MPI processes as lightweight coroutines instead of operating-system processes, enabling each coroutine to have its own MPI

rank. In order to support fine-grain coroutines, FG-MPI provides mechanisms to create additional ranks when MPI is initialized, similar to the approach described by the static endpoints interface. FG-MPI programs have been run with as many as 100 million MPI ranks using this technique (University of British Columbia, 2013). Li et al. (2013) recently demonstrated significant performance improvements in collective communication when the node is not partitioned into individual MPI processes; that is, endpoints are used instead of one process per core. Hybrid MPI (HMPI) transmits ownership for shared buffers to improve the performance of intranode communication (Friedley et al., 2013); techniques such as this can be leveraged to optimize intranode communication between endpoints.

3. Communication endpoints

We define an *MPI endpoint* as a set of resources that supports the independent execution of MPI communications. The central resource that backs an MPI endpoint is a rank within an MPI communicator, which identifies the endpoint in MPI operations. In support of this rank, the MPI implementation may utilize additional resources such as network command queues, network addresses, or message queues. One or more threads are logically associated with an endpoint, after which the thread can perform MPI operations using the resources of the endpoint. In this context, a conventional MPI process can be thought of as an MPI endpoint and a set of threads that perform operations using that endpoint.

The current MPI standard defines a one-to-one mapping between endpoints and MPI processes. As shown in Figure 1, MPI endpoints relax this restriction by allowing an MPI process to contain multiple endpoints. 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 created, how ranks are associated with endpoints, and how threads are mapped to endpoints.

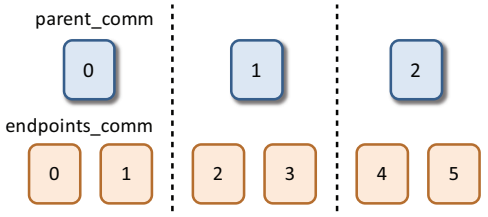

Figure 1. Flexible communication endpoints extend MPI with a many-to-one mapping between ranks and processes.

3.1. Static endpoint creation

During initialization, MPI creates the `MPI_COMM_WORLD` communicator, which contains all MPI processes. Static approaches to the endpoints interface make endpoints members of this communicator and require endpoints to be created during launching or initialization of an MPI execution. In order to use endpoints created in the `MPI_COMM_WORLD` communicator, threads must first be associated with an endpoint through an attach function, described in Section 3.3.

In one approach to supporting static endpoints, the programmer requests additional endpoints as an argument to mpiexec, as in Dinan et al. (2010) and Kamal and Wagner (2010). This approach can require `MPI_Init` or `MPI_Init_thread` to be extended so that it can be called multiple times per instance of the MPI library, once for each endpoint. This is perceived to be a disruptive change to the existing MPI specification, which states that multiple calls to `MPI_Init` are erroneous.

As an alternative to this approach, a new MPI initialization routine can be added that allows the programmer to call the MPI initialization routine once per operating system process and indicate the number of endpoints required by each of those processes.

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

In this new initialization routine, argc and argv are the command line arguments, count indicates the desired number of endpoints at the calling 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.

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 initialized the network, for example, in the case of implementations based on the IBM® parallel active message interface (PAMI) low-level communication library (Kumar et al., 2012). A significant drawback to this approach is that statically specifying the number of endpoints restricts the ability of MPI to interoperate fully with models where the number of threads is dynamic. In addition, statically selecting the number of endpoints can limit opportunities for libraries to utilize endpoints. To better support such use cases, we next describe an alternative approach that allows endpoints to be created dynamically.

3.2. Dynamic endpoint creation

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.[2]

In the dynamic endpoints interface we extend the group component of communicators, such that multiple ranks in a new group can be associated with each rank in the parent group. An example of this interface is as follows.

```
int MPI_Comm_create_endpoints(
    MPI_Comm parent_comm, int my_num_ep,
    MPI_Info inf, 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`. These ranks, or endpoints, of the output communicator are ordered sequentially and numbered by using the same relative 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 must be performed concurrently on all ranks (i.e. endpoints) in the given communicator.

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, one can define `MPI_Comm_free` in a way that enables a single thread to call `MPI_Comm_free` separately for each endpoint. In order to support this semantic, the MPI implementation must internally aggregate endpoint-free requests and delay communication until the last endpoint associated with a given conventional process calls `MPI_Comm_free` on the given communicator. At this point, any communication needed can be performed by a subset of the endpoints in the communicator.

3.3. Associating threads with endpoints

In order to use an endpoint, a thread must associate its operations with a rank in the given communicator. Identifying the specific endpoint can be accomplished through explicit or implicit approaches. The design choice can have a significant impact on performance and interoperability with threading models.

3.3.1. Explicit binding. A goal of the static endpoints case is to allow the programmer the convenience of operating directly on `MPI_COMM_WORLD`, as is commonly done today. In order to communicate on a world communicator that contains multiple endpoints, each thread must bind itself to a specific rank, as follows.

```
int MPI_Endpoint_attach(int id)
```

This approach binds the calling thread to the local endpoint with the id between 0 and the number of local endpoints. A consequence of this interface is that the MPI implementation must store the association between a thread and its endpoint using thread-local storage (TLS), which requires support from the threading package that is being used. In addition, it requires a TLS lookup in every MPI operation on the given communicator, which can significantly increase software overheads in MPI processing.

The explicit binding approach can also be used with the dynamic endpoints interface, if an additional communicator argument is added to the routine. 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. A disadvantage to explicit binding is that integration of the MPI library with the specific threading package may be required in order to support this functionality. Moreover, TLS lookups are necessary to identify the endpoint in use by a given thread.

3.3.2. Implicit binding. The dynamic interface enables an alternate, more flexible strategy to associating endpoints with threads. In this interface, we return separate communicator handles, one per endpoint. When a thread wishes to use a particular endpoint, it simply uses that endpoint's communicator handle in the given MPI operation. For example, `MPI_Comm_rank` can be used to query the rank of the given endpoint in the communicator. This approach is agnostic to the threading package used, and it allows MPI to store any endpoint-specific information internally using a structure that is referenced through the communicator handle, rather than through thread-local storage.

3.4. Progress semantics

The MPI standard specifies a minimal progress requirement: Communication operations on a given process, 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, since they could be treated as independent processes or as part of the process that created them.

Treating endpoints as part of the process from which they were created leads to the simplest progress semantics. This would require that the MPI implementation ensures progress is made on all endpoints of a process whenever any MPI operation is performed. This is easy for users to reason about, especially for an endpoints interface that does not require threads to be bound 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; however, it provides additional opportunities for thread isolation and communication concurrency within the MPI library. In this model, it can be helpful to the MPI implementation if users declare an association of threads with endpoints through attach operations or performance hints in an MPI info object. Such mechanisms can be used to make the MPI runtime system aware that a limited number of threads communicate on a given endpoint. When only a single thread uses a given endpoint, the MPI implementation can eliminate internal synchronizations, potentially providing lockless multithreaded communication.

3.5. Proposed endpoints interface

After reviewing the various approaches to specifying an endpoints extension to MPI, the MPI hybrid working group has put forward the dynamic endpoints proposal for consideration (MPI Forum Hybrid Working Group, 2013). This proposal calls for the addition of a new `MPI_Comm_create_endpoints` function that returns an array of handles to the output endpoints communicator. No explicit binding of endpoints with threads is required; however, an MPI info object can be provided to this function to specify the level of threading support that will be used on the given endpoint, enabling internal optimizations within the MPI implementation. In this proposal, endpoints are defined to behave as MPI processes and are independent in the MPI progress semantic. For the remainder of this paper, we focus on this proposed endpoints interface.

4. Impact on MPI implementations

We now explore the expected impact of endpoints on existing MPI implementations. Our discussion focuses on the popular, open source MPICH implementation of MPI (MPICH, 2013) from which many commercial MPI implementations are derived. However, the discussion is generally applicable to other MPI implementations.

4.1. Internal communicator representation

MPI communicators are logically composed of an MPI group and a communication context. MPI groups are opaque data structures that represent ordered sets of MPI processes. Groups are local to an MPI process, are immutable once created, and are derived from existing communicators or other groups. The communication context refers to the internal information that an MPI implementation uses to match communication operations according to the communicator on which they were performed. Most MPI implementations identify the communication context using an integer value that is unique across the processes in the group of the communicator. The context ID is typically combined with the source process rank and tag into a single quantity (e.g. a 64-bit integer) that is included in the MPI message header and used for matching.

In MPICH, communicators are internally represented through a structure at each process that contains several important components: the process's rank in the communicator, the size of the communicator, and a table that is used to map a given destination rank to a *virtual connection* (VC). This mapping structure is currently a dense array indexed by communicator rank, though more compact implementations are possible (Goodell et al., 2011).

MPI applications refer to a given communicator using an `MPI_Comm` handle. A communicator handle is an opaque reference that can be used by the MPI implementation to retrieve the internal communicator structure. In MPICH, it is implemented as an integer value, which satisfies the MPI specification requirements and simplifies Fortran language bindings. 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. While the handles identify a particular endpoint in the given communicator, the MPI implementation is expected to share one or more underlying structures. For example, just as normal communicators that are duplicated by `MPI_Comm_dup` may easily share the rank-to-VC map, endpoints communicators within a process may also easily share this mapping structure within an MPI process among all the endpoints communicators derived from the same call to `MPI_Comm_create_endpoints`.

In order to extend MPI groups to endpoints communicators, groups must be generalized to incorporate endpoints as well as conventional MPI processes, and group and communicator comparison and manipulation functions must also be updated. We add the `MPI_ALIASED` result for communicator and group comparisons when the communicator handles correspond to different endpoints in the same communicator.

Group manipulations are traditionally viewed as mapping problems (Träff, 2010) from the dense group space to a *global process ID* of some type.[3] 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 or reused as long as any communicator or group contains a reference to the endpoint associated with that ID. The MPI-2 dynamic process interface has a similar requirement; thus, endpoints can utilize existing infrastructure that supports this interface.

4.2. Associating endpoints with connections

Each MPICH VC in the rank-to-VC mapping table represents a *logical* connection to another MPI process, even though the underlying network may be connectionless. Endpoint ranks must be associated with a VC and entered into this table for each endpoints communicator. This can be accomplished by creating additional VCs for each endpoint (e.g. by creating additional network addresses or connections), by multiplexing endpoints over existing VCs, or through some combination of these approaches. In current implementations, VCs must be unique within a communicator, since each process may hold only a single rank in a given communicator. The addition of endpoints may require us to relax the uniqueness restriction, breaking an otherwise injective relationship. While rank-to-VC translation is important, the reverse lookup is never performed; thus, the loss of uniqueness in the rank-to-VC mapping does not present any new challenges.

When endpoints are multiplexed across one or more VCs, and operations from multiple endpoints share receive queues, the destination rank must be included as an additional component in the MPI message-matching scheme to distinguish the intended endpoint. If the destination rank is not already included in the message header or match bits, it must be added in order to facilitate this additional matching or demultiplexing step. While this implementation option adds some complexity, it can reduce or eliminate the need to create additional network addresses or connections, and it potentially provides the user access to a greater number of endpoints.

The dynamic endpoints interface introduces challenges for interconnects that must create network endpoints when `MPI_Init` is called. Such networks may be unable to create additional network endpoints later, such as an implementation of MPI over PAMI (Kumar et al., 2012). For such a network, the implementation may create all endpoints when `MPI_Init` is called, possibly directed by runtime parameters or environment variables. Implementations on such networks can also 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 message matching

MPI message matching is performed at the receiver using the ⟨*Communicator_Context_ID*, *Source_Rank*, *Tag*⟩ triples, and operations that match a given triple, including those that use the `MPI_ANY_SOURCE` and `MPI_ANY_TAG` wildcards, must be matched in the order in which they were posted. Most MPI implementations utilize a set of queues to track receive operations and match them with incoming sends according to the required ordering.

While it is possible to utilize separate queues for each communicator, a single pair of queues is often used. These queues are referred to as the *posted* and *unexpected* receive queues. The posted receive queue contains the list of receive operations that have been performed for the given process. This includes blocking and nonblocking operations performed by all threads in a given process. The unexpected receive queue contains an ordered list of messages that arrived but did not match any posted receive. For small messages, the implementation may store the full unexpected message in system buffer space; for larger messages, additional data may need to be fetched from the sender. When a receive operation is posted, the MPI implementation must first search the unexpected message queue to check if it has already arrived. Otherwise, the receive operation is appended to the posted receive queue.

MPICH currently supports shared message queues that contain operations posted across all communicators. When shared message queues are used, MPI message matching is performed by using the full ⟨*Communicator_Context_ID*, *Source_Rank*, *Tag*⟩ triples. Shared message queues can pose significant challenges because of thread synchronization. Therefore, per-receiver/endpoint message queues may be needed for highly threaded MPI implementations. No receiver wildcard is present in MPI; thus message queues can be split per endpoint. Such an implementation can reduce or potentially eliminate synchronization when accessing per-endpoint message queues.

4.4. 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 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, and it also poses basic maintainability and testability challenges to MPI implementations.

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. Although some threading models and architectures have support for fast TLS access (Drepper, 2005), many do not. A design utilizing separate communicator handles per endpoint is advantageous because it keeps TLS off the communication critical path. The design tradeoff 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.

4.5. 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 network interface cards (NICs) direct memory access first in first out (DMA FIFOs), or PAMI contexts (Kumar et al., 2012). 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 by giving the programmer a greater amount of freedom in mapping ranks to processes. This capability can have a broad impact on interoperability as well as mapping of the computation. In this section, we demonstrate these capabilities by using the proposed endpoints interface described in Section 3.5. 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++, Coarray Fortran (CAF), X10, and Chapel. We also demonstrate that the relaxation of process–rank mapping enables new approaches to computation mapping 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, threads at all nodes are able to participate in a call to `MPI_Allreduce` within the OpenMP parallel region. This example highlights one possible use of endpoints. Many different usages are possible, including schemes where `MPI_COMM_SELF` is used as the basis for the endpoints communicator, allowing the programmer to construct a communicator that can be used to perform MPI communication among threads within a process, enabling a multilevel parallel structure where MPI can be used within the node and between nodes.

In this example, we assume 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, since the number of cores on a node typically drives the number of threads allowed as well as the available network resources. This example calls `MPI_Comm_create_endpoints` from the master thread within the OpenMP parallel region in order to create exactly one thread per endpoint. Each thread then obtains its handle to the endpoints communicator

```
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_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.

and utilizes this private handle to behave as a separate MPI process. Thus, each thread is able to call `MPI_Allreduce` with its communicator handle as if they were separate MPI ranks in separate processes. Since the threads share a process and address space, the MPI implementation can use a hierarchical collective algorithm to optimize the reduction of data between local threads. As we will show in Section 6, the MPI implementation can utilize the additional threads and endpoint resources to increase network throughput.

5.2. Systemwide hybrid programs

Endpoints can be used to enable interoperability with system-level programming models that use threads for on-node execution, for example, 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 (Dinan et al., 2010). The UPC specification allows UPC threads to be implemented by 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 Dinan et al. (2010), 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, calling MPI_Init multiple times violates the MPI specification, and it results in all endpoints being within `MPI_COMM_WORLD`, a situation that 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` endpoints 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, the approach used by several popular UPC implementations (UC Berkeley and LBNL, 2013). Once

```
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. Endpoints enable UPC threads to behave as MPI processes.

```
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) {
  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.

the execution has been bootstrapped, a 'flat' endpoints communicator is created, UPC threads are spawned, and threads register the endpoints communicator with an interoperability library and then 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 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 (Bhandarkar et al., 2001), 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 (e.g. mesh tiles) 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

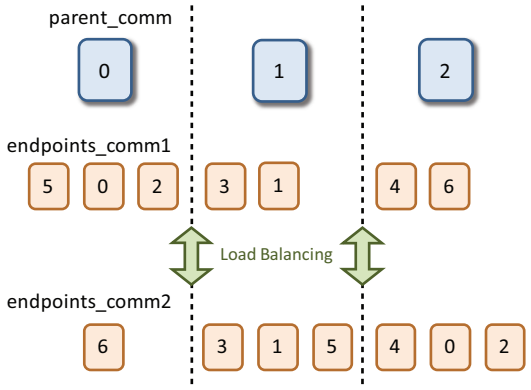

Figure 2. Communication-preserving dynamic load balancing via endpoints migration.

between endpoints is preserved. Such a communication-preserving approach to dynamic load balancing can provide an effective solution for adaptive mesh computations.

While this approach to load balancing is promising, it requires the programmer to manually communicate computational state from the previous thread or process responsible for an endpoint to the endpoint's new owner. This introduces interesting opportunities for the development of new parallel programming models and libraries that can harness such a feature. For example, a library could interface with existing load-balancing or work-partitioning tools to generate the remapped communicator. Endpoint or mesh tile state could then be automatically migrated from locations in the old communicator to locations in the new communicator.

6. Empirical performance study

We measure the impact of multithreaded communication on a commercial MPI implementation running on a modern, many-core HPC cluster. Through this study, we demonstrate the need for communication concurrency—for multiple cores to drive communications in parallel—in order to achieve high levels of communication performance. We compare the impact of shared versus private communication environments to illustrate that many cores must drive communication to achieve high levels of throughput and that a significant performance gap currently exists between multithreaded and single-threaded communication. This data suggests that an endpoints approach that enables threads to acquire private MPI communication contexts can be used to close this gap and relieve pressure on intranode thread/process count tradeoffs.

We conducted experiments using two Intel® Xeon Phi™ Coprocessor 5110P cards, each with 8 GB of memory, and 60 1.053 GHz in-order cores with 4-way hyperthreading. Each card runs a version 2.6.38 Linux* kernel adapted for the Xeon Phi. All code was run with the Intel® MPI Library version 4.1 update 1, the Intel® Manycore Platform Software Stack (MPSS) version 3.1, and compiled with the Intel® C Compiler version 13.1.2. Benchmarks were run in native mode on the Xeon Phi, in which application code and MPI operations are performed directly on the Xeon Phi. While this leaves the host processor idle, it is more representative of future systems where the many-core processor is no longer incorporated into the system as a separate device. The Xeon Phi cards are located within different compute nodes and are connected using Mellanox* 4X QDR InfiniBand* HCAs, with a maximum data rate of 32 Gb/s. The Xeon Phi coprocessor is representative of the growing trend in HPC toward many-core processors and highly multithreaded programs. Such scenarios are strong motivators for endpoints; however, endpoints are expected to be effective at improving communication throughput in any scenario where multithreaded MPI processes are used.

We adapted the Ohio State University (OSU) MPI message rate benchmark (Ohio State University, 2013) to support threads and allow varying of the unexpected and expected queue depths. This benchmark is executed on two nodes and is bidirectional; for a P process experiment, each node contains P sender processes and P receiver processes. Thus, each sender process is paired with a receiver process on the remote node. Threading support directly uses POSIX* threads, and MPI was initialized in the `MPI_ THREAD_MULTIPLE` mode to support concurrent MPI calls by multiple threads. In order to build expected queue lengths, the benchmark was adapted to post a given number of receive operations to the expected queue with a nonmatching tag. This strategy causes the receiving MPI process to traverse the messages with the nonmatching tag to check for matches each time a message arrives. In the case of the unexpected message queue, a given number of messages with nonmatching tags were sent to the receiver before the timed message sends occurred. This caused the receiver to traverse the list of unexpected messages to search for matches every time a message was received. We use these modifications to simulate the impact of multiple threads sharing a communication context; when multiple threads share a single MPI rank and communicator, MPI's message matching must be performed across all operations performed by all threads.

6.1. Impact of communication concurrency on network throughput

We first demonstrate the need for communication concurrency in order to achieve high levels of network throughput, by comparing throughput for shared versus private communication contexts. We model idealized endpoints as individual MPI processes with

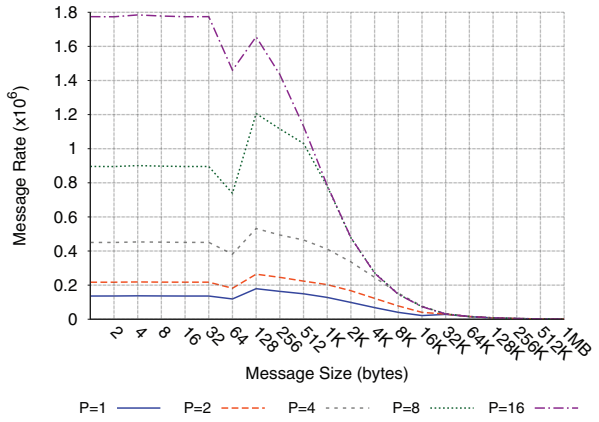

Figure 3. Impact of communication concurrency on message rate, for $P = 1\ldots16$.

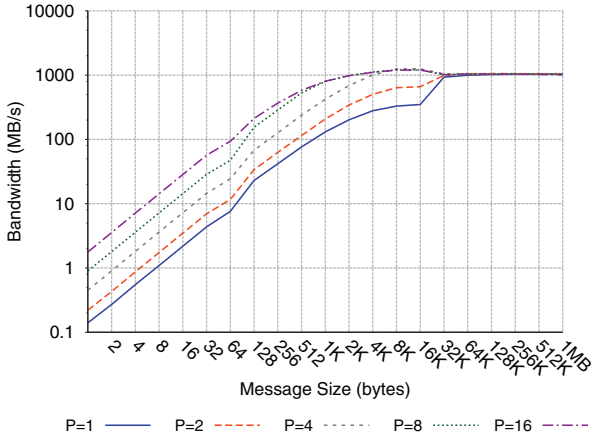

Figure 4. Impact of communication concurrency on bandwidth, for $P = 1\ldots16$.

private communication states. This represents an upper bound on the performance that can be achieved by MPI endpoints. In Figure 3, we show the message rate that can be achieved when using $P = 1\ldots16$ sending processes on each Xeon Phi processor to inject messages. This corresponds to a total of 2 to 32 active MPI processes per card, since each sending process is matched with a receiving process. From this data, we clearly see that multiple MPI processes enable improvements in the aggregate message rate achieved by each Xeon Phi processor. Message rate increases with process count, with the 16-process case resulting in an approximately 13× improvement over a single process for small messages. The dip in performance at 64 B messages corresponds directly to the cache line size of the Phi. The resulting increase in performance for 128 B messages is related to better utilization of the main memory on the Phi.

In Figure 4 we show the corresponding bandwidth for this experiment. We see that bandwidth scales with communication concurrency, with a greater number of processes having a positive impact on bandwidth between $P = 1$ and $P = 16$ cases, but with diminishing returns after $P = 8$ as the system becomes saturated. At 32 KB message sizes and higher, architectural limitations on our platform prevent bandwidth from scaling with greater concurrency. Newer CPU generations remove this architectural limitation, and we expect that bandwidth will continue to benefit from concurrency for large message sizes in future systems.

6.1.1. Impact of threads on network throughput

Figure 5 shows the message rate achievable using multiple threads, as well as multiple threads with multiple MPI processes. This should be regarded as the lower bound for the performance of endpoints, as the threads share MPI state between them for a given MPI process.

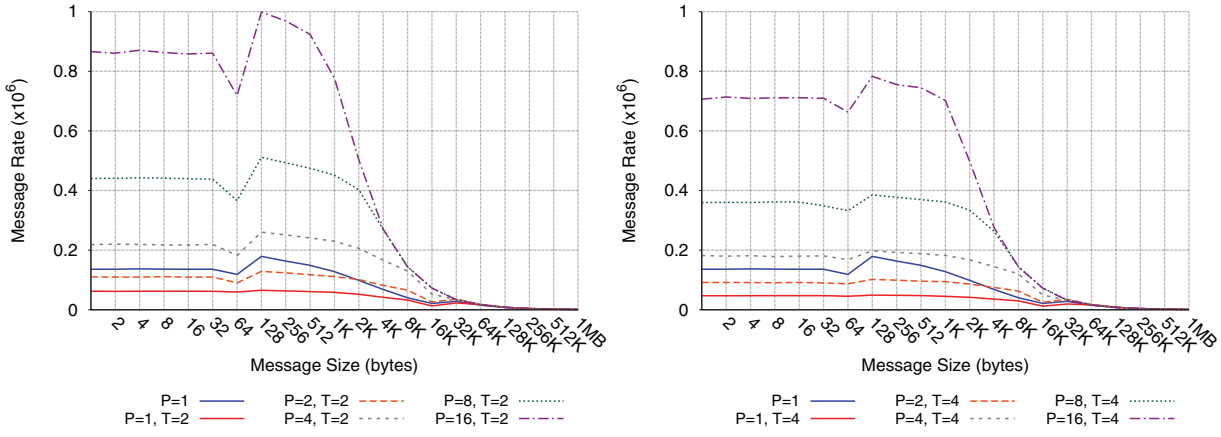

Figure 5. Impact of number of threads on message rate, with processes $P = 1\ldots16$ and threads $T = 2$ and $T = 4$ per process.

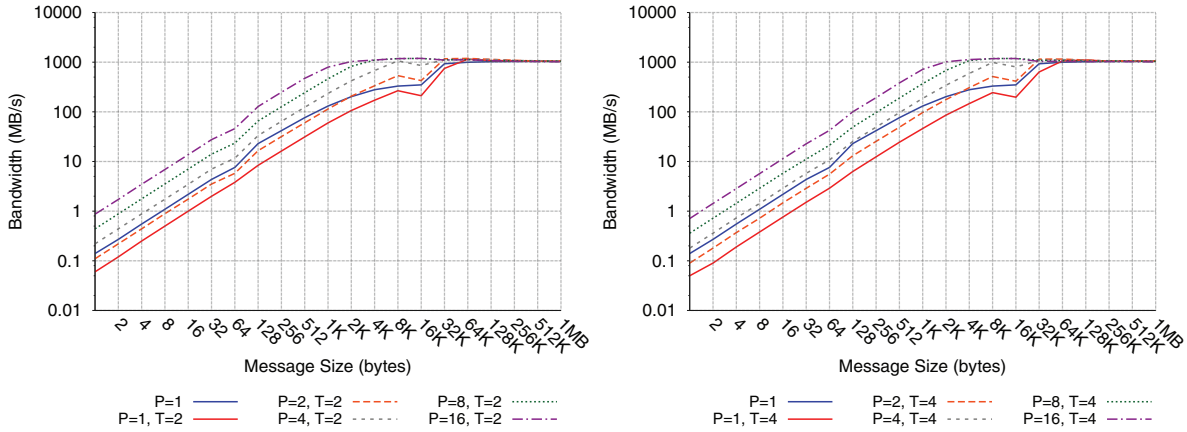

Figure 6. Impact of number of threads on bandwidth, with processes $P = 1 \ldots 16$ and threads $T = 2$ and $T = 4$ per process.

Therefore, the performance of endpoints should lie between the multithread performance in Figure 5 and the upper bound in Figure 3. The benchmark adaptations to enable multiple threads are considered in the calculation of the benchmark time; therefore the overhead involved in creating and joining the threads is included in our measurements. Performance-conscious concurrency methods can avoid this overhead by reusing threads after they have been created, avoiding the costs of multiple thread creations. In each experiment, however, the thread fork and join overheads are amortized over many communications and have no measurable impact on timings.

The multithreaded message rate results show lower performance with threads versus processes, as a result of threads sharing communication and MPI library state. We thus focus our exploration on a mixture of processes without threads, with two threads, and with four threads. This is representative of current thread/process tradeoffs made by users of HPC systems. For small messages, we see that the configuration of four processes with two threads yields message rates that exceed those of the single MPI process case. In Figure 6, we see that large message bandwidths can slightly outperform the single-process case with only two threads in a single process. However, this is a relatively small improvement that occurs within the region of the plot where bandwidth converges for all configurations. Having multiple processes and multiple threads clearly benefits bandwidth, with scaling occurring even for $P = 16$, where a $9.25\times$ performance improvement in bandwidth over the single-process case is observed for small messages.

6.2. Impact of shared communication state on message queue processing

The results shown thus far have measured performance of different message sizes when the incoming messages match the first item in the message-matching queue. Also important is MPI performance with queues of lengths that better approximate those seen in real applications (Barrett et al., 2013). Such a consideration is especially important in the multithreaded case, since threads sharing a communicator must share the message queues associated with that communicator in order to ensure MPI's FIFO message-matching semantics.

Because receive operations posted by all threads must be considered during message matching, the message-matching overheads can increase proportionally to the number of threads performing communication. Figure 7 shows the message rate for given expected queue lengths for 8 B messages. The performance of most process/thread schemes remains steady for queue sizes of 64 or fewer posted received operations. After the queue length grows beyond 64 items, performance decreases, with more dramatic dropoff occurring after the queue grows to a length greater than 128.

The unexpected queue results in Figure 8 show similar results to those observed for the expected queue. The overall message rate is roughly equal to that for the expected queue, except that for the unexpected queue, the dropoff in message rate is slightly higher for 2 threads at a 256-item-long queue. By providing individual ranks that isolate threads within the MPI implementation, endpoints can avoid these increases in message-matching costs.

7. Summary

Endpoints provide a natural evolution of MPI that relaxes the one-to-one mapping of ranks to processes. This change provides improved interoperability between MPI and parallel programming models that use threads or other flexible execution strategies. In addition, endpoints can enable greater network throughput by allowing threads in a parent MPI process to drive multiple, independent network endpoints.

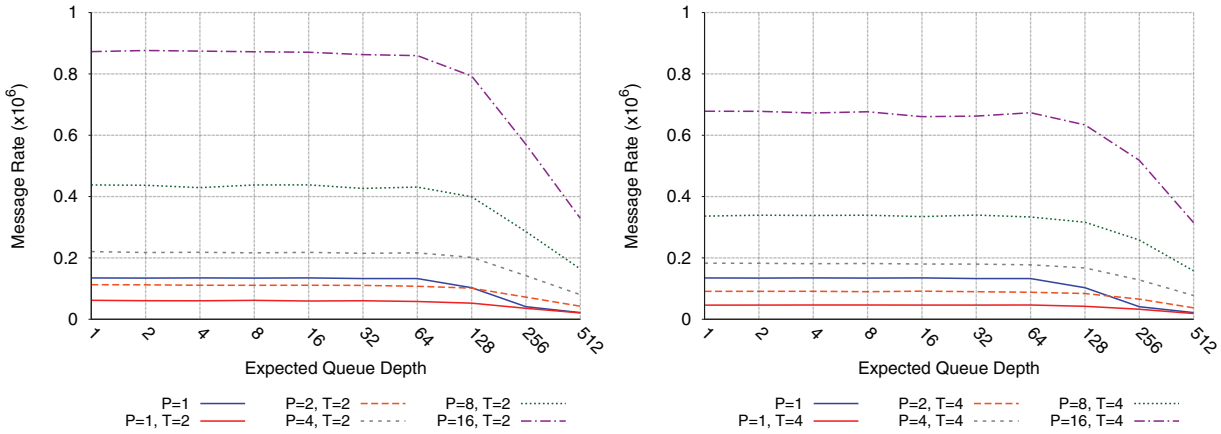

Figure 7. Impact of shared message queues on message rate for 8 B messages. The expected queue length is varied, and configurations with processes $P = 1 \ldots 16$ and threads $T = 2$ and $T = 4$ per process are evaluated.

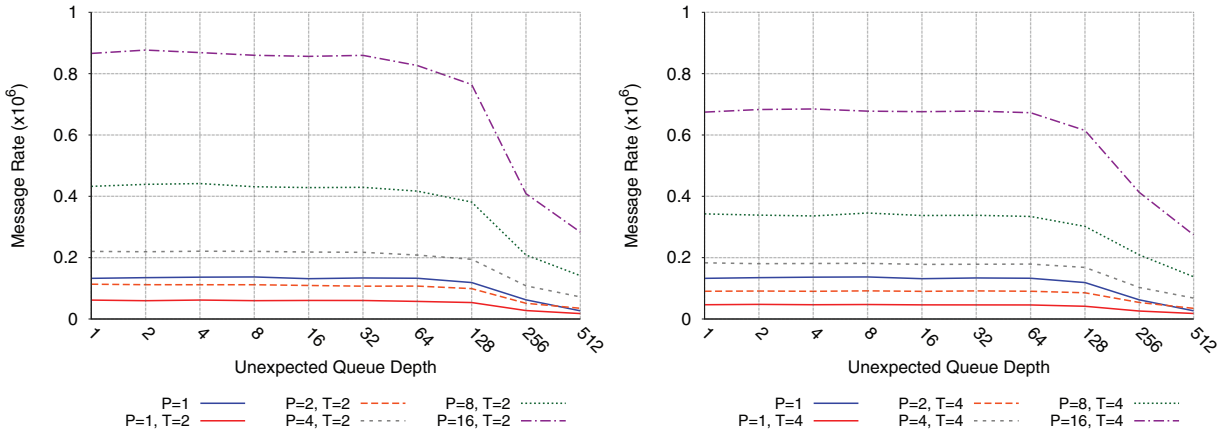

Figure 8. Impact of shared message queues on message rate for 8 B messages. The *un*expected queue length is varied, and configurations with processes $P = 1 \ldots 16$ and threads $T = 2$ and $T = 4$ per process are evaluated.

Using a modern many-core platform, we have demonstrated that increased communication concurrency results in greater network performance and that this level of performance is difficult to achieve in multithreaded environments where threads share communication states. We show that in this scenario, increasing the number of endpoints can provide a proportional boost to network performance.

The proposed dynamic endpoints extension requires the addition of a single new function and integrates with the MPI specification as an extension to the communicators interface. Endpoints make concrete the existing implicit association of threads with processes by treating a single thread or 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 the important flexibility that is needed to fully harness the performance potential of future systems.

We have summarized the lessons learned, motivations, and design decisions that led to the proposed MPI endpoints interface, and we have motivated it through studies of the hybrid programming and communication performance gaps that it is intended to address. However, more work is needed to explore efficient techniques for implementing the endpoints interface and incorporating it within applications.

Acknowledgements

We thank the members of the MPI Forum, the MPI Forum hybrid working group, and the MPI community for their engagement in this work.

Funding

This work was supported by the US Department of Energy (contract number DE-AC02-06CH11307). Sandia National Laboratories is a multiprogram laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary

of Lockheed Martin Corporation, for the US Department of Energy's National Nuclear Security Administration (contract number DE-AC04-94AL85000).

Notes

1. An earlier version of this paper was published in Dinan et al. (2013), and we extend this prior work with additional detail, including a detailed empirical study.
2. 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.
3. Though the same process need not be represented by the same value on two different processes, leading to the local process ID (LPID) concept of previous work (Goodell et al., 2011).

★ Other names and brands may be claimed as the property of others.

References

Barrett BW, Hammond SD, Brightwell R and Hemmert KS (2013) The impact of hybrid-core processors on MPI message rate. In: *20th european MPI users' group meeting (EuroMPI'13)*, Madrid, Spain, 15–18 September 2013, pp. 67–71. New York: ACM Press.

Bhandarkar M, Kale LV, de Sturler E and Hoeflinger J (2001) Object-based adaptive load balancing for MPI programs. In: *International conference on computational science (ICCS'01)*, San Francisco, USA, 28–30 May 2001, pp. 108–117. New York: Springer.

Blagojević F, Hargrove P, Iancu C and Yelick K (2010) Hybrid PGAS runtime support for multicore nodes. In: *4th conference on partitioned global address space programming models (PGAS'10)*. New York: ACM Press.

Demaine E, Foster I, Kesselman C and Snir M (2001) Generalized communicators in the message passing interface. *IEEE Transactions on Parallel and Distributed Systems* 12: 610–616.

Dinan J, Balaji P, Goodell D, Miller D, Snir M and Thakur R (2013) Enabling MPI interoperability through flexible communication endpoints. In: *20th european MPI users' group meeting (EuroMPI'13)*.

Dinan J, Balaji P, Lusk E, Sadayappan P and Thakur R (2010) Hybrid parallel programming with MPI and unified parallel C. In: *7th ACM international conference on computing frontiers (CF'10)*. New York: ACM Press.

Dózsa G, Kumar S, Balaji P, Buntinas D, Goodell D, Gropp W, et al. (2010) Enabling concurrent multithreaded MPI communication on multicore petascale systems. In: *17th european MPI users' group meeting (EuroMPI'10)*. Berlin: Springer-Verlag.

Drepper U (2005) ELF handling for thread-local storage. Report, Red Hat Incorporated. Available at : http://www.akkadia.org/drepper/

Friedley A, Hoefler T, Bronevetsky G, Lumsdaine A and Ma CC (2013) Ownership passing: Efficient distributed memory programming on multi-core systems. In: *18th ACM SIGPLAN symposium on principles and practice of parallel programming*. New York: ACM Press.

Geist A, Gropp W, Huss-Lederman S, Lumsdaine A, Lusk E, Saphir W, et al. (1996) MPI-2: Extending the message-passing interface. In: *2nd international Euro-Par conference (Euro-Par'96)* (ed L Bougé, P Fraigniaud, A Mignotte and Y Rober), Lyon, France, 26–29 August 1996, pp. 128–135. Berlin: Springer.

Goodell D, Gropp W, Zhao X and Thakur R (2011) Scalable memory use in MPI. In: *18th european MPI users' group meeting (EuroMPI'11)*.

Graham RL and Keller R (2009) Dynamic communicators in MPI. In: Ropo M, Westerholm J and Dongarra J (eds) *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin: Springer, pp. 116–123.

Hoefler T, Dinan J, Buntinas D, Balaji P, Barrett B, Brightwell R, et al. (2012) Leveraging MPI's one-sided communication interface for shared-memory programming. In: *19th european MPI users' group meeting (EuroMPI'12)*.

Hoefler T, Dinan J, Buntinas D, Balaji P, Barrett B, Brightwell R, et al. (2013) MPI+MPI: A new hybrid approach to parallel programming with MPI plus shared memory. *Computing* 95: 1121–1136.

Ji F, Aji AM, Dinan J, Buntinas D, Balaji P, Thakur R, et al. (2012) MPI-ACC: An integrated and extensible approach to data movement in accelerator-based systems. In: *14th international conference on high performance computing and communications (HPCC'12)*. Piscataway: IEEE Press.

Jose J, Kandalla K, Luo M and Panda DK (2012) Supporting hybrid MPI and OpenSHMEM over InfiniBand: Design and performance evaluation. In: *42nd international conference on parallel processing*, pp. 219–228.

Jose J, Luo M, Sur S and Panda DK (2010) Unifying UPC and MPI runtimes: Experience with MVAPICH. In: *4th conference on partitioned global address space programming model (PGAS'10)*, pp. 5:1–5:10. New York: ACM Press.

Kamal H and Wagner A (2010) FG-MPI: Fine-grain MPI for multicore and clusters. In: *11th international workshop on parallel and distributed scientific and engineering computing (PDSEC)*, pp. 1–8. Piscataway: IEEE Press.

Kumar S, Mamidala A, Faraj D, Smith B, Blocksome M, Cernohous B, et al. (2012) PAMI: A parallel active message interface for the Blue Gene/Q supercomputer. In: *26th international parallel & distributed processing symposium (IPDPS)*. Piscataway: IEEE Press.

Li S, Hoefler T and Snir M (2013) NUMA-Aware shared memory collective communication for MPI. In: *22nd international ACM symposium on high-performance parallel and distributed computing (HPDC'13)*. New York: ACM Press.

Luo M, Jose J, Sur S and Panda D (2011) Multi-threaded UPC runtime with network endpoints: Design alternatives and evaluation on multi-core architectures. In: *18th international conference on high performance computing (HiPC)*, pp. 1–10.

Message Passing Interface Forum (1997) MPI-2: Extensions to the Message-Passing Interface. Available at: http://www.mpi-forum.org/docs/docs.html

MPI Forum Hybrid Working Group (2013) MPI endpoints proposal. Available at: https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/380

MPICH (2013) MPICH - A portable implementation of MPI. Available at: http://www.mpich.org

Ohio State University (2013) OSU MPI benchmarks. Available at: http://mvapich.cse.ohio-state.edu/benchmarks

University of British Columbia (2013) *Over 100 Million MPI Processes with MPICH*, January 15. Available at: http://www.mpich.org/2013/01/15/over-100-million-processes-with-mpich/

Rabenseifner R, Hager G and Jost G (2009) Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: *17th euromicro international conference on parallel, distributed and network-based processing (PDP'09)*, pp. 427–436.

Smith L and Bull M (2001) Development of mixed mode MPI/OpenMP applications. *Scientific Programming* 9: 83–98.

Stuart JA, Balaji P and Owens JD (2011) Extending MPI to accelerators. In: *1st workshop on architectures and systems for big data (ASBD)*.

Tanase G, Almasi G, Xue H and Archer C (2012) Network endpoints for clusters of SMPs. In: *24th international symposium on computer architecture and high performance computing (SBAC-PAD)*, pp. 27–34. Piscataway: IEEE Press.

Träff JL (2010) Compact and efficient implementation of the MPI group operations. In: *17th european MPI users' group meeting (EuroMPI'10)*.

UC Berkeley and LBNL (2013) Berkeley UPC User's Guide version 2.16.0. Technical report, UC Berkeley and LBNL. Available at: http://upc.lbl.gov/docs/user/

Underwood KD, Levenhagen MJ and Brightwell R (2007) Evaluating NIC hardware requirements to achieve high message rate PGAS support on multi-core processors. In: *ACM/IEEE conference on supercomputing (SC'07)*. New York: ACM Press.

Wang H, Potluri S, Luo M, Singh A, Sur S and Panda D (2011) MVAPICH2-GPU: Optimized GPU to GPU communication for InfiniBand clusters. In: *26th international supercomputing conference (ISC'11)*, Hamburg, Germany, 19–23 June 2011. New York: ACM Press.

Yang C, Bland W, Mellor-Crummey J and Balaji P (2014) Portable, MPI-interoperable coarray Fortran. In: *19th SIGPLAN symposium on principles and practice of parallel programming (PPoPP'14)*, pp. 81–92. New York: ACM Press.

Author biographies

Dr Pavan Balaji holds appointments as a Computer Scientist at the Argonne National Laboratory, as an Institute Fellow of the Northwestern–Argonne Institute of Science and Engineering at Northwestern University, and as a Research Fellow of the Computation Institute at the University of Chicago. He leads the Programming Models and Runtime Systems group at Argonne. His research interests include parallel programming models and runtime systems for communication and I/O, modern system architecture (multicore, accelerators, complex memory subsystems, high-speed networks), and cloud computing systems. He has more than 100 publications in these areas and has delivered nearly 120 talks and tutorials at various conferences and research institutes.

Dr Balaji is a recipient of several awards including the US Department of Energy Early Career award in 2012, the TEDxMidwest Emerging Leader award in 2013, Crain's Chicago 40 under 40 award in 2012, the Los Alamos National Laboratory Director's Technical Achievement award in 2005, the Ohio State University Outstanding Researcher award in 2005, six best paper awards and various others. He has served as a Chair or Editor for nearly 50 journals, conferences and workshops, and as a technical program committee member for numerous conferences and workshops. He is a senior member of the IEEE and a professional member of the ACM. More details about Dr Balaji are available at http://www.mcs.anl.gov/ balaji.

Dr James Dinan is a Software Architect at Intel Corporation, where his work focuses on high-performance computing. Prior to joining Intel, he was the James Wallace Givens postdoctoral fellow at the Argonne National Laboratory. He received his PhD (2010) and MSc (2009) degrees in Computer Science from Ohio State University, in Columbus. He received his BSc degree (2004) in Computer System Engineering from the University of Massachusetts, Amherst. He is an active member of the MPI Forum standardization body and a contributor to the open source MPICH implementation of MPI. His research interests include parallel programming models, scalable runtime systems, scientific computing, operating systems, and computer architecture.

David Goodell graduated from the University of Illinois at Urbana–Champaign in 2005 with a BSc degree in Computer Engineering. He has served on the MPI Forum standards body starting with MPI 2.1 and has contributed to all subsequent revisions, including the currently proposed MPI 3.1 and MPI 4.0 standards. He is the author of numerous academic research papers and has served on the technical program committee for several conferences.

David is currently a Software Engineer at Cisco Systems Incorporated. He specializes in network driver and high-performance computing software development, with a particular focus on the MPI programming model. Prior to Cisco he worked at the Argonne National Laboratory on MPI implementation issues and various research projects. Before that he spent several years at Amazon working on a variety of projects, including the S3 storage service.

Ryan Grant is a postdoctoral appointee in the Scalable Software Systems group at Sandia National Laboratories. He graduated with a PhD degree in Computer Engineering from Queen's University in Kingston, Ontario, Canada in 2012. His research interests are in high-performance computing, with emphasis

on high-performance networking for Exascale systems. He is an active member of the Portals Networking Interface design team, a high-performance interconnect specification. He is an IEEE member and is actively involved in IEEE-sponsored conference organization.

Doug Miller is an Advisory Software Engineer at IBM, with 25 years' experience working with multiprocessor and multicore operating systems and applications. Doug has a BSc degree in Computer Science from Western Washington University and an Associate Degree in Computer Technology from Wake Technical College. Doug was an active contributor/author of the MPI 3.0 standard, focusing on multithreading and hybrid technologies.

Marc Snir is Director of the Mathematics and Computer Science Division at the Argonne National Laboratory and Michael Faiman and Saburo Muroga Professor in the Department of Computer Science at the University of Illinois at Urbana–Champaign. He currently pursues research in parallel computing.

He was head of the Computer Science Department from 2001 to 2007. Until 2001 he was a Senior Manager at the IBM TJ Watson Research Center where he led the Scalable Parallel Systems research group that was responsible for major contributions to the IBM® SP scalable parallel system and to the IBM® Blue Gene system.

Marc Snir received his PhD degree in Mathematics from the Hebrew University of Jerusalem in 1979, worked at New York University (NYU) on the NYU Ultracomputer project in 1980–1982, and was at the Hebrew University of Jerusalem in 1982–1986, before joining IBM. Marc Snir was a major contributor to the design of the Message Passing Interface. He has published numerous papers and given many presentations on computational complexity, parallel algorithms, parallel architectures, interconnection networks, parallel languages and libraries and parallel programming environments.

Marc is an Argonne Distinguished Fellow, AAAS Fellow, ACM Fellow and IEEE Fellow. He has Erdős number 2 and is a mathematical descendant of Jacques Salomon Hadamard. He recently won the IEEE Award for Excellence in Scalable Computing and the IEEE Seymour Cray Computer Engineering Award.

Rajeev Thakur is the Deputy Director of the Mathematics and Computer Science Division at Argonne National Laboratory, where he is also a Senior Computer Scientist. He is also a Senior Fellow in the Computation Institute at the University of Chicago and an Adjunct Professor in the Department of Electrical Engineering and Computer Science at Northwestern University. He received his PhD degree in Computer Engineering from Syracuse University in 1995. His research interests are in the area of high-performance computing in general and particularly in parallel programming models, runtime systems, communication libraries, and scalable parallel I/O. He is a member of the MPI Forum that defines the MPI standard. He is also coauthor of the MPICH implementation of MPI and the ROMIO implementation of MPI-IO, which have thousands of users all over the world and form the basis of commercial MPI implementations from IBM, Cray, Intel, Microsoft, and other vendors. MPICH received an R&D 100 Award in 2005. Rajeev is a coauthor of the book *Using MPI-2: Advanced Features of the Message Passing Interface* published by MIT Press, which has also been translated into Japanese. He was an Associate Editor of *IEEE Transactions on Parallel and Distributed Systems* (2003–2007) and was Technical Program Chair of the SC'12 conference.