# Enabling Efficient Multithreaded MPI Communication Through a Library-Based Implementation of MPI Endpoints

Srinivas Sridharan, James Dinan, and Dhiraj D. Kalamkar
Intel Corporation
{srinivas.sridharan, james.dinan, dhiraj.d.kalamkar}@intel.com

***Abstract*—Modern high-speed interconnection networks are designed with capabilities to support communication from multiple processor cores. The MPI endpoints extension has been proposed to ease process and thread count tradeoffs by enabling multithreaded MPI applications to efficiently drive independent network communication. In this work, we present the first implementation of the MPI endpoints interface and demonstrate the first applications running on this new interface. We use a novel library-based design that can be layered on top of any existing, production MPI implementation. Our approach uses proxy processes to isolate threads in an MPI job, eliminating threading overheads within the MPI library and allowing threads to achieve process-like communication performance. We evaluate the performance advantages of our implementation through several benchmarks and kernels. Performance results for the Lattice QCD Dslash kernel indicate that endpoints provides up to $2.9\times$ improvement in communication performance and $1.87\times$ overall performance improvement over a highly optimized hybrid MPI+OpenMP baseline on $128$ processors.**

***Categories and Subject Descriptors*** —
D.1.3 [*Concurrent Programming*]: Parallel programming,
D.4.4 [*Communications Management*]: Message sending

***General Terms*** — Design, Performance

***Keywords*** — MPI, Hybrid Parallel Programming, Endpoints

## I. Introduction

Current high-speed interconnection networks are designed with increased capabilities to support communication by multiple cores. For such networks, HPC applications and runtime systems must drive communications through multiple processor cores in order to achieve high levels of communication performance. Because of overheads incurred through thread synchronization and shared communication state, increasing the number of MPI processes per node yields the greatest improvement in communication performance [1], [2], [3], while increasing the number of cores allocated to each MPI process enables better utilization of node resources and can improve computational performance. Thus, current hybrid MPI+"X" (e.g. MPI+OpenMP) applications must make a delicate tradeoff at the time their job is launched between the number of processes per node and the number of threads, or cores, per process.

The need to balance process and thread counts according to the communication and computational performance needs of a given application, is driven in part by MPI's one-to-one mapping between MPI processes and MPI ranks. MPI ranks represent independent communication injection and extraction points within an MPI execution, which currently must be shared by all threads within an MPI process. A consequence of sharing ranks is that the MPI library must induce a message ordering on communication operations issued by threads using the shared rank. In addition to performance challenges, threads sharing an MPI rank are entangled within the MPI message matching model. Currently, programmers must utilize out-of-band methods for addressing individual threads in their communication operations, such as using tags or communicators to distinguish communication operations targeted at individual threads. In addition to introducing complexity, these approaches also inhibit the use of certain features, such as wildcard matching, and restrictions are also placed on the participation of threads in some MPI operations, such as collectives.

The MPI endpoints extension [2], [4], [5], currently under consideration by the MPI Forum, was developed to address these performance and productivity challenges to hybrid parallel programming with MPI. The endpoints extension relaxes the one-to-one mapping of ranks to processes by providing the ability to create additional ranks within a process. These ranks can be assigned to threads, enabling threads to act as MPI processes. This extension allows threads to fully participate in MPI operations, exposing greater communication parallelism to the MPI library. In addition, MPI implementations can leverage the endpoints interface to isolate threads from each other, eliminating synchronization on the communication critical path both within the MPI runtime and the interface to the communication subsystem. By isolating threads within the MPI runtime, it becomes possible for threads to provide process-like communication performance, relieving the pressure on complex processes/threads tradeoffs.

In this work, we present the first implementation of the MPI endpoints interface. Our implementation, EP-lib, is a library that is compatible with any existing MPI implementation. EP-lib uses a novel thread isolation technique, where internal proxy processes are created to isolate threads in an MPI job, providing threads with private communication interfaces. Through the use of endpoints proxies, EP-lib enables threads in the same process to drive multiple network endpoints, while eliminating thread synchronization from the critical path. By eliminating negative interference between threads, EP-lib enables threads to achieve process-like communication performance. Because EP-lib can be used with existing, highly-tuned MPI libraries it can provide application developers with immediate access to the benefits of the MPI endpoints extension, which is not anticipated to become available in MPI implementations for several years.

We demonstrate the first applications running on the new endpoints interface and explore the performance advantages of EP-lib through several benchmarks, including a Lattice Quantum Chromodynamics (QCD) Dslash application kernel [6], [7] and a Fast Fourier Transform (FFT) communication benchmark [8], [9]. We measure overheads and demonstrate that EP-lib's improvements to communication bandwidth and message rate outweigh the cost of endpoints proxies, while enabling

978-1-4799-5500-8/14 $31.00 © 2014 IEEE
DOI 10.1109/SC.2014.45

significant improvements when compared with conventional approaches to multithreaded MPI communication. Through the Lattice QCD kernel, we demonstrate that endpoints can provide greater access to the communication capabilities of the system interconnect while preserving the parallel decomposition of the application. Using EP-lib, the highly-tuned Lattice QCD kernel achieves a communication performance improvement factor of $2.9\times$ and an overall performance improvement factor of $1.87\times$ on 128 processors.

The rest of the paper is organized as follows. Section II provide background information on MPI and the proposed MPI endpoints interface. In Section III we present EP-lib, our implementation of the MPI endpoints interface as a library. Section IV presents an empirical evaluation of our implementation and explores the performance potential of the endpoints extension. In Section V we discuss design alternatives for MPI endpoints and EP-lib. Section VI briefly describes related work in this topic. Finally, we conclude with Section VII and discuss opportunities for future work.

## II. MPI and The Endpoints Extension

The Message Passing Interface (MPI) [10] is a popular library-based model for large-scale parallel programming. An MPI execution is comprised of multiple processes executing the same program in an SPMD execution, or different programs in an MPMD execution. Each process is given an integer rank that identifies it in the MPI execution, and it communicates with other processes through explicit calls to communication functions. MPI supports two-sided send and receive operations; collective operations, such as reductions and barriers; and one-sided communication operations, such as get and put.

MPI is often combined with a shared memory parallel programming model in the popular "MPI+X" hybrid parallel programming paradigm [11], [12]. This model contains two levels of parallelism – MPI provides a coarse level of parallelism and supports interprocess communication, while a shared memory or node-level parallel programming model, such as OpenMP*, provides finer-grain parallelism and shared memory.

### A. Challenges to Hybrid MPI+X Parallel Programming

In the hybrid MPI+X parallel programming paradigm, "X" is often a multithreaded programming model, such as OpenMP. During hybrid execution, the threads in this model share an MPI rank, which can introduce both programmability and performance challenges.

MPI defines a message matching semantic that is first-in, first-out (FIFO) for a given communicator and tag. When threads communicate concurrently sharing an MPI rank, the programmer must be careful to avoid or tolerate races in MPI message matching. For example, if two threads perform concurrent send or receive operations using the same communicator and the same tag, the order in which the operations are posted is nondeterministic unless the threads enforce an ordering.

Programmers can avoid these ordering challenges either by using different sets of tags for each thread or by using different communicators. However, each approach has drawbacks. When using different tags and the same communicator, it becomes challenging to use the `MPI_ANY_TAG` and `MPI_ANY_SOURCE` wildcards. In addition, only one thread per process can participate in collective operations. Using multiple communicators can sidestep some of these issues, but at the expense of partitioning threads into individual communicators.

In addition to productivity challenges, threads sharing an MPI process can encounter performance difficulties because of destructive interference within the MPI runtime. Thread safety within the MPI library is known to be expensive [13], [14]. In addition, threads that perform MPI communication operations using a shared rank can increase the MPI message matching state, resulting in increased message processing overhead [1], [2]. These overheads are at odds with the growing need for multiple cores to drive communications in order to achieve the full capabilities of modern interconnects.

### B. MPI Endpoints Extension

MPI 2.0 [15] defined the interaction between MPI and threads in terms of levels of thread safety provided by an MPI implementation. Today, hybrid parallel programming with MPI has become a standard practice for scalable parallel programming. Because of this trend, there is a growing need to go beyond thread safety and to provide mechanisms to better integrate hybrid programming with MPI.

The MPI endpoints extension has been under development by the members of the MPI Forum hybrid working group for several years. During this time, multiple interfaces have been explored [2], [4]. In this paper, we focus on the dynamic endpoints proposal that has been brought forward as a formal proposal and is under consideration for inclusion in the next major release of the MPI standard [5].

The MPI endpoints interface defines the following new function.

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

This function allows the programmer to create new MPI ranks from a process in an existing MPI communicator. The output of this function is an array of communicator handles of length `my_num_ep`, where each handle corresponds to a new local rank in the output communicator. In Figure 1, we show an example use of this new interface where a parent communicator containing three processes is used to create a new endpoints communicator that contains seven ranks. Several associations between endpoints and threads are shown to illustrate various usage models; we use "M" to denote a logical master thread that has created child threads denoted "T".

Once created, endpoints behave as MPI processes. For example, all ranks in an endpoints communicator must participate in collective operations. A consequence of this semantic is that endpoints also have MPI process progress requirements; that operations on that endpoint are required to make progress only when an MPI operation (e.g. `MPI_Test`) is performed on that endpoint. This semantic enables an MPI implementation to logically separate endpoints, treat them independently within the progress engine, and eliminate synchronization in updating their state.

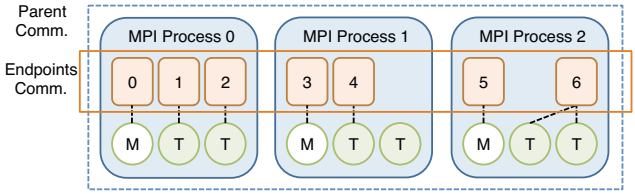



Fig. 1: Endpoints communicator creation example, showing different associations between threads (M denotes the master thread) and endpoint ranks.

```
int tl, max_threads = omp_get_max_threads();
MPI_Comm ep_comm[max_threads];
MPI_Init_thread(&argc, &argv, MULTIPLE, &tl);

#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(WORLD, nt,
        MPI_INFO_NULL, ep_comm);
  }
#pragma omp barrier
  MPI_Comm_rank(ep_comm[tn], &ep_rank);
  ... // Compute; can split work 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 acquire ranks for all OpenMP threads.

In Listing 1 we provide a more detailed example hybrid MPI+OpenMP program that utilizes the endpoints interface. In this example, the master thread calls `MPI_Comm_create_endpoints` on `MPI_COMM_WORLD` and the resulting endpoint handles are distributed to individual threads. Threads can use their endpoint communicator handle in any MPI routine, including the `MPI_Comm_rank` routine to query their rank in the endpoints communicator. Using this communicator, all threads across all MPI processes are able participate in a call to the collective `MPI_Allreduce` routine.

## III. The MPI Endpoints Library

MPI endpoints are defined to behave like individual MPI processes. We leverage this semantic to create a novel implementation of the MPI endpoints interface as a library on top of MPI. While implementing the MPI endpoints interface within an MPI implementation can reduce some overheads, our approach backs each endpoint with a private communication stack, enabling threads to perform lockless communication operations. In addition, it can be used to provide the benefits of endpoints using any existing MPI library. This, in turn, enables early exploration and rapid adoption of the endpoints interface.

Our MPI endpoints library, called EP-lib, supplies the new `MPI_Comm_create_endpoints` function, and utilizes the MPI profiling interface to intercept existing MPI calls.

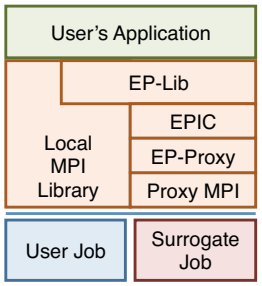



Fig. 2: EP-lib software architecture showing mapping of the application's MPI operations to the underlying system.

The software architecture of EP-lib is shown in Figure 2. Most MPI calls pass through EP-lib directly to the underlying MPI implementation. Operations that are performed on an endpoints communicator are handled by the endpoints client (EPIC) layer. In parallel with the user's job, EP-lib launches a larger set of proxy MPI processes in a surrogate MPI job; the number of processes in the surrogate job is equal to the total number of endpoints that will be created in the user job. When operations on endpoints communicators are performed by the application, the EPIC layer forwards these operations to proxy processes that perform the operation in the context of the surrogate job.

Alternative implementations that do not use proxies are possible, but they are not able to support the full breadth of MPI functionality. For example, the tag bits could be partitioned into user tag and internal endpoint ID components. However, this approach cannot support the use of source rank or tag wildcards in point-to-point communications. It also presents significant challenges to supporting collectives, as they would need to be performed using a communicator that contains fewer MPI processes than endpoints participating in the collective operation. In contrast, proxy MPI processes avoid such issues by backing each endpoint with an MPI process, ensuring that the endpoint rank can behave as a full MPI process.

In Figure 3, we show an example of endpoint communications occurring through proxy processes in the surrogate job. Because they are acting only as communication helpers, proxy processes execute in the more efficient `MPI_THREAD_SINGLE` mode, eliminating threading overheads. In addition, because proxies are responsible only for communication, they are idle during computation phases and consume minimal processor resources. For example, proxies can occupy SMT thread slots that are typically unused by HPC jobs.

### A. Optimizing Client-Proxy Interactions

Interactions between clients and proxies are the primary source of overhead in EP-lib. Because of this, we have carefully designed the EPIC layer to incur low overheads, and we have applied optimizations to reduce synchronization penalties and eliminate data copying between the client and the proxy contexts. However, some amount of overhead is unavoidable, and we measure this overhead in Section IV-A. In spite of this cost, we show that EP-lib can provide a significant improvement to communication throughput for multithreaded MPI processes.

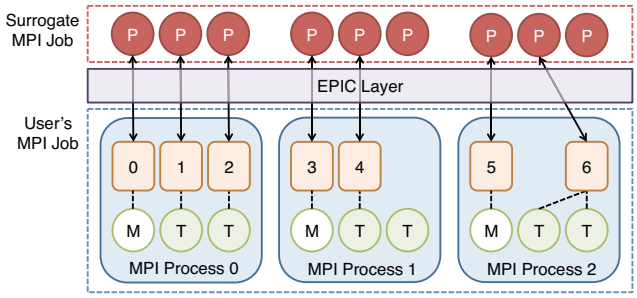



Fig. 3: EP-lib execution snapshot, showing connections between threads and endpoint proxies.

The EPIC layer uses the POSIX⋆ shared memory interface to create a memory region that is shared between the user's MPI process and its proxy processes. This shared region is used to create and share data-structures between the user's threads, called *clients*, and proxies in an efficient manner. Clients interact with proxies through command queues that are stored in the shared memory region. Command queue entries are populated by the client when issuing a request, e.g. to perform send or receive operations, or to wait for completion of a nonblocking operation. Collectives are also implemented by forwarding the operation from the client to the proxy through a command queue entry, and performing the operation on the endpoints communicator in the surrogate job. Once an operation has completed, the proxy notifies the client of the result of that operation through the same command queue entry.

When performing a communication operation, the proxy process must be able to access the user's local buffer to read data to send or write data that has been received. Any of the well-known techniques for interprocess shared memory communication can be used if a copy is required between buffers in the client and proxy processes. However, copying is undesirable because it consumes significant memory capacity and bandwidth resources. Instead, we provide an additional memory allocation function that can be used to allocate communication buffers out of the shared memory region. This allows the proxy to directly communicate through the user's buffer, eliminating the data copy between the client and proxy and significantly reducing EP-lib's overhead.

Our custom allocator can be made transparent to the application by intercepting calls to `malloc` and performing allocations out of the shared memory region. Even with such an approach, communication buffers on the stack, in the static data segment, or in Fortran common blocks will not be accessible to the proxy process. For such buffers, cross-memory attachment to map pages from the client process into the proxy's address space can provide a low-overhead solution. Otherwise, EP-lib must fall back to an interprocess data copy between client and proxy.

### B. Management of MPI Objects In the EPIC Interface

EP-lib must perform translation on communicator handles in order to distinguish endpoints communicators from conventional communicators in MPI operations. EP-lib creates endpoints communicators only in the context of the surrogate job; thus a handle to an endpoints communicator is not valid in the user's job. Because it is not possible to reserve communicator handle values, it is possible that a communicator in the user's job could use the same handle value as an endpoints communicator in the surrogate job. Therefore, special handle values must be used so that endpoints communicators can be distinguished from conventional communicators. However, replacing the endpoints communicator handle with a different handle value can also alias a communicator handle within the user's job. Therefore, we replace all communicator handles with an `EP_Comm` handle that fits within the range that can be stored in an `MPI_Comm` handle object (typically 64-bits).

When EP-lib intercepts an MPI operation on a communicator, it casts the `MPI_Comm` handle to an `EP_Comm` handle and performs a lookup using this value. The lookup results in either an `MPI_Comm` handle for client communicators, or an `EPIC_Comm` object that contains all of the information needed for EPIC to forward the given operation to the responsible proxy.

In addition to translating MPI communicator handles, some MPI objects that can be used in both client and endpoints contexts must be created in both contexts. We create both MPI info and derived datatypes objects in both contexts and EPIC maintains a directory of objects that have been created in the proxy so that operations performed on the object (e.g. freeing the object) occur in both contexts.

### C. Handling Nonblocking MPI Operations

Nonblocking MPI operations return an `MPI_Request` handle that can be used to query the status of or to complete the given operation. Nonblocking operations performed on endpoints communicators return request handles that are valid only in the context of the proxy process, whereas operations performed on a conventional communicator return request handles that are valid in the client process. Thus, EP-lib must maintain a request translation structure similar to the structure used for managing communication handles.

When a nonblocking operation is performed, the request handle is registered by EP-lib, and an `EP_Request` handle is substituted and returned to the user. When an operation is performed on a request, a lookup is performed on the given request handle to determine if it is a client or a proxy communicator and to translate the `EP_Request` into its corresponding `MPI_Request` handle. If a given request handle corresponds to an endpoints operation, EPIC creates a completion flag and status object in the shared segment and registers them with given request in the client and proxy processes. When the request completes at the proxy process, the proxy updates the flag and status object to notify the client, and frees the request.

The MPI standard progress rules indicate that communications involving a given process (with the exception of passive target RMA operations) need only make progress when the process performs MPI operations. Thus, the proxy process must check for completion of requests only when it receives test or wait commands from the client; otherwise, it remains idle. Alternatively, EP-lib supports a mode of operation where proxies poll for completion on outstanding requests to provide asynchronous progress on endpoints communications. While this mode consumes additional resources, the increase in responsiveness can provide significant performance benefits for some applications. For the results we present in this paper, we

do not enable the asynchronous progress mode. This allows us to evaluate the effectiveness of the baseline semantics provided by the MPI endpoints extension.

MPI also provides several variations of test and wait operations that complete some, all, or any of the operations in an array of requests. This array can contain a mixture of requests from both conventional and endpoints communicators. Test operations are implemented by testing individual requests in the array. However, blocking at either the client or the proxy in a wait operation with a mixture of conventional and endpoints requests would prevent either the client or proxy from making progress, potentially leading to deadlock. Thus, two approaches to waiting on mixed request arrays are possible. Any wait operation can be accomplished by repeatedly testing requests until the wait condition is satisfied. We use this simple approach in EP-Lib, but more optimized approaches are possible. For a wait-all operation, a more efficient implementation is possible by sorting the array of requests into client and proxy arrays. EPIC can issue a wait-all operation to the proxy and then perform the wait-all on the array of client requests. Once the client requests have all completed, EPIC can wait for completion notification on all of the proxy requests. For wait-any or wait-some variants on mixed request arrays, both client and proxy poll for completion in parallel. The must periodically check a shared completion flag to break out of polling when the completion semantic is satisfied in either the client or the proxy, because there is no guarantee on which requests will be completed by these operations.

### D. Creation of Endpoints Communicators

When the user creates an endpoints communicator, the communicator's ranks in the user's job are connected to proxy processes and the corresponding communicator must be created in the surrogate job. We use the following algorithm, which makes use of the MPI 3.0 `MPI_Comm_create_group` operation to avoid the need for the parent communicator to exist within the proxy job. Note that in this algorithm, it is also permitted for the parent communicator to be an endpoints communicator.

1) Perform an all-gather on the parent communicator of the `my_num_ep` argument values supplied by each process or endpoints in the parent communicator.
2) Using the `my_num_ep` values, build an MPI group in the proxy process that contains the ranks of the corresponding proxy processes in the surrogate `MPI_COMM_WORLD`. We assume here that every proxy process also has access to any needed information about the layout of the surrogate job. Typically, the same number of proxies is run on every node and the only topology information that is needed is the number of proxies per node. The result of the all-gather is no longer needed after this step and it can be freed.
3) In the `MPI_COMM_WORLD` of the surrogate job, proxy processes create the endpoints communicator. This is accomplished by calling the MPI 3.0 routine, `MPI_Comm_create_group` with the group argument generated in step 2. `MPI_Comm_create_group` differs from other MPI communicator creation routines because it is collective only on the processes that will be members of the output communicator. We require this semantic, because proxy processes that will not be members in the output communicator do not call the communicator creation routine and could be blocked performing other MPI operations.
4) Once creation of the proxy communicator has completed, EPIC registers the output `MPI_Comm` handles as handles to an endpoints communicator. The corresponding array of `EP_Comm` handles is returned to the user.

## IV. Experimental Evaluation

We conduct an experimental evaluation of EP-lib and provide the first exploration of the performance potential of the MPI endpoints extension. We first examine the overheads of the proxy approach used by EP-lib and demonstrate that overheads incurred through client-proxy coordination result in a fixed, roughly $320ns$ overhead. Next, we explore the performance impact of endpoints on bandwidth and message rate and show that, in spite of client-proxy coordination overheads, EP-lib provides significant improvements in communication performance for multithreaded MPI applications. Endpoints communication performance is on-par with performance of multiple individual processes, which represent an upper bound on the performance that can be achieved.

Through a highly tuned Lattice QCD Dslash kernel and an FFT communication benchmark, we demonstrate that endpoints can be added to existing hybrid MPI+OpenMP programs to increase communication performance without increasing the number of MPI processes, preserving the application's parallel decomposition and partitioning of node resources. For the QCD kernel on 128 processors, we trade two threads per processor for endpoints proxies, resulting in a computational slowdown of 1.24x, a communication speedup of 2.91x, and an overall total speedup of 1.87x.

Our experiments were conducted using the Intel® Endeavor HPC benchmarking cluster located in Rio Rancho, NM. Endeavor contains 300 nodes, and each node consists of two 2.7 GHz Intel® Xeon® E5-2697 processors; nodes are connected with Mellanox⋆ FDR InfiniBand⋆ network adapters in a 2-level fat tree topology. Each processor has 12 cores and each core has two threads with hyper-threading enabled. Cores have private 32KB L1 and 256 KB L2 caches, and they share a 30 MB L3 cache with all other cores in the same socket. We used the Intel® MPI Library (version 4.1.3) and Intel® C Compiler (version 14.0.2). No modifications were made to the Intel® MPI Library; EP-lib is intended to work with the tuned MPI library that is provided on a given HPC machine, without any modifications.

### A. Measurement of Overheads

In Figure 4, we show the half-round-trip latency versus message size between two processes. This experiment measures the software processing overhead required to issue a communication command to the proxy through the shared memory command queue. In addition to this processing overhead, EP-Lib also incurs memory overheads, which we discuss in Section V.

In the single-threaded (ST) case, two MPI processes communicate with each other; in the multithreaded (MT) case,

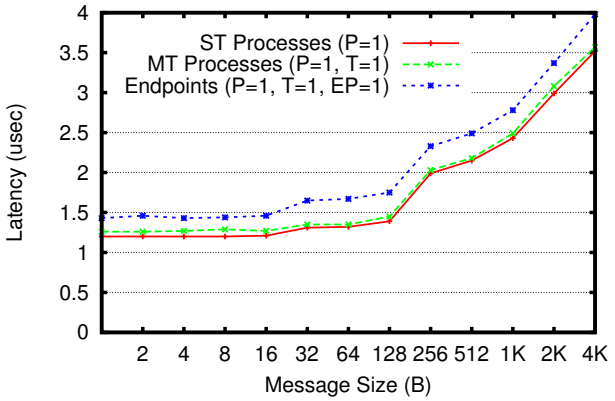



Fig. 4: Communication latency versus message size; $P$, $T$, and $EP$ represent the number of processes, threads, and endpoints per node in each configuration.

two threads in different MPI processes communicate with each other; and in the endpoints case, two threads in different MPI processes communicate with each other using an endpoints communicator with one rank per node. For all experiments, we pin threads to cores to reduce noise, and for endpoints cases, we also pin proxy processes to a hyper-threading slot adjacent to the thread that will communicate through the proxy. This allows the thread and the proxy to communicate efficiently through the shared L1 cache. We map communication buffers into the address space of the proxy process so that message data is not copied. This is accomplished in user-space by intercepting allocation operations and allocating communication buffers out of a shared segment. Alternatively, operating system functionality, such as XPMEM, can be used to map heap, stack, and data segments into the proxy's address space.

The single-threaded case achieves the best latency, as it incurs no thread synchronization. The multithreaded case shows that there is a small overhead generated by enabling thread safety within the MPI runtime system, even when communication is single-threaded. Finally, the endpoints case incurs an average additional latency of $320ns$ due to communication with the proxy process. The impact of this overhead is most severe for small messages, and becomes negligible for larger messages. We observe a fixed overhead, because communication buffers are shared between the user and proxy processes, and the cost to translate MPI object handles and issue commands to the proxy process remains fixed even as message sizes vary. If buffers cannot be shared between the user process and the proxy, data must be copied, resulting in an overhead that is proportional to message size.

### B. Internode Communication Performance

In Figure 5 we measure the bandwidth versus message size between two nodes with varying numbers of processes, threads, or endpoints per node. This experiment measures the unidirectional bandwidth that is achieved through the network endpoint at a given node in each configuration. Our benchmark is adapted from the OSU bandwidth benchmark [16], and utilizes one sender and one receiver process on separate nodes with a window of 64 nonblocking operations. We use the same pinning strategy used in Section IV-A.

Processes represent an upper bound on the throughput achievable for this benchmark. We see that adding threads leads to a significant reduction in bandwidth achieved because of negative interference within the MPI runtime system. The endpoints case uses the same configuration of processes and threads as the multithreaded case, but performs communications through endpoints. Endpoints eliminate threading interference, resulting in significant performance improvements as additional threads drive communication. In addition, endpoints enables separate message matching queues to be maintained per endpoint, whereas a multithreaded process shares message matching structures across all threads, resulting in higher message matching overheads.

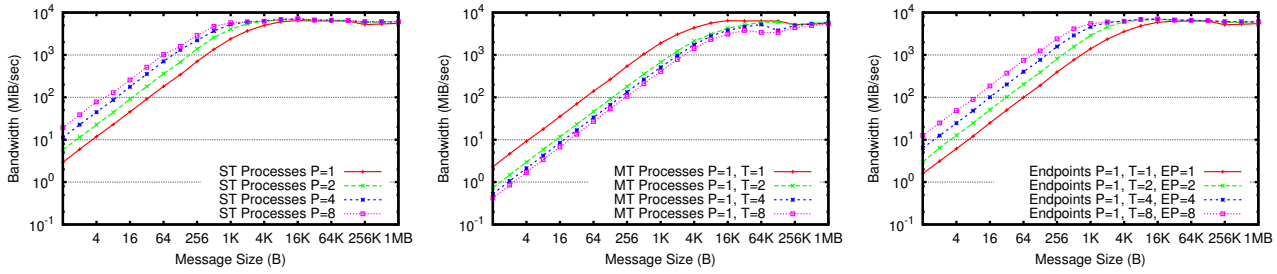
For 4kB message size, the 8 endpoints case per node case matches the bandwidth of the 8 process per node case, which is $4.38\times$ greater than the equivalent multithreaded MPI case without endpoints. The performance of the endpoints case begins to approach that of the process case as we increase the number of endpoints per node, and in either case they begin to saturate the network bandwidth sooner than the multithreaded MPI case. For larger messages, all approaches are able to saturate the network as transfers are offloaded to the InfiniBand adapter's DMA engine and processing overheads are amortized over longer communication times.

We repeat a similar experiment to measure message rate, shown in Figure 6. We again plot the message rate achieved for a node, varying the number of processes, threads, and endpoints. From this data we see that the multithreaded case results in a reduction of an order of magnitude or more for small messages, but that this is corrected when threads communicate through endpoints rather than conventional MPI communicators. Similar to the bandwidth results, for larger messages, longer transfer times amortize overheads and performance converges to a similar level across all configurations.

### C. Lattice Quantum Chromodynamics Kernel

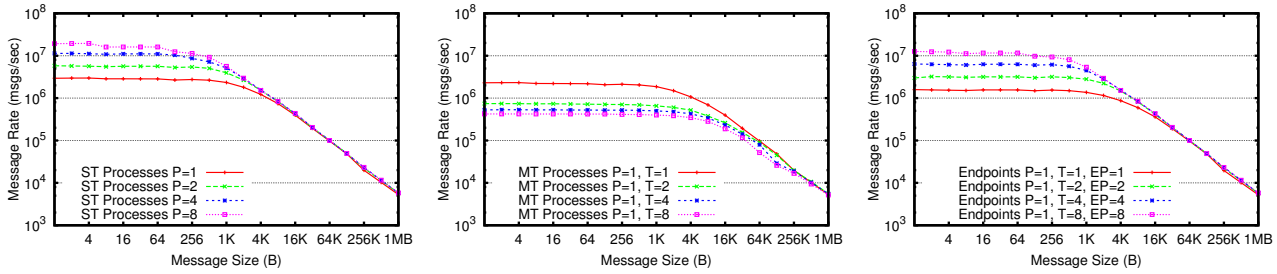
Lattice Quantum Chromodynamics (QCD) [6], [7] is used in modeling strong forces in nuclear and high energy physics applications. The Wilson Dslash operator is one of the most important and computationally intensive kernels in Lattice QCD. It describes the interaction between quarks and gluons on a 4 dimensional space-time lattice and performs a sparse matrix vector multiplication formulated as a 9-point nearest-neighbor stencil computation. The 4-dimensional lattice is equally partitioned and distributed across multiple nodes using MPI. OpenMP is used within each MPI process to divide the local lattice volume across multiple threads. An even-odd preconditioning is used, wherein a lattice is divided into even and odd sub-lattices. The computation phase operates on the internal volume of the one sub-lattice, performing a nearest-neighbor stencil operation on each grid point of that sub-lattice and storing the output in the other sub-lattice. The communication phase involves exchanging the boundary region of the sub-lattices with neighboring nodes using MPI. Communication-computation overlap is achieved by performing the computation on the local lattice volume in between posting non-blocking MPI communication operations and waiting for them to complete

The highly optimized baseline hybrid MPI+OpenMP implementation of the Wilson Dslash operator, using only the




(a) MPI Processes (b) Multithreaded MPI Processes (c) Multithreaded MPI Proc. with Endpoints

Fig. 5: Bandwidth versus message size, comparing performance of MPI processes, multithreaded MPI, and multithreaded MPI processes with endpoints. $P$, $T$, and $EP$ represent the number of processes, threads, and endpoints per node.




(a) Single-threaded MPI Processes (b) Multithreaded MPI Processes (c) Multithreaded MPI Proc. with Endpoints

Fig. 6: Message rate versus message size, comparing performance of single-threaded MPI processes, multithreaded MPI, and multithreaded MPI processes with endpoints. $P$, $T$, and $EP$ represent the number of processes, threads, and endpoints per node.

master thread to perform MPI operations, has been used in a number of past studies to showcase both shared-memory and MPI optimizations, and was shown to achieve best-in-class performance on systems using Intel® Xeon® and Intel® Xeon Phi™ processors [6], [7]. We extended this code to perform multithreaded MPI communication and demonstrate the benefits of the MPI endpoints interface; this new code is shown in Listings 2 and 3.

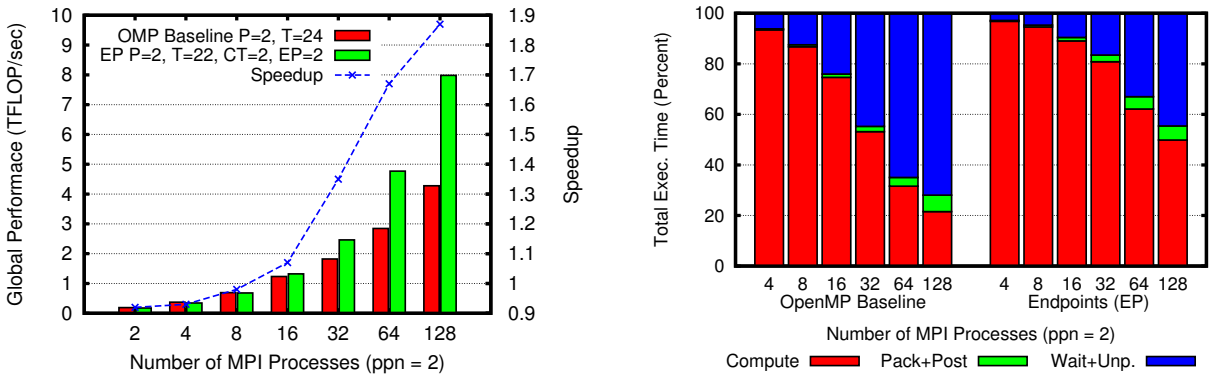
Listing 2 presents the pseudocode representing the initialization and the main loop of the kernel. During initialization, we first determine the target ranks (at most eight neighboring ranks; 4D decomposition). Next we assign each of the eight boundary regions to different threads (at most eight threads will be required). This enables different threads to post communication operations and wait on them in parallel. Additionally for the endpoints version (highlighted code), we map the target rank to the corresponding endpoint rank using a simple mapping scheme.

Listing 3 presents the pseudocode for the pack-and-send operation. In the multithreaded MPI case, all the threads are used perform the packing operation and wait on a barrier. However, only the communication threads (CT) are used to drive the actual communication. In the endpoints case, instead of determining the neighboring rank in the `MPI_COMM_WORLD` communicator, threads need to identify the remote endpoint rank corresponding to the respective boundary region in the neighboring sub-domain. For instance, the endpoint rank responsible for the $T$ forward direction will need to communicate with the endpoint rank responsible for the $T$ reverse direction in the neighboring sub-domain along the forward direction and vice-versa. Except for these changes, the rest of the source code, including the non-blocking send/recv and wait calls, remain unchanged. The code for wait-and-unpack is similar to pack-and-send, and is omitted because of space.

Figure 7 presents a strong scaling study of the Dslash kernel using a $32 \times 32 \times 32 \times 256$ grid. The x-axis shows the number of MPI processes, one per socket and 2 per node. The y-axis shows the aggregate single-precision Dslash performance in TFLOPs. The grid is split along the last dimension ($T$ dimension) up to 64 processes. It is split across the last two dimensions ($Z$ and $T$ dimension) for 128 processes since the sub-lattices along the $Z$ dimension become too small to split the grid further. We could potentially use four threads (and four endpoints) for the 128 processes case; however, the loss in performance from fewer computation threads and the increase in load-imbalance from barrier overheads outweigh the communication benefits provided by the additional endpoints.

The baseline optimized OpenMP version uses all 12 cores, or 24 threads, for parallelizing the computation and the master thread performs the communication. The endpoints version (EP) uses two endpoints per MPI rank, one for communicating with the forward neighbor and another for communicating with the backward neighbor. The two endpoints require two proxy processes and occupy both hyperthread slots of a single core. The remaining 11 cores (i.e. 22 threads) are available to the application. Further, two CTs drive communication in the boundary exchange phase, each through its own endpoint.

Figure 7b and Table I present the breakdown of percentage time spent and absolute time spent (in msec) respectively




(a) Kernel Performance

(b) Timing Breakdown

Fig. 7: Lattice QCD Dslash kernel strong scaling study. Optimized OpenMP (OMP) baseline and multithreaded processes with endpoints (EP) configurations are shown with $P$ processes per node, $T$ threads per process, and $EP$ endpoints per process.

```
#define MAX_DIM 4 // Maximum dimensions
#define MAX_DIR 2 // Directions per dimension
#define NUM_EP  2 // Number of endpoints
#define MAX_NBR MAX_DIM*MAX_DIR
int nb_rank[MAX_NBR]; // Target ranks
int ep_rank[MAX_NBR]; // Target endpoints
int cthd_id[MAX_NBR]; // Comm. thread ids

MPI_Comm ep_comm[MAX_NBR];
MPI_Comm_create_endpoints(MPI_COMM_WORLD,
       NUM_EP, MPI_INFO_NULL, ep_comm);

// Determine target ranks and comm. thread IDs
for (dim=0; dim < MAX_DIM; dim++) {
  for (dir=0; dir < MAX_DIR; dir++) {
    int nbr_idx = MAX_DIR*dim+dir;
    nb_rank[nbr_idx] = calc_neighbor_rank();
    // Thread id responsible for this neighbor
    cthd_id[nbr_idx] = nbr_idx;
    // Map target rank to corr. endpoint
    ep_rank[nbr_idx] =
      nb_rank[nbr_idx]*NUM_EP +
      ((MAX_DIR*dim+(1-dir))%NUM_EP);
  }
}

for (iter = 0; iter < MAX_ITER; iter++) {
#pragma omp parallel {
    1. Pack and send data to neighbor ranks
    2. Compute internal volume (comm. overlap)
    3. Receive data from neighbors and unpack
  }
}
```

Listing 2: Pseudocode of Lattice QCD Dslash Kernel; endpoints code is highlighted.

```
for (dim=0; dim < MAX_DIM; dim++) {
  for (dir=0; dir < MAX_DIR; dir++) {
    // Pack data into buffers in parallel
    // Overlap with communication for dim-1
  }
#pragma omp barrier
  // Comm-Comm overlap for two directions
  for (dir=0; dir < MAX_DIR; dir++) {
    int tn = omp_get_thread_num();
    int nbr_idx = MAX_DIR*dim+dir;
    if (tn == cthd_id[nbr_idx]) {
#ifdef USE_ENDPOINTS
      MPI_Comm comm = ep_comm[nbr_idx%NUM_EP];
      int target = ep_rank[nbr_idx];
#else
      MPI_Comm comm = MPI_COMM_WORLD;
      int target = nb_rank[nbr_idx];
#endif
      MPI_Isend(target, comm, ...);
      MPI_Irecv(target, comm, ...);
    }
  }
}
```

Listing 3: Pseudocode of Lattice QCD Dslash Kernel pack-and-send step; endpoints code is highlighted.

for the OpenMP baseline and endpoints (MPI-EP $P = 2$, $T = 22$, $EP = 2$) versions. The *Pack+Post* operation indicates the time taken to pack the communication buffer and to post the non-blocking send and receive operations. The *Wait+Unpack* operation represents the time taken to wait for the communication operations to complete and to unpack the buffers. Table I additionally presents the slow-down for the computation phase, speedup for the communication phase, and overall application speedup for the endpoints version over the baseline version. As mentioned earlier, the endpoints configuration dedicates two cores for endpoints proxy resulting in a slow-down for the computation portion of the kernel. However, as we increase the number of MPI processes, the execution time is dominated by communication. Therefore, the improvement in communication performance (up to $3.34x$ for 32 nodes) offsets the slowdown in computational performance.

In Figure 8, we compare the performance of the optimized OpenMP baseline version against a version that performs multithreaded communication. The multithreaded MPI version (MT) uses all 12 cores for computation but uses dedicated communication threads (CT) for driving communication in

TABLE I: Lattice QCD Dslash timing breakdown in msec. The endpoints configuration re-purposes two cores to act as endpoints proxies; the resulting computational slowdown, communication speedup, and overall speedup is shown.

| Num. | OpenMP Baseline | | | | Endpoints | | | | Compute | Comm. | Overall |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Proc. | Compute | Pack+Post | Wait+Unpck | Total | Compute | Pack+Post | Wait+Unpck | Total | Slowdown | Speedup | Speedup |
| 2 | 11140.80 | 25.30 | 375.57 | 11541.67 | 12432.00 | 22.63 | 164.70 | 12619.33 | 1.12 | 2.14 | 0.92 |
| 4 | 5571.60 | 24.80 | 366.72 | 5963.11 | 6206.80 | 22.67 | 181.57 | 6411.04 | 1.11 | 1.92 | 0.93 |
| 8 | 2766.00 | 24.27 | 397.76 | 3188.03 | 3053.60 | 22.78 | 151.77 | 3228.15 | 1.10 | 2.42 | 0.98 |
| 16 | 1329.20 | 22.98 | 428.19 | 1780.37 | 1484.00 | 22.90 | 159.17 | 1666.07 | 1.12 | 2.48 | 1.07 |
| 32 | 647.20 | 25.44 | 544.84 | 1217.48 | 719.20 | 23.62 | 146.99 | 889.81 | 1.11 | 3.34 | 1.35 |
| 64 | 244.40 | 26.65 | 502.33 | 773.38 | 290.00 | 22.84 | 154.07 | 466.91 | 1.19 | 2.99 | 1.67 |
| 128 | 111.20 | 33.68 | 371.38 | 516.26 | 137.60 | 15.35 | 123.67 | 276.62 | 1.24 | 2.91 | 1.87 |

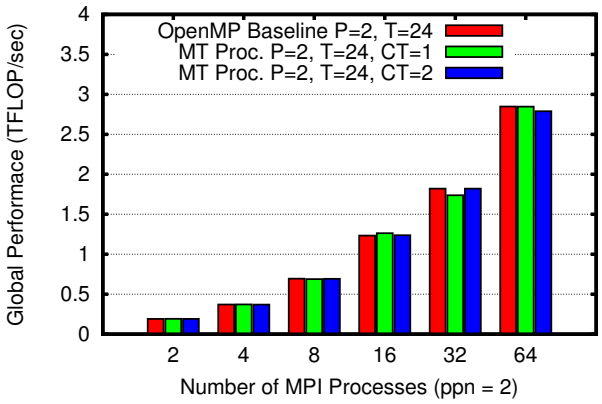



Fig. 8: Lattice QCD Dslash kernel strong scaling study: Baseline optimized OpenMP vs. multithreaded MPI comparison.

the boundary exchange phase. Figure 8 shows that the multithreaded MPI version with a single CT performs similar to the baseline version, but incurs some slowdown from initializing MPI in multithreaded mode. Adding a second CT further decreases performance as threads generate negative interference within the MPI library.

### *D. Fast Fourier Transform Benchmark*

Fast Fourier Transform (FFT) calculations contain a communication intensive data exchange step, which is typically all-to-all. We create a benchmark that models the communication performed by the highly optimized low-communication 1D FFT kernel of Tang et al. [8], [9]. We model only the communication performed in this algorithm to isolate the impact of endpoints; overall speedup to an FFT computation would be scaled proportional to the Amdahl fraction occupied by communication. Further, some FFT implementations are able to incorporate overlap of communication with computation, which we do not model in our benchmark. In the baseline version of this benchmark, when run on $P$ MPI processes, each process sends a message to $P-1$ other processes. This is analogous to a multithreaded execution where MPI executed in `MPI_THREAD_FUNNELED` mode and the OpenMP master thread performs all communications.

In a multithreaded execution, where all threads drive communication, the threads are arranged into groups. Given $N$ threads per MPI process and $P$ MPI processes. $N$ groups are created, with each group having $P/N$ MPI tasks; groups are numbered $0 \ldots N-1$. Thread $i$ in a given MPI process is responsible for communicating with the $P/N$ tasks in group $i$. When a multithreaded execution is used, threads communicate using a shared rank. When endpoints are used, $N$ endpoints are created per process and threads communicate through the respective endpoint. Thus, in order to communicate the same volume of data per process, each of $N$ threads or endpoints communicates with $P/N$ MPI tasks, while each process communicates with $P$ other tasks.

The bandwidth achieved for single-threaded processes, the OpenMP master thread model, multithreaded process, and multithreaded process with endpoints cases is shown in Figure 9. We use the same pinning strategy used in Section IV-A. We see that the multithreaded case incurs significant overheads for small to medium-sized messages, and that an OpenMP master thread communication model would be more efficient. When endpoints are added to the multithreaded process case, the bandwidth over the OpenMP master case increases by a factor of more than 2.8x for messages up to 128B, and more than 2x for messages up to 4kB. This increase is achieved because EP-lib does not encounter thread synchronization while issuing communication operations. In addition, endpoints enables separate message matching queues to be maintained per endpoint, whereas a multithreaded process shares message matching structures across all threads, resulting in higher message processing overheads. Single-threaded processes represent an upper bound on throughput, and an equal number of endpoints as processes results in similar performance. Larger messages amortize these overheads and both the multithreaded and endpoints configurations perform better than the OpenMP master configuration because both can issue communication operations at a higher rate.

A performance anomaly occurs at 256kB in the OpenMP master configuration because the larger messages generated by the OpenMP master thread cross from the MPI implementation's eager protocol to its rendezvous protocol at this message size. Because the data transfer is broken into smaller chunks in the other two cases, this protocol transition does not occur until a larger volume of data per node is transferred.

## V. Discussion

EP-lib takes a library approach to implementing MPI endpoints on top of an existing MPI library, without making any modifications to the underlying MPI library. This approach has several advantages: it provides immediate access to the benefits of endpoints on any system; it can utilize existing highly tuned MPI implementations; proxies enable threads in the same process to drive multiple network endpoints, without requiring support for this feature from the low-level

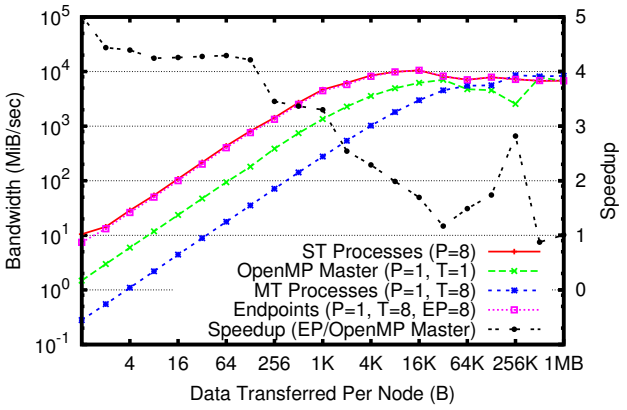



Fig. 9: FFT communication benchmark showing bandwidth versus message size on 32 nodes. $P$, $T$, and $EP$ represent the number of processes, threads, and endpoints per node.

communication layer; and it enables an implementation where threads incur no synchronization when issuing communication operations.

While these advantages are significant, additional exploration into the benefits of integrating the endpoints functionality within an existing MPI library is warranted. EP-lib utilizes a novel approach to thread isolation, where surrogate proxy processes are used as communication agents that provide the communication endpoints which are mapped to MPI endpoints in the client application. While this approach enables threads to perform lockless communication, it also consumes extra resources by duplicating MPI structures and objects, creating additional network connections, mapping shared structures that are used for client-proxy interactions, and consuming thread/core resources by the proxy.

Fully integrated implementations of endpoints within the MPI library are expected to eliminate some of these overheads, and provide finer control over others, such as network resources. Additionally, integration of endpoints within the MPI runtime provides opportunities to reduce resource consumption when multiple threads are engaged in the progress engine on a mixture of conventional and endpoints communication operations.

Many networks do not provide a mechanism that can be used to isolate threads in the low-level networking layer. On such networks, a native MPI implementation can apply our proxy-based method of thread isolation to create low-level proxy processes managed from within the MPI implementation. These low-level proxies would provide independent interfaces to the network, eliminating synchronization overheads in the networking layer. Rather than acting as MPI-level agents, such internal proxies can be treated as low-level communication agents, further reducing memory and latency overheads, while still providing the throughput advantages of EP-Lib.

One of the most significant benefits from integration of endpoints with the MPI communication subsystem is the ability to dynamically create resources to support additional MPI endpoints and to control their mapping to network endpoints. Because EP-lib is implemented on top of a conventional MPI implementation that does not support endpoints, a given proxy is not able to support more than one MPI endpoint at a time in the same communicator, because of MPI's 1-to-1 mapping between ranks and processes. In order to achieve an implementation where MPI endpoints share network endpoints, the MPI endpoints interface must be implemented inside of MPI.

### A. Portability EP-Lib

EP-lib is implemented entirely on top of a standard MPI implementation, and can be used with any MPI 3.0 compliant MPI library. EP-lib intercepts MPI operations performed by the application using library interposition and interfaces with the MPI library through the standard MPI profiling interface. Support for the new `MPI_Comm_create_endpoints` function is added by building on top of existing MPI routines.

While these qualities make EP-lib agnostic of the underlying MPI implementation, EP-lib's MPI object handle translator must be configured for the specific MPI implementation that will be used. As discussed in Section III-B, EP-lib manipulates communicator, request, and other object handles to enable translation of the MPI object handle to an object in the user or surrogate job, depending on the context in which a given operation will be performed. In order to perform this translation, EP-lib must be aware of the representation that the MPI library uses for these handles, often an integer or pointer type. EP-lib substitutes its own integer handle that is cast to the MPI library's handle type and returned to the user. EP-lib translates this handle back to a native MPI object handle before any operations are passed to the MPI library. This method of handle substitution is valid, because handle types are opaque in the MPI standard and it does not impact the user's ability to perform assignment and comparison on MPI object handles. While EP-lib does not require the same MPI library to be used by both the client and surrogate jobs, the EP-lib handle translator would need to be extended to support differing handle types if user and surrogate MPI libraries do not use the same underlying MPI handle representations.

### B. Supporting Additional MPI Functionality in EP-lib

In this paper, we have focused on communication operations that are performed in the context of an MPI communicator, i.e. two-sided and collective communication. This focus results from the focus of the proposed endpoints interface, as an extension to the MPI communicator interface.

One area of the MPI interface that we have not discussed in this paper is the MPI remote memory access (RMA) interface. The RMA interface operates on MPI windows, which are created from an MPI communicator. In EP-lib, the MPI endpoints communicator exists only within the proxy process. Thus, the window must also be created in the proxy's context. MPI library calls can be forwarded to the proxy using the same mechanisms that we have described. Direct load/store accesses to window data by the client process may present challenges, because these operations are not visible to the MPI or EP-lib libraries. In order to support such accesses, a buffer containing the window data must be visible to the client process. The most efficient solution is to map window buffers into the address spaces of both the client and proxy processes, either through cross-processes memory attachment or by intercepting calls to the `MPI_Alloc_mem` or `MPI_Win_allocate` routines

that allocate window buffers. If neither approach can be used for the given buffer, EP-lib can provide only the "separate" memory model for the given window and maintain two copies of the window buffer, one in the client and one in the proxy. Supporting only the separate memory model is valid for all versions of the MPI standard thus far, and it enables software-maintained consistency across noncoherent window copies.

### C. Supporting Multithreaded Endpoints in EP-lib

We have focused on a usage model where a single application thread drives each endpoint. This provides the best performance; however, it limits the possible usages of EP-lib. Supporting multiple threads sharing an endpoint could require proxies themselves to be multithreaded. Otherwise, a blocking operation in one thread, which also causes its proxy to block, could prevent another thread from making progress and lead to deadlock.

To work around this issue, blocking MPI operations, e.g. point-to-point and collective communication operations, can be converted into their nonblocking counterparts in the proxy. This avoids blocking in the proxy, enabling it to service requests by other threads while it polls the nonblocking operation for completion. Once the nonblocking operation completes, the proxy can notify the client that their blocking operation has completed. Several MPI operations do not have nonblocking counterparts, notably several communicator management and the dynamic process routines. For routines where proxy blocking is unavoidable, the proxy process could create a thread to perform the blocking operation. While this would require proxies to incur additional overhead from the `MPI_THREAD_MULTIPLE` mode of operation, the application would still obtain the performance benefits from driving multiple independent network endpoints.

## VI. Related Work

Multiple studies have demonstrated that communication performance can be improved when multiple MPI processes are used per node [1], [2], [3] and when additional threads are used to progress communication [17]. Indeed, performing experimentation to balance the number of processes with the number of threads or cores per process has become common practice for users of hybrid MPI+OpenMP applications on HPC systems.

While endpoints provides one approach to easing this tradeoff and increasing communication performance, a variety of other approaches have been explored. Notably, IBM Blue Gene® systems provide an implicitly multithreaded MPI runtime system that can internally parallelize MPI communication operations across multiple cores using communication helper threads [18], [19]. This approach is effective at automatically engaging multiple cores in an application's communication without the need for any modifications to the code. However, it places the burden of extracting and managing communication parallelism on the MPI implementation.

An alternative approach, provided by MVAPICH opens multiple network endpoints within the MPI runtime system, enabling a multithreaded process to implicitly drive multiple network endpoints through a single, shared MPI rank [20], [21]. While MPI endpoints requires changes to the application, this approach can be used with existing, unmodified applications. However, communication performance and usability are still inhibited because threads share a single MPI rank. Because MPI message matching order is FIFO, messages from multiple threads must be serialized as they leave the sender; messages are also serialized as they arrive at the receiver, before they can be consumed by the receiver process' threads. The MPI endpoints extension adds ranks to enable independent message streams and eliminate this serialization point that exists in the current multithreaded MPI model. In comparison with implicit approaches, endpoints also encourages the application to explicitly parallelize communications, eliminating packing operations, allowing the user to leverage data locality, and generating greater amounts of communication concurrency.

Several MPI implementations have been created, where MPI processes are implemented as threads, and detailed analyses have been conducted to investigate and address thread safety challenges that arise in such implementations [22]. Fine-Grain MPI (FG-MPI) [23] implements MPI processes as lightweight coroutines that are spawned by the parent MPI process and run on top of user-level threads. FG-MPI generates fine-grain ranks that are assigned to each co-routine. FG-ranks are similar to endpoints; however, they are created once at every host process during MPI initialization. The FG-MPI runtime system represents one of the first known implementation of an endpoints-like extension to an existing MPI runtime, and FG-MPI has achieved scalability of over 100 million MPI processes on a 540 node system with 12 cores and 16,000 fine-grain ranks (i.e. endpoints) per node [24]. The MultiProcessor Communications environment (MPC) [25] implements MPI processes as user-level threads through process virtualization. MPC employs a sophisticated scheduler that is integrated with the MPI runtime system, enabling efficient multiplexing and scheduling of MPC processes across cores. By implementing processes as threads, MPC can provide better management of node-level resources and efficient intranode communication. MPC has recently been extended to support OpenMP threading [26].

## VII. Conclusions

Today's hybrid MPI+"X" applications must strike a delicate balance between the number of processes per node and the number of threads per process. While threads can enable efficient computation management and data sharing, processes enable higher levels of communication performance. The MPI endpoints extension provides a new mechanism to enable multiple threads and their cores to drive independent MPI communication endpoints, easing thread/process tradeoffs. This capability is especially important as HPC systems increasingly require multiple cores to engage in communication in order to reach peak throughput for the interconnect.

We implement the MPI endpoints extension as a library, EP-lib, that provides a novel implementation of the endpoints extension that can be used with any existing MPI implementation. EP-lib utilizes proxy processes in a surrogate MPI job to act as MPI endpoints. Such proxies can take advantage of SMT thread slots that are typically under-utilized in HPC workloads. While not demonstrated in this paper, EP-lib is also effective on accelerator-based systems, such as those that use the Intel® Xeon Phi™ coprocessor. On such systems, under-utilized host processor cores can be used to accelerate communication through the MPI endpoints interface.

Through experimentation on an InfiniBand cluster – a popular HPC computing platform – we have demonstrated that endpoints provides significant performance advantages over existing approaches to multithreaded MPI communication. EP-lib incurs a modest overhead, which is amortized by the increased communication throughput that is achieved when multiple endpoints are utilized in parallel. We have demonstrated the first applications running on the MPI endpoints interface, and analyzed the performance advantages of endpoints through these benchmarks, including an FFT communication benchmark and a Lattice QCD kernel. Significant performance improvements were observed in all cases, especially for QCD, which achieved an overall speedup of more than $87\%$ when run on 128 processors.

## VIII. Acknowledgments

The authors would like to thank the Intel CRT-DC team that operates the Endeavor cluster for their outstanding support and assistance. The authors would also like to that Bharat Kaul and Naveen Mellempudi for useful comments and feedback.



## References

[1] B. W. Barrett, S. D. Hammond, R. Brightwell, and K. S. Hemmert, "The impact of hybrid-core processors on MPI message rate," in *Proc. 20th European MPI Users' Group Meeting*, ser. EuroMPI '13. ACM, 2013.

[2] J. Dinan, R. E. Grant, P. Balaji, D. Goodell, D. Miller, M. Snir, , and R. Thakur, "Enabling communication concurrency through flexible MPI endpoints," *Submitted to Intl. J. High Performance Computing Applications (IJHPCA)*, Dec. 2013.

[3] G. Dózsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur, "Enabling concurrent multithreaded MPI communication on multicore petascale systems," in *Recent Advances in the Message Passing Interface*, ser. Lecture Notes in Computer Science, R. Keller, E. Gabriel, M. Resch, and J. Dongarra, Eds. Springer Berlin Heidelberg, 2010, vol. 6305, pp. 11–20.

[4] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur, "Enabling MPI interoperability through flexible communication endpoints," in *Proc. Recent Adv. in MPI - 20th European MPI Users' Group Meeting*, ser. EuroMPI '13, September 2013.

[5] MPI Forum Hybrid Working Group, "MPI endpoints proposal," Online: https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/380.

[6] M. Smelyanskiy, K. Vaidyanathan, J. Choi, B. Joó, J. Chhugani, M. A. Clark, and P. Dubey, "High-performance Lattice QCD for multi-core based parallel systems using a cache-friendly hybrid threaded-MPI approach," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 69:1–69:11.

[7] B. Joó, D. D. Kalamkar, K. Vaidyanathan, M. Smelyanskiy, K. Pamnany, V. Lee, P. Dubey, and I. Watson, William, "Lattice QCD on Intel® Xeon Phi™ Coprocessors," in *Intl. Supercomputing Conf.*, ser. ISC 13, 2013.

[8] P. T. P. Tang, J. Park, D. Kim, and V. Petrov, "A framework for low-communication 1-D FFT," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 42:1–42:12.

[9] J. Park, G. Bikshandi, K. Vaidyanathan, P. T. P. Tang, P. Dubey, and D. Kim, "Tera-scale 1D FFT with low-communication algorithm and Intel Xeon Phi Coprocessors," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA: ACM, 2013, pp. 34:1–34:12.

[10] MPI Forum, "MPI: A message-passing interface standard version 3.0," University of Tennessee, Knoxville, Tech. Rep., Oct. 2012.

[11] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes," in *Proc. 17th Euromicro Intl. Conf. on Parallel, Distributed and Network-based Processing*, ser. PDP '09, Feb 2009, pp. 427–436.

[12] L. Smith and M. Bull, "Development of mixed mode MPI / OpenMP applications," *J. Scientific Programming*, vol. 9, no. 2,3, pp. 83–98, Aug. 2001.

[13] R. Thakur and W. Gropp, "Test suite for evaluating performance of MPI implementations that support MPI_THREAD_MULTIPLE," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, F. Cappello, T. Herault, and J. Dongarra, Eds. Springer Berlin Heidelberg, 2007, vol. 4757, pp. 46–55.

[14] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "Toward efficient support for multithreaded MPI communication," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, A. Lastovetsky, T. Kechadi, and J. Dongarra, Eds. Springer Berlin Heidelberg, 2008, vol. 5205, pp. 120–129.

[15] MPI Forum, "MPI-2: Extensions to the message-passing interface," University of Tennessee, Knoxville, Tech. Rep., 1996.

[16] Network-Based Computing Laboratory, Dept. Computer Science and Eng, The Ohio State University, "OSU Micro-Benchmarks," Online: http://mvapich.cse.ohio-state.edu/benchmarks/.

[17] M. Wittmann, G. Hager, T. Zeiser, and G. Wellein, "Asynchronous MPI for the masses," *CoRR*, vol. abs/1302.4280, 2013.

[18] S. Kumar, A. Mamidala, D. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burrow, "PAMI: A parallel active message interface for the Blue Gene/Q supercomputer," in *Proc. 26th Intl. Parallel Distributed Processing Symposium*, ser. IPDPS '12, May 2012, pp. 763–773.

[19] G. Tanase, G. Almasi, H. Xue, and C. Archer, "Network endpoints for clusters of SMPs," in *Proc. 24th Intl. Symp. on Computer Architecture and High Performance Computing*, ser. SBAC-PAD '12, Oct 2012, pp. 27–34.

[20] M. Luo, J. Jose, S. Sur, and D. Panda, "Multi-threaded UPC runtime with network endpoints: Design alternatives and evaluation on multi-core architectures," in *Proc. 18th Intl. Conf. on High Performance Computing*, ser. HiPC '11, Dec 2011, pp. 1–10.

[21] M. Luo, X. Lu, K. Hamidouche, K. Kandalla, and D. K. Panda, "Initial study of multi-endpoint runtime for MPI+OpenMP hybrid programming model on multi-core systems," in *Proc. 19th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, ser. PPoPP '14, 2014, pp. 395–396.

[22] G. Saxena, "Thread safety for hybrid programming in the thread-as-rank model," Ph.D. dissertation, 2013.

[23] H. Kamal and A. Wagner, "FG-MPI: Fine-grain MPI for multicore and clusters," in *11th IEEE Intl. Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC)*, 2010, pp. 1–8.

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

[25] M. Pérache, H. Jourdren, and R. Namyst, "MPC: A unified parallel runtime for clusters of NUMA machines," in *Proc. 14th Intl. Euro-Par Conference on Parallel Processing*, ser. Euro-Par '08, 2008.

[26] P. Carribault, M. Prache, and H. Jourdren, "Enabling low-overhead hybrid MPI/OpenMP parallelism with MPC," in *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, ser. Lecture Notes in Computer Science, M. Sato, T. Hanawa, M. S. Mller, B. M. Chapman, and B. R. de Supinski, Eds. Springer Berlin Heidelberg, 2010, vol. 6132, pp. 1–14.