Space Performance Tradeoffs in Compressing MPI Group Data Structures

Sameer Kumar
IBM India Research Lab.
Manyatha Embassy Business Park
Bangalore, India
sameer_km@in.ibm.com

Philip Heidelberger
IBM TJ Watson Research Center
1101 Kitchawan Road,
Yorktown Heights, NY
philiph@us.ibm.com

Craig Stunkel
IBM TJ Watson Research Center
1101 Kitchawan Road,
Yorktown Heights, NY
stunkel@us.ibm.com

ABSTRACT

MPI is a popular programming paradigm on parallel machines today. MPI libraries sometimes use `O(N)` data structures to implement MPI functionality. The IBM Blue Gene/Q machine has 16 GB memory per node. If each node runs 32 MPI processes, only 512 MB is available per process, requiring the MPI library to be space efficient. This scenario will become severe in a future Exascale machine with tens of millions of cores and MPI endpoints. We explore techniques to compress the dense `O(N)` mapping data structures that map the logical process ID to the global rank. Our techniques minimize topological communicator mapping state by replacing table lookups with a mapping function. We also explore caching schemes with performance results to optimize overheads of the mapping functions for recent translations in multiple MPI micro-benchmarks, and the 3D FFT and Algebraic Multi Grid application benchmarks.

Keywords

MPI, MPI collective communication, MPI 3.0, MPI communicators and MPI_Comm_split.

1. INTRODUCTION

In addition to achieving low latency and high throughput, the MPI library must also be efficient in terms of space usage. With current approaches in the design of MPI libraries, on an Exascale machine the libraries themselves may consume a significant portion of system memory available to applications. The Blue Gene/Q (BG/Q) Sequoia machine has a maximum of 6.3 million processes in its largest configuration. The BG/Q machine also has limited memory of 256MB available per hardware thread. A recent document from DOE estimates 16 million cores for a homogeneous Exascale system [8, 14], where applications may create tens of millions of MPI endpoints.

EuroMPI '16, September 25 - 28, 2016, Edinburgh, United Kingdom
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4234-6/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2966884.2966911.

Even in the presence of accelerators, where the total number of the host nodes is expected to be smaller, application algorithms may require threads in accelerators to be endpoints of communication operations. Such a feature will need a very large number of communication endpoints in Exascale machines. Even though the concept of communication endpoints is not yet part of the MPI 3.x standard, both point-to-point and collective communication between endpoints has been explored in the MPI Forum hybrid working group.

In the presence of a large number of endpoints on Exascale machines, both `O(N)` space data-structures and algorithms with time complexity of `O(N)` are undesirable and un-scalable [7,11]. Here N is the number of endpoints. On networks that require connection setup between endpoints, the connection data-structure is an example of an `O(N)` data structure. For example, the InfiniBand Reliably Connected (RC) mode requires `O(N)` queue pairs per endpoint to be set up to connect all endpoints with each other. A solution commonly in use today is lazy connection allocation to limit the connection space to the number of active communication endpoints. The new Dynamic Connection Transport (DCT), where a queue-pair can be reused across multiple destination endpoints with minimal connection setup and teardown penalties, can limit the number of queue pairs to a constant in DCT mode vs. `O(N)` in RC mode. Thus, increasing the scalability of messaging libraries to a large number of endpoints.

Even MPI library calls that require `O(N)` memory to be passed in input parameters may not scale to Exascale machines. In MPI 3.0, we have a new scalable distributed graph create function where input parameters and the internal state of the graph is distributed on all the processes in the communicator. In addition, the neighborhood collective extensions in MPI 3.0 provide scalable interfaces for MPI_Alltoallv and MPI_Allgatherv. Here, only bytes and offsets for the active communication neighborhood are passed to the MPI call, eliminating the use of `O(N)` data structures when each process communicates with only a limited number of the other processes.

An example of a dense O(N) structure in MPICH [2,3] libraries is the communicator logical process ID (LPID) to the global rank mapping table that is also called the Virtual Channel Reference (VCR) table. Such data structures are also common in MPI libraries on other architectures [7, 16, 17]. This mapping data structure is typically used to optimize MPI group operations such as MPI_Group_compare and MPI_Group_union. This dense table is also used to convert the MPI communicator rank to the torus coordinates in the BG/Q MPI library. This dense mapping data-structure must be built for each communicator on creation. Past work [7] showed that eliminating replication of the VCR table in MPI_Comm_dup call resulted in significant space optimization on IBM Blue Gene/P [9]. However, the space scalability problem persists if new communicators are created via MPI_Comm_create or MPI_Comm_split, where the VCR table in the sub-communicator is different from the parent communicator. For example, an application that has 2D near neighbor communication may split MPI_COMM_WORLD into row and column communicators. Similarly, linear algebra applications such as High Performance Linpack [15] execute communication operations on row and column communicators. These applications create multiple topological communicators and then execute communication operations on those sub-communicators. In the case of 2D row and column communicators, the LPID table requires O(√N) storage. But, with higher dimensional Cartesian communicators the space usage will increase significantly.

In this paper, we explore algorithms to compress the O(N) VCR table in the MPICH communicators. This compression is accomplished by replacing the dense mapping table lookup by calls to a computational routine for geometrically defined communicators. Since the overhead to compute the mapping function occurs in the critical path of MPI sends and receives, we explore caching of recent translations as a mechanism to optimize its overhead.

We explore this compression scheme on the IBM BG/Q machine [1]. Each BG/Q node has 16 embedded PowerPC processor cores available to the application, 64 hardware simultaneous multi-threading (SMT) threads, and a total of 16GB of memory. Applications often run with 32 MPI processes per node, so that each MPI process only has 512MB available. We show that on large node counts the space overhead of the LPID dense mapping table can be significantly reduced by our techniques. We present performance results on several different MPI micro-benchmarks that characterize the space savings achieved and the performance overheads of the compression scheme on BG/Q. We show the performance overhead is in most cases is minimal (within 1%) even with very small messages and this overhead decreases as either the message size increases or the number of partners in an exchange increases.

2. RELATED WORK

Techniques to optimize space usage in MPI libraries have been presented in [7, 16, 17, 18]. Scalable allocation of resources on IB networks via the use of shared receive queues is presented in [18]. Techniques to scale MPI to a million cores is presented in [7], where MPI_Comm_dup calls are optimized by not duplicating MPI group data-structures in the sub-communicator. Instead the sub-communicator references the parent's group data structure resulting in lower space usage. Exploration of compression of MPI group state for a range of ranks, strided distribution of ranks and bit-maps have been explored by Charawi et. al [17]. Charawi et. al presented overheads from compressed data structures in the latency of MPI point to point calls. Charawi et. al also presented performance of the Linpack benchmark with and without compressed group data structures. However, it should be noted that the Linpack benchmark is not known to be communication intensive and is hence used to benchmark various supercomputers in the Top500 list. Our approach is designed for compressing n-dimensional Cartesian communicators typically found in applications such as Adaptive Multi-Grid solvers and 3D Fast Fourier Transform. We also present a performance analysis of our techniques in communication intensive benchmarks.

3. MPI ON BLUE GENE/Q

The MPICH [2, 3] libraries provide an implementation of the MPI standard on several target architectures such as BG/Q and Intel x86 InfiniBand (IB) clusters via the MVAPICH variant [13].. Target architectures must implement the MPICH abstract device interface (ADI) [4] to enable MPICH on the new architecture. On BG/Q, the new MPICH ADI from IBM called *pamid* enabled the MPICH port over the Parallel Active Message Interface [5, 6] (PAMI) libraries.

Group data structures in MPICH assign logical process IDs, or LPIDs, in the range **[0 – P)** where P is the size of the group, to each process in the group. MPICH also creates a map within the group to convert the local LPIDs to the MPI_COMM_WORLD global IDs in a dense array of size P. This dense table enables efficient translation from the LPID in the communicator to the global rank in MPI_COMM_WORLD. The global rank is used by the PAMI libraries on BG/Q to route each message to its destination torus coordinates. In addition, MPICH maintains a virtual connection reference (VCR) table for each process to store connection state to every other process. Both the LPID mapping and virtual connection tables are O(N) data structures with limited scalability. MPICH also enables an optimization where the virtual connection table stores connection objects only for processes that have active communication with the calling process.

On BG/Q, all processes in the job exchange messages on a 5D torus network, that provides reliable datagram message passing services between nodes connected over this torus network. As a result, no connections need to be established between processes to initiate communication operations on the torus network. In the MPICH pamid device, the virtual connection table and the group LPID mapping table is merged into a single communicator VCR table. This VCR table is used to map LPIDs of communicators to MPI_COMM_WORLD global ranks that are used to initiate PAMI library calls. The global rank is used in PAMI to route to the destination torus node and MPI process on that node. The VCR table requires `O(P)` storage and on large communicators it can consume a significant fraction of the DDR memory available to an application thread on BG/Q. For example, if the Exascale system has 16 million cores and 16 million MPI ranks, a communicator spanning all the ranks will require a VCR table of size 64MB/rank assuming each connection slot stores just a 32-bit LPID to global rank mapping. Note, this is a significant space requirement for each communicator.

4. COMPRESSION OF GROUP STATE

In this paper, we explore compression of Cartesian communicators. Such communicators are typically obtained by the MPI_Cart_create call that is used in a wide variety of parallel applications. For example the 3D-FFT computations can be executed by partitioning the processor group into a 2D Cartesian topology where processors communicate with their neighbors along rows and columns. Similarly linear algebra applications call 2D cart create that results in row and column communicators where sub-matrices can be broadcast to compute algorithm solutions. In lattice-QCD, processors communicate with neighbors on a 4D application grid. This can be achieved by a 4D cart-create and point-to-point messages exchanged with near neighbors on this 4D Cartesian communicator. Our algorithms to compress Cartesian communicators are general and subsume rank ranges and strided communicators (presented in [17]) as these can be treated as 1D and 2D Cartesian communicators. In addition, our schemes can also compress n-D Cartesian communicators. When compressed the communicator state stores the bottom left and top right coordinates in the n-D mesh instead of translations for all the ranks. On network architectures that have the n-D torus topology, we take advantage of the location information to compute the overlaying mesh over all the process ranks in the communicator. On compute clusters with other topologies, we store a few n-D virtual grid mappings for MPI_COMM_WORLD based on the number of ranks in the MPI application. Typically most cart-create and comm-splits from MPI_COMM_WORLD that do not reorder ranks should be compressed by our scheme. In addition to space optimizations we show minimal performance overheads by caching translations of group rank to global rank in the communication sensitive 3D-FFT benchmark.

We explore an aggressive technique where we compress the LPID to global rank mapping table on topological communicators. On commonly occurring rectangular communicators, where the ranks are arranged in an n-dimensional rectangle, it is possible to just store the BG/Q torus coordinates of the bottom left and the top right corners and compute the ranks of the intermediate processes. This reduces storage requirements by replacing the VCR table lookup with computation. This compression technique cannot be used with every possible MPI_Comm_split call. For example this technique will not work for an MPI_Comm_split that creates a random subset communicator or a communicator that is created from an arbitrary graph data-structure.

On other architectures that do not have a torus topology, we extend this idea by creating a cache of virtual *world rectangles* of various dimensions where the product of all dimensions is the same as the number of ranks in the parent communicator. MPI applications typically call MPI_Dims_create and then use the outputted dimensions to call MPI_Cart_create and then create communicators that are sub-rectangles of the topology returned by MPI_Dims_create. MPI_Dims_create in MPICH calls a factorization function to create a rectangle with the number of dimensions specified in the input parameters. The product of all the dimensions in this rectangle is also same as the size of the MPI job. In our implementation, we cache virtual 2D, 3D and 4D world rectangles that have dimensions that match the outputs returned by MPI_Dims_create for 2, 3 and 4 dimensions.

Our implementation modifies MPI_Comm_split so that the VCR table of the communicator is compared with the various cached virtual world rectangles to verify if the communicator is a sub-rectangle of one of the cached world rectangles. If that is the case then the bottom left and top right coordinates in it are stored in the VCR table structure. Using these coordinates in the virtual world rectangle, the LPID in the communicator can then be used to compute the global rank when initiating a communication operation. To avoid this computation on all communications, several caching schemes are explored in the next section.

The VCR table for a communicator is created during MPI_Comm_create, MPI_Comm_split and other MPI calls that create the new communicators. Our optimized MPICH library first creates a dense VCR table and then for topological sub-communicators, the VCR table is compressed and memory allocated to it is released. The dense VCR table is temporarily required to verify that the communicator is compressible. We use the following algorithm to compress the VCRT. We first find the sub-rectangle in the world rectangle structure to compute the bottom left and top right coordinates of the sub-rectangle. These coordinates are later used by the mapping function to compute the global rank from the communicator LPID.

1. *First the global rank of the calling process in the communicator is mapped to coordinates in the world rectangle.*

2. *Next a min followed by a max allreduce computes the smallest and largest coordinates in each dimension. These form a sub-rectangle in the world rectangle. The computed sub-rectangle is saved for future comparison.*
3. *The size of the sub-rectangle from the coordinates in Step 2 is compared with the size of the communicator.*
4. *If the sizes don't match then the compression function returns failure.*
5. *If the sizes match, we compute the mapping order of dimensions in the sub-rectangle by observing the neighbors of the origin rank of the sub-rectangle.*
6. *Next, we verify if there is a shuffle in the ranks of the communicator. A shuffle is an irregular permutation of ranks in the communicator. In the presence of a shuffle, mapping LPIDs to global ranks cannot be done via a topological function. To detect shuffles, we first compute the LPID of the calling process in the new communicator via a search in the dense VCR table. This LPID is then used to compute coordinates in the sub-rectangle. This computes the adjusted relative coordinates of the calling process in the new sub-communicator.*
7. *The lower left coordinates of the sub-rectangle in Step 2 are subtracted from the world coordinates of the calling process (Step 1).*
8. *If the coordinates from steps 6 and 7 match, we have verified there is no shuffle and then we return success. If they don't match, there may be a shuffle and so we return failure in compression.*
9. *A bitwise AND allreduce of return values from the compression function on all the processes in the communicator then computes whether the compression is successful on all processes in the communicator.*
10. *If the compression operation is successful, the dense VCR buffers are freed, sub-rectangle structure from Step 2 is saved in the compressed VCR object and the type is set to compressed.*

Observe that the above algorithm has a time complexity of `O(P)`, where P is the number of processes in the newly created communicator. Typically, its performance is dominated by the time to do allreduce operations on the communicator. In the MPICH library, allreduce operations are already executed during communicator creation. So, we don't expect the additional allreduce calls from the compression operation to significantly increase communicator creation time. To summarize, our scheme supports compression of the following types of communicators.

- Strided pattern: This will match with a row or a column in the 2D virtual grid when the stride is the size of one of the dimensions.
- N-Dimensional sub-communicator: This can match with N+1 or higher dimension virtual grids, if the N-Dimensional sub-communicator is a subset of the higher dimensional virtual grid.
- K-Dimensional sub-communicator created from a compressed N-Dimensional sub-communicator: In this case the K-Dimensional sub-communicator will also be the subset of the higher dimensional virtual grid that was used to compress the N-Dimensional communicator.

```
struct Rectangle {
  int ll[NDIMS]; //lower left coordinates
  int sizes[NDIMS]; //dimension sizes
  int order[NDIMS]; //traversal order
};
struct MPID_VCRT {
  Rectangle rect; //Other params not shown
};
int MapLpidToGlobal(MPID_VCRT vcrt,
               int LPID, Rectangle *world) {
  //Compute world coordinates for LPID
  int cur = LPID, global_rank=0;
  for (j = MPIDI_MAX_DIMS-1; j >= 0; --j) {
    sx = vcrt->rect.sizes[j];
    k = vcrt->rect.order[j];
    //Compute coordinates in sub-rectangle
    cx = cur / sx;
    //Add lower left coord. to compute world
    coordinates
    coords[k] = cx + vcrt->rect.ll[k];
    cur -= (sx * cx);
  }
  //Compute global rank from world coord.
  for (j = 0; j < MPIDI_MAX_DIMS; ++j){
    sx = world->sizes[j];
    cx = coords[world->order[j]];
    global_rank += cx * sx;
  }
```

Figure 1: Subroutine showing the coordinate to rank mapping function.

In our current implementation, we cache virtual world rectangles of dimensions 2, 3 and 4, where the sizes of the dimensions correspond to the output of MPI_Dims_create for each of those dimension sizes, respectively. Our algorithm should be extensible to an arbitrary N dimensional Cartesian communicator and the cache of world rectangles can be extended to store significantly larger number of rectangles depending on the application's requirements. The cache can also store virtual world rectangles for the largest perfect square and perfect cube subsets of ranks as several scientific problems use such processor grids. The cache of world rectangles of various dimensions is created in the first call to MPI_Comm_split. In each MPI communicator creation call, steps 1-11 in above algorithm are repeated with each of cached world rectangles to compute if compression is possible using any of them. If the algorithm returns success the corresponding world rectangle structure is also copied into the compressed VCR table structure.

During an MPI communication call the application typically passes the destination processes LPID as an input parameter. This LPID must be converted to the global rank that is then passed to the PAMI library. In MPICH this translation is done by a lookup in the dense VCR table. In our modified version of MPICH, as the VCR table can be compressed, the global rank must be computed from the LPID of the destination process. The mapping function to map the LPID to its global rank is shown Figure 1. It first computes the coordinates in the saved sub-rectangle (member *rect* in the MPID_VCRT structure) and then computes coordinates in the virtual world rectangle. Next, the mapping function converts the world rectangle coordinates to the PAMI global rank. As this function has integer divisions, it can impact the latency of MPI communication calls. We next study two different caching schemes to cache recently translated LPID to global rank mappings so that the mapping function is not called on every MPI call if there is a hit in the caches.

4.1 Examples of communicator compression

Figure 2 shows a 3D virtual grid of dimensions 5x4x4 created for an MPI job with 80 MPI ranks numbered from 0-79. The 3D virtual world rectangle dimensions X, Y and Z are also shown in Figure 2. The ranks increase along the X dimension and then along Y and Z dimensions. Our scheme should compress the following example sub-communicators and many more:

- Ranks 0, 1, 2, 3, 4 is the simplest example as it will match with a row in the X dimension of the virtual world rectangle.
- Ranks 0,5,10,15, as these will match a column in the XY plane of the virtual grid
- Ranks 10-19, as these will form a 2D sub-mesh in the 3D virtual grid
- Ranks 15,35,55,75, as these form a column along the Z-axis in the 3D virtual grid
- Ranks 15,16,35,36,55,56,75,76, as these form 2 adjacent columns in the Z dimension
- Ranks 10,11,15,16,30,31,35,36,50,51,55,56,70,71,75,76, as these for a sub 3D grid of dimensions 2x2x4 starting at rank 10 in the 3D virtual world rectangle.

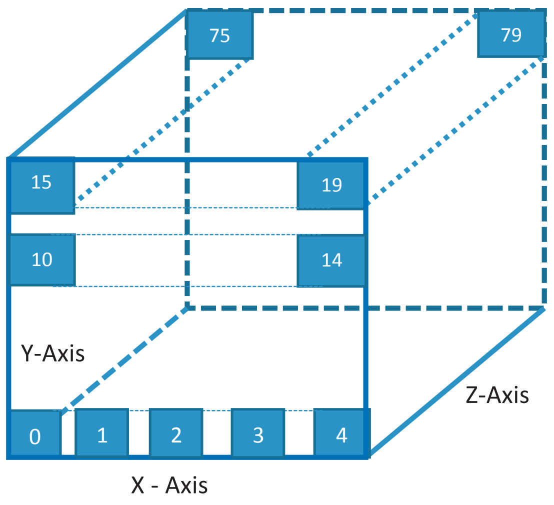

Figure 2: A 5x4x4 3D virtual world rectangle in an MPI job with 80 ranks

4.2 Caching to optimize overheads

We explore two different caching schemes to store recent LPID to global rank translation calls. The first caching scheme is a linear cache that stores the most recent *k* translations in a linear array of the size k. With the linear scheme, the cache is searched using the input LPID and if a match is found the mapping function is not called.

The main drawback of the linear caching scheme is that in the worst case it requires a search of the whole cache. This limits the size of the cache. In order to support larger caches, we experiment with a *2D set-associative* scheme with a low overhead hashing function. In particular, if there are S sets, then we use a mask, HASH_MASK, with the HASH_BITS=$\log_2(S)$ least significant bits (LSBs) set. The hash function computes the row in the 2D cache as (LPID ^ (LPID >> HASH_BITS)) & HASH_MASK. A linear search in the row of the 2D cache will find the most recent translation for that LPID.

Typical values are S=128 (HASH_BITS=7) with 8 entries per row. This can store up to 1024 cached translations. With these values, the 14 LSBs of the LPID are used in the hash function, so all bits are used if the communicator has less than 16K (=2^{14}) ranks. For the same number of rows (sets), a similar function with modestly more masks and shifts would be needed if the hash function were to use all bits in the LPID for larger communicators. To further keep overhead low, a simple round-robin replacement algorithm is used, to avoid tracking reference counts or LRU lists. Since the 128x8 caching scheme requires caching 1K entries, a production quality implementation would not use this optimization for communicators with less than 1K ranks; in such a case the normal LPID to global rank table lookup would be more efficient. Similarly, a production quality implementation would have this compression approach as an option that would only be turned on as needed, when the space impact is significant.

5. PERFORMANCE EVALUATION

We ran a neighborhood communicator create micro benchmark that splits the n-dimensional communicator along each dimension into two n-1 dimensional communicators at each point in that dimension. For example, if the A dimension size is 4 on the ABCDET torus of BG/Q, this test will create 6 communicators with full BCDET dimensions and A cords {(0), (0,1), (0,1,2), (1,2,3), (2,3), (3)}. (Note ABCDE are the dimensions of the BG/Q 5-D torus, while T represents the number of processes per node.) Observe there are only 3 communicators per rank. This process of splitting is repeated along all the dimensions. This communicator-splitting step is then repeated along the remaining BCDET dimensions. As each dimension is split at every coordinate point, this test can create relatively large communicators thereby stress testing the space usage increase in the MPI library from creation of communicators.

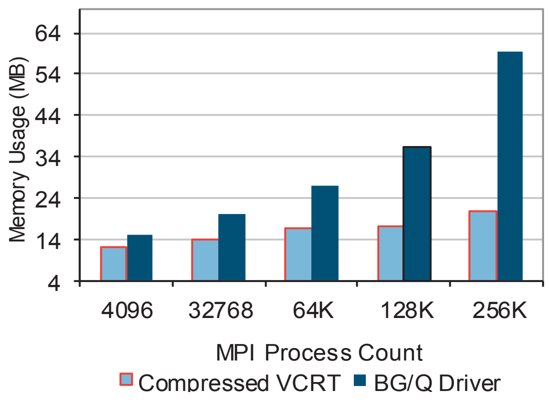

Figure 3: Space usage per node in the communicator split test on BG/Q

Figure 3 shows the per process space usage in Mega Bytes (MB) for the BG/Q communicator split micro benchmark on up to 8192 nodes of BG/Q. For each of the node counts, 32 processes per node were used in the runs. So, at 8192 nodes the MPI job had 262,144 processes. On BG/Q, at 32 processes per node, the memory available per process is only about 512 MB. The total number of communicators created per process is 68 at 8192 nodes. Observe that for 8192 nodes the compression scheme presented in this paper results in space usage reduction from 60MB to 20MB. Also observe with the compressed scheme, space usage increases from about 12MB to 20MB as the node count increases from 128 to 8192, while in standard MPICH memory increases from 15MB to 60MB as the node count increases from 128 to 8192. As an extreme case, we expect that the same communicator split micro-benchmark, when run on a 16 Million MPI rank job and a 4D world rectangle where each dimension is of size 64, will create 252 communicators per rank. The total space usage without compression will be about 2GB/rank, while with compression or elimination of all communicator `O(N)` data structures this space usage would be reduced significantly. Assuming each communicator typically stores only a few Kilo Bytes, the total space allocated to 252 communicators will only be a few Mega Bytes of memory per MPI process.

In our space optimized MPICH libraries, the MPI send, receive and RMA operations use a mapping function to map the communicator LPID of the destination to the PAMI global rank. The overhead of this mapping function can be optimized by caching the most recent translations. Figure 4 shows the ping-pong latency of the various schemes. Note, the 8-way cache scheme is a linear caching scheme with cache sizes 8. The V1R2M0 line represents the performance of the IBM Blue Gene/Q V1R2M0 driver, while, the "No Cache" line shows performance with compressed communicators and no caching. Latency at 1 byte is highest in the No Cache scheme. As the ping-pong benchmark only communicates with one neighbor, the 8-way linear caching scheme effectively minimizes overheads from the communicator compression. Note, as the message size increases, the mapping function overhead becomes less important and all the latency curves merge. On fast superscalar processors such as the latest X86 processors and the IBM Power8 processor we expect the performance overheads of the caching schemes to be further reduced compared to the standard MPI stack.

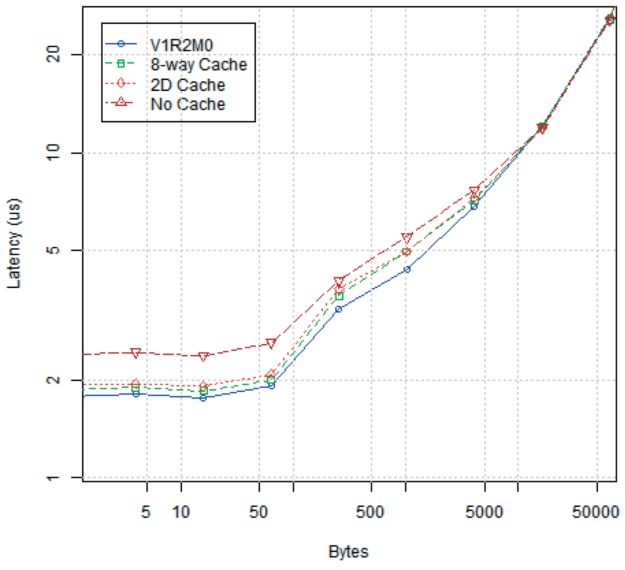

Figure 4: BG/Q Log scale MPI ping pong latency of various caching schemes on 2 neighbor nodes and one MPI processes per node

We also ran micro-benchmarks with more messages exchanged per benchmark iteration. First, we ran a neighbor exchange micro-benchmark, similar to linktest in the Sequoia Phloem test suite [10], on 512 nodes of BG/Q. In this benchmark, all ranks communicate with 6, 8 or 10 nearest neighbors on the 5D BG/Q torus. In the exchange loop, first all receives are posted via the MPI_Irecv call and then all sends to neighbors are posted via the MPI_Isend call. This is followed by a call to MPI_Waitall to wait for the send and receive operations to complete.

The latency of the neighbor exchange with 6, 8 and 10 neighbors with the different caching schemes is shown in Figure 5 with 128-byte message payloads. To accommodate 10 neighbors a cache size of 10 entries or more is required. So we present performance with a linear cache of size 16 (16-way bar in the Figure 5). In addition performance results for the 2D cache, the IBM BG/Q V1R2M0 driver and the un-cached compressed communicator scenario are also presented.

From Figure 5, observe that both the caching schemes perform significantly better than the No-cache scheme. The 16-way caching scheme reduces overhead of compression to within 0.8% when compared with the IBM BG/Q V1R2M0 driver. As the 10-neighbor micro-benchmark does not result in cache misses for the 2D and 16-way schemes, and to test the overheads when there are many more outstanding messages, we ran another micro-benchmark with more partners and for which the cache hit ratio is not necessarily 100%. This micro-benchmark is a multi-hop 3D nearest neighbor exchange in which a process sends messages to all neighbors within d hops for each dimension, for d =1 to 4. The test was run on 512 nodes of BG/Q with 8 processes per node, for a total of 4K processes. For d=1, there are 26 neighbors, with d=2 there are 124 neighbors, with d=3 there are 342 neighbors, and with d=4 there are 728 neighbors.

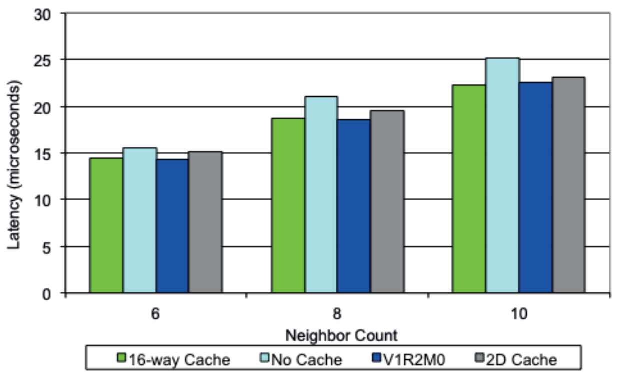

Figure 5: Latency of various caching schemes in the 10 near neighbor benchmark with a 128 Byte payload on 512 nodes of BG/Q

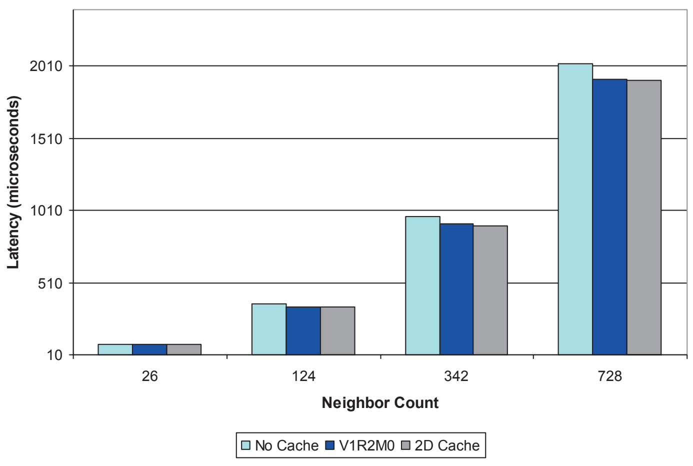

Figure 6: Latency of caching schemes in the multi-hop benchmark with 128B payload on 4096 MPI processes of BG/Q

With the 128x8 2D caching scheme, there are a sufficient number of cached translations so that with perfect hashing there would be no cache misses. The miss rate is effectively zero when d is less than 4, but when d is 4 we observed the hit rate is about 60%. Results for 128B messages are shown in Figure 6. The 2D caching scheme fully eliminates overheads from the mapping function for all values of d, despite the 60% hit rate when d=4. So we can conclude from Figures 5 and 6, in almost all cases the overheads of the mapping function is minimal with our caching schemes.

5.1 3D FFT Evaluation

Fast Fourier Transform (FFT) is an efficient algorithm to compute the discrete Fourier transform (DFT) and the inverse of the DFT. In a three dimensional Fast Fourier Transform (3D-FFT) operation, FFT operations must be performed along the X, Y and Z dimensions. The 3D-FFT operation is used in many scientific applications, such as molecular dynamics Particle Mesh Ewald computation, quantum chemistry and distributed navier stokes (DNS).
The 3D-FFT is parallelized via a 2D pencil decomposition as the 1D decomposition of 3D FFT has limited scalability. In the 2D decomposition, each processor has a subset of the 3D input complex numbers along 2 dimensions and all the 3D FFT input data on the third dimension. This subset is called a pencil. The 3D-FFT operation has three phases of computation along the X, Y and Z dimensions. In each phase of a 3D-FFT operation of size (N * N * N) on P processors, each processor has (N^3/P) data-points and exchanges messages with $\sqrt{P}$ other processors where the size of each message is $(N / \sqrt{P})^3$ elements. With the 2D pencil decomposition processors exchange data with row and column neighbors. During the forward FFT, each processor exchanges data first with its row neighbors in the XY phase, and then column neighbors in the YZ phase.

With backward FFT, each processor exchanges data with its column neighbors during the ZY phase and row neighbors during the YX phase.

We explore 3D-FFT with MPI Point-to-point messages on row and column sub-communicators. These sub-communicators are created via two MPI_Comm_split calls on a 2D Cartesian communicator that has all the MPI ranks in the job. Our communicator compression scheme also compresses these two communicators to minimize space usage. Performance of the 3D-FFT operation with and without communicator VC table compression on 512 nodes and 4096 MPI ranks of BG/Q is presented in Table 1. The columns labeled "Run" show the time for a pair of forward and backward 3D-FFT operations, while the column labeled "Comm." show the communication overhead in the 3D FFT computation. Note, the 3D FFT computation is communication intensive where each rank has multiple communication phases with 64 neighbors and it benefits from our caching schemes. With the 2D caching schemes, the overhead of our compressed MPI libraries over the IBM V1R2M0 driver is about 1% for the 512 problem size and there is no overhead of compression in the 1024 problem size. Although, compression of communicators of size 64 does not lead to significant reduction in MPI library space usage, we expect minimal performance overheads when larger communicators are compressed via our caching schemes that would also results in lower space usage as shown in Figure 3.

Problem Size (N)	V1R2M0		No Caching		2D Caching	
	Run	Comm	Run	Comm	Run	Comm
512	22.8	3.3	23.2	3.6	23.0	3.45
1024	199	26.6	201	29.6	199	28.1

Table 1: Time (ms) to execute a pair of forward and backward 3D FFT computation operations on 4096 MPI ranks.

5.2 AMG Evaluation

The Algebraic Multi Grid application benchmark [19] solves partial differential equations using a hierarchy of grids with different resolutions. Its communication pattern is near neighbor, where processes that have coarser grids send several messages to processes with finer grids. On BG/Q, the AMG application creates a 3D sub-communicator via the MPI_Cart_create call. The size of this Cartesian communicator is the same as MPI_COMM_WORLD. This sub-communicator is topologically mapped to the 5D torus network to minimize communication overheads on the BG/Q torus. Our enhanced MPI library can compresses this MPI sub-communicator using the techniques presented in this paper. When compressed, all communication over this sub-communicator must use the translation functions to convert LPID to global rank. We also explore the 2D caching technique to optimize the overhead of the translation functions. We ran "solver 4" from the standard AMG source distribution with an 8x8x8 refinement on up to 16384 MPI ranks at 32 MPI processes per node and results are shown in Table 2. We present the figure of merit (FOM), that is computed as FOM = (System Size * Iterations) / (Iteration Solve Time). Higher values of the figure of merit show that the application is computationally more efficient. From Table 2, the 2D caching scheme FOM is better at 256 nodes and within 0.3% at 512 nodes than the IBM default V1R2M0 driver. Thus, showing that the AMG application can benefit from the reduced space usage of compressed communicators with comparable performance.

Nodes	Ranks	V1R2M0	No Caching	2D Caching
512	16384	4.493 e9	4.447 e9	4.514 e9
256	8192	2.333 e9	2.313 e9	2.327 e9

Table 2: AMG Figure of Merit (FOM) at 32 MPI processes per node with and without compression.

6. SUMMARY

We showed that space overheads are critical to MPI library design. We explored an algorithm to compress the VCR table on commonly occurring topological communicators. The table lookup is replaced by a call to a mapping function to maps the process LPID in a communicator to its global rank. On BG/Q, the 5D torus coordinates of the compressed communicator are stored and used in this mapping function. To generalize this scheme to other architectures, such as a cluster interconnected by the InfiniBand network with a fat-tree topology, we construct virtual world rectangles in two, three and four dimensions corresponding to the sizes returned in the MPI_Dims_create call. Sub-communicators will be compressed if they are sub-rectangles of the 2D, 3D or 4D virtual world rectangles. We also quantified the overhead of this mapping function in micro benchmarks and explored caching schemes to minimize the mapping function overhead.

Performance results show reduction in space usage of about 3x in the communicator split micro-benchmark on 8K nodes and 256K MPI processes with 68 relatively large communicators per process. The best caching schemes result in near-peak performance in the ping-pong benchmark; the performance impact is minimal for short messages and that impact decreases as the message size increases. For the multihop neighbor micro-benchmark, with many neighbors, the performance impact was negligible for small messages and lower neighbor counts. The impact was also eliminated in the benchmarks as the message size increased. These are overheads on the extremely simple in-order, single (integer) issue BG/Q cores, and we expect the overheads to be less on superscalar processor cores. However, in the 3D FFT benchmark and the AMG application kernel, the 2D caching scheme eliminated overheads from compression of communicators in almost all cases, thus showing the effectiveness of our techniques to reduce space usage in MPI libraries.

7. ACKNOWLEDGMENTS

We would like to thank James Sexton, Dong Chen, Robert Senger, Ruud Haring and Yutaka Sugawara for their support. The work presented in this paper was funded in part by the US Government Department of Energy contract number 7078416.

8. REFERENCES

[1] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker. The IBM Blue Gene/Q interconnection network and message unit. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pages 26:1-26:10. ACM, 2011.

[2] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. MPICH: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. Parallel Computing, 22(6):789-828, September 1996.

[3] MPICH2. http://www.mcs.anl.gov/mpi/mpich2

[4] W. Gropp and E. Lusk. MPICH ADI Implementation Reference Manual, August 1995.

[5] 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-Burow. PAMI: A parallel active message interface for the BlueGene/Q supercomputer. In Proceedings of 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Shanghai, China, May 2012.

[6] K. D. Ryu, R. Bellofatto, M. A. Blocksome, T. Gooding, T. A. Inglett, S. Kumar, A. R. Mamidala, M. G. Megerian, S. Miller, M. T. Nelson, B. Rosenburg, K. D. Ryu, B. Smith, J. Van Oosten, A. Wang and R. W. Wisniewski, IBM Blue Gene/Q System software stack, IBM Journal of Research and Development, Volume 57 Issue 1, January 2013, Pages 55-65.

[7] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff, MPI on Millions of Cores. Parallel Processing Letters, 21(1):45--60, March 2011.

[8] Stevens, R., White, A.: Report of the workshop on architectures and technologies for extreme scale computing. http://extremecomputing.labworks.org/hardware/report.stm (December 2009)

[9] IBM Blue Gene/P Team, Overview of the IBM Blue Gene/P project, IBM Journal of Research and Development, Volume 52, No. 1/2, 2008

[10] ASC Sequoia Benchmark Codes: Phloem. https://asc.llnl.gov/sequoia/benchmarks/#phloem (May 2011)

[11] David Goodell, William Gropp, Xin Zhao and Rajeev Thakur, Scalable Memory Use in MPI: A Case Study with MPICH2, Recent Advances in the Message Passing Interface Lecture Notes in Computer Science Volume 6960, 2011, pp 140-149

[12] Buntinas, Darius, Guillaume Mercier, and William Gropp. "Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem." Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on. Vol. 1. IEEE, 2006.

[13] MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE, http://mvapich.cse.ohio-state.edu

[14] J.A. Ang, R.F. Barrett, R.E. Benner, D. Burke, C. Chan, D. Donofrio, S.D. Hammond, K.S. Hemmert, S.M. Kelly, H. Le, V.J. Leung, D.R. Resnick, A.F. Rodrigues, J. Shalf, D. Stark, D. Unat and N.J. Wright, Abstract Machine Models and Proxy Architectures for Exascale Computing, http://www.cal-design.org/publications/publications2

[15] A. Petitet, C.Whaley, J. Dongarra, and A. Cleary, HPL: A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. Innovative Computing Laboratory, 2000. Available at http://icl.cs.utk.edu/hpl.

[16] Jesper Larsson Traff, "Compact and Efficient Implementation of the MPI Group Operations", in proceedings of recent advances in the message passing interface, EuroMPI', volume 6305 pp 170-178.

[17] M. Charawi and E. Gabriel, Evaluating Sparse Data Storage Techniques for MPI Groups and Communicators, In Proceedings of the International Conference on Computational Science, Krakow Poland, 2008, pp 297-308.

[18] Sayantan Sur, Matthew J. Koop, and Dhabaleswar K. Panda. High-performance and scalable MPI over In_niBand with reduced memory usage: an in-depth performance analysis. In Proceedings of the 2006 ACM/IEEE conference on Supercomputing, page 105, New York, NY, USA, 2006. ACM.

[19] Sequoia Algebraic Multi Grid (AMG) Benchmark https://asc.llnl.gov/sequoia/benchmarks/#amg