Memory Compression Techniques for Network Address Management in MPI

Yanfei Guo*, Charles J. Archer†, Michael Blocksome†, Scott Parker*, Wesley Bland†, Ken Raffenetti* and Pavan Balaji*
*Argonne National Laboratory, Lemont, IL, USA †Intel Corporation, Santa Clara, CA, USA

***Abstract*—MPI allows applications to treat processes as a logical collection of integer ranks for each MPI communicator, while internally translating these logical ranks into actual network addresses. In current MPI implementations the management and lookup of such network addresses use memory sizes that are proportional to the number of processes in each communicator. In this paper, we propose a new mechanism, called AV-Rankmap, for managing such translation. AV-Rankmap takes advantage of logical patterns in rank-address mapping that most applications naturally tend to have, and it exploits the fact that some parts of network address structures are naturally more performance critical than others. It uses this information to compress the memory used for network address management. We demonstrate that AV-Rankmap can achieve performance similar to or better than that of other MPI implementations while using significantly less memory.**

I. Introduction

MPI is the most commonly used programming model for scientific computing on large supercomputing systems. Consequently, keeping up with the growing scale of such systems is a critical aspect in the design of MPI implementations. In the past few years, tremendous improvements have been made in MPI implementations with respect to avoiding data structures that scale linearly or superlinearly with system size [1]. Despite these improvements, however, MPI implementations today still have scalability limitations. One example is MPI's network address management.

MPI processes are logically represented as integer *ranks* within communicators, while the MPI implementation internally maintains the physical network addresses of these processes. When an application moves data between processes by addressing them as ranks, the MPI implementation internally translates such ranks into the corresponding network addresses before communication is performed. Such network address management has two closely related aspects. First, a ***network address*** for each peer process needs to be maintained; this includes the hardware communication address itself as well as other associated data structures. Second, a mapping between the ranks and the network addresses, namely, a ***rank-address mapping***, needs to be maintained for each communicator. These structures exist in practically every MPI implementation today, even though the terminology used is sometimes different. For example, MPI implementations such as MPICH, MVAPICH, Intel MPI, Cray MPI, IBM Blue Gene MPI, Microsoft MPI, Tianhe MPI, and Sunway MPI use the terminology of virtual connections (VCs) for network addresses and virtual connection reference tables (VCRTs) for the rank-address mapping structures. Open MPI and Fujitsu MPI, on the other hand, use the terminology of "proclist" and "plist", although conceptually they are no different from VCs and VCRTs.

Most MPI implementations manage network address and rank-address mapping structures in a way that is optimized for performance. For example, the number of dereferences or lookups is minimized, and few or no branches occur in the performance-critical path. Unfortunately, not much attention has been paid to the memory usage of these data structures. For example, data structures related to different transports (e.g., network and shared memory) are embedded in the network address structures, rather than referenced from it. Similarly, data structures related to the shared-memory topology, which are required for transport selection, are embedded in the network address structures for fast lookup. Such a model, while optimized for performance, costs substantial memory.

This situation is also true for the rank-address mapping structures. Maintaining the rank-address mapping metadata is complicated by the fact that the MPI standard allows users to create new communicators by arbitrarily reordering the ranks compared with the parent communicator. That is, a single process can have two completely different ranks within two different communicators, with no correlation between the two. Because there are $P!$ (P factorial) valid mappings of communicator ranks to network addresses, where P is the number of processes in the communicator, lookup tables—such as VCRTs—remain the common practice for maintaining this metadata. Although this simple approach works for any possible reordering of ranks, it is not memory efficient. It takes $O(P)$ memory space on each process for each communicator, that is, $O(nP)$ total memory space on each process, where n is the number of communicators created by the process. Consequently, such metadata can result in the MPI implementation consuming a significant fraction of the available memory, particularly for very large supercomputers.

Our objective in this paper is to reduce the memory consumption of such network address management for MPI processes by using appropriate compression techniques. The important aspect here is not how we can compress the network address management but, rather, how we can do so with virtually no performance degradation. For most MPI implementations performance is the most significant metric for measuring impact, and generally any nonzero performance overhead is considered too high. We propose a new mechanism, AV-Rankmap, for network address management that uses several compression techniques to minimize the memory space used for network address management. We study the behavior of AV-Rankmap with respect to two properties: memory usage and performance.

With respect to memory usage, although AV-Rankmap does not improve memory usage in the worst-case scenario, it does significantly improve the common cases into which most ap-

This work is supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357. This research used Blues, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility.

plications fall. For example, we decouple the network address structure to distinguish elements based on various properties such as commonality of use, compressibility, and network-specific attributes. This decoupling allows applications that use only a subset of the features in MPI, without relying on the full generality of MPI, to benefit from a smaller overall memory footprint. Similarly, for applications that create communicators whose rank-address mapping follows certain patterns—three forms of which are studied in this paper: *direct*, *offset*, and *strided*—we detect such patterns and use them to reduce the rank-address mapping metadata storage. When we are unable to detect any pattern in the rank-address mapping, we simply fall back on the original lookup-table-based model.

With respect to performance, we study both the communicator creation path (which is typically not performance critical) and the data communication path (which is typically performance critical) and measure the overhead added in each case. For the noncritical path, we aim to keep the additional cost low, although some extra overhead is often tolerable. The additional cost is due primarily to the rank-mapping pattern detection involved in AV-Rankmap.

For the performance-critical path, however, the overhead needs to be virtually invisible for the proposed approach to be viable. For a more quantitative measure, a high-performance and finely tuned MPI implementation would cost as little as $\sim$50 instructions on the communication path all the way from the application to the low-level network communication layer (e.g., in `MPI_Put`). Adding just five additional instructions in this path would lead to a 10% overhead in performance (assuming, for simplicity, that all instructions are equally expensive) which can be too expensive if this performance loss is not balanced elsewhere. Keeping this cost in mind, we perform a detailed instruction and cache-level analysis of the performance-critical path and show that although AV-Rankmap adds a few additional instructions for address translation in the performance-critical path, the lost performance due to the additional instructions is more than compensated for by the improved cache activity because of the smaller memory footprint, thus leading to similar or better performance while using significantly less memory.

The overall design of AV-Rankmap and a detailed evaluation with both microbenchmarks and real applications are showcased in this paper on up to 786,432 MPI processes. We demonstrate that AV-Rankmap can improve the memory usage of current MPI implementations by several orders of magnitude in many important cases.

II. Survey of Communicators in Applications

MPI provides a number of different communicator creation techniques. An overview of these communicator creation techniques can be found in [2]. In practice, however, some techniques are more commonly used than others. To understand the communication creation models used in various applications, we performed a survey of a large number of applications comprising the NAS parallel benchmarks [3], CORAL benchmarks [4], DOE codesign applications [5], [6], and other large applications that consume significant compute cycles at large supercomputing centers [7], [8]. Although our survey covered 62 applications, we highlight only a small subset of the survey here, because of space constraints.

Nek5000. Nek5000 is a highly scalable spectral-element method for solving computational fluid dynamics problems. Its computational model relies on solving computational grids at an increasing level of refinement. That is, for a given computational problem, it creates multiple grids, each at a different granularity or coarseness. Solving each grid gives an approximate solution to the problem. Solving the next-finer grid then refines the result based on the approximation generated by solving the previous coarser grid.

Because each grid requires its own communication context, Nek5000 creates a new communicator for each grid. These communicators are essentially duplicates of `MPI_COMM_WORLD`, and the number of communicators created increases as the number of levels of refinement desired by the application increases. A rule of thumb is that as the problem size grows, the number of refinement levels (and hence the number of communicators created) grows as well. In current production runs of the application, a "medium-scale" problem typically creates 24 refinement levels (i.e., 24 new communicators), and a "large-scale" problem typically creates 86 refinement levels (i.e., 86 new communicators).

NWChem. NWChem is a quantum chemistry application suite featuring a broad set of simulation capabilities targeted at many areas including quantum simulation of molecules with heavy isotopes and multiscale methods relevant to environmental chemistry. For its core linear algebra computations, NWChem (using libraries such as ScaLAPACK) creates virtual two-dimensional data grids on which the computation is carried out. It splits the MPI processes into "row" and "column" communicators using `MPI_Comm_split`, so each process is a part of a "row" and a "column" communicator. Data exchange and synchronization are then limited along these smaller communicators. We note that during the split, process ranks are not reordered. Thus, the ranks of the processes in the "row" communicator are essentially the same as the ranks of the processes in `MPI_COMM_WORLD` but are offset by a constant value. Similarly, the ranks of the processes in the "column" communicator can be calculated with fixed offset and stride values. Such row/column communicators are common in other applications, such as QBOX [9], as well.

Apart from the core linear algebra computations, NWChem spends a large fraction of its computation on force calculations. These are typically done through one-sided communication that, after passing through multiple layers of the software stack, eventually use MPI-3 one-sided communication (or RMA) windows internally. For each RMA window, the MPI implementation internally creates a new communicator that is a duplicate of the parent communicator from which the RMA window is being created, in order to perform the required data movement and synchronization. NWChem creates three to four RMA windows, depending on the problem being solved. When it is used together with the Casper [10] software stack for asynchronous progress, for each window created by NWChem, Casper creates as many duplicate windows as the number of cores on each node of the machine. For example, when

TABLE I: Mapping models for communicators.

Application	Dup	MPI_Comm_split			Topo	Intercomm
		Offset	Stride	Irreg.		
Nek5000	x					
NWChem	x	x	x			x
HACC					x	
QBOX	x	x				
QMCPACK		x	x			
CAM-SE	x	x				
NAMD	x	x				
LSMS		x		x		
SP	x	x				
BT	x	x	x			
FT	x	x				
Graph500	x	x	x			
Nekbone	x					
SNAP	x				x	
MCB	x					
cian2		x				
MCCK		x	x			
mocfe_bone		x	x			
pynamic	x	x	x			
MACSio	x					
AMG2013	x					
CNS	x					
SMC	x					
AMR	x					

NWChem is executed on the IBM Blue Gene/Q with 16 processes on each node, it creates 64 RMA windows (and thus 64 new communicators). When it is executed on the Intel Xeon Phi with 60–70 processes on each node, it creates on the order of 300 RMA windows (and thus 300 new communicators).

HACC. HACC is an astrophysics framework that simulates the formation of structure in the expanding universe. HACC's computational model is similar to the linear algebra portion of NWChem in that they both rely on multidimensional data grids for their computation. However, HACC does not split its communicators; instead, it creates multiple topology-aware Cartesian communicators (using `MPI_Cart_create`) representing three-dimensional, two-dimensional, and one-dimensional distributions of the problem. Like NWChem, HACC does not reorder the processes in the new communicator. Thus, apart from the additional topology information that is attached to the communicator, the process mapping itself is identical to that of the parent communicator.

Summary. Table I summarizes the communicator creation models used in a number of applications. In this table, we have classified the communicator creation models into several categories: "dup" refers to the case where a duplicate communicator is created either directly by using one of the communicator duplication functions or indirectly (e.g., creating an RMA window); "offset" refers to the case where the process ranks in the new communicator are identical to that of the parent communicator, but at a fixed offset; "stride" refers to the case where the process ranks in the new communicator can be calculated based on a fixed offset and stride from the parent communicator; "irreg" refers to the case where no pattern in the mapping is detected; "topo" refers to the cases where a topology-aware communicator is created; and "intercomm" refers to the case where intercommunicators are used.

III. Design of AV-Rankmap

In this section, we first focus on the traditional VC-VCRT approach, the data structures it maintains, and its advantages. Then we describe AV-Rankmap, its overall design, how it differs from VC-VCRT, and its benefits and disadvantages.

A. *Traditional VC-VCRT Approach*

The traditional VC-VCRT approach used in most MPI implementations uses a simple two-level hierarchy. At the top level is a VCRT structure, which essentially is a collection of pointers to each VC structure. The VCRT is an $O(P)$ structure that is statically allocated at initialization time.

At the bottom level is a VC structure that contains the required information for communicating with a process. In theory, the VCs themselves can be fully dynamically allocated, for example at the time of the first communication with the corresponding process. In practice, however, most MPI implementations today choose to statically allocate a small part of the VC (basic bookkeeping information) and dynamically allocate the more expensive portions of the VC on demand (such as network connections and communication buffers). Thus, the VCs use another $O(P)$ memory space.

The VC is organized to minimize the number of dereferences. Consequently, all the information required for communication is embedded in this structure, rather than referenced from it. We classify the elements of the VC into three categories: core network access information, multitransport functionality, and functionality for dynamic processes.

Core Network Access Information. The core network access information refers to the basic network-specific functionality that is necessary for accessing a remote process. This includes information such as target endpoint information. For example, for InfiniBand, this would be the queue-pair information to which we can send data. Such information is the most basic and essential part of the VC and is required for any communication operation.

Multitransport Functionality. Almost every MPI implementation allows for data to be communicated over multiple transports. At least two transports are provided by all MPI implementations—shared memory for intranode communication and a network interconnect for internode communication—although some MPI implementations allow for more than two transports to be used simultaneously. Each transport has its own collection of information, such as communication functionality to use, communication thresholds for eager/rendezvous communication, and queues for temporary communication buffers. For fast lookup, such information is directly embedded in the VC structure itself, thus avoiding a dereference. The cost of doing so, however, is that (1) the transport-specific information is replicated a large number of times across the different VCs and (2) some VCs might maintain more information than what they need for communication with the peer process that they correspond to (despite minimizing such additional information using unions).

Functionality for Dynamic Processes. Dynamically spawned or connected processes are a core part of the MPI standard, although they are rarely used in applications. However, current VCs tend to give such functionality importance (with respect to performance) equal to that of more commonly used functionality such as send/receive. Consequently, elements that are required to implement dynamic processes are embedded in the VC structure as well and use up memory space irrespective of whether the application uses dynamic processes or not.

B. AV-Rankmap: Address Vector Elements

AV-Rankmap retains the concept of network address and rank-address mapping structures and instantiates them with data structures called *address vector elements (AVEs)* and *Rankmap*, respectively. These would be logically equivalent to VCs and VCRTs in the VC-VCRT model. AVEs reduce the size of the network address, compared with VCs, based on various properties such as commonality of use, compressibility, and network-specific attributes. The collection of AVEs for all peer processes is together referred to as the *address vector (AV)*. Rankmap reduces the memory footprint required for rank-address mapping, where possible, by detecting common mapping patterns, unlike VCRTs that always allocate an $O(P)$ lookup table.

In this section, we discuss AVEs and the AV. As described in Section III-A, the traditional VC structure has three classes of components. Of these, the core network access information is the most critical part and is retained mostly as-is in the AVE structure. The only, relatively minor, difference is that AVE allows the actual network address to be either directly embedded in the structure (if it is small) or dereferenced from it (if it is large enough that it needs to be dynamically allocated on-demand). Multitransport and dynamic process functionality, on the other hand, are significantly compressed, compared with VCs, as we demonstrate below.

Compressing the Multitransport Functionality. As mentioned in Section III-A, traditional VCs maintain functionality associated with multitransport communication within the VC structure itself. This approach is highly redundant since the number of transports used is typically much smaller than the number of VCs. The number of VCs is equal to the total number of processes in the system, while the number of transports is equal to the number of networks being used (which is typically just two: shared memory and an internode network). Consequently, if we can decouple the transport-specific functionality, such information can be highly compressible. The challenge, however, is that such decoupling must be done in a way that it still retains fast lookup of this data—the primary reason these fields were embedded in the VC in the traditional model.

In AV-Rankmap, we carefully decouple such functionality from the corresponding AVE structure by considering two sets of transport-specific variables: (1) a single variable that identifies which transport to use and (2) a collection of variables that are used by the transport itself. These two sets have significantly different properties.

Identifying Which Transport to Use. The variable that identifies which transport must be used for a given peer process should either be embedded directly in the AVE (similar to what we do in VCs) or be easily and quickly computable. Fast computability is, unfortunately, not easy, particularly when the processes are not laid out in a homogeneous manner. Thus, we chose to always store this information inside the AVE structure, using just enough bits to store the number of available transports (single bit for two transports). While in theory this is an $O(P)$ data structure, in practice network transport addresses tend to have unused bits that can be used here without adding additional memory overhead. For example, the libfabric [11] network API allows network transports to use 63-bit network addressing, thus leaving behind one bit for such transport-selection functionality. Similarly, the UCX [12] network API uses aligned pointers for network addressing where the last two to three bits are unused (depending on whether the alignment is 4-byte or 8-byte). Extracting this information at runtime requires a bit-mask or bit-shift operation, which is a single (fast) instruction on most architectures.

Accessing Transport-Specific Information. Decoupling transport-specific information from the AVE structure improves compressibility, but it adds an additional address dereference (e.g., a pointer lookup) to access this information. There is no escaping this dereference, unfortunately. But we can attempt to minimize its cost. Two costs are associated with this additional dereference: (1) cache penalty for looking up the additional information and (2) instruction costs (or instructions per cycle). Of these, we expect the cache penalty not to be a significant issue. Specifically, for applications that perform frequent communication, these fields would already be in the processor cache anyway, and embedding them in the AVE structure or not does not add any additional penalty as long as the processor cache has sufficient associativity. On the other hand, for applications that do not perform frequent communication and might not be able to retain the transport-specific information in their cache, the communication cost itself will likely not be as big a concern, and the additional cache-miss to access this information will not be as important.

For the instruction costs, however, we have not yet been able to identify a satisfactory solution. The additional dereference results either in additional instructions (to load the transport-specific information to registers) or in more expensive instructions (e.g., memory-based instructions rather than register-based instructions). Even if the data is in cache, memory-based instructions seem to be fairly expensive compared with register-based instructions, thus impacting the instructions per cycle that we can achieve. While this is certainly a concern, as we will demonstrate later, the smaller memory footprint of AV-Rankmap leads to fewer cache misses and hence more than compensates for such additional instruction cost. From an overall performance perspective, therefore, such decoupling of transport-specific information is still a win.

Deprioritizing Dynamic Processes. As described in Section III-A, the traditional VC structure gives equal importance to all fields, irrespective of how widely they are used in applications. In particular, dynamically spawned or connected processes need additional information such as which process group they belong to. Embedding this information into AVE can improve performance, but it also increases the size of the data structure for applications that do not use them. In AV-Rankmap, we decided to deprioritize dynamic processes so that applications that do not use dynamic processes use lesser memory. However, this deprioritization comes at a cost:

applications that do use dynamic processes can, in some cases, use more memory than VC-VCRT does.

Specifically, communicators that contain a combination of processes where some of them are from one MPI_COMM_WORLD and some others from a different MPI_COMM_WORLD need to maintain two pieces of information: the process group that the remote process belongs to and its rank within that process group. In VC-VCRT, both these pieces of information were stored inside the VC. Thus, it would increase the base memory usage but would not add extra memory usage for each new communicator created. In AV-Rankmap, however, we move this information out of AVE and into the communicator structure. Thus, the base memory usage would be small; but if the ranks of the new communicator are arbitrarily reordered compared with the parent communicator, the incremental memory usage per communicator would be $O(P)$, where P is the size of the communicator. Fortunately, as we describe in Section III-C, this is the worst-case scenario. In most cases, we can detect patterns in the formation of such communicators and can substantially compress this information.

Based on these compression techniques, we have been able to reduce the size of the AVE structure to 12 bytes compared with the 480 bytes needed by the VC structure: a 40-fold compression.

C. AV-Rankmap: Rank-Address Translation with Rankmap

Rank-address translation is essentially the process of finding the appropriate network address to communicate with, given a communicator and a rank within that communicator. For any given process, the index of the AVE for that process is referred to as the *lpid* (local process ID). If the application spawns or connects to another MPI_COMM_WORLD, then a new AV is created that contains AVEs corresponding to the new group of processes. Each such spawned or connected group is referred to as a "process group" and has its own unique ID, called the *pgid*. A *pgid* refers to both the process group and the AV associated with that process group. Thus, a *pgid* and an *lpid* together can uniquely identify any process in the application. The rank-address translation process can be formally represented as the following mapping.

$< comm, rank > \rightarrow < pgid, lpid >$

Once the AV is created, the next step is to create a mapping between the communicator and one or more AVs. As discussed in Section II, for the communicators created in most applications, the rank-address mappings are not arbitrary: they follow a simple, predefined pattern. AV-Rankmap takes advantage of this behavior to try to identify common patterns and use this information to compress the memory space required for maintaining such mapping. It identifies three regular mapping models: *direct*, *offset*, and *stride*, which represent the most common use cases in applications.

The ***direct*** model indicates that the ranks in the new communicator map to the same AV and its *lpids* in exactly the same order as MPI_COMM_WORLD. Thus, we do not need any additional storage other than the AV. The index in the AV (i.e., *lpid*) is the same as the communicator rank in this model. Communicators that are duplicates of MPI_COMM_WORLD fall into this model. We note that in VC-VCRT such communicators still needs at least one $O(P)$ VCRT.

The ***offset*** model indicates that the ranks in the new communicator map to the same AV and its *lpids* in exactly the same order, but at a fixed offset, as MPI_COMM_WORLD. In this model, apart from the AV itself, the only additional piece of information that needs to be stored is the offset. The index in the AV (i.e., the *lpid*) can be calculated as the communicator rank plus the offset. Communicators that have been split without reordering from MPI_COMM_WORLD or one of its duplicates fall into this model.

The ***stride*** model allows the rank in a communicator to be mapped to a noncontiguous subgroup of MPI_COMM_WORLD with a fixed stride. The stride model has two parameters: stride and offset. The stride is the interval between the start of each block. The offset is the start index of rank 0 in the communicator.

Aside from these models, AV-Rankmap considers several other regular models, namely, *blockstride*, *md-blockstride*, *lut-stride*, *mlut-stride*, *lut-blockstride*, *mlut-blockstride*, *lut-md-blockstride*, and *mlut-md-blockstride*. *blockstride* is a generalization of the *stride* model that allows multiple contiguous ranks at each stride (e.g., a 2D subarray of ranks). *md-blockstride* is a generalization of the *blockstride* model for higher-dimensional arrays of ranks (e.g., a 3D subarray of ranks). *lut-stride*, *mlut-stride*, *lut-blockstride*, *mlut-blockstride*, *lut-md-blockstride*, and *mlut-md-blockstride* are generalizations of the *stride*, *blockstride*, and *md-blockstride* models where the parent communicator uses the *lut* or *mlut* models. We do not present additional details on these models here because, in our survey, we have not encountered any applications that can use these models; thus they might not be of practical relevance at this point, although, academically, they are still interesting to study. All regular mapping models use constant memory regardless of the number of ranks in the communicator.

When the rank-process mapping does not fit any of the regular mapping models, AV-Rankmap falls back to irregular mapping using a lookup table. We designed two lookup tables for irregular mapping: *lut* and *mlut*. The lut is a dense array of lpids, similar to VCRT. It is used when all the associated processes are in the same process group. The mlut is an array of $< pgid, lpid >$ pairs. It is used only when some of the processes in the communicator belong to a different process group from others. We note that the lut requires only that all ranks have the same pgid; this pgid does not need to be zero. An example is an intercommunicator that is created when a new process group is connected. The remote group and the local group have different pgids, but the ranks in each group have the same pgid. Hence, both the remote group and the local group are represented by using lut instead of mlut. We need mlut only when we merge these two groups together using MPI_Intercomm_merge.

Note that there is an inclusive property in both the regular and the irregular mapping models. For example, a direct model can also be described as an offset model with the offset value equal to zero. This inclusive property is carefully exploited

TABLE II: Child communicator mapping models for a given parent communicator mapping model (row) and *indirect mapping array* model (column).

	Direct	Offset	Stride	Irregular
direct	direct	offset	stride	lut
offset	offset	offset	stride	lut
stride	stride	lut	lut	lut
lut	lut[a]	lut[a]	lut	lut
mlut	mlut[a]	mlut[a]	mlut	mlut

[a] The child communicator does not create a new lookup table but points to the lookup table of the parent communicator.

during communicator creation.

For each communicator, AV-Rankmap stores the mapping model and its corresponding metadata in a structure called *Rankmap*. As mentioned earlier, Rankmap is logically equivalent to the VCRT, though the VCRT always uses an $O(P)$ lookup table.

1) Creating the Rank-Address Translation for a Communicator: Users create new communicators based on existing communicators. When a new communicator is created, we have two pieces of information: (1) the *indirect mapping array* that maps the ranks from the new communicator to the ranks in the parent communicator and (2) the compressed mapping model that allows us to translate a rank in the parent communicator to the corresponding AVs and lpids. Our task here is to create a similar compressed mapping model that allows us to translate a rank in the new communicator to the corresponding AVs and lpids. This is done in three steps. First, we try to detect a pattern in how the ranks in the child communicator correspond to the ranks in the parent communicators, i.e., the mapping model of the *indirect mapping array*. Second, we use this detected pattern together with any mapping pattern that exists between the parent communicator and the AVs/lpids, to generate a new mapping pattern between the child communicator and the AVs/lpids. Third, if the child has an irregular mapping, we try reduce it to one of the regular mapping models.

During the creation of a communicator, the MPI implementation first creates an *indirect mapping* between the ranks in the child and the parent communicators. This indirect mapping is a constant-size data structure for simple duplicated communicators but can be an $O(P)$ array for more complex communicator creation routines, although it is only temporarily allocated and is deallocated at the completion of the communication creation operation. The mapping information directly follows from the communicator creation function being used. Once this information is obtained, the next step is to convert this indirect mapping to a compressed mapping pattern. To this end, we start by assuming that the mapping pattern is offset based, and we calculate the offset value based on *rank 0* in the child communicator. We then try to validate the offset value with the remaining ranks in the indirect mapping. If successful, we set the mapping pattern between the child and parent communicator ranks to be offset based. If not, we move on to stride mode and attempt to calculate the possible block size; and we follow a similar validation process before finally falling back to the irregular model if the stride mode cannot be validated.

After detecting the mapping pattern between the child and parent communicator ranks, the next step is to detect the mapping pattern between the child communicator ranks and the AVs/lpids. Recall that the mapping model of a communicator describes how the ranks are translated to AV table indices. Therefore, the mapping model of a child communicator is determined by both the rank-address mapping of the parent communicator and the pattern of the mapping between the child and parent communicators. Table II shows the state machine for determining the child communicator's mapping.

If the child communicator has an irregular mapping model (lut or mlut), the ranks are not necessarily irregularly ordered. The child communicator might have reordered the ranks in the parent communicator to create a regular mapping between the ranks and AV indices. AV-Rankmap performs an additional scan of the ranks to determine whether an irregular model can be converted to a regular mapping model. After all the indirect mappings are processed and the rank-address mapping is created, AV-Rankmap scans the lookup table using the same algorithm as the first step.

2) Accessing the Rank-Address Mapping: In this section, we discuss how the rank-address mapping is accessed in AV-Rankmap. This translation is accessed inside the performance-critical path, and hence any overhead created in this path can slow practically every performance-critical operation in the MPI implementation. Therefore, we need to be particularly cautious with respect to the performance overheads of our implementation choices in this path.

The translation is done in three steps: (1) checking the mapping model of the communicator, (2) calculating the corresponding index in the AV table for the rank, and (3) accessing the AVE structure for the network address. To understand the overhead of our implementation, we use the Intel Software Development Emulator (SDE) [13] to obtain the instructions that are being executed for the translation. We studied three different implementations for such translation.

```
mov  rax, qword ptr [rbp+0x60]
mov  rdx, qword ptr [rip+0x43566d]
mov  rax, qword ptr [rax+0x188]
mov  eax, dword ptr [rax+r12*4+0xc]
shr  eax, 0x1
cdqe
mov  rax, qword ptr [rdx+rax*8+0x8]
```

Fig. 1: Rank-address translation: VC-VCRT.

```
cmp    dword ptr [rdi+0x1a8], 0xa
jnbe   0x435039
mov    eax, dword ptr [rdi+0x1a8]
jmp    qword ptr [rax*8+0x5cfcc8]
movsxd rax, esi
add    rax, 0x1
shl    rax, 0x4
add    rax, qword ptr [rip+0x4682c6]
mov    rax, qword ptr [rax]
```

Fig. 2: Rank-address translation: AV-Rankmap (*direct* mode).

The most intuitive implementation of the rank-address translation is using a `switch` statement where each case contains the translation code for a specific model. Figures 1 and 2 show the assembly code for the translation in VC-VCRT and AV-Rankmap. For a communicator with *direct* mapping, AV-Rankmap uses two additional instructions compared with VC-VCRT. For other mapping models, like *offset* and *stride*, there are additional instructions for calculating the index from the communicator rank. Note that the `jnbe` instruction in the switch statement is the branch to the default case. This is an unfortunate overhead because the AV-Rankmap approach does not have a "default" case and it is not possible to explicitly tell the compiler not to add this branch.

We also studied two additional implementations—one using hybrid `if-switch` statements and another using `goto` statements—to improve the instruction count for the network address lookup. However, neither approach yielded positive results compared with the `switch`-based implementation. Because of space limitations, we do not describe those two approaches in this paper, but their details can be found in [2].

IV. Evaluation

In this section, we experimentally compare VC-VCRT and AV-Rankmap from two perspectives: memory usage and performance. We used two different test platforms for our evaluation. The first platform was the Mira supercomputer at Argonne National Laboratory, which is a 49,152-node IBM BG/Q system. Each node on Mira has 16 cores and 16 GB memory, which allow running 768K processes at the full system scale. Most of our experiments were performed on Mira. However, the IBM BG/Q environment does not provide some capabilities such as MPI dynamic processes and special tools such as the Intel SDE, which is available only on Intel processors. For experiments that needed these capabilities, we used the Argonne "Blues" cluster. Each Blues node has two Intel Xeon E5-2670 processors (8 cores on each processor) and 64 GB memory. We ran experiments on Blues up to 256 nodes (4K processes). The baseline implementation for our comparison was MPICH 3.2, which uses VC-VCRT. The libraries and applications in all experiments were compiled by using GCC 4.7.2 with the `-O2` option and were statically linked.

A. Memory Usage for Split Communicators

In this experiment, we used a benchmark that splits the odd and even ranks of `MPI_COMM_WORLD` into two subcommunicators without reordering the ranks. Thus, the ranks in the split communicator have the *stride* mapping model in AV-Rankmap. This process is repeated a number of times in order to create a fixed number of split communicators.

Figure 3(a) shows the memory usage of 10 and 100 split communicators with up to 768K processes in the parent communicator. AV-Rankmap uses significantly less memory than does VC-VCRT. At the full scale on Mira, AV-Rankmap uses only 9 MB of memory for 100 split communicators. VC-VCRT, on the other hand, consumes more than 40% of system memory for 10 communicators and exceeds the total system memory for 100 communicators.

Figure 3(b) shows the breakdown of the memory usage for 10 communicators in AV-Rankmap and VC-VCRT with increasing numbers of processes in the parent communicator. We note that the memory usage of both approaches is $O(P)$ with respect to the total number of processes in the system: this is because both approaches need to store the network physical addresses, which takes $O(P)$ memory. But, AV-Rankmap has a memory usage advantage in two aspects. First, since the size of the network addresses used in AV-Rankmap is smaller (based on the implementation choices described in Section III-B), the constant associated with the $O(P)$ increase in memory is smaller. Second, for the rank-address mapping: since AV-Rankmap does not use a lookup table in common communicator patterns but instead dynamically computes the rank-address mapping (based on the implementation choices described in Section III-C), in the common case we use constant memory instead of an $O(P)$ structure, while VC-VCRT uses an $O(P)$ structure for the lookup.

Figure 3(c) shows a different breakdown of the memory usage, this time keeping the number of processes fixed at 768K but increasing the number of communicators created. As expected, the memory usage of VC-VCRT grows quickly with the number of communicators because it uses a new VCRT for each new communicator. In AV-Rankmap, on the other hand, each new communicator uses only a small constant memory space (e.g., to store the offset and stride, which are two integers) if its rank-address mapping is regular, as is the case in our benchmark.

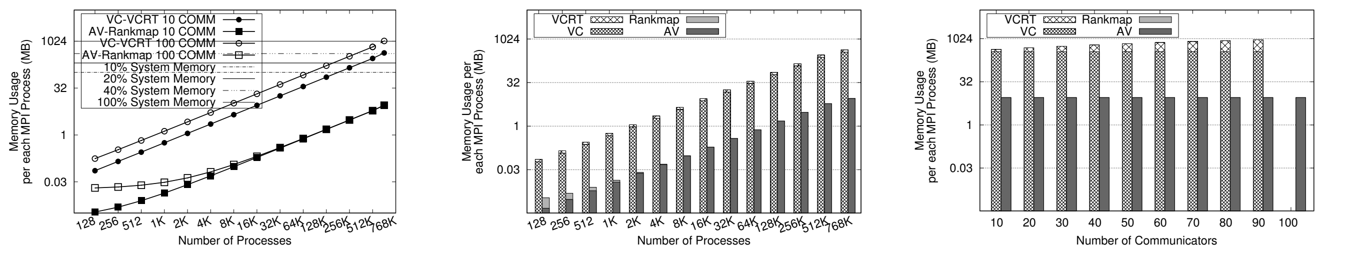
Aside from split intracommunicators, we have also studied the memory usage of duplicated intracommunicators, irregular intracommunicators, intercommunicators without dynamic processes (split and duplicated) and intercommunicators with dynamic processes. Because of space limitations, we do not present those results in this paper, but they can be found in [2].

B. Performance

We studied two performance aspects associated with AV-Rankmap: (1) communicator creation overhead and (2) network address lookup overhead.

As discussed earlier, communicator creation is not on the performance-critical path for most applications. Thus, while we do not want to make it too expensive, some overhead is typically acceptable. In our implementation, we noticed 3–8% overhead for communicator creation with AV-Rankmap. More detailed results for this part are omitted here because of space restrictions but can be found in [2].

Network address lookup, on the other hand, must show no observable overhead for the approach to be practically viable. As mentioned in Section III-C2, in order to look up the network address corresponding to a communicator rank, the MPI implementation must first check what mapping model that communicator is using and then use that information to either compute or look up the actual network address. To study the performance impact of AV-Rankmap on the network address lookup, we developed a microbenchmark that issues 1 million `MPI_Put` operations to each rank in the communicator in a round-robin fashion (message size of 8 bytes). We tested five mapping models: *direct*, *offset*, *stride*, *lut*, and *mlut*. All five communicators had the same number of ranks (half of the ranks in `MPI_COMM_WORLD`) to ensure that all experiments sent the same total number of messages. The *direct* communicator splits `MPI_COMM_WORLD` into two contiguous halves and performs the experiment on the first-half communicator. The *offset* communicator also splits `MPI_COMM_WORLD` into two contiguous halves but performs the experiment on the second-half communicator. The *stride* communicator splits



(a) Different numbers of processes (b) Detail memory usage of 10 communicators (c) Detail memory usage with 768K processes

Fig. 3: Memory usage of `MPI_Comm_split` with different numbers of processes and communicators.

TABLE III: Instruction Counts for Rank-Address Mapping.

	direct	offset	stride	lut	mlut
VC-VCRT	7	7	7	7	7
AV-Rankmap	9	11	13	11	15

`MPI_COMM_WORLD` into odd/even ranks and performs the experiment on the even half. The *lut* communicator is similar to the *stride* communicator but with the ranks reversed. The *mlut* communicator is created by merging ranks from a dynamically spawned group of processes. We studied the performance from three aspects: instruction-count, cache misses, and message issue rate.

Instruction-Count Analysis: For this analysis, we added trace points (i.e., markers) before and after the network address lookup and used Intel SDE to record the instructions executed between the two markers. As noted earlier, Intel SDE is available only on x86 architecture, so these experiments used the Blues cluster at Argonne. Table III shows the instruction counts for rank-address mapping with VC-VCRT (MPICH 3.2) and AV-Rankmap. AV-Rankmap adds 2–8 additional instructions compared with VC-VCRT. One aspect to note is that the *lut* mode, which uses a lookup table similar to VC-VCRT, is still four instructions more than VC-VCRT. The reason is that AV-Rankmap still needs to determine which mode the communicator is using and then branch using a `switch` statement to get to the actual lookup table implementation—thus adding the extra instruction overhead that we notice in the table.

Cache Analysis: As discussed in Section III-B, AV-Rankmap uses multiple techniques to compress the network address structure. As a result, the AVE in AV-Rankmap is significantly smaller than the VC in VC-VCRT (12 bytes vs 480 bytes). While AV-Rankmap costs additional instructions as discussed above, its smaller memory footprint significantly reduces the cache misses incurred during communication. In this section, we study this cache impact. We performed two sets of experiments. The first experiment used just two processes (one process per node) within each communicator, while the second experiment used 16 processes (eight processes per node) within each communicator. In our experiments, each process issued 8 million `MPI_Put` operations to each of the other ranks in the communicator in a round-robin fashion. We measured the cache misses using PAPI on Blues and BGPM on Mira. The cache was warmed up by using additional iterations of the same operation before our measurements.

Table IV shows the cache misses on Blues and Mira. We can see that VC-VCRT experiences significantly more misses

TABLE IV: Per-process cache misses.

Cache	VC-VCRT	AV-Rankmap				
		direct	offset	stride	lut	mlut
L1D (8 Procs, Blues)	120491	103	168	141	98	192
L2 (8 Procs, Blues)	237	47	43	69	53	60
L3 (8 Procs, Blues)	47	48	47	46	47	45
L1D (8 Procs, Mira)	180491	184	191	189	178	N/A
L2 (8 Procs, Mira)	422	57	53	61	68	N/A
L1D (2 Procs, Blues)	4438	102	157	133	82	168
L2 (2 Procs, Blues)	107	47	45	64	56	58
L3 (2 Procs, Blues)	48	46	43	45	46	47
L1D (2 Procs, Mira)	6540	186	189	190	180	N/A
L2 (2 Procs, Mira)	218	56	54	61	66	N/A

in L1D and L2 caches than does AV-Rankmap. The reason is the smaller memory footprint of AV-Rankmap. The cache lines on both Blues and Mira are 64 bytes. Thus, each VC object in VC-VCRT is spread across eight cache lines. One AVE object in AV-Rankmap, on the other hand, is only 12 bytes; and thus one cache line can fit multiple AVE objects.

With two processes, VC-VCRT experiences more than a ∼40-fold higher number of L1D cache misses and 2- to 4-fold higher number of L2 cache misses than does AV-Rankmap. This is a significant difference that is attributed to the smaller memory footprint of AV-Rankmap. With eight processes, the difference in cache misses goes up to nearly three orders of magnitude for L1D and ∼5-fold for L2. With a larger number of processes, because the microbenchmark issues `MPI_Put` to multiple ranks in a round-robin fashion, the cache impact is more pronounced.

Message issue rate: Here, we measure the message issue rate of VC-VCRT and AV-Rankmap for `MPI_Put`. We performed two experiments. The first experiment was using real networks on Blues (InfiniBand network) and Mira (BG/Q network), to show the performance impact of AV-Rankmap in practice. The second experiment was a worst-case measurement for a theoretical infinitely fast network; to emulate such a network, we went through the entire MPI stack but bypassed the actual network communication. This experiment was designed to help us understand how AV-Rankmap would behave on future, more-efficient, networks. Table V illustrates the message issue rate for both experiments; the top two lines correspond to the first experiment, and the bottom two lines correspond to the second experiment. We used two nodes and performed experiments with two processes in each communicator (one process per node) and 16 processes in each communicator (eight processes per node). All `MPI_Put` messages were sent to ranks on the remote node.

In the first experiment (real network), the message issue rate is bound by the network speed on the machine, and we

TABLE V: Per-process issue rate (million/second) with *switch*-based rank-address translation.

	VC-VCRT	AV-Rankmap				
		direct	offset	stride	lut	mlut
Mira (8 procs, real)	0.8999	0.8989	0.8994	0.8963	0.8969	N/A
Blues (8 procs, real)	4.325	4.338	4.334	4.322	4.324	4.318
Mira (8 procs, theoretical)	11.08	16.63	15.48	15.08	16.00	N/A
Blues (8 procs, theoretical)	64.76	98.79	92.13	89.64	96.03	84.51
Mira (2 procs, real)	0.8899	0.8879	0.8884	0.8863	0.8969	N/A
Blues (2 procs, real)	4.302	4.311	4.313	4.303	4.304	4.297
Mira (2 procs, theoretical)	13.08	16.63	15.48	15.08	16.00	N/A
Blues (2 procs, theoretical)	78.20	98.89	93.43	89.91	96.96	85.27

note that AV-Rankmap and VC-VCRT show identical issue rates for all communicator types. Hence, we conclude that AV-Rankmap can significantly improve memory usage without hurting communication performance.

In the second experiment (theoretical infinitely-fast network), the message issue rate is bound only by the CPU processing speed and the processing cost of MPI. In this case, AV-Rankmap outperforms VC-VCRT in all scenarios, achieving up to 50% higher maximum issue rate. The performance improvement is mainly due to the improved cache locality of AV-Rankmap. The difference in maximum issue rate between different mapping models reflects the cost of the extra work in the rank-address mapping. The measured maximum issue rates concur with the results on the instruction counts. That is, the *direct* model is expected to have the highest issue rate because it has the least overhead in the rank-address mapping. The maximum issue rate decreases as the instruction count increases from the *offset* model to the *mlut* model. Note that the performance difference between Mira and Blues is mainly because Mira's CPU has a lower frequency (1.6 GHz vs 2.7 GHz of Blues) and fewer integer units.

C. Applications

We evaluated AV-Rankmap with all the applications surveyed in Section II, but many of them follow similar trends so we do not show results for all of them in this paper. Instead, we show the results for the *Nek5000* application and several miniapps including *BT*, *FT*, and *SP*, from the NAS parallel benchmarks [3]; *AMG2013* and *Nekbone*, from the CORAL benchmarks [4]; and *AMR*, from the ExACT codesign center [6]. We conducted experiments to measure both performance and memory usage with VC-VCRT and AV-Rankmap. There was no observable difference in performance between the two approaches. Hence the performance results for the applications are not shown in this paper. Instead, we focus on the memory usage results. All applications were run on 512K processes of Mira.

We first evaluated Nek5000 with two problem sizes: medium and large. The medium-sized problem uses the XXT solver, which creates 24 duplicates of `MPI_COMM_WORLD`. The large-sized problem uses the AMG solver, which creates 86 duplicates of `MPI_COMM_WORLD`. Figure 4 presents the per-process memory consumption for the network address management functionality. On 512K processes, VC-VCRT uses around 250 MB per process, which is nearly 25% of the memory available on the node. With AV-Rankmap, on the other hand, the memory usage is negligible. The figure also shows the breakdown of the memory used by the VCs

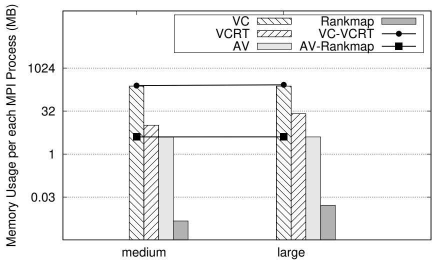


Fig. 4: Network address management memory consumption (Nek5000, 512K processes).

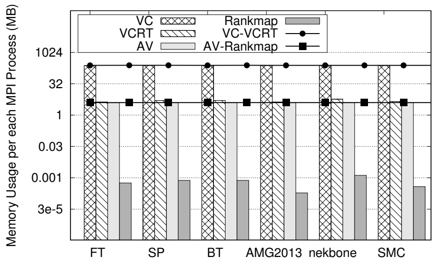


Fig. 5: Memory usage of miniapps on 512K processes.

and VCRTs. The size of the VC portion of memory usage is identical for both problem sizes because it depends on the number of processes, which is the same for both problem sizes, and not the number of communicators. The size of the VCRT portion of memory usage, on the other hand, depends on the number of communicators created, which is higher for the larger problem.

In Figure 5 we present the memory usage of the miniapps discussed above. Overall, most of the miniapps create between two and seven user communicators, a small number. Thus, the amount of memory used by the VCRTs, which depends on the number of communicators, is small. The amount of memory used by the VCs, on the other hand, depends on the number of processes and is the dominating factor in the overall memory usage. The memory usage of these miniapps on 512K processes is 244 MB ($\sim$25% of the system memory) with VC-VCRT. In contrast, AV-Rankmap uses only 4 MB memory on 512K processes.

V. Related Work

Several other works focus on reducing the memory usage of MPI communicators and groups [14], [15], [16]. In order to support the various possible group patterns, these approaches use complex models for storing the ranks in groups. While these approaches demonstrate good reduction in memory usage, they do so at the cost of high overhead in rank-address translation: this is a fundamental downside in these approaches. As mentioned earlier, in the performance-critical path, the overhead needs to be virtually invisible for the proposed approach to be viable. For example, Träff et al. use binary decision diagram for to compress the rank-address mapping [15]. It needs $O(\log p)$ time, where p is the number of processes, to translate a rank into its corresponding network address. Another example is the sparse group proposed in [14] which has been adopted as an optional feature in Open MPI. In this approach, the rank-address translation is from a child communicator to a parent communicator, rather than directly from a child communicator to the actual network address. Hence, as more communicators are created,

TABLE VI: Message Issue Rate (million messages/sec).

Communicator	Sparse Group	AV-Rankmap
`MPI_COMM_WORLD`	13	98.78
Gen 1	10.27	89.65
Gen 2	9.26	89.63
Gen 3	8.41	89.65
Gen 4	7.57	89.64

the library needs more time to iteratively traverse the tree of the ancestor communicators for network address lookup, thus significantly degrading performance. For a more quantitative comparison, we performed a case study on the performance of the sparse groups approach. In this comparison, we first performed an odd/even split on `MPI_COMM_WORLD` to create a first-generation split communicator. Then we derived the next-generation split communicator by splitting the previous-generation communicator in the same odd/even manner. Therefore, all split communicators have a stride mapping model. Table VI shows the maximum message issue rate on different generations of split communicators using Open MPI with sparse groups and AV-Rankmap. Because of the iterative traversal of the tree of the ancestor communicators, Open MPI with the sparse group implementation has around 10% performance loss for each additional generation of communicators—with as few as four generations of communicators, this adds up to nearly 40% performance overhead compared with that of the `MPI_COMM_WORLD`. On the other hand, AV-Rankmap achieved consistent on all four generations of split communicators. A more recent study proposed to exploit the topology of Cartesian communicators for compressing the rank-address mapping [16]. In order to use this technique on non-Cartesian communicators, the MPI library has to maintain virtual topologies which forces the rank-address translation being done by expensive topological conversion even for the simplest *direct* communicator.

Other researchers have proposed approaches to distribute the table of ranks among multiple processes [17], [18], [19]. Sack and Gropp [19] proposed a distributed algorithm for ordered communicator construction that uses $O(n/p)$ memory by using distributed tables for storing the ranks. However, such a design also leads to nonconstant time for translating a rank, and is thus unfit for a high-performance MPI implementation. Recent work by Moody et al. [20] mentions a generalized `MPI_Comm_split`. They propose creating and storing process groups as chains in $O(1)$ memory and $O(\log n)$ construction time. They perform collectives by exchanging appropriate process ids during the operation. Because of the distributed nature of the lookup table, however, some rank-process translations need additional communication, which adds significant performance overhead. Moreover, as we demonstrated in our case study, the lookup table is not always necessary.

A more successful communicator memory compression technique used by MPICH, MVAPICH, Intel MPI, and many other MPI implementations was proposed by Goodell et al. [21]. This approach allows duplicate communicators to share the same VCRT as their parent, thus removing multiple copies of the same VCRT. While this approach eliminates the need for VCRTs in some limited cases, however, it is not applicable to most cases. Even for simple duplicate communicators, internal communicators used inside MPI are not direct duplicates and thus cannot use this approach. Nevertheless, this approach is widely used in many MPI implementations and is, in fact, the baseline case that we compare against in this paper.

Compared with previous studies, AV-Rankmap has four advantages: (1) it eliminates the need for a lookup table for the majority of use cases; (2) it uses a simple process mapping model that avoids the overhead of complex mapping techniques and maintains a constant time complexity for rank-address translation; (3) it tackles compression in both the network address and rank-address mapping structures; and (4) it performs such memory compression with no practically observable performance degradation.

VI. Concluding Remarks

In this paper we proposed a new mechanism, called AV-Rankmap, for network address management in MPI. AV-Rankmap detects patterns in rank-address mapping that applications naturally tend to have, as well as the fact that some parts of the network address structures are naturally more performance critical than others. It uses this information to compress the network address management structures. We demonstrated that AV-Rankmap significantly reduces the memory usage of communicators on large-scale systems with no practically observable degradation in performance.

References

[1] P. Balaji, et al. MPI on a Million Processors. *Parallel Processing Letters*, 21:45–60, 2011.

[2] Y. Guo, et al. Memory Compression Techniques for Network Address Management in MPI (Extended Preprint Version). In *Preprint, ANL/MCS-P6051-0916, http://www.mcs.anl.gov/~yguo/pubs/ANL-MCS-P6078-1016.pdf*, 2016.

[3] NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html.

[4] CORAL Benchmarks. https://asc.llnl.gov/CORAL-benchmarks.

[5] Center for Exascale Simulation of Advanced Reactors. https://cesar.mcs.anl.gov.

[6] Center for Exascale Simulation of Combustion in Turbulence. https://cesar.mcs.anl.gov.

[7] M. Valiev, et al. NWChem: A Comprehensive and Scalable Open-Source Solution for Large Scale Molecular Simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.

[8] Nek5000. https://nek5000.mcs.anl.gov.

[9] QBOX. http://computation.llnl.gov/projects/qbox-computing-structures-quantum-level.

[10] M. Si, et al. Casper: An Asynchronous Progress Model for MPI RMA on Many-Core Architectures. In *IPDPS*, pages 665–676, 2015.

[11] OpenFabrics Interfaces (OFI). https://ofiwg.github.io/libfabric/.

[12] P. Shamis, et al. UCX: An Open Source Framework for HPC Network APIs and Beyond. In *HotI*, 2015.

[13] Intel Software Development Emulator. https://software.intel.com/en-us/articles/intel-software-development-emulator.

[14] M. Chaarawi et al. Evaluating sparse data storage techniques for MPI groups and communicators. In *ICCS*, 2008.

[15] J. Träff. Compact and Efficient Implementation of the MPI Group Operations. In *EuroMPI*, 2010.

[16] Space Performance Tradeoffs in Compressing MPI Group Data Structures. In *EuroMPI*, 2016.

[17] H. Kamal, et al. Scalability of Communicators and Groups in MPI. In *HPDC*, 2010.

[18] H. Kamal et al. An Integrated Fine-Grain Runtime System for MPI. *Computing*, 96(4):293–309, 2014.

[19] P. Sack et al. A Scalable MPI_Comm_Split Algorithm for Exascale Computing. In *EuroMPI*, 2010.

[20] A. Moody, et al. Exascale Algorithms for Generalized MPI_Comm_Split. In *EuroMPI*, 2011.

[21] D. Goodell, et al. Scalable Memory Use in MPI: A Case Study with MPICH2. In *EuroMPI*, 2011.