# Improving MPI Reduction Performance for Manycore Architectures with OpenMP and Data Compression

Hongzhang Shan, Samuel Williams
*Computational Research Division*
*Lawrence Berkeley National Laboratory*
*Berkeley, CA 94720, USA*
{*hshan, swwilliams*}*@lbl.gov*

Calvin W. Johnson
*Department of Physics*
*San Diego State University*
*San Diego, CA 92182*
*cjohnson@sdsu.edu*

***Abstract*—MPI reductions are widely used in many scientific applications and often become the scaling performance bottleneck. When performing reductions on vectors, different algorithms have been developed to balance messaging overhead and bandwidth. However, most implementations have ignored the effect of single-thread performance not scaling as fast as aggregate network bandwidth. In this work, we propose, implement, and evaluate two approaches (threading and exploitation of sparsity) to accelerate MPI reductions on large vectors when running on manycore-based supercomputers. Our benchmark results show that our new techniques improve the MPI_Reduce performance up to 4× and improve BIGSTICK application performance by up to 2.6×.**

## I. Introduction

When parallelized, many numerical methods require dot products or norm calculations that give rise to global reductions on scalar variables. More challenging, multidimensional parallelization often results in broadcasts or reductions on large vectors of size $N/\sqrt{P}$ (i.e. megabytes to gigabytes). Whereas in the former, reductions on scalars are often latency-limited, reductions in the latter can be particularly bandwidth-intensive.

The MPI collective operations `MPI_Reduce` and `MPI_Allreduce` succinctly express the requisite functionality and are widely used in many scientific and engineering applications due to their ease of use, high performance, and portability. A five-year application profiling study [13] showed that more than 40% of application MPI time is spent in these two functions.

In the manycore era, memory capacity is limited, single-thread performance is sacrificed, while core concurrency on a chip is maximized. In practice, the memory capacity constraint can restrict many challenging applications to only a few MPI processes per manycore node — far fewer than the available number of cores or threads. Concurrently, high parallel concurrency and low single-thread performance can conspire to result in MPI collectives becoming performance scaling bottlenecks [7], [16]. As a consequence, when `MPI_Reduce` or `MPI_Allreduce` are called, many cores are simply idle waiting for the reductions to finish, wasting the precious computing resources. To illustrate this, Figure 1 shows the average `MPI_Reduce` run times for two MPI processes on a pair of manycore (KNL) nodes at NERSC. As we vary the number of OpenMP threads per MPI process from 1 to 256 and the vector size for 1K, 1M, and 200M single-precision floating-point numbers (floats), we observe that the `MPI_Reduce` time remains roughly constant with thread concurrency while increasing with vector size (albeit slowly at first). A run time of about 530 milliseconds for 200M singles (0.8GB) implies a bandwidth of only 1.5GB/s — far less than the Aries injection bandwidth.

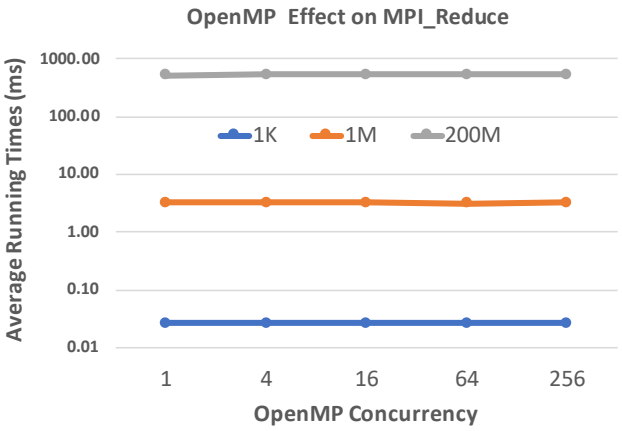


Figure 1. The average running times of `MPI_Reduce` for 2 MPI processes, one per node. The OpenMP has almost no performance benefit at all. The bandwidth for 200M single-precision floats is only about 200M*4 / 530ms = 1.5GB/s, far less than the Aries injection bandwidth.

In this paper, we present two techniques to accelerate MPI reductions running on manycore nodes. First, we exploit the idle threads on a manycore node in order to accelerate the local reduction computations in `MPI_Reduce`. To that end, we augmented the MPICH [19], [11] algorithms with OpenMP by partitioning the local reduction work among OpenMP threads. This approach is different from other popular optimization approaches that focus on latency and

bandwidth optimization. Instead, we focus on local (on-node) work. Our results show that `MPI_Reduce` begins to benefit from our OpenMP optimizations when the message size exceeds 8K single-precision floating-point numbers (i.e. reductions on vectors). Moreover, as we increase the vector size, we observe a larger benefit. For large messages of 128K single-precision floating-point numbers and beyond, our new algorithm delivers up to a 4× performance speedup — largely due to OpenMP parallelization and avoiding local (superfluous) copies.

Motivated by the observation that, at high concurrency, the local vectors injected into a reduction might be mostly zero, we also examined the performance benefit from data compression. To that end, we developed two straightforward algorithms that exploit sparsity to compress the data for `MPI_Reduce`. One approach (*Idx*) compresses the non-zero array elements into a new data array and uses an additional array to store the indices of the non-zero elements. Our second approach (*Bits*) eliminates all zero elements from the payload array but uses a bit mask array to indicate whether the corresponding array element is nonzero. We implemented both approaches using Intel intrinsics to vectorize the packing/unpacking on KNL. We found that if the zeros follow a uniform data distribution the data sparsity should exceed 30% to overcome the packing and unpacking overhead for the *Bits* algorithm, while for the *Idx* algorithm, the sparsity needs to be above 90%.

To demonstrate the efficacy of our techniques, we integrated our newly developed `MPI_Reduce` algorithms into the application BIGSTICK [2], [6], [7], whose scaling performance is very sensitive to the `MPI_Reduce` performance. BIGSTICK's dominant computation is essentially a 2.5D parallelized sparse-matrix vector multiplication (although the compression of the matrix is far more complex than CSR) inside a Lanczos eigensolver. Such methods necessitate a `MPI_Reduce` to reduce per-process partial sums of vector data to the root (diagonal) process. In practice, the data volume for `MPI_Reduce` does not change with the number of MPI processes. As such, when more processes are used, although the time in local matvec decreases, the time spent in the `MPI_Reduce` operation increases to the point where it is the bottleneck. Our results show that our new algorithms improve overall matvec's performance by up to 2.6×. Interestingly, data compression did not provide a noticeable performance benefit. Further profiling revealed that this is due to a wildly uneven distribution of nonzeros among processes. Although most processes inject vectors that were only 10% nonzero, a few processes in each communicator injected vectors that were more than 90% nonzero (essentially dense). Note, the resultant vector ejected to the root is always dense. These processes slowed down the whole `MPI_Reduce` operations.

The rest of the paper is organized as follows. The related work is discussed in Section II. In Section III, we described the three implementations of `MPI_Reduce` and their differences with the corresponding MPICH implementation. Two data compression implementations are also been described in this section. Next, we describe about our experimental platform in Section IV and the performance results are analyzed in Section V. The performance effects on application BIGSTICK are examined in Section VI. Finally, we summarized our results and discussed the future work in Section VII.

## II. Related Work

The related work can be divided into two categories, performance optimization and data compression.

Optimizing the performance of MPI collective reduction operations has been an active research topic. Thakur and Gropp described the algorithms used by MPICH [20], [19], [12]. Some specific implementations on the IBM BlueGene/Q platform have been discussed in [10]. Kandalla et al. discussed how to develop the topology-aware algorithms for Infiniband clusters [8] and the MIC clusters [9]. For the Aries network used on the Cray XC30 platform, Jain et al. [5] found that using random job placement can often avoid hot-spots and deliver better performance. Faraj et al. developed a scheme that will automatically generate topology specific routines and use an empirical approach to select the best implementations [3]. Rabenseifner et at. discussed some optimizations for non-power-of-two number of processes [14]. Some research has also been done developing collective communication algorithms for clusters of SMPs [21], [17].

Our approach differs from these approaches mainly in that we focus on optimizing local reduction work instead of latency and bandwidth. Furthermore, our OpenMP strategy is platform independent.

Another category is data compression, which can be used to optimize the `MPI_Reduce` and `MPI_Allreduce` performance by reducing the communication volume for sparse data. Hofmann et al. used a Run Length Encoding (RLE) scheme and directly incorporated it into the reduction process [4]. Traff et al. discussed how to exploit algebraic (neutral element) properties for compression and computation natural for MPI reduction operations [22]. Renggli et al. generalized standard collective operations by allowing processes to contribute sparse input data vectors [18]. A set of communication efficient protocols for sparse input data have been designed and implemented. We developed two different, straightforward compression schemes, *Idx* and *Bits*, which have been fully vectorized and parallelized by OpenMP to reduce the packing/unpacking overhead.

## III. Algorithms and Implementation

Our implementation follows MPICH [11] using a binomial tree algorithm for small vectors and the reduce-scatter algorithm for large vectors. In addition, we implemented the

ring algorithm. All the communication is carried out by MPI point-to-point messaging using `MPI_Send`, `MPI_Recv`, `MPI_Wait`, and `MPI_Sendrecv`. For better performance, all these point-to-point functions can be replaced by more efficient, lower-level communication API calls. The main differences between our implementation and MPICH are that 1) we avoid the *MPIR_Localcopy* by explicitly designating which data buffer to use in the later stages, and 2) OpenMP parallel regions are employed to perform the local reductions. When data compression is enabled, OpenMP regions are also used for data compression and decompression.

### A. Cost Model

Before we jump to the details of different algorithms, we first describe a simple cost model that we use to understand the performance characteristics of the various algorithms. We use a similar model as the one used in [20], which assumes the cost to send a message from one process to another is $\alpha + N\beta$, where $\alpha$ is the message latency, $\beta$ is the transfer time per float (inverse bandwidth), and $N$ is the message size in floats. The model also assumed that these parameters are independent of how many pairs of processes are communicating with each other, independent of the distance between the communicating nodes, and that the communication links are bidirectional. In addition, we add a $\gamma$ cost per float to incorporate the cost of the local reduction on each process — a key component given the discrepancy between single-thread performance and aggregate network bandwidth. Finally, we assume that there is no overlap between local computation and communication.

### B. Binomial Tree (BTree)

The binomial tree algorithm is simple. Based on the bit representation of the rank relative to the root, starting from the right, if the bit is 1, one sends data to the process with the corresponding bit cleared. If the bit is 0, it will receive data from the process with that bit set as long as that process exists. It then performs the reduction operation between its its own data and the received data. This process will repeat $logP$ steps where $P$ is the total number of processes involved in the communicator. Based on our simple model, the estimated cost is $logP \times (\alpha + N\beta + N\gamma)$.

### C. Reduce-Scatter (RedScat)

The cost model for the binomal tree indicates that when the message size $N$ becomes large, both the bandwidth term $N\beta$ and the local reduction term $N\gamma$ could easily outweigh the latency term $\alpha$ and become the dominant performance factors. The motivation behind the reduce-scatter algorithm is to reduce the bandwidth cost term $logP \times N\beta$.

The reduce-scatter algorithm is divided into two phases — reduce-scatter followed by the gather operation. The reduce-scatter phase is illustrated in Figure 2 with $P = 4$. The vector data is divided into $P$ blocks on all processes.

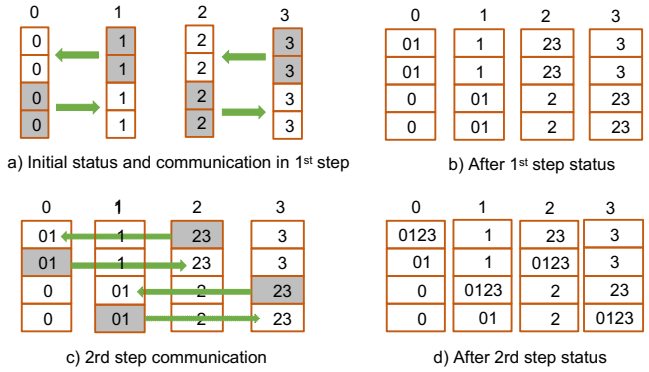


Figure 2. The illustration of reduce-scatter phase for the *RedScat* algorithm when P=4. The numbers in the block represent the process ranks whose data has already been reduced for that block. The shaded blocks represents the communicated data. Each process holds one block of the final results.

During the first step, each process will send $N/2$ data to the process whose rank differs in only the least significant bit. After receiving the data, a local reduction operation is performed. During the next step, each process will send $N/4$ data to the process whose rank differs in only the second bit. This process will repeat $logP$ steps. This is a standard vector halving and distance doubling procedure. Finally, each process will own one block of the final results. A gather operation is needed at the end to gather all the results for the root. The estimated cost for reduce-scatter phase is $\alpha logP + \frac{P-1}{P}N\beta + \frac{P-1}{P}N\gamma$. and for the gather phase is $\alpha logP + \frac{P-1}{P}N\beta$. So the total cost is $2\alpha logP + 2\frac{P-1}{P}N\beta + \frac{P-1}{P}N\gamma$. Compared with the binomial tree, the latency term doubled but the bandwidth and local reduction terms could be reduced substantially when $N$ is large.

If $P$ is not a power of 2, extra steps are needed for the reduce-scatter phase. We will pair some of the processes so that the total number of processes participating in the reduce-scatter phase will be the nearest lower power of two. For each pair, each process will be responsible for performing the reduction for half of the data. Then, one process will send its data to its partner and wait for the whole reduce-scatter phase to finish. Only its partner will participate in the following reduce-scatter phase. The extra cost is $2\alpha + N\beta + \frac{1}{2}N\gamma$.

### D. Ring

Similar to the reduce-scatter algorithm, the *ring* algorithm also has two phases — the reduce-scatter phase and the gather phase. However, during the reduce-scatter phase, all the processes will form a ring and only communicate with their neighbors. One process will send the data to its left neighbor and receive data from its right neighbor. The data is also divided into $P$ blocks. Figure 3 illustrated the whole

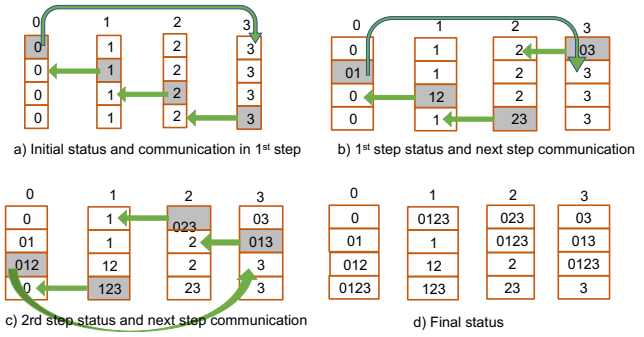


Figure 3. Illustration of the *ring* algorithm for P=4. The numbers in the block represent the process ranks whose data has already been reduced. The shaded blocks represents the communication data. Each process holds one block of the final results. Note that by controlling which block to start with in the first step, we can control which final block will be left on the root.

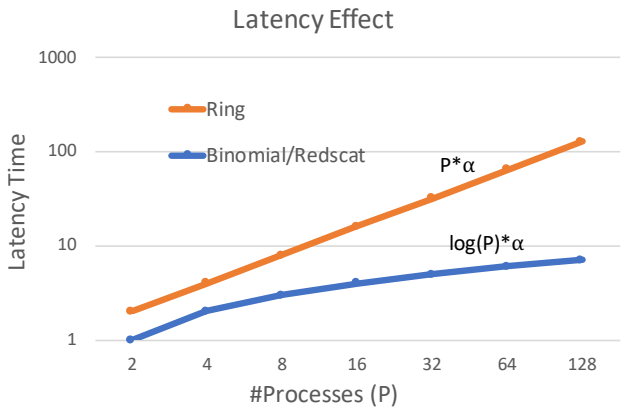


Figure 4. The relationship between latency and number of processes $P$ for our three algorithms. Observe the log scale and the superiority of the *binomial* and *redscat* algorithms at high concurrency.

process with $P = 4$. The whole process will repeat $P$ steps instead of $logP$. The estimated cost for the reduce-scatter phase is $P\alpha + \frac{P-1}{P}N\beta + \frac{P-1}{P}N\gamma$. And the total cost with the gather stage is $(P + logP)\alpha + 2\frac{P-1}{P}N\beta + \frac{P-1}{P}N\gamma$. Compared with the reduce-scatter algorithm, the latency term becomes much larger and could become the performance bottleneck when $P$ is large. However, its implementation is much simpler and the needed intermediate buffer is much smaller ($N/P$ vs. $N/2$ in reduce-scatter). Furthermore, it does not mandate $P$ be a power of 2.

### E. Performance Characteristics

Figure 4 shows the relationship between the latency cost and the number of processes $P$ for our three algorithms. For the *Ring* algorithm, its latency cost scales linearly with $P$ while for the other two algorithms, the cost only scales as $log(P)$. The bandwidth cost relationship is shown in

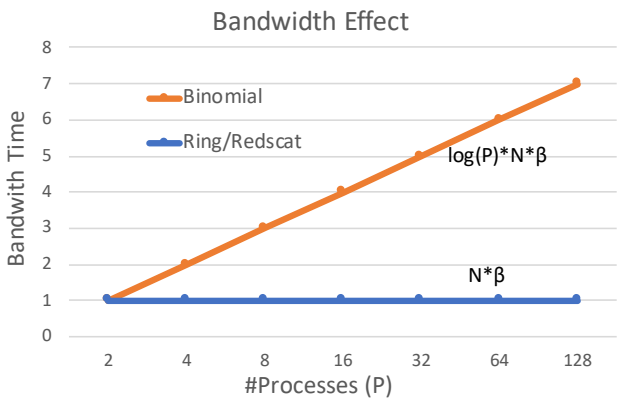


Figure 5. The relationship between bandwidth cost and number of processes $P$ and data size $N$. For large $N$, this bandwidth term should dominate. Observe the log scale and the superiority of the *ring* and *redscat* algorithms at high concurrency.

Figure 5. The *Binomial* algorithm has $log(P)$ times higher cost than the other two. The local reduction cost is similar to bandwidth. Theoretically, the *redscat* algorithm should perform the best of the three. Nevertheless, due to the simplicity of our cost model, the actual running cost may be different, especially for the cases when $P$ is not a power of 2 and extra phases are needed. However, none of these algorithms address how to improve the local reduction time ($\gamma$ term). This motivates us to introduce OpenMP to parallelize and accelerate local reduction time (drive the $\gamma$ term down by `OMP_NUM_THREADS`) and ensure that single-thread performance is not an impediment to network bandwidth.

### F. Data Compression

We also consider the algorithms for sparse data cases and implemented two approaches to reduce the volume of the communication data. One approach (*Bits*) is to pack the nonzero array elements into a new data array and use an auxiliary array of 1 bit per uncompressed element to indicate whether the corresponding array element is a nonzero. Suppose, we have a float (4 bytes per element) array of size $N$ with nonzero ratio of $R$. This packing could reduce the total communication volume to $4RN + N/8$ bytes. The corresponding overhead is the time to pack and unpack the data at sender and receiver side in each stage of the distributed memory reduction algorithm. To maximize performance, we use Intel intrinsics to vectorize the packing/unpacking operations.

Another approach (*Idx*) is to pack the non-zero array elements into a new data array and use an auxiliary array to capture the corresponding indices of the nonzero elements. For this approach, the communication volume would be $8RN$ (32-bit values plus 32-bit indices). Based on the

received indices of the nonzero elements, the local reduction could be performed using scattered indirect data access.

## IV. Experimental Platform

We evaluate performance on a HPC platform named Cori installed at NERSC. Cori consists of two partitions, one with Intel Xeon "Haswell" (HSW) processors, the other with Intel Xeon Phi "Knights Landing" (KNL) processors. Both partitions share the same Cray Aries global interconnect. In this paper, we use only the KNL partition for all our experiments.

Each KNL node contains a single, self-hosted Intel Xeon Phi processor with 68 out-of-order superscalar cores running at 1.4GHz. Each core has a 32KB L1 data cache, two 512-bit vector units, and four hardware threads. Each tile (2 cores) includes a 1MB L2 cache. The node contains 16GB of MCDRAM and 96GB DDR4-2133 memory providing about 350GB/s of MCDRAM and 80GB/s of DDR bandwidth. The MCDRAM is configured as a direct mapped L3 cache and the directory is configured for quadrant mode (*quadcache*).

The programming environment includes the following modules:

- `PrgEnv-intel/6.0.4`,
- `MPI version : cray-mpich/7.7.0`,
- `Compiler : intel/18.0.1.163`.

In addition, when performance is measured, the following environment variables are set to enable DMAPP communication.

- `MPICH_NEMESIS_ASYNC_PROGRESS=MC`
- `MPICH_MAX_THREAD_SAFETY=multiple`
- `MPICH_USE_DMAPP_COLL=1`

Without enabling DMAPP communication, the performance of MPI_Reduce function would be significantly lower. Finally, environment variable `OMP_NUM_THREADS` is used to control the number of OpenMP threads per MPI process.

## V. Performance of Microbenchmarks

In this section, we will investigate the performance of our `MPI_Reduce` implementations on KNL against the default Cray MPICH implementation version `cray-mpich/7.7.0` with DMAPP and quantify the benefit of threading. We start from the large data cases, then move on to the smaller data sets. Finally, we discuss the additional benefit from data compression.

### A. Performance for Large Data Sets

We collected the performance of different algorithms with fixed vector size of 200M single-precision floating points (floats) as this is representative of the vector sizes in the BIGSTICK application. Figure 6 shows the scaling results for 2 to 512 MPI processes. To quantify the benefits of OpenMP more clearly, we always run with one MPI process per node. This orthogonalizes these forms of parallelism and ensures there is no subtle tradeoff between MPI parallelism and OpenMP parallelism. The best Cray MPICH implementation is labeled *Cray*, while our three algorithms are labeled as *Binomial*, *Redscat*, and *Ring*.

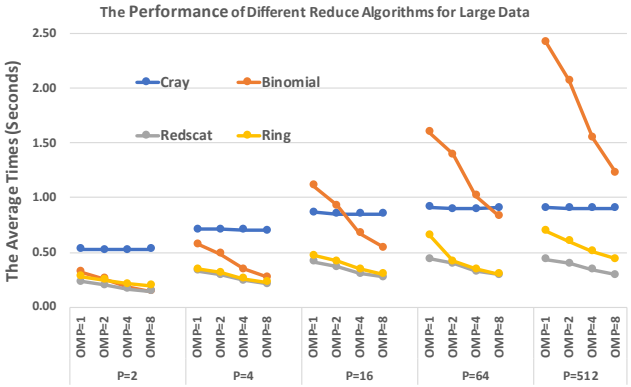


Figure 6. MPI_Reduce times on KNL as a function of algorithm, MPI concurrency (one process per node), and OpenMP parallelization for N=200M floats. Our *Redscat* and *Ring* algorithms deliver superior performance compared to Cray's implementation with 1 thread and see even greater speedup at high thread parallelism.

Although the array size remains constant with an increasing number of MPI processes, the average total running time increases. For *Redscat* and *Ring* algorithms, the increased times are mainly due to their latency cost, which increases $logP$ and $P$ times separately. The bandwidth and local reduction cost also increase slightly with $\frac{P-1}{P}$ times. For *Binomial* algorithm, all three terms increase $logP$ times, causing its running times jump sharply higher with larger $P$.

Observe that OpenMP has no performance effect on the Cray implementation across all MPI concurrency. Conversely, OpenMP can improve `MPI_Reduce` time for all three of our algorithms. As the *binomial* implementation can become dominated by a large $logP\gamma$ term, the benefit of OpenMP is much more pronounced with performance scaling almost linearly with thread concurrency.

In order to understand performance and scalability, we breakdown `MPI_Reduce` time at 512 processes for different algorithms and thread concurrency. To that end, Figure 7 breaks time into four components:

- *Comm*: the MPI time to send and receive data between processes
- *Local*: the time to perform the local reduction
- *Gather*: the MPI time to gather final data blocks into root process (not applicable to *binomial*)
- *Other*: the rest of the running time

We depend on HPCToolkit [1] for profiling the *Cray* implementation results as we cannot manually add timers to Cray's implementation as we did for our implementations. The `MPIC_Sendrecv` time is tabulated under *Comm* time, the `MPIR_SUM time` is tabulated under *Local* time, and

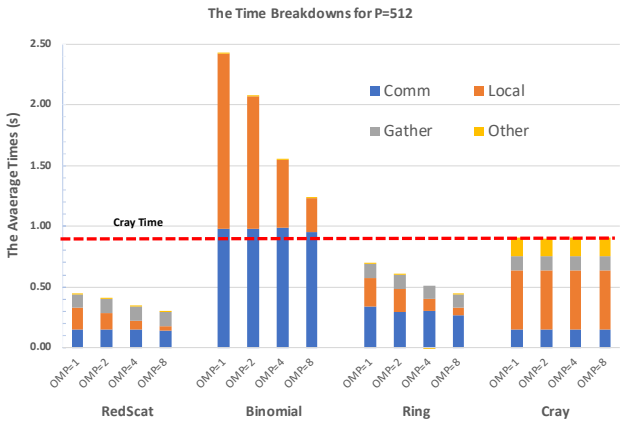


Figure 7. The time breakdown for different reduction algorithms with 200M element array sizes using 512 processes (one per node) and 8 OpenMP threads per process. The horizontal line is the running time of the *Cray* implementation. Observe the high MPI time in the *Binomial* implementation and the large and constant *local* time in the *Cray* implementation.

the `MPIC_Send` and `MPIC_Recv` times are tabulated under *Gather* time as they are used only in the gather phase.

There are a couple of key observations. First, *local* reduction time is substantial (dominate for *Binomial*) on KNL. As such, improving the performance of the local reduction is imperative. Using multiple OpenMP threads significantly reduces the local reduction times for *Binomial*, *Ring*, and *RedScat* algorithms, while it shows no benefit for the *Cray* implementations where HPCToolkit showed local reduction time remained constant across with respect to OpenMP concurrency.

Second, even without OpenMP parallelism (OMP=1), the *RedScat* and *Ring* algorithms clearly outperform the *Cray* implementation. This is mainly due to the following two reasons. First, the *Cray* implementation has much higher *Local* reduction time, indicating that our highly-tuned, vectorized local reduction is much more efficient. Additionally, the *Cray* implementation also has much higher *Other* time. HPCToolkit profiling results tells that this is caused by the `MPIR_Localcopy` function. In MPICH, this function is used to copy data from sendbuf to recvbuf at the start of the `MPI_Reduce` function. In our implementation, this copy has been removed by explicitly designating whether the sendbuf or recvbuf will be used in the later stage.

The third observation is that while our *RedScat* and *Ring* implementations perform better than the *Cray* implementation, our *Binomial* implementation performs worse due to its high local reduction and data movement times (both scaled by $logP$ in the performance model).

Finally, we observe that increasing OpenMP parallelism beyond 8 threads (total of 8 cores per node) will deliver asymptotic improvements to *RedScat* performance as it is dominated by communication. Further performance improvements can only be through increasing the number of cores driving the network. Nevertheless, as shown in Table I, the *RedScat* implementation delivers the best performance attaining a 2.1-3.7× speedup over the best *Cray* implementation.

Table I
THE SPEEDUP OF OUR *RedScat* IMPLEMENTATION RELATIVE TO THE BEST *Cray* IMPLEMENTATION FOR A 200M VECTOR SIZE AS A FUNCTION OF MPI (NODE) AND OPENMP (THREAD) CONCURRENCIES.

| MPI= | OMP=1 | OMP=2 | OMP=3 | OMP=4 |
|---|---|---|---|---|
| 2 | 2.3× | 2.6× | 3.2× | 3.7× |
| 4 | 2.2× | 2.4× | 2.9× | 3.3× |
| 16 | 2.1× | 2.3× | 2.8× | 3.1× |
| 64 | 2.1× | 2.3× | 2.7× | 3.1× |
| 512 | 2.1× | 2.3× | 2.7× | 3.0× |

### B. Performance for Small and Medium Data Sets

While it is clear that using more OpenMP threads can accelerate the local reduction operation for large vector sets, we are also interested in determining at what vector size the performance starts to see a benefit from using OpenMP threads. We measured the performance of all four algorithms for vector sizes ranging from 4 to 16M floats. Figure 8 shows our best speedup relative to the *Cray* implementation as a function of MPI concurrency (one process per node) and vector size (OpenMP is fixed at 8 threads). Note, we prefix each entry with a letter to represent the best algorithm (R=*RedScat*, C=*Cray*, B=*Binomial*, G=*Ring*).

For small arrays up to 2K floats, the Cray implementation usually delivers the best performance. This is probably due to the fact that the Cray implementation uses more efficient lower-level communication APIs instead of the point-to-point MPI functions that have much higher overhead. When vector sizes exceed 512K floats, the *Local* reduction times become more important. As such, the *RedScat* algorithm starts to perform best and deliver a 2-3× speedup over the *Cray* implementation. There exist the transition areas from small arrays to big arrays, where the *Binomial* algorithm delivers the best performance. Figure 9 displays the time breakdowns for 128K floats data size using 16 MPI processes (one per node) and eight OpenMP threads. Unlike Figure 7, the times are shown in microseconds instead of seconds. The main reason the *Binomial* algorithm outperforms the *RedScat* in this region is that it eliminates the relatively expensive gather stage.

### C. Performance Effects of Data Compression

Another approach to improving `MPI_Reduce` performance is to compress the data to reduce the communication volume when the reduced data is sparse (most injected vector elements are 0.0). We examine performance of two straightforward algorithms: the *Bits* and the *Idx* algorithms.

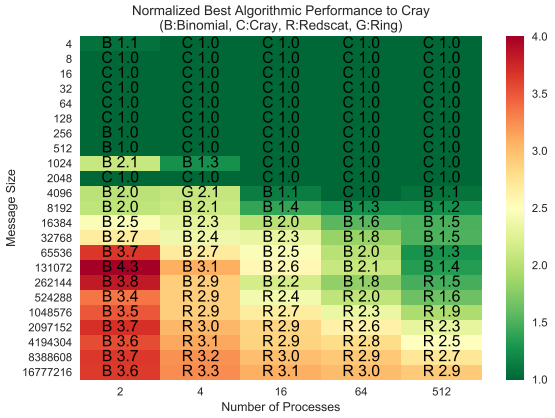


Figure 8. Normalized best algorithmic performance using the Cray implementation as the baseline for eight OpenMP threads. Note, one process per node, 8 threads per process, "C"=*Cray*, "B"=*Binomial*, "G"=*Ring*, and "R"=*RedScat*.

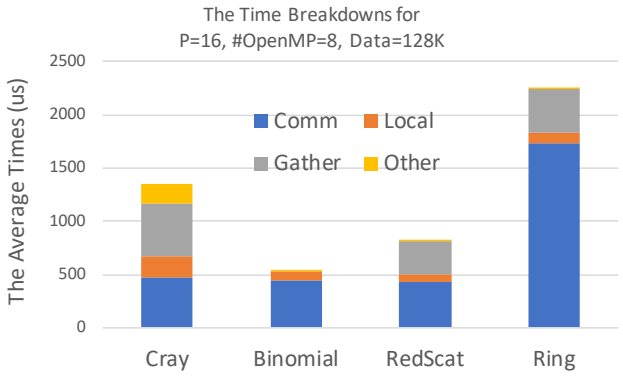


Figure 9. The time breakdowns of different reduction algorithms for 128K floats data size when using 16 processes (one per node) and 8 threads per process.

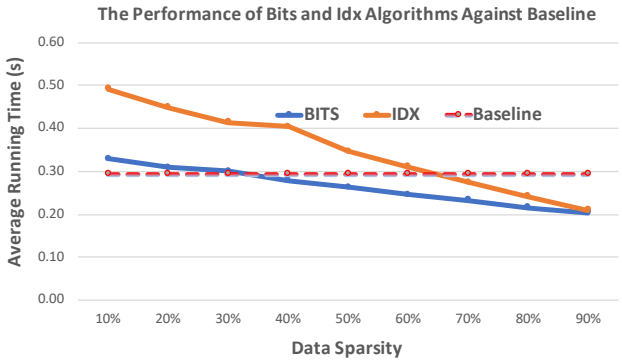


Figure 10. The performance of *Bits* and *Idx* algorithm for different data sparsity when P=64 and OpenMP=8 compared to the best (*RedScat*) baseline implementation without compression.

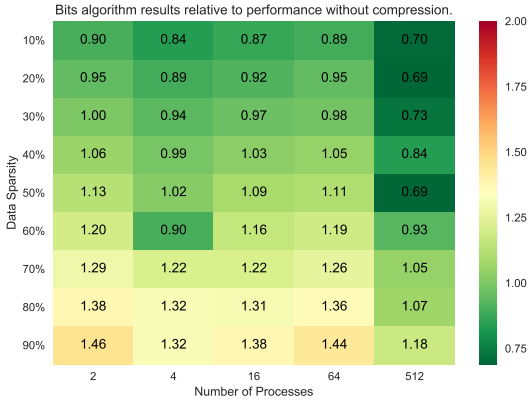


Figure 11. Performance of the *Bits* implementation relative to no compression as a function of sparsity and concurrency (one process per node, 8 threads per process). Observe that to break even (1.0), one needs increasing sparsity with increasing concurrency.

Figure 10 shows the average running times of the *RedScat* algorithm, where the vector data is compressed using either the *Bits* or *Idx* algorithms. In both cases, we use a uniform distribution of nonzeros with sparsity (measured upon injection) ranging from 10% to 90% on vectors of 200M floats. When the data sparsity is low (left) and the data is mostly nonzero (dense vectors), the *Bits* approach outperforms *Idx* due to less data being needed in the auxiliary data array. As the sparsity increases (right) and the vectors become mostly zero, the difference of these two algorithms shrinks. The *Bits* algorithms needs one bit for every elements, so the auxiliary array size is $N/8$ bytes while the index array size is $4N \times (1 - Sparsity)$ for the *Idx* algorithm, where $N$ is the array size. The data sparsity needs to be higher than 97% for Idx algorithm to require smaller auxiliary size.

When compared to the baseline (uncompressed) *RedScat* algorithm, we observe that when the number of nonzeros falls below 65% (sparsity exceeds 35%), the advantage of compressing the data starts to appear for the *Bits* algorithm. Conversely, for *Idx*, the number of nonzeros must fall below 35%. Figure 11 shows the performance of *Bits* algorithm relative to the case without data compression under different parallelisms. The higher the concurrency, the higher sparsity is needed for compression algorithms to outperform the default.

## VI. Application Performance of BIGSTICK

In this section, we describe how `MPI_Reduce` is used in BIGSTICK and why its performance is critical to BIGSTICK's scalability. We then quantify the performance benefits of our newly developed reduction algorithms and compression techniques on BIGSTICK.

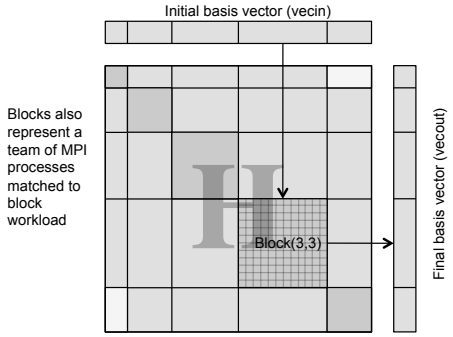


Figure 12. A schematic diagram of the SpMV operation. The application of the Hamiltonian matrix to the initial basis vector to produce the final basis vector, is organized as operations from initial fragments to final fragments.

### A. BIGSTICK Algorithm Description

BIGSTICK implements the configuration interaction method on large-scale, distributed memory platforms. It uses the iterative Lanczos algorithm to solve the matrix form of the the non-relativistic nuclear many-body Schrodinger equation [7]. Its dominant computation is essentially a sparse matrix-vector multiplication between the Hamiltonian matrix(H) and the basis vectors. The matrix and basis vector sizes could be very large, up to tens of billions. To reduce the memory pressure, the non-zero matrix elements are computed on the fly (i.e. it is not CSR) and only the data structure information necessary to reconstruct the matrix is stored. This reduces the memory capacity requiredment by 100-1000×. Nevertheless, the memory requirements remain immense and can be challenging to distribute among processes. As a result, we are often limited to running only one or a few processes per node.

To partition the workload evenly among all the processes, the basis vector is divided into fragments based upon the proton substate eigenvalue $M_p$, which is the $z$ component of angular momentum for the proton substate. Once the basis state vectors are divided into $F$ *fragments*, the Hamiltonian matrix will be divided into $F \times F$ *blocks* correspondingly. Figure 12 illustrates this process.

Each block $(i, j)$ includes all the jump operations from fragment $i$ of the input basis state vector to fragment $j$ of the output vector. Because of physical constraints, mostly quantum selection rules, the nonzero matrix elements (including the data required to reconstruct those nonzero matrix elements on the fly) are not uniformly distributed with the basis elements. To aid in distributing work and memory, the control information for reconstructing matrix elements is contained in data structures we call *bundles*.

We can exactly predict the *number* of operations in a block of Figure 12 (by operation we mean reconstruction and application of a matrix element) from the bundle information. MPI processes are assigned to blocks based on the amount of associated work required for the blocks. All processes assigned to the same block form a team. The operations and associated data will be evenly distributed among the team members. Figure 12 shows the MPI processes divided into a two dimensional array of teams. Each MPI process now owns a set of unique bundles and stores one copy of a fragment of the input and output vectors.

All the processes in a row will form a row communicator so that after the matrix-vector local computation, the results could be reduced using the sum operation to the root process. Then the results will be broadcasted in the column direction using column communicator. Therefore, the root process must be selected from the diagonal block and its rank in the global communicator will not necessary be 0. All the row communicators will communicate at the same time while overlapping with some column communicators.

When more processes are used, each block will be assigned more processes and correspondingly each process in the block will get less work. However, the data size in `MPI_Reduce` function won't change (we're only increasing the 0.5D axis of a 2.5D decomposition). As more processes participate in this collective operation, its running time grows quickly and can ultimately become the scaling bottleneck. Further performance characteristics of BIGSTICK and implementation details could be found in paper [16], [15].

### B. BIGSTICK Performance

Our test problem is $^{130}Cs$ which has 5 valence protons and 27 neutrons, with a frozen $^{100}Sn$ core. The basis vector size is 10.6 billion and there are approximately $2.4 \times 10^{12}$ nonzero matrix elements. The benchmark is run with five Lanczos steps as each step takes similar running time. We focus on our *Redscat* algorithm as it delivers the best performance for large data sizes and higher concurrencies. Figure 13 shows the total MatVec times and the corresponding `MPI_Reduce` times for the *Cray* and our *RedScat* algorithms. The performance is for 2048 MPI processes with 4 processes running on a node. The reduce vector size is about 200M floats.

Our *Redscat* algorithm performs significantly better than the default *Cray* implementation. The performance speedup increases from 1.2× when using one OpenMP thread to 2.6× when using 16 OpenMP threads. The total MatVec time improvements are completely due to the underlying different `MPI_Reduce` implementation.

Figure 14 shows a detailed timing breakdown when using 2048 MPI processes as a function of the number of threads. Similar to the *RedScat* results in Figure 7, the *Local* time scales almost linearly with the number of OpenMP threads. One may attribute the mere 1.3× speedup between 1 and 2 threads to OpenMP overhead. Unlike the *RedScat* data in Figure 7, the *Other* times become much more significant. This is related with the properties of the row communicator. In Figure 7, the size of the MPI communicator is 512 — a power of 2. In Figure 14, although the total number of

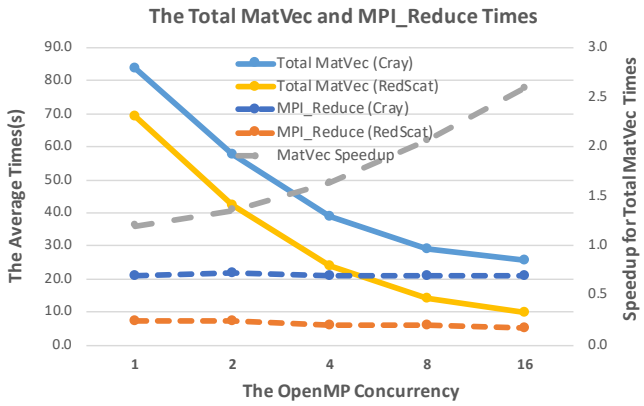


Figure 13. BIGSTICK total MatVec and `MPI_Reduce` time as a function of algorithms and threading for a fixed 2048 processes with 4 process per node. Observe that we attain substantially better performance over using the Cray implementation by optimizing the reduction.

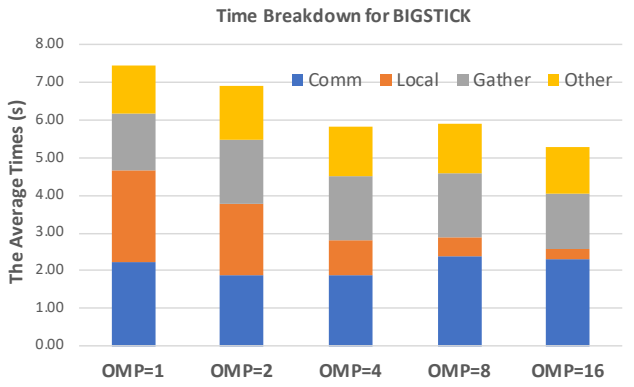


Figure 14. Time breakdown of `MPI_Reduce` in BIGSTICK for our *RedScat* implementation using 2048 MPI processes with 4 processes per node.

processes is a power of 2 (2048), the size of each row communicator is not a power of 2 as it is determined based on the workload distribution. Therefore, as we described in Section III, an extra step is needed to pair some of the processes so that the total number of processes participated in the reduce-scatter phase will be the nearest lower power of two. This extra step is non-trivial and its time is included in the *Other* portions.

### C. Data Compression

We also measured the performance effect of using data compression. Figure 16 shows the `MPI_reduce` times for the *Bits* and *Idx* compression methods and compares to the baseline (no compression). The average nonzeros constitute 8% of each vector on average. The *Idx* method has the highest running times. The *Bits* method performs a little better then the baseline results, not so significantly

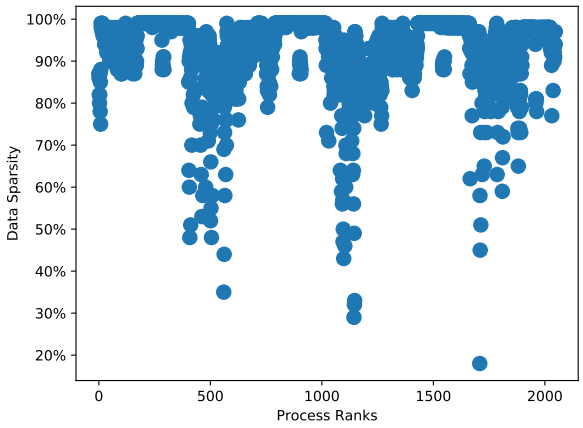


Figure 15. The data sparsity for BIGSTICK `MPI_Reduce` operation across MPI processes. Each dot represents one MPI process plotted at its MPI rank and sparsity for its 200M float vector. Observe there are only a few processes that inject very dense vectors while most processes inject very few nonzeros.

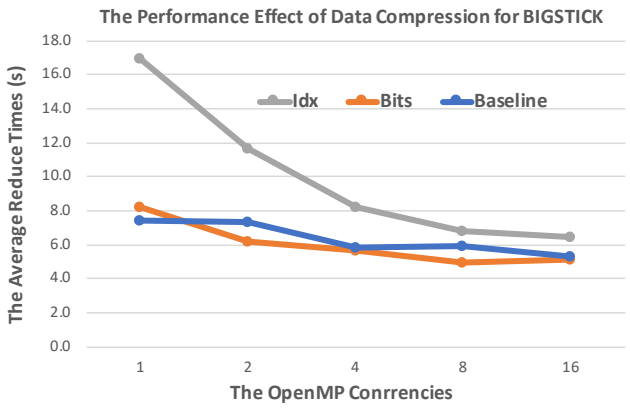


Figure 16. The performance effect of the *Bits* and *Idx* data compression methods compared to the default baseline (no compression).

as expected. Figure 15 displays the data sparsity for the 2048 MPI processes. We see that a couple of processes have very high nonzero density (low sparsity), which is different from our micro benchmarks where all processes have similar nonzero density. A single process in a subcommunicator injecting a dense vector can quickly destroy the bandwidth advantage of compression. A single slow subcommunicator can slow the entire iterative solver. This density imbalance destroys the performance advantage one would hope for from the 10% nonzero global average density.

## VII. Conclusions

In this work, we developed and evaluated new techniques that exploit the massive thread parallelism available on KNL processors to improve the overall `MPI_Reduce` performance. Results showed that this approach is very effective and promising, especially for large vector sizes, and is capable of delivering up to 4× better performance. We then evaluated these techniques within the BIGSTICK application attaining a speedup of 2.6×. We've also applied similar changes to `MPI_Allreduce` and looking for more applications for evaluation.

Our future work is four-fold. First, we see the need to develop better algorithms to address the non-power-of-two cases to alleviate the overhead of pairing, and develop a hierarchical scheme that combines the advantages of the different algorithms presented in this paper. Second, we will deploy new optimizations in our implementations. This includes improving the point-to-point communication both inside a node with direct read and write data access as well as between nodes, exploring alternate partitioning mechanisms that balance the benefits from compression and decomposition of nonzeros against the cost of injecting dense vectors into the reduction network, and exploring more complex, compression techniques amenable to both reductions on floating-point and graph analytic data. Third, we will expand our evaluation methodology to include other HPC platforms (e.g. Infiniband FatTrees) and commodity clusters to quantify the benefits of algorithms, implementations, and compression. Finally, we will integrate and evaluate our threaded reduction into other applications and domains.

## Acknowledgements

This material is based upon work supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

## References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. Concurrency and Computation. In *Practice and Experience*, 22(6), pages 685–701, 2010.

[2] BIGSTICK. http://github.com/cwjsdsu/BigstickPublick/.

[3] A. Faraj and X. Yuan. Automatic Generation and Tuning of MPI Collective Communication Routines. *ICS 05*, 2005.

[4] M. Hofmann and G. Rnger. MPI Reduction Operations for Sparse Floating-point Data. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 94–101, 2008.

[5] N. Jain, A. Bhatele, N. J. W. Xiang Ni, and L. V. Kale. Maximizing Throughput on a Dragonfly Network. *The International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014.

[6] C. Johnson, W. E. Ormand, K. S. McElvain, and H. Shan. Bigstick: A flexible configuration-interaction shell-model code. *arXiv:1801:08432*.

[7] C. W. Johnson, W. E. Ormand, and P. G. Krastev. Factorization in large-scale many-body calculations. *Computer Physics Communications*, 184:2761–2774, 2013.

[8] K. Kandalla, H. Subramoni, A. Vishnu, and D. K. D. Pandaj. Designing Topology-Aware Collective Communication Algorithms for Large Scale InfiniBand Clusters: Case Studies with Scatter and Gather. *IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010.

[9] K. Kandalla, A. Venkatesh, K. Hamidouche, S. Potluri, D. Bureddy, and D. K. Panda. Designing Optimized MPI Broadcast and Allreduce for Many Integrated Core (MIC) InfiniBand Clusters. In *2013 IEEE 21st Annual Symposium on High-Performance Interconnects*, 2013.

[10] S. Kumar, A. Mamidala, P. Heidelberger, D. Chen, and D. Faraj. Optimization of MPI collective operations on the IBM Blue Gene/Q supercomputer. *International Journal of High Performance Computing Applications*, 28:450–464, 2014.

[11] MPICH is a high performance and widely portable implementation of the message passing interface (mpi) standard. https://github.com/pmodels/mpich.

[12] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance Analysis of MPI Collective Operations. *Cluster Computing, June 2007, Volume 10, Issue 2, PP127-143*.

[13] R. Rabenseifner. Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512. *n Proceedings of the Message Passing Interface Developers and Users Conference 1999 (MPIDC 99)*, pages 77–85, 1999.

[14] R. Rabenseifner and J. L. TrŁff. More efficient reduction algorithms for non-powerof-two number of processors in message-passing parallel systems. *Proceedings of EuroPVM/MPI. Lecture Notes in Computer Science, Springer-Verlag*, 2004.

[15] H. Shan, S. Williams, C. Johnson, and K. McElvain. A Locality-based Threading Algorithm for the Configuration-Interaction Method. In *Parallel and Distributed Scientific and Engineering Computing (PDSEC)*, June 2017.

[16] H. Shan, S. Williams, C. Johnson, K. McElvain, and E. Ormand. Parallel Implementation and Performance Optimization of the Configuration-Interaction Method. In *SC '15 Proceedings of 2015 International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2015.

[17] S. Sistare, R. vandeVaart, and E. Loh. Optimization of MPI collectives on clusters of large-scale SMPs. *In Proceedings of SC99: High Performance Networking and Computing*, 1999.

[18] High Performance SParse Communication for Machine Learning. https://arxiv.org/pdf/1802.08021.pdf.

[19] R. Thakur and W. Gropp. Improving the Performance of Collective Operations in MPICH. *10th European PVM/MPI User's Group Meeting*, 2003.

[20] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications, Spring 2005, Vol. 19*, 2005.

[21] V. Tipparaju, J. Nieplocha, and D. K. Panda. Fast collective operations using shared and remote mem- ory access protocols on clusters. *In Proceedings of the 7th International Parallel and Distributed Process- ing Symposium (IPDPS 03)*, 2003.

[22] J. L. TrŁff. Transparent neutral element elimination in mpi reduction operations. *Recent Advances in the Message Passing Interface*, pages 275–284, 2010.

## APPENDIX

This artifact contains the compiler and environment descriptions and the instructions to reproduce the results.

### A. Obtaining the Code

Our work need the following source codes:

- The MPI_Reduce benchmark: available from Bitbucket : git@bitbucket.org:shz0116/myreduce.git
- Application Bigstick: available from Github : http://github.com/cwjsdsu/BigstickPublick/
- Profiling Tool HPCToolkit: available from http://hpctoolkit.org/software.html

### B. Hardware dependencies

The platform we used is located at NERSC, called cori. More details regarding how to use this platform can be found at http://www.nersc.gov/systems/cori/.

### C. Software dependencies

The results are collected on Cori using following modules:

- Compiler: intel/18.0.1.163
- MPI Version: cray-mpich/7.7.0
- Programming Environment: PrgEnv-intel/6.0.4

### D. More Description

There is a Makefile in the *src* directory of Bigstick. Just type "make cori-openmp-mpi", Bigstick will automatically compiled the code to generate the executable file. Execution of Bigtsick needs an input file to define the data set.

The `MPI_Reduce` benchmark can be directly compiled and linked with "cc" command. To run this benchmark, it requires the following parameters "Algorithm, Compress Approach, Data Size". If no explicit "Data Size" is given, the program will run default data range from 4 to 16 million floating points.

The compiling and usage of HPCToolkit can be referenced from hpctoolkit.org.

The timing information are reported at the end of each run.

We plan to create a reproductivity directory to include all the job scripts and related python tools for post-processing in the repository.