LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-484653

Exascale Algorithms for Generalized MPI_Comm_split

A. T. Moody, D. H. Ahn, B. R. de Supinski

May 23, 2011

EuroMPI 2011
Santorini, Greece
September 18, 2011 through September 21, 2011

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Exascale Algorithms for Generalized MPI_Comm_split

Adam Moody, Dong H. Ahn, and Bronis R. de Supinski

Lawrence Livermore National Laboratory,
Livermore, CA 94551, USA
{moody20,ahn1,bronis}@llnl.gov

Abstract. In the quest to build exascale supercomputers, designers are increasing the number of hierarchical levels that exist among system components. Software developed for these systems must account for the various hierarchies to achieve maximum efficiency. The first step in this work is to identify groups of processes that share common resources. We develop, analyze, and test several algorithms that can split millions of processes into groups based on arbitrary, user-defined data. We find that bitonic sort and our new hash-based algorithm best suit the task.

Keywords: MPI, MPI_Comm_split, Sorting algorithms, Hashing algorithms, Distributed group representation

1 Introduction

Many of today's clusters have a hierarchical design. For instance, typical multi-core cluster systems have many nodes, each of which has multiple sockets, and each socket has multiple compute cores. Memory and cache banks are distributed among the sockets in various ways, and the nodes are interconnected through hierarchical network topologies to transmit messages and file data.

Algorithmic optimizations often reflect the inherent topologies of these hierarchies. For example, many MPI implementations [1] [2] [3] use shared memory to transfer data between processes that run within the same operating system image, which typically corresponds to the set of processes that run on the same compute node. This approach is considerably faster than sending messages through the network, but the implementation must discover which processes coexist on each node. Some collective algorithms consider the topology of the network to optimize performance [4] [5]. As another example, fault-tolerance libraries must consider which processes share components that act as single points of failure, such as the set of processes running on the same compute node, the same network switch, or the same power supply [6] [7].

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. (LLNL-CONF-484653)

The first step in these algorithms identifies the processes that share a common resource. A process can often obtain information about the resource on which it runs. However, obtaining information about the resources that other processes in the job use is usually more difficult. We could issue a gather operation to collect the resource information from all processes. However, this approach is prohibitively expensive in both time and memory with millions of processes.

Alternatively, we can almost directly offload the problem to `MPI_Comm_split` since each resource is often assigned a unique name. The challenge lies in mapping the resource name, which may be arbitrary data like a URL string, into a unique integer value that can be used as an `MPI_Comm_split` color value. We could hash the resource name into an integer and then call `MPI_Comm_split` specifying the hash value as the color. However, the hash function may produce collisions, in which case, processes using different resources would be assigned to the same group. We would need to refine this group, perhaps by applying a different hash function and calling `MPI_Comm_split` again in a recursive manner. This process is both cumbersome and inefficient.

We propose a cleaner, faster interface by extending `MPI_Comm_split` to enable the user to provide arbitrary data for color and key values along with user-defined functions that can be invoked to compare two values. `MPI_Comm_split` allows one to split *and* to reorder processes. In our generalized interface, the caller may specify special parameter values to disable either the split or reorder functions. When the reorder function is disabled, processes are ordered in their new group according to their rank in the initial group. The reorder function is often unnecessary and, by allowing the caller to disable it, we can split processes in logarithmic time using a fixed amount of memory under certain conditions. Our key contributions in this paper are: a generalized `MPI_Comm_split` operation; a scalable representation for process groups; implementation of collectives using that representation; implementation of several `MPI_Comm_split` algorithms; and large-scale experiments of those algorithms. While existing algorithms for `MPI_Comm_split` require $O(N)$ memory and $O(N \log N)$ time in a job using N processes, we present algorithms that require as little as $O(1)$ memory and $O(\log N)$ time.

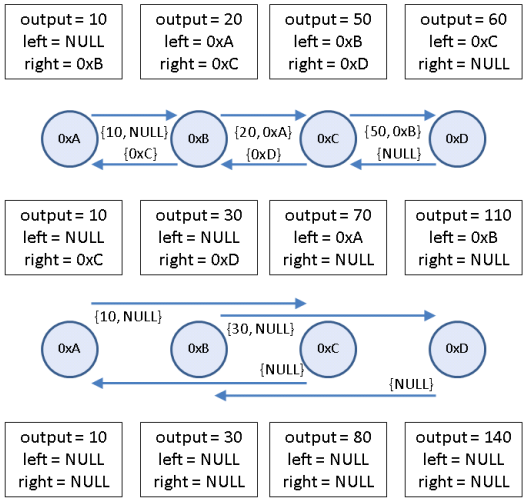
The rest of this paper is organized as follows. Section 2 discusses the linked list data structure that we use to represent groups and illustrates how to implement collectives using it. Section 3 presents several algorithms for a generalized `MPI_Comm_split`, and Section 4 presents experimental results.

2 Groups as chains

Each process in current MPI implementations typically stores group membership information as an array that contains one entry for each process in the group. This array maps a group rank ID to an MPI process address, such as a network address, so that a message can be sent to the process that corresponds to a given rank. Each process can quickly find the address for any process in the group using this approach. However, it requires memory proportional to the group size, which is significant at large scales.

```
scan(input, output, chain) {
  output = input
  left   = chain.left
  right  = chain.right

  while (left != NULL or right != NULL)
  {
    if (right != NULL)
      send {output, left} to right

    if (left != NULL)
      send {right} to left

    if (right != NULL)
      recv {nextright} from right
      right = nextright

    if (left != NULL)
      recv {data, nextleft} from left
      output += data
      left = nextleft
  }
}
```

(a) Pseudo code



(b) Example on 4-process chain

Fig. 1: Inclusive scan on a chain

To represent process groups in a scalable way, we store the group mapping as a doubly-linked list that is distributed across the processes of the group. Each process represents a node in the list, and each records a small, fixed-size portion of the group mapping consisting of the number of processes in the group, its rank within the group, its process address, and the addresses of the processes with a rank one less and one more than its own. We develop our algorithms on top of MPI, so we simply record MPI rank IDs as process addresses. We refer to this doubly-linked list as a *chain*.

Conceptually, we align the chain horizontally with increasing ranks from left to right. Given a particular process as a reference, the *left neighbor* is the process with rank one less and the *right neighbor* is the process with rank one greater. The first rank of the group stores a NULL value as the address of its left neighbor, and the last rank of the group stores this NULL value for its right neighbor.

Although this chain data structure limits the destinations to which a process can directly address messages, many collectives can be implemented in logarithmic time by forwarding process addresses along with the messages that contain the data for the collective. For example, in Figure 1 we illustrate how to implement a left-to-right inclusive scan operation. With N processes in the chain, this collective executes in $\lceil \log N \rceil$ rounds, where in round $i \in [0, \lceil \log N \rceil)$, each process exchanges messages with left and right partners with ranks 2^i less and 2^i greater than its own. Each process first sets its left and right partners to be its left and right neighbors in the chain, and each process initializes its current scan result to the value of its contribution to the scan. Then a process sends its current scan result to its right partner and appends the address of its left partner to the message. The process also sends the address of its right partner to its left partner.

Each process combines the scan data that it receives from its left partner with its current scan result and sets its next left partner to the address included with that message. Each process also receives the incoming message from its right partner to obtain the address of its next right partner. If the address for the process on either side is NULL, the process does not exchange messages with a process on that side. However, it forwards the NULL address values as appropriate. After $\lceil \log N \rceil$ rounds, the current scan value on each process is its final scan value.

One could similarly implement a right-to-left scan, and we use this technique to implement a *double scan*, which executes a left-to-right scan simultaneously with a right-to-left scan. Our algorithms use double scans to implement inclusive scans, exclusive scans, and associative reduction operations that require $O(1)$ memory and run in $O(\log N)$ time. Further, we can implement tree-based collectives on a chain, including gather, scatter, broadcast, and reduction algorithms. Our group representation does not directly support general point-to-point communication, but it is sufficient for all of the algorithms that we present.

3 Algorithms

3.1 Serial sort algorithms

Many existing MPI implementations first gather all color/key/rank tuples into a table at each process to implement `MPI_Comm_split`. Each process extracts the entries in the table with its color value and places those entries into another list. Finally, each process sorts this list by key and then by rank using a serial sort such as `qsort`. If N is the number of MPI processes, each process uses $O(N)$ memory to store the table and $O(N \log N)$ time to execute the sort.

We implement two variants of this algorithm. Our first variant, *AllgatherGroup*, executes an allgather using the chain to collect data to each process. Our second variant, *AllgatherMPI*, calls `MPI_Allgather`. AllgatherMPI relies on the optimized MPI library to collect the data to each process to show how much we could optimize the communication in AllgatherGroup. Neither of these algorithms will scale well to millions of processes in terms of memory or time. However, we include them as a baselines since they emulate the algorithms used in existing MPI implementations [1] [2] [3].

3.2 Parallel sort algorithms

Our second approach to split processes uses a parallel sort. Given a chain of mixed colors and unordered keys, we can split and sort the chain into ordered groups using a parallel sort, a double scan, and a few point-to-point messages. First, each process constructs a data item that consists of its color value, its key value, its rank within the input group, and its process address. We redistribute these data among the processes using a parallel sort, such that the i^{th} process from the start of the chain has the data item with the i^{th} lowest color/key/rank tuple. Each process next exchanges its sorted data item with its left

and right neighbors to determine group boundaries and neighbor processes. We then use a double scan to determine rank IDs and the size of each group. A final point-to-point message sends this information back to the process that originally contributed the data item. Mellor-Crummey et al. proposed this approach for a similar operation in Co-Array Fortran [8].

For this approach, we implement four different parallel sort algorithms. In our first algorithm, *GatherScatter*, we use a tree communication pattern to gather all data items to a root process. We sort items during each merge step of this gather operation so that the items are sorted after the final merge at the root. We then scatter the items from the root to the processes. Similar to our allgather algorithms, GatherScatter uses $O(N)$ memory at the root, but it executes the sort in $O(N)$ time instead of $O(N \log N)$.

In our second parallel sort, we implement an algorithm that is similar to the one that Sack and Gropp describe [9]. In this scheme, we gather the color/key/rank tuples to a subset of processes that then execute Cheng's algorithm to sort them [10]. We then scatter the data items back to the full process set. In our tests, we set the maximum number of data items that a process may hold to a fixed value, M. We use several different values ranging from 128 to 8192, and we label each algorithm as *ChengM*. The number of processes, P, that perform the sort is an important parameter this algorithm. These algorithms use $O(\frac{N}{P})$ memory and $O(P \log N + \log^2 N + \frac{N}{P} \log P)$ time.

Third, we implement Batcher's bitonic sort [11], which we label *Bitonc*. This algorithm uses $O(\log N)$ memory and $O(\log^2 N)$ time.

Finally, for standard `MPI_Comm_split`, in which the color and key values are integers, we implement a divide-and-conquer form of radix sort. This algorithm, *Radix*, splits chains into subchains based on the most significant bits of the key values. We then recursively sort each subchain, and rejoin the sorted subchains into a single, sorted chain. Radix uses $O(1)$ memory and $O(\log N)$ time.

3.3 Hash-based algorithm

Our third method to split processes employs hashing. This method avoids sorting processes when only a split is required. We first hash the color value of the calling process to one of a small number of bins, ensuring that we assign the same color value to the same bin on each process. Then, we execute a double exclusive scan on the chain. For each direction, the scan operates on a table that includes an entry for each bin. Each entry contains two values. The first value encodes the address of the process that is assigned to that bin and is next in line along a certain direction (left or right) from the calling process. The second value counts the number of processes along a certain direction from the calling process that belong to that bin.

Each process initializes all table entries to a NULL address and a zero count, except for the entry corresponding to its bin, in which case it sets the address field to its own address and sets the count field to one. We then perform the double exclusive scan operation, after which the result of the left-to-right scan

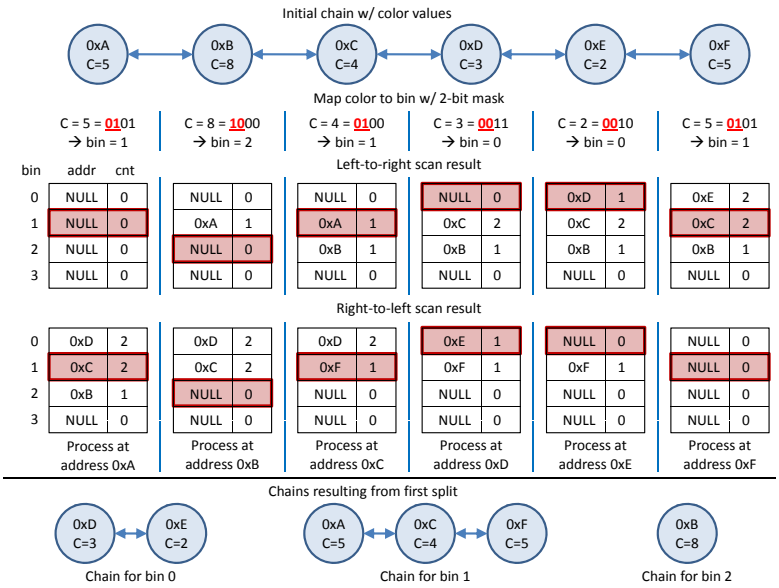
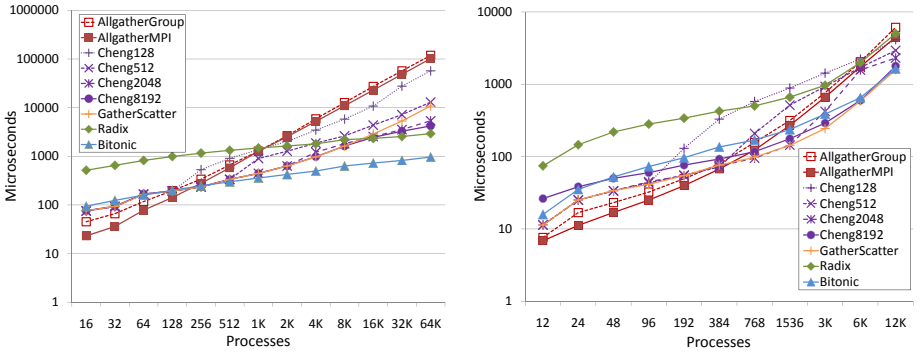


Fig. 2: Splitting a 6-process chain using 4 bins

lists the address of the next process to the left and the number of processes to the left of the calling process for each bin. Similarly, the result of the right-to-left scan lists the address of the next process to the right and the number of processes to the right of the calling process for each bin.

We then create chains that consist only of the processes mapped to a given bin. Each process uses the address and count fields from the table entry corresponding to its bin and assigns the process address from the left-to-right scan to be its left neighbor and the process address from the right-to-left scan to be its right neighbor. It sets its rank to be the value of the count field from the the left-to-right scan and it adds one to the sum of the count fields from the left-to-right and right-to-left scans to compute the total number of processes in its chain. This operation splits the input chain into a set of disjoint chains, potentially creating a new chain for each bin. This new chain may contain processes with different colors. However, the hash function guarantees that all processes with the same color value are in the same chain. An example split operation is illustrated in Figure 2 in which the hash function uses the first two bits of a 4-bit color value to select one of four bins.

We then iteratively apply this split operation to the chains produced in the prior step. Each iteration uses a new hash function so that processes that have different colors eventually end up in separate chains. For this work, we pack the color value into a contiguous buffer and then apply Jenkin's one-at-a-time hash [12] [13]. For different iterations, we apply the same hash function but rotate the bytes of the packed color value and mask different regions of the hash



(a) Dawn BG/P (b) Sierra

Fig. 3: Time for reorder without split

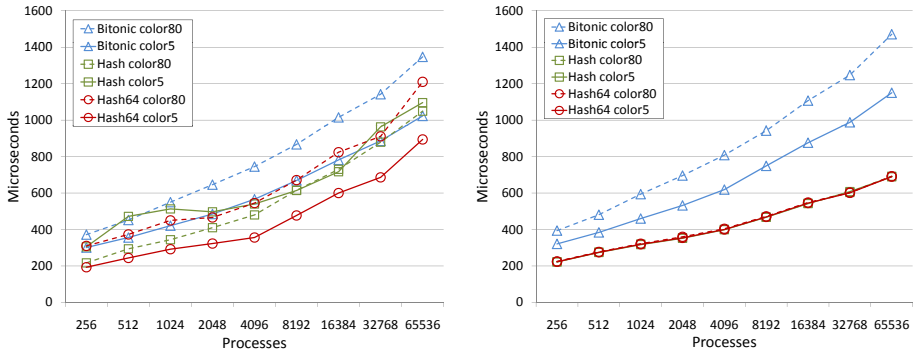
value to obtain new bin numbers. If needed, we invoke a sort algorithm to finish splitting and reordering the chains.

We implement two variants of this algorithm, *Hash* and *Hash64*. Hash repeatedly applies the hash operation until the initial chain is completely split. Hash64 iterates until the chain is completely split or its length falls below a threshold of 64 processes, at which point, we invoke AllgatherGroup to finish the split. Each version stops iterating if a single color value is detected throughout the chain. We check for this condition using an allreduce whenever a split iteration does not reduce the length of the chain. If we need to reorder the chain after completing the hash iterations, we use Bitonic sort. When reordering is required, these algorithms have the same time and memory complexity as Bitonic. When only a split is required, these algorithms use $O(1)$ memory. Due to the nature of hash functions, one may only determine probabilistic upper time bounds. However, one can show strict lower time bounds of $\Omega(\log^2 N)$ when the number of groups equals the number of processes and $\Omega(\log N)$ when the number of groups is small and independent of the number of processes.

4 Results

We test each algorithm on two clusters at Lawrence Livermore National Laboratory. We use Dawn, an IBM BlueGene/P system that has 128K cores on 32K nodes. The second system, Sierra, has over 1,800 compute nodes, each with two Intel Xeon 5660 hex-core chips for a total over 21,600 cores. The Sierra nodes are connected with QLogic QDR Infiniband.

We first investigate the performance of the various sorting algorithms. Disabling the split, and using an integer value as the key, we show the time required to complete a reorder operation on each platform in Figure 3. The two platforms produce significantly different results. The plots all follow clear, distinct trends on Dawn. However, on Sierra, the plots generally converge at higher process counts, at which we conject that network contention limits performance. On



(a) One color per block of 16 processes

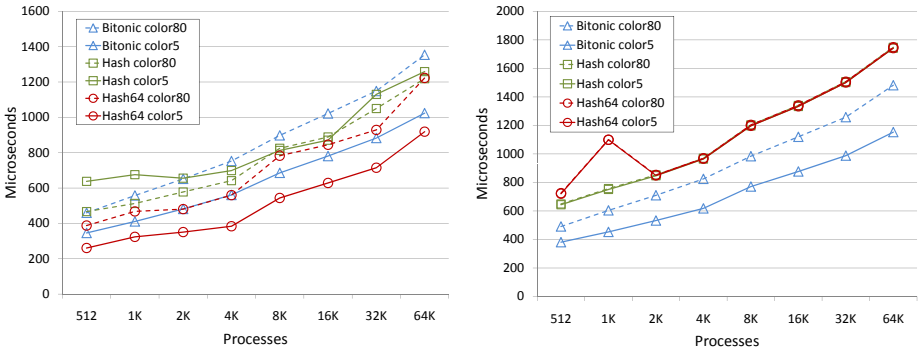
(b) Two colors total

Fig. 4: Time for split without reorder

both machines, the serial sort algorithms are best at small scale, but more scalable algorithms soon outperform them. On Dawn, Radix and Bitonic sort show the best scaling trends. As expected, Radix sort, with its $O(\log N)$ complexity, scales the best. However for all scales tested, Bitonic always has better performance. At 16 processes, Bitonic is 5.5 times faster than Radix. The difference is reduced to 3.0 times at 64K processes, but the hidden constants associated with the big-O notation are too high for Radix to surpass Bitonic. At 64K processes on Dawn, Bitonic sort is 100 times faster than the serial sort algorithms and 4.4 times faster than the fastest Cheng sort. Bitonic is still fast on Sierra, although the apparent contention limits its performance. Regardless, on both machines, the best approach is to use a serial sort algorithm for small scale and to switch to Bitonic sort at large scale.

We next focus on the task of just splitting processes. We compare Bitonic, the fastest parallel sort algorithm, to Hash and Hash64. To see how different color datatypes impact the algorithms, we use character strings of length 5 and of length 80 for color values. Figure 4 shows the results for Dawn. When the number of process groups (the number of distinct colors) is on the order of the number of processes, we find that the hash-based algorithms perform on par with Bitonic sort. However, when the number of groups is small, the hash-based algorithms outperform Bitonic with speedups between 1.7 and 2.1 at 64K processes, depending on the length of the color value. The size of the color value affects the performance of Bitonic, but it has little impact on Hash and only impacts Hash64 in cases where it must call AllgatherGroup. Since the hash operation always maps the color to an integer, its communication costs are not affected by the size of the color value. However, the sort algorithms send the color value in each message, so the cost of these algorithms increases with the size of the color value. Hash or Hash64 perform the best in all cases shown.

We also tested Bitonic, Hash, and Hash64 for splitting and reordering processes in the same operation. The results from Dawn are shown in Figure 5. We used an integer value for the key and character strings of different lengths for



(a) One color per block of 16 processes

(b) Two colors total

Fig. 5: Time for split with reorder

color values. As shown in Figure 5(a), with many groups, and when the groups are roughly equal in size, the timing results look very similar to Figure 4(a). However, with only a small number of groups, as shown in Figure 5(b), then both hash algorithms require more time than Bitonic. In this case, we incur overhead to execute the hash algorithm to split the initial chain, but since the resulting subchains are relatively long, the split does not significantly reduce the cost of sorting. Since we cannot know the size of the resulting groups *a priori*, Bitonic is the best option whenever reordering is required. The peak for Hash64 at 1K processes in Figure 5(b) is an artifact from a bug that invoked AllgatherGroup instead of Bitonic even though the chain was longer than 64 processes after the split. This bug only affected the data points for 512 and 1K processes in Figure 5(b).

5 Conclusions

Developers will soon need scalable algorithms to split millions of processes into groups based on arbitrary, user-defined data. In this work, we developed several algorithms that represent groups as a doubly-linked list, and we investigated their performance through large-scale experiments. We found that bitonic sort and a new hash-based algorithm offer the best results. We find that the hash-based algorithm is up to twice as fast as bitonic sort when only splitting processes. Compared to algorithms used in current MPI implementations, these new algorithms reduce memory complexity from $O(N)$ to as little as $O(1)$, and they reduce run time complexity from $O(N \log N)$ to as little as $O(\log N)$.

Although we focus on algorithms for a generalized `MPI_Comm_split` interface, our findings also apply to the simpler, standard `MPI_Comm_split` function. Thus, we expect MPI implementations to benefit from our results. Further, we can implement these algorithms and group representations directly in applications that need fast methods to identify sets of processes. With this approach, appli-

cations can create lightweight groups without the overhead of creating full MPI communicators.

References

1. Argonne National Laboratory, "MPICH2." `http://www.mcs.anl.gov/mpi/mpich2`.
2. Network-Based Computing Laboratory, "MVAPICH: MPI over Infiniband and iWARP." `http://mvapich.cse.ohio-state.edu`.
3. E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, (Budapest, Hungary), pp. 97–104, September 2004.
4. K. Kandalla, H. Subramoni, A. Vishnu, and D. K. Panda, "Designing Topology-Aware Collective Communication Algorithms for Large Scale Infiniband Clusters: Case Studies with Scatter and Gather," in *The 10th Workshop on Communication Architechture for Clusters (CAC '10)*, 2010.
5. A. Faraj, S. Kumar, B. Smith, A. Mamidala, and J. Gunnels, "MPI Collective Communications on The Blue Gene/P Supercomputer: Algorithms and Optimizations," in *High Performance Interconnects, 2009. HOTI 2009. 17th IEEE Symposium on*, pp. 63 –72, August 2009.
6. A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, (Washington, DC, USA), pp. 1–11, IEEE Computer Society, November 2010.
7. L. A. B. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka, "Distributed Diskless Checkpoint for Large Scale Systems," in *CCGRID'10*, pp. 63–72, 2010.
8. J. Mellor-Crummey, L. Adhianto, W. N. Scherer, III, and G. Jin, "A New Vision for Coarray Fortran," in *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, (New York, NY, USA), pp. 5:1–5:9, ACM, 2009.
9. P. Sack and W. Gropp, "A Scalable MPI_Comm_split Algorithm for Exascale Computing," in *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*, EuroMPI'10, (Berlin, Heidelberg), pp. 1–10, Springer-Verlag, 2010.
10. D. R. Cheng, A. Edelman, J. R. Gilbert, and V. Shah, "A Novel Parallel Sorting Algorithm for Contemporary Architectures," *Submitted to ALENEX06*, 2006.
11. K. E. Batcher, "Sorting Networks and Their Applications," in *AFIPS Spring Joint Computer Conference*, vol. 32, pp. 307–314, 1968.
12. B. Jenkins, "Algorithm Alley: Hash Functions," *Dr. Dobb's Journal of Software Tools*, vol. 22(9), pp. 107–109, 115–116, September 1997.
13. B. Jenkins, "Hash Functions for Hash Table Lookup." `http://burtleburtle.net/bob/hash/doobs.html`, 2006.