Designing Scalable Out-of-core Sorting with Hybrid MPI+PGAS Programming Models [*]

Jithin Jose
The Ohio State University
jose@cse.ohio-state.edu

Sreeram Potluri
The Ohio State University
potluri@cse.ohio-state.edu

Hari Subramoni
The Ohio State University
subramon@cse.ohio-state.edu

Xiaoyi Lu
The Ohio State University
luxi@cse.ohio-state.edu

Khaled Hamidouche
The Ohio State University
hamidouc@cse.ohio-state.edu

Karl Schulz
Intel Corporation
karl.w.schulz@intel.com

Hari Sundar
University of Utah
hari@cs.utah.edu

Dhabaleswar K. (DK) Panda
The Ohio State University
panda@cse.ohio-state.edu

Abstract

While Hadoop holds the current Sort Benchmark record, previous research has shown that MPI-based solutions can deliver similar performance. However, most existing MPI-based designs rely on two-sided communication semantics. The emerging Partitioned Global Address Space (PGAS) programming model presents a flexible way to express parallelism for data-intensive applications. However, not all portions of the data analytics applications are amenable to conversion using PGAS models. In this study, we propose a novel design of the out-of-core, k-way parallel sort algorithm that takes advantage of the features of both MPI and OpenSHMEM PGAS models. To the best of our knowledge, this is the first design of any data intensive computing application using Hybrid MPI + PGAS models. Our experimental evaluation indicates that our proposed framework outperforms existing MPI-based design by up to 45% at 8,192 processes. It also achieves 7X improvement over Hadoop-based sort using the same amount of resources at 1,024 cores.

Keywords Out-of-Core Sort, MPI, PGAS, Hybrid

1. Introduction

With the amount of data doubling every two years [2], the design of algorithms that can efficiently handle massive data sets is gaining prominence. Several frameworks like Hadoop [1] have been proposed over the years to provide scalable processing of large data sets, using commodity hardware. There has been a substantial increase in the number of high performance computing (HPC) sites around the world applying big data technologies and methods to their problems. A surprising two thirds of high performance computing (HPC) sites are now performing big data analysis as part of their workloads, as per the IDC briefing at SC'13 [3].

Sorting is one of the most common algorithms found in modern data analytics codes. Similar to the TOP500 [31] rankings in HPC, "Sort Benchmark" [19] ranks various frameworks available for large scale data analytics using a set of pre-specified benchmarks. In the 2013 Sort Benchmark rankings, the Hadoop framework tops the list with a sort rate of 1.42 TB/min. In our previous work [26], we proposed a new framework that uses a k-way parallel sorting algorithm backed by asynchronous data-delivery mechanism for efficient out-of-core sorting while maintaining high throughput on general-purpose, production HPC hardware. The framework interleaves stages of the sort while continuing to stream input data from a global file system, with the data delivery implemented using two-sided MPI. This framework was able to achieve a sort rate of 1.24 TB/min.

Earlier studies have shown that the two-sided method of data delivery presents inherent overheads like request matching and limited computation-communication overlap [21]. Although MPI offers one-sided communication semantics, they have not been widely adopted due to their complexity and lack of efficient implementations on many systems. On the other hand, realizations of the emerging Partitioned Global Address Space (PGAS) programming models, such as OpenSHMEM [20], present a flexible way for data intensive applications to express parallelism using one-sided communication semantics and a global view of data. They provide shared memory abstractions, expose locality information and define a light-weight, one-sided communication API that makes it easy to express irregular data movements while ensuring performance [5].

Hybrid MPI + PGAS models are gaining popularity as they enable developers to take advantage of the PGAS models in their MPI applications, without having to rewrite the complete application [7, 16, 23]. Such models enable the flexibility of implementing application sub-kernels using either MPI or PGAS models, based on the communication characteristics. The Exascale roadmap defines hybrid model as the 'practical' way of programming Exascale

[*] This research is supported in part by National Science Foundation grants #OCI-1148371 and #CCF-1213084. It used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

Eighth International Conference on Partitioned Global Address Space Programming Models October, 2014, Eugene, Oregon, USA
Copyright © 2014 ACM . . . $10.00

systems [9]. Unified communication runtimes, like MVAPICH2-X [18], are enabling efficient use of these hybrid models by consolidating resources used by two different runtimes, thereby providing performance, scalability, and efficient resource utilization. They also prevent codes using two different models ending in deadlocks, due to the unified runtime design [15].

In this paper, we take advantage of the global address space abstraction and one-sided communication primitives provided by the OpenSHMEM PGAS model to minimize the communication overheads, with efficient computation-communication overlap in the out-of-core Sort. To the best of our knowledge, this is the first such design of a data intensive computing application using hybrid MPI + PGAS programming model. We also propose and implement extensions to OpenSHMEM, such as non-blocking put, and non-blocking put-with-notify as part of this study. We make the following key contributions as part of this paper:

1. Identify major bottlenecks in the existing MPI-based design of out-of-core, k-way parallel Sort
2. Present the challenges involved in redesigning the data distribution and compute framework using the OpenSHMEM PGAS model
3. Propose extensions to the OpenSHMEM communication API in order to efficiently take advantage of the PGAS model
4. Design a scalable and high performance framework for out-of-core, k-way parallel Sort using hybrid MPI + OpenSHMEM models
5. Provide an in-depth analysis of performance and scalability of the proposed design

Our experimental evaluations indicate that our proposed hybrid design is able to outperform the existing design by up to 45% at 8,192 processes. Performance comparison with Hadoop, using the same amount of resources, indicates 7X improvement in sort rate. Further, our scalability experiments indicate that the hybrid design demonstrates good strong and weak scalability characteristics. This study also discusses the cost-efficiency of the proposed design.

The rest of the paper is organized as follows. Section 2 provides a high level overview of parallel sorting, the OpenSHMEM PGAS model, and the MVAPICH2-X unified communication runtime. In Section 3, we discuss the existing MPI-based framework for out-of-core, k-way parallel Sort and its bottlenecks. In Section 4, we discuss the design challenges and present the hybrid MPI + OpenSHMEM design. Section 5 presents the performance evaluations and analysis of the proposed hybrid design. In Section 6, we discuss and compare the design choices for implementing the one-sided communication using OpenSHMEM or MPI, and also discuss the performance results of Hadoop framework. Finally, we present our future work and conclude in Section 8.

2. Background

In this section we provide the necessary background material for our work.

Parallel SampleSort: SampleSort [6] is one of the most popular algorithms used for parallel sort. By sampling a subset of keys from input, it selects $P - 1$ splitters from that set as the boundaries for each bucket and does a global all-to-all data exchange to redistribute all the keys of every process to their correct bucket. An additional local sort is applied to finalize the output array. The challenge with SampleSort is that its performance is quite sensitive to the sampling and selection of splitters, which can result in load imbalance.

OpenSHMEM: SHMEM (SHared MEMory) [24] is a library-based approach to realize the PGAS model and offers one-sided point-to-point communication operations, along with collective and synchronization primitives. SHMEM also offers primitives for atomic operations, managing memory and locks. There are several implementations of the SHMEM model, which are customized for different platforms. OpenSHMEM [20] aims to create an open specification to standardize the SHMEM model to achieve performance, programmability and portability.

Hybrid MPI+PGAS Models: Hybrid MPI+PGAS models present an easier way to express problems with irregular and data intensive communication characteristics. It brings the best of both message passing and distributed memory programming. Several studies [7, 14, 16, 23] have been done in the past which explored the use of hybrid MPI+PGAS models for wide variety of applications and benchmarks such as MILC [30], IMPACT-T [22], Barnes Hut [7], Graph500 [29], and NAS benchmarks [4]. These studies claim significant performance improvement, making use of the concept of global view of data and one-sided communication.

MVAPICH2-X Unified Communication Runtime: Even though the concept of hybrid programming is encouraging, the use of multiple runtimes (one for MPI and one for PGAS model) limits the performance [14]. MVAPICH2-X [18] provides a unified communication runtime for MPI and PGAS (OpenSHMEM and UPC) models on InfiniBand clusters. It enables developers to port parts of large MPI applications that are suited for the PGAS programming model. This minimizes the development overheads that have been a substantial deterrent in porting MPI applications to PGAS models. The unified runtime also delivers superior performance compared to using separate MPI and PGAS libraries by optimizing use of network and memory resources [14, 15].

Apache Hadoop: The MapReduce model has been widely used to perform data intensive operations. Apache Hadoop [1] is a popular open-source implementation of MapReduce programming model and Hadoop Distributed File System (HDFS) is the underlying file system of Hadoop. The Hadoop framework allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage.

3. Existing MPI-Based Design and its Limitations

In this section, we describe the existing MPI-based design of Sort application, and uncover major bottlenecks. We first describe the overall algorithm, and then describe the details.

3.1 Overview of Existing MPI-based Design

In the Sort framework, the processes are divided into two distinct work groups — the `read group` and the `sort group`. The read group is a set of processes dedicated to reading input data from the global parallel file system and delivering this data in a 'streaming' manner to sort group processes. Since the data sets are larger than the combined main memories of both groups, the input `M` records are chunked into `q` chunks. In read stage, the read group processes sequentially read the data from the global filesystem and transfer this data to the sort group processes. The sort group processes determine the splits for SampleSort, based on the sampling of the initial few chunks and buckets (i.e., writing the records for each split into local disk) the incoming records into these bins, stored on the local filesystem on each node. Once all the `q` chunks have been read in, each node will have `q` buckets stored on the local filesystem. Finally, the bucketed data is sorted and is written back to the global filesystem. During the write stage, the flow of information is mostly reversed, with the sort group processes reading the `q` local files, one at a time, synchronized across all processes, sorting them globally, and then writing the final sorted data back to the global filesystem, as multiple files [25, 26].

Figure 1 presents the system architecture for the existing MPI based design. The parallel job is launched such that each node has multiple processes (equal to the number of CPU cores) per node. The read group and sort group processes form two MPI communicators, `READ_COMM` and `SORT_COMM`, respectively. Because of the difference in workloads and system characteristics, only one reader process per reader host is placed in `READ_COMM`. In sort hosts, one process is dedicated for receiving the data from reader hosts. These receiver processes along with the sender processes in reader hosts form a third communicator — `XFER_COMM`. In addition to the three primary communicators defined so far, there exists a family of binning communicators which form a subset of `SORT_COMM`. These `Nbin` communicators, defined as `BIN_COMM`, are built using one process from each sort host, in a vertical manner. These `Nbin` `BIN_COMM` groups are used to overlap the process of bucketing local data and saving to temporary storage with the receipt of new input data.

The reader process is threaded using OpenMP work-sharing constructs on the reader hosts such that one thread per host is dedicated solely to reading new input files and storing the streaming data in a FIFO queue. The companion transfer tasks on the same IO host in `READ_COMM` are in a constant spin loop checking for new data to become available. The *rank 0* process in `READ_COMM` gathers the amount of data available on each IO host, and assigns destination ranks (receiver tasks in `SORT_COMM`). This book-keeping by *rank 0* is for transferring the data to sort hosts, in a round-robin manner, to the receiver tasks.

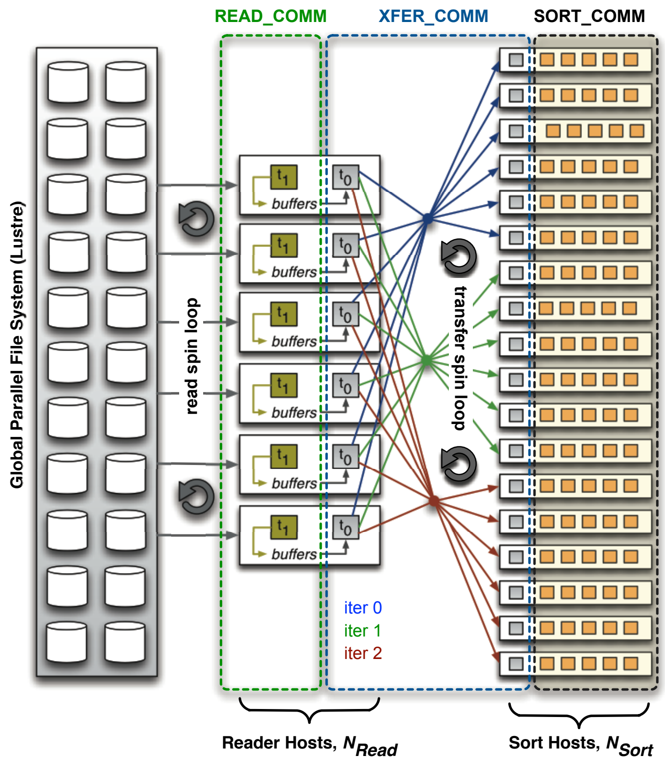


Figure 1. Architecture of Existing Out-of-Core Sort Framework (Courtesy - [26])

Once the destination ranks are assigned, IO tasks issue asynchronous `MPI_Isends` with whatever data they have read so far (not exceeding the receive buffer size), to the destination sort nodes. The data size is indicated in a separate message, sent using `MPI_Bsend`. The receiver tasks on sort nodes receive data in a round-robin manner, which is orchestrated by sending messages between receive tasks. That is, when a receiver task receives the message (token) from a neighboring receive task, it posts an `MPI_Recv` to read the length of incoming data, and then passes the token to the other neighboring task. The token will be updated with amount of incoming data. After passing the token, the receiver task posts another `MPI_Recv` to receive the actual data. The token is also used for identifying when to stop posting receives. At any point, the token indicates the total amount of data that has been received. If this equals to the total amount of expected data, no further receives are posted. The end of read stage is informed to all receiver tasks by using a special value for token.

3.2 Major Overheads/Bottlenecks in Current Design

The major overheads in the existing Sort framework are as listed below:

Poor resource Utilization and Lack of Overlap: As discussed earlier, one CPU core on each sort node is dedicated for the receiver tasks. This limits the compute resources available for the sort computation. Because of multiple `BIN_COMMs`, it is expected that there will be overlap between the receiver task and the computation at sort tasks. However, profiling results using HPC Toolkit [13] indicate that nearly 28% of execution time of the sort task is spent waiting for the receive task, as shown in Table 1. This indicates non-optimal overlap and results in wasted CPU cycles on all the cores.

Table 1. Percentage of Time Spent during Read Stage

Operation	Percentage (%)
Computation (Local Sort/Bucketing)	56.2
WaitForActivation	28.5
Final Barrier	15.3

One-sided routines using Remote Direct Memory Access (RDMA) can be employed to reduce these overheads and offload the communication to the Network Interface Controller (NIC), while freeing all the cores for computations.

Book-keeping and Synchronization Overheads: Yet another overhead is the book-keeping required at `rank 0` of `READ_COMM`. All processes in `READ_COMM` continuously participate in `MPI_Gather` for identifying the amount of read data at each process. Then `rank 0` assigns destination ranks to each of these and distributes them using `MPI_Scatter`. The book-keeping overhead and the successive collective communication limit the overall read bandwidth from global filesystem.

4. Design and Implementation

In this section, we first present the challenges of redesigning different components of the framework, and then present the detailed design.

4.1 Design Challenges

Read Task Coordination with Reduced Synchronization: As outlined in Section 3.2, in the existing implementation, *read* tasks use collective communication to determine the amount of data read by each read task and to identify the destination *sort* tasks. These overheads can be reduced by using one-sided communications. However, it is a challenge to establish coordination using one-sided communication operations over global shared memory.

Read Task-Write Task Synchronization and Overlap: Transferring the input data in multiple segments provides overlap between *read* tasks and *sort* tasks. However, a light-weight mechanism is needed for the *read* tasks to identify destination buffers, without interfering the *sort* tasks, that are busy computing.

One-sided Semantics and Detecting Remote Completion: The two-sided semantics inherently provide a notification to the receiving *sort* task, when the data is available. When one-sided communication is used, an additional mechanism is required for the *sort* tasks to identify the completion of incoming transfers. Similarly, a mechanism is required for the *read* task to know when the remote buffer is free to write into. Expensive schemes such as remote memory polling (which are common in shared memory programming) shall be avoided, as it increases network traffic and is detrimental to performance.

4.2 Proposed Hybrid Design Architecture

Figure 2 shows the modified architecture that we propose for the hybrid MPI + OpenSHMEM based design of out-of-core Sort. The use of global memory model and one-sided communication removes the need for dedicated receive processes, thereby adding more compute power to the *sort* group. We introduce multiple data staging buffers between the *read* group processes and the *sort* group processes to enable overlap between data movement and the sort computation. The coordination among the *read* group and selection of destination *sort* processes for data transfer is established using an atomic counter based design. We first present an overview of the logical global memory management and then go into the details of different components of the framework later in this section.

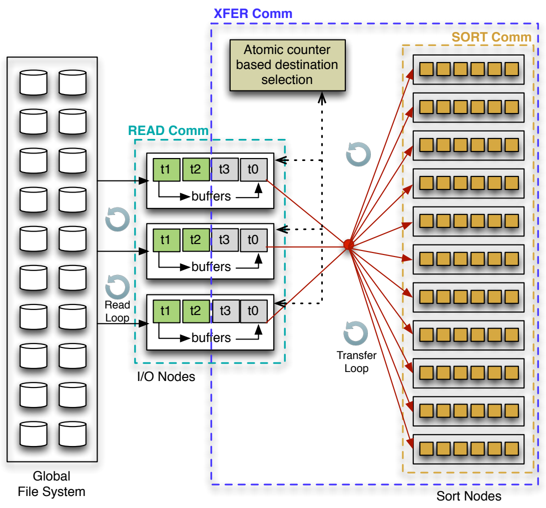


Figure 2. Architecture of the Proposed Hybrid MPI+PGAS Out-of-Core Sort Framework

Figure 3 depicts the overview of application stack for the design of Sort based on the hybrid MPI+OpenSHMEM model. The Sort application makes MPI and OpenSHMEM communication calls which are served by the underlying MVAPICH2-X [18] unified communication runtime. The application uses InfiniBand as the communication network and Lustre as the underlying global filesystem. Design details for the hybrid sort is described in detail in the following sections.

4.3 Detailed Design

Global View of Memory and Management: As discussed in Section 2, each process in the PGAS model exposes specific amount of memory that any process can access (read/write). Such global

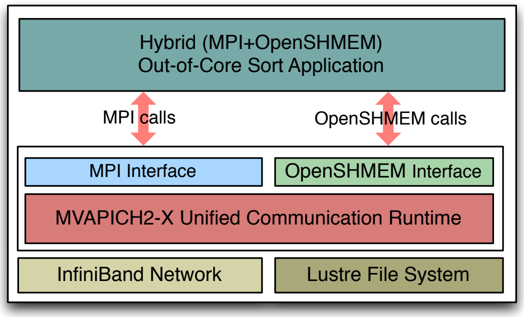


Figure 3. Hybrid MPI+OpenSHMEM Application Stack

view of data can be utilized for efficient data transfer mechanisms. Figure 4 depicts the organization of global memory allocated at all processes, for the hybrid Sort application.

The memory region that each process exposes is split into multiple buffer chunks. These buffers are used for data transfer. In addition, a $statusbuffer$ region is also maintained in the shared heap region to maintain the status of each data buffer chunk.

Similar to the current MPI based design, only one process per *read* host acts as the active *read* process while the other remain passive. This choice is due to the fact that one process alone is able to saturate the network bandwidth, as the message size is large enough. However, the active *read* process shall be able to use all the memory available on the node. To this end, the active *read* process takes control of the communication buffers allocated by all the other processes in the node, using the `shmem_ptr` functionality available in OpenSHMEM. This enables the active *read* process to directly load and store data into this memory, even though it is hosted by other processes. Due to the globally visible memory at the *sort* processes, the *read* processes can also write data directly to the destination *sort* process rather than requiring dedicated *receive* processes, like in the existing framework.

Destination Selection using Atomic Counter: In the existing framework, all the active *read* processes synchronize using an `MPI_Gather` operation to collect the amount of data each process has read so far. Then the *rank 0* among *read* processes assign destination ranks based on the gathered data. This assignment is dispersed to all the *read* hosts using an `MPI_Scatter` operation. These steps are repeated continuously until all the input data have been transferred. In the proposed design, we avoid these book-keeping overheads by using a global atomic counter. When a $readprocess$ has a data block ready to be transferred, it atomically increments the counter by one. Based on the counter value, the reader task can determine its destination rank and the buffer index at destination process, using the following equations. The equations are formed such that the distribution of data among the $sortprocs$ is done in a round-robin manner in column major mode. This is for enabling efficient bucketing of data, that happens among the sort processes within a `BIN_COMM`.

A sender task determines the destination process rank using the following Equation 1.

$$\begin{aligned} dest\_rank = {} & (numIoHosts * numTasksPerHost) + \\ & ((counter/numSortHosts)\%numTasksPerHost) + \\ & (counter\%numSortHosts) * numTasksPerHost \end{aligned} \quad (1)$$

where, the *numIoHosts*, *numSortHosts*, *numTasksPerHost* indicate number of reader hosts, number of sort hosts, and number of tasks per host, respectively. This approach completely removes the col-

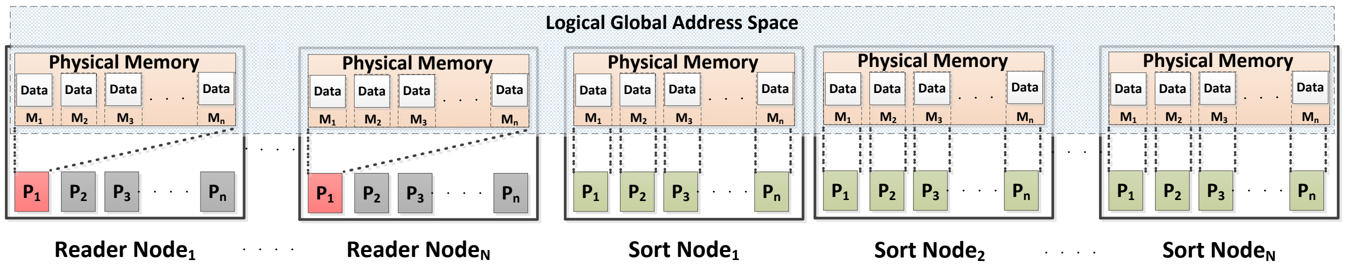


Figure 4. Logical Global View of Data

lective synchronization used in the earlier framework. Similarly, the buffer index at the destination process where the data needs to be placed can be calculated using Equation 2.

$$buffer\_index = (counter/(numSortHosts * numTasksPerHost))\% (MAX\_READ\_BUFFERS/numTasksPerHost) \quad (2)$$

where, `MAX_READ_BUFFERS` indicates the number of buffers allocated per host. This parameter can be configured through environmental parameters and shall be set based on the system and application characteristics.

Remote Buffer Co-ordination using Atomic Compare-Swap: The data delivery mechanism employed in the hybrid design is explained in Figure 5. Before writing the data to the remote side, the *read* process needs to make sure that the remote buffer is empty, and the *sort* process has finished processing the buffer. As discussed earlier in Section 4.1, remote memory polling mechanisms shall not be used, as it will degrade the performance, and increase network traffic. We use the atomic `compare - swap` routine to ensure buffer availability. For each buffer, we associate a status buffer (64bit) to keep track of the buffer status. The 64-bit length is selected, since the atomic operation granularity offered by Mellanox InfiniBand adapters is 64-bit. A value `1` in the status buffer indicates buffer is full, and a `0` indicates buffer is empty.

When a *read* process is ready to transfer data, it does a `compare - swap` operation on the remote buffer status, in which the buffer status is compared with `1`, and swaps it with `p+rank`, if comparison is true. Here `p` denotes the total number of processes, and `rank` denotes the rank of the *read* process. The `compare - swap` operation returns the original value. Thus, a return value of '0' indicates that the buffer was free, and then the *read* process can proceed with writing data to remote sort process.

If the return value of `compare - swap` operation is `1`, then it indicates that the *sort* process has not yet processed the buffer. In this case, the *read* process will wait on polling a local memory location, which is symmetric to the remote buffer status location. Finally, when the *sort* process finishes processing of buffer, it identifies that the buffer status is `p+rank`, and it unblocks the waiting *read* process by updating the symmetric buffer status location at *read* process (identified using `rank` information). The entire data delivery operation is depicted in Figure 5.

Data Delivery Using Non-blocking Put+Notify: The data write operation (from *read* to *sort* process) also needs to be designed efficiently. The current OpenSHMEM standard supports only blocking `put` operations. These operations do not ensure completion, and special synchronization operations can be used for waiting until all the outstanding operations are complete. However, for efficient overlap, non-blocking operations, and synchronization operations with per-operation granularity is desired. For example, if a *read* process wants to reuse a buffer and if it calls synchro-

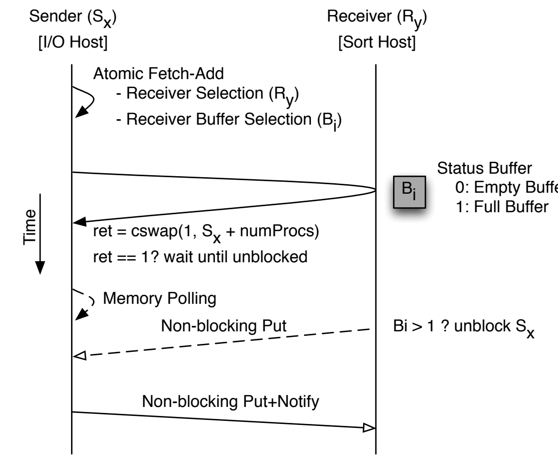


Figure 5. Data Transfer using One-sided Communication

nization operation to complete all the outstanding `put` operations, it will significantly limit the computation-communication overlap. Thus, we propose non-blocking `put-with-notify` as a simple extension to OpenSHMEM. The operation will return a handle, which can be used for checking operation completion. Such operations can be very handy in PGAS applications with irregular communication characteristics, such as Graph500 [8, 12, 16]. We use `put-with-notify` to write data to the destination buffer chunk and notify the destination process by writing `1` to the corresponding buffer chunk.

Data Reception in Sort Group: As indicated earlier, the input data is written to the *sort* process in a round-robin manner. Thus, each sort process can just poll the status buffer location corresponding to the expected buffer location, for checking incoming data. This avoids the need for polling multiple buffer locations, and avoids cache pollution.

Bucketing of received data into different bins is a collective operation among the processes within a single `BIN_COMM`. If every process processes the data in one receive buffer chunk granularity, then the total number of bucketing operations required can be calculated in advance. However, processing multiple buffers during each bucketing operation can reduce the total number of bucketing operations. This requires an extra collective synchronization among the processes within a `BIN_COMM` for checking termination condition. We employ a non-blocking `MPI_Iallreduce` for this check, so that maximum overlap is achieved. We evaluate both these design alternatives in the performance evaluation section. These schemes are denoted as Hybrid-SR (Simple Read) and Hybrid-ER (Eager Read), respectively.

Custom Memory Allocator using Shared Heap: During the final sort stage, data in each bucket is read from local disk, to construct a vector of data records. These records are then sorted and written to global file system. In the hybrid design, we reuse the OpenSHMEM shared heap for holding the data records. The data record vector is constructed with a custom memory allocator, which uses the OpenSHMEM shared heap region, and thereby avoiding extra memory allocations.

5. Performance Evaluation

In this section, we present detailed evaluation and analysis of our proposed hybrid design. We evaluate the data transfer time, overlap of computation and communication, final sort time, and scalability characteristics. We also present a performance evaluation with Hadoop, using the same amount of resources.

5.1 Experimental Setup

We used TACC Stampede [28] for our experimental evaluations. The compute nodes in Stampede are equipped with Intel Sandy-Bridge series of processors using Xeon dual eight-core sockets, operating at 2.70 GHz with 32 GB RAM. Each node is equipped with MT4099 FDR ConnectX2 HCAs (54 Gbps data rate) with PCI-Ex Gen3 interfaces. The operating system used is CentOS release 6.3, with kernel version 2.6.32-279.el6 and OpenFabrics version 1.5.4.1. The global filesystem in Stampede is Lustre v 2.1.3.

The algorithms and software developed to support the end-to-end disk sorting procedure were all written in C++. The existing MPI based Sort code used in this study is described in [26]. We used MVAPICH2 v2.0b as the underlying MPI library for pure MPI-based design. We used MVAPICH2-X Unified Communication Runtime for MPI+PGAS based on v2.0b for the hybrid design. The OpenSHMEM stack in MVAPICH2-X v2.0b is based on OpenSHMEM v1.0e.

All of the performance evaluations are measured on the system running in normal, production operation via batch job submission. Since the global file systems are a shared resource, available to all users for general purpose I/O in their applications, the I/O bandwidth delivered to our sort procedure is not guaranteed to be constant. Hence, we ran our experiments multiple times, and the lowest numbers are reported for both existing and the proposed hybrid designs. For all the experiments, we evaluate both hybrid 'simple-read' and 'eager-read' designs. These are denoted as 'Hybrid-SR' and 'Hybrid-ER', respectively.

5.2 Evaluation of Different Phases of Sort Operation

As discussed in Section 3, there are two main stages in the overall sorting procedure: 1) global filesystem read and transfer, and 2) final sort and writing to global filesystem. We measure the individual execution times for each of these stages, for both existing design and the proposed hybrid designs. These results are presented in Figure 6.

Here, 'Tx' indicates the transfer stage and 'FS' indicates the final sort stage. The value in parenthesis indicates the input size that the experiment was run with: 1 TB, 2 TB, and 4 TB, for system sizes 1,024, 2,048, and 4,096 processes, respectively. As it can be observed from the figure, the hybrid design reduces the execution time of data transfer stage significantly. The execution time for data transfer stage for 4 TB is 476, 303, and 298 seconds for Existing, Hybrid-SR, and Hybrid-ER designs, respectively. This indicates an improvement of 37% over the existing design. It can be noted that with increasing scale, the improvement increases. The improvement is mainly because of the one-sided communication and reduced overhead in case of hybrid design. Further, there is an extra bin of sort tasks (which are used as dedicated receiver tasks in existing design), which can also work on local sorting and bucketing.

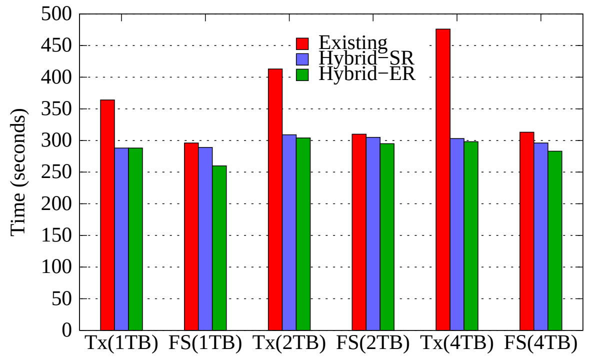


Figure 6. Split-up of Time Spent in Different Phases of Sort Operation

The execution time for final sort stage for 4 TB data size is 313, 296, and 283 seconds, respectively. As indicated in Section 4, the hybrid design does the data transfer in a uniformly distributed manner, such that each sort task receives almost equal amount of data. This is achieved by the use of atomic counter and sending the data (using remote put) in uniform chunk size. This uniform data distribution improves the load balancing among the sort tasks. Moreover, as discussed in Section 4.3, the custom memory allocator design using OpenSHMEM shared heap region evades the need for allocating memory dynamically, during the final sort stage. All these factors improve the performance of final sort stage.

5.3 Overlap Evaluation and Resource Utilization

In this experiment, we evaluate the extent of overlap of computation and communication. Here, the computation refers to the local sorting and bucketing of the records into local file system. Overlap is computed as the total time for computation divided by the overall time. If the computation and communication are perfectly (100%) overlapping, the value will be one. We measure the individual time for local sorting, binning and bucketing and thus the compute time.

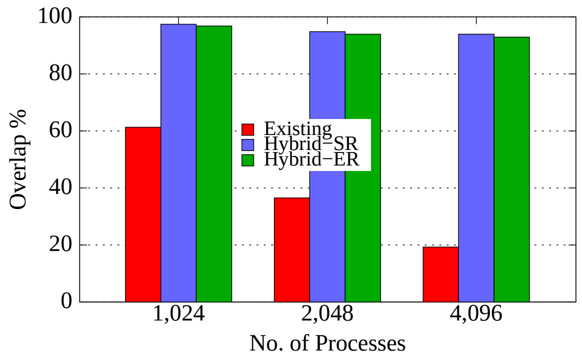


Figure 7. Overlap Evaluation

We conducted this experiment for an input size of 1 TB of data with varying scale – 1,024, 2,048, and 4,096 processes. The overlap results are presented in Figure 7. The existing design achieves about 60% overlap, at 1,024 processes. But as we scale up, the overlap percentage drops. At 2,048 processes, the overlap is about 20%. On the other hand, both the hybrid designs achieve very good overlap, nearly 95%, for all the different scales.

The hybrid design uses one sided communication operations leveraging RDMA feature. Thus, there is no need for explicit receiver tasks in sort nodes, as compared to the two-sided model in the existing design. These compute cores are utilized for sorting in hybrid design, thereby increasing the compute power.

5.4 Overall Execution Time of Sort

Figure 8 presents the overall execution time for the sort operation, for varying system sizes. For these experiments, we kept the input size as 1 TB, and varied the system scale from 512 processes to 8,192 processes.

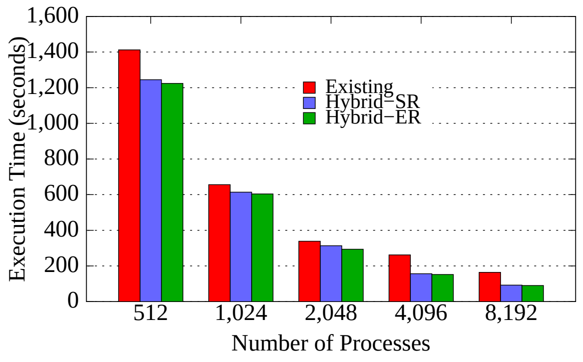


Figure 8. Overall Execution Time of Sort

As it can be noted from the figure, the execution time decreases with increasing system scale, since the input size is constant. The results highlight the benefit of hybrid design as we increase the scale. Both Hybrid-SR and Hybrid-ER designs perform better than existing design, as we scale up. In all the cases, Hybrid-ER performs slightly better than Hybrid-SR, as the number of binning iterations are lesser in case of former. At 8,192 processes, the total execution times reported are 164, 92.55, and 90.36 seconds, for Existing, Hybrid-SR, and Hybrid-ER, respectively. This is nearly 45% improvement in performance, as compared to the existing design.

5.5 Scalability Evaluation

We evaluate the weak and strong scaling characteristics of the proposed hybrid designs in this section. The results are presented in Figure 9. We report the aggregate sort rate (TB/min) for this evaluation.

The weak scaling results are presented in Figure 9(a). In this experiment, we kept a constant problem size per processor core as Input Size=1 TB per 512 cores, and varied the number of processes from 512 to 4,096. We doubled the input size with every step in problem size. We observe that both hybrid designs achieve better weak scaling results. Results at larger scale indicate that the hybrid design imposes no overheads with increase in system size. Another observation is that, as we scale up with increase in data size, the Hybrid-ER performs slightly better than Hybrid-SR. Here, the input data is aggressively being processed (binning/bucketing), and thereby reducing the total number of binning/bucketing. At 4,096 cores, the aggregate sort rates observed are 0.25, 0.34, and 0.37 TB/min, respectively for Existing, Hybrid-SR and Hybrid-ER designs. The Hybrid-ER performs nearly 33% better compared to the existing design.

The strong scalability results are presented in Figure 9(b). Here, we kept the problem size constant as 1 TB, and varied the system scale from 1,024 to 8,192 processes. The results indicate that the hybrid designs achieve better strong scalability characteristics. The results indicate similar performance for both the hybrid designs. At 8,192 processes, the sort rate reported are 0.36, 0.64, and 0.66 TB/min, respectively for Existing, Hybrid-SR, and Hybrid-ER designs. Thus, the Hybrid designs boost up the sort rates by nearly 1.83X times.

5.6 Performance Comparison with Hadoop

As indicated in Section 1, Hadoop is one of the most widely used data analytics framework for processing vast amounts of data. We compare our proposed framework with Hadoop, with same amount of resources, on the same experiment cluster. We used Apache Hadoop [1] v1.2.1 for this experiment. After parameter tuning, we choose 128 MB as HDFS block size (with replication factor = 3) and 8 concurrent maps and 4 concurrent reduces on each compute node. The input data was copied in a distributed manner from the global filesystem to HDFS. Then, we chose the standard Hadoop TeraSort to sort the data. Finally, the sorted data was written back to the global filesystem in parallel. MapReduce and HDFS were configured to use InfiniBand (IPoIB mode) network.

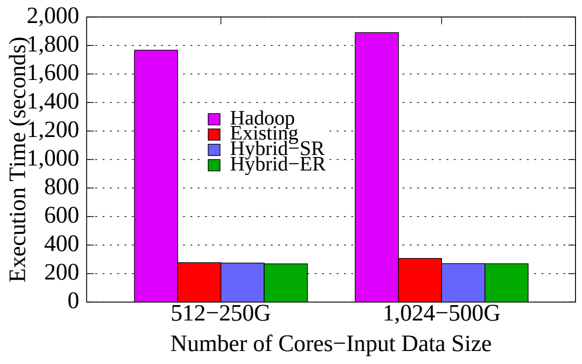


Figure 10. Performance Comparison with Hadoop

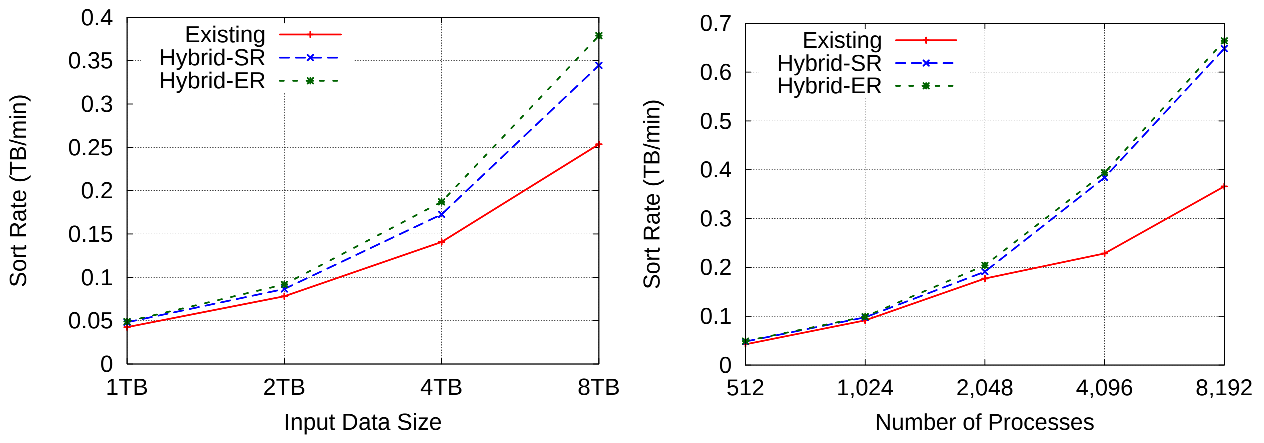
The evaluation results are presented in Figure 10. We conducted this experiment for 250 and 500 GB input size, with varying system sizes of 512 and 1024 cores, respectively. As it can be noted from the results, the Hadoop execution times are much higher. At 1024 cores, the total execution times reported are 1890, 306.34, 270.22, and 269.52 seconds for Hadoop, Existing, Hybrid-SR, and Hybrid-ER, respectively. This is nearly 7X improvement for hybrid design, over Hadoop. The main reason here is because of the single local disk (HDD) present in compute nodes in the experiment cluster. Local disk throughput is one of the main factors affecting Hadoop performance, and many of the Hadoop clusters use multiple hard disks per node [10, 11]. However, not many large scientific clusters (like TACC Stampede) provide support for multiple disks/node. The authors believe that these evaluations are fair, given that these evaluations use same amount of resources for Hadoop, MPI and hybrid designs.

6. Discussion

In this section, we provide a general discussion on the different choices for designing the Out-of-Core Sort framework. First, we discuss about the choice between MPI+PGAS hybrid model versus MPI-3 RMA. Further, we provide a detailed discussion about using Hadoop for the Sorting framework and explore reasons for the observed performance limitations.

6.1 PGAS and MPI-3 RMA

The one-sided communication semantics for MPI was introduced in MPI-2.0, and have been significantly extended in the recently



(a) Weak Scaling (1TB per 512 processes)

(b) Strong Scaling (1TB Data Size)

Figure 9. Scalability Results

standardized MPI-3.0 [17]. Either PGAS programming models like OpenSHMEM or MPI-3 RMA can be used for implementing the one-sided communication proposed in the new design. However, the one-sided communication model of OpenSHMEM is much easier than the semantics in MPI-3 RMA. Even though efficient implementations of MPI-3 and OpenSHMEM can achieve comparable performance, OpenSHMEM offers better programmability [32]. Further, since the MPI-3 RMA standards are relatively recent, there is a lack of efficient MPI-3 implementations.

To substantiate our design decision of using OpenSHMEM for one-sided communication, we review the complexities involved in designing the out-of-core Sort application using MPI-3 RMA. Our findings are listed here.

- For the MPI-3 based design, application developer has to create three memory windows (for actual buffer regions, status buffer, and the atomic counter). Application developer also needs to keep the window handles for further data movement and any synchronization operations. An alternative would be to use dynamic windows, attaching buffers onto the same window, but this will increase complexity of managing the attached buffers. In case of OpenSHMEM, these buffers can be simply allocated from the shared heap. For any subsequent communication operations, these buffers can be used, as if they were in local node (similar to shared memory programming), and the OpenSHMEM runtime will translate into actual destination address.
- As discussed in Section 4, each reader process consists of I/O reader threads and communication threads. Based on the file system characteristics study, it is optimal to keep one I/O reader thread per node [25]. Thus, we enable one reader process per node in the proposed hybrid design. The active read process can utilize the communication buffers allocated by all other processes within the node, using OpenSHMEM `shmem_ptr` functionality. This enables the active read process to directly load and store data into this memory, even though it is hosted by other processes.

  Similar designs can be done using MPI-3 shared window (MPI_WIN_ALLOCATE_SHARED). However, this requires creation of a communicator on each of the read nodes, followed by window creation on each of these nodes. Another way to implement this is to create a window involving just the active read process, and all the sort process. However, this also requires a separate communicator creation. Thus creating the communicators and separate windows, and maintaining them adds additional complexities in case of MPI-3.
- In MPI-3 RMA, the displacement at remote memory region needs to be specified for every data transfer (eg. `MPI_Put`) operation. In case of OpenSHMEM, displacement calculation is taken care of by the OpenSHMEM runtime, and thus it improves the programmability. Additionally, we propose simple extensions for OpenSHMEM communication operations (such as non-blocking put, and non-blocking put-with-notify), which further improves the flexibility and programmability of OpenSHMEM model.

6.2 Hadoop on Commodity Clusters

As indicated in Section 1, many of the HPC clusters are being used for big data analytics workloads [3] using Hadoop MapReduce frameworks. Even though MapReduce was proposed as a framework for commodity clusters, many of the production clusters use special configurations such as multiple storage disks per node, and memory intensive nodes [11, 27]. Even the rank #1 system in 'Sort Benchmark' uses twelve, 3 TB disks per node [10]. On the other hand, frameworks such as the one proposed in this study, can run on commodity clusters and with good performance; thus proving to be more cost-efficient. The productivity aspect of PGAS models is another factor, adding to cost-efficiency. The global view of data and the shared memory abstractions in PGAS models improve the productivity.

6.3 Benefits of Proposed Hybrid Design

The performance analysis reveals that the proposed hybrid MPI + OpenSHMEM framework improves over the existing design on multiple angles. The use of one-sided communication removes request processing overheads and minimizes synchronization overheads at the *sort* processes. Collective synchronization among *reader* processes is eliminated by using a global atomic counter to coordinate data distribution. These collectively improve the overall *Performance of Data Delivery*. We show this empirically as a reduction in the time spent by *sort* processes in data transfer stage, in Section 5.2. The data movement is pipelined using a pool of buffers at each *sort* process. The use of one-sided communication frees up the *sort* processes to continue computation while data movement is in progress. This considerably improves the *Overlap of Computation and Communication*, as shown in Section 5.3. The use of one-sided communication and direct delivery of data to the *sort* processes also removes the need for a dedicated receiver process, improving the *Effective Utilization of CPU Cores*. Owing to the improvements along these different dimensions, the proposed framework is able to reduce the overall sort time by around 45% compared to that using the existing framework, on 8,192 processes.

7. Verification of Sort Output

We verified the output of the sort operation obtained from the Hybrid design to ensure correctness. Since the hybrid design transfers data in a uniform manner, the output file sizes are different for both designs. Thus, for ensuring correctness, we merged the output files into a single file, and compared the `md5sum`. The `md5sum` values in both designs were identical.

8. Conclusion and Future Work

In this study, we identified various bottlenecks in the existing implementation of k-way SampleSort and presented the challenges involved in redesigning the data delivery using the OpenSHMEM PGAS model. We proposed a scalable and high performance design of sort using hybrid MPI+OpenSHMEM models and proposed simple extensions to OpenSHMEM communication. To the best of our knowledge, this is the first such design of any data intensive computing application using the Hybrid MPI+PGAS programming models. Our performance evaluations reveal that the hybrid design improves the performance significantly. At 8,192 processes, the sort execution time is reduced by 45% using the hybrid design as compared to existing design. Performance comparisons with Hadoop, using the same amount of resources (1,024 cores) indicated an improvement of 7X times. Our scalability experiments indicate that hybrid design demonstrates good strong and weak scaling characteristics.

We plan to further enhance the proposed framework to make use of the read hosts also for sorting and global write, as they are idle during the write stage. We also plan to explore the use of accelerator/coprocessors for efficient sorting.

References

[1] The Apache Hadoop Project. http://hadoop.apache.org/.

[2] 2012 DataNami Study. http://www.datanami.com/datanami/2012-07-16/top_5_challenges_for_hadoop_mapreduce_in_the_enter-prise.html.

[3] 2013 IDC Worldwide HPC End-User Study. http://www.idc.com/getdoc.jsp?containerId=prUS24409313.

[4] D. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. Simon, V. VenkataKrishnan, and S. Weeratunga. The NAS Parallel Benchmarks Summary and Preliminary Results. In *Supercomputing, 1991. Supercomputing '91. Proceedings of the 1991 ACM/IEEE Conference on*, pages 158–165, 1991.

[5] A. Basumallik and R. Eigenmann. Optimizing Irregular Shared-memory Applications for Distributed-memory Systems. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, 2006.

[6] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, SPAA '91, pages 3–16, New York, NY, USA, 1991. ACM.

[7] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur. Hybrid Parallel Programming with MPI and Unified Parallel C. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, 2010.

[8] J. Dinan, C. Cole, G. Jost, S. Smith, K. Underwood, and R. W. Wisniewski. Reducing Synchronization Overhead Through Bundled Communication. In *First OpenSHMEM Workshop (OpenSHMEM)*, 2014.

[9] J. Dongarra, P. Beckman, T. Moore, and P. e. a. Aerts. The International Exascale Software Project Roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, Feb. 2011.

[10] T. Graves. GraySort and MinuteSort at Yahoo on Hadoop 0.23. 2013.

[11] Greenplum Analytics Workbench. http://www.greenplum.com/news/greenplum-analytics-workbench.

[12] D. Grunewald and C. Simmendinger. The GASPI API specification and its implementation GPI 2.0. In *The 7th Conference on Partitioned Global Address Space (PGAS)*, 2013.

[13] HPCToolkit. http://hpctoolkit.org/.

[14] J. Jose, K. Kandalla, M. Luo, and D. Panda. Supporting Hybrid MPI and OpenSHMEM over InfiniBand: Design and Performance Evaluation. In *41st International Conference on Parallel Processing (ICPP)*, 2012.

[15] J. Jose, M. Luo, S. Sur, and D. K. Panda. Unifying UPC and MPI Runtimes: Experience with MVAPICH. In *The 4th Conference on Partitioned Global Address Space (PGAS)*, 2010.

[16] J. Jose, S. Potluri, K. Tomko, and D. K. Panda. Designing Scalable Graph500 Benchmark with Hybrid MPI+OpenSHMEM Programming Models. International Supercomputing Conference (ISC), 2013.

[17] Message Passing Interface Forum. http://www.mpi-forum.org/.

[18] MVAPICH2-X: Unified MPI+PGAS Communication Runtime over OpenFabrics/Gen2 for Exascale Systems. http://mvapich.cse.ohio-state.edu/.

[19] C. Nyberg, M. Shah, and N. Govindaraju. Sort Benchmarks. http://sortbenchmark.org/.

[20] OpenSHMEM. http://openshmem.org/.

[21] S. Potluri, P. Lai, K. Tomko, S. Sur, Y. Cui, M. Tatineni, K. W. Schulz, W. L. Barth, A. Majumdar, and D. K. Panda. Quantifying Performance Benefits of Overlap Using MPI-2 in a Seismic Modeling Application. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 17–25, New York, NY, USA, 2010. ACM.

[22] J. Qiang, S. Lidia, and R. D. Ryne. Three-dimensional Quasistatic Model for High Brightness Beam Dynamics Simulation. Phys. Rev. ST Accel. Beams 9.

[23] H. Shan, B. Austin, N. J. Wright, E. Strohmaier, J. Shalf, and K. Yelick. Accelerating Applications at Scale Using One-Sided Communication. In *The 6th Conference on Partitioned Global Address Space (PGAS)*, 2012.

[24] Silicon Graphics International. SHMEM API for Parallel Programming. http://www.shmem.org/.

[25] H. Sundar, D. Malhotra, and G. Biros. HykSort: A New Variant of Hypercube Quicksort on Distributed Memory Architectures. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, pages 293–302, New York, NY, USA, 2013. ACM.

[26] H. Sundar, D. Malhotra, and K. W. Schulz. Algorithms for High-throughput Disk-to-disk Sorting. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 93:1–93:10, New York, NY, USA, 2013. ACM.

[27] S. Sur, H. Wang, J. Huang, X. Ouyang, and D. K. Panda. Can High Performance Interconnects Benefit Hadoop Distributed File System? In *Workshop on Micro Architectural Support for Virtualization, Data Center Computing, and Clouds, in Conjunction with MICRO 2010*, Atlanta, GA, 2010.

[28] TACC Stampede Cluster. www.xsede.org/resources/overview.

[29] The Graph500. http://www.graph500.org.

[30] The MIMD Lattice Computation (MILC) Collaboration. http://physics.indiana.edu/ sg/milc.html.

[31] Top500 Supercomputing System. http://www.top500.org.

[32] K. Underwood. OpenSHMEM on Portals. In *First OpenSHMEM Workshop (OpenSHMEM)*, 2014.