A Study of the Bucket-Exchange Pattern in the PGAS Model Using the ISx Integer Sort Mini-Application

Jacob Hemstad*, Ulf R. Hanebutte†, Ben Harshbarger‡ and Bradford L. Chamberlain‡

*University of Minnesota
Minneapolis, MN 55455
Email: hemst013@umn.edu

†Intel Corporation
Seattle, WA 98104
Email: ulf.r.hanebutte@intel.com

‡Cray Inc.
Seattle WA 98164
Email: {bharshbarg,bradc}@cray.com

***Abstract*—ISx is an open-source[1] integer sort mini-application that was released as a research vehicle for the study of irregular all-to-all communication patterns. The mini-app is uniquely valuable to the PGAS community because its dynamic, data dependent communication pattern presents an opportunity to show the benefit of the PGAS abstraction and one-sided communication. Its original release featured an implementation in OpenSHMEM as well as MPI two-sided for comparison. Our analysis showed the OpenSHMEM version spent up to 65% less time in the communication phase compared to the MPI two-sided implementation. From there, we began to explore porting ISx to other PGAS languages and created a Chapel implementation to compare performance and programmability trade-offs. This paper summarizes some of the results from the first year of research findings using ISx and discusses plans for future work.**

***Index Terms*—Parallel programming, Performance analysis, Computer languages.**

1. Introduction

ISx was introduced to the research community at PGAS 2015 [1] as a new scalable integer sort mini-application. It was inspired by the Integer Sort (IS) kernel found in the seminal NAS Parallel Benchmark Suite [2] first introduced in 1991 [3]. The NAS suite has been widely used in evaluation studies as the kernels are representative of the patterns found in many scientific applications [4], [5], [6]. Researchers at the University of Houston [7] created the first OpenSHMEM implementation of the NPB IS and made it available to the community.

It was shown in [1] that the reference implementation of the NAS IS kernel had several drawbacks for studies on machines of today's scale, including: a non-uniform data distribution with ineffective static load balancing; fixed problem sizes; and an inability to do weak-scaling studies. In order to address these shortcomings, we designed and implemented ISx as a new bucket sort mini-application.

What are the features of ISx?

- Parallel bucket sort featuring an irregular, data dependent *all-to-all* communication pattern
- Demonstrated ability to scale up to 12K cores [1]
- Highly modular application implemented in OpenSHMEM, MPI two-sided, and Chapel
- Supports both strong and weak scaling studies
- Uses a uniform random key distribution and guarantees static load balance
- Includes a verification step

A parallel bucket sort consists of five main stages as illustrated in Figure 1:

1) Key generation
2) Local bucketing
3) Assigning global buckets
4) Key exchange
5) Local sort

The communication required in the exchange of keys makes integer sort a prototypical benchmark for data movement performance. The key exchange requires an irregular *'all-to-all'*, where each PE sends an input dependent amount of data to every other PE—stressing the interconnect's ability to handle large amounts of simultaneous communication. The *all-to-all* usually dominates the execution time of the algorithm and is also a fundamental pattern in many other important applications, e.g. matrix transpose and fast Fourier transforms—making it an important target for analysis and optimization.

The performance of the *all-to-all* is sensitive to the performance of the interconnect and communication primitives used. Furthermore, the irregular nature of the communication makes it ill-suited for a two-sided, message passing pro-

1. https://github.com/ParRes/ISx

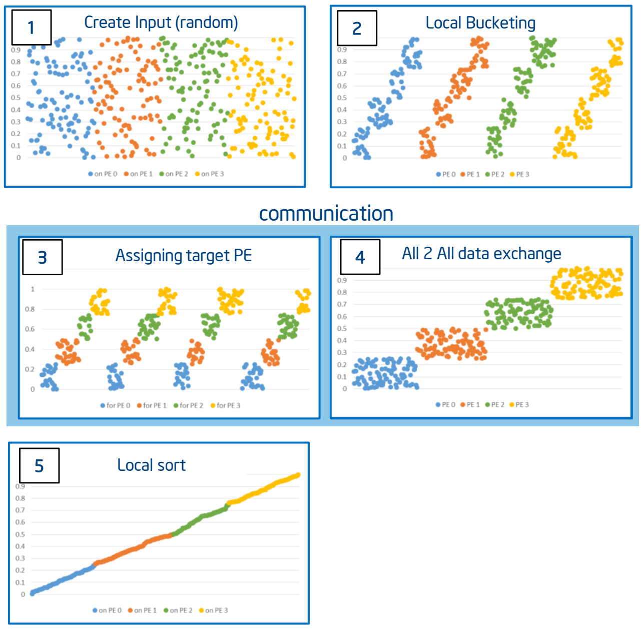

Figure 1. Depiction of the parallel bucket sort phases. For illustrative purposes, only a small number of keys are sorted on 4 processing elements.

gramming model. However, the partitioned global address space (PGAS) model built on one-sided communication is well-suited for this problem of its low communication overhead due to its closeness to the underlying architecture [8].

2. PGAS Community Involvement

Since its inception, ISx was created and open-sourced [9] with the PGAS community in mind. ISx's value to the PGAS community comes from its dynamic, data dependent communication pattern which presents an opportunity to show the benefit of the PGAS abstraction and one-sided communication. In the last year, it has been fruitfully used in several PGAS related activities.

For example, Sandia-OpenSHMEM (SOS) [10] is an implementation of the OpenSHMEM specification over multiple networking APIs including Portals 4 and the Open Fabric Interface (OFI). Early on, the SOS development team integrated ISx into its continuous integration tests. Furthermore, ISx has been used in SOS co-design activities around OFI [11] including development of Libfabric on the Aries interconnect [12].

Additionally, researchers at Rice University created an implementation of ISx in asynchSHMEM [13], their asynchronous many-tasking variant of OpenSHMEM. They used their ISx port to compare performance between OpenSHMEM and asynchSHMEM [14].

Lastly, the Chapel team at Cray Inc. has recently shifted their performance-tuning focus from shared-memory and embarrassingly parallel distributed memory codes to key communication patterns. As a result, the Chapel team chose to target ISx and worked with the ISx authors to create a Chapel implementation. The initial implementation effort took under 4 hours, which demonstrates the productivity advantage of Chapel and the compact nature of the ISx source code. The Chapel team has since evolved the initial implementation with elements unique to the Chapel language. This new implementation is described in more detail in Section 3.

3. Chapel implementation

Chapel is an open-source PGAS language currently in development at Cray Inc. which is designed to simplify parallel programming for the desktop and at scale [15], [16]. The Chapel implementation of the ISx benchmark [17] is included in the public 1.13 release of Chapel, and leverages a number of unique language features to implement an elegant SPMD-style program similar to the SHMEM reference version.

One such feature is the *domain map*, which describes the mapping of domain indices and array elements to *locales*. The *Block* domain map is used to distribute an array of arrays, where the inner arrays act as destination buckets for keys. This *distributed* array is a global variable, allowing each PE to access every other PE's destination bucket:

```
const Space = {0..#numBuckets};
const Dist  = Space dmapped Block(Space);
var recv : [Dist] [0..#recvSize] int(32);
```

Every Chapel program starts as a single task on the root locale. The *coforall* and *on* statements are used to create a task per PE, where the bulk of the work will be done. The *Barrier* standard module is used to coordinate between these tasks.

Once all tasks are created, the Chapel implementation follows the same general steps as the SHMEM reference version. The final unique portion involves the exchange step. Instead of calling a SHMEM function like *shmem_int_put*, the Chapel implementation uses assignment between *array slices* to perform the equivalent *put*:

```
for i in 0..#numBuckets {
  const size, dst, src = ...;
  recv[i][dst..#size] =
    myBucketedKeys[src..#size];
}
```

The left-hand side of the assignment will be a slice of the destination bucket's array, which is visible as an element of the global distributed array created at module scope. The right-hand side of the assignment is the local PE's bucketed keys. The Chapel compiler, modules, and runtime work together to insert communication calls in order to perform the assignment between these two array slices. For assignment between dense arrays, like those used in ISx, Chapel implements a bulk-transfer optimization. This optimization performs a single large *put* containing all elements, instead of many fine-grained communication calls for each element in the array slice.

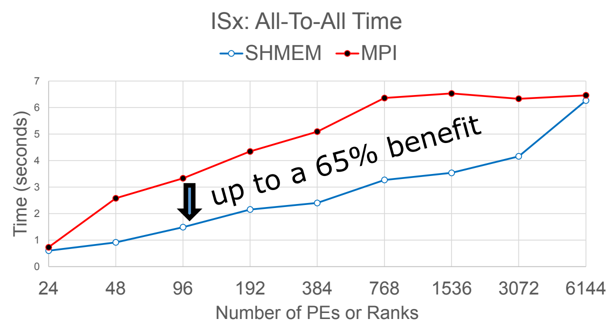

Figure 2. Weak scaling comparison of MPI to SHMEM on Cray XC30 with 2^{31} initial keys per rank.

4. Evaluation

4.1. MPI vs. SHMEM Comparison

In order to study the performance of ISx under a one-sided PGAS model versus the two-sided message-passing model, we conducted a comparison of the OpenSHMEM version and the MPI two-sided implementation. These results were gathered on Edison [18], a Cray® XC30™ at NERSC that utilizes a Dragonfly network topology with an Aries™ interconnect [19]. A weak scaling comparison of time spent in the *all-to-all* communication phase for the MPI and SHMEM version of ISx is shown in Figure 2. For each rank size, we submitted 10 unique jobs and recorded the total time across all PEs for the communication phase. We are reporting the minimum of those 10 timings. For this weak scaling study, the initial number of keys randomly generated by each rank was 2^{31}.

Figure 2 shows that the OpenSHMEM implementation had up to a 65% reduction in time spent in the communication phase compared to the MPI two-sided implementation. There is no performance difference at 24 ranks because only one node was used and therefore we do not expect any difference. Similarly, there is no performance difference at 6144 ranks because the number and size of the messages has saturated the network bandwidth and therefore we do not expect any difference. We attribute the performance difference between these two points to the more efficient one-sided communication primitives used in the OpenSHMEM implementation.

4.2. Random vs. Round Robin Exchange

The all-to-all communication phase of ISx dominates execution time for sufficiently large problems. The implementation of the all-to-all in ISx is performed by a loop of point-to-point communications over all the processing elements (PEs). In the naïve strategy, every PE carries out the loop in an identical fashion, i.e. first send to PE0, second to PE1, etc. The problem with this approach is that it results in an *in-cast*—where a PE receives a message from all other PEs at approximately the same time. An in-cast of sufficient size can overwhelm the interconnection network and have a dramatic, negative impact on performance. Therefore, we studied two alternate schemes on Edison for carrying out the all-to-all communication:

- *Random*: each PE sends its payload to every other PE in a random order
- *Round Robin*: a PE with id p first sends to PE $p+1$, second to $p+2$, and eventually circles back around to $p-1$ [20]

Both of these alternate schemes attempt to spread out the communication across the network and avoid the performance hit caused by in-casts. However, the Round Robin scheme may lead to an in-cast at intermediate points in the network topology. Furthermore, consecutive indexing does not necessarily indicate close locality because Edison schedules jobs on a first-fit basis across the entire system.

Figure 3 shows the observed per node bandwidth when performing weak scaling runs with the SHMEM version of ISx on 12k and 16k PEs. Each run is comprised of 20 timed bucket sort iterations with *all-to-all* exchange. To guarantee the same job layout for both the random and the round robin run, we submitted both as a single batch job. Data was collected for 2 submissions each at 12k and 16k PEs. The average observed bandwidth per node is shown with the observed range. For both job sizes, the random scheme resulted in higher per node bandwidth than the round robin. The highest per node bandwidth is observed with the random scheme at 12k PEs. The lowest per node bandwidth is observed for the round robin approach at 16k PEs.

The Dragonfly network of the Cray® XC30™ can be configured with fewer than the maximal number of optical links than the system architecture supports. For the Edison system, the network has 6 of the max 17 optical cables per group pairs [21] provisioned. Thus the Edison network provides 35% of the peak global bandwidth of a fully connected Dragonfly which supports up to 11.7GB/s per node [22]. Therefore, as configured, Edison supports up to 4 GB/s per node and the bandwidth for a 64% payload efficiency [22] is 2.6 GB/s. The average bandwidth observed at the application level for ISx scheduled during normal operating times is in the range of 1.5 GB/s to 2.6 GB/s, which is near the limit we derived.

4.3. Chapel vs. SHMEM Comparison

The Chapel implementation is compiled with the flags `--fast` and `-snoRefCount`. `--fast` disables runtime checks and enables optimizations for the backend C compiler (gcc 5.1.0 for this study). `-snoRefCount` disables reference counting for Chapel arrays, domains, and distributions. The 1.13 release of Chapel takes a conservative approach to reference counting, which can hurt performance in benchmarks like ISx that involve remote array slicing.

The 1.13 Cray release module was used to gather these results, and was configured to use the *ugni* communication layer and the *qthreads* tasking layer. The scaling method

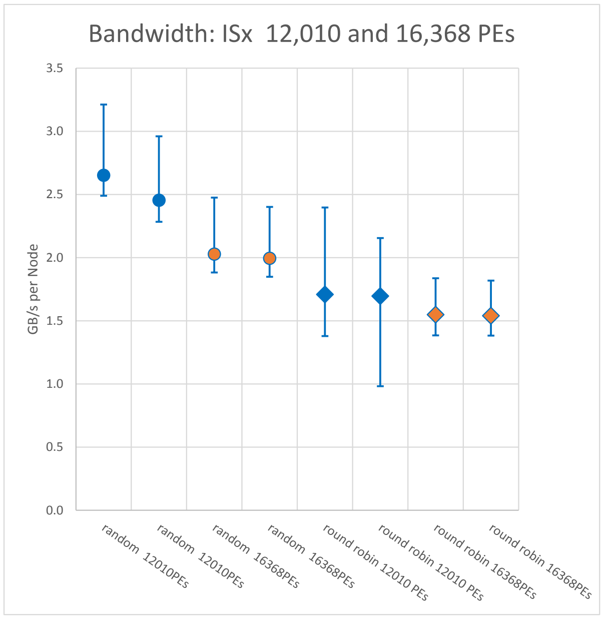

Figure 3. Observed bandwidth per node comparing random vs. round robin for 12k and 16k PEs on NERSC Edison system with 24 PEs per node. Circles are used for the Random scheme and diamonds for round robin. Blue indicates 12K PEs were used and orange indicates 16K PEs. Error bars indicate the minimum and maximum of the observed bandwidth range.

was set to `weakISO` and the problem size was set to 2^{27} keys per bucket.

Figure 4 shows the results of the ISx weakISO scaling problem. These results were gathered on an Cray® XC30™ system internal to Cray with 36 Broadwell cores per compute node. The Chapel version achieves roughly half the performance as the SHMEM reference version at all compute node levels. We believe the additional overhead to be due to the array slice of the likely-remote destination bucket array. In the 1.13 release, Chapel's implementation of an array slice constructs a new class object and initializes metadata to represent the indices of the subdomain used to slice the array. Some optimizations for arrays rely on the underlying data pointer being located on the same locale as the metadata class. This is accomplished by performing an *on-statement* onto the locale on which the sliced array lives. This can be a blocking remote operation, and will create a new task on the destination locale. With each PE slicing the destination array of every other PE, the number of remote operations and tasks result in significant overhead as the number of compute nodes increases.

In order to address this performance issue we will explore the option of implementing a compiler optimization targeted at sliced array assignment. This optimization will need to provide the following information to the existing bulk-transfer runtime calls:

- A pointer to the source data
- A pointer to the destination data
- The locale where the remote data lives

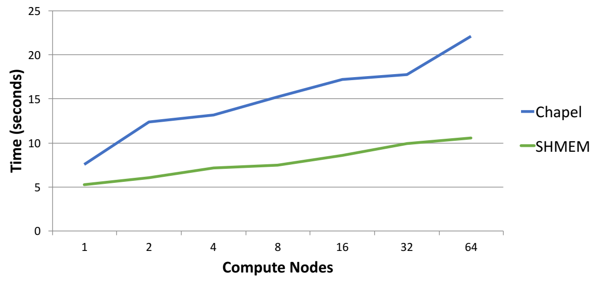

Figure 4. Comparison of Chapel to SHMEM for total execution time on a Cray XC30 (36 PEs per node)

- The size of the transfer

Prior to the slice it is already possible to determine where the remote data lives by using Chapel's locale-query language support. If we restrict this optimization to apply to dense 1-dimensional arrays, we should be able to infer the remaining information from the domains used to slice the arrays. The lower bound of the domain provides the offset needed to build the pointer to the source or destination data, and the size of the domain is equivalent to the size of the transfer. With all of the required information available without needing to slice the array, we should be able to avoid the costly on-statement entirely.

5. Future Work

As the community has started to utilize ISx, we have to balance new development with maintaining consistency. For example, when we updated ISx code to be compliant with the OpenSHMEM 1.3 standard, we included backwards compatibility with Cray SHMEM.

Driven by our own observations and by community feedback, we have recently started the development of version 1.2. Our plans for version 1.2 are:

- **Bucketization**: We can replace the integer division used to determine to which bucket a key belongs with a bit shift if the size of the buckets is a power of 2. We have implemented this in the development branch and observed on the Edison system a 20% reduction in non-communication time of ISx for weak scaling problem with 2^{27} keys per PE.
- **Support of larger key sizes**: We intend to include explicit options for two key sizes (signed int 32 bit and unsigned 64). Note that Chapel already supports a 64 bit random number generator based on PCG [23], the same random number generator utilized by the reference version of ISx.
- **Communication model integration**: We plan to extend the current performance instrumentation with a communication model to provide bandwidth metrics and statistics.
- **Hybrid implementation**: Discussions are under way on a hybrid ISx implementation utilizing a threading model such as OpenMP combined with OpenSHMEM

or MPI. This is of particular interest to the OpenSHMEM standards community, as it is actively working on threading support.

6. Conclusion

We created and open-sourced the ISx bucket sort mini-application as a vehicle for research and co-design in the PGAS community. Its bucket-exchange pattern is a prototypical example of data-dependent, irregular all-to-all communication, not limited to sorting. Our experiments on Edison showed that the OpenSHMEM implementation spent up to a 65% less time in the communication phase compared to the MPI two-sided implementation. This demonstrates where the PGAS abstraction and one-sided communication can outperform traditional message-passing approaches. Furthermore, the adoption and use of ISx by other members of the PGAS community has lead to valuable insights and directions for further research. This can be seen in efforts of the Chapel team who used ISx to identify opportunities for optimizing the Chapel implementation.

Acknowledgments

This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] U. Hanebutte and J. Hemstad, "ISx: A Scalable Integer Sort for Co-design in the Exascale Era," in *9th International Conference on Partitioned Global Address Space Programming Models (PGAS), 2015*, Sept 2015, pp. 102–104.

[2] "NAS Parallel Benchmarks," https://www.nas.nasa.gov/publications/npb.html.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks: Summary and Preliminary Results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 158–165. [Online]. Available: http://doi.acm.org/10.1145/125826.125925

[4] H.-Q. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," 1999.

[5] J. Subhlok, S. Venkataramaiah, and A. Singh, "Characterizing NAS benchmark performance on shared heterogeneous networks," in *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*. IEEE, 2001, pp. 9–pp.

[6] S. Seo, G. Jo, and J. Lee, "Performance characterization of the NAS Parallel Benchmarks in OpenCL," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 137–148.

[7] S. Pophale, R. Nanjegowda, T. Curtis, B. Chapman, H. Jin, S. Poole, and J. Kuehn, "OpenSHMEM performance and potential: A NPB experimental study," in *The 6th Conference on Partitioned Global Address Space Programming Models, PGAS*. Citeseer, 2012.

[8] J. Dinan, C. Cole, G. Jost, S. Smith, K. Underwood, and R. W. Wisniewski, "Reducing Synchronization Overhead Through Bundled Communication," in *Proceedings of the First Workshop on OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools - Volume 8356*, ser. OpenSHMEM 2014. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 163–177. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-05215-1_12

[9] "ISx Open-Source," https://github.com/ParRes/ISx.

[10] "Sandia OpenSHMEM," https://www.github.com/Sandia-OpenSHMEM/.

[11] K. Seager and R. Grant, "Introducing Sandia OpenSHMEM with Multi-Fabric Support Using LIBFABRIC," in *12th Annual OpenFabric Alliance Workshop, Monterey, CA, April 4 - 8, 2016*. [Online]. Available: https://www.openfabrics.org/images/eventpresos/2016presentations/407OpenSHMEM.pdf

[12] K. Seager, S. Choi, J. Dinan, H. Pritchard, and S. Sur, "Design and Implementation of OpenSHMEM using OFI on the Aries Interconnect," in *OpenSHMEM Workshop 2016, Baltimore, MD, August 2-4, 2016*.

[13] "Asynchronous Extensions to OpenSHMEM," https://github.com/openshmem-org/openshmem-async.

[14] M. Grossman, V. Kumar, Z. Budimlic, and V. Sarkar, "Integrating Asynchronous Task Parallelism with OpenSHMEM," in *OpenSHMEM Workshop 2016, Baltimore, MD, August 2-4, 2016*.

[15] B. L. Chamberlain, "Chapel," in *Programming Models for Parallel Computing*, P. Balaji, Ed. MIT Press, November 2015, ch. 6, pp. 159–160.

[16] "The Chapel Parallel Programming Language," http://chapel.cray.com/.

[17] B. Chamberlain, B. Harshbarger, L. Duncan, and J. Hemstad, "ISx in Chapel: Early Observations and Results," in *CHIUW 2016: Chapel Implementers and Users Workshop, Chicago, IL, May 27, 2016*, 2016.

[18] " NERSC Edison home page," https://www.nersc.gov/users/computational-systems/edison/.

[19] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, R. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard, "Cray cascade: a scalable HPC system based on a dragonfly network," in *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, J. K. Hollingsworth, Ed. IEEE/ACM, 2012, p. 103. [Online]. Available: http://dl.acm.org/citation.cfm?id=2389136

[20] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of Collective Communication Operations in MPICH," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 1, pp. 49–66, Feb. 2005. [Online]. Available: http://dx.doi.org/10.1177/1094342005051521

[21] "NERSC Edison Configuration," http://www.nersc.gov/users/computational-systems/edison/configuration/interconnect/.

[22] "Cray® XC™ Series Network: WP-Aries01-1112," http://www.cray.com/sites/default/files/resources/CrayXCNetwork.pdf.

[23] M. E. O'NEILL, "Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation," http://www.pcg-random.org/pdf/toms-oneill-pcg-family-v1.02.pdf.