Optimizing MPI Collectives on Intel MIC Through Effective Use of Cache

Pinak Panigrahi[1], Sriram Kanchiraju[2], Ashok Srinivasan[3], Pallav Kumar Baruah[4], C D Sudheer[5]
[1,2,4]Dept. Of Mathematics and Computer Science, Sri Sathya Sai Institute of Higher Learning,
Prashanthi Nilayam, India
[3]Dept. Of Computer Science, Florida State University, Tallahassee, USA
[5]IBM Research India, New Delhi, India
{panigrahi.pinak[1], sriramkanchiraju[2]}@gmail.com,
asriniva@cs.fsu.edu[3], pkbaruah@sssihl.edu.in[4], sudheer.chunduri@in.ibm.com[5]

***Abstract*—The Intel MIC architecture, implemented in the Xeon Phi coprocessor, is targeted at highly parallel applications. In order to exploit it, one needs to make full use of simultaneous multi-threading, which permits four simultaneous threads per core. Our results also show that distributed tag directories can be a greater bottleneck than the ring for small messages when multiple threads access the same cache line. Careful design of algorithms and implementations based on these results can yield substantial performance improvement. We demonstrate these ideas by optimizing MPI collective calls. We obtain a speed-up of 9x on barrier and a speed-up of 10x on broadcast, when compared with Intel's MPI implementation. We also show the usefulness of our collectives in two realistic codes: particle transport and the load balancing phase in QMC. Another important contribution of our work lies in showing that optimization techniques - such as double buffering - used with programmer controlled caches are also useful on MIC. These results can help optimize other communication intensive codes running on MIC.**

***Index Terms*—Intel Xeon Phi; MIC; MPI; barrier; broadcast; shared memory; double buffering;**

I. Introduction

Intel Xeon Phi is a co-processor that is based on the Intel Many Integrated Core (MIC) architecture. It consists of 61 cores which are capable of 4-way simultaneous multi-threading. Data movement is often a performance bottleneck with a large number of threads or processes. Consequently, applications can often benefit through efficient use of the memory subsystem. We provide more details on this architecture and its memory subsystem in Sec II. There has been much interest in characterizing the performance of the memory subsystem, as summarized in Sec III. These results have often been obtained using one thread or process per core. We have extensively characterized the performance of the memory subsystem with more than one simultaneous thread or process per core. We present our significant results in Sec IV. In particular, we show that use of remote L2 caches is much more promising than suggested by earlier work that did not use simultaneous multi-threading in studying the memory subsystem.

We show how applications can use our results to minimize data movement overheads. For example, while double buffering is commonly used with programmer controlled caches, such as shared memory in GPUs, it is also effective on the more traditional cache of Xeon Phi. We use MPI collective calls to demonstrate the optimizations. We have implemented four collective calls, but describe, in Sec V, algorithms only for *barrier* and *broadcast* due to space constraints. We show in Sec VI that our best barrier implementation improves on Intel's MPI by a factor of up to 9. For small messages, our broadcast improves on that of Intel's MPI by a factor of up to 10, and for large messages, by a factor of up to 4. We discuss the impact of architectural features on the performance of different algorithms for these collectives and also demonstrate improvement in performance of actual applications using these collective calls. We summarize our conclusions in Sec VII.

II. Xeon Phi Architecture

The Intel Xeon Phi co-processor contains 61 cores running at 1100 MHz. One of the cores is used to run a simplified linux kernel. Each core can support four hardware contexts simultaneously. Every core has a 32 KB L1 data cache, 32 KB L1 instruction cache and a 512 KB private L2 cache. The L2 caches are kept coherent using a Distributed Tag Directory (DTD) system. If data is not found in a core's local L2 cache, then it is sought from a remote L2 cache. If it is not found in a remote L2 cache, then it is brought in from memory. The cores, the tag directories, the memory controllers, and PCIe are connected by a bidirectional ring. The memory addresses are mapped in a pseudo-random fashion to the 64 tag directories [1] and 8 memory controllers using a hash function, leading to an even distribution on the ring. Fig. 1 depicts the basic blocks of the MIC architecture.

III. Related Work

Various aspects of a Xeon Phi-based system were studied using microbenchmarks in [2], and the performance of OpenMP directives and MPI functions were also evaluated in it. The performance of sparse matrix multiplication kernels – which are memory bound – was evaluated on the Xeon Phi in [3]. The memory subsystem performance of the Xeon Phi has been benchmarked in [4], where the authors identified some factors that need to be considered while designing kernels for a pre-production model of Xeon Phi. Their analysis

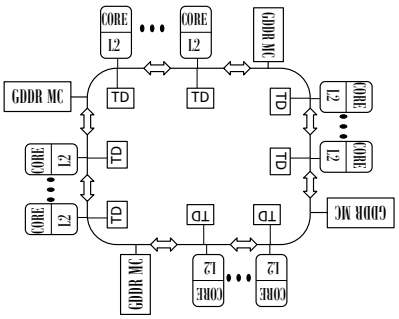


Fig. 1. Architecture of Intel MIC.

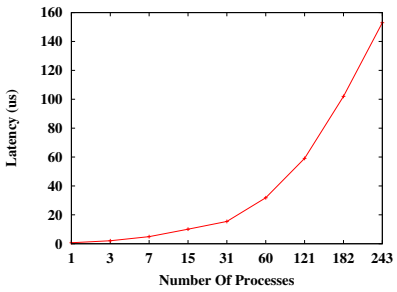


Fig. 2. Processes accessing a single cache line simultaneously.

deals with a small number of total threads – 4, and on the L1 cache, which is private. In contrast, we deal with a large total number of threads, up to 244, and at higher levels of the memory subsystem.

Inter-node collective communication in MIC clusters is optimized in [5]. In contrast, we have optimized intra-MIC collective communication. Communication between the cores in a cache coherent system is modeled in [6] using Xeon Phi as a case study. Using their model, they designed algorithms for collective communication. However, these are thread based implementations while ours are based on MPI processes. Point-to-point communication primitives are optimized for the Xeon Phi in [7]. They designed optimal algorithms for Gather, Alltoall, and Allgather using these point-to-point primitives. The above two works have not considered cases with large number of processes or threads – they consider 61 or fewer while Xeon Phi permits up to 244 simultaneous ones.

In contrast, we have analyzed the performance of the memory subsystem with up to 244 processes, and use our conclusions to develop efficient collective communication implementations that scale well. Our collectives are not built on top of point-to-point primitives. Rather, all processes directly use a common region of shared memory. Such use of shared memory has been shown to be more efficient than implementing collectives on top of point-to-point primitives [8] on the Opteron. In earlier work, we had used double buffering on the programmer controlled caches (local stores) of the Cell processor to optimize MPI [9]. Our current work shows that double buffering can be effective on the Xeon Phi too. Since the caches on this machine are not programmer controlled, we induce the same benefits by careful design of the algorithms.

IV. Performance of the Memory Subsystem

Understanding the memory bandwidth, memory access latency or the cache-to-cache bandwidth on the Xeon Phi is difficult because the latency potentially depends on not only contention and the distance from the core to the specific distributed tag directory for a given cache line, but also on the distances from the distributed tag directory to the memory controller and from the memory controller back to the requesting core. There are conflicting results in existing literature about the effect of distance between the cores. Ramos et. al. [6] have concluded that communication with the DTD dominates the access time and hence the distance between the cores is not a matter of concern. However, Fang et. al. [4] conclude that the distance between the cores does play a role in cache line transfer times. Our results (not presented here) are consistent with the former.

We use the STREAM benchmark [10] in our experiments to study the memory bandwidth. There are several other studies that have analyzed the performance of the STREAM benchmark on MIC. They report results when each thread accesses around 10 MB of data. When using such a message size, we obtain a memory bandwidth of 163 GB/s that is comparable to the results reported in [4] and [2]. However, we wish to consider situations involving MPI messages of smaller sizes, which are common in many applications. For example, an analysis of workloads of scientific applications used at Argonne National Lab shows that the average message sizes for collective communication calls range from around 10 B to the order of 100 KB [11]. We, therefore, focus more on smaller sizes, where the performance tends to be lower than for large message sizes, and compare the results with cache-to-cache transfers. Note that cache-to-cache transfers on the Xeon Phi take the shortest path on the ring.

Whenever there is a cache miss on a core, the DTD is queried. Hence, when many processes access the same cache line, all of them will generate requests to the same tag directory. This will result in contention at the tag directory and the performance will depend on the speed at which the DTD services these requests. This is benchmarked by having all receiving processes copy a cache line from a root process into their private buffer. The results are averaged over 1000 accesses. There is a barrier call before every access to ensure that all processes access the cache line simultaneously.

When many processes access a single cache line simultaneously, the latency increases linearly with the number of processes as shown in fig. 2, with a slightly higher than linear increase with a large number of processes. This is caused by contention at the tag directories.

We also study the effect of this contention when receiving processes copy larger message sizes from the root process simultaneously. We observed that when processes copy a message that consists of more than a single cache line simultaneously, the latency does not increase as much as expected from the results of the case where processes copy a single

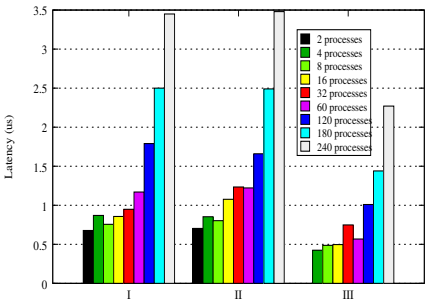


Fig. 3. Processes accessing cache lines across different caches with three different data access patterns.

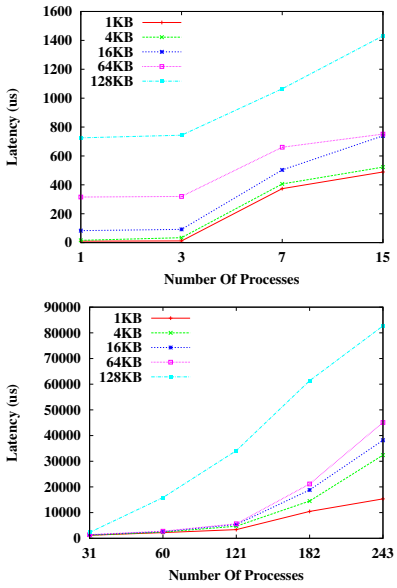


Fig. 4. Processes copying the same data from a single process simultaneously.

cache line simultaneously. This observation is shown in fig. 4. As a direct consequence of the hashed mapping that exists between the memory addresses and the tag directories, requests for consecutive memory addresses do not go to the same tag directory. Hence, when accessing data with spatial locality, the memory requests are sent to different tag directories. Therefore, contention at a particular DTD is reduced. Another factor that can be influencing the performance in this scenario is prefetching. When data is accessed with spatial locality, prefetching helps in hiding the latency. There is, perhaps, a tradeoff between contention and prefetching that is responsible for this behavior.

We have also studied how accessing cache lines across different caches impacts the performance. We designed several benchmarks to study this, and report three typical results. Let P be the number of processes. In the first one (I), process i copies data from process $(i + 1) \bmod P$. In the second one (II), process i copies data from process $(i+2) \bmod P$. In the third one (III) 4 processes are grouped together, and they communicate in an interleaved fashion. The results are shown in fig. 3. When there are 61 or fewer processes involved, there is not much of a difference in latency, as observed by others too. However, we find that there is a significant increase beyond that point. On the other hand, the increase is still small compared with that for accessing a single cache line as shown in fig. 2. This shows that contention at the DTDs has a greater effect on the performance than contention on the ring.

We summarize the following conclusions regarding the performance of the memory subsystem.

- The aggregate bandwidth for cache-to-cache transfers is more than the memory bandwidth and increases when the number of processes is increased. This makes techniques like double buffering potentially effective.
- It is advantageous to have processes access different cache lines as compared to all of them accessing the same cache line.
- When many processes simultaneously read data that consists of more than a single cache line, the performance does not degrade as significantly as we would expect from the single cache-line results.
- The effect of contention on the ring caused by many processes communicating is much smaller than the effect of DTD contention.

V. Algorithms for Intra-MIC Collective Communication

We now present the algorithms for barrier and broadcast. We used POSIX shared memory as a communication channel between the processes using POSIX functions *shm_open* and *mmap*. In our approaches, every collective call requires a shared memory segment to be associated with it. The size of the segment varies from one collective to the other. We assign shared memory of a fixed size to each MPI communicator. This region is contiguous and is memory mapped at the time of the creation of the communicator and destroyed with it. We design our algorithms and implementations based on our observations on the performance of the memory subsystem. We explain our rationale further in Sec VI.

Barrier. Let the number of processes be P. All the algorithms except dissemination, follow the *gather/broadcast* style. In this approach, a designated process, called the root, waits for all the processes to enter the barrier. After that, the root broadcasts this information to all processes. When a process receives the broadcast, it exits the barrier. All the algorithms use a shared memory segment of size 64 P bytes unless otherwise mentioned. We use the term flag to refer to the cache line associated with the process. For example, when we say that a process i sets a flag on process j, we mean that process i sets a value in the cache line belonging to process j.

Centralized: In Centralized-I, the shared memory segment consists of P bytes (1 byte per process is used as a flag). In the *gather* phase, a process sets its flag and polls on it until it is unset. The root process unsets the flags after all the processes have entered the barrier. A process exits the barrier only when its flag has been unset. The Centralized-II algorithm works on the same principle except that it uses a cache line per flag.

Dissemination [12]: In the *k*th step, process *i* sets a flag on process $i + 2^k$ (*mod* P) and polls on its flag until it is set by process $P + i - 2^k$ (*mod* P). This algorithm takes $\lceil log_2 P \rceil$ steps.

Tree [13]: The processes are logically mapped to the nodes of a tree of degree k. Every process calculates its children using the following formula: $(k \times rank) + i$, where $i = \{i \in N \wedge (1 \leq i \leq k) \wedge (1 \leq i \leq P)\}$. We have modified the algorithm slightly from [13] to avoid false sharing; in the *gather* phase, each node polls on the flags of its children till they are set before setting its own flag. In the *broadcast* phase, every node unsets the flags of its children before exiting the barrier. The algorithm takes $2\lceil log_k(P) \rceil$ steps.

Tournament [12]: In each round of the *gather* phase of the algorithm, processes are paired and the winner goes to the next round. A process with lower rank is considered to be the winner and waits for its flag to be set by its losing partner. The overall winner initiates the broadcast phase. The broadcast phase is similar to the one used in the *tree* algorithm. The *gather* phase takes $\lceil log_2(P) \rceil$ steps and the *broadcast* phase takes $\lceil log_k(P) \rceil$ steps.

Binomial Spanning Tree (BST) [14]: The processes are logically mapped to a binomial spanning tree. The working principle differs from the *tree* algorithm only in the fashion in which it constructs the tree. Each process calculates its children by adding 2^i to its rank, where $i = \{i \in N \wedge (log_2(rank) < i < \lceil log(P) \rceil) \wedge (rank + 2^i < P)\}$.

Broadcast. We now describe our algorithms for broadcast[1]. The shared memory segment allocated for broadcast is divided into two logical regions, namely, a *data* region, which is used by the root to store data and a *notification* region, which is used for notification purposes. There is a designated flag per process in the *notification* region which is used for synchronization.

Centralized: In this implementation, the root copies the message into the *data* region and notifies the receiving processes via the *notification* region by setting their flags. The receiving processes copy out the message from the *data* region after their flags have been set. When the size of the message exceeds the size of the data region, the message is divided into chunks that can individually occupy the whole data region. The root copies a chunk into the *data* region and notifies the receiving processes. After the receiving processes copy out the message from the *data* region, they unset their flags in the *notification* region. When all the flags of all the processes are unset, the root copies the next part of the message into the *data* region and notifies the receiving processes. This procedure is repeated until the entire message is broadcast.

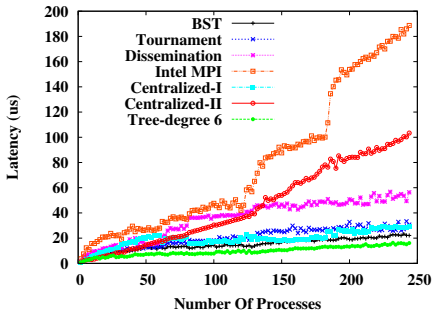
Double Buffering: This algorithm varies from the previous one only in the case where the message size exceeds the size of the *data* region. We use the idea of double buffering in these cases. The *data* region is divided into two parts, each of them is used as a buffer by the root. We introduce an extra flag per process in the *notification* region so that each of the buffers has a corresponding notification flag. The root fills the first buffer and notifies the receiving processes by setting the flags corresponding to the first buffer. While the receiving processes are copying out the data from this buffer, the root copies the next part of the message into the other buffer and notifies the receiving processes by setting the flags corresponding to the second buffer. Before copying the next part of data into the first buffer the root waits for the receiving processes to unset the flags that correspond to first buffer and similarly with the second buffer. These steps are repeated until the whole message is broadcast.

[1]We described a preliminary implementation in [15], which did not take into account the memory subsystem's performance characterization.

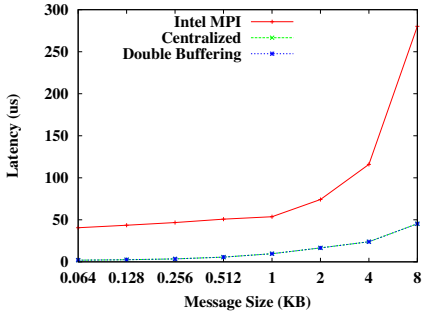
VI. Performance Evaluation

Our experimental platform is the Texas Advanced Computing Center's (TACC) Stampede supercomputer. The nodes that we used had an Intel Xeon Phi SE10P with 61 cores running at 1100MHz. The host machine is a dual 8-core Intel Xeon E5-2680 Sandybridge running at 2.70 GHz. The Intel MIC software stack is the MPSS Gold-update 3 with the Intel compiler v13.1.1.163 and Intel MPI v4.1.1.036. The performance measurements were taken using a simple benchmark code that uses a loop to call the routines 1000 times. The average of the obtained timings were used as the result.

Barrier Performance. The algorithms for barrier take advantage of the fact that a remote cache access is faster than access to the memory. Increasing the tree degree beyond 6 in the *tree* algorithm does not help. We therefore use a tree of degree 6 when comparing the various algorithms and also in the *broadcast* phase of the BST algorithm. The algorithms that are based on tree structures perform better than others. This is because in the tree algorithms processes access different cache lines across various caches. We have shown that this is a favorable situation because accessing different cache lines does not cause DTD contention. In the *Centralized-I* algorithm, many processes access the same cache line. This creates contention at the DTD. In the *Centralized-I* algorithm, the flags corresponding to the various processes are consecutive bytes in memory. Hence, in the broadcast phase, when the root has to set the flags of each of the processes, the operations are vectorized by the compiler. This gives it an advantage over the *Centralized-II* algorithm. In the *Centralized-II* algorithm, every process is polling on its flag to be unset by the root process. So, while every other process wants to read, the root wants to perform a write. In the worst case, there will a cache miss for every process that the root is updating. A similar situation arises in the *dissemination* algorithm; the flag of a process is updated by a different process in each round. Each time the cache line is updated, it becomes a candidate to be invalidated in the other caches. Another factor that should be influencing the performance is the fact that in every round of the algorithm there are P processes communicating across the cores, whereas in the tree algorithm there are much fewer processes communicating across the cores in a typical step. In the worst case, the number of processes simultaneously communicating will be less than P in the *tree* algorithm,

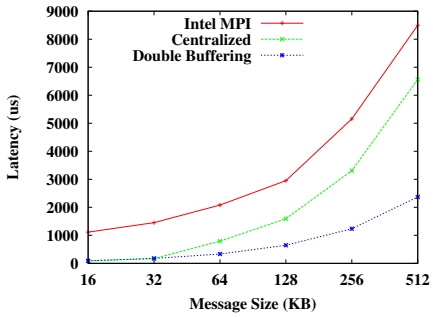


(a)

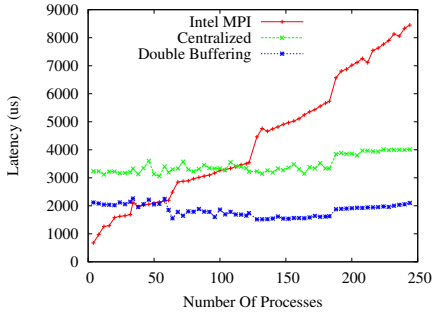


(b)

Fig. 5. *(a)* Comparison of our barrier implementations with that of Intel MPI. *(b)* Performance of 244 process broadcast when the message size is less than the size of the shared buffer.



(a)



(b)

Fig. 6. *(a)* Performance of 244 process broadcast for larger message sizes. *(b)* Performance of broadcast with 512 KB messages.

with typical steps using much fewer. A tree of degree higher than 2 will also have fewer steps than the *dissemination* algorithm. Fig. 5 *(a)* compares our barrier implementations with that of Intel MPI. The best of our algorithms improves the performance by up to a factor of 9.

When the number of processes increases, our implementation incurs a small additional performance penalty. Intel's MPI, on the other hand, shows a large increase in time, with significant jumps at multiples of 61 processes. Our *Dissemination* and *Centralized-II* implementations do not scale well for reasons given above. But the other algorithms scale well.

Broadcast Performance. Benchmarking results showed that when many processes copy the same message, the latency incurred scales well if the message is big enough. Our algorithms exploit this feature to extract good performance. We have fixed the shared buffer size to be 32 KB. Fig. 5 *(b)* shows the performance of our broadcast when message sizes are smaller than the shared buffer, with 244 processes. Results for other process counts are similar. Our algorithms improve the performance over Intel MPI by a factor of 10 for message sizes less than 256 B and up to a factor of 4-5 for larger message sizes.

When the message size exceeds the size of the shared buffer, the *centralized* algorithm suffers because of the single buffering approach. After copying out a part of the message from the buffer, the receiving processes have to wait for the root process to refill the buffer before they can start copying out the next part of the message. Double buffering brings with it the advantage that two steps happen concurrently; while the receiving processes are copying the data out of the *data* region, the root writes the next part of the message into a different buffer. This will reduce the time receiving processes spend waiting for the root to copy the next part of the message. Also, this action by the root will result in accesses to its private L2 cache and will not generate any requests to the DTDs except the first time it uses a buffer. This explains the additional advantage double buffering gives as the message size is increased. Double buffering improves performance by a factor of 1.5-2 over *Centralized*, which uses a single buffer. Figure 6 *(a)* shows the performance of our broadcast for larger message sizes. Our broadcast[2] gives a performance improvement of up to a factor of 4 over Intel MPI. The scalability of our algorithms can be seen from the fact that our algorithms hardly incur any extra latency when the number of processes is increased from 2 to 244, while Intel MPI incurs an extra latency of a factor of 10, as shown in fig. 6 *(b)*. The ability of all processes to access the same shared memory is beneficial for algorithms like broadcast that deal with data that is common to all processes.

We now show the effectiveness of our implementations of two other collectives which we lack space to describe. A *reduce* operation is used in a *Particle Transport* application, which tracks a set of "Random Walkers" in a Monte Carlo simulation [16]. We use *all-gather* in a load balancing com-

[2]Our experiments did not show performance degradation when the core that runs the operating system was included in those hosting MPI processes. Hence, we report results by using all the available cores on the Xeon Phi.

putation described in [17]. Figure 7 shows the percentage improvement in time (higher is better) over Intel MPI.

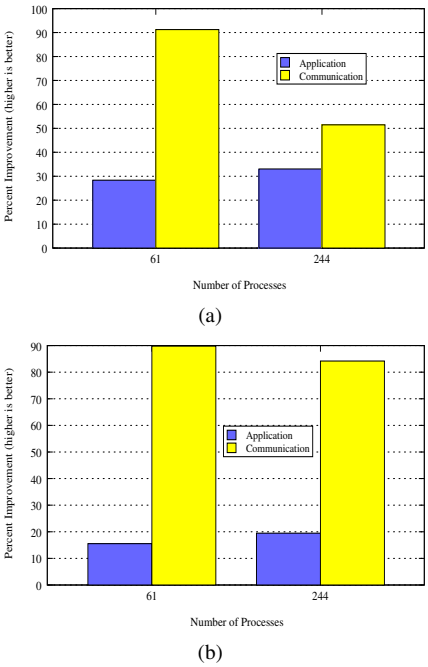


Fig. 7. Performance gains obtained by our Intra-MIC collective implementations relative to Intel MPI for the *(a) Particle Transport* and *(b) Load Balancing* codes.

VII. Conclusions

Through micro-benchmarking, we have identified architectural characteristics of the memory subsystem that impact application performance. We showed that accessing remote caches in certain ways produces substantial performance benefits over accesses to the memory, in contrast to suggestions in existing literature. We showed that optimization techniques such as double buffering, which have been used at the cache level with programmer controlled caches, can also be used effectively on MIC, even though it does not provide a programmer controlled cache. Using these ideas, we have developed algorithms and implementations for MPI collectives. We obtained an improvement in performance by up to a factor of 9 on barrier, an improvement of up to factor 10 for small broadcasts, and factor 4 for large broadcasts. Our algorithms and implementations are also more scalable than the existing Intel MPI implementation. Furthermore, since our optimization techniques are based on the fundamental characteristics of the memory subsystem, they can be used by non-MPI applications to optimize their performance. We have also demonstrated the effectiveness of our reduce and all-gather implementations on practical applications.

Acknowledgment

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575. We would also like to thank John McCalpin for his useful comments on the performance of the STREAM benchmark and issues related with the sustained memory bandwidth.

References

[1] Intel Xeon Phi Coprocessor System Software Developers Guide. http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-system-software-developers-guide.

[2] Subhash Saini, Haoqiang Jin, Dennis Jespersen, Huiyu Feng, Jahed Djomehri, William Arasin, Robert Hood, Piyush Mehrotra, and Rupak Biswas. An early performance evaluation of many integrated core architecture based sgi rackable computing system. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 94:1–94:12, New York, NY, USA, 2013. ACM.

[3] Erik Saule, Kamer Kaya, and Umit V Catalyurek. Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi. *arXiv preprint arXiv:1302.1078*, 2013.

[4] Jianbin Fang, Ana Lucia Varbanescu, Henk Sips, Lilun Zhang, Yonggang Che, and Chuanfu Xu. Benchmarking Intel Xeon Phi to Guide Kernel Design. Technical Report PDS-2013-005, Delft University Of Technology, 2013.

[5] K. Kandalla, A. Venkatesh, K. Hamidouche, S. Potluri, D. Bureddy, and D.K. Panda. Designing optimized mpi broadcast and allreduce for many integrated core (mic) infiniband clusters. In *High-Performance Interconnects (HOTI), 2013 IEEE 21st Annual Symposium on*, pages 63–70, Aug 2013.

[6] Sabela Ramos and Torsten Hoefler. Modeling communication in cache-coherent SMP systems: a case-study with Xeon Phi. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 97–108. ACM, 2013.

[7] Sreeram Potluri, Akshay Venkatesh, Devendar Bureddy, Krishna Kandalla, and Dhabaleswar K Panda. Efficient Intra-node Communication on Intel-MIC Clusters. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 128–135. IEEE, 2013.

[8] Richard L Graham and Galen Shipman. MPI support for multi-core architectures: Optimized shared memory collectives. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 130–140. Springer, 2008.

[9] M.K. Velamati, A. Kumar, N. Jayam, G. Senthilkumar, P.K. Baruah, S. Kapoor, R. Sharma, and A. Srinivasan. Optimization of collective communication in intra-Cell MPI. In *Proceedings of the 14th IEEE International Conference on High Performance Computing (HiPC)*, pages 488–499, 2007.

[10] John McCalpin. STREAM benchmark. http://www.streambench.org/, 1995.

[11] Pier Giorgio Raponi, Fabrizio Petrini, Robert Walkup, and Fabio Checconi. Characterization of the communication patterns of scientific applications on blue gene/p. *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, 0:1017–1024, 2011.

[12] Debra Hensgen, Raphael Finkel, and Udi Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988.

[13] Michael L Scott and John M Mellor-Crummey. Fast, Contention-Free Combining Tree Barriers for Shared-Memory Multiprocessors. *International Journal of Parallel Programming*, 22(4):449–481, 1994.

[14] Nian-Feng Tzeng and Angkul Kongmunvattana. Distributed shared memory systems with improved barrier synchronization and data transfer. In *Proceedings of the 11th international conference on Supercomputing*, pages 148–155. ACM, 1997.

[15] Sriram Kanchiraju, Pinak Panigrahi, and Pallav Kumar Baruah. Efficient Intra-MIC Broadcast. Poster with extended abstract presented at Student Symposium, HiPC 2013, Bangalore, India.

[16] Giray Okten and Ashok Srinivasan. Parallel quasi-monte carlo methods on a heterogeneous cluster. In Kai-Tai Fang, Harald Niederreiter, and FredJ. Hickernell, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2000*, pages 406–421. Springer Berlin Heidelberg, 2002.

[17] C. D. Sudheer, S. Krishnan, A. Srinivasan, and P. R. C. Kent. Dynamic load balancing for petascale quantum Monte Carlo applications: The alias method. *Computer Physics Communications*, 184(2):284–292, 2013.