Native Mode-Based Optimizations of Remote Memory Accesses in OpenSHMEM for Intel Xeon Phi

Naveen Namashivayam
Department of Computer Science
University of Houston
Houston, Texas
nravi@cs.uh.edu

Sayan Ghosh
Department of Computer Science
University of Houston
Houston, Texas
sgo@cs.uh.edu

Dounia Khaldi
Department of Computer Science
University of Houston
Houston, Texas
dkhaldi@uh.edu

Deepak Eachempati
Department of Computer Science
University of Houston
Houston, Texas
dreachem@cs.uh.edu

Barbara Chapman
Department of Computer Science
University of Houston
Houston, Texas
chapman@cs.uh.edu

ABSTRACT

OpenSHMEM is a PGAS library that aims to deliver high performance while retaining portability. Communication operations are a major obstacle to scalable parallel performance and are highly dependent on the target architecture. However, to date there has been no work on how to efficiently support OpenSHMEM running natively on Intel Xeon Phi, a highly-parallel, power-efficient and widely-used many-core architecture. Given the importance of communication in parallel architectures, this paper describes a novel methodology for optimizing remote-memory accesses for execution of OpenSHMEM programs on Intel Xeon Phi processors.

In *native mode*, we can exploit the Xeon Phi shared memory and convert OpenSHMEM one-sided communication calls into local load/store statements using the `shmem_ptr` routine. This approach makes it possible for the compiler to perform essential optimizations for Xeon Phi such as vectorization. To the best of our knowledge, this is the first time the impact of `shmem_ptr` is analyzed thoroughly on a many-core system. We show the benefits of this approach on the PGAS-Microbenchmarks we specifically developed for this research. Our results exhibit a decrease in latency for one-sided communication operations by up to 60% and increase in bandwidth by up to 12x. Moreover, we study different reduction algorithms and exploit local load/store to optimize data transfers in these algorithms for Xeon Phi which permits improvement of up to 22% compared to MVAPICH and up to 60% compared to Intel MPI. Apart from microbenchmarks, experimental results on NAS IS and SP benchmarks show that performance gains of up to 20x are possible.

Keywords

OpenSHMEM, Intel Xeon Phi, RMA, Reduction

1. INTRODUCTION

One of the key architectural trends in High Performance Computing (HPC) is the increasing number of processing cores on chips. The current generation of the Intel Xeon Phi, named "Knight's Corner" (KNC), functions as a many-core coprocessor on Intel Xeon-processor based systems. The next generation of Intel Xeon Phi, named "Knight's Landing" (KNL), is designed to operate as a main CPU rather than a coprocessor. Considering this technological direction, it is therefore of key importance to understand how the Phi architecture may be effectively programmed in the so-called *native mode*, where a full program is run directly on the Phi without host-to-coprocessor interaction. Porting applications to the Phi architecture is relatively straightforward compared to programming with GPGPUs since the Phi is essentially a modified x86 processor; it can natively support many of the familiar programming models used on conventional Intel processors, including MPI and OpenMP. In contrast, programming with GPGPUs requires an explicit host-to-accelerator programming paradigm.

Developing applications for Phi-based systems might benefit from new programming models. For instance, OpenSHMEM is a library interface standard which follows the Partitioned Global Address Space (PGAS) paradigm. PGAS programming models are well suited for large-scale, global address space platforms that provide non-uniform memory accesses. The PGAS approach offers better scaling properties compared to OpenMP because it supports explicit mechanisms for controlling data locality. It also provides a shared memory abstraction which makes it amenable to more aggressive code optimization compared to message-passing models such as MPI.

Communication operations are a major obstacle to scalable parallel performance and are highly dependent on the target architecture. However, to date there has been no work on how to efficiently support OpenSHMEM running natively on

Intel Xeon Phi. We describe thus in this paper techniques we have developed to exploit the memory hierarchy of Xeon Phi in order to improve performance of OpenSHMEM programs running in native mode.

Our approach has three main thrusts. First, we use the function `shmem_ptr` that returns a pointer to a so-called symmetric variable, belonging to a different Processing Element (PE). Subsequent load/store accesses to this symmetric variable can be performed directly through this pointer, enabling more opportunities for manual and compiler optimizations, including vectorization and reduction of the number of memory copies. Note that the MPI based implementation of OpenSHMEM (OSHMPI) [12] uses the shared memory windows extensions of MPI-3 and direct load/store access on the target memory within the same compute node. However, our OpenSHMEM implementation is built on the top of GASNet and offers a straightforward and efficient way of introducing load/store operations.

Second, we show the benefits of the use of `shmem_ptr` through the use of PGAS-Microbenchmarks [1], a suite of program kernels we specifically developed for this research. Beside the fact that our microbenchmarks permit to assess achievable performance for communication operations in different PGAS languages/libraries, another difference with the OSU OpenSHMEM Microbenchmarks [3] which provide latency tests for single pair PUT and GET, bandwidth tests for single pair PUT, latency and bandwidth tests for atomics is that our OpenSHMEM microbenchmarks provide latency and bandwidth tests for PUT and GET, can provide measurements for multiple communicating pairs, and also provide an option to test latency and bandwidth for load/stores using `shmem_ptr`. Our results exhibit a decrease in latency for one-sided communication operations by up to 60% and increase in bandwidth by up to 12x.

Third, we observe performance of OpenSHMEM collective operations, more specifically reduction calls (such as the routine `shmem_double_sum_to_all`), to be generally very poor when using OpenSHMEM on Intel Xeon Phi. Yet, our approach provides some improvements here, which suggests that native mode-based constructs open new perspectives for enhancing high-level operations as well.

The contributions of this paper are:

- a methodology for enhancing data transfers in OpenSHMEM programs on Xeon Phi using `shmem_ptr` instead of SHMEM *put/get* routines;
- a comparative analysis of common reduction algorithms implemented using OpenSHMEM *put/get* and `shmem_ptr` to efficiently supporting reductions on Intel Xeon Phi;
- the development of PGAS-Microbenchmarks [1], which contains a set of microbenchmarks for testing reductions, latency and bandwidth of communications for PGAS programming models such as OpenSHMEM and Coarray Fortran [16]. We also developed the SHMEM version of the well-known microbenchmark STREAM [15].
- validating our proposed optimization methodology using PGAS-Microbenchmarks, STREAM and benchmarks from the NAS Parallel Benchmarks [6]. The experimental results on IS and SP show that performance improvements of up to 20x are possible.

This paper is organized as follows. We describe the architecture of Intel Xeon Phi in Section 2. A brief overview of the implementation of OpenSHMEM is given in Section 3. Our proposed `shmem_ptr` optimization with experiments using PGAS-Microbenchmarks and application on STREAM are presented in Section 4. We study several reduction algorithms and apply `shmem_ptr` on them in Section 5. Experimental results using some of the NAS benchmarks are shown in Section 6. We survey different approaches which have targeted Intel Xeon Phi in Section 7. We discuss future work and conclude in Section 8.

2. INTEL XEON PHI ARCHITECTURE

Intel's popular Many Integrated Core (MIC) architectures are marketed under the name of Xeon Phi and they are the product of their KNC project. With the steady growth in requirements for the available number of computing cores in HPC, the popularity of Xeon Phi coprocessors among these computing-intensive applications has increased tremendously. GPUs are another example of accelerators that are able to accomplish such highly computing requirements. However, using GPUs, all the applications have to be ported to specific programming paradigms like CUDA, OpenCL, etc. Focusing on more abstract and generic approaches, we are using Xeon Phi which supports various popular programming models like MPI and OpenMP and which can also be targeted easily using OpenSHMEM library.

Xeon Phi provides x86 compatibility and runs on a Linux operating system. Intel Xeon Phi coprocessor consists of up to sixty-one (61) cores connected by a on-die bidirectional interconnect. In addition to the cores, there are 8 memory controllers supporting up to 16 GDDR5 (Graphics Double Data Rate, version 5) channels delivering up to 5.5 GT/s and special function devices such as the PCI Express system interface.

As shown in Fig. 1, each core includes a Core Ring Interface (CRI), interfacing the core and the ring interconnect. It comprises mainly the L2 cache and a distributed Tag Directory (TD) that ensures the coherence of this cache. Each core is connected to a bidirectional ring interconnect through the Core Ring Interface. The L1 and L2 cache have a data limit of 32KB and 512KB respectively. The three key aspects of Xeon Phi are its support of 512-bit vector instructions (each core includes a 512 bit-wide vector processor unit (VPU)), simultaneous multi-threading and in-order execution, which offers some ways to exploit Instruction Level Parallelism and introduces some caveats not otherwise encountered in other X86-based architectures.

For reasons mentioned before, we will only focus on the *native* mode of execution, wherein an application runs exclusively on the Xeon Phi co-processor. The native mode of Xeon Phi offers some benefits such as minimal code-porting overhead from existing architectures and not having to deal with host-to-co-processor data transfer latency (which is at least 15x slower than Xeon Phi intra-node latency). Al-

though one could achieve modest performance by porting existing CPU code on Xeon Phi, according to our observation it requires reasonable optimization effort to exploit its capabilities fully. Typically, the available memory varies, but it is within 8GB in the current architecture. As such, the number of active threads ($4 \times 61 = 244$) could saturate the memory, and being an in-order processor, this could result in excessive stalls unless memory is prefetched into the caches. Also, a Xeon Phi core could issue 1 or 2 instructions per cycle (the cores are typically clocked at 1 GHz), and only one of them could be a vector instruction such as division; prefetch instructions such as `VPREFETCH1` are not considered vector instructions. On the other hand, a thread can only issue vector instructions in every other cycle, which mandates the use of at least 2 threads/core to fully utilize a vector unit.

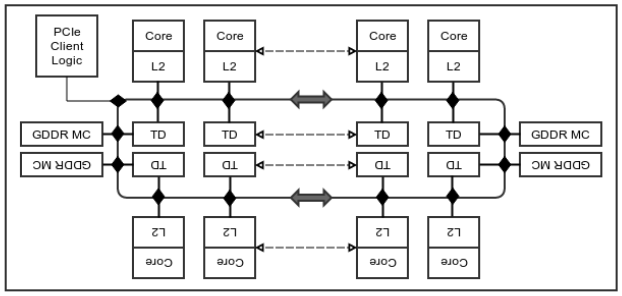

Figure 1: Xeon Phi Microarchitecture

3. OPENSHMEM OVERVIEW

OpenSHMEM [8] is a library interface specification, with bindings for C, C++, and Fortran, that unifies various specifications of the SHMEM programming API and adheres to the PGAS programming model. It is designed with a chief aim of performance, exploiting support for Remote Direct Memory Access (RDMA) available in modern network interconnects and thereby allowing for highly efficient data transfers without incurring the software overhead that comes with message passing communication.

OpenSHMEM programs follow an SPMD-like execution model, where all processing elements (PEs) are launched at the beginning of the program; each PE executes the same code and the number of processes remains unchanged during execution. OpenSHMEM PEs are initialized when the `start_pes` function is called. We present in the following paragraphs the main features of OpenSHMEM.

3.1 Remote Memory Access

OpenSHMEM supports the PGAS memory model where it proceeds by one-sided communications to transfer data between PEs. `shmem_put` is the function that copies contiguous data from a local object to a remote object on the destination PE. `shmem_get` copies contiguous data from a remote object on the destination PE to a local object.

3.2 Synchronization

We distinguish both types of synchronization: collective synchronization using for example the primitive `shmem_barrier_all`, that blocks until all other PEs issue a call to this particular statement, and point-to-point synchronization such as the `shmem_wait` primitive.

3.3 Collective Communication

One type of collective communications is reduction. It is implemented using `shmem_`*operator*`_to_all` which performs a reduction operation where *operator* can be `sum, and`, etc. on symmetric arrays over the active set of PEs. Symmetric arrays are remotely accessible data objects; an object is symmetric if it has a corresponding object with the same type, size and offset on all other PEs [5].

3.4 Mutual Exclusion

Mutual exclusion is implemented using locks of type integer. The `shmem_set_lock` function is used to test a lock and block if it is already acquired; the `shmem_set_unlock` function is used to release a lock.

4. FROM REMOTE MEMORY ACCESSES TO LOCAL LOAD/STORE OPERATIONS

This section describes our first contribution, namely how transforming remote memory accesses (`shmem_put/shmem_get`) into local load/store native operations using `shmem_ptr` can greatly improve OpenSHMEM performance. We show also the application of `shmem_ptr` on the STREAM microbenchmark.

4.1 Methodology

One of the most desirable features of the shared memory programming environment is accessing data via local load / store operations, which makes programming simpler and more efficient. This approach makes it possible for the compiler to analyze and perform essential optimizations for Xeon Phi. To enable such local load and store operations in OpenSHMEM, we suggest to employ the `shmem_ptr` builtin that returns the address of a data object on a specific PE.

On a shared memory machine, it is beneficial to use it as opposed to OpenSHMEM communication primitives such as `shmem_put` or `shmem_get`. Apart from saving function call overhead which is nominal, `shmem_ptr` could be effectively employed to enable optimizations.

Our methodology consists of: (1) computing the memory address of a symmetric variable on a given PE only once at the beginning of the program and, (2) performing the translation of communication function calls to simple assignments. A simple example that illustrates the use of this methodology is provided in Figure 2.

```
shmem_int_put(target,
       source,m,pe);
```

```
int *ptr=(int *)shmem_ptr
           (target,pe);
for (i = 0; i < m; i+=1)
  ptr[i] = source[i];
```

Figure 2: The use of `shmem_ptr`

This yields significantly better performance in terms of latency as demonstrated in Figure 3. Moreover, Figure 4 and

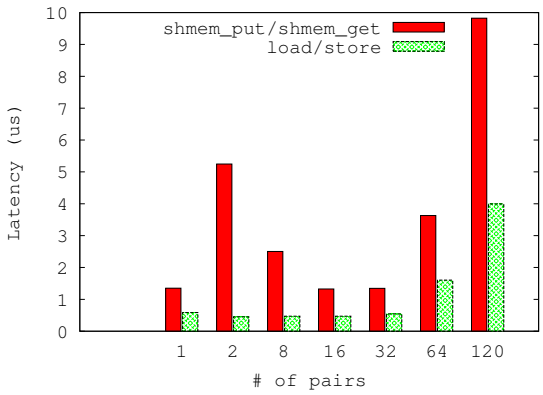

Figure 3: Latency comparison put/get vs. `shmem_ptr`

Figure 5 show the impact of this methodology on the memory bandwidth. When the message size is very large (block size $> 16K$), the difference in terms of memory bandwidth between local load/store operations and remote access functions becomes small; this is due to the increasing number of cache misses. These results are obtained running the PGAS-Microbenchmarks [1] that are well suited for evaluation performance for both point-to-point operations and reductions in various PGAS implementations. For n pairs, the number of running PEs is $n \times 2$ on an Intel Xeon Phi in native mode. Note that the OpenSHMEM reference implementation was modified to support `shmem_ptr`.

4.2 Application of `shmem_ptr` on STREAM

We conducted some basic experiments to understand the memory bandwidth variations of Intel Xeon Phi and how compiler options can be used to optimize OpenSHMEM programs. Since OpenSHMEM deals with moving data across memories, the STREAM benchmark should give us insights to tune applications written with OpenSHMEM. Indeed, STREAM [15] is a simple, synthetic benchmark that is designed to measure sustainable memory bandwidth (in MB/s) and a corresponding computation rate for four simple vector kernels (Copy, Scale, Add and Triad).

Since the critical performance factor for OpenSHMEM programs is data movement between PEs, we employed the STREAM kernels to evaluate and compare performance for these operations. As such, we have created an OpenSHMEM implementation of the STREAM benchmark. Since we want to explore Intel Xeon Phi in native mode, we have modified STREAM kernels so that each PE works with a distinct and fixed-length portion of the input arrays at a time, necessitating an all-to-all reduction after each PE is done with its current block. The `shmem_ptr` routine enables us to avoid the use of OpenSHMEM put/get calls for target PEs on the same shared memory node, and furthermore opens up opportunities for vectorization which we found to be crucial to attain good performance on Xeon Phi. The periodically occurring reduction operation is a major performance bottleneck, and we observe significantly better bandwidth results when we replace the default all-to-all reduction in OpenSHMEM reference implementation with a flat-tree reduction[1] using `shmem_ptr`. We have dedicated Section 5 to discussing these optimizations in detail.

In the OpenSHMEM version of STREAM we have developed, each PE has its own local arrays, and it is necessary to perform parallel reductions between PEs to ensure correctness. Since we are exploring Intel Xeon Phi in native mode, a reduction operation can be optimized by replacing it with local load/store accesses enabled by the usage of `shmem_ptr`. The Intel compiler offers many directives for vectorization and prefetching into cache which we have employed and with which we have noticed significant improvements on Intel Xeon Phi. A common obstacle to vectorization is data alignment; most of the time the compiler is unable to determine whether a particular pointer is aligned to specific byte boundaries which differ from one platform to another. This necessitates explicit hints to the compiler to ensure that data structures are indeed aligned so that it could proceed to vectorize loops. In Figure 6, we observe the bandwidth of STREAM Copy, Scale, Add and Triad kernels is approximately increased by 40x when we use an optimized reduction algorithm with vectorization directives (such as `#pragma vector align`), as compared to the original OpenSHMEM implementation without the use of vectorization.

5. REDUCTIONS: A USE CASE

In this section, we study different reduction algorithms, using OpenSHMEM. Our ultimate motivation is to use `shmem_ptr` to optimize data transfers in these algorithms.

5.1 Reduction Algorithms Implementations

We describe here four algorithms for parallel array reduction, and describe their performance characteristics using a performance model for the Xeon Phi architecture. We chose reduction as a representative collective operation because of its wide usage in many applications, and the fact that there is some computation associated with it (not only communication as in scatter/gather). It is possible for an architecture like Intel Xeon Phi to utilize vector pipelines to overlap computation and communication.

In the following, we describe each of these algorithms and our reasoning for why we considered them for Xeon Phi. We have looked at both power-of-2 cases and non-power-of-2 cases for the number of PEs in our evaluations. There has been significant amount of study on improving collective performance of MPI, as we describe in Section 7.

We use a simple cost model to estimate the total time taken by these algorithms based on three parameters: latency α, bandwidth β, and local computation cost per byte γ. We denote n the size of arrays and p the number of processes. This cost model assumes that all processes can send and receive one message at the same time.

5.1.1 Flat Tree (FT)

The flat tree or linear algorithm uses a single root process that communicates with the remaining processes for carrying out the all-reduce operation, and it is considered to work well for arrays of short to medium size. This algorithm is carried out in two stages. In the first stage, the root process

[1]The root process gathers from all other processes, performs reduction and scatters to all the processes.

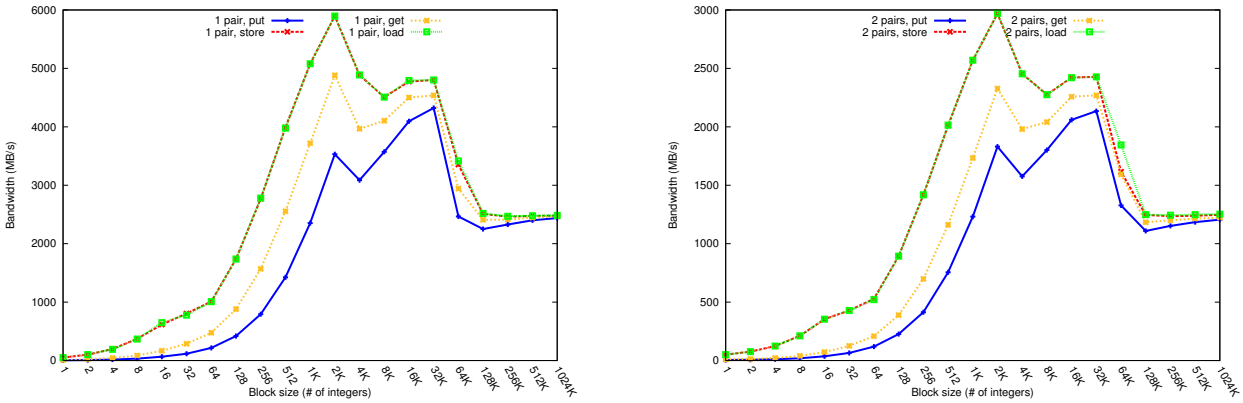

Figure 4: Bandwidth comparison of communications between 1 and 2 pairs of PEs: store/load using shmem_ptr vs calls to shmem_put/shmem_get; note that green and red lines overlap

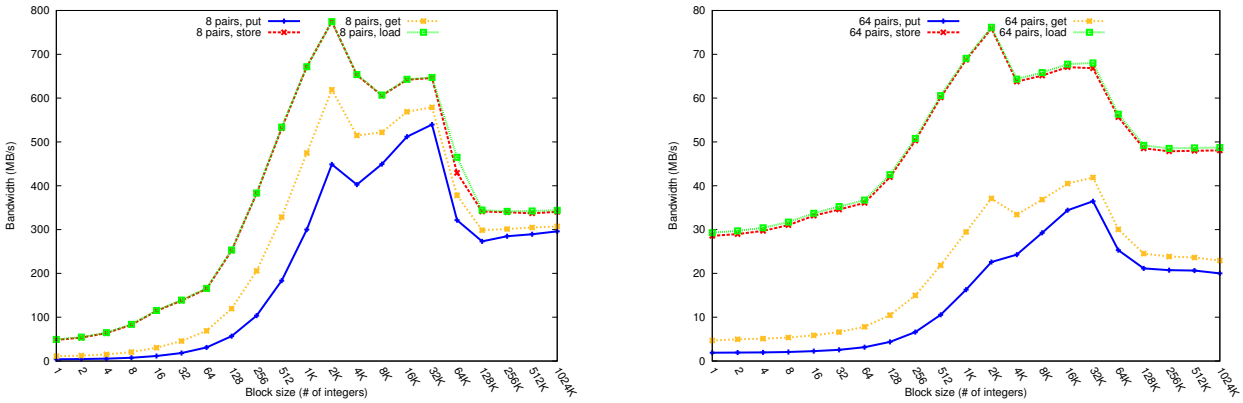

Figure 5: Bandwidth comparison of communications between 8 and 64 pairs of PEs: store/load using shmem_ptr vs calls to shmem_put/shmem_get; note that green and red lines overlap

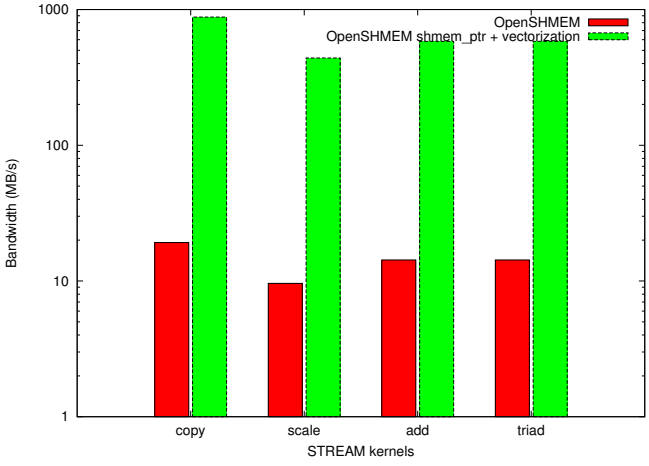

Figure 6: OpenSHMEM STREAM bandwidth (MB/s) using original OpenSHMEM and optimized OpenSHMEM (`shmem_ptr` + vectorization) reduction on 2 PEs of Intel Xeon Phi

will gather the data from each of the other processes and apply the reduction operation on them. In the second stage, the results are broadcasted back to the other processes. Different strategies may be employed for gathering the data and then performing the subsequent broadcast, depending on factors such as array size and total number of processes involved in the reduction. We consider here the simplest strategy, where the root process will sequentially read the source buffers from each of the other processes and compute the reduced result, and then write the result back to each of the other processes sequentially.

The time taken in this algorithm is:

$$T_{FT} \quad = \quad p\alpha + 2n(p-1)\beta + n(p-1)\ \gamma.$$

5.1.2 *Recursive Doubling (RD)*

The recursive (distance) doubling algorithm [21] is a technique that can be applied for a variety of collective algorithms, including all-gather, barrier, and all-reduce. For cases where the number of processes performing the reduction is a power of 2, the algorithm completes in $\lg p$ steps. On the first step ($s = 0$), processes with even rank i will exchange their initial array values with the process of rank $i + 1$, and then all processes will perform the reduction operation using the received data. On each subsequent step s, each process i will exchange its updated results with either process $i + 2^s$ if $\mathrm{mod}(i, 2^{s+1}) < 2^s$ or with process $i - 2^s$ if $\mathrm{mod}(i, 2^{s+1}) \geq 2^s$, and then again perform the reduction operation with the received data.

When the number of processes is not a power of 2, each of the first r even ranked processes with rank i (where r is $p - 2^{\lfloor \lg p \rfloor}$) will send its data to the process with rank $i + 1$. These receiving processes will perform the reduction operation using the received data. We then have a set of $2^{\lfloor \lg p \rfloor}$ processes to carry out the core reduction algorithm that requires a power of 2 number of participants, namely the first r processes with odd rank and then the last $2^{\lfloor \lg p \rfloor} - r$ processes. After the core algorithm is completed among these processes, they will each contain the final reduction result. We complete the algorithm by having the first r odd ranked processes with rank i send their data to processes with rank $i - 1$.

The time taken in this algorithm is:

$$T_{RD} \quad = \quad \lg p \; \alpha + n \lg p \; \beta + n \lg p \; \gamma.$$

In order to optimize this algorithm for native mode execution on Xeon Phi, we used the `shmem_ptr` routine which allows each PE to obtain a direct access to another PE's symmetric data. We also experimented with a "left computes" strategy, where for each pair of communicating processes the one with smaller rank (1) updates its local array by applying the reduction operation and using a direct reference to the array on its partner, (2) writes the resulting values back to the partner's array, and (3) notifies the partner that its array has been updated. This strategy allowed us to avoid the use of an additional receive buffer and `memcpy` operations for the data exchanges, and improve performance significantly for large array sizes.

5.1.3 Bruck (BR)

We have developed a variant of the Bruck all-gather algorithm [7] for reductions, which we will refer to here for simplicity as Bruck. As in the recursive doubling algorithm described above, the core algorithm described here works for a power of 2 number of processes, and we deal with the case where it is not a power of 2 as described above. For Bruck, on each step s a process with rank i will send its data to the process with rank $\mathrm{mod}(2^{\lfloor \lg p \rfloor} + i - 2^s, 2^{\lfloor \lg p \rfloor})$. Upon receipt of this data, the process will perform a reduction using it. After $\lg p$ steps where p is a power of 2, all p processes performing the reduction will contain the final result.

The time taken in this algorithm is:

$$T_{BR} \quad = \quad \lg p \; \alpha + n \lg p \; \beta + n \lg p \; \gamma.$$

5.1.4 Rabenseifner (RSAG)

We also implemented Rabenseifner's algorithms [21] in order to more efficiently carry out reductions on very large array sizes. As with recursive doubling, the core algorithm works for a power of 2 number of processes, and when the number of processes is not a power of 2 it can be handled as described in 5.1.2. The algorithm can be implemented in two phases. In the first phase, the PEs perform a Reduce-Scatter (RS) operation, using a distance doubling and vector halving procedure which completes in a $\lfloor \lg p \rfloor$ steps. In the second phase, the PEs perform an All-Gather (AG) operation, using a distance halving and vector doubling procedure which can again complete in $\lfloor \lg p \rfloor$ steps. While there is a greater number of messages being communicated in this algorithm among the PEs over the two phases, because on each step only a portion of the full array is being exchanged between the PEs this turns out to be more efficient for reductions on large array sizes.

The time taken in this algorithm is:

$$T_{RSAG} \quad = \quad 2 \lg p \; \alpha + 2 \tfrac{p-1}{p} n \; \beta + \tfrac{p-1}{p} n \; \gamma.$$

For native-mode execution on Xeon Phi, we applied the following optimizations to improve the performance. We first combined the last step of the reduce-scatter phase with the first step of the all-gather phase. In effect, the processes that are paired with each other in the last step of reduce-scatter can exchange corresponding portions of their array data and perform the reduction operation using the received data. Thus, we can think of this algorithm as now divided into 3 stages: (1) the first $\lfloor \lg p \rfloor - 1$ steps of reduce-scatter, (2) an exchange and reduction of corresponding array blocks between PEs which are a distance $2^{\lfloor \lg p \rfloor}/2$ apart, and (3) the final $\lfloor \lg p \rfloor - 1$ steps of all-gather. We also use `shmem_ptr` to avoid having to introduce an additional work buffer and perform `memcpy` operations. Instead, each PE can read directly from its partner's array to update its own array using the reduction operation. Finally, we also applied the "left computes" optimization described in 5.1.2 for the new middle stage of our algorithm.

5.1.5 Discussion

We implemented the aforementioned reduction algorithms in OpenSHMEM without using our `shmem_ptr` optimization (see next subsection), and tested the performance when running in native mode on Intel Xeon Phi using the reduction microbenchmarks in our PGAS-Microbenchmark suite. In Fig. 7, we found that the flat tree algorithm performed the worst, as predicted from our cost model. Moreover, we ob-

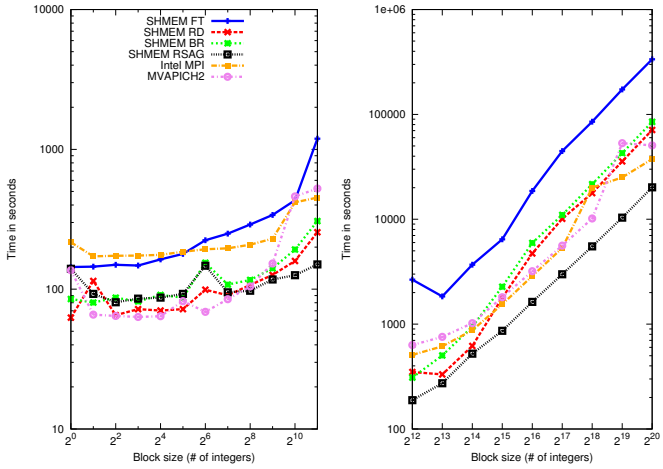

Figure 7: Performance of different reduction algorithms implemented in OpenSHMEM vs. MPI using 64 PEs on Intel Xeon Phi (1 to 2^{10} and 2^{12} to 2^{20} number of integers)

serve significant improvement when using recursive doubling for small message sizes (up to 4KB). Rabenseifner's algorithm performs the best for large message sizes compared

to the other existing algorithms. This is due to a lessened data transfer cost resulting from less congestion on the ring network of the Phi. However, for small message sizes, MVAPICH2 performs the best; this leads to the idea of using `shmem_ptr` in our SHMEM implementation of reduction algorithms, which we will be exploring in the next section.

5.2 Application of `shmem_ptr` on Reduction Algorithms

The use of `shmem_ptr` serves a dual purpose for running OpenSHMEM codes more efficiently on Intel Xeon Phi. It allows us to eliminate the use of some intermediate buffers, which is important because there is a limited amount of main memory of typically less than 8 GB. Furthermore, it exposes more instruction-level parallelism to the compiler, enabling vectorization opportunities which can take advantage of the Phi's 512-bit-wide vector units. Therefore, we have applied our methodology of using `shmem_ptr` in our implementation of the reduction algorithms presented in Section 5.1.

We ran the optimized version of these algorithms on Intel Xeon Phi using 64 PEs in native mode. The experimental results are shown for small message sizes in Fig. 8 and for large message sizes in Fig. 9. We observe that the use of `shmem_ptr` improves the execution time of these reduction operations. Especially, in Fig. 10, the optimized versions of the recursive doubling and RSAG algorithms perform better than both Intel MPI's and MVAPICH2's default implementations for small and large message sizes respectively.

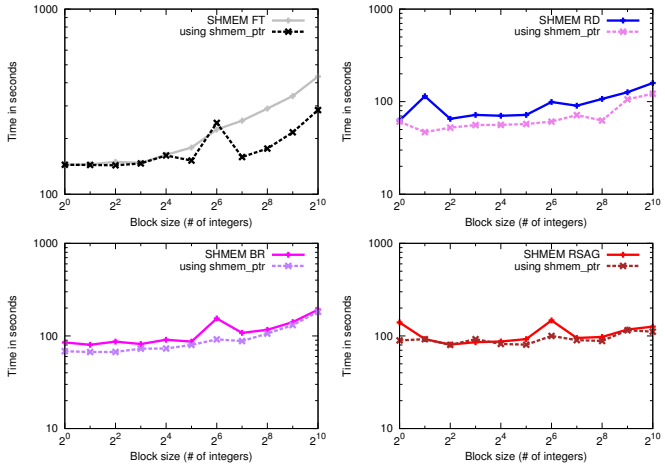

Figure 8: SHMEM reduction algorithm performance using put/get vs. `shmem_ptr` using 64 PEs for small data sizes

6. EXPERIMENTAL RESULTS USING IS AND SP NAS PARALLEL BENCHMARKS ON INTEL XEON PHI

In this section, we present experimental results for our OpenSHMEM code optimizations using the NAS parallel benchmarks IS and SP on Intel Xeon Phi. Our goal here is to assess whether the two OpenSHMEM optimization techniques introduced above to promote Xeon Phi native mode (`shmem_ptr`, reduction) already carry over to more general benchmarks.

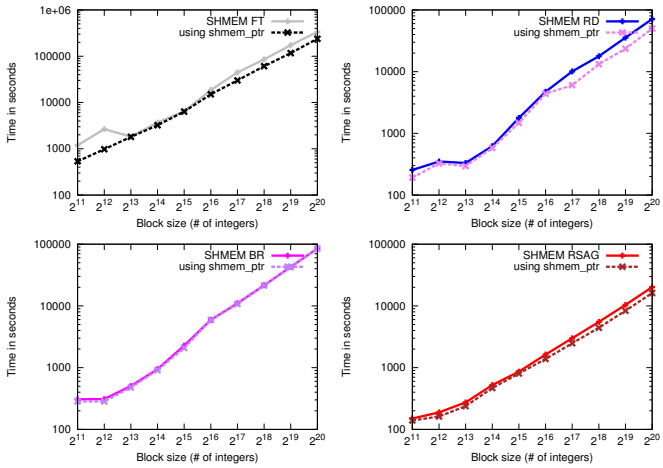

Figure 9: SHMEM reduction algorithm performance using put/get vs. `shmem_ptr` using 64 PEs for large data sizes

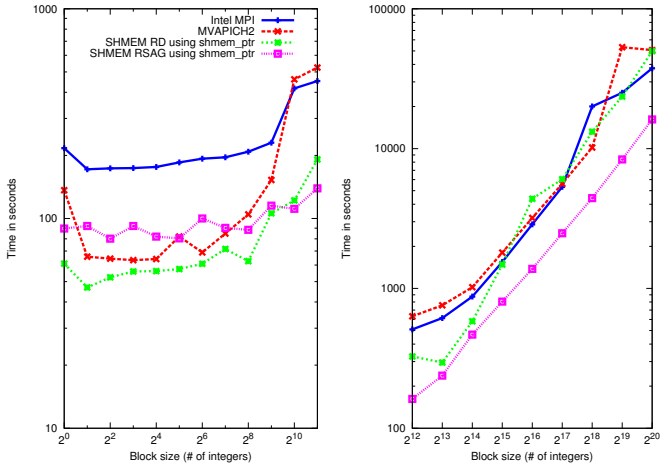

Figure 10: Performance of different reduction algorithms using `shmem_ptr` vs. Intel MPI and MVAPICH using 64 PEs on Intel Xeon Phi

6.1 Experimental Setup

We used the Stampede supercomputing system at Texas Advanced Computing Center (TACC) [4] for all the experiments presented in this paper. Each Xeon Phi node (61 Xeon Phi SE10P) on Stampede has 61 cores running at 1.1GHz. Each has 8GB GDDR5 of global memory, 32KB of L1 cache and 512 KB of L2 cache. The Phi processors run a simplified Linux-based OS on one of the 61 cores. We used the icc 14.1 compiler and, for comparison, Intel MPI 4.1.0. We used and extended OpenSHMEM v1.0f with SMP conduit of GASNet. All the codes used in our experiments were compiled at the *-O3* optimization level.

We show here the impact of modifying some of OpenSHMEM NAS benchmarks [2] to support `shmem_ptr` and compare the result with Intel MPI. We then further optimize our OpenSHMEM version by using the best performing reduction algorithms, recursive doubling for small message sizes and Rabenseifner for large message size, and compare this with the MPI version of IS using Intel MPI. We chose the benchmarks depending on the importance of reductions or

communications within them in order to better assess the benefits of our optimizations on real benchmarks. Table 1 summarizes the properties of the NAS benchmarks (we run them using Class C); we ran these results using Scalasca profiling tool [11] on the *Crill* cluster at the University of Houston which consists of 16 nodes with four 12-core AMD Opteron processors per node. We used only one 48-core node to simulate a Xeon Phi. Based on the data shown in this table, we chose IS and SP to evaluate the `shmem_ptr` optimizations, and IS to evaluate the reduction algorithm optimizations.

Benchmark	Reductions %	Remote accesses %	SLOC	OpenSHMEM version available in [2]
MG	0.1	19.6	1639	√
BT	0	15.5	2436	√
EP	1.6	0	190	√
SP	0	44.1	2166	√
IS	12.4	11.7	628	√
CG	0	33.2	935	×
FT	0.8	31.2	1319	×
DT	0	10	869	×
LU	0.1	14.8	3509	×

Table 1: NAS Benchmarks

6.2 Experimental Results Using IS and SP NAS Benchmarks

Integer Sort (IS) and Scalar Pentadiagonal (SP) are two of the eleven benchmarks in the NAS Parallel Benchmarks suite. In this section, we showcase the `shmem_ptr` optimization on IS and SP.

6.2.1 *Application of* `shmem_ptr` *on IS*

Figures 11, 12 and 13 show the results for IS comparing the original reference implementation of OpenSHMEM, Intel MPI, and our optimized OpenSHMEM on Intel Xeon Phi. We evaluated for Classes A, B and C. We found replacing remote memory accesses by local load and store operations is beneficial when the number of PEs is greater than 32 in this setup. This is consistent with the commonly held notion that performance using Intel Xeon Phi increases slowly and is obtained using only a large number of cores. We note an average improvement of up to 13% using `shmem_ptr`. Moreover, these figures demonstrate that OpenSHMEM is more scalable than Intel MPI.

Figure 14 shows the impact of using optimized versions of the recursive doubling and Rabenseifner reduction algorithms introduced in Section 5. We compare them with the default all-to-all reduction algorithm of OpenSHMEM. We notice an average improvement of 25% compared to the default implementation. It could be observed that reduction time drops significantly from 64 to 128 PEs for both RD and RSAG, because the communication cost is reduced due to the usage of `shmem_ptr`. On a large number of PEs, compiler optimizations such as vectorization, and elimination of excessive put/get calls are key factors to improve performance for Xeon Phi.

6.2.2 *Application of* `shmem_ptr` *on SP*

Figures 15, 16 and 17 show the results for SP comparing the original reference implementation of OpenSHMEM, Intel MPI, and our optimized OpenSHMEM on Intel Xeon Phi. We evaluated for Classes A, B and C. Replacing remote memory accesses by local load/store operations gives good results, but the benefit is inversely proportional to the message size; the average improvement is up to 17%. Here, both optimized OpenSHMEM and Intel MPI are scalable.

7. RELATED WORK

In this section, we survey different existing implementations of reductions and work that have targeted Intel Xeon Phi using different programming languages, and compare them with our approach.

7.1 Reduction

New algorithms for collective operations in MPICH are presented in [20]. MPICH uses a recursive doubling algorithm if operations are not commutative and recursive halving for commutative operations. Also, it switches between different algorithms depending on the message size; for example, for long messages ($\geq$ 512KB), it uses a pair-wise exchange algorithm. In our paper, we study the behavior of different reduction algorithms on Xeon Phi specifically, using OpenSHMEM. Moreover, we optimize these algorithms for shared memory using local load and store instructions instead of OpenSHMEM library calls which permits improvement of up to 22% compared to MVAPICH for small message sizes and up to 60% compared to Intel MPI for large message sizes.

A new implementation of collectives using 1-sided communication operations is proposed in [22], where two-sided message-passing operations are replaced by remote memory accesses. Collective communication operations such as broadcast are implemented using shared memory buffers and flags. In our paper, we use a similar approach to implement all-reduce operations for shared memory, but to the best of our knowledge this is the first work that targets the native mode of Xeon Phi with efficient optimizations for one-sided communication and reduction operations using the OpenSHMEM library.

Li et al. [13] investigated optimizations for MPI collectives over clusters of NUMA nodes. They developed performance models for collective communication using shared memory and experimentally validated these models with various implemented collective algorithms. They presented three algorithms: reduce-broadcast, dissemination, and tiled reduce-broadcast. The reduce-broadcast algorithm uses a standard binary tree approach for reduction and then broadcast, but such an approach can suffer from load imbalance. Their dissemination algorithm completes the reduction in $\lg n$ steps, and does so by keeping all threads busy. But this algorithm may suffer from congestion. The tiled reduce-broadcast has all threads performing a reduction and broadcast on a section of the vector being operated on. This is used for intra-socket reduction, with a tree reduce-broadcast used for inter-socket reduction. In our paper, we proposed an optimized version of four reduction algorithms; moreover, we converted one-sided communication calls into simple load/store operations inside the Intel Xeon Phi.

7.2 Programming for Xeon Phi

Cramer et al. in [10] presented the overheads and memory constraints involved in porting various OpenMP applications onto Xeon Phi under the native as well as symmetric

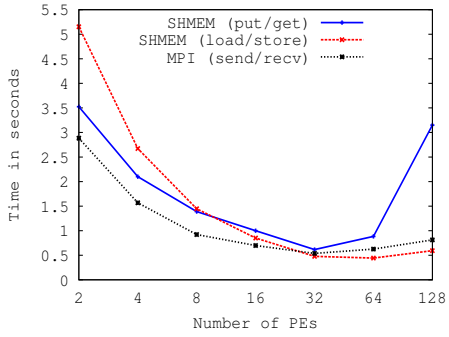

Figure 11: IS CLASS A execution time comparison: store/load using `shmem_ptr` vs calls to `shmem_put/shmem_get`

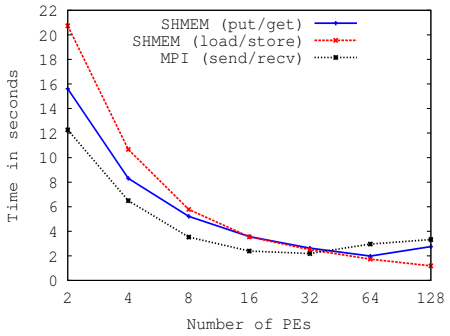

Figure 12: IS CLASS B execution time comparison: store/load using `shmem_ptr` vs calls to `shmem_put/shmem_get`

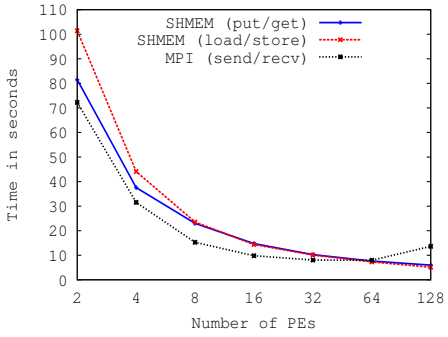

Figure 13: IS CLASS C execution time comparison: store/load using `shmem_ptr` vs calls to `shmem_put/shmem_get`

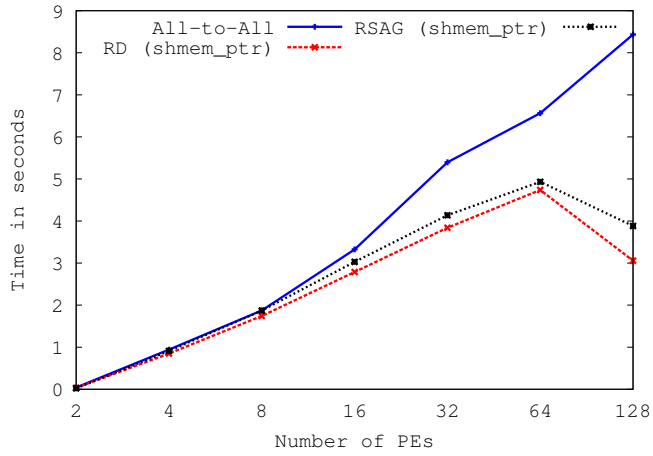

Figure 14: IS CLASS C reduction execution time comparison: RD and RSAG vs. default all-to-all algorithm

modes. Luo et al. [14] have implemented UPC applications on Xeon Phi in both symmetric and native modes. They have proposed a new thread-based model for data communication from Host-to-MIC called the "leader-to-all". Based on this proposed model, they discussed the optimal strategy that can be used for intra-MIC, MIC-to-Host, and Host-to-MIC separately. Also, the MVAPICH-PRISM [17] framework minimizes the internode communication overheads. However, in these three works, no specific optimizations for the native mode were proposed. In our approach, we explore code optimizations for OpenSHMEM programs running under this mode.

An efficient implementation of intra- and inter-node point-to-point communications in MVAPICH is proposed for Xeon Phi in [19]. In our paper, however, we show how one can remove completely intra-node communication calls and replace them by simple load/store operations based on the address of the remote data, consequentially enabling more effective compiler optimizations.

A performance model for cache-coherent SMP systems is developed in [18]. Xeon Phi is used to showcase the applicability of this model. In our paper, we exploited the performance model developed in this work; for example, the fact that the distance between cores is irrelevant to the communication cost informed our decision to not consider this factor when optimizing our reduction algorithms for Xeon Phi. The results we get in this paper match with the performance model in the referred work.

8. CONCLUSION

The peak performance of Intel Xeon Phi doubles the theoretical peak achievable on Intel Sandy Bridge. However, in order to realize this benefit, it is critical to carefully consider the potential parallelization overheads and exploit vectorization as much as possible. Moreover, considering optimizations for programs executing in native mode on Xeon Phi is of particular interest. Indeed, this can provide insights on effective optimization strategies for the next generation of Intel's Xeon Phi processors, named "Knight's Landing", which is designed to operate as a main CPU rather than a coprocessor. This paper exploits two levels of optimizations that are directly related to the communication wall problem which can inhibit scalable performance on Xeon Phi. Firstly, replacing function calls of remote memory accesses by local load and store memory operations makes it possible for the compiler, besides simplifying memory accesses, to optimize the code. In particular, it creates more opportunities for the compiler to automatically vectorize the code, which is crucial when targeting the Xeon Phi. And secondly, developing efficient reduction algorithms and integrating local load/store within them significantly enhance the performance of OpenSHMEM programs.

We plan, as future work, to automate the methodology we describe in this paper, translating communication calls into load/store instructions, in the OpenUH [9] compiler to optimize OpenSHMEM programs for Xeon Phi and shared memory platforms more generally.

Acknowledgments

The authors would like to thank Pierre Jouvelot, Tony Curtis, Pengfei Hao whose comments improved this work. Special thanks also to Zhifeng Yun who provided us his advice whenever needed. The authors acknowledge the Texas Advanced Computing Center (TACC) at the University of

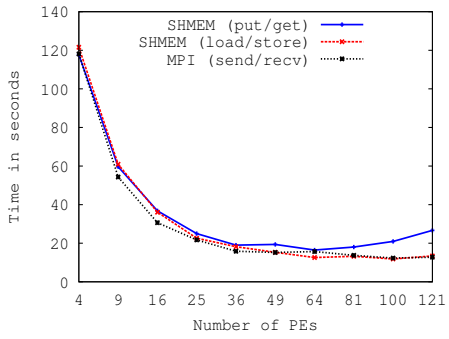

Figure 15: SP CLASS A execution time comparison: store/load using `shmem_ptr` vs calls to `shmem_put/shmem_get`

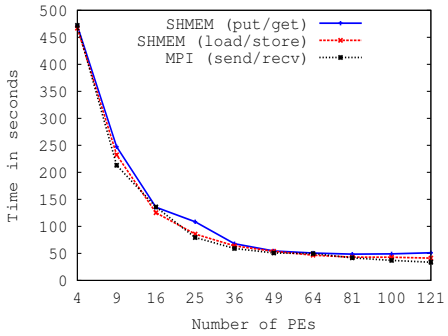

Figure 16: SP CLASS B execution time comparison: store/load using `shmem_ptr` vs calls to `shmem_put/shmem_get`

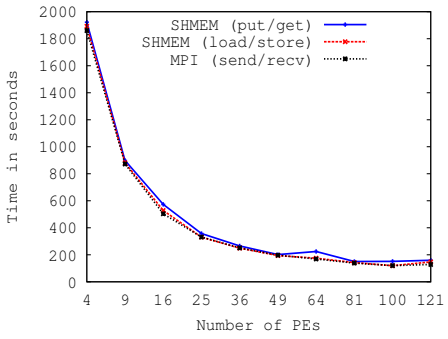

Figure 17: SP CLASS C execution time comparison: store/load using `shmem_ptr` vs calls to `shmem_put/shmem_get`

Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper.

Development at the University of Houston was supported in part by the National Science Foundation's Computer Systems Research program under Award No. CRI-0958464. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

This work was supported in part by the United States Department of Defense and used resources of the Extreme Scale Systems Center at Oak Ridge National Laboratory.

9. REFERENCES

[1] HPCTools PGAS-Microbenchmarks. `https://github.com/uhhpctools/pgas-microbench`.

[2] OpenSHMEM NAS Parallel Benchmarks, Version 1.0a. `http://www.openshmem.org/site/Downloads/Examples`.

[3] OSU Micro-benchmarks. `http://mvapich.cse.ohio-state.edu/benchmarks/`.

[4] STAMPEDE - Dell PowerEdge C8220 Cluster with Intel Xeon Phi coprocessors. `http://www.tacc.utexas.edu/resources/hpc`.

[5] OpenSHMEM Application Programming Interface (version 1.0). `http://upc.gwu.edu/documentation.html`, 2012.

[6] David H Bailey. *NAS Parallel Benchmarks*. Springer, 2011.

[7] J. Bruck, Ching-Tien Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems. *Parallel and Distributed Systems, IEEE Transactions on*, 8(11):1143–1156, Nov 1997.

[8] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing OpenSHMEM: SHMEM for the PGAS Community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 2:1–2:3, New York, NY, USA, 2010. ACM.

[9] Barbara Chapman, Deepak Eachempati, and Oscar Hernandez. Experiences Developing the OpenUH Compiler and Runtime Infrastructure. *Int. J. Parallel Program.*, 41(6):825–854, December 2013.

[10] T. Cramer, D. Schmidl, M. Klemm, and D. an Mey. OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison. In *Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University*, pages 38–44, November 2012.

[11] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca Performance Toolset Architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[12] Jeff R Hammond, Sayan Ghosh, and Barbara M Chapman. Implementing OpenSHMEM Using MPI-3 One-Sided Communication. In *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*, pages 44–58. Springer, 2014.

[13] S. Li, T. Hoefler, and M. Snir. NUMA-Aware Shared Memory Collective Communication for MPI. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 85–96. ACM, Jun. 2013.

[14] Miao Luo, Mingzhe Li, Akshay Venkatesh, Xiaoyi Lu, and Dhabaleswar K. Panda. UPC on MIC: Early Experiences with Native and Symmetric Modes. In *PGAS '13: Proceedings of the Seventh Conference on Partitioned Global Address Space Programing Models*. ACM, 2013.

[15] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. Technical report, University of Virginia, [Online]. Available: http://www.cs.virginia.edu/stream/, 1991-2007.

[16] Robert W. Numrich and John Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17:1–31, Aug 1998.

[17] Sreeram Potluri, Devendar Bureddy, Khaled Hamidouche, Akshay Venkatesh, Krishna Kandalla, Hari Subramoni, and Dhabaleswar K. Panda. MVAPICH-PRISM: A Proxy-based Communication Framework Using InfiniBand and SCIF for Intel MIC Clusters. In *Proceedings of the International Conference on High Performance Computing,*

Networking, Storage and Analysis, SC '13, pages 54:1–54:11, New York, NY, USA, 2013. ACM.

[18] S. Ramos and T. 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, 06 2013.

[19] D. Bureddy S. Potluri, K. Hamidouche and MVAPICH2-MIC D. K. Panda. MVAPICH2-MIC: A High-Performance MPI Library for Xeon Phi Clusters with InfiniBand. In *Extreme Scaling Workshop*, August 2013.

[20] Rajeev Thakur. Improving the Performance of Collective Operations in MPICH. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. Number 2840 in LNCS, Springer Verlag (2003) 257-267 10th European PVM/MPI User's Group Meeting*, pages 257–267. Springer Verlag, 2003.

[21] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[22] V. Tipparaju, Jarek Nieplocha, and Dhabaleswar K. Panda. Fast Collective Operations Using Shared and Remote Memory Access Protocols on Clusters. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 10 pp.–, April 2003.