Accelerating Applications at Scale Using One-Sided Communication

Hongzhang Shan, Brian Austin, Nicholas J. Wright, Erich Strohmaier,
John Shalf and Katherine Yelick
CRD and NERSC
Lawrence Berkeley National Laboratory, Berkeley, CA 94720
hshan, baustin, njwright, estrohmaier, jshalf, kayelick@lbl.gov

***Abstract*—The lower overhead associated with one-sided messaging as compared to traditional two-sided communication has the potential to increase the performance at scale of scientific applications by increasing the effective network bandwidth and reducing synchronization overheads. In this work, we investigate both the programming effort required to modify existing MPI applications to exploit one-sided messaging using PGAS languages, and the resulting performance implications. In particular, we modify the MILC and IMPACT-T applications to use the one-sided communication features of Unified Parallel C (UPC) and coarray Fortran (CAF) languages respectively. Only modest modifications to the source code are required where fewer than 100 lines of the source code out of 70,000 need to be changed for each application. We show that performance gains of more than 50% can be achieved at scale, with the largest benefits realized at the highest concurrencies (32,768 cores).**

I. Introduction

With the advent of petascale computing platforms, the number of cores inside a compute node and the number of nodes across the whole system have increased dramatically, and this trend, of greater and greater parallelism, seems likely to continue onto foreseeable exascale systems. The legacy MPI programming model has started to face challenges to efficiently exploit such massive and hierarchical parallelism [10], [30], [9]. These challenges include a reduced amount of memory per core, reduced memory and network bandwidth per core, a lack of mechanisms to express parallelism inside a private address space, and the inefficiency of using two-sided messages to handle large amounts of fine-grain communication. To address these challenges, researchers are starting to investigate other programming models [15]. Among these future programming models, the Partitioned Global Address Space (PGAS) family of languages, in this work represented by CoArray Fortran (CAF) [24] and Unified Parallel C (UPC) [3], have attracted great attention.

A big difference between MPI and PGAS languages is that PGAS languages provide a global shared address space which enables a thread to name and access distributed data structures directly. Remote data can be accessed either implicitly (through assignment statements that will eventually be translated into one-sided messages by the compiler) or explicit one-sided message functions. One-sided communications are also supported by MPI-2 [20], although we do not explore their use in this paper. The principle reason is that performance measurements on our experimental platform show that MPI-2 one-sided performs significantly worse than PGAS and in fact worse than the MPI two-sided.

CAF and UPC have both been around for a decade or so. Neither of them however has been widely adopted by the user community; partly because of the lack of a developer environment, and partly because of the lack of convincing performance results for real applications, especially at large-scale, that demonstrate why they are viable (or superior) alternatives to the MPI programming model.

In our earlier work [28], we investigated the programming and performance differences between PGAS languages (UPC and CAF) and MPI using micro benchmarks and the FFT benchmark from the NAS suite on a Cray XE6 platform. In this work we study real, large-scale scientific applications. We have selected two, MILC [18] and IMPACT-T [26], which are two important consumers of computing cycles on NERSC (and other) platforms. IMPACT-T simulates the dynamics of a beam of charged particles within a particle accelerator and MILC is developed to study quantum chromodynamics within the framework of lattice gauge theory to understand the strong dynamics of quarks and gluons. Both applications were originally developed in the MPI programming model.

Instead of developing PGAS implementations from scratch, we incrementally replace the MPI communication with one-sided communication features of UPC or CAF. The purpose is to minimize programming effort and to demonstrate the benefits of one-sided messaging without commingling the results with other scalar code optimizations or changes to the algorithm.

To motivate one-sided messaging, we also show messaging rate results collected for both MPI and CAF on the Cray XE6 platform. Similar results have been reported earlier [28]. However, the previous results were collected under a uni-directional communication pattern, which does not reflect with the scientific applications we have chosen. Therefore, we changed the micro benchmark code accordingly and measured the performance using bi-directional communication patterns. Due to the anticipated reduction in memory per core on future exascale platforms, messaging rate is likely to become an increasingly important performance metric, especially for

small messages.

The principle contributions of this paper are

- We quantify the potential performance benefits of one-sided communication as compared to MPI for a bidirectional communication pattern on a Cray XE6 platform and examine the dependence of this on message size.
- We show how to minimally modify two existing MPI-based scientific applications, MILC and IMPACT-T , in order to exploit the performance benefits of the one-sided communication primitives of UPC and CAF languages respectively.
- We show that the naively modified version of the MILC code, which uses global barrier synchronization in order to ensure data consistency, has performance worse than the original MPI version. By using point-to-point synchronization instead we improve the performance significantly, with the largest performance benefits demonstrated at the largest concurrencies. Our results show that at 32K cores the one-sided UPC version of the code is 1.5 times faster than the original MPI version.
- Our analysis indicates that the faster performance of the one-sided communication versions of the MILC code using UPC is due to the greater effective bandwidth it achieves at the message sizes of interest. The lower overhead of one-sided messaging achieves a greater messaging rate than is achievable using a 2-sided messaging protocol.
- We show that by modifying the dominant communication kernel in the IMPACT-T code to use CAF we are able to make the performance of the entire code 1.18 times faster than the MPI one at 16K cores, with the performance of the kernel itself 1.5 times faster.

The rest of the paper is organized as follows. Section II discusses the approach we used to modify IMPACT-T and MILC to exploit the one-sided communication features of PGAS languages. Section III describes the Cray XE6 experimental platform and the Gemini interconnect. Our messaging rate micro benchmark is described in Section IV. In Section V, we describe in detail the code changes to modify MILC to use UPC one-sided communication and compare the performance and scalability between the UPC and MPI implementations. Similarly in Section VI, we discuss how IMPACT-T has been modified to use CAF and the corresponding performance changes. Related work is discussed in Section VII. Finally we summarize our conclusions and future work in Section VIII.

II. Approach

In this section we describe the general approach we took to modify each of the applications to exploit the one-sided communication features of UPC and CAF. We use an incremental approach to modify IMPACT-T and MILC, taking advantage of the interoperability of MPI with PGAS languages on the Cray XE6. The following are the main steps we take to modify MPI applications to use the one-sided communication features of UPC and CAF:

1) Identify the communication performance critical piece of the application and the corresponding communication related variables and change their definition from local in MPI to global definitions (coarray in CAF and shared in UPC). For CAF, the coarray size needs to be the same across all images, whether they are dynamically allocated or statically defined. For UPC, this is not necessary. However, we need to define the data locality for the shared variables. A block size can be used to control how the shared data will be distributed cyclically across UPC threads.
2) Allocate memory space for dynamically allocated data. In CAF, the `allocate` function can be used to allocate space for coarrays. In UPC, different functions may be called depending on the desired data affinity.
3) Replace the MPI sending/receiving functions with data assignment statements which will be translated into one-sided put or get operations by the compiler. The granularity of the assignment statement can be used to control the message sizes. UPC also provides corresponding one-sided put and get functions. In UPC, non blocking put and get functions are provided on by the Cray compiler although they are not in the UPC standard. In CAF, the compiler directive pgas defer_sync can be used to delay the synchronization of the PGAS data references until the next synchronization instruction.
4) Add synchronization when necessary to maintain data consistency and guarantee data correctness. This is usually needed before get or after put operations. For example, a thread has to make sure that the correct data has been placed in the source buffer by another thread before it initiates a get operation.

This incremental approach enables us to modify large application codes to exploit one-sided communication with a minimum of programming effort. Such an approach is essential for a new programming model to succeed, because a huge majority of existing scientific applications running on high-performance computing platforms today are developed in message passing style, whether directly or not [9]. A more radical approach is to rewrite the full applications to exploit a fine-grained communication pattern. However, this approach requires substantial programming effort and has not been pursued here as this work is focused upon investigating the performance advantages of one-sided communication.

III. Experimental Platform

Our work was performed on a Cray XE6 platform, called Hopper, which is located at NERSC and consists of 6,384 dual-socket nodes each with 32GB of memory. Each socket within a node contains an AMD "Magny-Cours" processor with 12 cores running at 2.1 GHz. Each Magny-Cours package is itself a MCM (Multi-Chip Module) containing two hex-core dies connected via hyper-transport. (See Fig. 1.) Each die has its own memory controller that is connected to two 4-GB DIMMS. This means each node can effectively be viewed as having four chips and there are large potential performance

penalties for crossing the NUMA domains. Every pair of nodes is connected via hypertransport to a Cray Gemini network chip. Hopper's Gemini chips collectively form a 17x8x24 3-D torus. In this work we used the Cray compiler version 8.0.2 which supports both Coarray Fortran and Unified Parallel C.

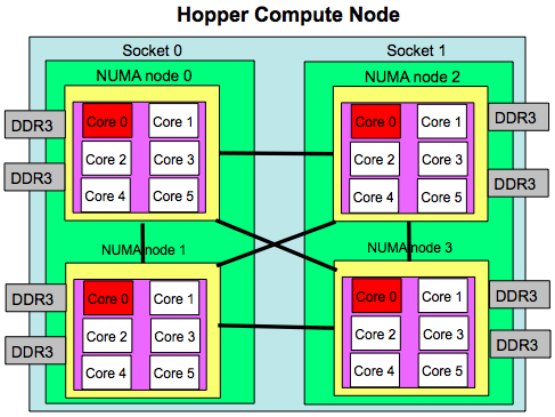

Fig. 1. The node architecture of Hopper.

A. Gemini

The defining feature of the Cray XE6 architecture is the Gemini interconnect, which provides hardware support for global address spaces. There are two mechanisms to transfer the internode messages using one-sided communication with Gemini: Fast Memory Access (FMA) and Block Transfer Engine (BTE). The FMA transport mechanism involves the CPU, has low latency and allows more than one transfer to be active at the same time. Transfers using the BTE are performed by the Gemini network chip, asynchronously with CPU so that the communication and computation can overlap. In general, FMA is used to transfer short messages and BTE for long messages. The point at which this transition occurs is controlled by the environment variable `MPICH_GNI_RDMA_THRESHOLD` for MPI and by `PGAS_OFFLOAD_THRESHOLD` for PGAS languages. The torus in the XE6 is asymmetrical in its performance capabilities, the links in the X and the Z directions are capable of double the bandwidth of the Y (9.4 GB/s vs 4.7 GB/s). The injection bandwidth of a single node is 6.1GB/s (peak).

IV. Messaging Rates

Measuring the messaging rate is important to determine the viability of interconnects on HPC platforms, especially for one-sided and PGAS programming models [29]. As we move closer to exascale we expect this to increase in importance as memory constraints are likely to result in more and more strong-scaled problems.

In our earlier work [28], we have measured the messaging rate performance of the Gemini interconnect using a uni-directional communication pattern. In this section, we report the performance results using a bi-directional communication pattern, which matches our selected applications more closely.

A. Implementation

The MPI messaging rate code was obtained from OSU Micro benchmark suite [21]. It was originally designed to measure uni-directional performance. Here we changed the code slightly so that it can be used to measure the bi-directional messaging rates. The codes are executed using two sets of processes, one on each node. The set of processes on node A simultaneously send messages to their partners on node B individually, and vice versa. In the MPI version, a process sends a series of same size messages to its partner using nonblocking MPI_Isend. The number of messages is determined by a variable called "window size" and our experiments show that setting the window size to 64 is large enough to achieve converged performance. Same number of MPI_Irecvs are posted in advance to receive the messages. Finally, MPI_Waitall is used to wait for all messages to finish. This process will be repeated in a certain number of times. In the CAF implementation, the nonblocking MPI_Isend and MPI_Irecv functions are substituted by a loop with direct load/store assignment (corresponding to one-sided put operation). However for each loop iteration the starting address is incremented by one so that the data sent is not contiguous between loop iterations. (This prevents the compiler collapsing the loop over messages into one message.) In order to ensure that non-blocking communication was used the delayed synchronization compiler directive pgas defer_sync was used. Then, at the end, a synchronization call is made to ensure all data has been received.

B. Performance

The codes are executed using two sets of processes, one set on each node. The messaging rate between two nodes using 1, 6, and 24 communicating pairs per node for MPI and CAF are shown in Fig. 2 and 3 respectively. (The two nodes have a 1-hop network distance.) For small messages, MPI achieves the best performance when 6 pairs are used, a rate of 10 million messages per second. Using 24 pairs, the message rate drops significantly. More contention for a specific hardware resources causes big performance degradation. On the contrary, the message rates of CAF for small messages increase steadily with the number of communicating pairs used. The best performance is obtained when 24 pairs are used, which is about 7.4 times better than the best MPI message rate. CAF clearly shows much better scalability for small messages. However, when message size becomes larger than 64 bytes, the performance for 24 pairs also drops below of that for 6 pairs.

For very small messages aggregation is still beneficial. As shown in Fig. 3, the messaging rate for 8-byte and 16-byte messages is very close, thus using 16-byte messages will achieve almost double the bandwidth of 8-byte messages. Even so, because of the increased messaging rate, in the latency limit the PGAS effective bandwidth for 8-bytes messages is about 7.4× the MPI one.

The corresponding bandwidths for the message rates shown in Fig. 2 and 3 are shown in Fig. 4. For clarity, only results

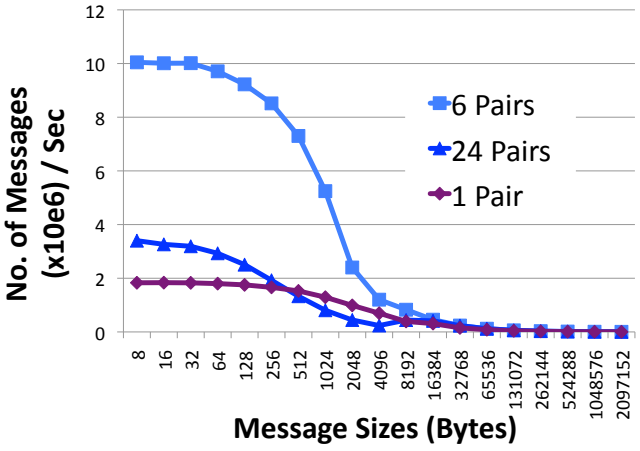

Fig. 2. The bi-directional messaging rate for MPI using 1, 6, and 24 pairs per node.

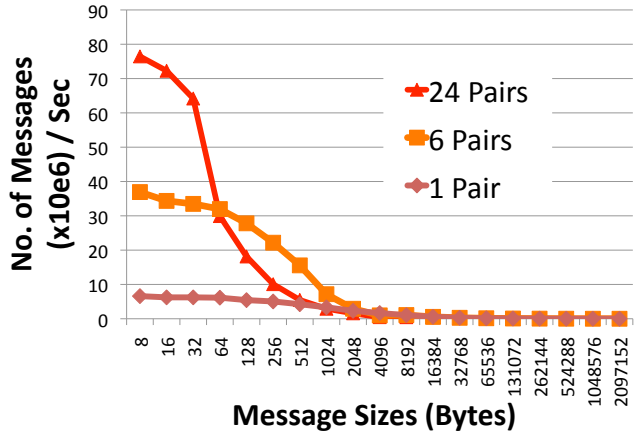

Fig. 3. The bi-directional messaging rate for CAF using 1, 6, and 24 pairs per node.

for 6 pairs and 24 pairs are shown. The CAF performance increases much faster with increasing message size. The main performance difference between CAF and MPI occurs for small message sizes where the benefits of single-sided messaging are mostly strongly felt. For a 32 byte message, using 24 pairs, the CAF performance has reached around 2GB/s, which is much higher than the corresponding MPI bandwidth of 0.1 GB/s.

There is a performance drop for CAF when the message size reaches 4096 bytes, which is the threshold in CAF to switch from the FMA mechanism to using the BTE. The startup cost for the BTE is the reason for the sudden performance drop, which cannot be amortized well for such small message sizes. We also note that as more communicating pairs are used, the phenomenon becomes more explicit, which is simply because the BTE processes requests through the kernel and therefore sequentially which means the startup cost will be accumulated as more communicating pairs are used. For large messages, CAF usually performs better or close than MPI except for

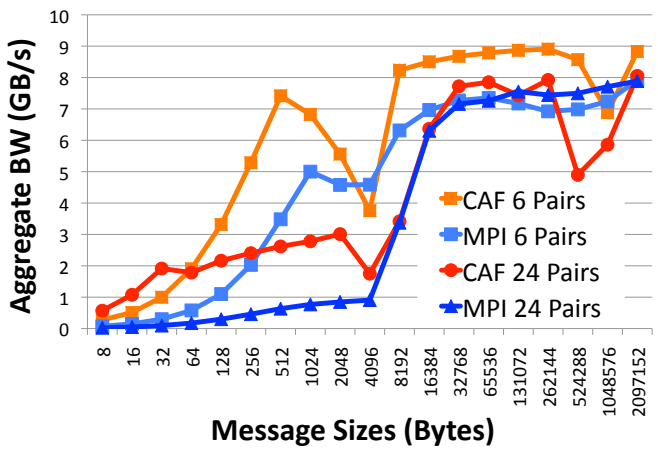

Fig. 4. The corresponding bandwidth of MPI and CAF for message rates in Fig. 2 and 3 for 6 and 24 pairs.

message sizes between 512KB and 1MB, where the CAF has another sudden performance drop due to protocol change. Compared the performances of 6 pairs and 24 pairs, we can find that for most cases using 6 pairs actually delivers better performance than using 24 pairs. This is probably due to that using 24 pairs can exhaust the interconnect resources very fast and causes more contention.

We also performed the same measurements using UPC and found that it delivers almost identical performance to CAF. (Results not shown.)

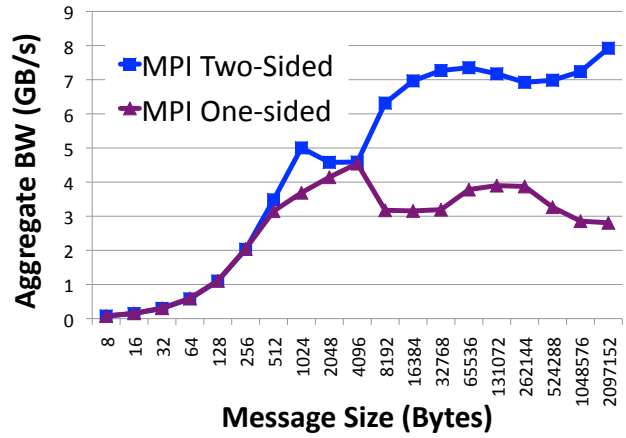

Fig. 5. The corresponding bandwidth for MPI one-sided and two-sided using 6 pairs per node.

We also developed the messaging rate benchmark in MPI-2 one-sided using MPI_Put. The results in Fig 5 show that MPI-2 one-sided performs poorly overall. Both MPI one-sided and two-sided deliver similar performance for small messages up to 4096 bytes, but, for messages larger than 4096 bytes, the MPI one-sided implementation performs significantly worse than the MPI two-sided one, by greater than 2.5x. Thus it is unlikely any performance advantage will be observed using MPI-2 one-sided at this time, and we do not explore it further

here. Performance measurements using the OSU MPI_Put and MPIGet benchmarks [21] show same results as our messaging rate benchmark.

V. MILC

MILC is a large scale numerical simulation code developed by MIMD Lattice Computation (MILC) Collaboration [18] to study quantum chromodynamics (QCD). The goal of these calculations is to understand the strong dynamics of quarks and gluons to extract the fundamental parameters for the Standard Model of particle physics. The basic structure of the problem is a four-dimensional rectangular lattice connected with links. The 4-D lattice encompasses the three continuous dimensions of space (X, Y, Z) and a fourth dimension of time (T). The most time-consuming part of MILC is the computation of the force coming from the dynamical quark fields, which is a non-local force on the gluon fields. Computation of this force requires solution of a large Hermitian positive-definite sparse matrix problem, which is solved using the conjugate-gradient algorithm [19].

In this work we use v 7.6.0.1 of the MILC code which is part of the NERSC6 benchmark suite. We use two problem sizes defined as 32^3x72 and 64^3x144 global lattices. Each has 2 quark flavors, four trajectories and 15 steps per trajectory; this results in over 35,000 CG iterations per run.

A. Parallel Implementation

MILC is developed to be run on large scale computing platforms. The global lattice is evenly divided among the processes and each process is assigned a sub-lattice. The partition method is called "Hyper Prime", which partitions the longest dimension first. In turn, it prefers to divide dimensions that have already been divided to reduce off-node directions. In a four-dimensional lattice, each site has eight nearest neighbors, four in the positive directions and four in the negative directions. In order to reduce the error caused by the "lattice distance", the third nearest neighbors also need to be fetched. If the nearest neighbor of a site does not belong to the same process, communication via message passing is used.

Once the partition is defined, the message patterns and message sizes become fixed. The operations to access data of the nearest neighbor sites are organized as *gather* operations. There are total 16 *gather* operations, 8 for the nearest neighbors (4 for positive directions and 4 for negative directions) and 8 for the 3rd nearest neighbors. Each *gather* operation is assigned a pointer array giving either the address of local field data or address of the remote data in the message receive buffer. Prior to each *gather* operation, a data structure is built containing information about what sites have to be sent to other processes or received from other processes. Data from those sites destined for the same remote process will be merged and sent in a single message.

All the communications in MILC are organized into one source file and the same interfaces are used. The purpose is to limit the changes into one file if different communication models or algorithms are used and only the underlying implementations need to be replaced.

B. MPI Implementation

The communication in MILC is organized based on message passing. Therefore, implementation using MPI does not require code modification. MPI_Isend and MPI_Irecv are used in the function `do_gather` to carry out the non-blocking functions. MPI_Waitall is used in `wait_gather` to wait for all messages to finish. The message tags used in the MPI_Isend and MPI_Irecv functions are used to differentiate `gather` operations from different directions and match the send and receive messages. Typically in between `do_gather` and `wait_gather` calls some computation is performed in order to maximize the potential communication/computation overlap. MPI is the only programming model currently in production use for MILC on large-scale distributed platforms.

C. UPC Implementation

In this section, we will describe in detail how we modified MILC code from MPI to UPC using an incremental approach. Our first target is to replace the nonblocking MPI functions in `do_gather` using one-sided get or put operations. In the standard UPC, only blocking functions `upc_memget` and `upc_memput` are defined. However, on the Cray XE6, Cray provides corresponding non-blocking functions `upc_memget_nb` and `upc_memput_nb`. These functions return a upc_handle which can be used by `upc_sync_nb` to ensure the function has finished. The potential advantage of using non-blocking functions is to overlap the communication and computation. We choose to use `upc_memget_nb` to replace the MPI_Isend and MPI_Irecv in `do_gather` and `upc_sync_nb` to replace function MPI_Waitall in `wait_gather`.

A major difference between one-sided and two-sided messaging is that the one-sided functions need to know the addresses of both the source and destination buffers, though they do not belong to the same thread. One of them should be a global address. `Upc_memget_nb` has three parameters, `source, destination`, and `message size`. The `destination` and `message size` can be obtained from the parameters of MPI_Irecv. However, the `source` parameter can not be directly copied from the parameters of MPI_Isend. We need to promote it from local variables in MPI to shared variables in UPC.

In MPI implementation, the source buffer is dynamically allocated in function `prepare_gather`. In UPC, if we follow the same approach to dynamically allocate the buffer, the thread which initiated the get call has to get this address first from the thread which actually allocates the buffer. As a result, one more round of communication is required. To avoid such complexity, we allocate the source buffer in advance, as the fixed communication patterns of MILC enable us to compute the buffer size from the beginning. Each gather operation needs at least one buffer associated with it. As there are total 16 gather operations, eight for the nearest neighbor

in positive and negative X,Y,Z,T directions and eight for the 3rd nearest neighbors. In the conjugate-gradient solver, all 16 gather operations will be on the fly and two fields need to be accessed at the same time, therefore, we associate two buffers for each gather operation. The source buffers are defined as follows in our UPC implementation and allocated using `upc_alloc` which allocates the shared space with affinity to the calling thread.

```
shared [] char *shared upcbuf[THREADS][32];
```

Due to the one-sided messages, the buffer management in UPC becomes more explicit while in MPI, the matching of sending and receiving buffers is implicitly done through message tags and process ids.

Another difference is that before calling `upc_memget_nb` to fetch the data, users have to guarantee that the data in the shared source buffer has become available. For this purpose, our first method is to call a `upc_barrier` before calling `upc_memget_nb` to assure the data correctness.

Surprisingly, the `upc_barrier` turns out to be very expensive on Cray XE6 (see next subsection) though the size of the sub lattice assigned to the UPC threads are equal, and thus the code should be load-balanced. Further study reveals that this is the due to topology effects on the torus interconnect; some messages for the barrier synchronization have to go through a longer path than others. To avoid the expensive barrier operation, we implement a point-to-point synchronization through volatile shared variables, thanks to the shared address space provided by UPC. A UPC thread can spin on a volatile shared variable to wait for its content to be changed before calling get to fetch the data. On the other side, once the data has been prepared, the thread responsible for preparing the data will notify the receiver by changing the contents of the corresponding volatile shared variable. This optimized version is called "UPC Opt" and the above implementation using *upc_barrier* is called "UPC Naive".

Compared with original MPI implementation, the total change is about 60 lines. Though the number of changed lines is small, it is not straightforward to automate this process. `Upc_memget_nb` has three parameters, `source`, `destination`, and `message size`. While the `destination` and `message size` can be derived directly from the corresponding MPI_Irecv function, finding the matching source buffer is difficult, especially when the neighbor sites for one direction lie on more than one UPC threads. Furthermore, the source buffer is dynamically allocated in the MPI version, it changes whenever the function *prepare_gather* is called. It will be challenging for an automatic program to figure out a method to get the address.

Note that changing the code to use UPC does not change the communication pattern. The same size point-to-point messages are organized in the same order as in the MPI version. The synchronization characteristics in each case are slightly different though. In two-sided MPI, synchronization is implicit and managed by the underlying library implementation while in UPC it is explicit, users have to guarantee the data availability and correctness.

D. Performance and Scalability

A performance metric sites/second has been used which measures how many lattice sites can be processes per second. The calculation formula is:

$$(N_x \times N_y \times N_z \times N_t)/TotalRunningTime$$

where N_x, N_y, N_z, N_t are the lattice sizes in the X, Y, Z, and T directions. Two global lattice sizes are used, $32^3 \times 72$ and $64^3 \times 144$ in X, Y, Z, and T directions, respectively. The second lattice is 8 times larger than the first. The strong scaling results from 512 to 32,768 cores for the small and large problems are shown in Figs. 6 and 7 respectively.

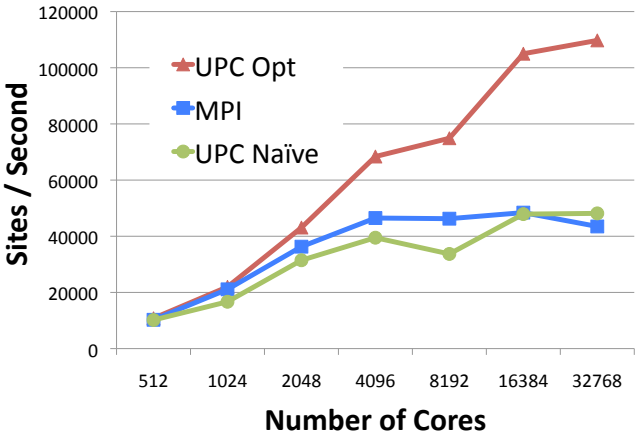

Fig. 6. The performance of MILC for a small lattice $32^3 \times 72$.

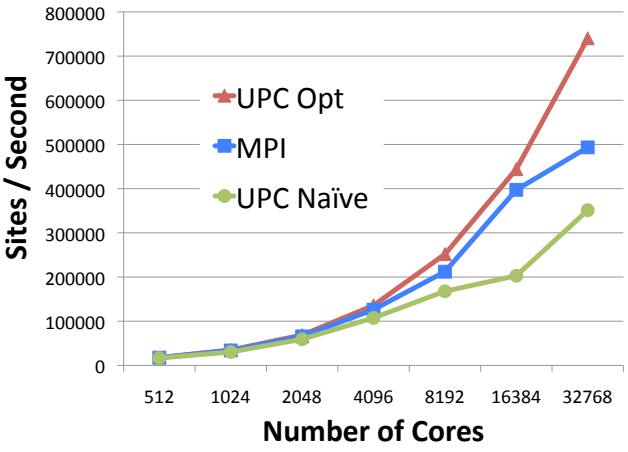

Fig. 7. The performance of MILC for a large lattice $64^3 \times 144$

The UPC implementation using point-to-point synchronization (labeled 'UPC Opt') performs best for both problem sizes. When 512 cores are used, it delivers very close performance to MPI. However, when 32,768 cores are used, the UPC performance is 1.5 times that of the MPI implementation for the large lattice and 2.5 times for the small lattice.

The naïve UPC code always delivers the worst performance. By examining the time breakdowns, we find that this is mainly caused by the *upc_barrier* operation, which becomes very expensive, especially when the number of cores becomes large. For example, over 70% of the total running time is consumed by this barrier when 32,768 cores are used, even for the larger lattice. By applying the performance tool CrayPat [6], we found that this is mainly caused by the waiting time inside the barrier operation to wait for all the threads to finish. But, the higher waiting time is not caused by load imbalance, as the workload has been perfectly partitioned among the threads. It's the topology effect of the torus. Some messages pass through more interconnect hops than others.

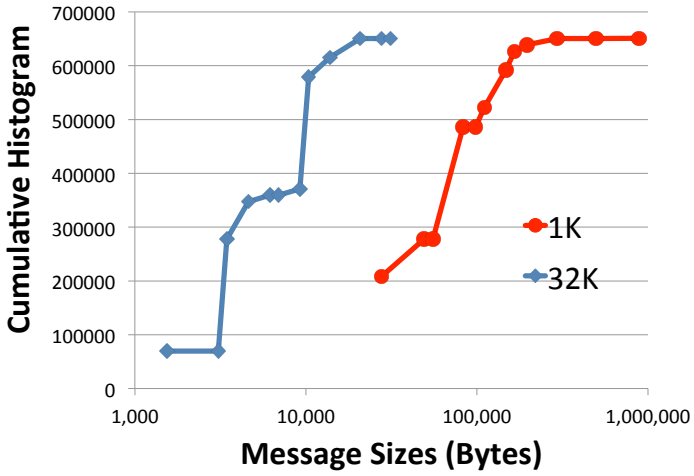

Fig. 8. The cumulative message size distribution for lattice size $64^3 \times 144$ when 1k and 32k cores are used.

As one scales these MILC calculations to larger core counts in strong scaling mode, the sub-lattice assigned to a core will become smaller, causing the surface-to-volume ratio of the local sub-lattice to increase. As a result, the total communication volume will increase with the number of cores while the individual message size will become smaller at the same time. Fig. 8 shows the message size transition for the large lattice ($64^3 \times 144$) when the number of cores being used increases from 1,024 to 32,768. When 1,024 cores are used, the message size for the gather operations range from 27KB to 880KB. When the number of cores reaches 32,768, the range of message sizes decreases to between 1.5KB and 27KB. Thus the performance advantage at 32,768 cores for the one-sided UPC version as opposed to the MPI one can be understood with reference to Fig 4. At 32,768 cores, the message sizes are in the 'sweet-spot', where one-sided messaging has the greatest performance advantage over two-sided MPI.

VI. IMPACT-T

IMPACT-T simulates the dynamics of a beam of electrons (or other charged particles) within a particle accelerator. IMPACT-T is used for both accelerator design and theoretical beam physics. The performance of a new (or proposed) accelerators can be evaluated and optimized on the basis of IMPACT-T calculations. Detailed understanding of collective phenomena effecting beams, such as space-charge driven microbunching, can only be acheived using large-scale simulations with a realistic number of particles.

IMPACT-T uses the Particle-in-Cell (PIC) technique to simulate the particles' motion; particles move through a 3-D phase-space and the electrostatic field is stored on a grid. Typical runs begin by sampling the initial particle positions and velocities from a user-selected probability density function. A two-step leapfrog algorithm with small time-step errors is then used to integrate the particles relativistic equations of motion. For each time-step, the positions are incremented according to their current velocities, the electromagnetic (EM) fields are computed at the updated positions, then the momenta are incremented based on the EM fields. Evaluating the EM field is the most time-consuming part of an IMPACT-T calculation. The EM field has two components: an external field created by the accelerating structure and the space-charge field due to Coulomb interactions among the particles. The PIC approach accelerates the evaluation of the space-charge field by replacing direct Coulomb summation with the solution of Poisson's equation on a grid. IMPACT-T uses an Poisson solver based on the Fast Fourier Transform (FFT) and an integrated Green's function to reduce discretization errors for cells with large aspect ratios [26].

A. Parallel Implementation

IMPACT-T is parallelized using a 2-D domain decomposition, whereby the computational domain is divided into subdomains by slices through the Y and Z axes. Each process is assigned one subdomain and is responsible for storing and updating the particles and grid points within its subdomain. The particle load is balanced within the sliced dimensions by adjusting the width of each slice so that all slices have the same number of particles. However, residual load imbalances are an unavoidable result whenever the 3-D particle distribution functions cannot be factored into three 1-D functions.[1] Particles are transferred between processes whenever they move between subdomains, and the subdomain boundaries are periodically recomputed to maintain optimal load balance.

The parallel algorithm involves four communication phases: 1) redistribution of subdomains to ensure load balancing, 2) particle movement between processor subdomains, 3) exchange of ghost cells between neighboring subdomains, and 4) solution of Poisson's equation. Here, we port only the Poisson equation solver because the other communication phases scale well and typically contribute very little to the total run time.

The Poisson solver used by IMPACT-T follows the FFT-based approach of Hockney and Eastman[11], which requires 3-D Fourier transforms of the Green's function and charge density, followed by a 3-D inverse FFT of their product. The 3-D FFTs are implemented as a sequence of 1-D FFTs. In

[1]The condition for load balancing is actually somewhat weaker- only the integrated 2-D PDF need be factorizable, i.e. $\int \rho(x,y,z)dx = \rho_y(y)\rho_z(z)$.

the 2-D domain decomposition, one dimension of the grid is always stored locally and contiguously, which permits each 1-D FFT to be executed with an optimized serial algorithm. The data must be transposed before the 1-D FFT can be applied to another dimension. The solver's parallel performance depends critically on this series of parallel transposes.

B. MPI Implementation

The communication in IMPACT-T was originally implemented with MPI and no changes were required for our experiments. Each process is a member of two MPI communicators, one for members of the same Y-slice, and another for members of the same Z-slice. A parallel transpose is accomplished by calling MPI_Alltoallv within either the Y- or Z-communicator. The performance is optimized by message aggregation: the Green's function and charge density buffers are merged prior to the transpose so that both arrays can be transposed with a single call to MPI_Alltoallv.

C. CAF Implementation

The essence of our transition to CAF is, simply, to replace the MPI_Alltoallv calls with direct accesses to remote coarrays. Our baseline CAF implementation, "CAF-get", uses one-sided get operations, as shown:

```
DO j=1,np_comm
   srcid = order_comm(j)
   spos  = sdispls( rank_comm+1 )[ srcid ]
   rpos  = rdispls( srcid )
   rnum  = recvcounts( srcid )
   recvbuf( rpos+1 : rpos+rnum ) = &
     sendbuf( spos+1 : spos+rnum )[ srcid ]
ENDDO
```

In this example, the sendbuf, sdispls, recvbuf, rdispls and recvcounts variables have the same meanings used by the MPI standard [20], though sdispls and sendbuf must be declared as coarrays. Instead of an MPI communicator data structure, we use the variables np_comm, order_comm, and rank_comm to store, respectively, the size of the 1-D communicator, an ordered list of images in the communicator, and the rank of the local image within the communicator. Deriving the values for order_comm is not straightforward as it is related with how the global domain is partitioned along the Y and Z directions, making this process difficult to be automated. The remaining variables, srcid, spos, rpos and rnum, are temporary and used only to improve readability. The sendtype and recvtype arguments to MPI_Alltoallv are not necessary, as the CAF compiler obtains this information from the declarations of sendbuf and recvbuf. One-sided communication also obviates the sendcounts argument. The restriction that coarrays have the same size for all images introduces a minor complication relative to MPI, which is easily overcome by determining the maximum size before allocating the coarrays.

An improved CAF implementation, "CAF-opt", incorporates two optimizations. First, the loop order is altered so that image j gets data from image order_comm(j) first, followed by order_comm(j+1) and so forth. This avoids the circumstance where all images get from the same image simultaneously. Second, each *get* is split into smaller messages (blocks) in order to maximize the effective bandwidth as shown in Fig. 4.

We have also implemented the baseline and optimized CAF algorithms using put operations. The put versions performed similarly to the get versions.

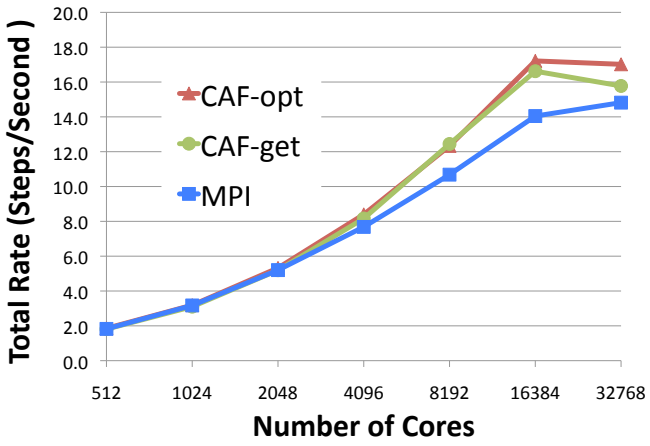

Fig. 9. Strong scaling of the full IMPACT-T application using MPI, naive CAF and optimized CAF.

D. Performance

Our IMPACT-T performance benchmark simulates 50 million particles using a 128x128x1024 grid in the X, Y and Z directions, respectively. The simulation lasts 400 time-steps, and the first 100 steps, which are not representative the normal performance and use of IMPACT-T , are excluded from our measurements. Fig. 9 plots the strong scaling performance of the full IMPACT-T application from 512 to 32,768 cores. At low concurrencies, the performance of the CAF versions is not significantly different from the MPI version. As more cores are used, the CAF versions' performance increase more rapidly than the MPI code, and reaches a maximum of 18% improvement at 16,384 cores. Except at the highest concurrencies, the CAF-get performance is comparable to CAF-opt.

The performance of the transpose portion of IMPACT-T is measured by transpose "batches" per second. A batch of transposes consists of all the variously sized transposes during one step of the simulation. Fig. 10 shows the strong scaling performance of the transpose kernel. Again, there is little difference between the MPI and CAF versions up to 2048 cores. The performance advantage of CAF increases at higher concurrencies, up to nearly 50% at 16,384 cores. This explains the overall performance increase observed, as the transpose phase is approximately 45% of the total runtime.

The better performance of CAF is mainly due to the greater effective bandwidth it achieves. By choosing a block size such that CAF always performs better or close to MPI. we are able to maximize the one-sided communication benefits. Also, as the CAF version sends messages out in a round-robin fashion we avoid causing hot spots on the network.

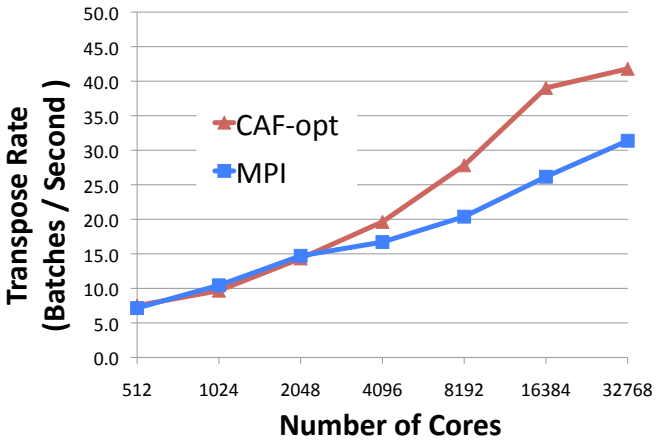

Fig. 10. Strong scaling of the transpose kernel in IMPACT-T using MPI and optimized CAF implementations.

VII. Related Work

The related work can be classified into two categories based on the programming languages targeted: CAF and UPC. Several researchers have studied how to port MPI applications into UPC and compared their relative performance [16], [8], [2], [14], [27], [22]. But the studies mainly focused on synthetic benchmarks including NPB [23] and Challenge [12]. Mallon et.al. [16] evaluated the performance of MPI, OpenMP, and UPC on a machine with 142 HP Integrity rx7640 nodes interconnected via InfiniBand. The authors claim that MPI is the best choice to use on multicore platforms, as it takes the highest advantage of data locality. El-Ghazawi and Cantonnet [8] discussed UPC performance and potential advantage using NPB applications. With proper hand tuning and optimized collective libraries, UPC delivered comparable performance to MPI. Shan [27] and Jin [14] also compared the performance of NPB on several different platforms and found that UPC delivered comparable performance to MPI. Nishtala and other researchers [22], [2] discussed the scaling behavior and better performance of NAS FT for UPC on several different platforms using the Berkeley UPC compiler with GASNET communication system. The UPC version is developed from scratch to take advantage of the lower overhead of the one-sided communication and the overlap of communication and computation. Recently, UPC has been used to study the performance of NBODY problems [32], [7].

Porting MPI applications into CAF has also been the subject of a lot of research publications. Preissl et al. [25] ported a communication skeleton extracted from a Gyrokinetic Fusion Application from MPI into CAF and showed the better scalability of the new CAF algorithm. However, their focus is to develop an efficient new communication algorithm by taking advantage of the one-sided communication provided by CAF. A radical approach has been taken which requires programmers to understand every detail of the algorithm. Critian et al. examined the performance for MG, CG, BT, and SP from the NAS parallel benchmark suite and the Sweep3D neutron transport benchmark on several small clusters. The CAF programs show nearly equal or slightly better performance than their MPI counterparts [4], [5]. Recently, the Challenge benchmark suite [12] has been converted into CAF programs using CAF2.0 [17] and tested on a Cray XT4 platform [13]. The authors have scaled the benchmark applications to 4096 CPU cores and shown CAF2.0 is a viable PGAS programming model for scalable parallel computing. Barrett [1] studied different Co-array Fortran implementations of Finite Differencing Methods on Cray X1 and found that CAF exhibits better performance for smaller grid sizes and similar results for larger ones. Similarly, Worley and Levesque [31] also found using CAF for latency sensitive communications for the Parallel Ocean Program can improve the performance and scalability.

The principal difference of our study is that we focus on how to modify real scientific applications written in the two-sided MPI programming model into one-sided communication with minimal programming effort. Secondly, we are more interested in results on large-scaling computing platforms than on small clusters to prepare us for the future exascale platforms.

VIII. Summary and Conclusions

In this paper, we selected two real scientific applications, MILC and IMPACT-T , and investigated the performance benefits and the programming effort associated with replacing their two-sided MPI communication with one-sided communication expressed by UPC and CAF. The results indicate that naive implementations of the one-sided algorithms may not perform as well as the corresponding MPI versions; the expense of the addition of global synchronization to ensure data correctness outweighs the performance gain due to the one-sided communication itself. However, by employing synchronization strength reduction (replacing global synchronization with cheaper point-to-point synchronization) the one-sided versions outperform the MPI two-sided versions significantly, especially at large-scale. For MILC we find that it performs over 1.5 times better using one-sided communication via UPC than the original MPI version running on 32K cores of a Cray XE6. Our analysis of the MILC message sizes and the message rate microbenchmark indicate that the optimized UPC MILC code has the largest performance advantage over the MPI code when the message sizes allow UPC to provide the greatest increase in effective bandwidth. For IMPACT-T we find the one-sided version performs 1.2 times better at 16K cores than the original MPI one, and the transpose kernel of the code, which is the dominant communication routine in the code, performs 1.5 times faster.

The programming effort needed for partial adoption of UPC or CAF is minimal. We changed fewer than 100 lines of the source code for each application, both of which have over 70,000 lines of code in total. At this point in time it seems that automating this process will be difficult. However, we think that the process we have outlined, to identify the crucial communication routines of an application and to replace those with one-sided communication routines is straightforward. This procedure should allow application developers to gain

performance benefits of one-sided communication today, as well as allow them to begin to explore potentially greater performance gains from moving to more fine-grained messaging schemes [25].

The performance of the naively programmed PGAS versions was comparable (or worse) than the MPI two-sided versions. However, by applying good PGAS programming practices we achieved significantly greater performance. Thus although some performance tuning was required, the effort was not significant, and certainly no greater than that required to tune the performance of an MPI two-sided application.

In future work we plan to focus upon developing better communication optimizations based on one-sided communication to address problems such as load imbalance, as well as to apply the techniques we have outlined here to more applications. Also we plan to monitor the progress of the MPI-3 one-sided standard, and revisit the question of whether MPI one-sided can achieve similar performance to that of PGAS when appropriate.

IX. Acknowledgements

We would like to thank Nathan Wichmann of Cray for useful comments and suggestions. All authors from Lawrence Berkeley National Laboratory were supported by the Office of Advanced Scientific Computing Research in the Department of Energy Office of Science under contract number DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] Barrett, R. Co-Array Fortran Experiences with Finite Differencing Methods. In *The 48th Cray User Group meeting, Lugano, Italy* (May 2006).

[2] Bell, C., Bonachea, D., Nishtala, R., and Yelick, K. Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap. In *20th International Parallel and Distributed Processing Symposium IPDPS* (April 2006).

[3] Carlson, W. W., Draper, J. M., Culler, D. E., Yelick, K., Brooks, E., and Warren, K. Introduction to UPC and language specification. In *Tech. Rep. CCS-TR-99-157 May* (May 1999).

[4] Coarfa, C., Dotsenko, Y., Eckhardt, J., and Crummey, J. M. Co-Array fortran performance and potential: An NPB experimental study. In *In Proc. of the 16th Intl. Workshop on Languages and Compilers for Parallel Computing* (2003).

[5] Coarfa, C., Dotsenko, Y., and Mellor-Crummey, J. Experiences with sweep3d implementations in co-array fortran. In *The Journal of Supercomputing, 36:101-121* (May 2006).

[6] CrayPat. http://docs.cray.com/books/S-2315-50/html-S-2315-50/z1055157958smg.html.

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

[8] El-Ghazawi, T., and Cantonnet, F. UPC performance and potential: A NPB experimental study. In *In Supercomputing* (2002).

[9] ASCAC Subcommittee Report: The Opportunities and Challenges of Exascale Computing. http://science.energy.gov/~/media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf.

[10] Geist, A. Sustained Petascale: The Next MPI Challenge. In *EuroPVMMPI* (October 2007).

[11] Hockney, R. W., and Eastwood, J. W. *Computer Simulation Using Particles*. Taylor & Francis, Jan. 1989.

[12] HPC Challenge Benchmark. http://icl.cs.utk.edu/hpcc/index.html.

[13] Jin, G., Mellor-Crummey, J., Adhianto, L., III, W. N. S., and Yang, C. Implementation and Performance Evaluation of the HPC Challenge Benchmarks in Coarray Fortran 2.0. In *25th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Anchorage, AK. May 16-20,* (2011).

[14] Jin, H., Hood, R., and Mehrota, P. A practical study of UPC with the NAS parallel benchmarks. In *Partitioned Global Address Space Languages* (Oct., 2009).

[15] Extrascale Programming Challenges Workshop Report. http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/ProgrammingChallengesWorkshopReport.pdf.

[16] Mallon, D. A., Taboada, G. L., Teijeiro, C., Tourino, J., Fraguela, B. B., Gomez, A., Doallo, R., and Mourino, J. C. Performance evaluation of MPI, UPC, and OpenMP on multicore architectures. In *Euro PVM/MPI 2009* (Sept 7-10, 2009).

[17] Mellor-Crummey, J., Adhianto, L., III, W. N. S., and Jin, G. A new vision for Coarray Fortran. In *In Proceedings of the 3rd Conference on Partitioned Global Address Space Programming Models, PGAS '09, pages 5:1-5:9, New York, NY, USA* (2009).

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

[19] Montvay, I., and Munster, G. Quantum Fields on a Lattice. In *Cambridge University Press, Cambridge* (1994).

[20] MPI-2: Extensions to the Message Passing Interface. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[21] Osu micro-benchmark. http://mvapich.cse.ohio-state.edu/benchmarks/.

[22] Nishtala, R., Hargrove, P., Bonachea, D., and Yelick, K. Scaling Communication-Intensive Applications on BlueGene/P Using One-Sided Communication and Overlap. In *23rd International Parallel and Distributed Processing Symposium (IPDPS)* (2009).

[23] NAS Parallel Benchmarks. http://www.nas.nasa.gov/Resources/Software/npb.html.

[24] Numrich, R. W., and Reid, J. Co-array Fortran for parallel programming. In *ACM SIGPLAN Fortran Forum, vol. 17, no. 2, pp. 1-31* (August 1998).

[25] Preissl, R., Wichmann, N., Long, B., Shalf, J., Ethier, S., and Koniges, A. Multithreaded Global Address Space Communication Techniques for Gyrokinetic Fusion Applications on Ultra-Scale Platforms. In *SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (November 2011).

[26] Qiang, J., Lidia, S., Ryne, R. D., and Limborg-Deprey, C. Three-dimensional quasistatic model for high brightness beam dynamics simulation. *Phys. Rev. ST Accel. Beams 9* (Apr 2006), 044204.

[27] Shan, H., Blagojevic, F., Min, S. J., Hargrove, P., Jin, H., Fuerlinger, K., Koniges, A., and Wright, N. J. A Programming Model Performance Study Using the NAS Parallel Benchmarks. In *Scientific Programming-Exploring Languages for Expressing Medium to Massive On-Chip Parallelism, Vol. 18, Issue 3-4* (August 2010).

[28] Shan, H., Wright, N. J., Shalf, J., Yelick, K., Wagner, M., and Wichmann, N. A preliminary evaluation of the hardware acceleration of the cray gemini interconnect for PGAS languages and a comparison with MPI. In *In SIGMETRICS Performance Evaluation Review 40(2)* (2012).

[29] Underwood, K. D., Levenhagen, M. J., and Brightwell, R. Evaluating NIC hardware requirements to achieve high message rate PGAS support on multi-core processors. In *SC07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing. New York, NY, USA* (2007).

[30] Challenges for the Message Passing Interface in the Petaflops Era. www.cs.uiuc.edu/homes/wgropp/bib/talks/tdata/2007/mpifuture-uiuc.pdf.

[31] Worley, P., and Levesque, J. The Performance Evolution of the Parallel Ocean Program on the Cray X1. In *Cray User Group Conference (CUG)* (2004).

[32] Zhang, J., Behzad, B., and Snir, M. Optimizing the Barnes-Hut algorithm in UPC. In *SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011).