Title
Evaluation of PGAS Communication Paradigms with Geometric Multigrid

ISBN
9781450332477

Authors
Shan, Hongzhang
Kamil, Amir
Williams, Samuel
et al.

Publication Date
2014-10-06

DOI
10.1145/2676870.2676874

Peer reviewed

Evaluation of PGAS Communication Paradigms with Geometric Multigrid

Hongzhang Shan, Amir Kamil, Samuel Williams, Yili Zheng, Katherine Yelick
Computational Research Division
Lawrence Berkeley National Laboratory, Berkeley, CA 94720
hshan, akamil, swwilliams, yzheng, kayelick@lbl.gov

ABSTRACT

Partitioned Global Address Space (PGAS) languages and one-sided communication enable application developers to select the communication paradigm that balances the performance needs of applications with the productivity desires of programmers. In this paper, we evaluate three different one-sided communication paradigms in the context of geometric multigrid using the miniGMG benchmark. Although miniGMG's static, regular, and predictable communication does not exploit the ultimate potential of PGAS models, multigrid solvers appear in many contemporary applications and represent one of the most important communication patterns. We use UPC++, a PGAS extension of C++, as the vehicle for our evaluation, though our work is applicable to any of the existing PGAS languages and models. We compare performance with the highly tuned MPI baseline, and the results indicate that the most promising approach towards achieving performance and ease of programming is to use high-level abstractions, such as the multidimensional arrays provided by UPC++, that hide data aggregation and messaging in the runtime library.

1. INTRODUCTION

Partitioned Global Address Space (PGAS) languages support efficient one-sided communication and offer programming abstractions similar to shared memory, while also exposing locality to the programmer. This enables users to develop applications and tailor the communication paradigm to suit their performance and productivity needs. At one extreme, programmers can aggregate communication in MPI's message-passing style to maximize performance on current high-performance computing (HPC) platforms. At the other extreme, programmers can use fine-grained messaging to reap the productivity benefits of shared memory programming by not having to distinguish whether accesses are local or remote. Between these two extremes, there exists a wide programing design space in which users may explore the tradeoffs. Ultimately, quantifying the performance benefits and qualitatively assessing the ease of programming requires empirical data obtained by running experiments on today's HPC systems.

In this paper, we explore this design space using a compact geometric multigrid (MG) benchmark called miniGMG [31, 30, 23], which is designed to proxy the multigrid solvers in adaptive mesh-refinement (AMR) applications. As a standalone benchmark, it provides an excellent testbed for understanding the implications of dramatic changes to programming models and communication paradigms. Its communication is primarily between nearest neighbors with a static set of neighbors and predictable message sizes. Although MPI's traditional message-passing approach is well-suited for this kind of regular communication, the process of manually packing and unpacking the communicated data is complex, labor intensive, and potentially error prone due to the necessity of calculating the intersections and unions of subdomains with a variable-depth ghost zone. Conversely, PGAS languages often excel with irregular and dynamic applications and can be quite productive to use. Here, we evaluate the productivity and performance of the PGAS model in the context of a regular application.

Although the tradeoffs could be evaluated in any of the myriad PGAS languages and libraries, we use UPC++ [34] as our vehicle for evaluating different communication paradigms in the context of miniGMG. UPC++ is a library-based PGAS extension of C++; unlike UPC [5], it does not need special compiler support. Instead, it is developed based on C++ templates and runtime libraries. This compiler-free approach enables it to be easily ported across different platforms and interoperate well with other parallel programming models.

We develop three new implementations of miniGMG using different communication paradigms in UPC++, namely *bulk*, *fine-grained* and *array*. The *bulk* version has essentially the same communication patterns as the MPI version with the caveat that the traditional two-sided message passing is replaced with one-sided communication. The *fine-grained* version expresses communication productively and naturally in the data granularity of the algorithm without manual message aggregation. The *array* version leverages the multidimensional array constructs in UPC++ to express communication in terms of high-level algorithmic operations by allowing entire ghost zones to be copied with a simple sequence of calls.

We study the performance characteristics of four miniGMG implementations (three in UPC++ and one in MPI) on two supercomputers – a Cray XC30 and an IBM Blue Gene/Q. Our results show that in most cases, the UPC++ *bulk* version performs similarly to the highly tuned MPI code and better than the other two UPC++ versions. Conversely, the UPC++ *fine-grained* version, whose communication is dominated by 8-byte short messages, performs poorly when using only one core per socket to inject messages. How-

ever, by decomposing the application problem among multiple processes on a socket (instead of threads), the performance gap with the *bulk* version can be sharply reduced. As the memory capacity per core is expected to shrink in manycore architectures, smaller size messages are likely to be more pervasive on upcoming power-efficient computers. In our experience, using higher-level data abstractions, such as multidimensional arrays, provides both programming ease and portable performance because the communication optimizations required for realizing full performance potential (e.g., message aggregation) are implemented by the runtime software rather than by the user. We believe this is the most promising programming approach for end users, enabling maximum code reuse across applications.

2. RELATED WORK

Many studies have been done on the performance of PGAS languages and libraries. However, most previous studies have focused on comparing the performance of bulk messaging using PGAS languages to MPI rather than evaluating the different communication paradigms supported by PGAS languages. To name a few such studies, T. El-Ghazawi and F. Cantonnet [12] examined the performance and potential of UPC using the NAS Parallel Benchmarks. H. Shan et al. [25] demonstrated the performance advantage of one-sided communication over two-sided MPI at scale for two applications, MILC and IMPACT-T. J. Zhang et al. [33] studied the performance of the N-Body problem in UPC. J. Mellor-Crummey et al. [22] examined the performance of the Challenge Benchmark Suite in CAF 2.0. P. Ghosh et al. [17] explored the ordering of one-sided messages to achieve better performance. GPI-2 is an open-source PGAS communication library similar to GASNet [15] and ARMCI [1] and has been used in a number of computational applications and performance studies [26, 21, 19, 18].

Gerstenberger et al. [16] showed that MPI one-sided communication can be implemented efficiently on Cray Gemini interconnects using the DMAPP API, whereas our study focuses on an application with different computation and communication characteristics (miniGMG has much lower surface to volume ratio than MILC, which is the only comparable benchmark in that paper), uses higher-level data abstractions (multidimensional arrays vs. plain buffers), and evaluates the performance on more recent interconnects (Cray Aries and IBM BGQ).

Other studies have focused on high-level programming abstractions, such as the multidimensional arrays in UPC++. K. Datta et al. [10] studied the performance and potential of Titanium, the language on which the UPC++ array library is based. They argued that Titanium provides greater expressive power than conventional approaches, enabling concise and expressive code and minimizing time to solution without sacrificing performance. T. Wen and P. Colella compared an implementation of the Chombo adaptive mesh refinement framework using Titanium arrays to the original Fortran/C++/MPI version [29]. They showed that the Titanium version was much more succinct and productive at the cost of somewhat worse performance than the original implementation. A similar performance and productivity study was done by B.L. Chamberlain et al. [7] in ZPL, which in turn inspired the Titanium array library. Multidimensional array is also supported by Global Array [13]. However, only limited algebraic operations are directly supported. A. T. Tan et al. [28] studied the implementation of an automatic taskification on shared-memory systems for a domain-specific embedded language NT^2, which provides a Matlab-like syntax for parallel numerical computations inside a C++ library.

The novelty of our work is that we compare the *bulk*, *fine-grained*, and *array* implementations together. To our best knowledge, this is the first time that the three different PGAS communication paradigms have been evaluated in the context of a single application.

3. MINIGMG

miniGMG [31, 30, 23] is a compact geometric multigrid benchmark designed to proxy the multigrid solvers in AMR MG applications built using BoxLib [4] and Chombo [9]. As it is a standalone benchmark, it provides an excellent testbed for understanding the implications of dramatic changes to programming models and communication paradigms.

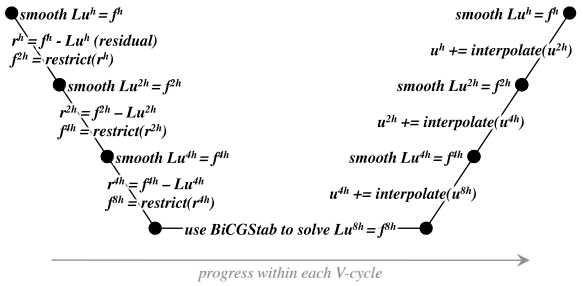

Figure 1: The truncated V-cycle (U-cycle) used in miniGMG for solving the elliptic PDE $L^h u^h = f^h$, where superscripts denote grid spacings and GSRB is the Gauss-Seidel Red-Black smoother.

3.1 Geometric Multigrid

Geometric multigrid is a recursive solver for elliptic PDEs on structured grids. In the PDE $L^h u^h = f^h$, the vectors u and f are elements of a structured rectahedral grid while the linear operator L is simply a stencil operating on the elements of the grid. The superscript h denotes the grid spacing. As shown in Figure 1, a truncated multigrid V-cycle (or U-cycle) recursively smooths (Gauss-Seidel Red-Black) and coarsens (restriction) the current grid and operator until further coarsening is impractical. At this point, the MG solver computes a solution to the resultant coarse grid problem ($L^{8h} u^{8h} = f^{8h}$) using an iterative solver like BiCGStab. Once a solution to the the coarse grid problem has been calculated, it is interpolated and used as a correction to the finer grid problems. The multigrid solver iterates over multiple V-cycles until some convergence criterion is met.

3.2 Parallelization in miniGMG

As shown in Figure 2, miniGMG uses the same straightforward domain decomposition at every level. This ensures that restriction and interpolation are entirely local operations, since a given process owns all grid spacings of data in a prescribed spacial subdomain. In Figure 2, the fine-grid domain of 8^3 cells is decomposed among eight MPI or

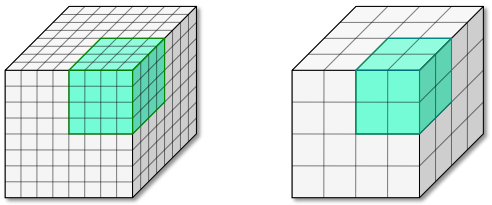

Figure 2: Domain decomposition in miniGMG is preserved across restrictions. When the fine-grid in (a) is restricted, the volume of the resultant grid (b) is reduced by a factor of eight while the surface area is reduced by a factor of four.

UPC++ ranks, each of which owns a 4^3 subdomain. After one level of restriction, each rank owns a 2^3 subdomain. This implies that:

- the amount of work (stencils) decreases by a factor of eight on each subsequent level
- ranks communicate with the same set of neighbors at every level
- the amount of data exchanged between ranks decreases by a factor of four on each subsequent level

Together, these characteristics bound both the number of stencils and the volume of interprocess data movement in the MG solver to $O(N)$ and $O(N^{0.66})$, respectively. On the other hand, the communication overhead is in $O(\log N)$, so that low-overhead communication paradigms are important for performance at all levels in miniGMG.

3.3 Communication in miniGMG

In miniGMG, communication within the V-cycle takes the form of nearest-neighbor ghost-zone or halo exchanges for the smooth and restriction operations. In MPI, these take the familiar form highlighted in Figure 3(a):

1. the local rank packs a copy of the surface of its subdomain (box) into a series of 1D MPI buffers
2. the local rank initiates `MPI_Isend` operations
3. remote ranks post `MPI_Irecv` and `MPI_Waitall` operations and wait for data
4. remote ranks unpack the 1D buffers into the ghost zones of their 3D subdomains

Although this method ensures pairwise synchronization and aggregates data to amortize overhead, the development of a high-performance implementation is error prone, as one must deal with unions of subdomains, deep ghost-zone exchanges, and communication with edge and corner neighbors. For simplicity, we focus on experiments that only require the exchange of one-element-deep ghost zones with six neighbors in three dimensions.

3.4 miniGMG Configuration

In this paper, we configure miniGMG to solve a second-order, variable-coefficient, finite-volume discretization of the Helmholtz operator ($Lu = a\alpha u - b\nabla \cdot \beta \nabla u = f$) on a cubical domain, parallelized with one cubical subdomain per rank. We use a V-cycle truncated when subdomains reach 4^3 cells, at which point we switch to a matrix-free BiCGStab iterative solver. In addition to ghost-zone exchanges, BiCGStab requires global dot products, and we use `MPI_Allreduce` to compute them in all implementations. For timing consistency, we perform a total of ten V-cycles in each experiment.

4. UPC++ OVERVIEW

UPC++ is a library for C++ that leverages standard C++ language features to provide a PGAS programming model. In this section, we give a brief overview of UPC++, focusing on the features used in our miniGMG implementations, including shared objects, dynamic global memory management, communication, and multidimensional arrays. A more complete discussion of UPC++ can be found in [34].

The memory model of UPC++ is PGAS: each rank has its own private address space as well as a partition of the globally shared address space. UPC++ provides both low-level PGAS programming primitives similar to UPC and high-level parallel programming features inspired by other PGAS languages such as Titanium [32], Chapel [6], Phalanx [14], and X10 [8]. A notable syntactic distinction is that all PGAS extensions in UPC++ are implemented by standard C++ templates, functions, or macros and thus require no change in the C++ compiler. For example, shared objects in UPC are expressed through the `shared` type qualifier while in UPC++ they are expressed through the `shared_var` and `shared_array` templates. From the application user's perspective, the programming experience is very similar.

The majority of today's HPC applications are programmed with a mixture of MPI, OpenMP, CUDA, and/or OpenCL. The library approach of UPC++ helps to provide good interoperability with these existing programming systems and enables an incremental transition path for algorithms that fit the PGAS model.

As depicted in Figure 4, our UPC++ implementation includes two main components: a set of template header files and a runtime library. In UPC++ header files, we use a combination of C++ programming techniques such as generic programming, operator overloading, and template metaprogramming to implement PGAS features. User code written in UPC++ can be conceptually thought of as being translated to runtime function calls through language hooks provided by the C++ standard.

Table 1 summarizes the basic UPC++ programming idioms. All UPC++ extensions are packaged in the `upcxx` namespace to avoid naming conflicts with other libraries. For brevity, the code examples in this paper assume that the `upcxx` namespace is being used.

4.1 Shared Objects

UPC++ has two categories of shared objects: single-location shared variables (`shared_var`) and block-cyclically distributed arrays of shared objects (`shared_array`). Regardless of their physical location, shared objects are accessible by any UPC++ rank.

Shared data types are implemented as generic templates parameterized over the object type and can work with both built-in and user-defined data types (e.g., `structs`). Since it is common to access members of a struct individually and the C++ standard does not allow overloading of the class

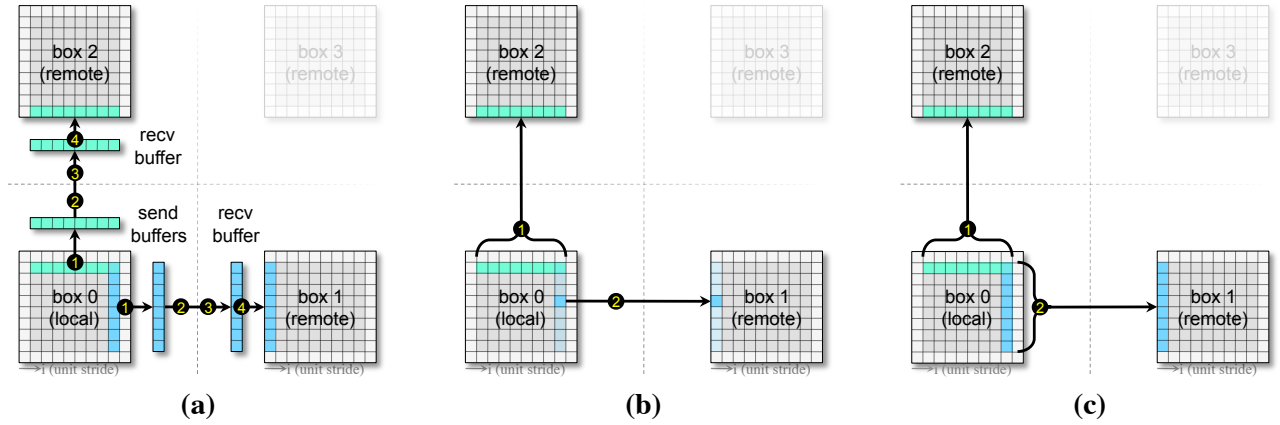

Figure 3: Communication styles explored in this paper: (a) point-to-point MPI and bulk UPC++, (b) fine-grained UPC++ based on puts of contiguous data, (c) multidimensional arrays with automatic message aggregation and active messages. Collectively, these implementations span the performance and productivity design space.

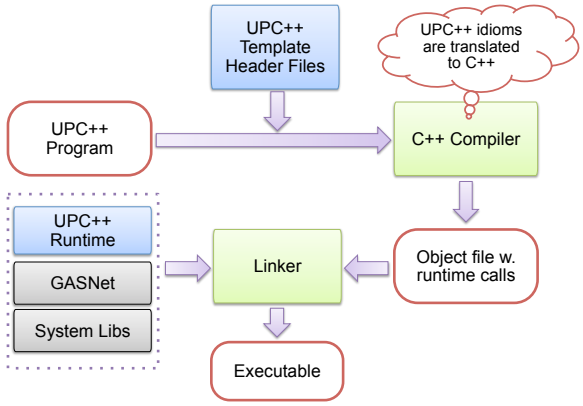

Figure 4: UPC++ software architecture and compilation workflow. By including UPC++ template header files in the user application, UPC++ programming idioms are "translated" to regular C++ code and runtime function calls by the standard C++ compiler and preprocessor. The object code from the C++ compiler is linked with the UPC++ runtime, which is implemented on top of the GASNet communication library.

member operator ".", UPC++ introduces new syntax for referencing a member of a shared object. Given an object and a member name, the `memberof` operation creates a global reference to the given member of the object, which can be used as either an l-value or an r-value. The following is an example of how to use `memberof`:

```
struct Box {
  int i, j, k;
  global_ptr<double> data;
};
shared_array<Box> boxes;
boxes.init(128*ranks());
memberof(boxes[0], i) = 1; // boxes[0].i = 1;
```

4.2 Global Memory and Communication

Table 1: Basic PGAS primitives in UPC++

Programming Idiom	UPC++
Number of ranks	`ranks()`
My ID	`myrank()`
Shared variable	`shared_var<Type> v`
Shared array	`shared_array<Type> A(count)`
Global pointer	`global_ptr<Type> p`
Memory allocation	`allocate<Type>(rank, count)`
Data transfer	`copy<Type>(src, dst, count)`
Synchronization	`barrier()` & `async_wait()`

Global memory is represented by the generic global pointer type `global_ptr<T>`, which points to one or more shared objects of type `T`. A global pointer encapsulates both the rank and the local address of the shared object referenced by the pointer. Pointer arithmetic with global pointers in UPC++ works the same way as arithmetic on regular C++ pointers. Memory in the global address space can be allocated and freed using the UPC++ `allocate` and `deallocate` function templates.

Communication in UPC++ applications may appear in two forms: 1) explicit data transfer using one-sided `copy` functions; 2) implicit communication when shared objects appear in an expression. For example, if a shared object is used as an l-value, then a *put* operation occurs. On the other hand, if a shared object is read from, then a *get* operation occurs.

UPC++ also supports non-blocking data movement using the `async_copy` function template:

```
async_copy(global_ptr<T> src,
           global_ptr<T> dst,
           size_t count);
```

The `src` and `dst` buffers are assumed to be contiguous. A call to `async_copy` merely initiates data transfer, enabling overlapping of communication with computation or other communication. The user can query the completion status

of a non-blocking copy using `async_try` or wait for completion using `async_wait`. UPC++ also allows a user to register an `async_copy` operation with an `event` (similar to an `MPI_Request`) and synchronize all operations in an `event` at some later point.

4.3 Multidimensional Domains and Arrays

The bulk `copy` and `async_copy` functions described above can only be used to transfer contiguous data from source to destination. They do not eliminate the need to pack and unpack data into contiguous buffers in the ghost-exchange process described in §3, since most ghost zones are non-contiguous. As a result, a ghost-zone exchange using `copy` or `async_copy` must be a two-sided process, where the sender packs data and initiates a transfer and the receiver unpacks the data after the transfer is completed. Significant programmer effort is required to implement and coordinate packing and unpacking, negating the productivity benefits of the one-sided PGAS model.

In order to address these limitations, UPC++ includes a multidimensional domain and array library based on that of Titanium. Full details on the array library can be found in [20]. Here, we provide an overview of the features that are used in miniGMG.

The UPC++ domain and array library includes the following components:

- *points* are coordinates in N-dimensional space
- *rectangular domains* consist of a lower-bound point, an upper-bound point, and a stride point
- arrays are constructed over a rectangular domain and indexed by points

An array consists of memory in a single memory space to hold its elements, as well as a descriptor that encodes the location of that memory and the layout of the array. An array is represented using a C++ template

```
template<class T, int N, class L = local>
class ndarray;
```

where T is the element type, N is the dimensionality, and L is an optional locality specifier that may be `local` or `global`. The former specifies that the elements are located in local memory, while the latter allows the elements to be located in a remote space. The `ndarray` template overrides the element access operator "`[]`", allowing multidimensional arrays to be accessed with point indexes. In the case of a `global` array, the element access operator allows an array to be accessed remotely in a one-sided fashion.

A multidimensional array can be constructed over any rectangular domain, so that an array's index space matches the logical domain in the application. The library allows different views to be created of the same underlying data, providing reinterpretation operations such as restrictions to a smaller domain, slicing to a smaller dimensionality, and permuting or translating the domain of an array. Most importantly, the library provides a copy operation, invoked as `A.copy(B)`, which copies data from array `B` to array `A`. The two arrays need not be located in the same memory space, and their underlying domains need not be equal. The library automatically computes the intersection of their domains, obtains the subset of the source array restricted to that intersection, packs elements if necessary, sends the data to the rank that owns the destination, and copies the data to the destination array, unpacking if necessary. The entire operation is one-sided, with active messages performing remote operations, using an implementation similar to the GASNet Vector, Indexed, and Strided (VIS) extensions [3]. Copying a ghost zone requires the single statement

```
A.constrict(ghost_domain).copy(B);
```

where `ghost_domain` is the domain of the ghost zone. The library also provides a non-blocking `async_copy` counterpart to the `copy` method.

A final feature in the array library that is relevant to the miniGMG application is that it allows an array descriptor to be created over an existing piece of memory. This enables an application to create descriptors to take advantage of the copy operations without changing the rest of the program. Thus, a programmer can improve the productivity of the communication code without touching the computation piece of an application.

5. MINIGMG IN UPC++

In this section, we describe in detail the three miniGMG implementations in UPC++, namely *bulk*, *fine-grained*, and *array*. These three versions differ in how the ghost zone exchange operation is implemented.

5.1 Shared Features

All three implementations use non-blocking one-sided operations to transfer data from sender to receiver. To avoid race conditions, synchronization is necessary at the beginning of the communication phase to ensure that the destination targets are available, as well as at the end to signify that data transfer has completed. UPC++ does not currently provide point-to-point synchronization operations, so global barriers are used instead. This is in contrast to the MPI implementation, which relies on the semantics of two-sided message transfer to synchronize between sender and receiver.

The MPI version of miniGMG uses the `MPI_Allreduce` collective operation for computing dot-products and checking convergence of the result. The UPC++ versions also use this same operation, since we measured no performance gain in using the equivalent UPC++ collective. The template- and library-based implementation strategy of UPC++ allows it to interoperate very well with MPI and other programming models such as OpenMP and CUDA. UPC++ also requires no changes in the computation part of the miniGMG code. Our experience indicates that for many legacy applications, the developers can reuse the majority of the existing code and only rewrite the parts that match the PGAS model or that can otherwise benefit from UPC++ features.

5.2 Bulk Version

As a first step, we implemented a version of miniGMG in UPC++ that follows the same communication structure as the MPI code, but with one-sided rather than two-sided data transfer. We refer to this as the *bulk* version of the code. All data that need to be transferred from one rank to a neighboring rank are manually packed into a single message. Communication buffers are allocated in the shared address

space, allowing the sender to perform one-sided puts using the non-blocking `async_copy` function template. Upon completion of the transfer and synchronization, incoming data are manually unpacked into the destination ghost zones.

5.3 Fine-Grained Version

Compared to the bulk-message implementation, the fine-grained version of miniGMG in UPC++ makes full use of the global address-space abstraction. All box data are stored in the shared address space. When ghost-zone data are needed, a UPC++ rank can simply locate the neighboring box ID and use it to reference the data directly without worrying whether the data are local or remote. The following code is a generalization of the operations that copy the ghost data from one box to another:

```
for (int k = 0; k < dim_k; k++)
  for (int j = 0; j < dim_j; j++) {
    int roff = recv_i + (j+recv_j)*rpencil +
      (k+recv_k)*rplane;
    int soff = send_i + (j+send_j)*spencil +
      (k+send_k)*splane;
    async_copy(sbuf+soff, rbuf+roff, dim_i);
  }
```

For each contiguous piece of data, the code computes the offsets into the send and receive boxes before making a call to `async_copy`. The resulting code is equivalent to the shared-memory version, with `async_copy` taking the place of `memcpy`. Non-contiguous data is just a special case with *dim_i* equal to 1. Figure 3(b) illustrates this communication algorithm. The UPC++ runtime takes care of accessing both remote and local data, and the tedious and error-prone packing and unpacking steps are no longer needed. The data layout of a 3-D box has only one contiguous dimension in memory (dimension i in the code above), so message sizes in the fine-grained version of miniGMG can vary from one double-precision floating-point number to the whole box size in dimension i.

The ease in both reasoning about and implementing an algorithm with fine-grained communication comes at a cost of performance. Its communication is dominated by 8-byte messages, so its performance is more sensitive to message rate and network latency than bandwidth. However, the fine-grained communication paradigm enables faster application development with less code, and performance can always be improved through incremental optimizations. Future innovations in network hardware and runtime systems may also help close the gap between fine-grained and bulk communication.

5.4 Array Version

We implemented a third version of miniGMG to take advantage of the multidimensional array support in UPC++. Each box is represented as a multidimensional array, with a domain corresponding to the box's position in the global index space. In order to minimize the changes required, the code creates array descriptors over the memory that is already allocated for each box and only uses these descriptors for the ghost-zone exchange.

In the setup phase of the algorithm, for each box in the ghost-zone exchange, views are created of the send and receive arrays restricted to the subset of the data involved in the exchange. After a simple circular domain shift to handle the boundaries, the code to create these views is as follows:

```
rectdomain<3> ghost_domain = dst.domain() *
  src.domain().shrink(ghost_zone_depth);
send_arrays[PT(level, id, dir, i, j, k)] =
  src.constrict(ghost_domain);
recv_arrays[PT(level, id, dir, i, j, k)] =
  dst.constrict(ghost_domain);
```

The first statement computes the ghost domain as the intersection of the destination domain and the interior of the source domain. The latter two statements construct views of the two boxes restricted to the ghost domain, storing them in six-dimensional arrays according to the level number in the V-cycle, grid ID, neighbor direction, and box number in each dimension. Then in the ghost-zone exchange itself, a single call is required to copy each ghost zone:

```
ndarray<double, 3, global> recv =
  recv_arrays[PT(level, id, dir, i, j, k)];
recv.async_copy(send_arrays[PT(level, id, dir,
                               i, j, k)]);
```

From the user point of view, an entire ghost zone is transferred in each copy, as illustrated in Figure 3(c). No packing or unpacking is required in the user code, and the resulting code is even simpler than the fine-grained version.

6. EXPERIMENTAL SETUP

In this section, we describe the computing platforms we use to evaluate the different versions of miniGMG, as well as the configurations we use to run our experiments.

6.1 Systems

The first of our two experimental platforms is the Cray XC30 (*Edison*) system located at NERSC [11]. It is comprised of 5,576 compute nodes, each of which contains two 12-core Intel Ivy Bridge processors running at 2.4 GHz, and is connected by Cray's Aries (Dragonfly) network. Each core includes private 32KB L1 and 256KB L2 caches, and each processor includes a shared 30MB L3 cache. Nominal STREAM [27] bandwidth to DRAM exceeds 40 GB/s per processor. In all experiments, we disable HyperThreading and use only eight cores per processor, which often maximizes performance on this machine. We compile all code with the default (`icc`) backend compiler.

The second platform is the IBM Blue Gene/Q (*Mira*) located at Argonne National Laboratory [24]. Mira is composed of 49,152 compute nodes, each of which includes 16 multithreaded PowerPC A2 cores for user code and one additional core for operating system services. Each core runs at 1.6 GHz, supports four threads, and can simultaneously issue instructions from two different threads. Unlike Ivy Bridge, at least 32 threads per node are required to efficiently utilize the A2 processor; we run with the full 64 threads supported by each node. The cache hierarchy is very different from the Ivy Bridge processor in that each core has only a private 16KB L1 cache, while all cores on a node share a 32MB L2 cache. The STREAM bandwidth is approximately 26GB/s, so we expect an XC30 socket running miniGMG to significantly outperform a Blue Gene/Q socket in computation time. Nodes are interconnected using IBM's high-performance proprietary network in a 5D

torus, and the observed performance for collectives such as `MPI_Barrier` and `MPI_Allreduce` is substantially superior to the XC30. We compile all code with the `mpixlc_r` compiler.

6.2 Parallelization and Scaling Experiments

In all cases, we run weak-scaling experiments for miniGMG with a fixed problem size of 128^3 cells per socket. In order to differentiate the injection rates of MPI and GASNet, we explore two parallelization configurations: one process per socket and eight processes per socket, with an MPI or UPC++ rank mapped to each process. When using only a single process per socket, each process owns a 128^3 box, but with eight processes per socket, each process has one 64^3 box. This ensures that the work, off-node communication, and solve time are roughly the same, but the latter is capable of higher injection rates. Within each process, we use OpenMP to parallelize the operations on a box. On the XC30, we run either one 8-thread process per socket or eight single-threaded processes. Similarly, we run either one 64-thread process per socket or eight 8-threaded processes per socket on the Blue Gene/Q. We use only one box per process to ensure that "communication" time is not skewed by intraprocess data copies. In miniGMG, the solver is run four times — we consider the first three to be warmups and only report performance for the last solve.

7. EXPERIMENTAL RESULTS

In this section, we quantify the performance differences of the four miniGMG variants on our two evaluation machines and compare observed performance with the ideal behavior of the multigrid algorithm.

7.1 Communication Characterization

In miniGMG, data decomposition is extremely regimented and preserved across all levels in the V-cycle. A simple 7-point variable-coefficient operator (stencil) necessitates communication with six neighbors, and at each level, a process communicates with the same six neighbors. In the multigrid algorithm, the dimension of each subdomain is reduced by a factor of two at each level, so the total volume of interprocess communication decreases by a factor of four at each level. Although the volume of communication is deterministic, the choice of communication paradigm (e.g. bulk vs. fine-grained) dictates the actual size of each message. Figure 5 presents a histogram of the frequency of message sizes as seen by GASNet for each communication paradigm for both one process per socket and eight processes per socket. As expected, the MPI and bulk UPC++ implementations send large message sizes ranging from 128 bytes to 128KB in factors of four. At each level of the ten V-cycles, we expect the implementation to send nine messages to each of six neighbors. However, the coarse-grid BiCGStab solver can require significantly fewer messages depending on the convergence rate, so fewer 128-byte messages are sent.

In comparison, the fine-grained implementation's message characteristics are more nuanced. As fine-grained communication of contiguous data is naturally aggregated, we expect two different scenarios — communication of a pencil in a $i-j$ or $i-k$ plane of a box or communication of an element in a $j-k$ plane of a box. The latter results in a flood of 8-byte (one `double` value) messages, while the former results in message sizes equal to the subdomain dimension — 4, 8, 16, 32, 64, or 128 elements, with eight bytes per element. The number of messages increases with box size, since a 4^2 plane requires four 32-byte (4-double) messages while a 128^2 plane requires 128 1KB (128-double) messages.

Importantly, when moving from one 128^3 process per socket to eight 64^3 processes per socket, each process sends only a quarter of the 8-byte messages. Given 98% of the communication consists of these small 8-byte messages, the overhead of fine-grained messaging must be minimal in order to ensure that performance with one process per socket is comparable to eight processes per socket in the fine-grained case.

Message distribution in the array version generally matches the MPI/bulk implementations until messages become very small at which point the differences in protocol become apparent. Specifically, the behavior in miniGMG of the array implementation can be categorized into three modes:

1. If both source and destination are contiguous and have the same layout, it performs a one-sided put to directly transfer the data.
2. If both source and destination are non-contiguous but the amount of data (plus array metadata) fits in a medium active message (AM) [2], then a single medium AM is initiated.
3. If both source and destination are non-contiguous and the data plus metadata do not fit into a medium AM, then the following procedure occurs:
   (a) A short AM is sent to the destination to allocate a temporary buffer.
   (b) A one-sided put transfers the array data into the remote buffer.
   (c) A medium AM transfers the array metadata. The temporary buffer is deallocated after unpacking is completed in the AM handler.

Therefore, for messages larger than the AM medium threshold (960 bytes on the Cray machine and 4096 bytes on the IBM machine in our experiments), the array version generally behaves as the MPI and bulk versions, with some additional small messages for metadata (at 144 bytes in Figure 5). For messages smaller than the threshold, the array version also has the same numbers as the bulk version, but the message size is shifted to also include the metadata.

7.2 Performance Comparison

Figure 6 presents miniGMG time to solution as a function of platform and parallelization at an overall concurrency of 512 sockets. We observe that the bulk UPC++ implementation consistently delivers performance comparable to the MPI implementation, which comes as no surprise as it simply trades point-to-point synchronization for small-scale barriers. On the other hand, the fine-grained implementation significantly underperforms the MPI code, and the array version is also generally slower. Though the time breakdown shows that across all configurations, the time spent in local computation remains the same, the time spent waiting on synchronization (interprocess barriers, local `async_wait`, processing incoming AM's) and the time actually spent sending data (puts) become an impediment to performance for the fine-grained and array implementations.

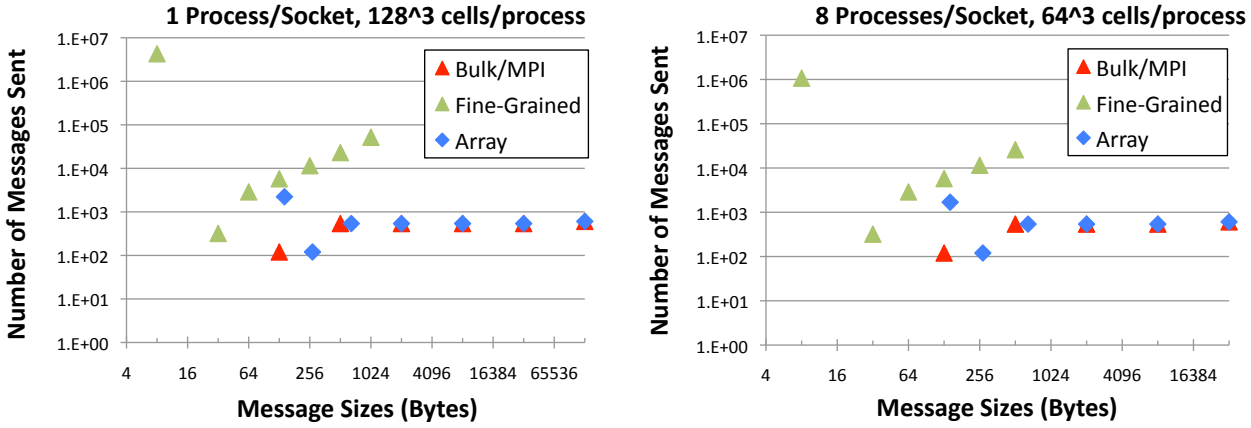

Figure 5: Histograms of the frequency of message sizes sent per process in miniGMG across the communication styles for eight sockets. For fine-grained communication, when eight processes per socket are used, each process sends only one-quarter (three million fewer) of the 8-byte messages sent by each process when using one process per socket.

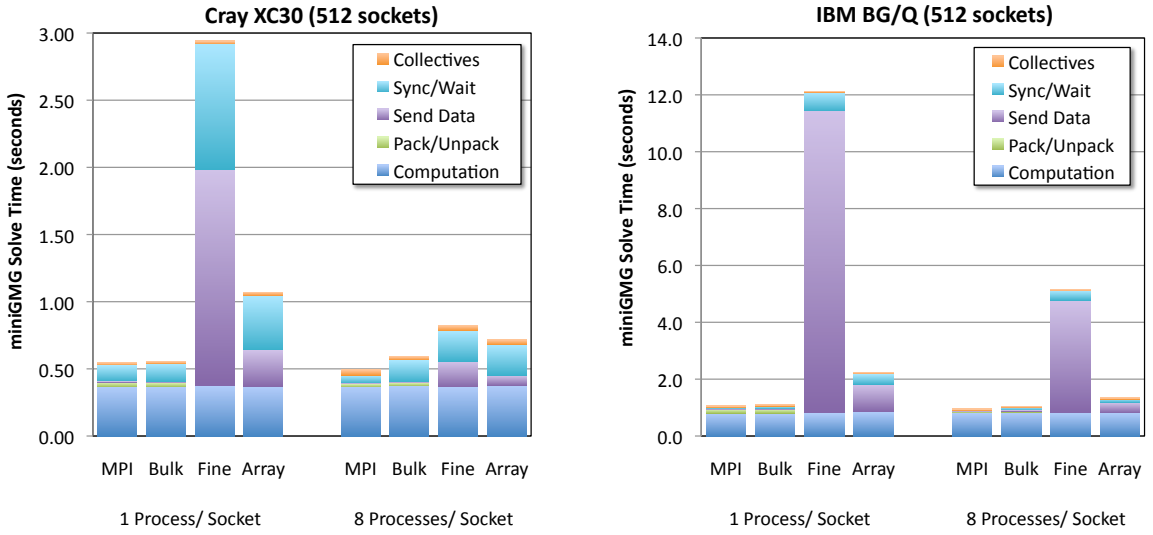

Figure 6: miniGMG solver times for the Cray XC30 (left) and the IBM BGQ (right). For fine-grained communication, the performance benefit of running with eight processes per socket and sending 4× fewer 8-byte messages per process outweighs the penalty of doubling the total number of messages sent per socket.

When using fine-grained communication, moving from one process per socket to eight processes per socket doubles the number of 8-byte messages sent per socket, but it reduces the number of messages sent per process by a factor of four. If the overhead for sending small messages is high, then the benefit of reducing the number of messages per process is high. Conversely, if the per-message overhead is low, then doubling the total number of messages per socket can impede performance. It is quite possible that the XC30 is in the former scenario while the BGQ is somewhere in between. The array implementation also suffers from high overhead for sending data when there is only a single process per socket, but improves when there are eight processes. Ultimately, as it can aggregate messages, it incurs lower overhead for sending data than the fine-grained implementation. Nevertheless, the array implementation is about 15-80% slower than the MPI implementation on the Cray XC platform and up to 40% slower on the IBM BG/Q.

The performance loss in the array version can be attributed to two factors. The first is the additional messaging required when the data are not contiguous and the volume is larger than the AM medium limit. Internal experiments have demonstrated that performance can be improved by 20% by increasing the AM medium limit to 64KB. Unfortunately, this limit cannot be changed by environment variables and requires a recompile of GASNet. The second factor is that the array version does not use multithreading to parallelize packing and unpacking, in order to avoid the additional overheads in the multithreaded version of GASNet. As a result, moving from one process per socket to eight processes improves performance considerably. Efficient thread-

ing of packing and unpacking in the array code is a topic of current and future research.

Overall, on a per-socket basis, the XC30 delivers about twice the performance as the BGQ despite having roughly 50% more streaming bandwidth from DRAM. Ultimately, the Ivy Bridge processor makes better use of the memory interface while the BGQ system makes better use of its network.

7.3 Ideal Multigrid Behavior

For optimal multigrid efficiency, the time spent in each level must decrease exponentially. Any inefficiencies or unexpectedly high overheads can impede performance. Figure 7 shows the time spent in computation and the ghost-zone exchange at each level of the V-cycle in miniGMG, using 4096 processes with eight processes per socket. The amount of data communicated to each neighbor on each level is the square of the grid size (e.g. 64^2 doubles on level 0). As expected, there is an eightfold reduction in compute time at each level on each platform. On the other hand, there are substantial differences in overhead and effective bandwidth across implementations and platforms that result in communication dominating the run time for all levels coarser than 32^3. The ideal factor of four reduction in communication only occurs on the first level or two for the MPI implementation, after which time approaches an asymptotic overhead limit. Interestingly, the one-sided bulk UPC++ implementation of the MPI algorithm consistently underperforms the MPI implementation even for large messages — likely an artifact of the barriers required for each exchange. Since the fine-grained implementation sends many small messages, each of which incurs some overhead, the effective bandwidth is substantially degraded across all levels as seen on the BGQ.

Ultimately, the MPI time per exchange approaches an asymptotic limit — the overhead or α in an $\alpha-\beta$ model. However, despite the overhead of barriers in each exchange, the performance of the UPC++ implementations continue to improve. It is likely that if the coarse-grained global synchronizations were replaced with fine-grained point-to-point synchronization, the UPC++ implementations would deliver better performance on the coarser levels. In essence, this would be nothing more than a GASNet-based implementation of MPI.

Looking forward to exascale-class processors, we expect the local compute time to be dramatically reduced. If such increases in raw compute performance are not accompanied by both increases in bandwidth and substantial reductions in overheads, the resulting performance will fall far below the potential for exascale machines.

7.4 Scalability

miniGMG is designed to proxy the multigrid solvers in weak-scaled applications. Figure 8 presents weak-scaled miniGMG time to solution as a function of scale and communication implementation when moving from one socket to 4096 for both the Cray XC30 (top) and the IBM BGQ (bottom). Ideally, the code should provide a constant time to solution in weak scaling.

With one process per socket, as shown in the left of Figure 8, the performance of the fine-grained and array implementations are impeded by the overhead associated with millions of 8-byte puts, global synchronizations (barriers), and `async_wait` (which includes the processing of incoming AM's). However, as previously discussed, running multiple processes per socket mitigates much of the effect. In the resultant regime of eight processes per socket, as shown in the right of Figure 8, performance for the XC30 is quite similar across implementations up to 4096 processes (512 sockets). Beyond this point, the performance of the PGAS implementations become highly sensitive to the performance of the synchronization mechanism (barriers) and scalability is diminished. Conversely, the extremely fast barriers on the BGQ ensure the bulk and array implementations deliver performance comparable to the MPI implementation at all scales. Unfortunately, the fine-grained implementation consistently performs worse on the BGQ.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we studied the performance and productivity tradeoffs of different communication paradigms supported by PGAS languages in the context of miniGMG, a geometric multigrid solver. As communication in miniGMG is primarily point-to-point with a static set of neighbors and predictable message sizes, it was no surprise that the bulk implementation delivered the best performance of the PGAS implementations and performance comparable to MPI's point-to-point implementation. Unfortunately, like MPI, it required manual, labor-intensive, and error-prone packing and unpacking of messages, but unlike MPI, it required the addition of barriers for synchronization.

Conversely, a fine-grained messaging style offered simple and fast code development, but due to the flood of small messages, delivered by far the lowest performance when there was a single process per socket. Although increasing the number of processes per socket increases the total number of messages sent per socket, it decreases the number of messages sent per process, providing better overall performance.

We improved on the performance of fine-grained communication using higher-level multidimensional array constructs to automatically aggregate communication. The array-based implementation delivered performance closer to the bulk implementation while actually providing greater productivity than the fine-grained version, demonstrating that high-level programming interfaces can deliver on both the productivity and performance promises of PGAS languages.

Our future work will focus on a few areas. First, we observed that the fine-grained implementation's performance is significantly worse on the IBM machine than the Cray machine. We will investigate whether this is a GASNet issue or something inherent in the architecture. Second, for applications like miniGMG with simple communication patterns, global barriers provide more synchronization than is necessary. Instead, we plan to add fine-grained synchronization features such as synchronization variables, signaling puts, and phasers. In addition, automating the aggregation of communication is as important as efficient synchronization mechanisms. We are working on modifying the communication software to detect communication patterns at runtime and coalesce fine-grained messages dynamically. We are also in the process of adding a new API for array communica-

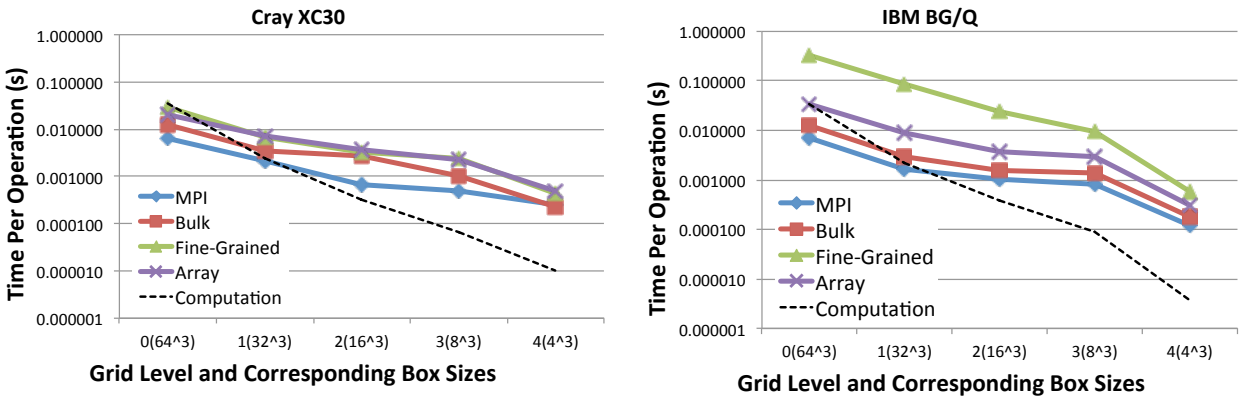

Figure 7: Individual ghost-zone exchange time (only) for each level of the V-cycle for our four communication implementations running on the Cray XC30 (left) and the IBM BGQ (right). In all cases, the results are for 4096 processes with eight processes per socket. For reference, we also include the time spent in computation at each level to highlight the transition from compute-limited to communication-limited regimes.

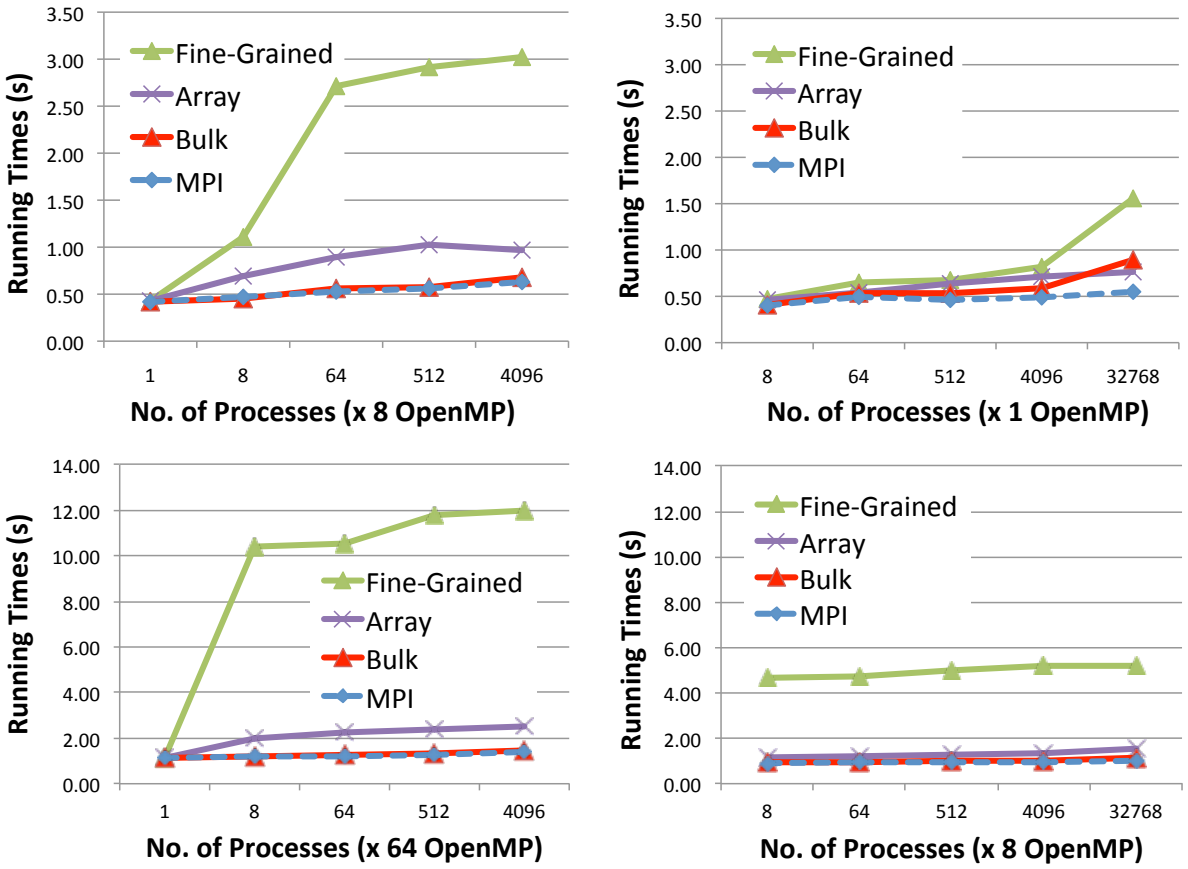

Figure 8: With sufficient concurrency on a socket, weak-scaled miniGMG time to solution for the Cray XC30 (top) and the IBM BGQ (bottom) using the UPC++ implementations is comparable to the MPI implementation. The figures on the left show performance for one process per socket while those on the right show eight processes per socket.

tion that will enable the runtime to aggregate multiple array copies, preallocate remote buffers, and minimize the amount of metadata that needs to be transferred. Finally, we will investigate applications that depart from miniGMG's predictable, static, and limited-radix communication pattern. Applications that dynamically determine which neighbors

to communicate with or how much data needs to be exchanged may better highlight the potential of PGAS and the UPC++ technologies described in this paper.

Acknowledgments

Authors from Lawrence Berkeley National Laboratory were supported by DOE's Advanced Scientific Computing Research under contract DE-AC02-05CH11231. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory and the National Energy Research Scientific Computing Facility (NERSC) at Lawrence Berkeley National Laboratory, which are supported by the Office of Science of the U.S. Department of Energy under contracts DE-AC02-06CH11357 and DE-AC02-05CH11231, respectively.

9. REFERENCES

[1] Aggregate Remote Memory Copy Interface. `http://hpc.pnl.gov/armci/`.

[2] Bonachea, D. GASNet specification. Tech. Rep. CSD-02-1207, University of California, Berkeley, October 2002.

[3] Bonachea, D. Proposal for extending the UPC memory copy library functions and supporting extensions to GASNet. Tech. Rep. LBNL-56495, Lawrence Berkeley National Lab, October 2004.

[4] BoxLib website. `https://ccse.lbl.gov/BoxLib`.

[5] The Berkeley UPC Compiler. `http://upc.lbl.gov`.

[6] Chamberlain, B., Callahan, D., and Zima, H. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications 21*, 3 (2007), 291–312.

[7] Chamberlain, B. L., Choi, S.-E., Deitz, S. J., and Snyder, L. The high-level parallel language ZPL improves productivity and performance. In *In Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing* (2004).

[8] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (2005), OOPSLA '05.

[9] Chombo website. `http://seesar.lbl.gov/ANAG/software.html`.

[10] Datta, K., Bonachea, D., and Yelick, K. Titanium performance and potential: an NPB experimental study. In *Proc. of Languages and Compilers for Parallel Computing* (2005).

[11] Edison Cray XC30. `http://www.nersc.gov/systems/edison-cray-xc30/`.

[12] El-Ghazawi, T., and Cantonnet, F. UPC performance and potential: A NPB experimentental study. In *Supercomputing2002 (SC2002)* (November 2002).

[13] Global Arrays Toolkit. `http://www.emsl.pnl.gov/docs/global/`.

[14] Garland, M., Kudlur, M., and Zheng, Y. Designing a unified programming model for heterogeneous machines. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), SC '12.

[15] GASNet home page. `http://gasnet.cs.berkeley.edu/`.

[16] Gerstenberger, R., Besta, M., and Hoefler, T. Enabling highly-scalable remote memory access programming with mpi-3 one sided. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2013), SC '13, ACM, pp. 53:1–53:12.

[17] Ghosh, P., R.Hammond, J., Ghosh, S., and Chapman, B. Performance analysis of the NWChem TCE for different communication patterns. In *The 4th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS13)* (September 2013).

[18] GPI website. `http://www.gpi-site.com/gpi2/benchmarks/`.

[19] Grünewald, D. BQCD with GPI: A case study. In *HPCS* (2012), W. W. Smari and V. Zeljkovic, Eds., IEEE, pp. 388–394.

[20] Kamil, A., Zheng, Y., and Yelick, K. A local-view array library for partitioned global address space C++ programs. In *ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (2014).

[21] Machado, R., Lojewski, C., Abreu, S., and Pfreundt, F.-J. Unbalanced tree search on a manycore system using the GPI programming model. *Computer Science - R&D 26*, 3-4 (2011), 229–236.

[22] 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).

[23] miniGMG compact benchmark. `http://crd.lbl.gov/groups-depts/ftg/projects/current-projects/xtune/miniGMG`.

[24] Mira IBM Blue Gene/Q. `http://www.alcf.anl.gov/user-guides/mira-cetus-vesta`.

[25] Shan, H., Austin, B., Wright, N. J., Strohmaier, E., Shalf, J., and Yelick, K. Accelerating applications at scale using one-sided communication. In *The 6th International Conference on Partitioned Global Address Space Programming Models* (October 2012).

[26] Simmendinger, C., Jägersküpper, J., Machado, R., and Lojewski, C. A PGAS-based implementation for the unstructured CFD solver TAU. In *Proceedings of the 5th Conference on Partitioned Global Address Space Programming Models, PGAS '11* (2011).

[27] STREAM benchmark. `http://www.cs.virginia.edu/stream/ref.html`.

[28] Tan, A. T., Falcou, J., Etiemble, D., and Kaiser, H. Automatic Task-based Code Generation for High Performance Domain Specific Embedded Language. In *7th International Symposium on High-Level Parallel Programming and Applications (HLPP 2014)* (2014).

[29] Wen, T., and Colella, P. Adaptive mesh

refinement in Titanium. In *The 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS05)* (April 2005).

[30] Williams, S., Kalamkar, D. D., Singh, A., Deshpande, A. M., Van Straalen, B., Smelyanskiy, M., Almgren, A., Dubey, P., Shalf, J., and Oliker, L. Implementation and optimization of miniGMG - a compact geometric multigrid benchmark. Tech. Rep. LBNL 6676E, Lawrence Berkeley National Laboratory, December 2012.

[31] Williams, S., Kalamkar, D. D., Singh, A., Deshpande, A. M., Van Straalen, B., Smelyanskiy, M., Almgren, A., Dubey, P., Shalf, J., and Oliker, L. Optimization of geometric multigrid for emerging multi- and manycore processors. In *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), SC '12, IEEE Computer Society Press.

[32] Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., and Aiken, A. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience 10*, 11-13 (September-November 1998).

[33] 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).

[34] Zheng, Y., Kamil, A., Driscoll, M. B., Shan, H., and Yelick, K. UPC++: A PGAS extension for C++. In *28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2014).