Implementing High-Performance Geometric Multigrid Solver With Naturally Grained Messages

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

***Abstract*—Structured-grid linear solvers often require manual packing and unpacking of communication data to achieve high performance. Orchestrating this process efficiently is challenging, labor-intensive, and potentially error-prone. In this paper, we explore an alternative approach that communicates the data with naturally grained message sizes without manual packing and unpacking. This approach is the distributed analogue of shared-memory programming, taking advantage of the global address space in PGAS languages to provide substantial programming ease. However, its performance may suffer from the large number of small messages. We investigate the runtime support required in the UPC++ library for this naturally grained version to close the performance gap between the two approaches and attain comparable performance at scale using the High-Performance Geometric Multgrid (HPGMG-FV) benchmark as a driver.**

***Keywords*-HPGMG; Naturally-Grained Messages; VIS functions; Group Synchronization; PGAS; UPC++;**

I. Introduction

High-Performance Geometric Multigrid (HPGMG-FV) is a benchmark designed to proxy linear solvers based on finite-volume geometric multigrid [10]. As a proxy application, it has been used by many companies and DOE labs to conduct computer science research. It implements the full multigrid F-cycle with a fully parallel, distributed V-Cycle. Its communication is dominated by ghost exchanges at each grid level and restriction and interpolation operations across levels. The primary data structure is a hierarchy of three-dimensional arrays representing grids on the physical domain. The computation involves stencil operations that are applied to points on the grids, sometimes one grid at a time and sometimes using a grid at one level of refinement to update another; the interprocessor communication therefore involves updating ghost regions on the phases of subdomains of these grids. Given the performance characteristics of current machines and the discontiguous nature of the data on some of the faces of these grids, the ghost-region data must be packed at the source process and correspondingly unpacked at the destination process to ensure high performance and scalability. The multigrid computation has several different types of operators that may involve different packing patterns, as one must deal with unions of subdomains, deep ghost-zone exchanges, and communication with edge and corner neighbors. The manual packing and unpacking process is therefore very complex and error-prone.

A different approach is to implement the algorithm in a more natural way by expressing communication at the data granularity of the algorithm (sequences of contiguous double-precision words) without manual message aggregation. The PGAS programming model provides a suitable environment for us to evaluate this approach. The global address space and efficient one-sided communication enable communication to be expressed with simple copies from one data structure to another on a remote process, analogous to shared-memory programming but using puts or gets rather than calls to `memcpy`. We refer to this as *naturally grained* communication, since the messages match the granularity of the memory layout in the data structure. For example, copying a face of a multidimensional array may be accomplished with a few large messages if it is in the unit-stride direction or numerous small messages consisting of individual double-precision words if it is in the maximally strided direction.

As described above, codes developed with natural message sizes often generate a large number of small messages. Unfortunately, current HPC systems often favor large messages, so flooding the network with millions of small messages may significantly degrade application performance. To address this issue, we investigate what features the runtime system can provide to enable a naturally expressed implementation to be competitive with a highly tuned but more complex version. We use the open-source UPC++ [20] library as our framework for this study, examining features such as 1) exploiting hardware cache-coherent memory systems inside a node to avoid message overhead, 2) library support for communicating non-contiguous data, and 3) group synchronization. With these three features, our naturally-grained implementation attains performance comparable to the highly tuned bulk-communication version at up to 32K processes on the Cray XC30 platform.

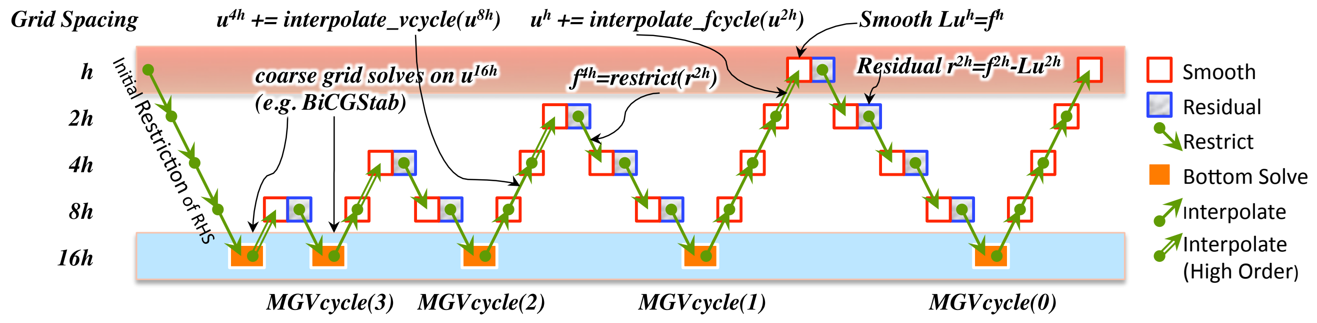

Grid Spacing
h
2h
4h
8h
16h
Initial Restriction of RHS
u^{4h} += interpolate_vcycle(u^{8h})
u^h += interpolate_fcycle(u^{2h})
Smooth $Lu^h=f^h$
coarse grid solves on u^{16h}
(e.g. BiCGStab)
f^{4h}=restrict(r^{2h})
Residual $r^{2h}=f^{2h}-Lu^{2h}$
MGVcycle(3)
MGVcycle(2)
MGVcycle(1)
MGVcycle(0)
Smooth
Residual
Restrict
Bottom Solve
Interpolate
Interpolate (High Order)

Figure 1. The full geometric multigrid cycle.

II. Related Work

To improve the performance of non-contiguous data transfers, Chen et al. studied coalescing fine-grained messages at compile time for UPC applications [4]. However, their work required compiler support to find optimization candidates, which is very difficult for complex C/C++ codes. Willcock et al. implemented message coalescing for AM++ based on message types [17]. Their work requires certain implementation parameters, such as the buffer length to hold the messages, to be set properly to improve performance. On the other hand, in our work, the runtime provides explicit support for non-contiguous messages, which maximizes performance at the cost of some programming effort from the application writer.

Regarding synchronization, Belli and Hoefler [1] proposed a notified-access mechanism to reduce synchronization overhead for one-sided messages in producer-consumer algorithms. D. Bonachea et al. [3] proposed a signaling-put mechanism to couple data transfer and synchronization for point-to-point communication. In our approach, we propose an asymmetric group-synchronization mechanism to avoid global barriers in nearest-neighbor communication. While it may potentially provide less performance than directly using point-to-point synchronization, it provides a simple interface that is similar to a barrier.

In our prior work [14], we developed a naturally grained version of the miniGMG benchmark. However, we did not attempt to optimize performance of the naive implementation. Furthermore, HPGMG is much more complicated due to its full multigrid cycle, level-by-level domain decomposition, and redistribution of the workload to a subset of the processes when there is not enough work at coarser levels in the multigrid cycle.

A number of performance studies have also been done using the Global address space Programming Interface, GPI-2 [15], [12], [9], [8], which is a library-based PGAS model and does not involve compiler optimizations specific to parallel computation or data movement. Our focus is somewhat different, namely to examine runtime support mechanisms that enable a naturally grained implementation to compete with highly tuned bulk-copy versions.

III. HPGMG-FV

In this paper, we use a UPC++ port of HPGMG-FV [10] as a driver for our PGAS communication optimizations. HPGMG-FV solves the variable-coefficient PDE $b\nabla\cdot\beta\nabla u = f$ using the finite-volume method on a structured grid. As such, it is superficially similar to miniGMG [13], [18] which we used as a basis for our previous work [14]. However, unlike miniGMG, HPGMG uses Full Multigrid (FMG) and a distributed V-Cycle. As shown in Figure 1, first, the right-hand side of the finest grid (e.g. 256^3 on $[0,1]^3$) is restricted to the coarsest grid (e.g. 1^3 on $[0,1]^3$) and an accurate solution is obtained at this level. The solution on the coarsest grid is interpolated to provide a good initial vector on the next finer level (e.g. interpolate the 1^3 solution to provide an initial guess for the 2^3 grid). A V-cycle starting with this new finer grid improves the solution (e.g. a V-cycle from 2^3 to 1^3 and back). Then, interpolation predicts the solution for the next finer grid (4^3) and a following deeper V-cycle makes it better, and so on. Ultimately, a good initial guess is interpolated to the 256^3 grid and one final V-cycle is performed. Within the recursive V-Cycle, a series of pre- and post-smooths are performed and the restriction of the residual is used to provide the right-hand side for the next coarser grid.

A. Domain Decomposition

The grids with different spacings (e.g. 256^3 vs. 128^3 on $[0,1]^3$) form hierarchical grids or levels. Each level is represented by a cubical domain and is partitioned into cubical boxes in i, j, k directions. The box is the basic partition unit and distributed among the processes. Boxes can be either smaller, the same size, or larger on coarser levels, depending on the size of the process subset assigned to work on the level, but the total number of cells is always $8\times$ less. Each box maintains a list of the vectors u, r, f, etc. shown in

Figure 1. In addition, the box size is augmented with ghost zones, which replicate data from neighboring boxes or are computed based on boundary conditions when non-periodic boundary conditions are applied. Therefore, only the data in direction i are contiguous. Figure 2 illustrates the box layout for the two-dimensional case. The shaded areas for box 0 represent the ghost zones. At most, the ghost-zone data may come from 8 different neighboring boxes to fill in the 8 ghost regions. In three dimensions, the neighboring boxes can be as many as 26, indicating that one box may have to communicate with 26 different neighbors to fill in its ghost zones.

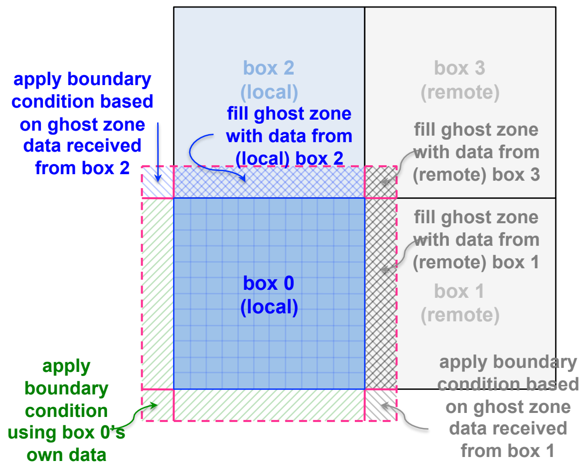

apply boundary condition based on ghost zone data received from box 2
box 2 (local)
fill ghost zone with data from (local) box 2
box 3 (remote)
fill ghost zone with data from (remote) box 3
fill ghost zone with data from (remote) box 1
box 0 (local)
box 1 (remote)
apply boundary condition using box 0's own data
apply boundary condition based on ghost zone data received from box 1

Figure 2. 2D illustration of the ghost zone exchange and boundary condition in the 3D HPGMG-FV.

One major difference from miniGMG lies in the domain decomposition. In miniGMG, the domain decomposition is preserved across restrictions, so no communication is needed for restriction and interpolation. However, in HPGMG-FV, the domain decomposition is on a level-by-level basis: when there is insufficient work, computation is redistributed onto a subset of the processes. At the extreme, only one process needs to work on the coarsest level. This changes the distribution of message sizes to favor smaller messages. Moreover, communication becomes necessary when restricting grids to coarser levels or interpolating the solutions to finer ones, which inflates the message count at the agglomeration threshold.

B. *MPI Communication Orchestration*

There are three major communication operations in HPGMG-FV. The first consists of the ghost exchanges inside the smooth and residual functions, which occur at the same grid level. The other two occur across levels inside the restriction and interpolation operations, respectively. In this section, we describe how these communications are setup and executed in the highly tuned MPI implementation.

The execution of each communication operation is divided into four steps and highlighted in Figure 3(a).

1) Packing: Post `MPI_Irecv` operations. Pack the data into a series of 1D buffers based on precomputed information, with each buffer targeting one neighbor process.
2) Sending: Initiate `MPI_Isend` operations, with one message for each neighboring process.
3) Receiving: Call `MPI_Waitall` to wait for incoming data.
4) Unpacking: Unpack the 1D buffers based on precomputed information into the corresponding ghost zones of the local boxes.

These steps are common in many MPI applications that use structured grids. They ensure pairwise synchronization and aggregate data to amortize overhead, and each process only needs to send and receive one message for each of its neighbors. However, setting up data structures to control the packing and unpacking operations is challenging; each process needs to figure out exactly which surfaces of its local boxes need to be sent to each neighbor and how to pack them into the 1D buffers so that they get correctly unpacked at the destination. This procedure is not only complex but also error-prone, as one must deal with unions of subdomains, deep ghost-zone exchanges, and communication with edge and corner neighbors. It comprises about 20% of the total HPGMG-FV source code and is the most difficult part of the code to write.

The following is a simplified description of the setup process.

1) Prepare for packing: For each local box and every direction, find the neighboring boxes, compute the sending surfaces, and record the surface information and the corresponding offset in the 1D sending buffer into an auxiliary data structure. Sort the data structure according to process ranks and box coordinates.
2) Prepare for unpacking: For each local box and every direction, find the neighboring boxes, compute the receiving ghost zone, and record the zone information and the corresponding offset in the 1D receiving buffer into an auxiliary data structure. Sort the data structure according to process ranks and box coordinates.

The purpose of sorting is to guarantee that the data packed at the source will be unpacked to their corresponding destination boxes correctly. The resulting metadata are cached for fast replay during execution.

The communication operations in the restriction and interpolation phases are organized similarly to the ghost-zone exchanges. The main difference is that the information in the auxiliary data structures represents inter-level rather than intra-level neighbors.

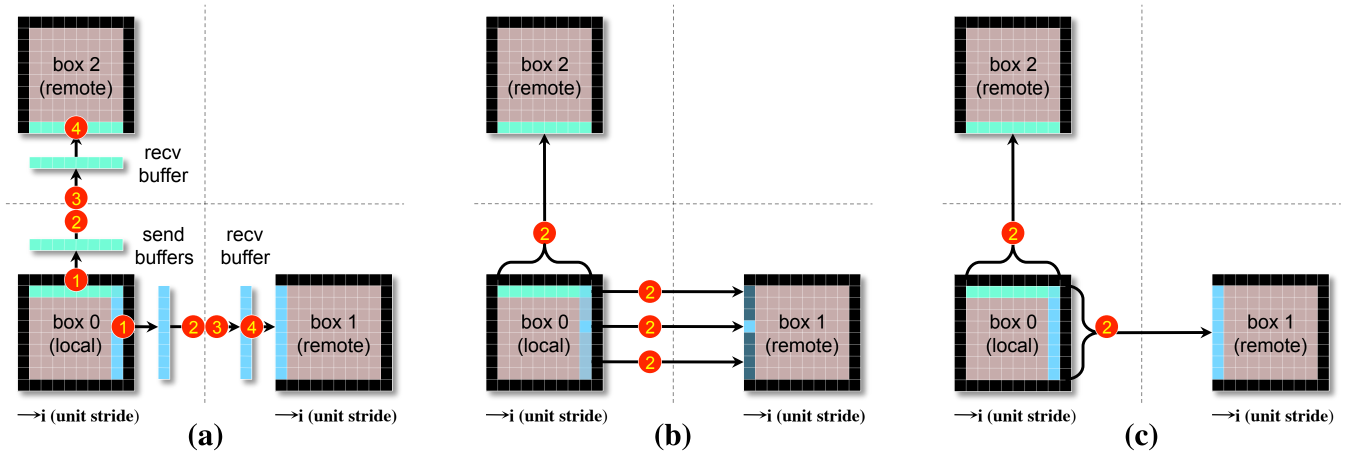
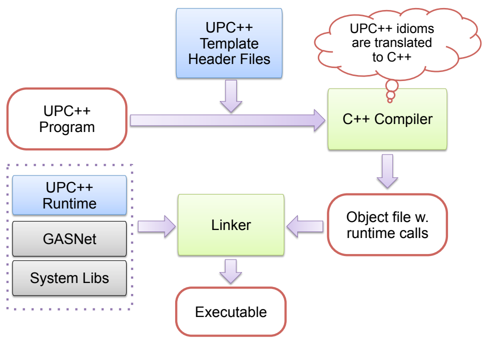


Figure 3. Communication styles for a) MPI, b) Naturally grained UPC++ (Baseline), and c) Natural version with VIS support.



UPC++ Template Header Files
UPC++ idioms are translated to C++
UPC++ Program
C++ Compiler
UPC++ Runtime
GASNet
System Libs
Linker
Object file w. runtime calls
Executable

Figure 4. UPC++ Implementation Software Stack

IV. UPC++

UPC++ [20] is a PGAS programming extension for C++. It adopts a template library-based implementation but includes PGAS language features such as those from UPC, Titanium [19] and Phalanx [6]. As shown in Figure 4, the UPC++ implementation is built on top of the GASNet communication library [7] and allows the application to call GASNet directly if needed. In this section, we describe the functions used in developing the UPC++ versions of HPGMG-FV, which include explicit data transfers, shared-memory support, VIS functions, and synchronization. These features enable us to write parallel programs with naturally grained messages and achieve performance comparable to the bulk version without manually packing and unpacking the communication data.

A. *Global Communication*

A UPC++ parallel job runs in SPMD fashion and each process has a unique rank, which can be used to identify the process for communication and remote task execution. In UPC++, a global address is represented by a `global_ptr<T>` type, which points to one or more shared objects of type `T`. Global data movement can be done by calling the non-blocking `async_copy` function:

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

Both the `src` and `dst` buffers are required to be contiguous, and `count` is the number of elements of type `T`. A call to `async_copy` initiates the data transfer and returns, allowing communication to be overlapped with computation or other communication. The user can query the completion status of a non-blocking copy using `async_try` or wait for its completion using `async_wait`. UPC++ also supports other communication operations such as blocking copies and implicit read and write through global pointers and references.

B. *Hardware-Supported Shared-Memory Access*

UPC++ also provides an address-localization (or *privatization* in UPC terminology) feature that takes advantage of hardware-supported cache-coherent shared memory if available. The user application can query if a global pointer can be addressed directly and if so, convert it to a local pointer as follows:

```
global_ptr<T> global_pointer;
if (global_pointer.is_local()) {
  T* local_pointer = (T *) global_pointer;
  ...
}
```

If the memory location referenced by `global_pointer` can be directly accessed by the calling process through a hardware-supported shared-memory mechanism (e.g., process-shared memory), the cast results in the corresponding local pointer to the memory location. This local pointer provides more efficient data access, since the runtime system can use direct load and store instructions, avoiding the overheads of address translation and going through the underlying message-communication layers.

C. Non-contiguous Remote Access

For non-contiguous data transfers, we developed a template function called `async_copy_vis` to support the strided data access.

```
template<typename T>
async_copy_vis(
     global_ptr<T> src, global_ptr<T> dst,
     size_t, *srcstrides, size_t *dststrides,
     size_t *count, size_t dims);
```

Here, `src` and —dst— are the starting addresses for the source and destination regions and `dststrides` and `srcstrides` are the stride lengths in bytes of all dimensions, assuming the data are linearly organized from dimension 0 to dimension `dims`. The `count` array contains the slice size of each dimension. For example, `count[0]` should be the number of bytes of contiguous data in the leading (rightmost) dimension.

The `async_copy_vis` function is built on top of the the Vector, Indexed and Strided (VIS) library in GASNet [2]. The GASNet VIS implementation packs non-contiguous data into contiguous buffers internally and then uses active messages to transfer the data and unpack at the destination.

The strategy used by the GASNet VIS implementation is very similar to the array-based code described in our previous work [14]. We chose to use GASNet VIS rather than higher-level arrays to minimize the changes required to the naturally grained implementation of HPGMG, and because we did not require the generality provided by UPC++ multidimensional arrays. However, a version of HPGMG that uses UPC++ multidimensional arrays is currently under development.

D. Group Synchronization

The primary communication pattern in stencil applications such as HPGMG-FV is nearest-neighbor communication, which requires synchronization between processes to signal when data have arrived and when they may be overwritten. The simplest way to implement this synchronization is to use global barriers. However, this tends to hurt performance at scale, since it incurs a significant amount of unnecessary idle time on some processes due to load imbalance or interference. In contrast, a point-to-point synchronization scheme that only involves the necessary processes can achieve much better performance and smooth workload variations over multiple iterations if there is no single hotspot, as in our case. As a result, we implemented a `sync_neighbor(neighbor_list)` function that only synchronizes with the group of processes enumerated by the `neighbor_list`. Unlike team barriers, the `neighbor_list` is asymmetric across ranks, so that only one call is required from each rank, whereas team barriers would require one call per neighbor on each rank.

The current experimental implementation is based on point-to-point synchronization, with UPC++ shared arrays representing flag variables, and by spin-waiting until all the individual synchronizations have completed. The algorithm is as follows:

```
for (i = 0; i < number of neighbors; i++)
   {set flag on neighbor i};
int nreceived = 0;
while (nreceived < number of neighbors) {
  for (i = 0; i < number of neighbors; i++)
    if (check[i] == 1) {
      if (received flag from neighbor i)
         {check[i] = 0; nreceived++;}
  advance();
}
```

The actual implementation also includes the proper fences to ensure operations are properly ordered. The `advance` function in UPC++ is used to make progress on other tasks while waiting.

V. IMPLEMENTATION IN UPC++

A. UPC++ Bulk Version

Our initial UPC++ version of HPGMG-FV, which we refer to as the *bulk* version, follows the same strategy of manually packing and unpacking communication buffers as in the MPI code. Unlike the MPI code, however, the bulk UPC++ implementation allocates the communication buffers in the global address space and uses one-sided put operations to transfer data instead of two-sided sends and receives. Synchronization is implemented using a point-to-point mechanism similar to signaling put [3]. The bulk version delivers performance similar to the highly tuned MPI implementation; Figure 5 shows the best solver time for both MPI and the UPC++ bulk versions, and the corresponding data are listed in Table I.

Table I
BEST SOLVER TIME (SECONDS) ON CRAY XC30

Cores	8	64	512	4096	32768
MPI	0.351	0.373	0.387	0.388	0.409
UPC++	0.348	0.366	0.382	0.385	0.405

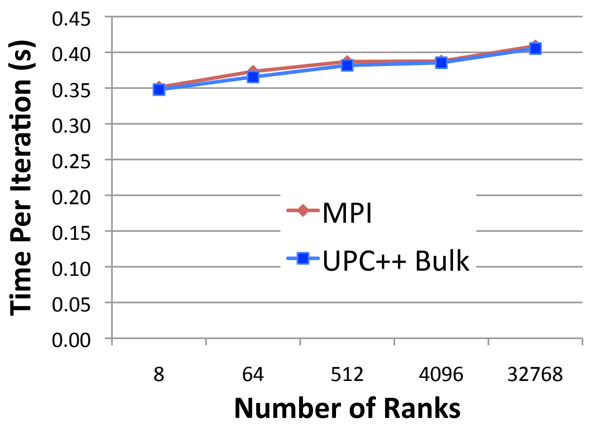


Figure 5. Performance of the MPI and UPC++ bulk implementations on the Cray XC30 platform.

B. UPC++ Natural Version

In order to avoid having to manually pack and unpack data, we rewrote the communication portion of the UPC++ bulk implementation to copy contiguous chunks of data. We refer to this as the *natural* or *naturally grained* version, since it performs copies at the natural granularity of the data layout, as would be done in shared memory. We made no changes to the computation part of the bulk code.

We allocate the HPGMG box data in the global address space, allowing remote ranks to directly access box data rather than going through send and receive buffers. When ghost data are needed, a rank can simply locate the neighboring box ID and use it to reference the data directly by calling `async_copy`, avoiding the tedious and error-prone packing and unpacking operations. This procedure is illustrated in Figure 3(b). Each contiguous piece of data is directly copied from source to destination in one message. Since the 3D box is linearized in memory, only the data in dimension i are contiguous. Furthermore, since the ghost zones are allocated together with box data, the message size can be no more than the size of the box in dimension i, while the minimum size is one double-precision value (8 bytes) in the case of the non-contiguous k dimension.

The following is the code used to copy a ghost zone directly from source to destination:

```
for (k = 0; k < dim_k; k++) {
  for (j = 0; j < dim_j; j++) {
    int recv_off = recv_i +
              (j + recv_j) * recv_pencil +
              (k + recv_k) * recv_plane;
    int send_off = send_i +
              (j + send_j) * send_pencil +
              (k + send_k) * send_plane;
    async_copy(send_buf + send_off,
                   recv_buf + recv_off, dim_i);
  }
}
```

The `dim_k, dim_j, dim_i` values are the ghost-zone sizes in dimensions k, j, and i, respectively. The `send_buf`, `send_i`, `send_j`, and `send_k` variables represent the starting address and offsets of the source region, while `recv_buf`, `recv_i`, `recv_j`, and `recv_k` are the starting address and offsets of the destination region. Finally, `send_pencil` and `send_plane` are the stride lengths of the source data in the j, and k dimensions while `recv_pencil` and `recv_plane` are the corresponding stride lengths of the destination region. To avoid redundant computation, we save the source and destination information for each ghost zone for reuse. Compared to the bulk implementation, we completely avoid the complicated communication setup.

1) Optimizations: The above baseline implementation avoids the extra copying to pack and unpack data into and out of communication buffers. However, it results in a large number of small messages, which can seriously hurt application performance.

One optimization is that if the source and destination ranks are on the same physical node, and the hardware and runtime system support shared memory across processes, we can directly copy the data from source to destination using standard `memcpy` rather than UPC++ `async_copy`. UPC++ allows us to do so by providing mechanisms for checking whether or not global pointers can be addressed directly and for casting them local pointers. We can then use the local pointer to treat a copy across co-located ranks as a local copy operation. While UPC++ automatically performs this check and optimization when `async_copy` is called, such a check would be performed in every iteration of the ghost-exchange loop above. Instead, we manually perform a single check before entering the loop and then defer to the local copy code if the destination is locally accessible.

A second optimization is to use the `async_copy_vis` functions when the destination is not locally accessible, which aggregate the small messages into large ones. Its effect is similar to manually packing and unpacking data but performed by the runtime library instead of the user.

Another optimization is to use group synchronization rather than global barriers. 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. For simplicity, the baseline implementation uses global barriers to perform these synchronizations, which results in excessive synchronization overhead. The optimized version, on the other hand, uses the group synchronization mechanism described in §IV-D, which has a similar interface as a barrier but only involves the neighboring ranks at the cost of requiring each rank to determine its set of neighbors before synchronization. However, we do amortize the cost over many iterations of the HPGMG algorithm by keeping track of neighbor information.

We further optimize synchronization by reducing the two synchronizations above to just one in each communication phase. In our experiments, we use the Chebyshev polynomial smoother, which alternately performs ghost exchanges on two variables, resulting in a double-buffering effect and allowing us to eliminate one synchronization on each variable.

VI. RESULTS

A. Experimental Setup

We evaluate flat MPI and UPC++ implementations of HPGMG-FV on Edison, a Cray XC30 system located at NERSC [5]. It is comprised of 5,576 compute nodes, each of which contains two 12-core Intel Ivy Bridge out-of-order superscalar processors running at 2.4 GHz, and is connected by Cray's Aries (Dragonfly) network. On Edison, the third rank of the Dragonfly is substantially tapered and, as in all experiments that run in production mode, there is no control over job placement. Each core includes private 32KB L1 and 256KB L2 caches, and each processor includes a shared 30MB L3 cache. Nominal STREAM [16] bandwidth to DRAM is roughly 103 GB/s per compute node.

The problem size is set as one 128^3 box per rank, and we report the best solve time. In order to simplify analysis, in our experiments, we set the `DECOMPOSE_LEX` macro in HPGMG to switch from recursive data ordering to lexicographical ordering of data. We run 8 ranks per socket, 16 processes per node, so that it is easier to maintain that all processes have equal number of boxes at the finest grid level under all concurrencies. Under such configurations, we expect the 64 processes attached to each Aries NIC to be arranged into a plane. Such a strategy has the effect of increasing off-node and network communication. Since we want to examine the communication effects at as large scale as possible on today's systems, we chose to use the process-only configuration.

B. Performance

Figure 6 shows the weak-scaling performance of different implementations as a function of the number of processes on the Cray XC30 platform. The enumerated optimizations are incremental from the baseline.

As expected, the highly tuned UPC++ bulk version (labeled as "Bulk") obtains the best overall performance (comparable to MPI as shown in §V-A), while the original implementation with naturally grained message sizes (labeled as "Baseline") delivers the worst performance. When 8 processes are used (all inside a single socket), the natural version is about $1.44\times$ slower than the bulk version, and when 64 processes are used (4 nodes), the performance gap increases to $2.2\times$. Nevertheless, at 32K processes, the performance gap only increases to $2.4\times$. Since a gap exists even on a single node, both on-node and intra-node performance issues must be addressed.

When shared-memory support is enabled in UPC++ (labeled as "+SHM"), performance improves substantially. Figure 7 illustrates the effect of enabling shared-memory support by comparing performance with and without such support in a microbenchmark that consists solely of one-sided put operations between two processes on the same node. As expected, there is a significant performance gap between the two versions.

Moreover, we distinguish the local data movements and remote ones explicitly in our code. This not only helps us to avoid the overhead of going through the UPC++ runtime but also enables us to perform special optimization for local operations, such as eliminating short loops. The improved performance is labeled as "+Local Opt", which maintains parity with the bulk version up to 64 processes. However, for higher concurrency, its performance stills falls far behind the bulk version.

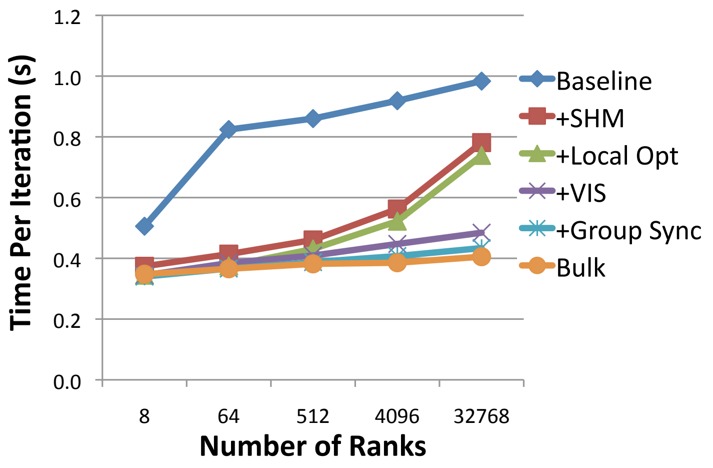
C. Scalability

To further improve performance at scale, we make use of the non-contiguous transfer function `async_copy_vis`, which can aggregate fine-grained (doubleword-sized) messages targeted at the same destination process into a small number of large messages. Due to the linear data organization of the 3D box data, the ghost zone communicated in the $\pm i$ directions are highly strided in memory, requiring many 8-byte messages if VIS is not used. As one scales (using lexicographical ordering), communication in $\pm i$ direction progresses from entirely zero-overhead, on-socket communication, to requiring communication over the PCIe bus, to communicating over the Aries NIC. Similarly, communication in $\pm j$ progresses from zero-overhead, on-node communication to requiring communication on rank-1 of the Aries Dragonfly. By using GASNet VIS to coalesce these small messages, the performance improves greatly at scale as shown by the "+VIS" line in Figure 6.

Table II illustrates the communication requirements by highlighting the on-node and network communication links exercised as a function of concurrency (assuming ideal job scheduling). The inflection points in Figure 6 are well explained by this model. The flood of small messages at low concurrencies map entirely to on-node links and thus do not substantially impede performance. Conversely, at 32K, PCIe overheads are exercised, resulting in degraded performance.

Table II
MAPPING OF HPGMG-FV COMMUNICATION PATTERNS (FINEST GRID SPACING ONLY) TO NETWORK TOPOLOGY AS A FUNCTION OF SCALE. HERE, "RANK" REFERS TO THE RANK OF THE ARIES DRAGONFLY ASSUMING IDEAL JOB SCHEDULING.

	messages	8	64	512	4K	32K
$\pm i$	$128^2\times$8B	on-socket	on-socket	on-socket	on-node	PCIe
$\pm j$	$128\times$1KB	on-socket	on-node	PCIe	rank-1	rank-1
$\pm k$	$128\times$1KB	on-socket	PCIe	rank-1	rank-2	rank-3



	Baseline	+SHM	+Local	+VIS	+Group	Bulk
Manual buffer packing						✓
Cast global pointers to local		✓	✓	✓	✓	✓
Eliminate short loops			✓	✓	✓	✓
Use GASNet VIS				✓	✓	
Synchronization	barrier	barrier	barrier	barrier	group	P2P

Figure 6. HPGMG Performance on Edison as a function of optimizations in the runtime. Here, we use 8 UPC++ ranks (or 8 MPI ranks) per socket, 16 per node. OpenMP is not used. The table shows which optimizations are employed for each implementation.

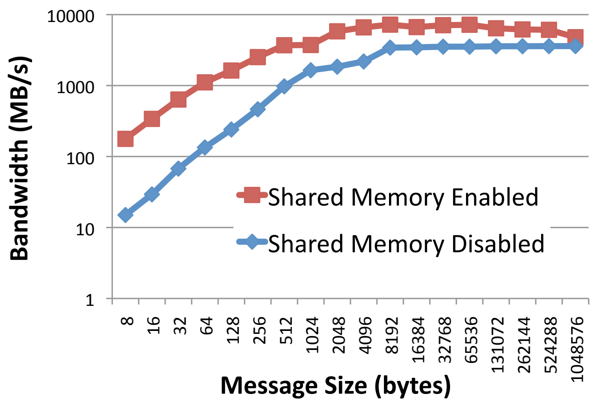


Figure 7. Microbenchmark showing GASNet's performance on put operations on the Cray XC30 platform using 2 processes inside one node. The lower blue curve represents the baseline where shared-memory support is exploited in neither the benchmark nor the runtime. The higher red curve represents the benefit of enabling shared memory in both the UPC++ and GASNet runtimes.

Our final optimization is to use group synchronization rather than the naïve global barriers in the previous shared-memory implementations. The communication pattern for HPGMG is primarily nearest-neighbor (either intra-level or inter-level), so it is only necessary to synchronize with neighboring ranks. While global barriers can simplify the implementation, they can cause performance to suffer as a result of interference or load imbalance. In order to minimize changes to the source code, we replaced barriers with the `sync_neighbor(neighbors)` function described in §IV-D, which uses point-to-point communication under the hood to synchronize with the ranks in the neighbor list. This further improves performance as shown by the "+Group Sync" line in Figure 6, resulting in a final performance that is comparable to the highly tuned bulk UPC++ and MPI implementations. Further performance improvement may be possible by overlapping communication and computation, but that is beyond the scope of this paper.

VII. Conclusions

In this paper, using High-Performance Geometric Multigrid (HPGMG-FV) as our driving application, we studied the runtime support needed for UPC++ to enable codes developed with naturally grained message sizes to obtain performance comparable to highly tuned MPI and UPC++ codes. Compared to the latter, which often require complex packing and unpacking operations, the natural versions provide substantial programming ease and productivity. However, their performance may suffer from a large number of small messages. To improve their performance, the runtime library needs to take advantage of hardware-supported shared memory, non-contiguous data transfers, and efficient group synchronization. With support for these features, the natural version of HPGMG-FV can deliver performance comparable to the version with manual packing and unpacking, showing that it is possible to obtain good performance with the lower programming effort of naturally grained message sizes.

To support non-contiguous data transfer, UPC++ provides a multidimensional domain and array library [11] that can automatically compute the intersection of two boxes and fill in the ghost regions. A version of HPGMG-FV that uses this domain and array library is currently under development.

In addition, we believe that many parallel applications with non-contiguous data-access patterns will gain in both productivity and performance if future network hardware supports: 1) scatter and gather operations for multiple memory locations; 2) remote completion notification of one-sided data transfers; 3) lower overheads and higher throughputs for small messages.

Acknowledgements

This material is based upon work supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award 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] R. Belli and T. Hoefler. Notified access: Extending remote memory access programming models for producer-consumer synchronization. In *29th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2015.

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

[3] D. Bonachea, R. Nishtala, P. Hargrove, and K. Yelick. Efficient Point-to-Point Synchronization in UPC . In *2nd Conf. on Partitioned Global Address Space Programming Models (PGAS06)*, 2006.

[4] W.-Y. Chen, C. Iancu, and K. Yelick. Communication optimizations for fine-grained upc applications. In *The Fourteenth International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[5] http://www.nersc.gov/systems/edison-cray-xc30/.

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

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

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

[9] D. Grnewald. BQCD with GPI: A case study. In W. W. Smari and V. Zeljkovic, editors, *HPCS*, pages 388–394. IEEE, 2012.

[10] https://hpgmg.org.

[11] A. Kamil, Y. Zheng, and K. Yelick. 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.

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

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

[14] H. Shan, A. Kamil, S. Williams, Y. Zheng, and K. Yelick. Evaluation of PGAS communication paradigms with geometric multigrid. In *8th International Conference on Partitioned Global Address Space Programming Models (PGAS)*, October 2014.

[15] C. Simmendinger, J. JŁgerskpper, R. Machado, and C. Lojewski. 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.

[16] http://www.cs.virginia.edu/stream/ref.html.

[17] J. J. Willcock, T. Hoefler, and N. G. Edmonds. AM++: A generalized active message framework. In *The Nineteenth International Conference on Parallel Architectures and Compilation Techniques*, 2010.

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

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

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