Experiences of Applying One-Sided Communication to Nearest-Neighbor Communication

Hongzhang Shan, Samuel Williams
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA 94720, USA
{hshan, swwilliams}@lbl.gov

Yili Zheng
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA 94720, USA
{yzheng}@lbl.gov

Weiqun Zhang
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA 94720, USA
{weiqunzhang}@lbl.gov

Bei Wang
PICSciE
Princeton University
Princeton, NJ 08540
{beiwang}@princeton.edu

Stephane Ethier
Princeton Plasma Physics Laboratory
Princeton University
Princeton, NJ 08540
{ethier}@pppl.gov

Zhengji Zhao
NERSC
Lawrence Berkeley National Laboratory
Berkeley, CA 94720, USA
{zzhao}@lbl.gov

***Abstract*—Nearest-neighbor communication is one of the most important communication patterns appearing in many scientific applications. In this paper, we discuss the results of applying UPC++, a library-based partitioned global address space (PGAS) programming extension to C++, to an adaptive mesh framework (BoxLib), and a full scientific application GTC-P, whose communications are dominated by the nearest-neighbor communication. The results on a Cray XC40 system show that compared with the highly-tuned MPI two-sided implementations, UPC++ improves the communication performance up to 60% and 90% for BoxLib and GTC-P, respectively. We also implement the nearest-neighbor communication using MPI one-sided messages. The performance comparison demonstrates that the MPI one-sided implementation can also improve the communication performance over the two-sided version but not so significantly as UPC++ does.**

***Index Terms*—Concurrent Programming; Performance measures**

I. Introduction

Nearest-neighbor communication appears in many contemporary applications and represents one of the most important communication patterns [2], [33]. It is often implemented using MPI two-sided messages. Also, for performance concern, large messages are preferred. The communicated data are packed at the source side and unpacked at the destination side. There is at most one message between any two processes for each communication phase.

Compared with two-sided communication, one-sided communication separates data transfer from synchronization. This separation enables multiple data transfers to use one synchronization operation so that the synchronization cost can be amortized well by such multiple data transfers. Another difference is that the one-sided message initiator needs both source and destination information to start the data transfer.

In this work, we study the performance of applying one-sided communication on an adaptive mesh refinement framework (BoxLib) [2], and a scientific application GTC-P [33], [19], whose communication patterns are dominated by the nearest-neighbor communication. Both applications exhibit dynamic communication characteristics. In BoxLib, the message size changes dynamically and the communication buffers are also dynamically allocated and freed for each iteration in response to adaptive mesh refinement. As such, each process needs to dynamically collect the requisite communication information before it can initiate the one-sided messaging. In GTC-P, which is a particle-in-cell code, a process will communicate with its left and right neighbors when exchanging particles. However, to save memory, it only maintains one receiving buffer so that one process (such as the left neighbor) will know the buffer address in advance and perform the one-sided communication. As a result the other process (the right neighbor) needs to know the buffer space used by the left neighbor before it can transfer the data.

One-sided communication is supported by many PGAS languages [35], [4], [7], [21], [6], [13]. In this work, we use UPC++, a library-based PGAS programming extension to C++ [35]. Unlike other PGAS languages that require new compiler front-end support, UPC++ adopts a C++ template meta-programming implementation strategy that reduces development and maintenance costs while being C++ standard-compliant. UPC++ is built on top of the GASNet communication API [11] and runs on diverse systems from laptops to supercomputers. For comparison, we also implement the communication using MPI one-sided interface. Since the official release of the MPI one-sided interface, few practices have been reported about its applications in real world scientific applications [26], [24]. This situation lasts even after its major update in 2012. Users are uncertain about whether they can realize the performance benefit on current HPC platforms and how.

Our approach is to focus on the communication because our applications have a large code base and our effort can be composed with other work for local optimizations. First, we try to directly replace the two-sided communication operations with the corresponding one-sided constructs. The big difference between two-sided and one-sided communications is that one-sided communication separates the data transfer and synchronization. Therefore, when applying one-sided messaging, special attention should be paid to synchronization to guarantee the correctness of the data.

Another advantage of separating the data transfer and synchronization is that the message originator has the full knowledge of both the source and destination information, it can easily control when and how the data should be transferred, making the maximum use of the network and overlapping the communication with local computation. We have applied this strategy to the GTC-P application and have achieved significant performance gain.

II. RELATED WORK

The performance and potential of one-sided communication have been studied by many researchers, although most of them focus on FORTRAN Co-Arrays (CAF) [5], [21] and UPC [31]. In this study we focus on MPI one-sided communication and UPC++. While the data transfer functions are similar, the synchronization mechanisms are different. El-Ghazawi and Cantonnet [10] discussed UPC performance and potential advantages using NPB applications. Shan et al. [28] studied the performance of MILC in UPC and IMPACT-T in CAF and found that the one-sided versions significantly outperform the two-sided versions at large-scale. Preissl et al. [25] studied the performance of a communication skeleton extracted from the GTS Gyrokinetic Fusion Application and showed better scalability and performance by using CAF and OpenMP parallel tasks to overlap computation and communication. Worley and Levesque [34] also found that using CAF for latency-sensitive communication for the Parallel Ocean Program can improve its performance and scalability. There are numerous other languages and libraries that support one-sided communication, such as Chapel [6], SHMEM [1], and X10 [7], GPI-2 [13], ARMCI [9].

Regarding MPI one-sided communication, more than ten years ago, Mirin et al. [26] developed the finite-volume dynamical core for the Community Atmosphere Model using MPI one-sided communication coupled with threading. Gropp et al. [14] studied the potential of MPI one-sided on several small clusters using a synthetic Halo exchange. They found that one-sided communication showed surprisingly good performance on the SGI Altix and Sun Fire platforms but poor performance on the IBM SMP machine. Potluri et al. [24] found that using MPI-2.2 one-sided functions and overlapping communication can improve the performance of a Seismic Modeling application around 10% on an Infiniband cluster platform. Maynard [20] showed the inferior performance of MPI one-sided versus UPC and SHMEM for a distributed hash table application on the Cray XE6 platform and related its poor performance partially to the design of the MPI window object. Recently, Gerstenberger et al. [12] described a scalable implementation of the MPI-3 RMA interface on a Cray XE6 platform.

Our work differs from the above in the following ways: first, our applications show different dynamic communication characteristics instead of fixed communication patterns. Some of them require reallocating the communication buffers for every iteration. This may cause inconvenience for one-sided communication and hurt its performance since the message originator needs to know all source and destination information. We studied different approaches to address this dynamic issue. Secondly, we analyze the performance difference and possible reasons between one-sided and two-sided communications in detail. Thirdly, instead of simply trying to overlap communication and computation as two-sided communication, we pipeline the one-sided communication with local computation and use the best message sizes.

III. ONE-SIDED COMMUNICATION

The potential advantage of separating data transfer and synchronization is that the synchronization cost can be reduced if there are a series of data transfer operations between two consecutive synchronization points. However, users have to handle the synchronization explicitly to guarantee data consistency and correctness. In two-sided communication, the synchronization is implicit, it is combined with the data transfer. The typical communication involves MPI_Isend, MPI_Irecv, and MPI_Wait operations. The matching between the sending and receiving messages is controlled by the value of the message envelope, including the source, tag, and communicator.

Another big difference is that one-sided data transfer functions require the originator to know all communication parameters, both for the sending side and for the receiving side while in two-sided communication, the sender only needs to know the sending address and the receiver knows the receiving address.

Next, we will describe the MPI and UPC++ one-sided data transfer functions and synchronization mechanisms used in our benchmarks.

A. MPI One-sided Communication

Before a process can initiate one-sided operations, it must create an MPI_Win object, which defines the memory exposed to the subsequent one-sided functions, also called RMA operation. Creating the window is a collective operation among all the processes of this window group. Windows can be created with system-allocated buffers or user-allocated buffers. In addition, applications may attach new memory buffers to a dynamic window during execution.

To transfer the data, we use MPI_Put function which requires the following parameters:

```
MPI_Put(
   origin_addr, origin_count,
   origin_datatype,
   target_rank, target_disp,
   target_count, target_datatype, win)
```

`origin_addr`, `origin_count` and `origin_datatype` specify the source information while `target_rank`, `target_disp`, `target_count` and `target_datatype` specify the destination information.

MPI one-sided includes three synchronization approaches. The first one is to call MPI_Win_fence before and after the one-sided functions. MPI_Win_fence is a collective operation and behaves like a barrier operation on the group of the processes sharing the window object. All MPI one-sided operations issued prior to the fence will complete before the fence call returns.

The second approach is to use pairwise synchronization (PSCW). This approach involves four functions, MPI_Win_post, MPI_Win_start, MPI_Win_complete, and MPI_Win_wait. This approach is potentially more efficient compared with the first approach MPI_Win_fence as it minimizes the synchronization requirements, and only pairs of communicating processes synchronize [23], [27]. Therefore we deployed this approach in our MPI one-sided implementation. The post and wait are used to receive data while the start and complete are used to send data. If they are used together, a proper order must follow.

Figure 1 illustrates how to use PSCW to synchronize between two processes P0 and P1, where MPI_Win_put was used to do the data transfer. P1 calle MPI_Win_post to request data from P0 and calls MPI_Win_wait to wait for the data arrival. P0 calls MPI_Win_start to prepare to send data. When P0 receives the synchronization signal post from P1, it calls MPI_Put to transfer the data from the memory of P0 to the memory of P1. A series of put operations can be issued. Finally P0 calls MPI_Win_complete to wait for all the put operations to finish. When all transferred data have arrived at the memory of P1, P0 will end the MPI_Win_complete call and send a synchronization signal to P1 to end the MPI_Win_wait call.

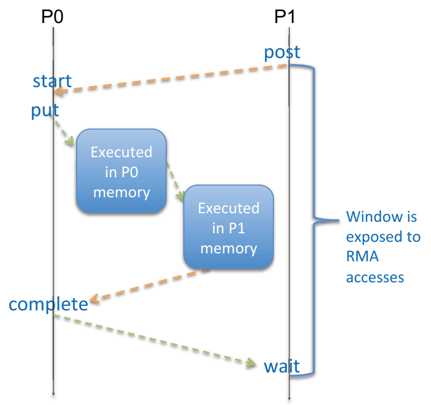

Fig. 1. The pairwise synchronization for MPI one-sided (PSCW). Dashed lines represent synchronization (extracted from [23])

The third approach is to use a shared lock with MPI_Win_lock and MPI_Win_unlock functions. This approach is similar to the UPC++ synchronization method via shared variables. However, this approach requires us to construct additional synchronization to maintain our program correctness, imposing additional programming challenges. Therefore, we did not explore this option. Instead we focus on the point-to-point synchronization primitives provided by the programming model itself.

B. UPC++

Data transfer can be realized by calling the non-blocking `async_copy` function to send data directly from `src` and `dst`. Because UPC++ provides a global address space, the source and destination buffers are represented by a `global_ptr<T>` type, which points to one or more shared objects of type `T`.

```
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.

A similar but more powerful function implements the signaling put [3] operation:

```
async_copy_and_signal(
           global_ptr<T> src,
           global_ptr<T> dst,
           size_t count,
           event *local_complete,
           event *remote_complete,
           event *signal_event);
```

The `signal_event` is the event to be signaled on the `dst` rank after data transfer finishes, `local_event` is the event on the sender when the local buffer can be reused, while `remote_event` is the event on the sender when the data transfer finishes. All of these event pointers can be set as NULL to prevent the event from occurring.

Different from MPI, UPC++ supports remote task execution:

```
async(rank r, event *e)(func, args...)
```

which enables one process to schedule a function with parameters `args...` to be executed on process with rank `r` and wait on event `e` to occur.

UPC++ supports three synchronization approaches: 1) upcxx::barrier() similar to MPI_Barrier and MPI_Win_fence operations; 2) shared variables in the global address space; 3) point-to-point synchronization via `async` and `async_wait`. Figure 2 illustrates how this works.

When P1 wants to get data from P0, it launches a remote task on P0 to request data. At the same time, it passes the receiving address and other information as remote task parameters to P0. After the remote task has been executed on P0, P0 can call async_copy or async_copy_and_signal to transfer the data to P1 without its involvement. After all the data has been transferred, both P0 and P1 will be signaled for completion. P0 can actually initiates the data transfer before the task execution. However, these operations will be queued on P0 and can only be allowed to proceed after the remote task has been executed.

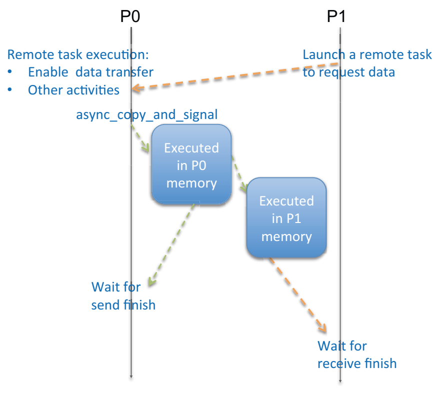

Fig. 2. The pairwise synchronization for UPC++. Dashed lines represent synchronization.

We use the point-to-point synchronization to implement the communication for both Boxlib and GTC-P.

IV. Experiment Platform and Methodology

Our performance evaluation was carried out on a Cray XC40 platform called Cori [8], which is currently deployed at NERSC. There are 1630 compute nodes total connected in a Cray Aries Dragonfly topology. Each node has two 16-core Intel Haswell processors running at 2.3GHz. Therefore there are 32 cores per compute node. Each core can run either 1 or 2 user threads, although we use only 1 thread per core in this study. The programming environment is PrgEnv-intel/5.2.82. However, we use cray-mpich/7.2.5 instead of the default cray-mpich/7.3.1 as we found that the former delivers much better performance for MPI one-sided communication and similar performance for MPI two-sided. We also link with the DMAPP library and setting environment variables

```
MPICH_RMA_OVER_DMAPP=1
MPICH_RMA_USE_NETWORK_AMO=1
```

as suggested by the intro_mpi man page. For UPC++, we use its most current version which can be downloaded from its bitbucket repository [32].

We selected two production codes for this study: BoxLib, and GTC-P, which show different dynamic characteristics. We use an incremental approach and focus only on the communication parts as these benchmarks have a very large code base and would require many years of development effort for a complete rewrite. Ultimately, our enhancements to the communication sections can be combined nicely with other efforts to optimize on-node computation or algorithmic changes.

In terms of a general methodology used throughout the paper, we first try to replace the MPI two-sided constructs with the corresponding one-sided constructs. Then we take advantage of one-sided communication to seek further optimization, through techniques such as message pipelining to overlap message packing and data transfer, or even directly sending the data to destination to avoid unpacking.

V. BoxLib

BoxLib [2] is a software framework for building parallel partial differential equation solvers. It uses the adaptive mesh refinement (AMR) technique to focus more computational resources on regions where high resolution is required. An example of a three-level hierarchy of block structured AMR grids is shown in Figure 3.

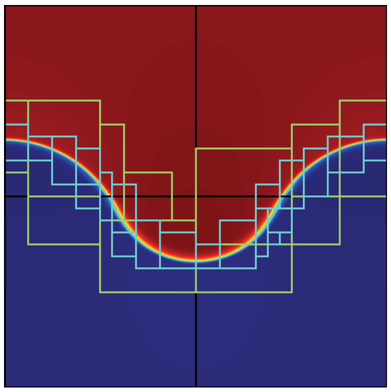

Fig. 3. An example of three-level BoxLib AMR grids, represented by black, yellow, and blue rectangular boxes.

Each AMR level contains a union of boxes. A fine level is strictly contained by a coarse level except that they can both touch the computational domain boundaries. The boxes typically have various sizes even at the same level. Each box has a multi-dimensional array of numerical cells with level-dependent uniform size. The numerical cell size at a fine AMR level is usually a factor of 2 or 4 smaller than that at the next coarse level. Boxes on the same level are non-overlapping, but ghost cell exchanges are needed for computations that require data from other boxes. Besides communication between boxes on the same level, there is also communication between AMR levels. Every few time steps, the grids are adaptively rebuilt in response to the evolving system. Although the communication pattern is complicated due to the complexity of the grids, BoxLib can use the knowledge of meta-data to perform point-to-point communication.

A. *MPI Two-sided Implementation*

Since the whole level grids are not static, but continually evolve during a run, it becomes difficult to predict the message size in advance. Therefore, the sending and receiving buffers need to be dynamically allocated for every ghost exchange. Also, given the discontiguous nature of the box data in some directions, the ghost regions are packed at the source and unpacked at the destination. Each process only needs to send and receive one message for each of its neighbors. Without packing and unpacking, there would be too many small messages, leading to performance slowdown. This performance concern also prevents us from pursuing the approach to directly send data from source to destination. A detailed performance analysis between fine-grained messages and bulk

TABLE I
BOXLIB COMMUNICATION TIME BREAKDOWN AT 4096 CORES (IN SECONDS). NOTE, SEND TIME IS INCLUSIVE OF EXCHANGE TIME.

	Packing	Sending	Waiting	Local	Unpacking	Exchange	Total
MPI(Two-sided)	0.77	0.20	14.29	0.48	1.57	N/A	19.00
MPI(One-sided, PSCW)	0.93	2.93	7.50	0.55	0.62	2.80	14.40
UPC++	0.69	3.56	3.60	0.49	0.73	N/A	10.50

messages for similar communication pattern can be found in [30], [29].

Once the communication buffers have been allocated, the communication stage follows the following typical steps to program in two-sided MPI communication:

1) Packing: Issue `MPI_Irecv` functions for all neighbors. Pack the ghost data into the corresponding sending buffers, 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 received data into the corresponding ghost zones of the local boxes.

The whole ghost exchange process is packaged into a function called *FillBoundary*.

B. MPI One-sided Implementation

Dynamically reallocating the receiving buffer is not a problem using two-sided communication, since the sender does not need to know where to put the data on the receiver side. It's the responsibility of the receiver itself. However, it causes complications for one-sided messaging since the sender needs to know both the source and destination information. To solve this problem, an extra stage before actual data transfer is needed to exchange the buffer address with neighbors first. We use two-sided point-to-point communication to finish this operation. Later we will show that it does affect the total performance.

To convert into MPI one-sided communication, we need to create one MPI_Win object that is needed for all the MPI one-sided functions. It is dynamically created using MPI_Win_create_dynamic and the communication buffers are attached to it. To transfer the data, we replace the MPI_Isend and MPI_Irecv pairs with MPI_Put. The remaining problem is synchronization. We select the potentially more efficient MPI PSCW point-to-point synchronization. To use this synchronization mechanism, we need to create the sending group and receiving group, which are the parameters needed by MPI_Win_post and MPI_Win_start.

C. UPC++ Implementation

In UPC++, dynamically allocating the communication buffer is not an issue. The receiver will call the `async` function with the newly allocated receive buffer address as its parameters to activate the data transfer on the sender. Therefore, we can avoid the extra stage needed by MPI one-sided. Data transfer is carried out by the `async_copy_and_signal` function.

D. Performance

In our BoxLib benchmark, performance is measured for a hierarchical grid generated from an AMR simulation of merging white dwarfs [18]. It has total 25680 boxes and around 840 million grid points at finest level with total five levels. Figure 4 shows the strong scaling performance of different approaches as a function of the number of processes. Overall, MPI one-sided performs better than MPI two-sided, especially at high concurrencies. UPC++ performs best.

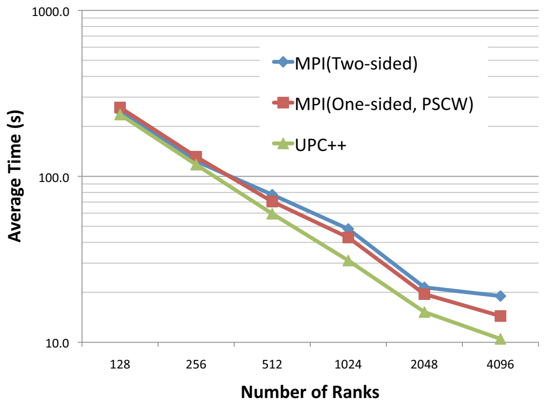

Fig. 4. The strong scaling performance of different programming models for BoxLib.

To further understand the performance difference among one-sided and two-sided communications, Table I lists the major components of the communication time, including "packing" the data from source into sending buffer, "unpacking" the data from receiving buffer to its destination, "sending" (time spent in the data transfer function), "waiting" for communication finish, and "local" (time spent in local computation). In addition, we also show the time used for exchanging the receive buffer address under MPI one-sided (exchange), which is included in the sending time.

For MPI two-sided, it is dominated by waiting time. This is expected as MPI_Isend and MPI_Irecv does not take much time. Its unpacking time is also surprisingly higher than both one-sided communications. We are not currently clear about the reason and is under further investigation. MPI one-sided spent less time waiting but more time in sending. The higher sending time is almost completely due to exchanging the buffer address and load balance plays a big factor in the exchange time. However, its shorter waiting time indicates the efficiency of MPI one-sided communication, probably less protocol overhead as the rendezvous protocol is used for two-sided communication [22]. UPC++ does not have the exchange time but it do spend more time in sending because it tries to move the data onto the network before it returns to better overlap the computation and communication.

VI. Gyrokinetic Toroidal Code (GTC-P)

The Princeton Gyrokinetic Toroidal Code (GTC-P) [33], [19] simulates plasma turbulence in magnetic confinement fusion devices called tokamaks. GTC-P follows the motion of ions and electrons in toroidal geometry by solving a 5D gyrophase-averaged Vlasov-Poisson equation using a particle-in-cell (PIC) algorithm. During each PIC time-step, the charge distribution of the particles is interpolated onto a grid, Poisson's equation is solved on that grid, the electric field is interpolated from the grid back to the particle positions, the phase-space coordinates of the particles are updated using a time-advance Runge-Kutta algorithm, and finally, the exiting particles are shifted to their new destination according to their new coordinates and the domain decomposition. Our work focuses on the shift operations since it is the most communication intensive part of this code. As pointed out by researchers in [15], network performance has become increasingly important to the overall GTC-P performance.

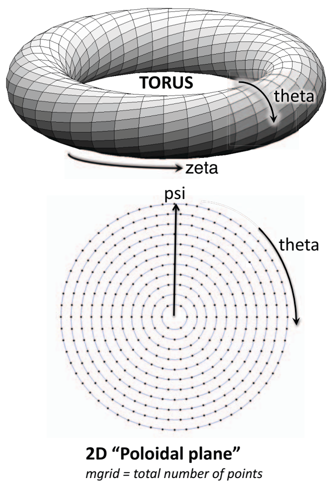

Fig. 5. The 3D grid for GTC-P.

Figure 5 shows the 3D toroidal grid. The 3D domain is partitioned in both the toroidal dimension (*zeta*) and the radial dimension (*psi*). The number of subdomains in toroidal direction is defined as *mzetamax*. A particle position in the torus can be described with three coordinates: the position in the toroidal direction, the position in the radial direction, and the position in the poloidal direction within a toroidal subdomain. Particles move much faster in the toroidal direction than in the radial direction at every time step. Therefore, the messages transferred within the toroidal communicator are of much larger size than the messages exchanged within the radial communicator. In addition, as particles will most likely move one or a few steps in each time iteration, the dominant communication occurs between two nearest neighbors. Therefore, the MPI_Sendrecv function is iteratively used to exchange moved particles with left and right nearest neighbors. The particle shift is organized as follows:

1) Scan particle array looking for particles that have moved out of local domain and compute the number of *msendleft* and *msendright*.
2) Calling MPI_Sendrecv function to exchange *msendleft* and *msendright* with left and right neighbors to get *mrecvleft* and *mrecvright*.
3) Pack the exiting particles into *sendbuffer*
4) Calling MPI_Sendrecv to exchange exiting particles with left and right neighbors and put them into *recvbuffer*.
5) Unpacking the received data to the end of the particle arrays

In the next round, the process will scan those newly received particles and repeat the above five steps. This process may repeat up to *mzetamax/2* times or until no particles are moved.

A. MPI One-sided Implementations

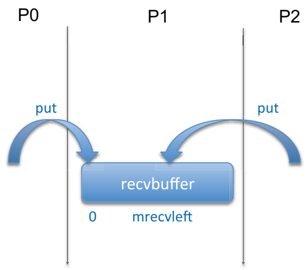

Fig. 6. The illustration of GTC-P buffer problem. Before P2 can fill the buffer for P1, it has to get *msendleft* from P0 first.

We first focus on step 4 by replacing the MPI_Sendrecv functions with MPI_Put. The *recvbuffer* is preallocated. So we can create a MPI_Win object for the *recvbuffer* and use PSCW synchronization mechanism. However, due to memory efficiency, each process only maintains one *recvbuffer*, i.e., both its left and right neighbors will put their exiting particles into it. Suppose the left neighbor fills the buffer starting from offset 0, the right neighbor does not know where to start. As illustrated in Figure 6, P0 fills P1's recvbuffer from offset 0 to *mrecvleft* and P2 should fill the buffer from offset *mrecvleft*. But it does not own this information. Therefore, one more piece of information must be exchanged in step 2 between P0 and P2. After this additional information is exchanged, we can then replace the MPI_Sendrecv function with MPI_Put. It should be noted that it is possible to maintain two receiving buffers to simplify this problem, but that requires more memory to be allocated.

1) UPC++ Implementation: For UPC++, we use remote tasks to synchronize. One advantage is that in step 2 we only need to explicitly exchange *msendright*. The offset needed in

TABLE II
THE TIME BREAKDOWNS (IN SECONDS) OF GTC-P COMMUNICATION FOR 256X8 CORE CASE

	Packing	Comm	Unpacking	Imbalance	Total
MPI(Two-sided)	17.2	39.0	4.9	24.2	85.3
MPI(One-sided, PSCW)	17.2	54.3	4.9	16.0	92.5
UPC++	18.8	18.4	7.2	15.1	59.4
	Packing+Comm		Unpacking	Imbalance	Total
MPI(One-sided, PSCW, pipeline)	58.2		0	15.6	73.7
UPC++ (pipeline)	34.8		0	14.8	49.6

Figure 6 is obtained through `async` parameters. The function `async_copy_and_signal` is used to transfer the data.

B. Performance

Figure 7 shows the strong scaling performance of *shift* under different programming models under label MPI(two-sided), MPI(One-sided, PSCW), and UPC++. The measured problem is size C, which corresponds to the JET tokamak, the largest device currently in operation [17]. There are 96 particles per grid cell and the domain is partitioned into a fixed 64 subdomains in the toriodal (*zeta*) direction. The number of partitions in the radial (*psi*) direction is *total number of processes / 64*. GTC-P is an MPI+OpenMP hybrid code and we run the code with four MPI or UPC++ ranks per node and eight OpenMP threads per MPI rank. Overall, MPI one-sided performs worse than MPI two-sided (except for the smallest concurrency) while the best performance is delivered by UPC++. Table II shows the time breakdown spent in the *shift* communication for the 256 $\times$ 8 case. The *Imbalance* time is the synchronization time to check whether some processes still have particles to move. If not, the *shift* phase will complete. We can see that the MPI one-sided messaging approach requires much more communication time than MPI two-sided, indicating the inefficiency of MPI one-sided for GTC-P. Conversely, UPC++, which delivers the best performance, requires much less communication time than either MPI variants but some more time for data unpacking. It should be noted that GTC-P has much larger message sizes than BoxLib.

C. Message Pipelining

Table II shows that a significant amount of time is spent in packing and unpacking data. As we mentioned earlier, one advantage of using one-sided messaging is to use a proper message size to pipeline the messages and overlap packing and network communication. Therefore, there is no need to wait for all the data to have been packed before initiating the data transfer. We can choose a proper message size. As long as enough data has been packed, we send it out immediately so that we can pipeline packing and data communication. Furthermore, we can control the packing to pack the data for each data field individually so that the data can be directly sent to the corresponding destination array (structure-of-arrays layout), eliminating the unpacking stage. This is different from BoxLib, where the large amount of fine-grained messages will create a performance bottleneck.

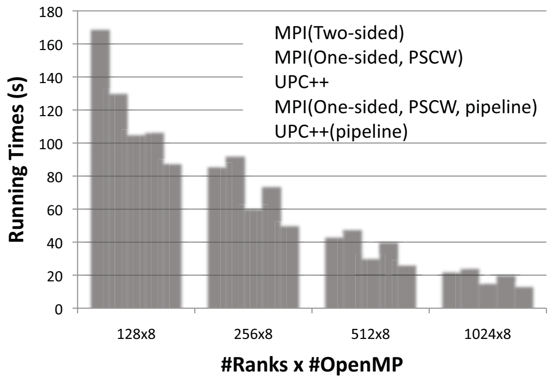

Fig. 7. The strong scaling performance of different implementations for GTC-P's *Shift* phase. Both MPI one-sided and UPC++ show substantial benefit when pipelined ('pipeline').

The resultant strong scaling performance for both pipelined MPI one-sided and UPC++ is shown in Figure 7 under label *MPI(One-sided,PSCW,pipeline)* and *UPC++ (pipeline)*. Compared with their corresponding versions without pipelining, both have improved their performance significantly. With pipelining, the performance of MPI one-sided finally outperforms MPI two-sided. The time breakdowns for the pipelined versions are also shown in Table II. We time the *Packing* and *Comm* phases together as it becomes difficult to differentiate them due to overlapping. For both MPI one-sided and UPC++, the sum of *Packing* and *Comm* time are reduced substantially. Also, the unpacking times are completely eliminated. Compared with the current, highly-tuned, two-sided MPI implementation, the pipelined MPI one-sided performs 10-60% better for the shift communication phase and 2-16% better for the whole application. Moreover, UPC++ implementation performs 60-90% better for the shift phase and 17-23% better for the whole application. The performance gains come from better overlapping of data transfer and packing and eliminating the unpacking stage.

VII. SUMMARY AND FUTURE WORK

In this paper, we examine the performance of one-sided communication for nearest-neighbor communication using two production codes, an adaptive mesh framework (BoxLib), and a scientific application GTC-P. Both of them show very frequent synchronization requirements as there is only one data transfer per communicating pair between consecutive synchronization points. Such communication characteristics often do not favor one-sided communication. However, compared with MPI two-sided, MPI one-sided could deliver better performance for BoxLib and GTC-P by applying message pipelining.

UPC++ performs significantly better than MPI one-sided for BoxLib and GTC-P, which is due to efficient point-to-point synchronization which provides programming ease and performance benefits when buffers are allocated dynamically. Moreover, it enables better overlaps of local computation and communication.

Regarding programming effort and productivity, in order to leverage one-sided MPI, BoxLib, GTC-P (without message pipelining), and GTC-P (with message pipelining) required 170, 178, and 460 lines of code to be modified respectively. Similarly, UPC++ required 330, 340, and 590 lines. These changes were small compared to the size of the respective code bases (around 80000 lines for BoxLib, and 14000 for GTC-P). More importantly, the changes were localized to the communication routines making incremental changes and modularity a reality.

References

[1] Barriuso, R., and Knies, A. SHMEM users guide for C.

[2] Bell, J., Almgren, A., Beckner, V., Day, M., Lijewski, M., Nonaka, A., and Zhang, W. BoxLib user's guide. Tech. rep., CCSE, Lawrence Berkeley National Laboratory, 2016.

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

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

[5] Co-Array Fortran. https://en.wikipedia.org/wiki/Coarray_Fortran.

[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] 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.

[8] https://www.nersc.gov/users/computational-systems/nersc-8-system-cori/.

[9] Dinan, J., Balaji, P., Hammond, J. R., Krishnamoorthy, S., and Tipparaju, V. High-Level, One-Sided Models on MPI: A Case Study with Global Arrays and NWChem. In *SC2011, Poster* (November 2011).

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

[11] GASNet home page. http://gasnet.lbl.gov.

[12] 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.

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

[14] Gropp, W., and Thakur, R. Revealing the Performance of MPI RMA Implementations. In *Proc. of the 14th European PVM/MPI Users Group Meeting (Euro PVM/MPI 2007)* (2007).

[15] Modern gyrokinetic particle-in-cell simulation of fusion plasmas on top supercomputers. http://arxiv.org/abs/1510.05546/.

[16] https://hpgmg.org.

[17] Joint european torus (jet). http://www.efda.org/jet/.

[18] Katz, M. P., Zingale, M., Calder, A. C., Swesty, F. D., Almgren, A. S., and Zhang, W. White dwarf mergers on adaptive meshes. i. methodology and code verification. *Astrophysical Journal 819* (Mar. 2016), 94.

[19] Madduri, K., Ibrahim, K. Z., Williams, S., Im, E.-J., Ethier, S., Shalf, J., and Oliker, L. Gyrokinetic toroidal simulations on leading multi- and manycore HPC systems. In *Proc. Int'l. Conf. for High Performance Computing, Networking, Storage and Analysis (SC '11)* (New York, NY, USA, 2011), ACM, pp. 23:1–23:12.

[20] maynard, C. M. Comparing one-sided communication with mpi, upc and shmem. In *The Cray User Group (CUG)) Meeting* (2012).

[21] 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).

[22] MPI Forum. MPI–2: a message-passing interface standard. *International Journal of High Performance Computing Applications 12* (1998), 1–299.

[23] MPI: A Message-Passing Interface Standard Version 3.1. http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[24] Potluri, S., Lai, P., Tomko, K., Sur, S., Cui, Y., Tatineni, M., Schulz, K. W., Barth, W. L., Majumdar, A., and Panda, D. K. Quantifying performance benefits of overlap using MPI-2 in a seismic modeling application. In *In Proceedings of the ACM International Conference on Supercomputing (ICS10)* (2010).

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

[26] Sawyer, W. B., and Mirin, A. A. A Scalable Implementation of a Finite-Volume Dynamical Core in the Community Atmosphere Model. In *International Journal of High Performance Computing Applications,19(3):203212, 2005.*

[27] Shan, H., , Williams, S., Zheng, Y., Kamil, A., and Yelick, K. Implementing High-Performance Geometric Multigrid Solver with Naturally Grained Messages. In *9th International Conference on Partitioned Global Address Space Programming Models (PGAS)* (October 2015).

[28] 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).

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

[30] Shan, H., Williams, S., Zheng, Y., Kamil, A., and Yelick, K. Implementing High-Performance Geometric Multigrid Solver With Naturally Grained Messages. In *9th International Conference on Partitioned Global Address Space Programming Models* (2015).

[31] UPC Consortium. UPC language and library specifications, v1.3. Tech. Rep. LBNL-6623E, Lawrence Berkeley National Laboratory, Nov. 2013.

[32] UPC++ website. https://bitbucket.org/upcxx/upcxx.

[33] Wang, B., Ethier, S., Tang, W., Williams, T., Ibrahim, K. Z., Madduri, K., Williams, S., and Oliker, L. Kinetic turbulence simulations at extreme scale on leadership-class systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2013), SC '13, ACM, pp. 82:1–82:12.

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

[35] 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).