Scaling Communication-Intensive Applications on BlueGene/P Using One-Sided Communication and Overlap

Rajesh Nishtala*, Paul H. Hargrove†, Dan O. Bonachea* and Katherine A. Yelick*†
*Computer Science Division, College of Engineering
University of California at Berkeley, Berkeley, CA, USA
†High Performance Computing Research Department
Lawrence Berkeley National Laboratory, Berkeley, CA, USA
Email: {rajeshn,bonachea,yelick}@cs.berkeley.edu, phhargrove@lbl.gov

Abstract

In earlier work, we showed that the one-sided communication model found in PGAS languages (such as UPC) offers significant advantages in communication efficiency by decoupling data transfer from processor synchronization.

We explore the use of the PGAS model on IBM BlueGene/P, an architecture that combines low-power, quad-core processors with extreme scalability. We demonstrate that the PGAS model, using a new port of the Berkeley UPC compiler and GASNet one-sided communication layer, outperforms two-sided (MPI) communication in both microbenchmarks and a case study of the communication-limited benchmark, NAS FT. We scale the benchmark up to 16,384 cores of the BlueGene/P and demonstrate that UPC consistently outperforms MPI by as much as 66% for some processor configurations and an average of 32%. In addition, the results demonstrate the scalability of the PGAS model and the Berkeley implementation of UPC, the viability of using it on machines with multicore nodes, and the effectiveness of the BG/P communication layer for supporting one-sided communication and PGAS languages.

1. Introduction

As the demand for computation power continues to grow at an exponential rate, system designers are turning to higher degrees of parallelism to deliver increased performance, due to the oft-cited difficulties with increasing serial performance within a processor core. They are also increasingly focused on high performance networks which enable users to take advantage of the increasing degree of parallelism. In our work we use the IBM BlueGene/P [1] (BG/P) as an example of a machine in this category of systems.

As the number of cores in a socket and within a system continue to grow at an exponential rate, the ability to scale communication systems, programming models, and applications to this large scale once again takes center stage. Further aggravating the problem, the pressures of power and machine cost at these large scales motivate the use of lower core clock rates leading to weak integer performance relative to lower-scale predecessor systems. The relatively weak compute cores on the BG/P magnify the software overheads associated with the communication subsystem. Semantic matching between the higher level programming model and the underlying network hardware can help alleviate some of these overheads.

A new class of languages, called Partitioned Global Address Space (PGAS) languages, has recently emerged to aid in the performance and scalability of High Performance Computing (HPC) applications. Rooted in traditional shared memory programming models, these languages expose a global address space that is logically partitioned across the threads. To provide the illusion of a globally shared memory these languages employ a one-sided communication model whereby a thread may directly read and write the memory located at a remote node, without the explicit cooperation of the application thread(s) on the remote node. One-sided communication semantics have been shown to enable increased performance by decoupling processor synchronization from data transfer, operations which are implicitly linked in two-sided communication models such as MPI message-passing [2]. One of the major contributions of this paper is to show the scalability of such a programming model. To the best of our knowledge our Berkeley Unified Parallel C (UPC) compiler [3] is the first PGAS compiler available on the BG/P and this work comprises some of the largest-scale PGAS application numbers published to date.

In order to benchmark communication performance we use the NAS Parallel Benchmark FT [4]. The core of the NAS FT benchmark is a large three-dimensional FFT, a problem that is known to be limited by the bisection bandwidth performance of the network. In our previous work [5] we demonstrated the utility of one-sided communication and communication/computation overlap to realize significant performance improvements relative to MPI message-passing. However the previous version only examined a one-dimensional decomposition of the problem grid which limited the maximum number of processor cores. In this paper we examine a two-dimensional decomposition of the

grid which has a much more aggressive communication pattern than its one-dimensional counterpart. In particular we focus on how the one-sided communication semantics found in UPC aid in scalability on the BG/P.

We outline the PGAS and UPC programming models in Section 2 and describe GASNet, our portable, high-performance communication system, in Section 3. Section 3 also discusses our implementation of UPC over the BlueGene/P communication APIs and associated microbenchmark performance. Section 4 details the NAS FT benchmark and outlines our strategies for overlapping communication and computation. Section 5 describes our experimental methodology, and application results are presented in Section 6. Sections 7 and 8 present related work and conclusions.

2. Background

PGAS languages provide the programmer with an abstraction of a global address space, which is logically partitioned to give each thread a portion of shared memory to which it has affinity. They offer the programmability advantages of shared-memory models, but are carefully designed to allow efficient implementation on distributed-memory architectures. UPC, Titanium, and Co-array Fortran are examples of modern PGAS languages, as are the DARPA HPCS languages (X10, Chapel and Fortress). The study in this paper is based on UPC, although the observations on communication techniques are more broadly applicable to the entire family of PGAS languages and other parallel systems providing one-sided communication.

UPC [6] is a parallel SPMD superset of the ISO C 99 programming language. It has language support for parallel data distribution and provides a shared memory abstraction to the programmer, regardless of the memory model provided by the underlying hardware. The language provides a hybrid strict/relaxed memory consistency model that enables aggressive compiler communication optimizations, such as message aggregation and the automatic use of non-blocking remote memory operations to help tolerate network latencies with communication/computation overlap.

In this paper we used the portable, high-performance Berkeley UPC compiler [3]. On shared memory machines, accesses to the UPC shared address space translate into conventional load/store instructions. On distributed memory machines such as the BG/P, remote shared accesses translate into calls to the GASNet communication layer [7]. The GASNet communication API provides an expressive and portable interface for high-performance one-sided communication as a compilation target for PGAS languages. It offers portable and native implementations on a wide variety of modern HPC platforms, and serves as the communication layer for six PGAS compiler efforts and various prototype systems [8].

3. GASNet on BlueGene/P

3.1. The BlueGene/P Overview

The processing element found on the BG/P is a Quad-Core 850MHz PowerPC 450 processor. This Quad-Core chip is combined with 2GB of memory to form a compute node. Thirty-two of these compute nodes form a Node Card. Thirty-two node cards form one rack of the machine. Thus each rack of the machine holds 4,096 PowerPC 450 cores. We have scaled our experiments to four racks (16,384 cores) of a BG/P at Argonne National Lab named "Intrepid" [9].

Most inter-node communication on BG/P is done via a three-dimensional torus network, and the machine offers separate network hardware for barriers, collectives, and I/O. In our implementations of NAS FT, the main point-to-point communication uses the torus network while the barriers use the auxiliary barrier network. Each torus link provides a peak hardware bandwidth of 425MB/s in each direction, thus the six links into and out of a node provide an aggregate peak bi-directional bandwidth of 5100MB/s at each node. However due to packet overheads, as shown by Kumar et al. [10], the peak messaging bandwidths available to applications are 374MB/s per link in each direction, and 4488MB/s aggregate bi-directional per node.

IBM has designed the Deep Computing Messaging Framework (*DCMF*) [10] as a semi-portable open-source communication layer that provides the low level communication APIs for higher level programming models such as UPC and MPI. The MPI implementation on the BG/P, MPICH2 1.0.7 [11], uses DCMF for all its communication interactions with hardware. To achieve the best PGAS language communication performance, the GASNet communication layer implementation on BG/P also targets this communication API. Although the only HPC machine providing a DCMF implementation is the BG/P, the API was designed to accommodate possible future systems. DCMF provides point-to-point communication operations, plus collective operations that have been specifically designed for the BG/P to take advantage of the hardware collective networks when possible.

3.2. DCMF

DCMF offers three mechanisms for point-to-point communication. While the API offers many more features we focus on these three since they are the salient ones to our discussion. For more complete details, see [10].

- `DCMF_Send()`: The send operation is the active message mechanism in DCMF. It accepts the function to invoke on the remote node, its arguments, and a message payload. The client provides a callback to be invoked on the initiator when the data is locally reusable. With a send, the remote processor needs to

run the specified handler when the active message is received. Therefore the target processor has some non-trivial involvement in message reception.

- `DCMF_Put()`: Unlike the send, the caller of a put provides both the source *and* destination addresses. In addition, the client provides two callbacks to run on the initiator: one that is invoked when the data movement is locally complete and the other when the data has been delivered to the remote memory. Notice that since the source node provides all the information required for delivery, this is a one-sided operation; the operation can be retired with no interaction from the remote processor.
- `DCMF_Get()`: The get is an analog of the put operation. Like put, the initiator provides both addresses (local and remote) eliminating the need for any involvement from the remote processor. Unlike the put however, the user only provides one callback that is invoked when the data transfer is complete and the data is locally available.

DCMF provides the ability to overlap communication with other communication or computation through the use of the callback mechanisms. When the DCMF communication operations return it implies that the communication operation is in-flight and completion is not guaranteed until the callback is invoked. Thus anything that is done between message injection and associated callback invocation is potentially overlapped with the communication.

In GASNet, there are also three primary mechanisms for point-to-point communication. These map directly onto the DCMF calls described above. GASNet's Active Messages are implemented directly over `DCMF_Send()`, while the Get and Put operations are implemented directly over `DCMF_Get()` and `DCMF_Put()`. The remote completion callbacks of `DCMF_Get()` and `DCMF_Put()` provide the completion semantics required by GASNet without the need for any additional network messages. Using the one-sided communication model the initiator of the transfer provides all the information about the communication operation. Since no additional information is needed from the target node, the communication operation can always immediately deposit data into its target location.

In MPI, there are a number of requirements that any conforming implementation must ensure. These include point-to-point ordering and message matching guarantees. For instance, the communicator and tag information from the sender must be matched to that of a previously posted receive operation at the remote node before data can reach its final destination. To implement these semantics, data movement must generally either be delayed (as in a rendezvous protocol) or copied (as in an eager protocol). The message matching requirement and associated ordering restrictions may impose overheads on any MPI implementation. Lacking hardware or firmware assistance, MPI message matching is performed in software on the BG/P. Because the BG/P cores are relatively weak compared to the network performance, the software overheads associated with two-sided messaging can impact messaging performance on BG/P more severely than on other platforms.

In summary, GASNet has been implemented over DCMF as a very light-weight layer with no need to enforce additional semantics that are unavailable from DCMF. On the other hand, the nature of the MPI semantics require any conforming implementation to do additional work in software on BG/P for every message. As demonstrated below, the result is that GASNet is able to initiate and complete communication operations with significantly less software overhead than MPI on BG/P.

3.3. Microbenchmarks

As described above, the communication APIs provided by DCMF provide a very good fit to the communication semantics of GASNet. This section shows how this match enables better point-to-point performance than is achievable using MPI message-passing on this system. To quantitatively see the benefits of using GASNet relative to MPI we present latency and bandwidth microbenchmarks.

3.3.1. Latency Advantages of GASNet. In our first microbenchmark we compare the roundtrip GASNet and MPI "ping-ack" latency performance. For MPI this test measures the time needed for the initiator to send a message of the given size and the remote side to respond with a 0-byte acknowledgment. This benchmark is written using `MPI_Send()` and `MPI_Recv()` for both operations. The GASNet test measures the time to issue a put or a get of the given size and block for remote completion (when DCMF runs the remote completion callback). This comparison is made because while GASNet takes advantage of the remote completion notification of DCMF, MPI needs an explicit acknowledgement to implement a roundtrip network traversal (such as those required in applications with fine-grained irregular accesses). Figure 1 shows the performance comparison.

As the data show, GASNet's use of the remote completion notification of DCMF yields about half the latency of MPI for an equivalent operation. In fact, for message sizes up to 32 bytes, the advantage is slightly larger than the factor of two which the message count alone can account for. While the FFT benchmark we study later in the paper is bandwidth limited, this latency comparison shows that the close semantic match between GASNet and DCMF allow implementation of a PGAS language at relatively low cost. This low software overhead has implications for the effectiveness of communication/communication and communication/computation overlap on this system.

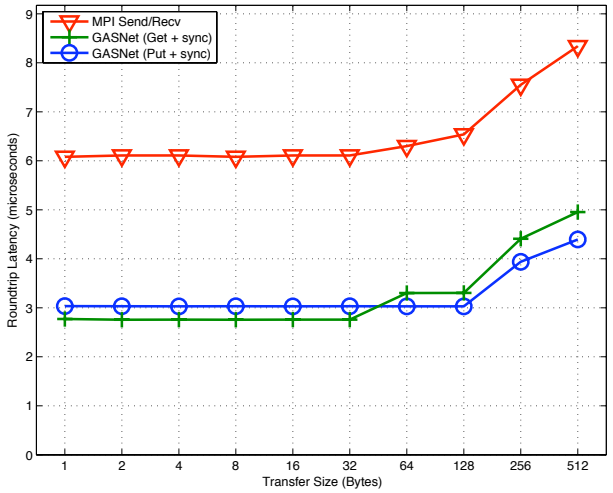

Figure 1: Roundtrip Latency Comparison

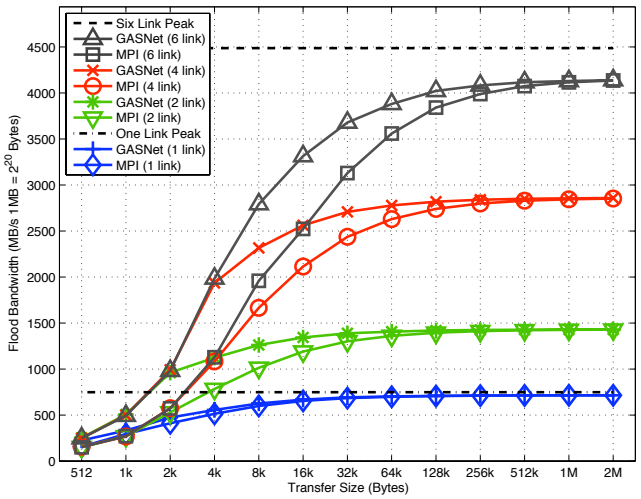

Figure 2: Flood Bandwidth Comparison

3.3.2. Multi-link Flood Bandwidth Performance. In our next microbenchmark we analyze the flood bandwidth performance of GASNet and MPI. Each BG/P compute node has six links to neighboring nodes, two links in each of the three dimensions of the torus. When measuring the effective bandwidth of the node, measuring the performance across only one link under-utilizes the bandwidth available at each node. Therefore our bandwidth test, like those presented by Kumar et al. [10], measures the bidirectional bandwidth performance on a varying number of links. Figure 2 shows the flood bandwidth performance. The maximum achievable payload bandwidth for one and six links are shown for comparison. For the GASNet tests one core per node initiates a long series of non-blocking puts of the specified size to the neighbors on each of the chosen links (round-robin). For MPI we implement the same pattern of communication using `MPI_Isend()`s and a preposted window of `MPI_Irecv()`s. We have varied the size of this window over powers of two and for each MPI data point report only the highest bandwidth achieved.

Unlike the latency microbenchmark, the message counts at the GASNet and MPI levels of the benchmark are identical for this comparison. Thus here we observe the communication/communication overlap that each software stack can achieve from the multiple hardware paths. At large message sizes, the cost of message injection is small relative to the data transfer times, hence the performance for both communication layers approaches the same asymptotic value. However for the mid-range message sizes, there is a significant difference in the achievable data transfer bandwidth between the two communication layers. GASNet adds less software overhead above the DCMF primitives than MPI, thus GASNet can inject messages into the network more efficiently than MPI at small and mid-range message sizes. Furthermore, MPI's two-sided message-passing semantics require the implementation to use a rendezvous protocol in order to leverage the high-performance, zero-copy RDMA hardware on this system, entailing additional DCMF-level messages that further magnify the overheads. Thus the data show that the MPI microbenchmark is able to extract far less of the available communication/communication overlap than its GASNet counterpart, leading to a loss in throughput for medium-sized messages. This effect is exacerbated in the presence of the additional network bandwidth made available through the multiple hardware links, while software overheads for message injection (and reception for MPI) remain serialized on the slow processor core. Later in Section 6 we will see that for the 3D FFT algorithm, running time is determined by the communication performance in this same middle range of message sizes.

4. Optimizing Bandwidth Limited Applications

As the previous sections show, GASNet and its one-sided communication model has performance advantages compared to MPI's two-sided message-passing. In this section we see how the performance advantages demonstrated in the microbenchmarks translate into overall application performance. To compare GASNet and MPI on the BG/P we use the NAS Parallel Benchmark [4] FT as a case study.

At the center of the NAS FT benchmark is a three-dimensional FFT. In a three-dimensional FFT the rectangular prism (of NX, NY, $\times$ NZ) points is evenly distributed amongst all the threads and FFTs must be done in each of the dimensions. Depending on how the data is laid out, the prism might need to be transposed in order to relocalize the data to perform the FFTs. We go into further details on this operation later in this section. At large scale this transpose step is often the performance-limiting step since it stresses the bisection bandwidth of the network.

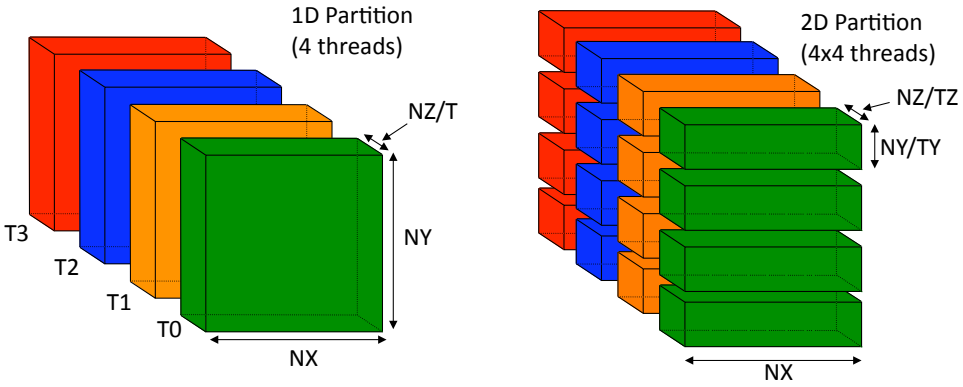

Figure 3: Comparison of 1D and 2D decompositions

As we have shown in our previous work [5], GASNet (and hence Berkeley UPC) allow effective overlap of communication and computation, enabling large performance gains. In that work we show that we can effectively use hardware features found in modern networks, such as Remote Direct Memory Access (RDMA), to significantly improve performance for an application that is often considered a bisection bandwidth limited problem. In addition, we show that GASNet is a better semantic match to these hardware features than MPI. One of the most significant results is that breaking up the exchange collective (i.e. `MPI_Alltoall` in MPI parlance) into point-to-point operations that are overlapped with computation leads to significant performance improvements[1].

Our previous work only examined a 1-D decomposition of the problem domain across the threads, which limited the maximum number of threads to be the $MIN(NX, NY, NZ)$. On BG/P and other systems that use high degrees of parallelism to realize performance, this limitation prohibited scaling to core counts found in typical installations. Thus we have extended our previous work to handle a 2-D thread layout. Figure 3 shows the differences between the two thread layouts. In a one-dimensional thread layout a particular thread owns $\frac{NZ}{T}$ planes of $NX \times NY$ points where T is the total number of threads. In a two-dimensional thread layout a particular thread owns $\frac{NZ}{TZ}$ planes and $\frac{NY}{TY}$ rows of NX points where the T threads are laid out in a $TY \times TZ$ thread grid.

In a one-dimensional thread layout two of the three dimensions of the FFT are fully located on one thread thus only one transpose needs to be performed: the one that relocalizes the data in the Z-dimension. In a two-dimensional thread layout only one dimension of the grid is contiguous on a thread and thus two rounds of transposes need to be performed to complete one FFT. The first relocalizes the data to make the Y-dimension contiguous and thus the communication is performed amongst teams of threads in the TY dimension (*i.e.* all the threads within the same thread plane in Figure 3). The second one relocalizes the data to make the Z-dimension contiguous amongst teams of threads in the TZ dimension (*i.e.* all the threads within the same thread row).

1. The way the exchange collective is specified in UPC and MPI, local data movement needs to be done before and after the collective to achieve a full matrix transpose operation. Throughout the rest of this paper when we use the term *transpose* we mean the entire matrix transpose including local and global data movement. When we refer to *exchange* we mean just the global data movement.

4.1. NAS FT Algorithms

In our previous work we showed that one can overlap the exchange step and the computation of the 1-D FFTs to realize performance improvements. While the principles carry over into the two-dimensional case the algorithms are slightly different from our previous work and we thus present them here. Throughout the rest of this paper we will define a *slab* as one of the planes a particular thread owns. Thus, for the initial data distribution given in Figure 3, each thread owns $\frac{NZ}{TZ}$ slabs, each of which is NX columns by $\frac{NY}{TY}$ rows.

4.1.1. Packed Slabs. Conventional wisdom suggests the best way to optimize this application is to perform the communication and the computation in two distinct stages, using what we shall refer to as the *Packed Slabs* algorithm. In the first stage this algorithm computes the FFTs for all the $\frac{NZ}{TZ} \times \frac{NY}{TY}$ rows that a thread owns across all the planes. The data is then packed and all the threads within the same thread plane exchange their data in one big communication step. After the first round of exchanges are finished the data is then unpacked and the next $\frac{NZ}{TZ} \times \frac{NX}{TY}$ rows of NY FFTs are performed. The data is then repacked and the the final communication step is done. Once the communication is done, the data is unpacked and the final $\frac{NY}{TZ} \times \frac{NX}{TY}$ rows of NZ length FFTs can be performed. This algorithm is more fully detailed in Algorithm 1 in the Appendix.

The Packed Slabs algorithm uses packing to maximize message sizes, in order to achieve the best bandwidth performance for those transfers. However this approach sacrifices the ability to overlap the communication and computation – at any given time either the communication or computational subsystems will be sitting idle.

4.1.2. Slabs. With the goal of overlapping communication and computation in mind we analyze a second algorithm. In the previous algorithm, each thread finishes all $\frac{NZ}{TZ}$ slabs before any communication is started. However, after a thread finishes a single slab there are no further dependencies that prevent the communication from being initiated. Thus in our *Slabs* algorithm, each thread initiates the communication on a slab as soon as the computation on that slab is finished. This overlaps the communication of the current slab with the computation of the next slab. This is more fully detailed in Algorithm 2 in the Appendix.

4.1.3. Summary. There exists a continuum of granularities of overlap ranging from initiating the communication after finishing one row of computation all the way to the method described in the Packed Slabs approach. On one extreme there are many fine-grained messages that are sent with abundant opportunities for overlap at the cost of smaller message sizes and higher message counts, and thus potentially higher network contention. On the other extreme we have fewer larger messages in the network at the cost of the ability to overlap computation with communication[2].

The Slabs algorithm is in the middle of this continuum, as it computes an entire slab of independent 1D FFT pencils before injecting them into the network. We explored finer-grained approaches (i.e. the Pencils approach described in our previous work) to achieve more aggressive communication/computation overlap, however on the BG/P system this finer-grained communication decomposition did not yield additional performance improvements for any of the configurations studied. For the sake of simplicity of explanation we do not show those results in this paper, however future work may validate how these algorithms perform on different architectures.

The two different algorithms have different impacts on the network hardware. The Packed Slabs approach sends larger and fewer messages while the Slabs approach sends more and smaller messages while simultaneously enabling communication/computation overlap (the total volume of data communicated is the same for all algorithms considered). Table 1 shows the difference in communication behavior of the two different FFT algorithms for what a single thread sends in each round. The Packed Slabs approach can have exactly one outstanding collective and that is not overlapped with any of the computation while the Slabs approach can have up to $\frac{NZ}{TZ}$ outstanding nonblocking collectives before needing to wait for the data movement to finish. In the later sections we will highlight the regions on the bandwidth microbenchmark described in the previous section where these two algorithms reside.

2. Note that all the algorithms exhibit communication / communication overlap.

5. Experimental Setup

5.1. Processor Layouts

Since each compute node has four cores, the BG/P has three different operating modes with four, two or one processes per node. In the latter two cases the presumption is that within the process, threading or OpenMP style parallelism will be employed to utilize the other cores. We conducted all our experiments using four processes per node (one per core), referred to as Virtual Node mode in IBM terms. The Virtual Node Mode adds a fourth dimension to the torus since at each grid point in the 3D torus there are four processes. Thus in this execution environment we are running our benchmark over a four dimensional torus network. However, all processes on a given compute node share the network hardware and thus the bandwidth.

An important step in application tuning on the BG/P is deciding how to map virtual process ranks to physical processor IDs to optimize communication patterns. We explored a variety of mappings and found that the default XYZT was optimal for both GASNet and MPI. For the sake of brevity we omit results for other mappings. Table 2 in the Appendix shows the processor sizes that we used along with the resultant processor grid.

5.2. Compiler and Runtime Information

To perform the 1D FFTs we use the IBM's vendor-supplied ESSL library that has been hand-tuned for BG/P [12]. ESSL gave consistently better serial performance for our usage pattern than FFTW 3.1.2 [13]. All our benchmarks are linked against the same ESSL library to ensure that the performance differences are a result of the communication differences arising from the different implementations rather than the serial FFT performance. The MPI implementation we use is MPICH2 version 1.0.7 that has been specially modified for the BG/P by IBM. The C compiler used is the IBM BG/P version of XLC 9.0. The UPC compiler used is Berkeley UPC 2.8.0 with GASNet 1.12.0.

6. Application Performance Results

Section 3 showed the microbenchmark bandwidth advantages of a one-sided communication model. In this section

	Packed Slabs	Slabs
Message Size in Round 1	$\frac{NZ}{TZ} \times \frac{NY}{TY} \times \frac{NX}{TY}$ elements	$\frac{NY}{TY} \times \frac{NX}{TY}$ elements
Number of Messages per Thread in Round 1	TY	$\frac{NZ}{TZ} \times TY$
Message Size in Round 2	$\frac{NZ}{TZ} \times \frac{NX}{TY} \times \frac{NY}{TZ}$ elements	$\frac{NX}{TY} \times \frac{NY}{TZ}$ elements
Number of Messages per Thread in Round 2	TZ	$\frac{NZ}{TZ} \times TZ$

Table 1: Summary of Message Counts and Sizes for the two different 3D FFT algorithms

we will see how this advantage translates into application scalability. We analyze the performance of two different common scaling metrics: (1) fix the problem size and vary the number of cores (*i.e.* strong scaling) and (2) vary the problem size linearly with the number of cores (*i.e.* weak scaling).

We try three different algorithms in each case: *Packed Slabs* written in MPI, *Slabs* written in MPI, and *Slabs* written in UPC. The MPI Packed Slabs implementation is the publicly available reference implementation of the NAS benchmark, written in Fortran with MPI. Minor modifications have been made to allow the use of an external FFT library such as ESSL along with adding new problem sizes in the build infrastructure. However, the critical parts of the code, especially relating to the communication have not been modified. The UPC Slabs implementation is a portable (machine-independent) implementation written entirely in UPC with calls to ESSL to perform the local 1-D FFT operations. It performs data transfers using the non-blocking memory copy library extension which is provided by Berkeley UPC [14], and that is expected to appear in the next UPC language specification. The MPI Slabs implementation is very similar to the UPC version (both are comprised primarily of C99 code), but this version performs the data transfers using `MPI_Isend()` and `MPI_Irecv()`. All of these implementations are available by request to the contact author.

6.1. Strong Scaling

In our first set of experiments we fix the problem size and use the NAS FT Class D ($2048 \times 1024 \times 1024$ double complex numbers). We vary the processor count from 512 cores (128 compute nodes) to 16,384 cores (4,096 compute nodes). The performance data is shown in Figure 4. We plot the performance on a log scale to show the application scalability. To highlight the differences, Figure 5 shows the same data normalized to the performance of the UPC Slabs algorithm, on a linear scale.

To estimate an upper bound on performance we calculate the bisection bandwidth based on the best one-link bandwidth performance for each of the two rounds of communication. Using this bandwidth we then estimate the time needed for the data transfer across the bisection at the given problem size. We then calculate the Gigaflop rate assuming that the computation is free and the communication is the only bottleneck. Thus the "Upper Bound" line shows a not-to-be-exceeded performance given that this is a communication bound problem. The upper bound line shows some interesting kinks, namely the jump from 1024 cores (256 nodes) to 2048 cores (512 nodes). At 256 nodes and below the network does not complete a three-dimensional torus, the network is just a 3D mesh. However when we cross over to 512 nodes the three-dimensional torus completes, thus doubling the bisection bandwidth and resulting in a super-linear performance improvement.

As the data show, MPI Packed Slabs realizes better strong scaling than MPI Slabs, indicating that when using MPI on this machine the advantages of overlapping communication and computation are outweighed by the added cost of sending more smaller messages. This supports the conventional wisdom that packing data into fewer larger messages yields the best performance. However, when comparing the Slabs implementations, we observe that UPC Slabs is consistently better than its MPI counterpart with speedup of between 33% and 110%, and a mean of 63% [3].

Additionally, UPC Slabs consistently outperforms MPI Packed Slabs by 11% to 67% (a mean of 37%). At the largest processor configuration, the UPC Slabs algorithm outperforms MPI Packed Slabs by 13% (1.61 Teraflops vs. 1.42). The superior performance of the UPC Slabs over MPI Packed Slabs is in contrast to the conventional wisdom of using the largest possible message sizes to achieve the best performance, but is consistent with our previous findings that performance can be improved by decomposing the exchange into smaller messages with an opportunity for overlap of communication with computation.

6.2. Weak Scaling

Another very common application scaling model is to keep a fixed problem size per core and vary the number of cores. In our experiments we run NAS FT Class D for the middle processor count (2048 cores) and scale the grid size linearly with the processor count [4]. Figure 6 shows

3. For all our percent difference calculations we use the formula: $(UPC - MPI)/MPI$.

4. Due to the nature of the FFT algorithm, scaling the problem domain by a factor of N increases the total memory and communication volumes by N, while the floating-point computation grows by $NlogN$. Results are given as flop rates and take this into account. Despite the increased computation per core at higher core counts, communication performance remains the dominant factor in all configurations examined.

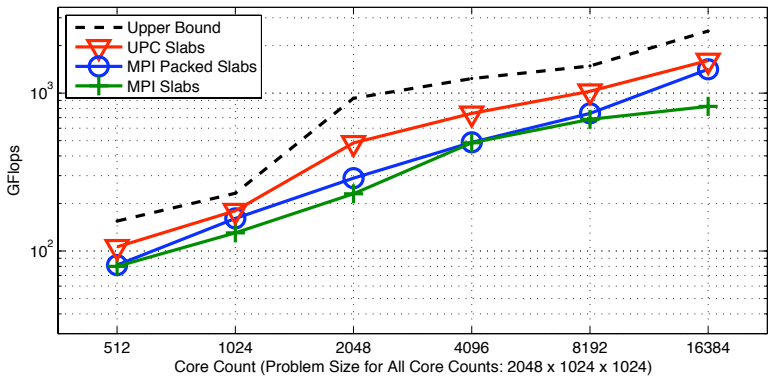

Figure 4: NAS FT Performance: Strong Scaling

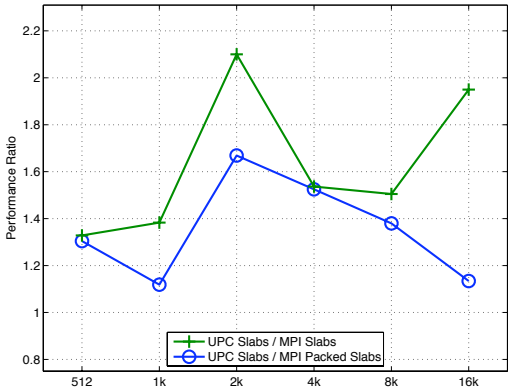

Figure 5: NAS FT Performance: Strong Scaling Performance Ratio

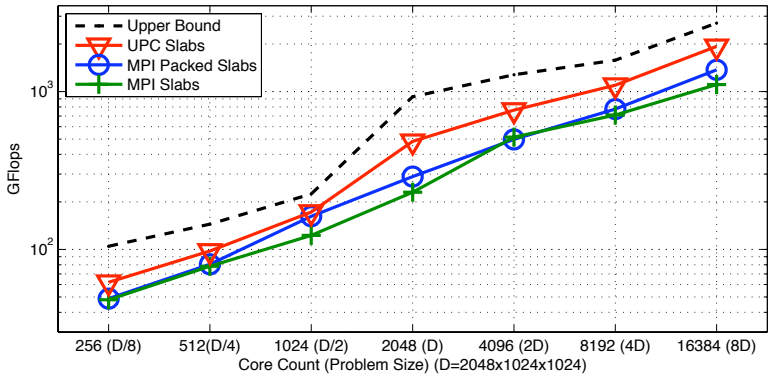

Figure 6: NAS FT Performance: Weak Scaling

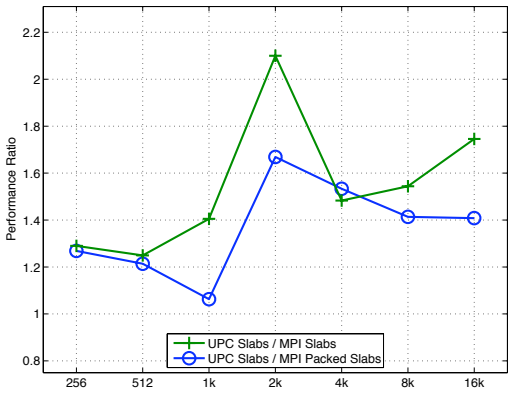

Figure 7: NAS FT Performance: Weak Scaling Performance Ratio

the weak scaling data. We again calculate the upper bound on performance based on the bisection bandwidth. Figure 7 shows the same data normalized to the performance to the UPC Slabs algorithm, to highlight the differences.

As the data show, the Slabs algorithm scales better in the weak scaling case than for strong scaling. In strong scaling, the fixed problem size implies a dramatic reduction in message size at large core counts, eventually becoming too small to effectively utilize the network or exploit overlap, and software overheads become an increasingly dominant factor. However, with weak scaling, the message sizes vary less, yielding consistently better performance. The message size for Slabs varies between 8KB and 32KB depending on the problem size and the processor grid.

As before the data show MPI Packed Slabs outperforms MPI Slabs for most processor configurations. Again this difference shows there to be no net performance advantages to reducing message size to achieve overlapping of communication and computation in MPI. However the data also show that using UPC to achieve to same overlap consistently produces the highest performance of the three implementations. This is consistent with the microbenchmark results that show GASNet better able to overlap communication with communication, but also suggests (as evidenced by the difference between MPI Slabs and UPC Slabs) that the UPC implementation achieves greater overlap of communication with computation as well, to yield improved application performance for an identical algorithm. We pursue this further below.

UPC Slabs outperforms the MPI Packed Slabs by 6% to 67%, with a mean advantage of 37%. The best MPI implementation runs at about 1.37 Teraflops while the UPC implementation runs at 1.93 Teraflops – a 40% performance improvement in overall application performance at 16,384 cores.

6.2.1. Application time breakdown (weak scaling). To further examine where the time is being spent in the weak scaling benchmarks we examine the performance breakdown in Figure 8 at 16,384 cores on a $4096 \times 2048 \times 2048$ grid. The times are grouped as follows: "Local FFT (ESSL)" shows the amount of time spent to perform local FFTs through ESSL, "Synchronous Communication" counts the time spent to initiate communication or wait for its completion, "In Memory Data Transfers" counts the time to pack and unpack data, "NAS Other" measures the time for the other parts of

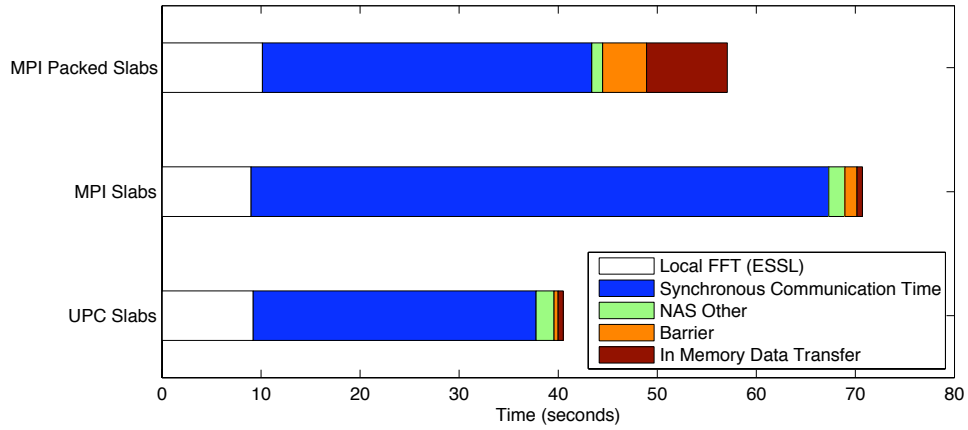

Figure 8: NAS FT Performance Breakdown on 16k cores. Grid Size=$4096 \times 2048 \times 2048$

the NAS FFT benchmark besides the 3D FFT (initial setup, local evolve computation and final checksum), and "Barrier" measures the time spent in barriers.

Notice that the MPI Packed Slabs algorithm spends a significant amount of the runtime in data movement to pack and unpack the slabs. This reference version was written so that all the FFTs are computed on unit stride pencils. However, the other two implementations were written so that the data could be read from a strided location, which most FFT libraries support. Since the cost of memory copies is expensive on the BG/P compute node due to the relatively slow cores, it is better to employ a strided FFT than to rely on data copying to allow a unit strided FFT.

As the data also show, the primary difference in execution time is the time spent on communication. At 16,384 cores the Packed Slabs algorithm induces 128KB messages and the Slabs algorithm induces 8KB messages. From the flood bandwidth microbenchmark (Figure 2 in Section 3) alone, one expects MPI Packed Slabs to complete the communication significantly faster than either Slabs implementation due to the higher bandwidths for 128KB (vs 8KB) messages. However, the actual result shows UPC Slabs spending less synchronous time in communication. This is consistent with our intuition and previous findings that the addition of communication/computation overlap in the Slabs implementations leads to a net reduction in synchronous communication costs. The cost of unoverlapped communication in MPI Slabs, however, is roughly twice that of UPC Slabs. This again shows how the software overheads of MPI are preventing the degree of overlap that UPC can achieve.

Overall, these results show the performance improvement achieved through the cumulative effects of allowing communication/computation overlap (which is absent in the Packed Slabs algorithm) and the switch from MPI to the one-sided communication of UPC over GASNet to maximize overlap.

7. Related Work

Due to the importance of the FFT algorithm, there is much related work on analyzing FFT performance for a variety of networks. Many have focused specifically on torus and mesh style networks [15]–[17]. While these efforts have tended to examine a 1D FFT some of the same principles of optimizing the communication patterns still apply. Our most significant contribution relative to this work is the scale to which we present the data and our analysis of the effectiveness of overlap.

There is significant prior work focused on the effectiveness of communication/communication overlap as well as communication/computation overlap. A good modern example of this is the work by Danalis et al. [18], who studied in the context of MPI's two-sided communication model. We argue that the one-sided communication model found in UPC provides additional opportunity for overlap of communication and computation. In addition we show the techniques of overlap are especially useful at large scale.

UPC is a relatively new programming language (the first mature specification was standardized in Feb, 2001), and compiler development efforts are underway at a number of major corporations and institutions [19]. Initial results are very promising and show that UPC application performance can be competitive with MPI [20], and offers improved usability and programmer productivity [21].

The Argonne ALCF group have submitted performance results for the 1D FFT [22] on the same BG/P system that we have used. Across 128k cores their best published result is 4.5 Teraflops. The IBM BlueGene group have also submitted performance results for the 1D FFT on BlueGene/L (the predecessor system to BG/P). Across 64k cores their best published result to date is 2.2 Teraflops. Barton et al. [23] also demonstrate that UPC can be scaled to extremely large core counts, and raised scalability questions concerning the design of our Berkeley UPC runtime. Our work validates that

Berkeley UPC scales well to large core counts. Nishtala et. al have also done a study of 3D FFT performance on the BG/L system [24] however their focus was more on the interfaces to the collectives in UPC and did not address the exploitation of communication/computation overlap.

8. Conclusions and Future Work

As demonstrated through the microbenchmarks, GASNet's one-sided communication model is a much better semantic fit to the network hardware on the BG/P, leading to better multi-link bandwidth. In addition we see that the roundtrip latency for a GASNet put or get is about half that for an equivalent ping-ack operation in MPI. This paper shows that our portable Berkeley UPC compiler achieves both scalability and portability. Our runs achieve the highest degree of parallelism to date for our compiler. To the best of our knowledge this is also the first compiler for any PGAS language available for the BG/P.

In order to benchmark overall application scalability we implemented a communication-bound benchmark, the NAS Parallel Benchmark FT, in this one-sided communication model and compared it to two-sided implementations. We compare two distinct algorithms; the first uses only communication/communication overlap and relies on the traditional method of packing data before performing the communication, whereas the second additionally exploits communication/computation overlap. We experiment with both weak and strong scaling of the different algorithms and found that the UPC implementation consistently outperforms the MPI counterparts. We also compare our data against the theoretical upper bound in which we assume computation is free and show the maximum performance given the limit is the bisection bandwidth and show that we are within a factor of two of the maximum possible performance.

Our FT application benchmark achieves 1.93 Teraflops across 16k cores on the BG/P for the UPC benchmark compared to 1.37 Teraflops for the best MPI implementation leading to a 40% improvement in overall application performance. The timing data show that the big difference in execution time arises from the time spent in the communication subsystem.

Our scaling study was bounded at 16k cores as a result of our allocation rather than any fundamental limits in GASNet or UPC. We have every reason to believe that this code can scale out to much higher core counts and future work will explore runs on a larger fraction of the machine. In addition, we will explore these scaling issues on other large parallel machines such as the Cray XT. Future work will also create a Packed Slabs implementation in UPC as well as revising the MPI Packed Slabs code to minimize in-memory data movement. Finally we have run all our experiments with one process per core, and future work will explore the use of one process per node with multi-threading for on-node parallelism.

In conclusion, our data shows that the one-sided communication model found in UPC is a better semantic fit for the BG/P network. We are able to better leverage communication/communication overlap as well as communication/computation overlap to realize linear scaling on an application that is known to be bounded by the performance of the communication subsystem.

Acknowledgements

We would like to thank Michael Blocksome, Douglas Miller, Sameer Kumar and the entire IBM DCMF team for their support in helping us port GASNet to BG/P. This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] "IBM BlueGene/P," IBM, http://www.research.ibm.com/journal/rd/521/team.html.

[2] MPI Forum, "MPI: A message-passing interface standard, v1.1," University of Tennessee, Knoxville, Technical Report, June 12, 1995.

[3] "The Berkeley UPC Compiler," http://upc.lbl.gov, 2002.

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks," *The International Journal of Supercomputer Applications*, vol. 5, no. 3, pp. 63–73, Fall 1991.

[5] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, "Optimizing bandwidth limited problems using one-sided communication and overlap," in *The 20th Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[6] "UPC language specifications, v1.2," Lawrence Berkeley National Lab, Tech. Rep. LBNL-59208, 2005.

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

[8] "GASNet home page," http://gasnet.cs.berkeley.edu/.

[9] "Intrepid system," Argonne Leadership Computing Facility, http://www.alcf.anl.gov/.

[10] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. J. Archer, "The Deep Computing Messaging Framework: Generalized scalable message passing on the BlueGene/P supercomputer," in *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*. New York, NY, USA: ACM, 2008, pp. 94–103.

[11] "MPICH2 web site," http://www.mcs.anl.gov/research/projects/mpich2.

[12] "ESSL User Guide," http://www-03.ibm.com/systems/p/software/essl.html.

[13] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[14] D. Bonachea, "Proposal for extending the UPC memory copy library functions and supporting extensions to GASNet, v1.0," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-56495, October 2004.

[15] C. Y. Chu, "Comparison of two-dimensional FFT methods on the hypercube," in *Proceedings of the third conference on Hypercube concurrent computers and applications*. New York, NY, USA: ACM Press, 1988, pp. 1430–1437.

[16] L. Díaz, M. Valero-García, and A. González, "A method for exploiting communication/computation overlap in hypercubes," *Parallel Computing*, vol. 24, no. 2, pp. 221–245, 1998.

[17] P. N. Swartztrauber and S. W. Hammond, "A comparison of optimal FFTs on torus and hypercube multicomputers," *Parallel Computing*, vol. 27, no. 6, pp. 847–859, 2001.

[18] A. Danalis, K.-Y. Kim, L. Pollock, and M. Swany, "Transformations to parallel codes for communication-computation overlap," in *Supercomputing 2005*, November 2005.

[19] "UPC consortium home page," http://upc.gwu.edu/.

[20] T. El-Ghazawi and F. Cantonnet, "UPC performance and potential: A NPB experimental study," in *Supercomputing2002 (SC2002)*, November 2002.

[21] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi, "Productivity Analysis of the UPC Language," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[22] "HPC challenge benchmark results," http://icl.cs.utk.edu/hpcc/hpcc_results.cgi.

[23] C. Barton, C. Casçaval, G. Almási, Y. Zheng, M. Farreras, S. Chatterje, and J. N. Amaral, "Shared memory programming for large scale machines," *SIGPLAN Not.*, vol. 41, no. 6, pp. 108–117, 2006.

[24] R. Nishtala, G. Almasi, and C. Cascaval, "Performance without pain = productivity: data layout and collective communication in UPC," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2008, pp. 99–110.

Appendix

Node Count	Core Count	X	Y	Z	T	TY × TZ
64	256	4	4	4	4	16 × 16
128	512	4	4	8	4	16 × 32
256	1024	8	4	8	4	32 × 32
512	2048	8	8	8	4	64 × 32
1024	4096	8	8	16	4	64 × 64
2048	8192	8	8	32	4	64 × 128
4096	16,384	8	16	32	4	128 × 128

Table 2: BlueGene/P Run Configurations

Algorithm 1 FFT Packed Slabs

1: Let myPlane = MYTHREAD / TY
2: Let myRow = MYTHREAD % TY
3: Let teamY = all threads who have same value of myPlane
4: Let teamZ = all threads who have same value of myRow
5: **for** $plane = 0$ to $\frac{NZ}{TZ}$ **do**
6: **for** $row = 0$ to $\frac{NY}{TY}$ **do**
7: do 1D FFT of length NX
8: **end for**
9: **end for**
10: Pack the slabs together
11: Do Alltoall on teamY
12: Unpack the slabs to make Y dimension contiguous
13: **for** $plane = 0$ to $\frac{NZ}{TZ}$ **do**
14: **for** $row = 0$ to $\frac{NX}{TY}$ **do**
15: do 1D FFT of length NY
16: **end for**
17: **end for**
18: Pack the slabs together
19: Do Alltoall on teamZ
20: Unpack the slabs to make the Z dimension contiguous
21: **for** $plane = 0$ to $\frac{NY}{TZ}$ **do**
22: **for** $row = 0$ to $\frac{NX}{TY}$ **do**
23: do 1D FFT of length NZ
24: **end for**
25: **end for**

Most FFT libraries such as ESSL [12] and FFTW [13] provide the ability to perform multiple strided FFTs with one call to the library. This enables memory hierarchy optimizations to be performed across multiple FFTs rather than just one. For example lines 5-9 in Algorithm 1 and lines 6-9 in Algorithm 2 can be realized as one call to the underlying serial FFT library.

Algorithm 2 FFT Slabs

1: Let myPlane = MYTHREAD / TY
2: Let myRow = MYTHREAD % TY
3: For MPI Prepost all recvs for First Communication Round
4: BARRIER
5: **for** $plane = 0$ to $\frac{NZ}{TZ}$ **do**
6: **for** $row = 0$ to $\frac{NY}{TY}$ **do**
7: do 1D FFT of length NX
8: **end for**
9: Pack the data for this plane
10: **for** $t = 1$; $t \leq TY$; $t = t + 1$ **do**
11: initiate communication to thread $myPlane \times TY + (t + myRow)\%TY$
12: **end for**
13: **end for**
14: Wait for all communication to finish
15: Unpack all the data to make Y dimension contiguous
16: For MPI Prepost all recvs for Second Communication Round
17: BARRIER
18: **for** $plane = 0$ to $\frac{NZ}{TZ}$ **do**
19: **for** $row = 0$ to $\frac{NX}{TY}$ **do**
20: do 1D FFT of length NY
21: **end for**
22: Pack the data for this plane
23: **for** $t = 1$; $t \leq TZ$; $t = t + 1$ **do**
24: initiate communication to thread $((t + myPlane)\%TZ) \times TY + myRow$
25: **end for**
26: **end for**
27: Wait for all communication to finish
28: Unpack all the data to make Z dimension contiguous
29: **for** $plane = 0$ to $\frac{NY}{TZ}$ **do**
30: **for** $row = 0$ to $\frac{NX}{TY}$ **do**
31: do 1D FFT of length NZ
32: **end for**
33: **end for**