Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap

Christian Bell[1,2], Dan Bonachea[1], Rajesh Nishtala[1], Katherine Yelick[1,2]
{*csbell,bonachea,rajeshn,yelick*}*@cs.berkeley.edu*
Computer Science Division, University of California at Berkeley[1]
Computational Research Division, Lawrence Berkeley National Laboratory[2]

Abstract

Partitioned Global Address Space languages like Unified Parallel C (UPC) are typically valued for their expressiveness, especially for computations with fine-grained random accesses. In this paper we show that the one-sided communication model used in these languages also has a significant performance advantage for bandwidth-limited applications. We demonstrate this benefit through communication microbenchmarks and a case-study that compares UPC and MPI implementations of the NAS Fourier Transform (FT) benchmark. Our optimizations rely on aggressively overlapping communication with computation but spreading communication events throughout the course of the local computation. This alleviates the potential communication bottleneck that occurs when the communication is packed into a single phase (e.g., the large all-to-all in a multidimensional FFT). Even though the new algorithms require more messages for the same total volume of data, the resulting overlap leads to speedups of over $1.75\times$ *and* $1.9\times$ *for the two-sided and one-sided implementations, respectively, when compared to the default NAS Fortran/MPI release. Our best one-sided implementations show an average improvement of* 15% *over our best two-sided implementations. We attribute this difference to the lower software overhead of one-sided communication, which is partly fundamental to the semantic difference between one-sided and two-sided communication. Our UPC results use the Berkeley UPC compiler with the GASNet communication system, and demonstrate the portability and scalability of that language and implementation, with performance approaching 0.5 TFlop/s on the FT benchmark running on 512 processors.*

1 Introduction

The one-sided communication model is typically viewed as advantageous for unstructured computations and irregular communication patterns in terms of performance and programmability [10]. One-sided communication is the primary mode of communication in Partitioned Global Address Space (PGAS) languages and has been integrated into the second revision of the Message-Passing Interface (MPI). Although these two one-sided models differ semantically and operationally in the mechanisms used to enforce synchronization, they all aim to improve performance by decoupling synchronization from data movement. While the benefits of the one-sided model are most pronounced for small message data transfers where the synchronization and software overhead is not amortized by transfer time, we argue that one-sided communication can also be beneficial in improving the performance of applications that are bandwidth-bound. In particular, we show

that replacing large bulk transfers with more frequent smaller messages allows UPC's implementation of the one-sided model to outperform an MPI two-sided implementation on bandwidth-bound operations.

Conventional wisdom holds that communication costs in applications can be minimized by sending a small number of large messages, especially for cluster networks where the per-message cost can be high. This practice is based on the observation that large messages have traditionally been needed to run at performance near the peak bandwidth. Many applications therefore adopt a bulk-synchronous communication paradigm, dividing program execution into clearly separated global phases of computation and communication. Two recent trends are challenging that wisdom: network vendors are increasingly offloading communication protocol processing onto network hardware; and the emergence of one-sided communication offers unique opportunities to further reduce overhead by decoupling synchronization from data transfer. In some applications the computational data dependencies could actually permit a large fraction of the communication to be initiated earlier or completed later. If one can find sufficient independent computation to overlap the communication latency such that negligible time is spent waiting for communication completion, then the primary cost of communication becomes the software overhead required to initiate and synchronize non-blocking communication. Once the latency component of communication has been entirely overlapped in this manner, the transfer bandwidth achieved by an individual message becomes less important, making it feasible to trade off smaller message size for improved overlap.

This paper explores the hypothesis that communication can be effectively overlapped in bandwidth-sensitive applications using one-sided communication. We use a series of communication microbenchmarks and the NAS Fourier Transform (FT) Parallel Benchmark as a case study to compare one-sided and two-sided communication models in order to validate this hypothesis. Our approach is to spread the communication of the transpose step (i.e., the all-to-all) required by the 3D FFT throughout the computation of the local slabs and send the data as soon as it is ready. This optimization alleviates bottlenecks in the communication and aggressively overlaps the communication behind the computation. Although the total volume of data communicated is consistent across all the algorithms considered, the number of messages per thread increases from $O(T)$ in the all-to-all based implementation to $O(n)$ where T is the total number of threads and n is the size of the maximum dimension. The promising results motivate an even more aggressive overlap strategy that sends $O(\frac{n^2}{T})$ messages while still keeping the total volume of data constant. As the results will show, the two-sided implementations that used the $O(n)$ algorithm achieve nontrivial gains (over $1.75\times$) compared to the traditional all-to-all based implementation. However the versions which utilize one-sided communication achieve an additional speedup using the algorithm that sends $O(\frac{n^2}{T})$ messages. These algorithms are consistently the best performers with speedups of up to $1.9\times$ over the traditional all-to-all based implementations. We also implement the $O(\frac{n^2}{T})$ version in MPI, but show they cannot achieve the same performance benefits as UPC due to the higher communication overhead. We argue that this added overhead is, at least in part, fundamental to the two-sided model.

We use Berkeley UPC [6] and MPI v1.1 [39] as representatives of the one and two-sided communication models, respectively. Although the MPI 2.0 standard [38] adds a one-sided communication interface, this interface has several semantic limitations that hinder its use in practice [9], and therefore we do not consider it further in this paper. Instead, we use use the Berkeley UPC implementation with GASNet [7] as our representative for one-sided communication. UPC [50], along with Co-Array Fortran [43] and Titanium [29], are modern examples of the Partitioned Global Address Space (PGAS) language approach to parallel computing. They expose language semantics that induce a one-sided communication model: processors logically issue direct loads and stores to the memory of remote processors using reads and writes to logically shared variables. GASNet is a portable, high performance communication compilation target that exposes a rich set of initiation and completion mechanisms for one-sided operations that enables the client (typically library and compiler writers) to compose flexible communication patterns and retain control over their synchronization. Our one-sided implementations of the benchmark are written from scratch in UPC, and leverage some minor library extensions to UPC for non-blocking bulk memory operations provided by the Berkeley UPC compiler.

Our two-sided versions are written in Fortran and C with MPI v1.1, starting with the standard NAS release of the FT benchmark.

The remainder of this paper is organized as follows: In Section 2 we give a brief introduction to PGAS languages and UPC. Sections 3 and 4 present the GASNet communications layer and show its bandwidth and latency performance compared to MPI. Section 5 goes into detail about how we leverage one-sided communication to obtain significant performance improvements over MPI in the NAS FT.

2 Partitioned Global Address Space Languages and UPC

Partitioned Global Address Space languages combine a Single Program Multiple Data (SPMD) programming model with a global address space, which is logically partitioned to give each thread a portion of shared memory to which it has affinity. The study in this paper is based on Unified Parallel C (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. In UPC's SPMD model, a fixed number of threads are created at program startup, and every thread runs the same program. Each thread has both a space for private local memory and some partition of the shared space to which it has *affinity*. A private object may only be accessed by its corresponding thread, whereas all threads can read or write any object in the shared address space. The partitioning of the shared space into regions with logical affinity to threads allows programmers to explicitly control data layout, which is then used by the runtime system to map threads and their associated data to processors: on a distributed memory machine, the local memory of a processor holds both the thread's private data and the shared data with affinity to that thread.

There are many commercial and open-source compilers available for UPC [6, 16, 28, 32, 40]. In this paper we used the portable, high-performance Berkeley UPC compiler [6], which translates UPC to ISO-compliant C using a compiler based on the Open64 infrastructure [45]. The translator performs both serial and parallel optimizations [13, 30, 54], although in this paper we will work with applications that are carefully hand-tuned and therefore do not take much advantage of the high-level optimizations. On a shared memory machine, accesses to the UPC shared address space translate into conventional load/store instructions. On distributed memory machines, which are of interest in this paper, such accesses translate into calls to the Berkeley GASNet layer [7]. Some of the application-level optimizations presented in this paper make use of Berkeley-specific extensions to the UPC language [8] and although these extensions are not part of the current UPC language specification, the results in this paper demonstrate their benefits and motivates their likely inclusion in the next language revision.

3 GASNet Communications Subsystem

GASNet provides a portable, language-independent communication interface designed as a compilation target for PGAS languages. GASNet delivers communication performance very close to the raw hardware peak across many interconnects, effectively leveraging platform and network-specific features such as RDMA support and block transfer engines.

Figure 1 illustrates the basic abstraction stack of the Berkeley UPC, GCC/UPC+UPCR [32] and Titanium [49] compilers over GASNet.

The GASNet API provides point-to-point data transfers that are fully one-sided and decoupled from inter-thread synchronization, with no relative ordering constraints between outstanding operations (in contrast to other one-sided communication interfaces such as ARMCI [42]). GASNet's point-to-point communication API includes simple blocking gets/puts, and several flavors of non-blocking data transfers with a flexible and expressive set of synchronization primitives crafted to support sophisticated communication optimizations. The GASNet implementation is designed in layers for portability: a small set of core functions constitute the basis for portability, and there is a reference implementation of the complete API written entirely in terms of

this core. In addition, the implementation for a given network (the *conduit*) can be tuned by implementing any appropriate subset of the general functionality directly upon the hardware-specific primitives, bypassing the core-based reference implementation. Our research has shown that the layered design approach is effective at providing robust portability as well as high-performance, with UPC performance comparable to vendor-provided compilers on architectures ranging from loosely-coupled clusters with a near-commodity network [12] to tightly-coupled MPP systems with a hardware-supported global memory system [5].

The GASNet interface has been natively implemented on Myrinet (GM) [41] , Quadrics QsNetI/QsNetII (Elan3/4) [46], InfiniBand (Mellanox VAPI) [31, 37], IBM SP Colony/Federation (LAPI) [33] , Dolphin (SISCI) [22], Cray X1 (shmem) [47] and SGI Altix (shmem) [1]. Aside from these high-performance instantiations of the GASNet interface, there are also fully portable GASNet conduits over MPI 1.1 (for any MPI-enabled HPC system not natively supported), GASNet on UDP (for any TCP/IP network, eg. Ethernet), and GASNet for shared-memory SMP's lacking interconnect hardware. Our GASNet implementation is written in standard C and is very portable across architectures and operating systems – thus far it has been successfully used on over fifteen different CPU architectures, twelve different operating systems, and twelve different C compilers, and porting existing GASNet conduits to new UNIX-like systems is nearly effortless. See [26] for further implementation details.

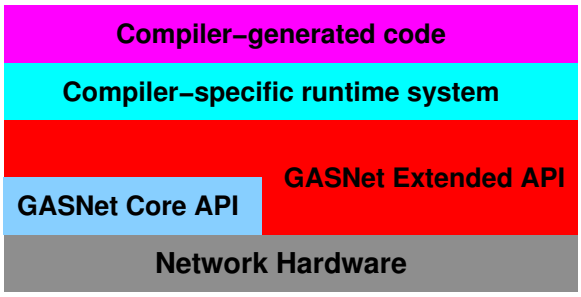

Figure 1. GASNet Communications System: the narrow, AM-based Core API implements the entire system but can be bypassed by the Extended API to exploit native hardware features

4 Performance Advantages of One-Sided Communication in Microbenchmarks

Partitioned Global Address Space languages are sometimes considered suitable only for shared memory hardware such as Symmetric Multiprocessors (SMPs), Distributed Shared Memory machines (e.g., the SGI Altix [1]), or machines with global address space support integrated into the processor (e.g., the Cray X1 [17]). However in this paper we demonstrate that the one-sided communication model underlying these languages is also a more effective match to modern cluster network hardware than two-sided message passing interfaces such as MPI.

The disadvantages of the MPI two-sided message-passing model are summarized in the following three points:

- Messages sends and receives must be matched to complete a transfer. The implementation is responsible for matching the MPI communicator, message tag and sender id between the sender and receiver, and the overhead of this matching can impose a significant performance penalty for small and medium sized messages.

- MPI guarantees point-to-point message ordering, despite the fact that many current and future HPC networks lack point-to-point ordering guarantees in hardware. Other studies [34, 35, 53] have shown there can be a non-trivial cost associated with enforcing ordering semantics upon fundamentally unordered network hardware.

- The semantic requirement for active participation from application-level code on both sides of the communication implies that observed latency in an MPI application may be significantly longer than predicted by a best case scenario - i.e., an application that is inattentive to the network may perform poorly even on a system with low best-case MPI latency.

High-quality MPI implementations on cluster hardware generally use a combination of algorithms to provide the required message-matching semantics and also provide good performance over a large range of message sizes. The *eager* algorithm (which is generally used for small messages) optimistically sends the data and messaging metadata to an anonymous buffer on the target process, which later performs message matching and copies the data to the user buffer. This approach minimizes the wire-time latency, but imposes CPU and memory bus overheads for the extraneous data copy operation and hence is unsuitable for sufficiently large messages where the copy costs would dominate. The *rendezvous* algorithm (generally used for larger messages) initially sends only the metadata to the remote process, which performs the matching and later initiates a zero-copy transfer of the data. This approach minimizes data copying overheads, but imposes the latency of at least one additional roundtrip on the wire, and hence is unsuitable for small messages.

A key advantage to the one-sided communication model is that no such tradeoff is required, because the initiator always provides complete information describing the data transfer to be performed. There are no overheads imposed by matching or synchronization semantics, and the implementation is free to perform the data transfer using the most efficient mechanism available. On modern cluster networks, this usually translates to the use of Remote Direct Memory Access (RDMA) hardware support, which allows efficient remote access without intervention by the remote host CPU.

4.1 Latency Advantages of One-Sided Communication

One of the key advantages of a one-sided communication model is that all relevant information about a communication operation is provided by the initiator – information is never required from the remote user code to complete message delivery. GASNet's one-sided data transfer operations are completely decoupled from inter-process synchronization, enabling data transmission to often begin immediately upon operation initiation (subject only to network congestion) and proceed autonomously from any action at the target process. For example, in a remote put operation the initiator can always transmit the remote destination address along with the data, providing a close semantic match to the requirements of high-bandwidth, zero-copy RDMA hardware. Conversely, a similar operation in MPI message-passing requires somehow retrieving the destination address from a matching receive operation posted by the target user process (possibly at some point in the future) before the transfer can be completed. This matching operation often dictates the performance of MPI implementations, and vendors consequently invest significant effort in optimizing its cost.

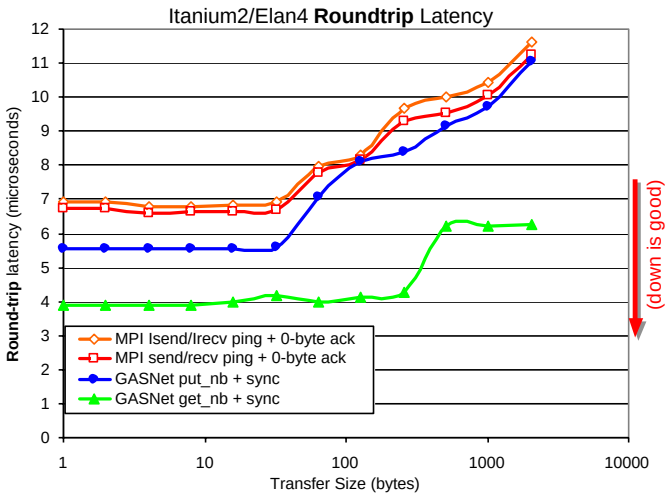

Figure 2. Latency of GASNet vs. MPI on Quadrics Elan4

The FT application is a bandwidth-limited problem, however the effective use of overlap in our implementations depends crucially on the per-message CPU overheads associated with initiating and completing non-blocking operations. These overheads can be difficult to measure directly, however comparisons of small-message latency performance on a given network can provide insight into the effects of software overhead, because it tends to comprise a large fraction of small-message latency. In network processor-based solutions such as Quadrics QsNetI/II, the network interface is capable of autonomously completing MPI message matching operations. Such approaches generally outperform host-based solutions that require attention from the remote host CPU (e.g., Myrinet and InfiniBand). We expect other networks to follow the lead of Quadrics in this type of MPI protocol offload, and therefore consider the Quadrics network in this section for our latency

analysis, since it provides the best-case for MPI message-passing latency.

Figure 2 compares the *round-trip* latency performance over varying data transfer size of GASNet's Quadrics/Elan conduit with Quadrics MPI on an Itanium2/Elan4 system. The GASNet tests measure the round-trip latency to issue a GASNet put or get operation and block for round-trip completion. The MPI tests measure the round-trip latency for a ping-pong test where the initiator sends a message of the given size, and the remote side issues a 0-byte acknowledgement message. Performance is measured using both the blocking (`MPI_Send/MPI_Recv`) and non-blocking (`MPI_Isend/MPI_Irecv`) MPI message passing primitives.

The Quadrics network hardware provides support for offloading MPI message matching overheads onto the NIC processor via the Elan Tports interface, freeing the host processor from most duties associated with the MPI message queue. However as evidenced by the figure, there is still a pronounced latency difference between this interface and the performance achievable through the lighter-weight, raw RDMA elan interfaces (`elan_put/elan_get`) targeted by the GASNet put/get implementation on Quadrics. One-sided communication is a better semantic match to RDMA-enabled hardware, and thus induces lower software overhead and delivers better latency performance for small and medium-sided messages.

4.2 Flood Bandwidth Advantages of One-Sided Communication

Figure 3 compares the flood bandwidth performance over varying message size of GASNet's InfiniBand/VAPI conduit with OSU MVAPICH [36], an extensively tuned implementation which is widely considered to be the best available MPI on InfiniBand. GASNet consistently and significantly outperforms MVAPICH on InfiniBand because the GASNet one-sided put/get semantics are fundamentally a better match for the capabilities of the underlying RDMA hardware than MPI's two-sided message passing semantics. GASNet's put/gets turn into simple, fully one-sided RDMA operations in the common case, and therefore reap the hardware peak performance, whereas MPI pays in performance for enforcing MPI's ordering and message matching semantics. GASNet's good performance on pinning-based RDMA network hardware such as InfiniBand and Myrinet can also be attributed to Firehose [4], our novel distributed algorithm for efficiently managing memory registration on these networks. Firehose effectively delegates the control of registration resources to the RDMA initiators, successfully exposing one-sided, zero-copy communication as a common case, while minimizing the number of host-level synchronizations required to support remote memory operations and amortizing the cost of synchronization and pinning over multiple remote memory operations. Another notable feature of the InfiniBand flood bandwidth is the performance drop-off beyond 256KB data transfer size - this artifact is due to a performance bug in the Mellanox hardware, which the GASNet/VAPI conduit has been tuned to avoid.

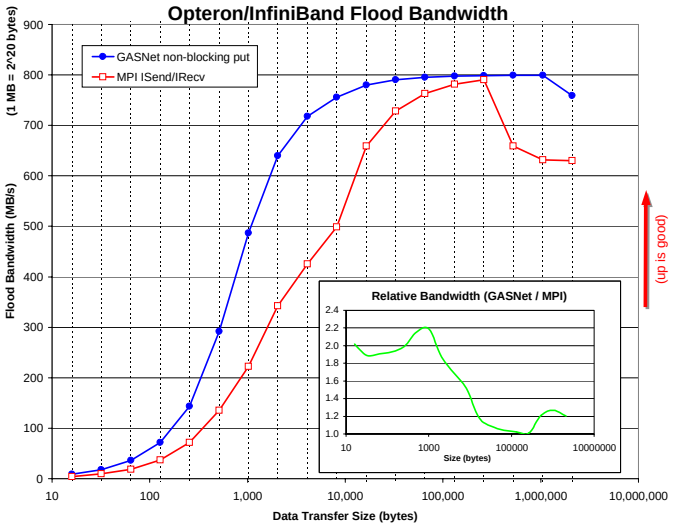

Figure 3. Bandwidth of GASNet vs. MPI on InfiniBand

This semantically-induced bandwidth performance gap between MPI's message passing and GASNet's one-sided communication is observable across a number of modern RDMA-enabled cluster interconnects. Figure 4 compares the flood bandwidth achievable with GASNet's one-sided put/get RMA primitives against `MPI_Isend/MPI_Irecv` message-passing for a 4KB and 512KB data transfer size across a number of production cluster supercomputers (as detailed in the appendix). The bar height is normalized to the theoretical peak bandwidth of the system (a minimum of the interconnect link speed and the I/O bus speed), and absolute

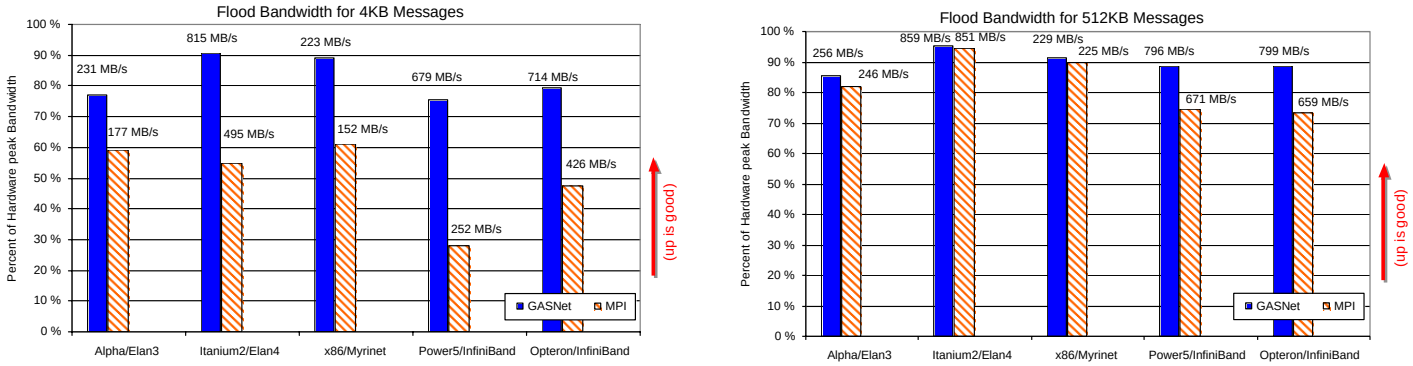

Figure 4. Flood Bandwidth of GASNet vs. MPI for 4KB and 512KB Messages

bandwidth performance is also shown. The figure demonstrates that for "large" messages, both one-sided and message-passing communication mechanisms typically saturate to similar peak bandwidth values (the only exception being due to the InfiniBand hardware performance bug described in the previous section). However, one-sided communication consistently provides a significant performance advantage at "mid-ranged" sizes of about 1KB - 100KB, where the raw payload transmission times are often too short to fully amortize or overlap the MPI message matching overheads. Similar patterns have been observed on other systems for such "mid-ranged" message sizes. Recent studies [27] of MPI usage across a range of real-world scientific applications have found that "mid-ranged" message sizes dominate many production applications and become even more prevalent at larger scales, motivating the importance of this range of message sizes. As described in subsequent sections, these message sizes are also often crucial in achieving efficient communication overlap.

5 Optimizing Bandwidth-Limited Applications

In this section we consider a problem that is often hailed as the canonical example of a problem limited by bisection bandwidth, the 3D FFT. Superficially, none of the latency advantages of a one-sided model would appear to be relevant, because the key to performance is the efficiency of a cross-processor transpose that happens between phases of the FFT. As typically coded, the messages are all large and have a fixed size that is known in advance, since it is a simple function of the problem size. The FFT kernel is used in many scientific applications and is a critical operation in its own right, but it also reflects a more general class of algorithms that are a challenge to scalability and performance. Machines with inadequate bisection bandwidth typically suffer relative to those with full crossbars on applications requiring a large volume of many-to-many or all-to-all communication [44].

5.1 NAS FT Benchmark

The NAS FT benchmark [2] implements a partial differential equation using a series of repeated forward and inverse Fourier Transforms over three dimensions. Since all dimensions are represented linearly in memory, the sets of 1-D FFTs must be transposed in memory in order to calculate the complete 3-D FFT, which translates into three sets of 1-D FFTs followed by transpositions. The data can be decomposed across parallel threads either in planes along one of the dimensions (1-D layout) or in two dimensional slices (2-D layout) when the number of threads exceeds the number of planes. As implemented in the original NAS Benchmark, two of the dimensions are computed and transposed locally while the remaining single dimension incurs a global exchange among all processors, after which the remaining FFT dimension can be computed. The reference implementation of the NAS FT Benchmark is realized in Fortran and the communication uses MPI message-passing. It has undergone several revisions since its original release and can be considered a mature benchmark. The only significant communication step in the 1-D layout version of the problem is implemented

by the MPI All-to-all collective, a bulk communication operation where each thread exchanges its portion of the 3-D FFT with every other thread. The existing exchange (All-to-all) version of this benchmark separates communication and computation into distinct phases: after computing the FFT in one dimension over all its planes, every thread locally transposes the computed data into an ordering suitable for the exchange operation, after which the data is re-transposed to complete the remaining FFTs. The communication can be placed between a local 1D-FFT and a local 2D-FFT or vice-versa and while the operation requires a transpose, the local 2D-FFT makes use of a cache-blocked algorithm to compute both the unit and non-unit stride dimensions.

5.2 Expressing NAS FT with One-sided Communication in UPC

If modeled according to the original NAS Fortran/MPI implementation, a straightforward one-sided UPC implementation could perform the exchange using either point-to-point bulk put operations or alternatively use the collective operations recently added to the UPC language [51]. Since large exchange operations are bandwidth-bound and are not noticably optimized beyond the performance of point-to-point communication, we expect UPC performance to at least match the performance of the original NAS version given the point-to-point performance results presented in section 4. Whereas the data movement and communication pattern are similar in both the one-sided and two-sided variants of this implementation approach, one-sided communication only differs in that each communication call provides complete information to the communication library. Unlike two-sided message-passing where the target thread must provide the target address, one-sided communication maps well to networks that can autonomously delivery data – the entire communication can proceed without involving the target processor. However, the size of the individual messages and overall communication in the exchange is sufficiently large to hide the implied synchronization costs imposed by the two-sided model.

5.3 Optimizing NAS FT on Modern Networks

Issuing a single collective communication to globally exchange all FFT planes makes use of large messages with the goal of maximizing the available bandwidth and simplifies the programmer's task in observing local dependencies: two of the three FFTs are complete prior to the exchange and the last FFT can begin once the exchange completes. The performance downside, however, arises from the increasing monetary cost and complexity in providing full network bisection bandwidth as the amount of nodes involved in the exchange increases. Networks that do not provide full bisection bandwidth at high node counts can benefit from any operation that can replace or at least amortize the cost of a global exchange operation. Those that do provide full bisection bandwidth at large scale can still reduce the cost of a global exchange if the communication network supports asynchronous communication, because portions of the exchange communication can be hidden behind computation. Since our target networks support such operations, our proposed approach is to decompose the FFT computation and communication into smaller pieces that permit overlap. In doing so, we have employed and implemented the FT benchmark by decomposing the two dimensional planes of the 1-D layout into small and even smaller contiguous pieces. These implementations are summarized by the following two approaches:

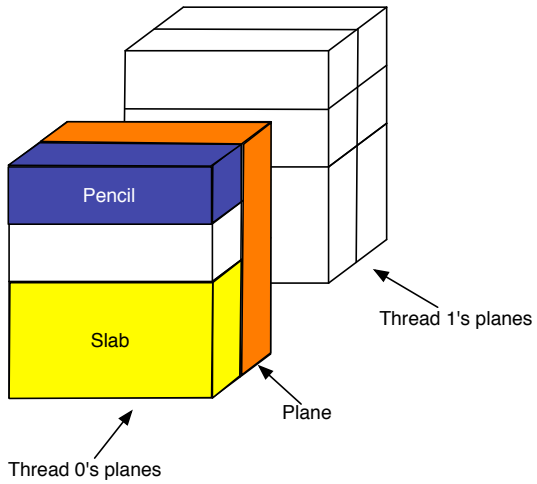

Figure 5. FT data decomposition for a 4x4x4 cube with 2 threads

- **Overlap Slabs**. Overlapping slabs is a method of decomposing the 3D-FFT to reduce the amount of time spent in communication-bound operations by overlapping the communication cost of sending previously computed slabs with the computation of remaining slabs. A slab is defined to be the portion of each FFT plane that has affinity to a single thread and will be sent to a single remote thread, such that communication incurs only a single point-to-point operation (see Figure 5). In the original NAS implementation, slabs destined for each remote thread are packed to be made contiguous prior to the global exchange operation.

- **Overlap Pencils**. The pencils-based approach is similar to the slabs-based approach, except that it further reduces the granularity of communication and overlap, sending more and smaller-sized messages. Point-to-point messages of a single FFT row are sent while computation happens over the next row. Figure 6 shows the major communication and computation steps in partitioning the 3-D FFT. Since the cube's decomposition assigns 2D-planes to each thread, each thread begins by computing the first of its FFT computations without communication in the non-contiguous dimension. Following a transpose, the set of contiguous FFTs is computed, and each row is computed as the previously computed row is sent to the new owning thread. The next owning thread is thread who requires affinity to the row as a result of transposing the cube for the second (and last) time. A barrier signals the point at which each thread has finished its row computations, which allows the final FFT dimension to be computed following a local transpose.

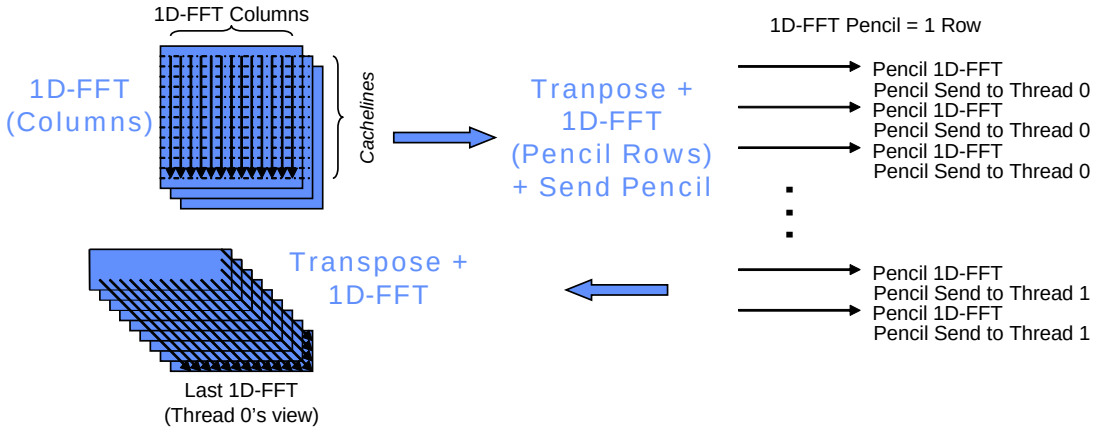

Figure 6. 3D-FFT by communicating overlapping pencils

Table 1 summarizes implementation algorithms and provides a measure of overlap efficiency as a ratio of the data that can be overlapped in one plane over the data that is computed in one plane based on the dimension of the FFT cube (n^3). The single exchange approach used by the default NAS FT implementation is represented by *UPC Exchange*. Both overlap algorithms have communication startup costs where the first and last units of communication, either slabs or pencils, cannot be overlapped with additional computation. Also shown is the number of total network messages sent by each thread in order to complete the global exchange. As can be expected, the finer the data decomposition, the greater the message count (up to a factor of the square of the input cube's dimensions for the Pencils algorithm). Also, since the volume of the data exchanged remains constant across all versions of the benchmark, sending more messages also implies that the messages are smaller.

6 Results

Performance results in this section are shown for three popular RDMA-based interconnect technologies: InfiniBand, Quadrics/Elan and Myrinet. The FT benchmark is fairly intensive in its floating point requirements

FT Implementation	*# Messages per thread*	*Per-plane Overlap Efficiency*
UPC Exchange	THREADS	0
UPC Overlap Slabs	$\text{THREADS} * \frac{n}{\text{THREADS}}$	$1 - \frac{2}{n}$
UPC Overlap Pencils	$\frac{n^2}{\text{THREADS}}$	$1 - 2 * \frac{\text{THREADS}}{n^2}$

Table 1. Summary of UPC FT Algorithms as a function of the dimension of the FFT cube (n^3)

and since each system uses different processors, the overall MFlops results that determine the time to solve a series of 3D FFTs cannot be used as a basis to compare interconnect technologies. Rather, the variety of interconnect solutions and generations are graphed to demonstrate that the communication optimizations employed by FT UPC generalize to a large class of high performance system configurations.

In order to measure the effectiveness of our approach over the mentioned RDMA-based networks, we wrote UPC and MPI-C versions of global exchange, Overlapped Slabs and Overlapped Pencils with the goal of comparing them to the original NAS Fortran/MPI Benchmark. To prevent performance offsets emanating from serial FFT performance variance across different languages, all benchmarks use a 1-D FFT decomposition of the domain and compute the 1-D FFTs using the FFTW package [25] (which consistently outperforms the Stockholm FFT used in the original NAS Fortran implementation by a small margin on all the platforms we encountered). The MPI-C and UPC versions of the benchmark are similar except for the language and/or library features they employ for non-blocking communication. UPC uses GASNet's non-blocking operations whereas MPI-C uses `MPI_Isend` combined with preposting of receive buffers well in advance such that no overheads resulting from unexpected messages are incurred by the underlying MPI communication layer. All but the original default Fortran with MPI version of the benchmark employ a configurable padding parameter that allows one of the power-of-two dimensions in the FT class to be padded in order to more effectively use the memory hierarchy. We have found the padding to be most effective when computing FFT over the non-unit-stride dimensions.

The MPI wallclock timer is used to profile the MPI implementations, and hardware cycle counters of sub-microsecond accuracy are used to time the UPC implementations. The data reported is the maximum performance across five trials - the standard deviation across the various trials was very low.

6.1 UPC Non-blocking Slabs and Pencil Results

The first set of results in Figure 7 show the performance speedup of the UPC implementation of Exchange, Slabs and Pencils over the original NAS Fortran implementation. Clearly, the approach of overlapping communication and computation over smaller units of the FFT is beneficial on all the tested interconnects and actually improves over newer generations of the same interconnect (Elan/Quadrics). Average speedups are on the order of 80%, with the most recent interconnects (InfiniBand and Elan4) showing 90% speedups. These speedups essentially demonstrate that overlapping slabs and pencils can produce higher overall efficiency on systems that allow networking and computational resources to be used concurrently and independently.

In comparing UPC Pencils to Slabs, all platforms show Pencils to be slightly faster. While the improvement is never beyond 10%, Pencils notably differ from Slabs by sending many more messages. This is contrary to the typical approach of sending fewer larger messages and validates that the GASNet communication library can effectively maintain or improve communication performance as it decreases the message size and increases the messaging rate.

In order to measure Pencil's impact in increasing the messaging rate, we identified two areas within the benchmark where Slabs and Pencils produced noticable differences across all platforms. The remaining 1-D FFT that occurs after all rows are communicated requires each received row to be reordered for a non-unit stride FFT. With Slabs, consecutive elements in the non-unit stride appear on different slabs, whereas Pencils allow the rows to be sent into an ordering that anticipates the remaining non-unit stride FFT and effectively reduces the stride to the size of a pencil. On systems such as the Alpha with a small TLB and/or high TLB miss

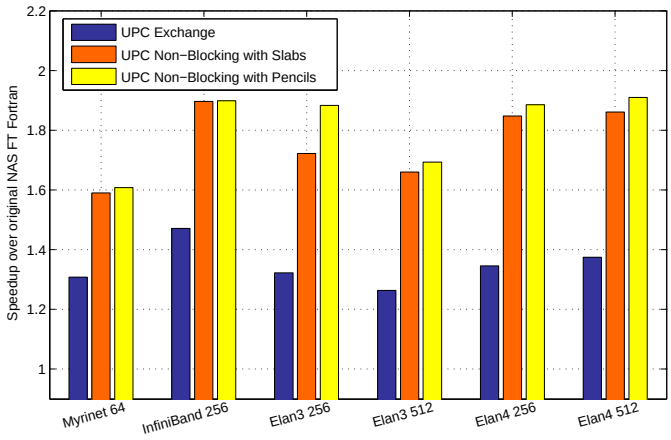

Figure 7. Speedup of UPC Exchange, Overlap Pencils and Overlap Slabs over Original Fortran FT

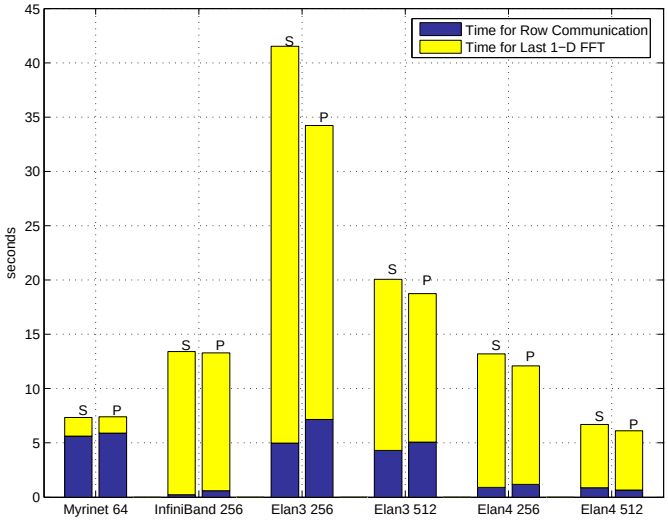

Figure 8. UPC Slabs (S) and UPC Pencils (P): Communication overhead and resulting computation performance

penalty, the smaller stride improved computational performance by reducing pressure on the memory system and minimizing the number of TLB misses. Pencils therefore reduces the amount of time spent computing the final FFT, although this improvement comes at a cost of a higher total message count. These combined costs are shown in Figure 8 for each platform, which illustrates that the time Pencils recovers in reordering is greater than the increased overhead of sending more messages. The overheads imposed by the increase in message rate and the decrease in message size are well amortized by the network through the use of the GASNet communication library. These overheads include operation initiation and completion costs as well as the inter-operation gap which represents the minimum amount of time between two consecutive message injections. These overheads are kept relatively low considering that many more messages are sent in Pencils compared to Slabs and that these messages are actually much smaller. For example, for FT's Class D problem size at 256 processes, each process sends 1024 messages of 128KBytes with Slabs and 8192 messages of 16KBytes with Pencils.

6.2 UPC and MPI Comparative Results for Overlap

In order to evaluate the effectiveness of our overlapping techniques with regards to one and two-sided communication, we also implemented the overlapped non-blocking Slabs and Pencils approaches for MPI. The MPI implementations make use of non-blocking sends and prepost receive buffers in a communication phase before non-blocking communication is initiated to maximize the potential for communication overlap and minimize the amount of unexpected MPI messages and extra memory copies. Under UPC, all communication is one-sided – the initiator provides both the source and destination addresses, and the non-blocking operations return an explicit handle which is later synchronized. Results comparing the UPC and MPI implementations of these non-blocking techniques are shown in Figure 9 in terms of total time each version of the benchmark spends in communication (all versions are always within 10% of each other for the times spent in computation). Perfect overlap would be represented as 0 seconds, and any time above 0 seconds represents the combined, non-overlapped cost of initiating and completing the non-blocking operations.

The Myrinet and Elan4 systems are the only configurations where the MPI implementations remain relatively competitive with those measured on UPC's FT implementations. The MPI configurations on other platforms either spent unacceptable time dealing with non-blocking communication messages or simply crashed. Although the MPI implementation of Overlapped Slabs and Pencils is compliant with the MPI 1.1 speci-

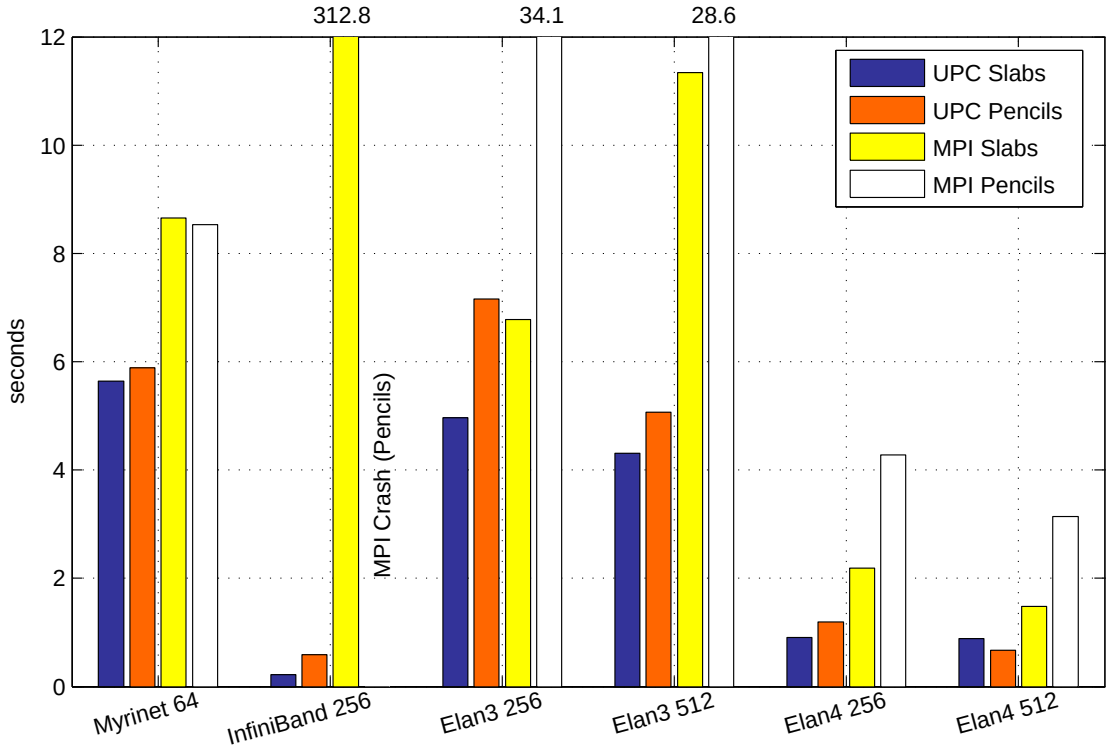

Figure 9. Time spent in Communication Initiation and Completion for Overlapped UPC and MPI

fication, the aggressive use of asynchronous point-to-point communication is not representative of the way MPI applications are typically written. While preposting buffers is always a good general strategy to prevent unexpected message costs, preposting and initiating several hundred communication operations on networks capable of highly asynchronous operation relies on correct and efficient MPI receive queue handling. The MPI results raise some obvious scalability concerns with regards to the number of point-to-point messages and the total amount of nodes exchanging messages. For example, for FT's Class D problem size at 256 processes using Pencils, each node sends and receives 2K messages for each plane it owns in the 3D cube decomposition and as such, leads to a sharp increase in the length of the MPI receive queues. Performance problems are particularly apparent in the Opteron/InfiniBand system, where the MVAPICH implementation has scalability and correctness problems: the Pencils approach causes the application to crash and the Slabs approach causes the library to spent all of its time completing asynchronous sends and receives in MPI's *Waitall* primitive. On this same InfiniBand platform, the one-sided UPC benchmark reaches its lowest communication overhead times for both Slabs and Pencils, indicating that the interconnect technology is certainly capable of producing significant speedups using either Pencils or Slabs Overlap.

UPC's overlapped approaches fare best on the more recent interconnect technologies we have benchmarked, where the commmunication times demonstrate a high messaging rate and low overhead for small-sized messages. Figure 10 presents a summary of the best result we could obtain on each platform for the default NAS Fortran implementation, the best overall MPI and best overall UPC implementations. All platforms use the largest FT problem size (Class D) with the exception of Myrinet where the problem size was too large to fit at 64 processors and on InfiniBand where the MPI implementation's unreliable handling of the largest class required the smaller Class C to be used. In all cases, the Best MPI constitutes of the better of either the NAS Fortran/MPI, Exchange-based MPI/C, Slabs MPI/C or Pencils MPI/C and happens to be Slabs MPI/C each time. This is not surprising since all the networks are capable of some form of asynchronous operation where any overlap is better than no overlap. The Best UPC happens to always be the Pencils version when compared to the Exchange-based and Slabs approaches. In many cases, the overlapped versions of the code nearly double

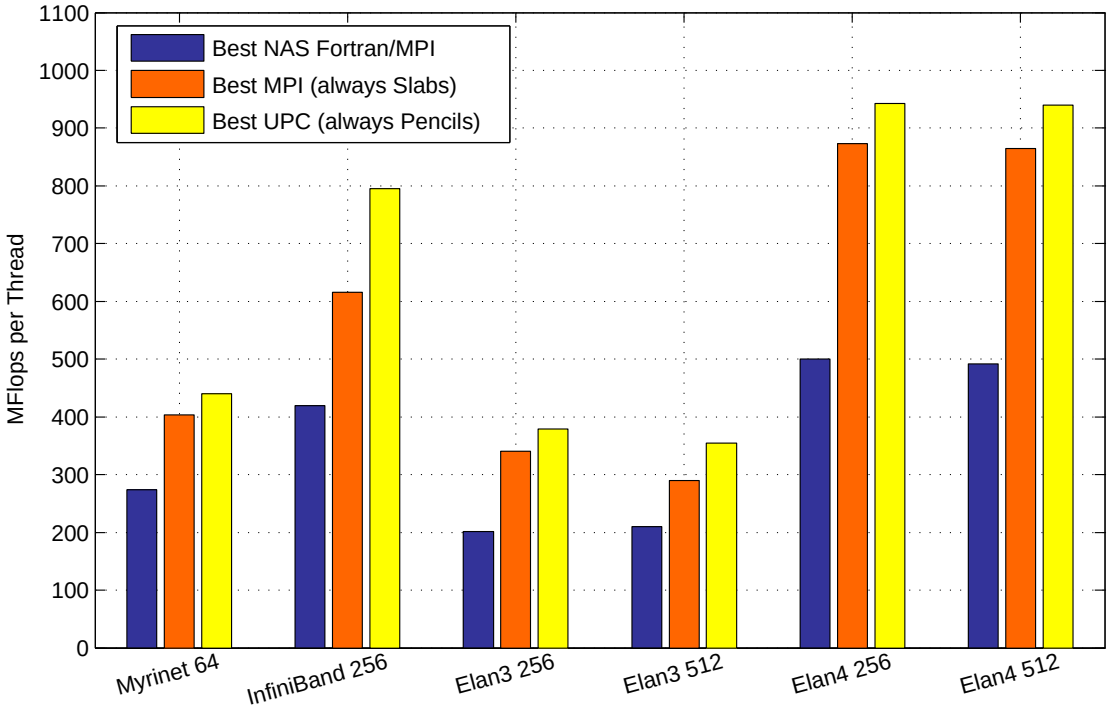

Figure 10. Best MPI and UPC Results

the performance of the original NAS FT implementation. While the original NAS FT differs from the other versions since it is based on Fortran, the other implementations share 90% of the code base with the overlap variants. In addition, the UPC Slabs and Pencils approaches differ in only about 10 lines of code, indicating the relative ease of programming for performance using finer-grained overlap in the one-sided communication model.

GASNet's low overhead for initiating many non-blocking communication operations allows a performance advantage through computation/communication overlap, trading latency associated with large bandwidth-bound operations such as the global exchange for minimal small message initiation and completion costs. The synchronization and messaging overheads imposed by the two-sided message-passing model are shown to come at a noticeable cost, and with some MPI implementations seriously limit the effectiveness of our overlap optimizations. The one-sided model successfully reduces these overheads and delivers good small message pipelining message rates which largely determine the overlap efficiency of the benchmark on RDMA-capable networks.

7 Related Work

As we have shown, large performance improvements are possible by distributing the communication operations throughout an application rather than segmenting the algorithms into a computation phase followed by a communication phase. This is evidenced by the improvements of our Pencils and Slabs implementations over the more traditional all-to-all implementations. Analyzing and optimizing the all-to-all communication pattern needed for a large parallel Fourier transform has been the subject of many papers. The analyses have ranged from modeling the performance on small commodity clusters [11, 21] through using highly specialized networks [14, 20, 48].

Many groups have also worked to tune the collectives implementations in network libraries. Automatic tuning efforts [24, 52] generate a set of good implementations of a collective operation and then choose the best one based on a search through the implementation space. Network vendors have also spent consider-

able efforts tuning their communication libraries so that they can leverage hardware-supported collectives [3]. However these methods generally assume that the algorithms are segmented into computation and communication phases. Thus, these solutions only optimize the communication phases and rely on communication-communication overlap. Part of our hypothesis is that by analyzing the communication pattern along with the computation pattern, the communication can be spread through out the computation, relieving communication bottlenecks that can occur during a communicate-only phase and deliver better overall performance through overlap.

Analyzing multiple communication calls within applications rather than just the communication itself has been the topic of many research groups. Yuan et. al [55] describe algorithms for analyzing the communication requirements of an application and then statically manage the communication metadata using knowledge of the underlying network. However, their work is mainly targeted at optical networks which have different semantics about how connections need to be setup than conventional interconnects. Iancu et. al [30] consider the benefits of breaking large messages into smaller ones to automate overlap, although their message segmentation is done implicitly by the compiler and therefore subject to the limitations of static analysis. Our implementations are based on explicit overlap where the programmer directly expresses the lack of data dependencies. Danalis et. al [18] have described techniques similar to our own to explicitly spread the communication across the computation to achieve better performance. However their main focus is on applications written for a two-sided model, which is very similar to our MPI Slabs and MPI Pencils algorithms. Danalis et. al also show the advantages of using RDMA and communication-computation overlap and show how utilizing a lower level communication library can result in better performance. The significant difference between our work and theirs is that we argue that applications written using one-sided semantics can realize even more performance gains because of the inherent advantages of the one-sided model, as evidenced by the difference between MPI Pencils and Slabs and UPC Pencils and Slabs. In addition, our work demonstrates the effectiveness of these techniques on a variety of cluster interconnects and shows that this approach scales to large processor counts and large problem sizes, further extending and validating their findings.

Finally, previous work on implementing the NAS parallel benchmarks in UPC [23] and Co-Array Fortran [15] was based on translating the MPI or OpenMP versions. In this work, the UPC implementations were written from scratch using a one-sided paradigm and thus are able to more effectively leverage the communication features of UPC/GASNet and demonstrate the capabilities of the system. We've applied some of the ideas from this work to our implementation of NAS FT in Titanium [19] (which also uses GASNet for communication) and achieved similar speedups over the MPI versions.

8 Conclusions

We have presented a detailed investigation into the relative performance of one-sided and two-sided communication, using UPC on GASNet for one-sided communication and MPI v1.1 for two-sided message-passing. Our microbenchmarks demonstrate that GASNet significantly outperforms MPI in latency performance and small to mid-size message bandwidth. As expected, both models reach the same asymptotic bandwidth at large message sizes on most platforms, but GASNet reaches the peak for smaller message sizes than MPI.

Our results suggest that one-sided communication offers an opportunity to revisit some commonly shared beliefs induced by two-sided message passing. Among these, one typical assumption is that performance is optimized by sending fewer and larger messages to asymptotically approach peak bandwidth on cluster networks. The one-sided model provides alternative mechanisms whereby small messages can provide lower startup and completion costs and that the programmer can retain explicit control over synchronization by separating it from data movement. Applying these techniques to the well-known NAS FT benchmark, we have shown improvements in two dimensions. First, the low startup and completions costs that determine the potential for efficient communication and computation overlap have produced almost 2x speedups over the existing reference NAS FT Fortran implementation. Second, by aggressively pipelining smaller-sized

messages and not imposing any particular synchronization or ordering constraints over these messages, the one-sided approach as implemented in Berkeley UPC/GASNet has produced more efficient and consistent results than the two-sided approach and various MPI implementations. These results are consistent across four different cluster networks (Myrinet, Infinband, and two generations of Quadrics) and highlight the viability of UPC as a high performance programming model for clusters.

These results provide evidence that the bulk-synchronous, message-passing style of communication popularized by MPI may not be the most effective use of cluster networking hardware. As the number of processors grows in future machines, and networks become a more significant component of system cost, optimizations such as communication and computation overlap, use of small messages to increase the depth of message pipelines, and reductions in communication overhead through one-sided communication models are likely to increase in importance.

References

[1] SGI Altix 3000 Supercomputer. http://www.sgi.com/products/servers/altix/.

[2] 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*, 5(3):63–73, Fall 1991.

[3] J. Beecroft, D. Addison, D. Hewson, M. McLaren, D. Roweth, F. Petrini, and J. Nieplocha. QSNETII: Defining high-performance network design. *IEEE Micro*, 25(4):34–47, 2005.

[4] C. Bell and D. Bonachea. A new DMA registration strategy for pinning-based high performance networks. In *Workshop Communication Architecture for Clusters (CAC03) of IPDPS'03, Nice, France*, 2002.

[5] C. Bell, W. Chen, D. Bonachea, and K. Yelick. Evaluating Support for Global Address Space Languages on the Cray X1. In *19th Annual International Conference on Supercomputing (ICS)*, June 2004.

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

[7] D. Bonachea. GASNet specification. Technical Report CSD-02-1207, University of California, Berkeley, October 2002.

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

[9] D. Bonachea and J. C. Duell. Problems with using MPI 1.1 and 2.0 as compilation targets. In *2nd Workshop on Hardware/Software Support for High Performance Scientific and Engineering Computing (SHPSEC-03)*, 2003.

[10] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi. Productivity Analysis of the UPC Language. In *IPDPS*, 2004.

[11] S. Chalasani and P. Ramanathan. Parallel FFT on ATM-based networks of workstations. In *HPDC '97: Proceedings of the 6th International Symposium on High Performance Distributed Computing (HPDC '97)*, page 2, Washington, DC, USA, 1997. IEEE Computer Society.

[12] W. Chen, D. Bonachea, J. Duell, P. Husband, C. Iancu, and K. Yelick. A Performance Analysis of the Berkeley UPC Compiler. In *Proceedings of the 17th International Conference on Supercomputing (ICS)*, June 2003.

[13] W. Chen, A. Krishnamurthy, and K. Yelick. Polynomial-time algorithms for enforcing sequential consistency in spmd programs with arrays. In *16th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2003.

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

[15] C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. Mellor-Crummey. Co-array Fortran performance and potential: An NPB experimental study. In *16th International Workshop on Languages and Compilers for Parallel Processing (LCPC)*, October 2003.

[16] Cray C/C++ reference manual. CrayDoc Manual 004-2179-003.

[17] Cray X1 system overview. CrayDoc Manual S-2346-23.

[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] K. Datta, D. Bonachea, and K. Yelick. Titanium performance and potential: an NPB experimental study. In *proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, October 2005.

[20] L. Díaz, M. Valero-García, and A. González. A method for exploiting communication/computation overlap in hypercubes. *Parallel Computing*, 24(2):221–245, 1998.

[21] P. Dmitruk, L. P. Wang, W. Matthaeus, R. Zhang, and D.Seckel. Scalable parallel FFT for spectral simulations on a beowulf cluster. *Parallel Computing*, 27(14):1921–1936, 2001.

[22] Dolphin Interconnect Solutions. *SISCI API User Guide, v1.0*, 2001. http://www.dolphinics.com.

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

[24] A. Faraj and X. Yuan. Automatic generation and tuning of

MPI collective communication routines. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 393–402, New York, NY, USA, 2005. ACM Press.

[25] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

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

[27] D. Han and T. Jones. Survey of MPI call usage. In *SciComp*, 2004.

[28] Hewlett-Packard Company. *HP UPC Version 2.0 for Tru64 UNIX*. http://h30097.www3.hp.com/upc/.

[29] P. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick. Titanium language reference manual. Tech Report UCB/CSD-01-1163, U.C. Berkeley, November 2001.

[30] C. Iancu, P. Husbands, and W. Chen. Message strip mining heuristics for high speed networks. In *Proc. 6th International Meeting on High Performance Computing for Computational Science (VECPAR)*, 2004.

[31] Infiniband trade association home page. http://www.infinibandta.org.

[32] Intrepid Technology, Inc. *GCC/UPC Compiler*. http://www.intrepid.com/upc/.

[33] LAPI programming guide. Technical Report IBM Technical report SA22-7936-00, IBM Corporation, 2003.

[34] J. Liu, A. Vishnu, and D. K. Panda. Building multirail Infiniband clusters: MPI-level design and performance evaluation. In *SuperComputing*, 2004.

[35] J. Liu, J. Wu, S. Kini, P. Wyckoff, and D. Panda. High performance RDMA-based MPI implementation over Infiniband, 2003.

[36] J. Liu, J. Wu, and D. K. Panda. High performance RDMA-based mpi implementation over Infiniband. *Int'l Journal of Parallel Programming*, 2004.

[37] Mellanox Technologies Inc. *Mellanox IB-Verbs API (VAPI)*, 2001. http://www.mellanox.com.

[38] MPI Forum. MPI–2: a message-passing interface standard. *International Journal of High Performance Computing Applications*, 12:1–299, 1998. http://www.mpi-forum.org/docs/mpi-20.ps.

[39] MPI Forum. MPI: A message-passing interface standard, v1.1. Technical report, University of Tennessee, Knoxville, June 12, 1995. http://www.mpi-forum.org/docs/mpi-11.ps.

[40] MuPC portable UPC runtime system. http://www.upc.mtu.edu/.

[41] Myricom. *The GM Message Passing System*. Myricom, Inc, GM v1.5 edition, July 2002.

[42] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Proc. RTSPP IPPS/SDP'99*, 1999.

[43] R. Numrich and J. Reid. Co-array fortran for parallel programming. In *ACM Fortran Forum 17, 2, 1-31.*, 1998.

[44] L. Oliker, A. Canning, J. Carter, J. Shalf, and S. Ethier. Scientific computations on modern parallel vector systems. In *Supercomputing '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing (CDROM)*, New York, NY, USA, 2004. ACM Press.

[45] Open64 compiler tools. http://open64.sourceforge.net.

[46] Quadrics Supercomputing. *Elan Programmer's Manual*.

[47] Man page collections: Shared memory access (SHMEM). CrayDoc Manual S-2383-22.

[48] P. N. Swartztrauber and S. W. Hammond. A comparison of optimal FFTs on torus and hypercube multicomputers. *Parallel Computing*, 27(6):847–859, 2001.

[49] Titanium home page. http://titanium.cs.berkeley.edu.

[50] UPC Community Forum. *UPC specification v1.1.1*, 2003. http://upc.gwu.edu/documentation.html.

[51] UPC Community Forum. *UPC specification v1.2*, 2005. http://upc.gwu.edu/documentation.html.

[52] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra. Automatically tuned collective communications. In *SC'00: High Performance Networking and Computing*, pages 46–46, 2000.

[53] V. Velusamy, C. Rao, S. Chakravarthi, J. Neelamegam, W. Chen, S. Verma, and A. Skjellum. Programming the Infiniband network architecture for high performance message passing systems. In *ISCA*, 2003.

[54] W.Chen, C. Iancu, and K. Yelick. Communication Optimizations for Fine-Grained UPC Applications. In *Submitted*, 2005.

[55] X. Yuan, R. Melhem, and R. Gupta. Algorithms for supporting compiled communication. *IEEE Transactions On Parallel and Distributed Systems*, 14(2), 2003.

Appendix: Platforms used in measurement

System	Processor	Network	Software	Location
Opteron/ InfiniBand	Dual 2.2 GHz Opteron (320 nodes 4GB/node)	Mellanox Cougar InfiniBand 4x HCA	Linux 2.6.5, Mellanox VAPI, MVAPICH 0.9.4, Pathscale CC/F77 2.2	NERSC Jacquard
Alpha/ Elan3	Quad 1 GHz Alpha 21264 (750 nodes 4GB/node)	Quadrics QsNet1 Elan3 w/ dual rail (one rail used)	Tru64 v5.1, Elan3 libelan 1.4.20, Compaq C V6.5-303, HP Fortran Compiler X5.5A-4085-48E1K	PSC Lemieux
Itanium2/ Elan4	Quad 1.4 Ghz Itanium2 (1024 nodes 8GB/node)	Quadrics QsNet2 Elan4	Linux 2.4.21-chaos, Elan4 libelan 1.8.15, MPI 1.24.45, Intel ifort 8.1.025, icc 8.1.029	LLNL Thunder
x86/ Myrinet	Dual 3.0 Ghz Pentium 4 Xeon (64 nodes 3GB/node)	Myricom Myrinet 2000 M3S-PCI64B	Linux 2.6.13, GM 2.0.19, Intel ifort 8.1-20050207Z, icc 8.1-20050207Z	UC Berkeley Millenium
G5/ InfiniBand	Dual 2.3 Ghz G5 (1100 nodes 4GB/node)	Mellanox Cougar InfiniBand 4x HCA	Apple Darwin 7.8.0, Mellanox InfiniBand OSX Driver v1.04, IBM XLC/XLF 6.0	Virginia Tech SystemX

All platforms used Berkeley UPC v2.1 [6], with the appropriate GASNet v1.5 [7] native conduit.