Title

Tuning collective communication for Partitioned Global Address Space programming models

Journal

Parallel Computing, 37(9)

ISSN

0167-8191

Authors

Nishtala, Rajesh
Zheng, Yili
Hargrove, Paul H
et al.

Publication Date

2011-09-01

DOI

10.1016/j.parco.2011.05.006

Peer reviewed

Tuning Collective Communication for Partitioned Global Address Space Programming Models

Rajesh Nishtala[b], Yili Zheng[a], Paul H. Hargrove[a], Katherine A. Yelick[a,b]

rajeshn@cs.berkeley.edu, {yzheng, phhargrove, kayelick}@lbl.gov

[a] *Lawrence Berkeley National Laboratory*
[b] *Department of Electrical Engineering and Computer Sciences*
University of California, Berkeley

Abstract

Partitioned Global Address Space (PGAS) languages offer programmers the convenience of a shared memory programming style combined with locality control necessary to run on large-scale distributed memory systems. Even within a PGAS language programmers often need to perform global communication operations such as broadcasts or reductions, which are best performed as collective operations in which a group of threads work together to perform the operation. In this paper we consider the problem of implementing collective communication within PGAS languages and explore some of the design trade-offs in both the interface and implementation. In particular, PGAS collectives have semantic issues that are different than in send-receive style message passing programs, and different implementation approaches that take advantage of the one-sided communication style in these languages. We present an implementation framework for PGAS collectives as part of the GASNet communication layer, which supports shared memory, distributed memory and hybrids. The framework supports a broad set of algorithms for each collective, over which the implementation may be automatically tuned. Finally, we demonstrate the benefit of optimized GASNet collectives using application benchmarks written in UPC, and demonstrate that the GASNet collectives can deliver scalable performance on a variety of state-of-the-art parallel machines including a Cray XT4, a IBM BlueGene/P, and a Sun Constellation system with InfiniBand interconnect.

Keywords: Partitioned Global Address Space Languages, Collective Communication, One-Sided Communication

1. Introduction

Partitioned Global Address Space (PGAS) languages combine the convenience of shared memory programming with the locality and data layout control of message passing. These languages run well on both shared memory and distributed memory hardware, taking advantage of shared memory support when it exists and leveraging the partitioned nature of the address space for performance on distributed memory supercomputers and clusters. The main difference between PGAS programming and the conventional message passing programming model, such as MPI, is that PGAS languages provide a global view of the memory across nodes and support one-sided communication, *read* and *write* (or *get* and *put*), for shared data. In contrast, in the two-sided communication model in MPI a process can only see its own local memory space and needs matching *send* and *receive* operations between itself and a remote process to transfer data. MPI is widely used due to its high quality implementations, portability, and good performance scalability, but the two-sided communication model is inconvenient for expressing asynchronous communication patterns in applications like histogram construction, mesh generation, and certain graph algorithms, in which only one side knows about a communication event. A global address space and one-sided communication primitives make developing such applications easier. In addition, by decoupling the transfer of data from the notification to a remote process, the one-sided communication can take advantage of Remote Direct Memory Access (RDMA) hardware and avoids some of the overheads of message and tag matching that are inherent to the two-sided model.

There are many different variations of PGAS languages and each has different design goals and programming styles but all of them fundamentally share the idea of a global address space in which a thread has affinity to part of the global address space. PGAS programming language implementations share a common set of runtime needs including: one-sided communication, collective communication, active messages, memory allocation, and the management of processes and threads. The GASNet communication library [1] is a portable runtime system for enabling PGAS implementations. GASNet is used to implement many PGAS programming languages including: Co-Array Fortran[2]), UPC[3], Titanium[4], Cray Chapel[5], in addition to some research prototypes.

One-sided communication involves only a single application level process and two-sided communication involves two, but it is often useful in both mod-

els to have operations involving a large set of processes working collectively to perform a global communication operation. Common *collective* operations such as broadcast and reduction could be performed by accessing shared variables in a global address space, but this approach is neither convenient nor scalable. To aid in productivity and performance, many languages provide a standard set of such collective operations. These operations encapsulate common data movement and inter-processor communication patterns, such as broadcasting an array to all the other processors or having all processors exchange data with every other processor. The abstraction is intended to shift the responsibility of optimizing these common operations away from the application writer, who is probably an application domain expert, into the hands of the implementers of the runtime systems. While this problem has been well studied in the two-sided communication model community, the primary focus of this paper will be to understand and improve the performance and productivity benefits of collective operations in PGAS languages such as UPC[3], Titanium[4], and Co-Array Fortran[2], with the emphasis on UPC. The semantic differences between one- and two-sided communication pose interesting and novel opportunities for defining and tuning collectives.

PGAS collectives are different than MPI collectives in several aspects:

- MPI collectives are node-centric, while PGAS collectives are data-centric. For example, in MPI broadcast, the user specifies the source by providing the rank of of root. In contrast, in a UPC broadcast, the user specifies the source by providing the pointer to the shared data buffer and the runtime determines the location of the data automatically.

- MPI collectives implementations are often built on top of two-sided communication primitives (send and receive) whereas PGAS collectives are often built on top of one-sided communication primitives (put and get).

- PGAS collectives have an added semantic complexity in that one process may be writing to remote data that is involved in a concurrent collective operation. In UPC, for example, a set of synchronization modes are used to limit such behavior.

To benefit as many PGAS implementations as possible, we have designed, implemented, and tuned collective functions for PGAS languages in the GASNet system. Because GASNet is designed as a low-level system software library, programmers usually use GASNet indirectly by programming in one

of the PGAS languages that implemented on top of GASNet. In this work, we use Berkeley UPC to implement application level benchmarks to evaluate the performance of the optimized GASNet collectives implementation.

In this paper we make the following contributions: i) explore the interface issues with collective communication in PGAS language, which includes trade-offs in performance and simplicity; ii) describe the implementation of a collective communication library for PGAS languages as part of the GASNet communication library; iii) enumerate some of the implementation approaches for various collectives and the need to automatically tune over these implementation to select a reasonable one for a given machine and usage scenario; iv) evaluate the collectives on some of the largest computer systems available today, including a Cray XT4, an IBM BlueGene/P, as well as a large Sun cluster with an InfiniBand interconnect.

2. Designing a PGAS Collectives Interface

Many parallel algorithms perform global communication operations, such as broadcasts and reductions, that involve data movement across all threads. In PGAS languages, because all threads have access to shared data, the first design question is whether these global operations should be performed as a *collective* operation in which all threads participate together, can a single thread invoke a global operation. In PGAS languages, a single thread has access to all data in the shared space, operations on data spread across the machine can be performed by a single thread. Nevertheless, UPC, like MPI, provides these global operations as collectives. This simple model provides a good abstraction of current machines and requires minimal runtime support, albeit at some loss of convenience. Several questions remain about the collectives interface and implementation strategy.

One key part of the collectives interface that we do not discuss here in detail is the exact set of collective operations. GASNet and UPC provide both rooted collectives, such as broadcasts and reductions with a single source or destination thread that acts as the root, as well as non-rooted collectives that perform operations such as all-to-all communication patterns. Some of the collectives perform computation, e.g., scans and reductions, whereas others simply move data. These are described for UPC and GASNet in their respective specifications [3, 1].

2.1. Global Address Space and Synchronization

In PGAS languages, all the threads may have global knowledge of the destinations of all the data. This knowledge can be used to avoid extra copies of data or synchronization within collective operations, since one thread can write directly into the final destination of another. However, this lack of copying raises semantic issues, since the data involved in a collective can be read or written by the destination thread or any other thread while a collective is in progress. Analogous problems arising with writing to input data in a collective, since the data could be modified by another thread during the collective operation. One can simplify the semantics by inserting a barrier before and after every collective but this over-synchronizes and will not fully expose the performance advantages of collectives in one-sided programming models. Currently the specification of Unified Parallel C (UPC) language is exposing this problem to the end user and having the user decide what the collective synchronization semantics need to be. At the strictest end of the spectrum, data movement can only occur after the *last* thread has entered the collective and before the *first* thread leaves the collective. At the loosest end of the spectrum, data movement can start as soon as the *first* thread enters the collective and can continue until the *last* thread leaves the collective. These options are controlled by synchronization flags on each collective operation, which significantly complicate the interface but also gain performance.

2.2. Nonblocking Collectives

The UPC synchronization flags in collectives control when the data can be safely accessed by other parts of the program, and they admit collectives to overlap with another, which can be useful in a collective-heavy computation. However, they do not permit a thread to begin a collective, perform other work, and then return to see that the collective has completed. UPC has split-phase barriers to allow for global synchronization of this kind, but not split-phase (also called *nonblocking*) collectives. The performance benefits of overlapping communication with computation have been well studied[6]. Some have further explored how these techniques can be applied to collectives and their benefits in applications that are written in two-sided communication models[7, 8].

To enable overlapping collective communication with computation or other types of communication, we have designed our collectives to be nonblocking. Like a nonblocking memory transfer, the operation returns a han-

dle that needs to be synchronized for completion. When these nonblocking collectives are used, we do not require any processor to be inside the collective routines while they are in progress. Special mechanisms (either based on interrupts or polling) will be needed to ensure that the collectives can run asynchronously with the rest of the computation. Full details of the implementation of the nonblocking collectives are available in [9].

2.3. Teams

The control model within the UPC language is a flat Single Program Multiple Data (SPMD) model, so all threads start a executing the main function at program startup and run to completion throughout the program. Threads may take different branches within the code, and do not run in lock step, but there is no language level mechanism for specifying that a subset of threads are working together on a computation and therefore perform collective operations as a team. The question of how to support teams has been an ongoing topic of discussion in UPC, and they are supported in other PGAS languages as well as MPI. Therefore, the GASNet design supports teams and they will be used in some of the UPC applications through extensions in the Berkeley UPC compiler.

2.4. Address Modes

One of the advantages of a one-sided communication model is that the initiator of communication has information about where remote data lives. In a collective operation, this potentially requires one address per thread. At the UPC level, collective operations are performed on distributed arrays, and the runtime system keeps track of where all of the blocks of the array live in memory. But a more general interface allows each thread to point to an independent data location, not necessarily part of a distributed array. GASNet therefore supports two modes of use, the first being faster, and the second more flexible:

1. **Single:** All the threads pass the addresses for all the other threads so that no address discovery is necessary. The base addresses for the remote access regions are usually exchanged when the job starts up so computing all the remote addresses can be done with local computation and no communication. This is the address mode that is used by the Berkeley UPC compiler.

2. **Local:** If providing all the addresses is not feasible a thread can only pass the address of the data on the local node. This is the relevant mode for clients that do not keep track of remote addresses for shared data structures and is the address mode employed by the current MPI collective interface.

Allowing the collective to only specify the local address, as is done with MPI, allows more flexibility to the end user. If the language features or application do not easily lend themselves to knowing all the different addresses then this mode is preferable. The collective has to either exchange the address information or use internal buffers that are located in a known place in the remote segments. Thus at scale the latency costs associated with address discovery might be much larger than the overhead induced by $O(N)$ data structures to specify the addresses.

3. Optimizing PGAS Collectives Implementation

To achieve scalable high performance, we have implemented several optimizations unique to PGAS in addition to common optimizations used in MPI collectives.

3.1. Leveraging Shared Memory

Most modern systems, including the ones presented in this study, contain multicore processors that have shared physical memory for all processor cores within a node. We use a "virtual node" to represent a collection of threads that share a common address space, system resources, and network endpoint resources. Each virtual node is implemented as an underlying OS process. The Berkeley UPC runtime supports two modes of operation: (1) a one-to-one mapping between UPC threads and GASNet processes or (2) each UPC thread in the shared memory domain is mapped to an operating system level thread within a single GASNet process (i.e. a many-to-one mapping). In the latter case, a data movement operation between the threads in a shared memory domain is a simple `memcpy()`.

For the experiments in this paper we use a one-to-one mapping between UPC level threads and the underlying operating system threads. Thus all UPC threads that are sharing the same physical address space can use shared memory to transfer the data amongst themselves. Our collectives implementation is aware of the hierarchical execution model of threads and processes.

Thus, to minimize the network traffic, a representative thread from each node manages the internode network communication with other nodes and uses `memcpy` to pack and unpack the data for communication within the node. Our study is conducted with POSIX threads [10], however the techniques discussed are applicable to other threading models as well.

Since communication amongst processes is handled through the network interface, the current collective library makes no distinction between virtual nodes that are co-located on the same physical node and virtual nodes on distinct physical nodes. Future work will address this issue and optimize the collectives to further be aware of the communication amongst virtual nodes that are co-located on a physical node.

3.2. Communication Topology Optimizations with Trees

Rooted-collectives are typically implemented by a logarithmic *Tree* algorithm. We define a tree as a directed acyclic graph constructed with the nodes such that all the nodes except the root has an in-degree of 1. An edge in the graph represents a virtual network link between the source and the destination. In many cases, it is useful for these virtual links to match what the underlying physical network provides, however this is not necessary for correctness. On all the platforms used in this study, the underlying networks can appropriately route the messages from the source to the destination without intervention from the collectives library. Tree algorithms include many parameters and variations. A full analysis of the tree geometries is not possible in the space provided here and is slightly tangent to the focus of this paper. Hence we refer the reader to [9] for a full treatment on the topic including a set of performance models that can be used to guide the choice of the optimal tree. The experiments presented in the rest of this paper show the results from the best tree picked by a search over the different tree geometries.

3.3. Data Transfer Optimizations

It is an important design issue about how the data is communicated between nodes. Since the intermediary nodes of a communication pattern will need to receive and forward data to their peers, they need to know when the sender has finished the data transfer. Thus along with the one-sided data transfer, we have developed mechanisms to signal when the data has arrived. We explore two different data transfer mechanisms and their tradeoffs.

3.3.1. Signaling Put

For *Single* address collectives, since the sender knows the destination address of the data, we use a GASNet *Long* active message [11] to transfer the data. The function invoked on the target node sets a state flag indicating that the data has arrived. For collective operations not requiring synchronizations, the data can be transferred to the destination nodes without the point-to-point synchronization.

3.3.2. Rendezvous

For *Local* address collectives that would like to avoid the latency costs of address exchange and *Single* address collectives that require the user buffers not be touched until the destination thread has entered the collective, the data needs to be transferred into an auxiliary buffer on the destination node. However we do not want to pay the cost to manage the internal memory to stage the data before copying it into the user's buffers. For machines with large processor counts and small per-node memory (e.g., the IBM BlueGene/P), it is impractical to use a significant percentage of the memory for bounce buffers. If one were to limit the amount of space available for these bounce buffers, then complicated flow control mechanisms are needed to ensure that the memory space is not exhausted. To avoid these issues, we employ a Rendezvous approach for larger messages. The sender first notifies the receivers about the address of the source data buffer. Then the receivers initiate *get* operations to fetch the data from the sender's source buffer once they know the address. After the *get* operations complete, the receivers increment an atomic counter on the sender to acknowledge the completion. The sender checks the acknowledgement counter and waits until all data transfers are done at which time it can exit the collective. The Rendezvous data transfer protocol naturally enforces that a node's data can be modified by other nodes only when the node is in the collective function.

3.4. Non-Rooted Collectives Optimizations

There are a large set of applications that rely on non-rooted collectives, in which every thread receive a contribution from every other thread. Naïve (Flat) implementations of these operations can lead to $O(T^2)$ messages while well tuned ones can perform the same task in $O(T\ log\ T)$ messages.

Non-rooted collectives commonly use some variant of Bruck's algorithm[12], which we refer to as dissemination algorithms. The dissemination algorithms send data to its final destination node through intermediaries in $O(N\ log\ N)$

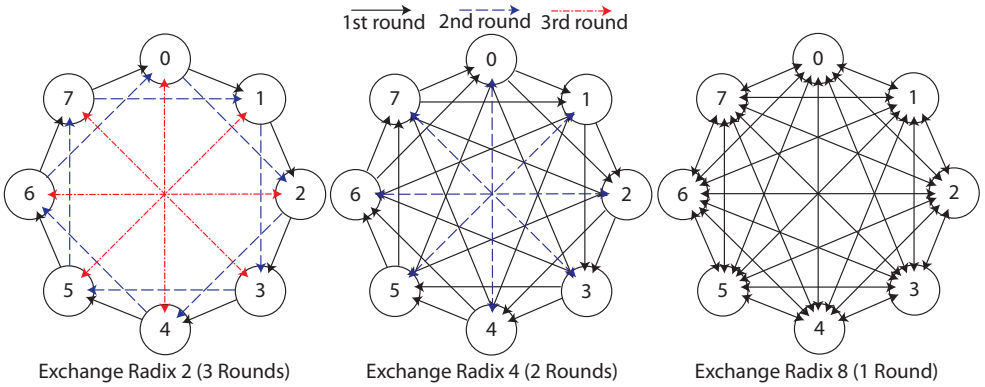

Figure 1: Example Dissemination Communication Patterns

steps. Since the same data is sent more than once, this approach requires more bandwidth than having each thread send direct messages to all other threads. Thus we can trade message counts for additional bandwidth. The most common dissemination radix is two; however higher radix algorithms are often useful in practice. As the radix increases, the replication of data is reduced at the expense of more messages per round. Figure 1 shows the communication patterns with 8 nodes for three different radices. The total number of messages can be approximated by $O(N(k-1)(log_k N))$ where k is the radix, and N is the total number of nodes. Notice that at the extreme when $k = N$, the dissemination algorithm uses the same number of messages as the Flat Algorithm. At each round, the message size is $O(\frac{BNt^2}{k})$ where B is the size of the personalized messages between the threads.

3.5. Hardware Collectives and Algorithm Compositions

More and more interconnects support collective communication in hardware, which usually is faster than software implementations. For example, IBM BlueGene supercomputers [13] have a special network for collective communication. The hardware collectives are orthogonal to the software implementations and are included in our library.

Complex collective operations can be implemented by composing basic collective algorithms. For example, **All_Reduce** can be implemented by **Reduce** followed by **Broadcast** or by **Reduce_scatter** followed by **All_Gather**. Similarly, **All_Gather** can be implemented by everyone **Broadcast** or by **Gather** followed by **Broadcast** from the root. Our collectives

framework facilitates advanced users to implement new algorithms via algorithm composition.

3.6. Automatic Tuning

The collectives implementation space is very large because GASNet includes many possible algorithms and parameters for each collective. We have thus implemented an automatic collective tuning system in GASNet that stores all algorithms for the various collectives in a large index implemented as a multi-dimensional table. For each collective, a collection of possible algorithms is available. Along with each algorithm, the list of applicable parameters and their ranges are stored (e.g. tree shapes and radices). As part of the index, each collective algorithm advertises its capabilities and requirements. By allowing different algorithms to have different capabilities, we can write more specialized collectives, which work well for certain input parameters but may not be applicable to others. For each of the combination of input parameters, at least one algorithm must exist to ensure a complete collectives library. In addition, the system has been designed to incorporate customized hardware collectives into the tuning framework.

It is an expensive process to find the best collective algorithm among all choices. There are many factors that affect the overall performance of the library. Some performance-influencing factors can be inferred at the library installation time while others are only known during the application execution.

- **Offline Tuning:** Offline tuning is the process of tuning the collective library outside the users application. This can be done either at the library install time or periodically by the system administrator. Because the tuning step is outside the critical path of the library, it can spend more time finding the best variant of the code. However, since the exact collective input parameters are not known from the application, a set of input tuples need to be searched. The set of sample input tuples can be adjusted as needed to tailor for specific applications. This is the tuning approach that is used by popular automatically tuned software packages such as ATLAS.

- **Online Tuning:** Online tuning is the process of tuning the collective library either implicitly or explicitly during the run of applications. The advantage of this approach is that the complete information about

input data, processor layouts and network loads are available to make tuning decisions. However, because the collective tuning can be a potentially expensive process (up to a few minutes per input tuple on a large node count), any gains from the collective tuning might be overwhelmed by the cost of the tuning.

To get the positive aspects of each method, we employ a combination of online and offline tuning in GASNet. The main aim of the offline tuning is to refine the search space and throw away obviously bad candidate algorithms. The online search space is responsible for fine tuning the algorithmic selection given runtime factors. The full design and implementation of the automatic tuner can be found in [9].

4. Experimental Platforms

We use three large scale supercomputers for our experiemetns: the IBM BlueGene/P at Argonne National Labs (Intrepid), the Sun Constellation System at the Texas Advanced Computing Center (Ranger), and the Cray XT4 at the National Energy Research Scientific Computing Center (Franklin). As of November 2010, these machines are respectively ranked 13^{th}, 15^{th}, and 26^{th} on the Top500 list[14], the list of the 500 most powerful computers in the world. Table 1 lists the architectural summary of these systems.

One of the features common to all these systems is a Direct Memory Access (DMA) device that is attached directly to the network card. This is hardware that allows the network interfaces to directly read and write data to the main memory (or the L3 cache in the case of the BlueGene/P) without having the processors actively manage communication in progress. In addition these devices can directly read or write data to the memory on remote nodes, a feature known as Remote Direct Memory access (RDMA) designed for one-sided communication. The hardware support for one-sided communication is leveraged heavily in our work. Our related work [6, 15] has also shown how these features can be utilized to realize significant performance advantages.

5. Performance Study with Micro-Benchmarks

We study the performance characteristics of our implementation of PGAS collectives by a set of micro-benchmarks.

	Cray XT4	IBM BlueGene/P	Sun Constellation
Machine Name	Franklin	Intrepid	Ranger
Machine Location	NERSC	ALCF	TACC
Top500 Rank (November 2010)	26	13	15
Processor Type	AMD Opteron (Budapest)	IBM PowerPC 450	AMD Opteron (Barcelona)
Clock Rate (GHz)	2.3	0.85	2.3
# Cores/Processor	4	4	4
# Processors/Node	1	1	4
Peak Perf. /Node (GFlop/sec) †	36.8	13.6	147.2
Memory BW (GB/s)	10.6	13.6	10.6
Network BW (GB/s)	7.6 (2-way)	0.85 (2-way)	1 (1-way)
One-way Network Latency (μs)	6.2	1.5	2.3

Table 1: Experimental Platforms (†All platforms support a peak of 4 double-precision floating point operations per cycle.)

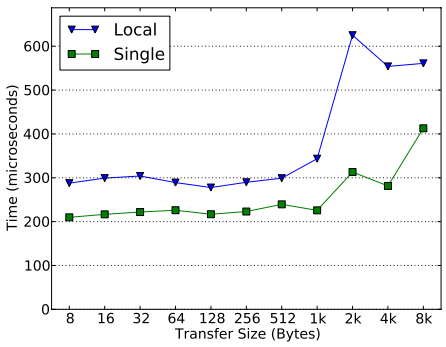

Figure 2: Comparison of Data Transfer Mechanisms (Broadcast performance on 2048 cores of the Cray XT4)

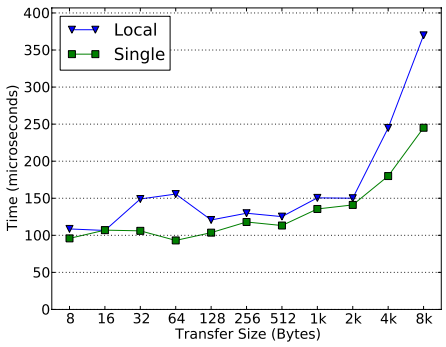

Figure 3: Comparison of Data Transfer Mechanisms (Broadcast performance on 1024 cores of the Sun Constellation)

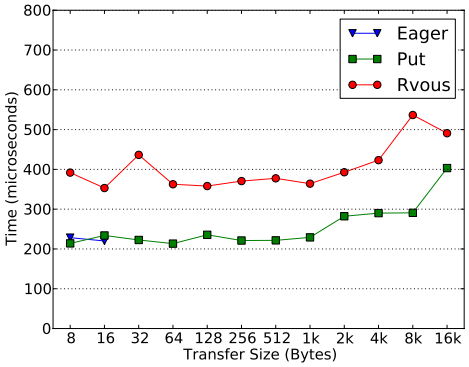

Figure 4: Comparison of Data Transfer Mechanisms (Broadcast performance on 2048 cores of the Cray XT4)

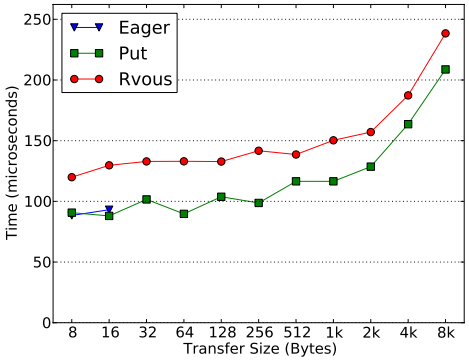

Figure 5: Comparison of Data Transfer Mechanisms (Broadcast performance on 1024 cores of the Sun Constellation)

5.1. Address Modes

Figures 2 and 3 show the performance advantages of knowing all the addresses versus having to discover the addresses. As the data show on both platforms, the *Single* address mode can consistently realize better performance than *Local*. One would expect the performance difference to be only significant at smaller message sizes. However, the cost of the added synchronization also has a significant impact on larger message sizes. The cost of the added synchronization needed to discover the addresses limits the amount of time a node spends transferring data and thus reduced the overall bandwidth. Comparing across the two platforms, the overhead of *Local* collectives is a lot higher on the CrayXT4 than the Sun Constellation. From the platform summary data in Table 1, the Sun Constellation has a lower network latency than the CrayXT4. Thus the cost of the communicating the addresses, which involves latency sensitive small message transfers, is higher and hence the performance penalty.

5.2. Data Transfer

Figures 4 and 5 compare the performance of the various transfer mechanisms for a Broadcast of various sizes. The times shown are the average for multiple back-to-back collectives. Each of the collectives is separated by a barrier to approximate the time taken for the data to reach the bottom of the tree. As the data show, the cost of the extra synchronization required for the Rendez-Vous approaches on both platforms increase the time for the total operation. The 3D Torus on the Cray XT implies that the signaling mechanisms

necessitated by the Rendez-Vous need to go through multiple hops further aggravating the latency of the operations. The Put mechanism consistently outperforms the Rendez-Vous mechanism. For comparison purposes we have also measured the performance of the same operations in MPI. In the case of the Cray XT4, the MPI performance is between that of the Put and Rendez-Vous. As the transfer size increases MPI and the Rendez-Vous are similar in performance. The data of the Sun Constellation show that the MPI performance is similar to the performance of Put. From the figures we can draw the conclusion that when applicable, taking advantage of the Global Address knowledge can yield good performance advantages. MPI may also utilize these operations but it needs extra support in the runtime to pre-exchange the addresses of the internal buffers used by the collective implementation. However, one-sided programming models eliminate the associated overheads of copying data to and from the collectives library's internal buffers.

5.3. Non-Rooted Collectives

Figures 6, 7, and 8 show the performance of different Exchange algorithms on 512 cores of the Cray XT4, 256 cores of the Sun Constellation, and 2048 cores of the IBM BlueGene/P. As the data show at small message sizes, the Dissemination algorithms yield the best performance by reducing the overall message count. For each platform we compare three different benchmarks. The *Flat* measures the performance of GASNet and having all the threads transferring data to each other directly without using intermediary nodes. This method incurs a total of $O(n^2)$ messages without any extra bandwidth consumption. The *Dissem* shows the performance of the best dissemination algorithm with a search across radices 2, 4, and 8 for Bruck's algorithm implemented in GASNet. The *MPI* line shows the corresponding performance for MPI. For all the GASNet experiments we have used the *Single* address mode. On both the Sun Constellation and the Cray XT4, the Dissemination algorithms yield the best GASNet performance at the lower scale, indicating that the advantages of sending $O(n\ log\ n)$ messages yield good improvements when the bandwidth costs are not very high. However, as the bandwidth costs become expensive, the Flat algorithms yield better performance. On the IBM BlueGene/P, however, the Flat algorithm yields consistently the best performance. This indicates that the cost of the extra hops in the network is so expensive that sending the data directly to the destination nodes is the best option. The cross over point is platform specific and thus a combination of performance modeling and search can yield the

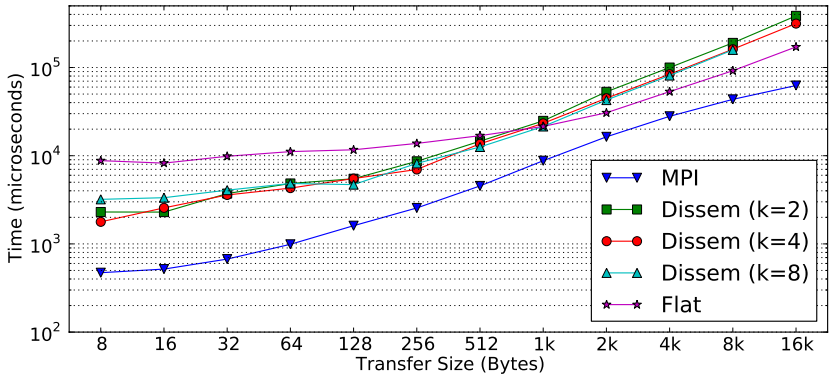

Figure 6: Comparison of Exchange Algorithms (512 cores of Cray XT4)

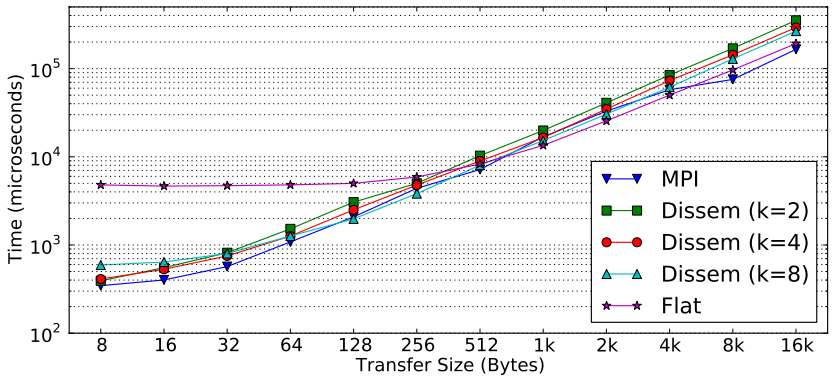

Figure 7: Comparison of Exchange Algorithms (256 cores of Sun Constellation)

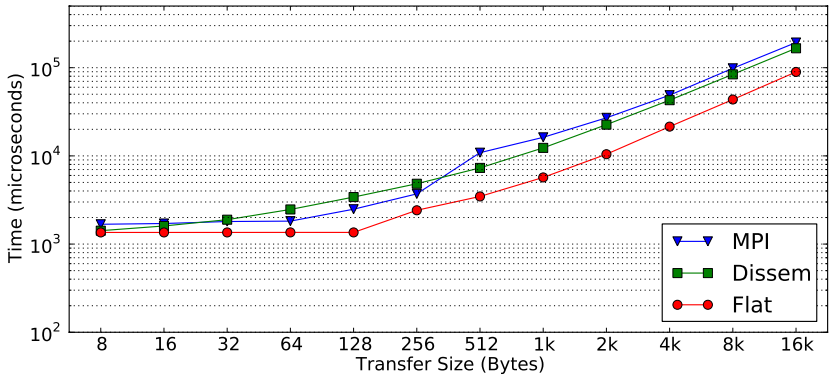

Figure 8: Comparison of Exchange Algorithms (2k cores of IBM BlueGeneP)

Data size (bytes)	Sun Constellation	Cray XT 4
4	0.012165	0.036255
128	0.013965	0.037965
2048	0.017625	0.047925
32769	0.087345	0.10884

Table 2: Tuning time (seconds) for Broadcast with different data sizes on three platforms

best performance. On the Cray XT4 the best GASNet performance tracks the MPI performance but further enhancement needs to be done to better optimize the performance. The performance of GASNet is on par with MPI on both the Sun Constellation and the IBM BlueGene/P.

5.4. Automatic Tuning

Table 5.4 lists the tuning time for a **Broadcast** operation. If the collective has been tuned before, the lookup operation is a simple table lookup, whose cost is negligible compared to the collective operation time. The memory requirement for the autotuning look up table is proportional to the number of entries in it. For each data point, it takes about 48 bytes to store the tuning results.

6. Application Level Benchmarks

To evaluate the overall application performance delivered by our collectives library, we have developed three popular and important application level benchmarks using the optimized collectives: dense matrix multiplication, Cholesky factorization and 3-D FFT. Both dense matrix multiplication and Cholesky factorization uses **upc_team_broadcast** (the UPC equivalent of **MPI_Bcast**). 3-D FFT uses **upc_team_exchange** (the UPC equivalent of **MPI_Alltoall**) for transposing the 3-D array during different phases of the row-column multi-dimensional FFT algorithm. The collectives autotuning is done *offline* for these applications.

6.1. Dense Matrix Multiplication

Dense matrix multiplication (also known as GEMM) is one of the most commonly used computational kernels in large scale parallel applications. This kernel computes $C = A \times B$ where A, B, and C are dense matrices of sizes $M \times P$, $P \times N$, and $M \times N$, respectively.

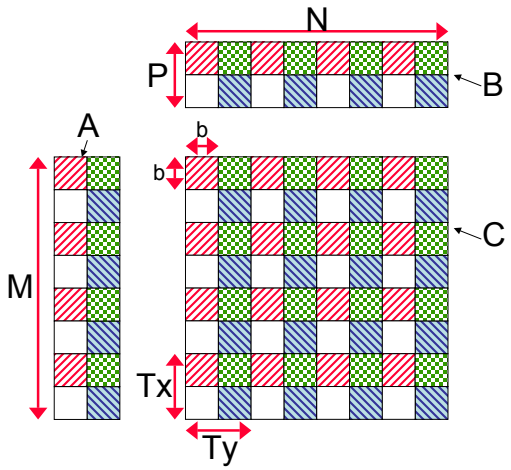

Figure 9: 2-D block-cyclic distribution for parallel matrix multiplication diagram. TX and TY are the dimensions of the 2-D thread grid.

In order to effectively parallelize the problem in this benchmark, each of these matrices are partitioned and distributed across the threads with a 2-D block-cyclic format (Figure 9). The data structures are implemented as arrays of sub-matrices with UPC *shared* pointers. The pieces of the matrix are color coded by the thread that owns the piece of the matrix. We can compute a particular block, $C[i][j]$ by performing the operation:

$$C[i][j] += \sum_{k=0}^{P-1} A[i][k] \times B[k][j]$$

For all blocks in a given row i, only the elements of $A[i][k]$ need to be broadcast-ed and stored into a temporary array. The subset of the threads that owns row i has size $O(\sqrt{T})$, where T is the total number of UPC threads. In the next step, $B[k][j]$ is broadcast-ed to a separate scratch array within every thread that shares a column of the matrix (the column teams). Column broadcasts occur over a set of $O(\sqrt{T})$ threads in the column dimension. This is a blocked implementation of the SUMMA algorithm [16]. The algorithm has further been modified to perform a "prefetch" of the rows and columns of matrices in order to overlap the communication needed for future panels of the matrix multiplication with the computation of the current one.

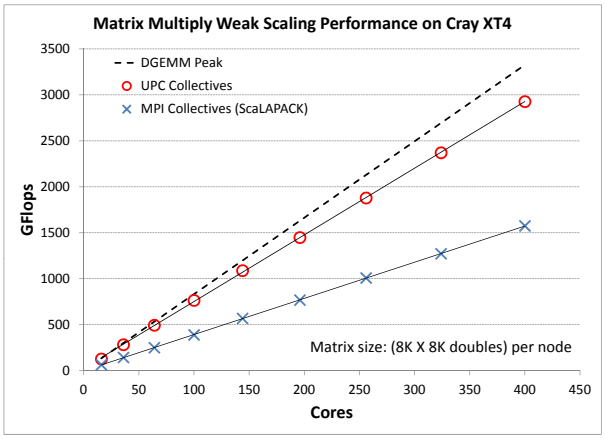

Figure 10: Matrix Multiply Performance on 400 cores of Cray XT4

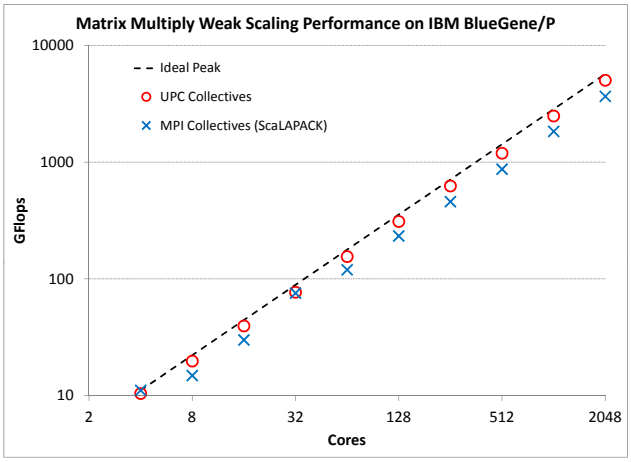

Figure 11: Matrix Multiply Performance on 2048 cores of IBM BlueGene/P

Thus we are able to leverage the non-blocking collectives in GASNet to yield communication/computation overlap.

Figures 10 and 11 show the performance of weakly scaled matrix multiplication (i.e. fixed number of points per node) on Cray XT4 and IBM BlueGene/P. Our UPC implementation is written from scratch by using collectives. Our experiments were conducted with one UPC thread per compute node with the local `DGEMM` operations performed through calls to the multi-threaded vendor-optimized BLAS library. UPC is used to manage the communication among the nodes and the math library uses all four cores to perform the local matrix multiplication. The number of nodes in our experiment is always a perfect square so that a square thread grid is assumed. Each node is allocated a square matrix with 8192×8192 double precision elements and thus the problem size is weakly scaled as the number of nodes grows. As the data show, the best performance is obtained when the code leverages the UPC non-blocking team collectives. The UPC code significantly outperforms the comparable MPI (ScaLAPACK) version. Effectively using non-blocking collective communication can lead to good performance improvements even for a computation-intensive program such as matrix multiplication.

6.2. Dense Cholesky Factorization

The second benchmark is dense Cholesky Factorization, which computes ($A = U^T U$) for a square $M \times M$ matrix A. As shown in Figure 12, the method performs serial computation on the upper right corner and updates the lower left and upper right quadrants of the matrix. Then a large parallel outer product of A_{LL} and A_{UR} is performed to update the lower right corner. After the update, the lower right quadrant is recursively factored. Because the matrix elements in the lower right corner tend to be more heavily used and updated compared to the other parts, a purely blocked layout would induce a poor load balance since the most heavily used elements will be concentrated among a few threads.

In order to alleviate this problem, a checkerboard layout[17] of threads is used to more evenly distribute the load across the threads. The operation requires three rounds of **broadcast** operations at each of the $\frac{M}{b}$ steps. The first round broadcasts the data from the panel that has just been factored to all the other panels in the same row to perform a triangular solve. An outer product with the result of the triangular solve updates the rest of the matrix. Because the input matrix is symmetric, the outer product is computed by performing ${A_{UR}}^T \times A_{UR}$ in Figure 12. However this reduces the opportunity

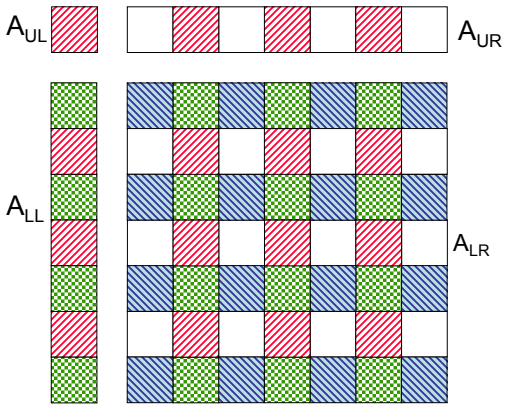

Figure 12: Cholesky Factorization Diagram with 2D block-cyclic distribution. The matrix is partitioned into 4 quadrants: a small upper left corner (A_{UL}), a tall skinny lower left corner (A_{LL}), a short and wide upper right corner (A_{UR}), and a large lower right corner (A_{LR}).

for overlapping communication and computation because the column and row teams are not as strictly adhered to. For example a thread that owns column 1 in the diagram (the first column of A_{UR}) is not part of the row team that needs the data. Hence the thread first has to directly communicate the data to a member of the row team which can then initiate the broadcast. This extra step and communication overhead drastically reduces the opportunity for nonblocking collectives. Hence our implementation of the benchmark only measures the performance of blocking algorithms. Once the data is passed on to a member of the appropriate team we used the same team broadcasts to run the outer-product as described in the Matrix Multiply benchmark.

Figures 13 and 14 show the performance of our Cholesky implementation, compared to the widely-used ScaLAPACK implementation. Our UPC implementation is written from scratch using the collectives. We use one UPC thread per node and use a multi-threaded optimized BLAS library to perform the local BLAS operations. All the different benchmarks use the same backing BLAS implementation so we expect the only differences to arise from communication between the different nodes. We use on all cores on a node and only use UPC to manage the communication among the nodes. As the data show, our implementation of the Cholesky factorization performs better than or equally well as ScaLAPACK. From the data we can conclude

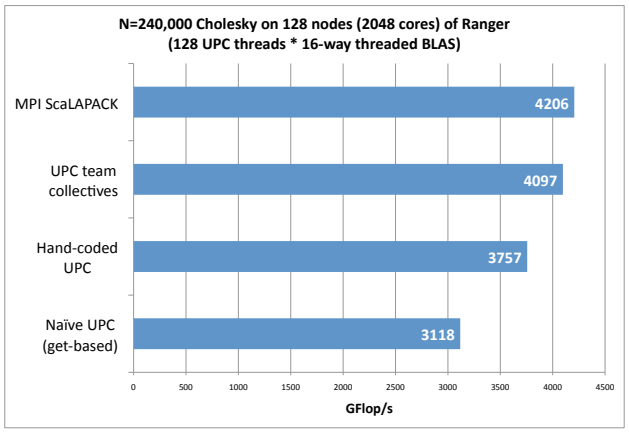

Figure 13: Cholesky Factorization Performance on 2048 cores of Sun Constellation

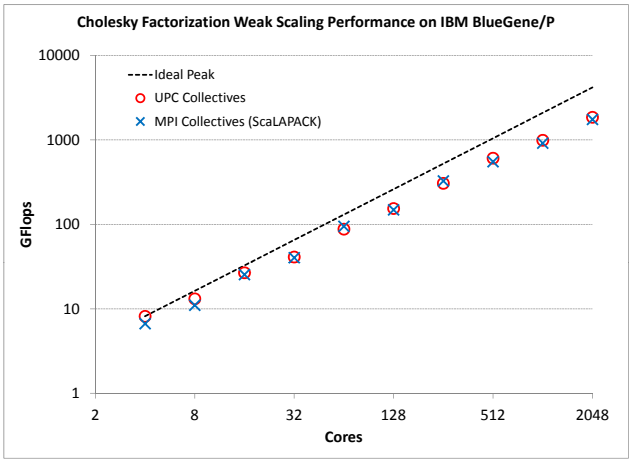

Figure 14: Cholesky Factorization Performance on 2048 cores of IBM BlueGene/P

that applying the collective communication is essential for delivering good performance and algorithms that are comparable with the current state of the art. For both ScaLAPACK and our implementation, there is still a gap between the machine peak performance and the achieved performance. Further improvements in the collective algorithms, load balancing, and serial performance can help parallel efficiency.

6.3. 3-D FFT

We use the NAS Parallel Benchmark [18] FT to study the performance of the non-rooted collective, **exchange**. The kernel of the NAS FT benchmark is a three-dimensional FFT. In a three-dimensional FFT, the rectangular prism (of NX $\times$ NY $\times$ NZ points) is evenly distributed among all the threads and FFTs must be done in each of the dimensions. Depending on the data layout, the prism might need to be transposed in order to re-localize the data to perform the FFTs. This transpose step is often the performance-limiting step at large scale since it stresses the bisection bandwidth of the network.

Figure 15 shows the data partitioning of a two-dimensional thread layout for 3-D FFT. 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 re-localizes the data to make the Y-dimension contiguous and thus the communication is performed among teams of threads in the TY dimension (*i.e.* all the threads within the same thread plane in Figure 15). The second one re-localizes the data to make the Z-dimension contiguous among teams of threads in the TZ dimension (*i.e.* all the threads within the same thread row).

Figure 16 shows the performance of the NAS FT benchmark on 1024 cores of Cray XT4. Figure 17 shows the performance of the NAS FT benchmark on power of two core counts up to 32,768 cores of IBM BlueGene/P. The problem size was also scaled as the number of cores grew so that the memory per thread remains constant throughout all the thread configurations. Since the benchmark is communication bound, using the maximum flop rate of the machine provides a meaningless upper bound on the application performance. Therefore we have created an upper bound analytic model that assumes the communication is the limiting factor [19]. Thus it assumes the total time taken by the benchmark is the time needed to perform the **exchange** operations. In order to get an approximation for the entire network we use the *Bisection Bandwidth* of the network. The Bisection Bandwidth is defined as the total bandwidth across the minimum number of links that

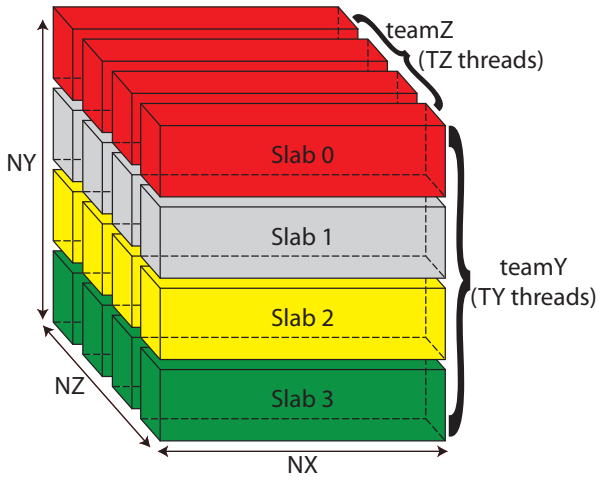

Figure 15: 2-D data partitioning for 3-D FFT. A thread owns $\frac{NZ}{TZ}$ planes and $\frac{NY}{TY}$ rows of NX points where the T threads are laid out in a $TY \times TZ$ thread grid.

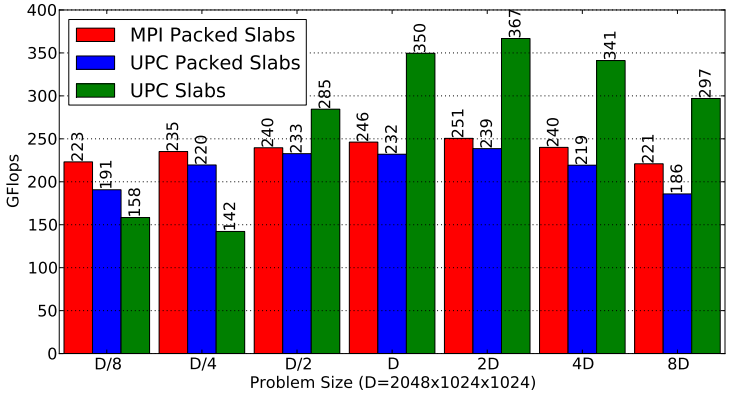

Figure 16: 3-D FFT Performance on 1024 cores of Cray XT4

need to be cut to sever the network into two equal halves. This thus provides an approximation for the aggregate Bandwidth a network can deliver in a large Exchange operation. The total number of flops in the benchmark at each problem size is divided by this time to yield the performance. As the data show, overlapping the collective with the computation yields significant performance improvements (17% at 32,768 cores, for example).

We examine the performance breakdown of the weak-scaling graph in Figure 18 at 32,768 cores on a $4096 \times 4096 \times 2048$ 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, "Other" measures the time for the other parts of the NAS FT benchmark besides the 3D FFT (initial setup, local evolve computation and final checksum), and "Barrier" measures the time spent in barriers. As the data also show, the primary difference in execution time is the time spent on communication. At 32,768 cores the algorithm without communication and computation overlap (Packed Slabs) induces Exchanges of 128KB messages per thread and the algorithm leveraging communication/computation overlap (Slabs) induces Exchanges of 8KB messages per thread. Thus one would expect that the Packed Slabs would be the winner since it is able to realize better bandwidth at higher message sizes. Since Slabs spends fewer amount of time in Communication, we can deduce that the communication and computation are being overlapped and that some of the communication cost is being hidden. Thus the performance advantage of Slabs over Packed Slabs can be attributed to communication and computation overlap.

We have demonstrated the performance of collectives in the NAS FT benchmark. As the data show, the non-blocking collectives in GASNet consistently yield good performance benefits at scale for communication bound problems. By overlapping communication with computation, one can hide the latency of the communication. By leveraging the overlap we are able to deliver a 17% improvement in performance over MPI on 32,768 cores of IBM BlueGene/P.

7. Conclusions

To satisfy scientists' demand for computational power, while at the same time keeping processor clock speeds unchanged, systems are growing rapidly

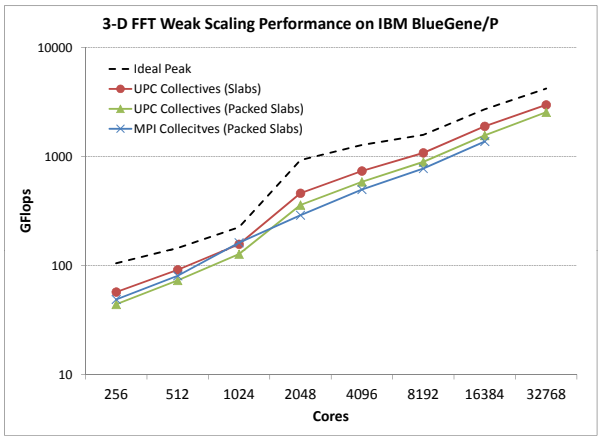

Figure 17: 3-D FFT (NAS FT) Performance on 32K-core IBM BlueGene/P. The UPC Slabs algorithm uses the UPC non-blocking exchange (alltoall) collective. The UPC Packed Slabs algorithm uses the UPC blocking exchange (alltoall) collective. The MPI Packed Slabs algorithm uses the MPI blocking alltoall collective.

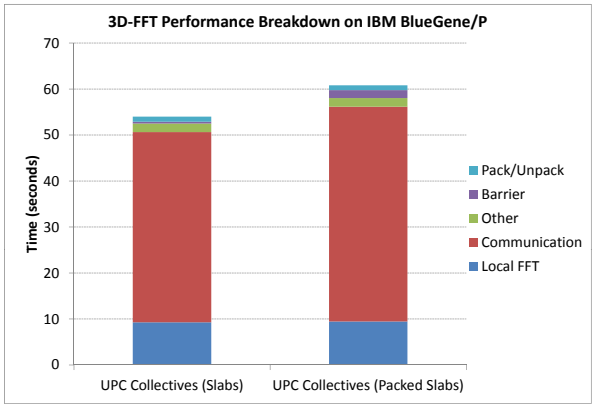

Figure 18: 3-D FFT (NAS FT) Performance Breakdown on IBM BlueGene/P

in the number of cores per node and in the number of nodes per system. As communication costs continue to grow relative to the aggregate performance on a processor chip, the need to carefully optimize communication becomes critical. Moreover, future high end systems are likely to be power limited, and communication to memory and between processors is a significant component of the power budget for a machine. This puts particular pressure on the runtime systems for the programming models to perform well on a mixture of shared memory and distributed memory hardware. A good implementation of collective communication operations is invaluable in providing fast, reusable code for common programming idioms that arise across a wide range of application domains.

We have presented a new collective communication infrastructure in GASNet that includes a large set of possible implementations for each collective operation in a framework that allows for automatic search and selection of the best implementation for a given instantiation. We explored some of the semantic issues with PGAS collectives, which are related to whether there is an implicit global synchronization before and after each collective. We found substantial performance gains–sometimes nearly an order of magnitude–for using loose rather than strict synchronization in microbenchmarks.

In addition, we explored the use of collectives in three application benchmarks written in the UPC language: matrix multiplication, Cholesky factorization, and a 3D FFT. We used the Berkeley UPC compiler and runtime system on three large HPC system systems, and take advantage of extension of the UPC collectives to include non-blocking and team-based collectives. We demonstrated that the applications benefited from the use of non-blocking collectives, which explicitly allow collectives to be overlapped beyond what is possible with UPC's collective synchronization modes. With these extensions, the benchmarking results show the performance advantages of leveraging a one-sided communication model in programs that have significant collective communication. The performance is comparable to MPI in the worst case, and often significantly better.

The UPC community has extensively debated the need for collective communication, since they represent a bulk-synchronous programming style that is not in keeping with the irregular applications with random access data structures that were the original motivation for the language. In this paper we show that the one-sided PGAS model has advantage even for bulk-synchronous programs, and that the collectives are a convenient and efficient way to express some computations. In addition, the collective abstraction al-

lows runtime system programmers who are knowledgeable about the specifics of a high performance network or memory system to develop optimized collectives which can then be used by the application community. Given the trends in computer hardware, in which most of the cost and power goes into moving data rather than computing on it, we believe highly optimized, expressive collective libraries will be essential to the success of PGAS languages.

Acknowledgments

This research was supported in part by the Department of Energy (DE-FC03-01ER25509, DE-FC02-07ER25799, DE-AC02-05CH11231) and by the National Science Foundation (OCI-0749190). It made use of resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, the National Energy Research Scientific Computing Facility (NERSC) at Lawrence Berkeley National Laboratory, and the Oak Ridge Leadership Computing Facility at Oak Ridge National Laboratory, which are supported by the Office of Science of the U.S. Department of Energy under contracts DE-AC02-06CH11357, DE-AC02-05CH11231 and DE-AC05-00OR22725, respectively. It also used resources at the Texas Advanced Computing Center (TACC) at the University of Texas at Austin.

References

[1] D. Bonachea, GASNet Specification, Technical Report CSD-02-1207, University of California, Berkeley, 2002.

[2] R. W. Numrich, J. Reid, Co-array fortran for parallel programming, SIGPLAN Fortran Forum 17 (1998) 1–31.

[3] UPC, UPC Language Specifications, v1.2, Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[4] P. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, K. Yelick, Titanium Language Reference Manual, Tech Report UCB/CSD-01-1163, U.C. Berkeley, 2001.

[5] B. L. Chamberlain, D. Callahan, H. P. Zima, Parallel programmability and the chapel language, International Journal of High Performance Computing Applications 21 (2007) 291–312.

[6] R. Nishtala, Architectural Probes for Measuring Communication Overlap Potential, Master's thesis, UC Berkeley, 2006.

[7] R. Brightwell, S. P. Goudy, A. Rodrigues, K. D. Underwood, Implications of application usage characteristics for collective communication offload, Int. J. High Perform. Comput. Netw. 4 (2006) 104–116.

[8] T. Hoefler, A. Lumsdaine, W. Rehm, Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI, in: Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07, IEEE Computer Society/ACM, 2007.

[9] R. Nishtala, Automatically Tuning Collective Communication for One-Sided Programming Models, Ph.D. thesis, University of California, Berkeley, 2009.

[10] D. R. Butenhof, Programming with POSIX threads, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

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

[12] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, D. Weathersby, Efficient algorithms for all-to-all communications in multiport message-passing systems, IEEE Transactions on Parallel and Distributed Systems 8 (1997) 1143–1156.

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

[14] Top500 List: List of top 500 supercomputers. `http://www.top500.org/`.

[15] C. Bell, D. Bonachea, R. Nishtala, K. Yelick, Optimizing bandwidth limited problems using one-sided communication and overlap, in: The 20th International Parallel and Distributed Processing Symposium (IPDPS 2006).

[16] R. van de Geijn, J. Watts, Summa: Scalable universal matrix multiplication algorithm, TR-95-13, Department of Computer Sciences, University of Texas (1995).

[17] HPL website, http://www.netlib.org/benchmark/hpl/algorithm.html.

[18] 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, S. K. Weeratunga, The NAS Parallel Benchmarks, The International Journal of Supercomputer Applications 5 (1991) 63–73.

[19] R. Nishtala, P. H. Hargrove, D. O. Bonachea, K. A. Yelick, Scaling Communication-Intensive Applications on BlueGene/P Using One-Sided Communication and Overlap (IPDPS 2009).