Automatically Tuning Collective Communication for One-Sided Programming Models

Rajesh Nishtala

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-168
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-168.html

December 15, 2009

Automatically Tuning Collective Communication for One-Sided Programming Models

by

Rajesh Nishtala

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Katherine A. Yelick, Chair
Professor James W. Demmel
Professor Panayiotis Papadopoulos

Fall 2009

Automatically Tuning Collective Communication for One-Sided Programming Models

Abstract

Automatically Tuning Collective Communication for One-Sided Programming Models

by

Rajesh Nishtala

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Katherine A. Yelick, Chair

Technology trends suggest that future machines will rely on parallelism to meet increasing performance requirements. To aid in programmer productivity and application performance, many parallel programming models provide communication building blocks called *collective communication*. These operations, such as Broadcast, Scatter, Gather, and Reduce, abstract common global data movement patterns behind a simple library interface allowing the hardware and runtime system to optimize them for performance and scalability.

We consider the problem of optimizing collective communication in Partitioned Global Address Space (PGAS) languages. Rooted in traditional shared memory programming models, they deliver the benefits of sophisticated distributed data structures using language extensions and one-sided communication. One-sided communication allows one processor to directly read and write memory associated with another. Many popular PGAS language implementations share a common runtime system called GASNet for implementing such communication. To provide a highly scalable platform for our work, we present a new implementation of GASNet for the IBM BlueGene/P, allowing GASNet to scale to tens of thousands of processors.

We demonstrate that PGAS languages are highly scalable and that the one-sided communication within them is an efficient and convenient platform for collective communication. We show how to use one-sided communication to achieve 3× improvements in the latency and throughput of the collectives over standard message passing implementations. Using a 3D FFT as a representative communication bound benchmark, for example, we see a 17% increase in performance on 32,768 cores of the BlueGene/P and a 1.5× improvement on 1024 cores of the CrayXT4. We also show how the automatically tuned collectives can deliver more than an order of magnitude in performance over existing implementations on shared memory platforms.

There is no obvious best algorithm that serves all machines and usage patterns demonstrating the need for tuning and we thus build an automatic tuning system in GASNet that optimizes the collectives for a variety of large scale supercomputers and novel multicore architectures. To understand the large search space, we construct analytic performance

models use them to minimize the overhead of autotuning. We demonstrate that autotuning is an effective approach to addressing performance optimizations on complex parallel systems.

Dedicated to Rakhee, Amma, and Nanna for all their love and encouragement

Contents

List of Figures

Acknowledgments

Firstly, I would like to thank my advisor, Kathy Yelick, for all her guidance, patience, enthusiasm, and encouragement through the years. Since first introducing me to the BeBOP group she has been a fantastic mentor. In addition to her invaluable technical advice, Kathy always ensured that I remained focused on the larger research and career goals. Her energy is inspiring.

I also owe a lot to Jim Demmel for his advice through the years. I will never forget how he spent a couple of days out of his busy schedule to patiently help me edit my first technical report, "When Cache Blocking Works and Why." His ability to explain complex mathematical concepts has allowed me to understand many interesting research projects outside my area of expertise. I am very fortunate to have Kathy and Jim as my mentors.

I greatly appreciate the help Dave Patterson and Panos Papadopoulos have given by agreeing to sit on my dissertation committee. Their feedback during the qualifying exam as well as during the dissertation was very useful.

I would like to thank the entire Berkeley UPC team, particularly Paul Hargrove, Dan Bonachea, Christian Bell, Costin Iancu, Wei Chen, Yili Zheng and Jason Duell for all their help. I would not have been able to do any of the work described in this thesis without their advice, foundations, and frameworks upon which my work was built.

Kaushik Datta has been my officemate for the past few years and we seem to find ourselves in similar situations and deadlines. He has been a great friend and collegue with whom I could discuss any subject related or not to computer science. Our Birkball games, Xbox breaks, Cheeseboard runs, and occasional ski trips to Lake Tahoe have made even the most frustrating days of graduate school seem fun.

I would like to thank all the members of the BeBOP group for their time and support to discuss my ideas. Rich Vuduc provided me guidance and assistance early on and is a great role model. I also would particularly like to thank Mark Hoemmen, Shoaib Kamil, Sam Williams, Marghoob Mohiyuddin, Ankit Jain, and Eun-Jin Im for their useful insights and suggestions on my work.

Outside of Berkeley I would like to thank George Almasi and Calin Cascaval at IBM for their help and mentorship during my summer internship and various conferences. They have provided an invaluable outside perspective.

In the Parallel Computing lab I found a nice sense of community. I had a chance to discuss parallel and distributed computing ideas with several of my colleagues who helped me frame my work so that it appeals to a broader audience. In particular I would like to thank Heidi Pan, Chris Batten, Krste Asanovic, Benjamin Hindman, Jimmy Su, Amir Kamil, Sarah Bird, Andrew Waterman, Bryan Catanzaro, and Jike Chong for their stimulating discussions and questions. I would also like to thank Jon Kuroda, Jeff Anderson-Lee, Tammy Johnson, and Laura Rebusi for all their help.

Rakhee, I could not have finished this without you. You always believed in me even when I didn't. Your love and emotional support mean more to me than I can ever put into words. Guess what, there's an "arm" between us now!

Finally, I am deeply grateful to my parents. Ever since my first days in elementary school you have always taught me to never cut corners, be meticulous and patient, and give

everything I have to whatever I do. These are critical lessons that I have relied on over the years. I will never be able to thank you enough for your sacrifices to ensure that I received a quality education. We've come a long way from the little boy that used to cry at the gate saying he didn't want to go to school 25 years ago.

This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. In addition this project was also supported in part by the Department of Energy under contracts DE-FC02-06ER25753 and DE- FC02-07ER25799. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. In addition this research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02- 06CH11357. This research also used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725. The Texas Advanced Computing Center (TACC) at The University of Texas at Austin provided HPC resources that have contributed to the research results reported within this dissertation. This research was also supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227).

Chapter 1

Introduction

As computational power continues to grow at an exponential rate, many large-scale parallel and scientific applications that were once considered too large and expensive are now within reach. Scientists are able to solve many interesting problems that impact all aspects of society, but many scientific simulations still require orders of magnitude more processing power than what is currently available today. In the past, the growth in high end computing needs have always been met by a combination of increased clock rate and increasing processor counts. However, given the recent technology limits[17], clock rates will remain virtually constant and future increases will be achieved almost entirely by increasing the number of processor cores.

This puts us at an interesting cross-roads in parallel computing. Currently the only way to scale the performance of a large parallel machine is to add many more processors and create better networks to connect these processors together. Current petascale supercomputers have hundreds of thousands of processor cores, and as the numbers of processor cores grow, the efficiency of many applications will be adversely affected. Applications that were only designed to scale to thousands of processors will see very poor scalability as we try to run them on the next generation of parallel machines. This will force us to reexamine some of the fundamental ways that we approach designing and using parallel languages and runtime systems.

Communication in its most general form, meaning the movement of data between cores, within cores, and within memory systems, will be the dominant cost in both running time and energy consumption. Thus, it will be increasingly important to avoid unnecessary communication and synchronization, optimize communication primitives and schedule communication to avoid contention and maximize use of bandwidth. The wide variety of processor interconnect mechanisms and topologies further aggravate the problem and necessitate either (1) a platform specific implementation of the communication and synchronization primitives or (2) a system that can automatically tune the communication and synchronization primitives across a wide variety of architectures. In this work we focus on the latter.

A new class of languages, called Partitioned Global Address Space (PGAS) languages, has emerged to aid in the performance and scalability of high performance applications [171].

Rooted in traditional shared memory programming models, these languages expose a global address space that is logically partitioned across the processors. The global address space allows programmers to create sophisticated distributed data structures that aid in building both regular and irregular applications. In order to present the global address space that is similar to traditional shared memory programming, these languages use one-sided communication. In this communication model, a processor is allowed to directly read and write the data of another without interrupting the application on the remote processor. Such semantics have been shown to increase performance by decoupling the synchronization from the data movement that is present in a two-sided communication model.

To aid the productivity of the application writers, many popular parallel languages and libraries, such as Unified Parallel C (UPC)[159] and the Message Passing Interface (MPI)[125], provide *collective* operations. These collective operations encapsulate common datamovement 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 dissertation will be to understand and improve the performance and productivity benefits of collective operations in PGAS languages (such as UPC, Titanium[96], and Co-Array Fortran[132]) and libraries (such as MPI-2[124]) that rely on one-sided communication. The semantic differences between one- and two-sided communication pose interesting and novel opportunities for tuning collectives.

While both MPI and the PGAS languages were designed for scientific computing, the problem of optimized collective communication has the potential to also impact applications written in the popular MapReduce model [67, 41]. In this programming model, the program is broken into two phases: Map and Reduce. In the Map phase the input data is transformed into a set of key-value pairs. In the Reduce phase all the values with the same keys are aggregated together. In addition, work by Michael Isard et al. [104] on the Dryad system has shown how the MapReduce framework can be extended to handle arbitrary directed acyclic computational graphs. Scheduling the computation and performing the aggregation operations found in these models in a scalable way are similar to the collective operations that we present in this work. Thus we argue that the techniques that we describe in this work are not limited to scientific computing.

As system vendors work to develop systems with unprecedented levels of concurrency, we are also witness to a large variety of processors and the networks that connect these processors together, which, combined with systems scale, makes it prohibitively expensive to hand-tune any collective implementation for each of these different machines. This necessitates a system that can automatically tune these collectives on a wide variety of processor architectures and networks. Automatic tuning is the method in which a library generates a set of implementations for the same algorithm and selects one of them based on a combination of search (i.e. running the implementation and measuring the performance), performance models, and other heuristics. The technique of automatic tuning has been proven

successful in a wide variety of serial applications such as Dense Linear Algebra[32, 166], Sparse Linear Algebra[99, 164], and spectral methods[85, 144]. Part of this dissertation will explore the design and construction of a system that can automatically tune the collectives. These automatically tuned collectives are incorporated into GASNet[35], the highly portable runtime layer currently used in several PGAS language implementations. To start, we will present a new implementation of GASNet designed for the BlueGene/P architecture, which serves as both a highly scalable platform for our collective tuning work and an interesting and valuable implementation of GASNet in its own right. This work has implications for many parallel language efforts, because GASNet is used in the Berkeley UPC[30] compiler, the GCCUPC compiler [92], Titanium[96], the Rice Co-Array Fortran[132] compiler, Cray Chapel[6] and other experimental projects.

1.1 Related Work

Since collective communication is such a critical part of many applications, there have been many papers and projects that have been devoted to optimizing these collectives. They have ranged from projects that provide algorithmic optimizations of these operations [44, 20, 79, 155, 118, 107] to work that focuses on optimizing collectives for very specific networks[145, 117, 14]. Most of these consider collectives in a two-sided communication model. Our main contribution is analyzing and optimizing collectives in a one-sided communication model that is found in many PGAS languages. The algorithms that are found in the literature will be part of our search space. Our work will determine how effective these algorithms are in practice in a one-sided communication model.

Many researchers have studied collective communication in the context of MPI, e.g., [142, 154]. All collective functions in the MPI 2.0 specification are blocking. Hoefler et al [97] discuss the implementation of non-blocking MPI collective operations in LibNBC and evaluate the communication and computation overlapping effects in applications. GASNet collective functions are non-blocking and support 9 different combinations of synchronization modes which enable more aggressive communication overlapping but makes the tuning space much larger. In addition, the GASNet collective implementation tailors the collective operations for one-sided communication model and targets the collective needs for Partitioned Global Address Space languages.

1.1.1 Automatically Tuning MPI Collective Communication

Automatic tuning [166, 85, 164] is a widely used technique in high performance computing for accommodating rapidly changing computer architectures, different machine configurations and various application input data. Applying automatic tuning to communication optimization is not a new technique. Brewer[42] discusses various techniques to automatically tune a communication runtime for a variety of platforms. In this work, techniques for tuning the communication alongside the application kernels to present good performance were shown. Our work draws on some of these concepts and helped guide the software architecture for the automatic tuning system, but our work goes well beyond these techniques

by discussing systems at very large scale; our largest experiments are conducted with 32,768 processor cores.

Recent related work has also focused on automatically tuning the collective operations found in MPI. Pješivac-Grbović et al [142] present a system to automatically select the best collective implementation by a decision algorithm based on quadtree data structure. The quadtree is used during application run-time to pick the best algorithm for a given collective and to minimize the time needed for search. Star-MPI [80] also uses online autotuning to dynamically tune collective operations for different application workloads.

Our automatic tuning system will examine a similar set of parameters as these other projects (such as processor count, message size, machine latency, machine bandwidth, etc.), however the addition of the looser synchronization modes and the impact of overlapping communication and computation through collectives will be novel additions to the automatic tuner. In addition, our collective library targets programming models that use one-sided communication which is different than the message passing model examined by previous work. The mechanisms, such as quad-trees, that previous work has shown have influenced the software architecture for the automatic tuner. Our system is also the first portable high performance implementations of the collectives for PGAS languages.

1.2 Contributions

In this dissertation we will show how programming models that use one-sided communication provide unique opportunities for optimizing collective communication. Since PGAS languages, which are the main set of languages that use one-sided communication, are designed with productivity and performance portability in mind, any optimized collective library that is designed must be able to adapt to whatever environment the user desires. We will also show that an automatic tuning system that will chose the best algorithms of these collectives on a wide variety of processors and interconnects can be built.

This dissertation makes the following contributions to the field:

- We outline the difference between one- and two-sided communication and show how it impacts the design and implementation of the collective communication library.
- We describe our implementation of the one-sided GASNet communication layer for the IBM BlueGene systems. We use this to demonstrates scalability of the GASNet interface and UPC language on top of it to tens of thousands of processors.
- We showcase the need for automatic tuning of collectives by studying a large set of algorithms for the major collective communication operations on both shared and distributed memory platforms and show how a few example applications can benefit from optimized collective communication. We target both large clusters with different network and processor configurations as well as modern multicore systems.
- Our distributed memory performance results demonstrate that this new collective communication library, which selects from a large set of possible algorithms, achieves

up to a 65% improvement in performance over MPI for a one-to-many collective and up to a 69% improvement in performance over MPI for a many-to-many collective.

- Results show that tuning collectives for shared memory decrease the latency for a Barrier synchronization by two orders of magnitude compared to what is found in traditional libraries. The results also show that by further tuning the collective communication for these platforms we can realize another 70% improvement in overall latency.

- We also demonstrate how the collectives can improve the performance of higher level algorithms largely due to better overlap with computation to achieve a 46% improvement in performance over MPI for a 3D FFT, a communication bound benchmark, on 1024 cores of the Cray XT4. The tuned collectives also show a 1.86× improvement in the performance of Parallel DGEMM on 400 cores of the Cray XT4 when compared against the PBLAS [52] and comparable performance for Dense Cholesky factorization on 2k cores of the IBM BlueGene/P. By tuning the collectives in Sparse Conjugate Gradient on shared memory platforms we see a 22% improvement in overall application performance when comparing against the untuned collectives.

- We construct performance models for these various algorithms using the LogGP framework to better understand performance tradeoffs in the different algorithms and to identify performance upper bounds that can serve as useful limits when tuning.

- We describe a software architecture that can *automatically tune* these operations for a variety of processor and network types and show how performance models can be used to prune the search space and aid in the automatic tuning process. In many cases the performance model picks the best algorithm, negating the need for search. However when search is needed, the performance model can guide the search and cut down the time needed for a search.

- We propose a novel interface to the collective communication library that is designed for Partitioned Global Address Space Languages. The interface allows users to specify the data that the collectives act upon rather than the traditional methods which rely on the user specifying exactly which threads are involved in the collective.

- We present the first automatically tuned collective library available for Partitioned Global Address Space Languages and use it in some of the largest scale runs of any PGAS programs to date.

1.3 Outline

The discussion starts with how modern systems are organized into compute nodes and how these nodes are connected together to form a large parallel system in Chapter 2. The design of these systems reveals the different mechanisms for communicating between cores and hence a non-uniform access time when data is located on two different nodes. In

Chapter 3 we show how the Partitioned Global Address Space languages address this and particularly how the one-sided communication model found in these languages aims to aid productivity and performance. We also detail how a one-sided communication library can be written on top of modern network hardware taking advantage of a variety of the available features. As a case study we use the IBM BlueGene/P.

Having built up a one-sided messaging framework we then turn our attention to the collective communication operations, operations designed to perform globally coordinated communication. We start by introducing the different collectives in Chapter 4 and highlight how the collectives found in Partitioned Global Address Space languages differ from MPI collectives in subtle but important ways. Chapters 5 and 6 go into further detail about how these different collective operations can be implemented and present benchmarks showing performance on our experimental platforms. Along with discussing the various algorithms we also show how performance models can be constructed to better understand the performance and aid in choosing the optimal algorithms.

While collective communication is normally targeted at distributed memory systems, collective communication operations can also be useful on shared memory systems built from one or more multicore chips. These arise either within the nodes of a larger distributed memory platform or on their own. The collective operations often stress the limits of the shared memory structures on these systems. As core counts continue to grow at a rapid pace, the number of cores within a chip and within a multicore system will soon be large enough where naïve collective algorithms realize poor scalability. We show how tuning a collectives library can aid in performance for these platforms in Chapter 7.

One of the continuing themes throughout the dissertation is to show the wide variety of algorithms (and parameters) that are available to perform a particular collective operation. In Chapter 8 we show the software architecture that can automatically tune these operations for a wide variety of platforms. We also show how the performance models can play a critical role to reduce the overheads associated with tuning without sacrificing the quality of the resultant algorithms.

In many applications, it is useful to perform collective operations over a subset of the threads rather than all of them. We call this subset a *team*. In Chapter 9 we show two different ways to construct teams. The first is a traditional method in which the user explicitly specifies the members. The second allows the user to specify only the data that is involved in the collective and it is up to the runtime system to automatically construct the teams. We present how such an interface would work and can be integrated into the language runtime systems.

We conclude the dissertation in Chapter 10 with a summary of the points raised and possible directions for future work.

Chapter 2

Experimental Platforms

Over the past several years we have witnessed to a tremendous increase in computational power, more than three orders of magnitude in ten years on the fastest machines in the world. In order to facilitate scientific discovery, computer engineers continually deliver machines that can deliver significantly more computational power, and until very recently, these increases in computational power have been delivered by increasing the clock rates of the processors and creating novel hardware mechanisms that allow serial applications to run faster. However, due to the technical challenges associated with constantly delivering computational improvements through this method [17], the performance improvements are now being delivered by connecting more processors together to create systems that have 10s of thousands and 100s of thousands of processors and relying on parallelism (and hence the programmer) to deliver the performance. Modern high end systems are built as a hierarchy in which different processors communicate through an interconnection network. This interconnection network is built out of a combination of switches that allow all the processors to be connected in some topology, typically some form of mesh or tree. There are many points in the design space on how to connect the switches and busses together [60]. One of the central aims of this dissertation is to analyze how to effectively utilize these different interconnection networks and build a software package that can automatically tune itself to the interconnection network of the platform it is installed on.

To organize the discussion we focus on four different large scale platforms: the Cray XT5 at Oak Ridge National Labs (Jaguar)[105], the IBM BlueGene/P at Argonne National Labs (Intrepid)[103], the Sun Constellation System at the Texas Advanced Computing Center (Ranger)[84], and the Cray XT4 at the National Energy Research Scientific Computing Center (Franklin)[83]. As of November 2009, these machines are respectively ranked 1^{st}, 8^{th}, 9^{th}, and 15^{th} on the Top500 list[5], the list of the 500 most powerful computers in the world. These platforms will also be the test beds for our experimental work. Throughout the rest of this chapter we will discuss some of the features that are common to these interconnection networks and how they differ.

The rest of this chapter is organized from the smallest processing core out to the large data-center switches that connect all the various nodes together.

2.1 Processor Cores

At the center of the systems lies a set of processing cores. These are hardware units that are capable of executing a serial sequence of instructions and update the contents of memory through a connection to the memory system [139]. A core also typically contains some of its own fast memory known as a *cache*. Through advances in modern silicon technology, a few of these processing cores can be placed on the same physical die termed *multicore* processors. Currently, the preferred method of communicating data amongst the various cores is to have them update and read a *shared* address space so that updates to one location in memory are visible to the other cores. Because the per-core caches may also hold copies of data, a coherence protocol is used to ensure that the cached copies are consistent.

This raises issues of when data is safe to read and write. There has been a wide body of literature [60, 169] analyzing how a memory system can be built including special *atomic* instructions, instructions with multiple memory accesses whose effects are guaranteed to be run to completion if the instruction is started. For example, a test and set operation involves read, a test to ensure that the variable has not been set, and then a write operation to set it. Atomicity ensures that no other core can access the variable during this sequence of operations. With these special mechanisms synchronization constructs are built to ensure that the data written to the shared memory space is safely written and read. In this dissertation we focus more on how these cores communicate efficiently with each other rather than focusing how to leverage the best performance out of these cores. The latter is the subject of much other related work [138].

2.2 Nodes

A *node* is a collection of multicore processors, the associated memory and the interface to the network. For all our platforms, all the cores within a node have access to the entire memory space on that node through a cache coherence shared memory system. In this section we go into greater depth of the different node architectures of our platforms as well as an important common feature to all of them that we will leverage for our communication system, Remote Direct Memory Access (RDMA).

2.2.1 Node Architectures

Each of the different platforms has a different node architecture. They have many similarities amongst them but they have key differences that make them interesting for our analysis. The following architectural discussion highlights some of the salient features. We focus on those features that are directly relevant to the shared memory and communication issues that are relevant to collective communication.

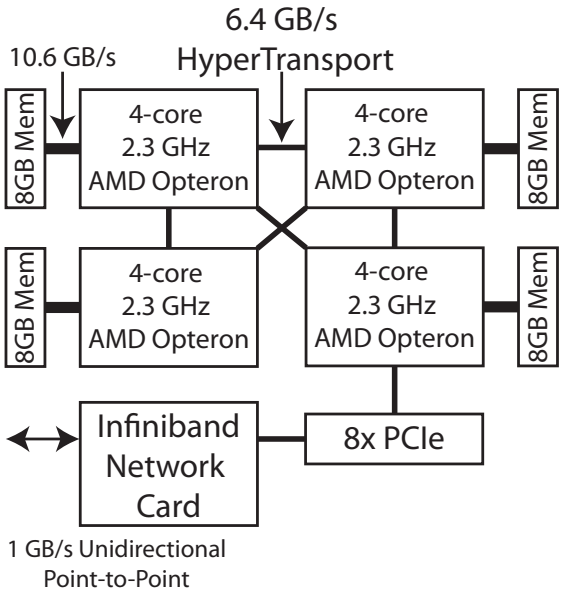


Figure 2.1: Sun Constellation Node Architecture

Sun Constellation

We first start our analysis with a "traditional" cluster node. The work by Culler et al. [59] and Sterling et al. [152] among others showed how to connect commodity machines through a network to produce a system that can deliver significant performance through commodity parts. As our representative example for this class of systems we choose the Sun Constellation system. The compute node of the Sun Constellation system is a SunBlade x6420 [7]. A simple block diagram of this is shown in Figure 2.1. The node contains 4 AMD quad-core Opteron (Barcelona) chips for a total of 16 cores. Each of the sockets are connected together through AMD's Hypertransport interface capable of a bandwidth of 6.4GB/s between the chips. The sockets are connected to their own memory at a much larger 10.6 GB/s. Thus, even though the system appears to have a uniform shared memory, there is a notion of locality. Accesses to memory attached to the socket will be about twice as fast as accesses to another socket's memory. Each of the sockets has three Hypertransport links. One of the sockets must dedicate one of these links for the network interface and thus two out of the four sockets are connected fully connected and the other two are not directly connected to each other.

Cray XT

The next platform we analyze is the Cray XT4[9]. A block diagram of the node is shown in Figure 2.2. The compute node of the Cray XT4 has a single quad-core 2.3 GHz AMD Opteron (Budapest). Attached to the processors is 8 GB of DRAM connected over a link that delivers 10.6 GB/s. In addition the processor is directly attached to the SeaStar2

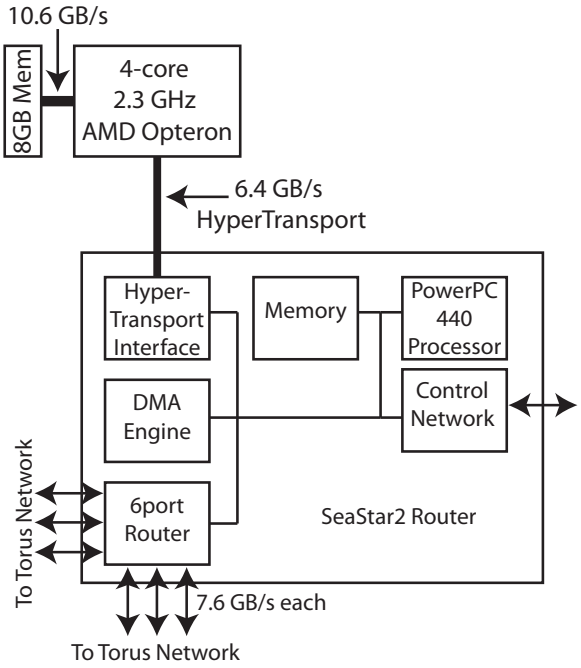


Figure 2.2: Cray XT4 Node Architecture

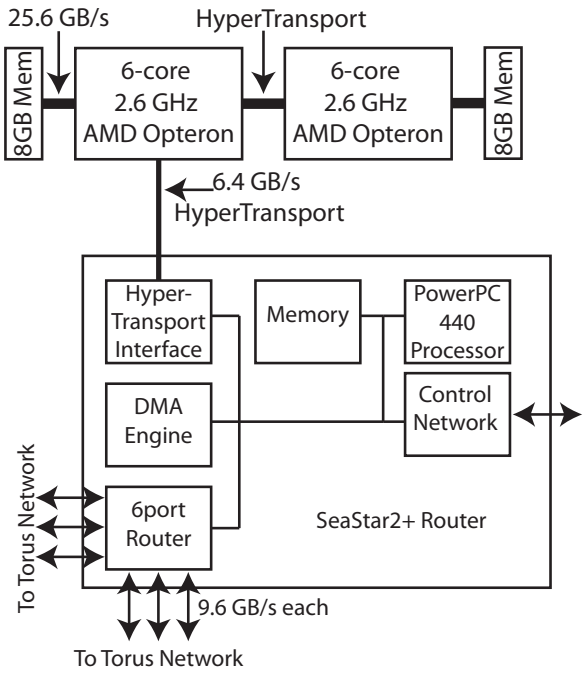


Figure 2.3: Cray XT5 Node Architecture

routing chip over a 6.4GB/s AMD Hypertransport link. This routing chip has 6 links to the rest of the network to connect the node to the rest of the system in a 3D-torus network. Each of the 6 links transport data in a different direction in the 3-dimensional space. We discuss the 3D-torus and how the different network chips are connected together in much greater depth in Section 2.3. Unlike "traditional" cluster architectures the nodes that compose the Cray XT4 system are not commodity components. It has been customized so that the network interface connects directly into the processor rather than through a normal I/O bus that most systems have. This system also only has one socket and thus all the cores within the node are equidistant from each other unlike the Sun Constellation.

The Cray XT5 is the next generation version of the Cray XT4. The compute nodes of the Cray XT5 (shown in Figure 2.3) have two hex-core AMD Opteron processors (Istanbul) running at 2.6 GHz leading to a total of 12 cores per node. One of the processors is directly connected to a SeaStar2+ routing chip (an updated version of the SeaStar2).

IBM BlueGene/P

The last platform that we analyze is the IBM BlueGene/P[150]. The BlueGene/P is a custom design from IBM. It uses 4 IBM PowerPC 450 cores running at 850MHz connected to another 3D torus network. The four cores are connected to the L3 through eight 13.6 GB/s unidirectional links for a total of 54.4 GB/s in each direction. The L3 cache is connected into the memory at 13.6 GB/s. Figure 2.4 shows a high-level block diagram of the architecture.

Unlike the other platforms though the network is much more tightly integrated with the processing elements and node elements. The torus network interface, for example, is able

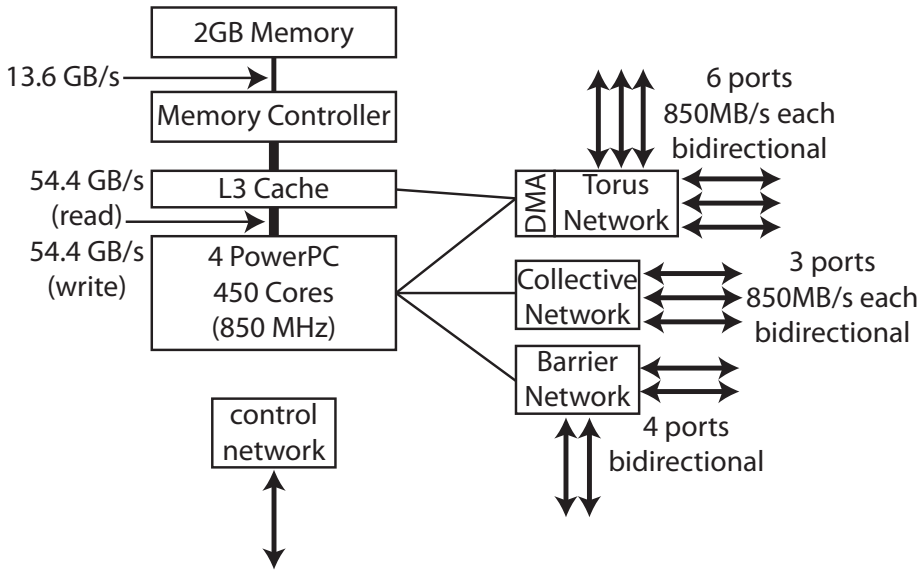


Figure 2.4: IBM BlueGene/P Node Architecture

to talk directly to the L3 cache rather than having to go through a Hypertransport like device. This platform also has a custom collective network and barrier network to support hardware collectives.

2.2.2 Remote Direct Memory Access

One of the features common to all the 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 memory (or the L3 cache in the case of the BlueGene/P) without having the processors actively manage communications that are in progress. In addition these devices also allow remote nodes to directly read or write the memory hence we call this feature Remote Direct Memory access or (RDMA). As we will show this is a natural fit to the one-sided programming model that will be discussed in greater depth in Chapter 3.

In order to allow such mechanisms to properly work, the operating system pages that are being written or read will need to be resident in the memory. Related work [26] has developed a system that can manage these resources to enable efficient communication with RDMA. Our related work [128, 28] has also shown how these features can be utilized to realize significant performance advantages. We go into much greater depth about these concepts and how they relate to the collective communication in Sections 5.1.5 and 6.3.

2.3 Interconnection Networks

The next step in the hierarchy is to analyze the network topologies that connect the different nodes together. In an ideal environment all the nodes are directly connected to each other to minimize the time spent transferring the data. However, this ideal case would require that each node have N hardware connection endpoints (where N is the total number of nodes) leading to a total of N^2 connections amongst them in the network. For any significant value of N this approach quickly becomes infeasible. Thus, to minimize the amount of network resources, data is transferred by routing it over many links before getting to the final destination, which adds a notion of distance between nodes. We define the distance metric here as the number of links between two communicating nodes. Our experimental platforms present two very different approaches to constructing network topologies: a CLOS network (or Fat Tree), which is a network built hierarchically out of switches; and a Torus network in which the nodes are laid out in a mesh. In the latter, each node is directly connected to its peers.

In all the networks we analyze, data is sent between the nodes through the network by quantizing the transfers into units called *packets*. These packets are then switched and routed through the network until they reach the final destination. One of the keys to a packet switched network is that the nodes do not need to have knowledge of how to send the data. The network contains all of the intelligence to properly route the packets. In addition, since the data is quantized into different packets, there is no need for all the packets between a source and destination node to take the same path through the network. In a torus network, for example, there are many paths between two nodes that have the same Manhattan distance and hence any one of these routes is equally valid. There have been many studies and related work[140] analyzing the performance and optimal routing techniques for these packets. While packet switched networks are the most common today, there are other network routing technologies, such as wormhole routing [16], however these are beyond the scope of this dissertation.

2.3.1 CLOS Networks

At the core of many modern interconnection networks is a *switch*. A switch is a device that will look at the destination address of a packet and route it to the appropriate output port. In our work we consider the switch as a black box that can route data from any port to any other port on the switch. These switches can be built up in a hierarchy. However the more switches that are added between the source and destination nodes, the higher the latency between them.

Figure 2.5 shows two example switch configurations with 4-port switches. In the example on the left, two out of the four ports are connected to other switches. Thus to send a packet from node 0 to node 2 the data will need to traverse through two switches. Another possible way to build up the switches in a hierarchy is to arrange them so that for each P port switch $P-1$ ports are connected to different nodes and the last one is connected to a higher level switch (as shown in the right of Figure 2.5). At the higher level we have one

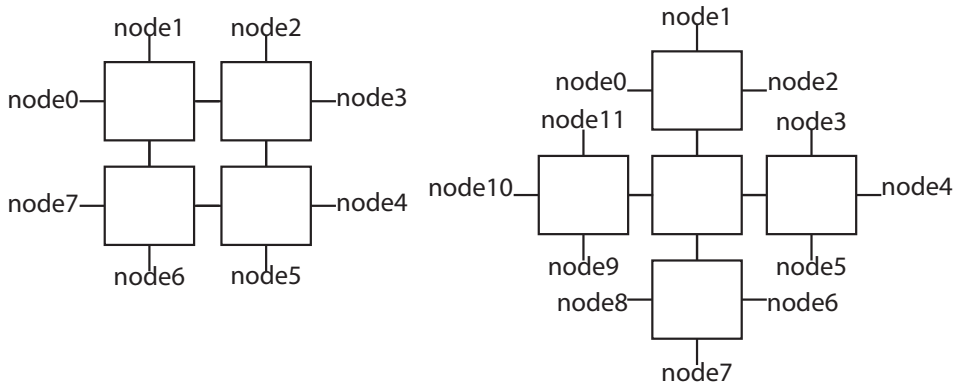


Figure 2.5: Example Hierarchies with 4 port switches

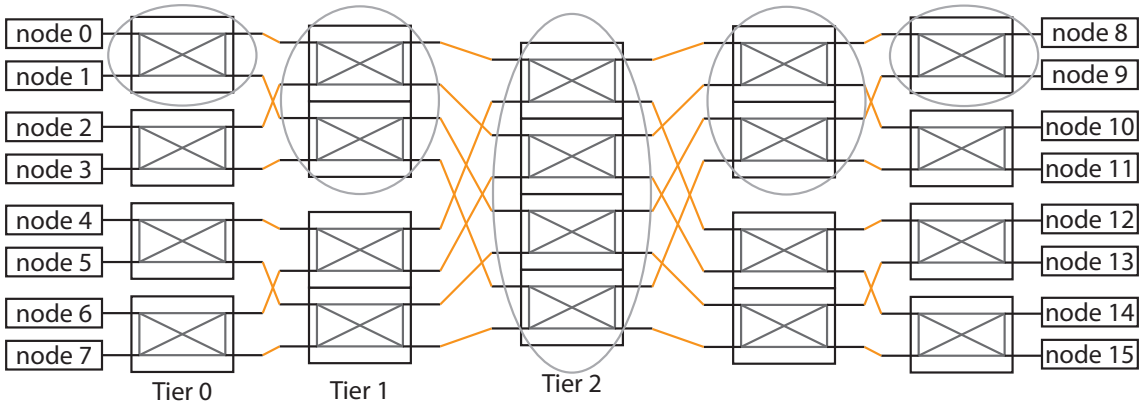


Figure 2.6: Example 16-node 5-stage CLOS Network

switch that connects all the other lower level switches. This approach allows many more nodes to be connected into the network. However, to communicate with a node that is not co-located on the same switch it will take an additional two hops adding to the latency of the communication. Further aggravating the problem, if the link to the switch has bandwidth B then all the nodes that are co-located on the switch will have to share the bandwidth to the central switch and thus effectively delivering a bandwidth of $B/3$. For large switching networks this can be a problem since it dramatically reduces the network performance.

To circumvent this bottleneck Charles Clos demonstrated how to connect many P port switches to form a larger switched network [54]. The key observation is that by adding redundant switches in the middle of the network one can get better end-to-end bandwidth. The switching network is broken up into T tiers. At tier 0 half of the ports of the switches are connected to the nodes and the other half are connected to the internal switching network. They are connected in a butterfly pattern similar to the one used in the Fast Fourier Transform (FFT) [58]. We can reclassify ports of the switch as "inputs" and "outputs."

For a four port switch there are 2 inputs and 2 outputs. At each level the switches are grouped into sets of $(P/2)^t$ switches where P is the number of ports on the switch and t is the tier number. For a fully connected four port switch network there are 16 nodes with 8 switches at the tier 0 level (shown in Figure 2.6). At tier 1 there are 8 switches organized in groups of 2 and at the final level there are 4 switches in a group of 4. To simplify the explanation we analyze the first group of switches at each tier and look at their connectivity. The extension to the rest of the switching network is straightforward. The total number of groups that connect to a higher level can be defined as $P/2$. For a group at tier t there are $(P/2)^{(t+1)}$ inputs and outputs. The i^{th} output from group g at tier t is connected to the $((P/2) \times i + g)^{th}$ input at tier $t+1$. In the example in Figure 2.6 to send a packet from node 0 to node 4 the packet will traverse 5 stages in the switching network. However, to send a message from node 0 to node 3 it will only need to traverse 3 stages since the switches at tier 1 will not forward the data. This gives the rise to a notion of locality in such a network. Nodes can be "close" (few network hops) or "far" (more network hops).

Nodes are connected to the switching network with one link. However, notice that the number of links that connect adjacent tiers double. There are twice as many links that connect groups in tier 0 to tier 1 than nodes to the switching network. Hence, this network has the desirable property that there is a uniform bandwidth between any pair of nodes, also termed a "Fat Tree." Thus different groups of nodes can communicate with each other without interfering on or being interfered by traffic amongst other nodes hence alleviating the problem with our original switching networks. The Fat Tree configuration however comes at a cost of many more switches.

A theoretical property that is important for analyzing networks is the *bisection bandwidth* of the network. This defined as the sum of the bandwidth across the minimum number of links that need to be cut in order to separate the network into two disjoint parts of equal size. This is an approximation for how efficiently the network can handle communication patterns in which all the nodes talk to all the other nodes. Chapter 6 goes into much greater depth on these communication patterns and their applications. In our example with 4 port switches, 8 links need to be cut to separate the network into two equal halves of 8 nodes each. In general for a full Fat Tree there are as many links to the root switching group as the number of input nodes. Thus Fat Trees are said to have full-bisection bandwidth. As we will shortly show, this is not always the case.

In practice the switches have a lot more ports. On the Sun Constellation switches, each switching chip has 24 ports. Figure 2.7 shows how a real Fat Tree network is implemented as hierarchy of units. At the periphery 12 nodes are connected to the input side of the network. This group of 12 nodes and the associated switch are then connected into two 3 tier (5 stages) CLOS network with 24 port switches. Thus there are a total of seven stages in the network: the 5-stage main switch plus the switches at the periphery for input and output. The second 5-stage switch is provided for extra bandwidth and redundancy. The system has a total 3,936 compute nodes with 16 cores each for a total of 15,744 cores.

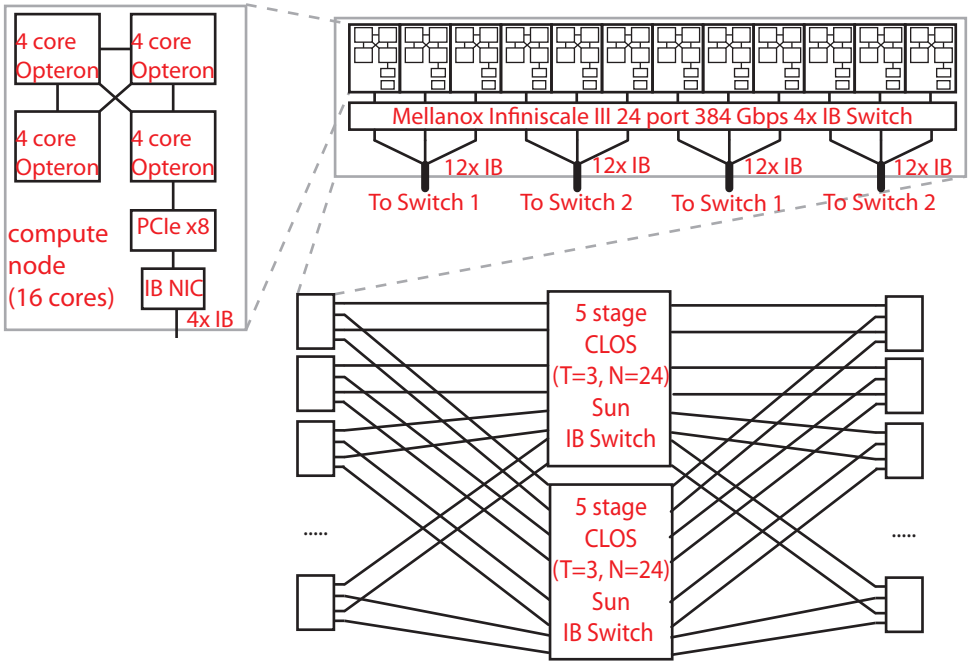


Figure 2.7: Sun Constellation System Architecture

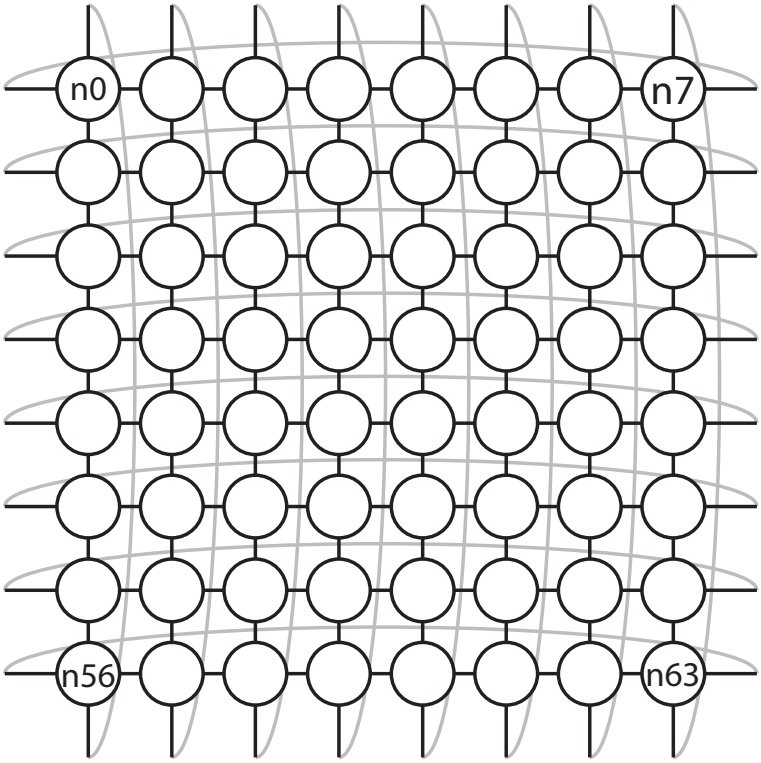


Figure 2.8: Example 8x8 2D torus Network

2.3.2 Torus Networks

One of the main drawbacks with full-CLOS network is the large number of switches that are needed to provide the connectivity. Thus in the second network model, the network interface cards are responsible for both injecting and receiving packets into the network as well as routing the packets within the network. One way to build such a system is to logically lay the processors out in a grid and then connect neighboring processors together to create a mesh. To further increase the network bandwidth and decrease the distances between the corners, the ends of each of the dimensions of the Torus network wrap around. Figure 2.8 shows an example of a set of processors laid out in a torus network. The torus can be generalized to an arbitrary rectangular prism. For a d dimensional torus with an edge of the prism having k nodes, there are a total of k^d nodes that must be connected. In order to send a message from one corner of the torus to the other, there are many possible routes that have the same Manhattan distance and thus choosing the best route on the torus requires the network cards at each of the nodes to intelligently route the signals. Three of our experimental platforms (the IBM BlueGene/P, the Cray XT4 and the Cray XT5) use this network topology. Logically the neighbors within the torus are directly connected together, however, in a real deployment physical distances between groups of nodes prevent the symmetric network performance between nodes co-located on the same rack and nodes located on different racks. Thus even though these nodes might logically be neighbors there is still a notion of distance that must be taken into account.

The bisection bandwidth of the torus networks however tends to be much smaller than Fat Tree networks with the same node counts. For a d dimensional torus network with k nodes per dimension $2 \times k^{d-1}$ links need to be cut to separate the network into two halves. Thus the bisection bandwidth is an order of magnitude smaller than the total number of nodes. This has important implications about properly mapping applications to the network to ensure that the communication is more localized.

Another more subtle downside to the torus networks are that they require all the points in the torus network to be fully populated with nodes (or at least active network cards) in order to properly handle the routing and deliver the advertised performance. Whereas in a full-CLOS network, since the network card are not relied upon to handle the routing, the system does not have to be fully populated to realize the full performance.

2.4 Summary

In summary, our experimental platforms have a lot of similarities and differences that will make them a good basis for understanding our parallel communication primitives for the rest of the dissertation. Table 2.1 provides a summary of the salient information that will be important for the rest of the dissertation. We will focus on how common communication patterns (namely collective communication) can be optimized for various networks. The platforms have different characteristics that affect the performance, which make them interesting for the analysis. The techniques we describe, however, are will be applicable to a broad range of platforms and not limited to these platforms.

	Cray XT4	Cray XT5	IBM BlueGene/P	Sun Constellation
Machine Name	Franklin	Jaguar	Intrepid	Ranger
Machine Location	NERSC	ORNL	ALCF	TACC
Top500 Rank (November 2009)	15	1	8	9
Processor Type	AMD Opteron (Budapest)	AMD Opteron (Istanbul)	IBM PowerPC 450	AMD Opteron (Barcelona)
Clock Rate (GHz)	2.3	2.6	0.85	2.3
# Cores/Processor	4	6	4	4
# Processors/Node	1	2	1	4
# Cores / Node	4	12	4	16
Peak Perf. /Node (GFlop/sec) †	36.8	124.8	13.6	147.2
Memory BW (GB/s)	10.6	25.6	13.6	10.6
# Nodes	9,572	18,688	40,960	3,936
Network Topology	3D Torus	3D Torus	3D Torus	7-stage CLOS
Network BW (GB/s)	7.6 (2-way)	9.6 (2-way)	0.85 (2-way)	1 (1-way)
One-way Network Latency (μs)	6.2	5.2	1.5	2.3

Table 2.1: Experimental Platforms (†All of our platforms support a peak of 4 double-precision floating point operations per cycle.)

Chapter 3

One-Sided Communication Models

As described in Chapter 2, modern high end systems are built as a hierarchy of multicore chips that are combined into shared memory nodes, and the nodes are further combined into large networks that form a distributed memory system. Thus, when implementing programming models on such systems and optimizing communication patterns, one must considered both shared and distributed memory in the underlying system. A the programming level, one can also consider multiple forms of communication, including reading and writing to shared variables or explicitly sending messages. In this chapter we describe some of these programming language issues and describe the main focus of this thesis: Partitioned Global Address Space languages. We then describe the one-sided communication model underlying these languages and a our own implementation of the GASNet one-sided model for the IBM BlueGene/P architecture. We end with some performance comparisons between our one-sided communication and more traditional two-sided in terms of their performance characteristics, revealing some of the performance advantages of the one-sided model.

3.1 Partitioned Global Address Space Languages

Two distinct parallel programming models are commonly found on a variety of systems today: (1) message passing and (2) shared memory. Message passing is a shared-nothing programming model. In order for different processor cores to communicate with each other they have to expliclty send messages amongst themselves. One of the major criticisms of a message passing model is that both sides have to agree on when messages are being sent and received and thus, typically, all communication has to be known ahead of time. The ubiquity of MPI (the Message Passing Interface) [125] demonstrates that this is indeed a usable and portable programming model that can tackle a wide range of systems. However, distributed data structures, such as distributed queues, are very difficult to encode since both sides have to agree upon when an enqueue of an event, for example, will occur. For highly asynchronous or irregular applications such a restriction can be very inconvenient.

At the other end of the spectrum is shared memory programming. In this approach the entire address space is shared and thus any thread that is part of that address space will have the abililty to read and write the data of any other thread. These programming models

are typically found within nodes of a larger system rather than across large systems. Since all the memory is shared it is possible to build distributed data structures however one has to take great care to ensure that there are no race conditions when two threads modify a common memory location. In addition, to keep the illusion of shared memory to the end programmer, hardware has to create complicated mechanisms, such as cache coherency systems, that are difficult to scale to large levels of parallelism. Thus, even though both approaches have gained popularity in the community, they each have limitations that make them far from an ideal programming model.

Recently a new class of languages have emerged with aims of bridging these two distinct styles of parallel programming. The primary aim of these new languages, called Partitioned Global Address Space (PGAS)[171] languages, is to provide a single programming model for shared memory and distributed memory platforms (and combinations of them) by exposing a globally shared address space to the user. These languages also explicitly expose memory affinity and non-uniform memory access to the end user by having each thread be logically associated with a part of the shared global address space. The shared address space allows the processors to directly read and write remote data without notifying the application running on the remote processor through language level one-sided operations (i.e. put and get versus send and receive). Similar to traditional shared memory programming, the user is responsible for handling any race conditions that might arise. Unlike efforts that combine shared-memory and message passing together, such as OpenMP and MPI, the PGAS langauges provide one programming model across the system rather than relying on two distinct programming models that must be melded together. Thus the PGAS langauges aim to deliver the same performance with a uniform programming model across all the threads. Many related projects have shown the performance and productivity advantages of such an approach[47, 49, 64].

Since the languages explicitly expose the non-uniform nature of memory access times to the memory of different processors, operations to local data (i.e. the portion of the address space that a particular processor has affinity to) will tend to be much faster than operations on remote data (i.e. any part of the shared data space that a processor does not have affinity to). Thus, unlike traditional shared memory programming, the languages necessitate global data re-localization operations in order to improve performance which will be served by the collective operations. One of the primary focuses of the dissertation will be to analyze the interaction between these collective operations and the one-sided communication model. Some of these interactions are unique to PGAS languages and do not arise in two-sided parallel programming models like MPI.

3.1.1 UPC

There are many different variations of PGAS langauges that each have 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 that address space. To better understand what these languages offer we explore one of the more popular PGAS langauges, Unified Parallel C (UPC). UPC is the PGAS dialect of ISO C99 [102] and thus UPC is a

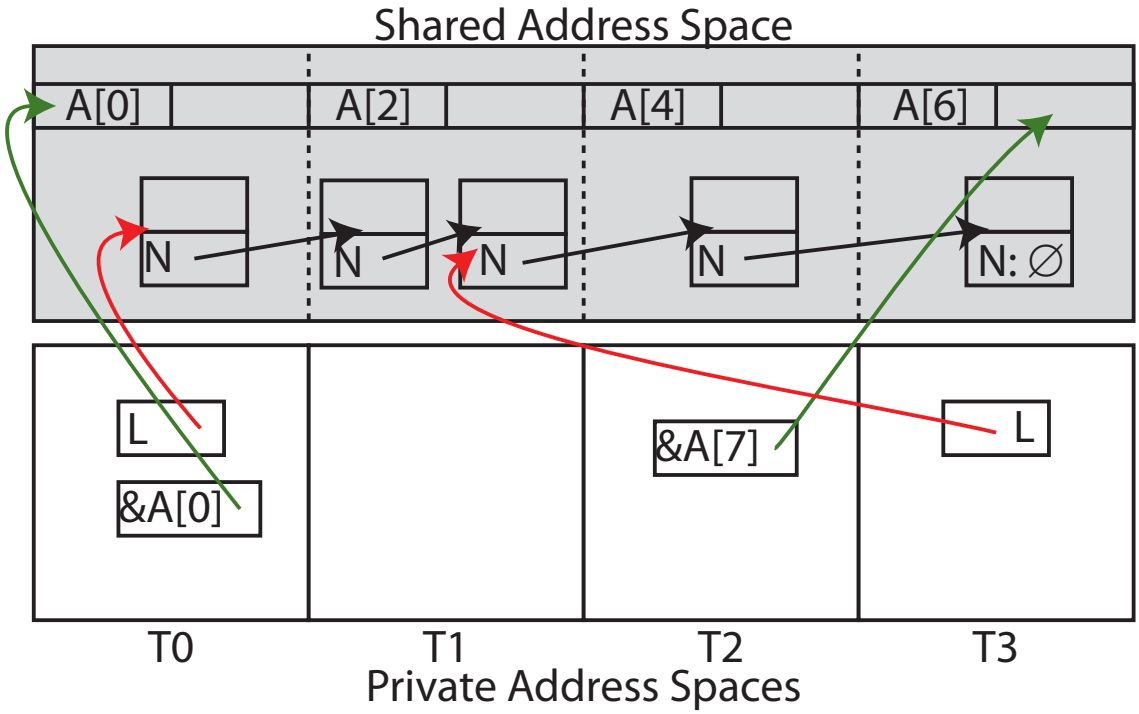


Figure 3.1: UPC Pointer Example

complete superset of C; any valid C program is a valid UPC program. To allow data to be read or accessed remotely UPC introduces the `shared` type qualifier into the langauge. Only variables declared with the `shared` qualifier will be part of the shared address space and all other variables are part of the private address space. By explicitly specifying the sharing UPC avoids the problem of inadvertant race conditions on data that the threads are not expecting to be altered by other threads.

UPC also provides C-style pointers to the shared address space that allow programmers to build distributed data structures. The UPC pointer also supports pointer arithmetic and dereferencing of variables that have affinity to different threads. In Figure 3.1 we show an example of a few of the different types of pointers in UPC. We can declare a linked list node with the following type:

```
typedef struct link_list_elem_{
 int value;
 shared struct link_list_elem_ *N;
} link_list_elem_t;
```

The linked list elements can then be instantiated and connected together to form the example shown in the figure. Notice that all the nodes do not need to have the same number of elements. We can declare pointers in the private address space to the shared linked list that can point to arbitrary locations in the list as follows:

```
shared link_list_elem_t *L;
```

In addition to using pure shared pointers UPC also allows the user to declare distributed arrays. To declare a shared array spread across 4 processors such that each processor has affinity to a set of two contiguous elements we use the following declaration:

```
shared int [2] A[2*THREADS];
```

We can then access this array through pointers or explicit array accesses. For processor 0 to modify the value of array element 5 it simply does `A[5]=42;`. Thus through simple declarations and type qualifiers we can build complicated data structures. A full summary of the language and all of the available features is beyond the scope of this work and we encourage the reader to refer to the langauge specification [159] or a productivity study by Cantonnet et al. [47] for further details.

3.1.2 One-sided Communication

PGAS languages and one-sided programming models rely on operations such as `put` (which allows a processor to write into a remote memory location) and `get` (which allows a processor to read from a remote memory location). In a one-sided operation the initiator has to provide both the source and destination of the data while the remote processor is not involved in the communication. In contrast, two-sided programming models (such as those found in MPI message passing) rely on operations `send` and `receive`. In order for a processor to write into a remote address space the source processor has to initiate a `send` operation and the destination initiates a corresponding `receive`. While the performance and semantic advantages of one-sided communication have been well studied[27, 37], our primary focus here will be on how the semantics one-sided programming models interact with the collectives. These interactions pose novel interface issues as well as novel tuning problems. Chapters 5 and 6 further explore how the performance of advantages of one-sided communication models can be translated into the performance of the collectives.

Throughout the rest of this dissertation we will make a distinction between a one- and two-sided programming model (i.e. the communication operations available at the language level) and a one- and two-sided communication model (i.e. the communication primitives provided by the network hardware). While the one-sided programming model maps naturally onto shared memory architectures as well as networks that support Remote Direct Memory Access (RDMA)[26], the Berkeley UPC[30] group has also done much related work analyzing how the one-sided programming model found in PGAS languages can map to a network that only provides two-sided communication primitives, such as Ethernet. In addition, there has been much related work in the two-sided programming model community[116, 86, 3] on how to take advantage of the one-sided communication model found in many networks, such as Infiniband[100], to suit a two-sided programming model such as MPI.

These issues raise two separate questions that we will address:

1. How can a one-sided communication model be used in the implementation of the collectives?

2. Does the use of a one-sided programming model allow novel opportunities for optimizing the collectives?

3.2 GASNet

GASNet [35] is the portable runtime layer that is used for the Berkeley UPC [30] compiler as well as other PGAS languages such as Titanium [96], Co-Array Fortran [132], and Cray's Chapel [6]. It is designed to provide a generalized communication and messaging framework for these languages. Unlike MPI, GASNet is designed as a compiler target, rather than an end-user library. It provides Active Messages, One-Sided point-to-point messaging, a set of collective operations, and many other features. The collective operations, which are the focus of this dissertation, are implemented within GASNet. Chapter 4 provides a summary of the collective interface available in GASNet.

The GASNet implementation is designed in layers for portability: a small set of core functions constitute the basis for portability, and we provide a reference implementation of the full API 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 reference implementation.

3.2.1 GASNet on top of the BlueGene/P

GASNet has been implemented on many processors (e.g. x86, PowerPC, MIPS, Opteron, SPARC, etc) as well as many interconnection APIs (e.g. Mellanox Infiniband, Cray Portals, IBM BlueGene DCMF, SHMEM, etc). A full list of supported platforms is available [91]. As a case study we examine one important platform, the IBM BlueGene/P. We show how the one-sided semantics found in PGAS programming models and GASNet map well to the GASNet software architecture. The exact implementations on other platforms vary depending on the network hardware and software libraries.

The BlueGene/P

The processing element found on the BlueGene/P (BG/P) is a Quad-Core 850MHz PowerPC 450 processor. This Quad-Core chip is combined with 2GB of memory to form a compute node. Thirty-two of these compute nodes form a Node Card. Thirty-two node cards form one rack of the machine. Thus each rack of the machine holds 4,096 PowerPC 450 cores. We have scaled our experiments to eight racks (32,768 cores) of a BG/P at Argonne National Lab named "Intrepid" [103].

Most inter-node communication on BG/P is done via a three-dimensional torus network, and the machine offers separate network hardware for barriers, collectives, and I/O. Each torus link provides a peak hardware bandwidth of 425MB/s in each direction, thus the six links into and out of a node provide an aggregate peak bi-directional bandwidth of 5100MB/s at each node. However due to packet overheads, as shown by Kumar et al. [112], the peak

messaging bandwidths available to applications are 374MB/s per link in each direction, and 4488MB/s aggregate bi-directional per node.

IBM has designed the Deep Computing Messaging Framework (*DCMF*) [112] as a semi-portable open-source communication layer that provides the low level communication APIs for higher level programming models such as UPC and MPI. The MPI implementation on the BG/P, MPICH2 1.0.7 [126], uses DCMF for all its communication interactions with hardware. To achieve the best PGAS language communication performance, the GASNet communication layer implementation on BG/P also targets this communication API. Although the only HPC machine providing a DCMF implementation is the BG/P, the API was designed to accommodate possible future systems. DCMF provides point-to-point communication operations, plus collective operations that have been specifically designed for the BG/P to take advantage of the hardware collective networks when possible.

DCMF

DCMF offers three mechanisms for point-to-point communication. While the API offers many more features we focus on these three since they are the salient to our discussion. For more complete details, see [112].

- `DCMF_Put()`: Unlike the send, the caller of a put provides both the source *and* destination addresses. In addition, the client provides two callbacks to run on the initiator: one that is invoked when the data movement is locally complete and the other when the data has been delivered to the remote memory. Notice that since the source node provides all the information required for delivery, this is a one-sided operation; the operation can be retired with no interaction from the remote processor.

- `DCMF_Get()`: The get is an analog of the put operation. Like put, the initiator provides both addresses (local and remote) eliminating the need for any involvement from the remote processor. Unlike the put however, the user only provides one callback that is invoked when the data transfer is complete and the data is locally available.

- `DCMF_Send()`: The send operation is the active message mechanism in DCMF. It accepts the function to invoke on the remote node, its arguments, and a message payload. The client provides a callback to be invoked on the initiator when the data is locally reusable. With a send, the remote processor needs to run the specified handler when the active message is received. Therefore the target processor has some non-trivial involvement in message reception.

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

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

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

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

Microbenchmarks

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

Latency Advantages of GASNet

In our first microbenchmark we compare the roundtrip GASNet and MPI "ping-ack" latency performance. For MPI this test measures the time needed for the initiator to send a message of the given size and the remote side to respond with a 0-byte acknowledgment. This benchmark is written using `MPI_Send()` and `MPI_Recv()` for both operations. The GASNet test measures the time to issue a put or a get of the given size and block for remote completion (when DCMF runs the remote completion callback). This comparison is made because while GASNet takes advantage of the remote completion notification of DCMF,

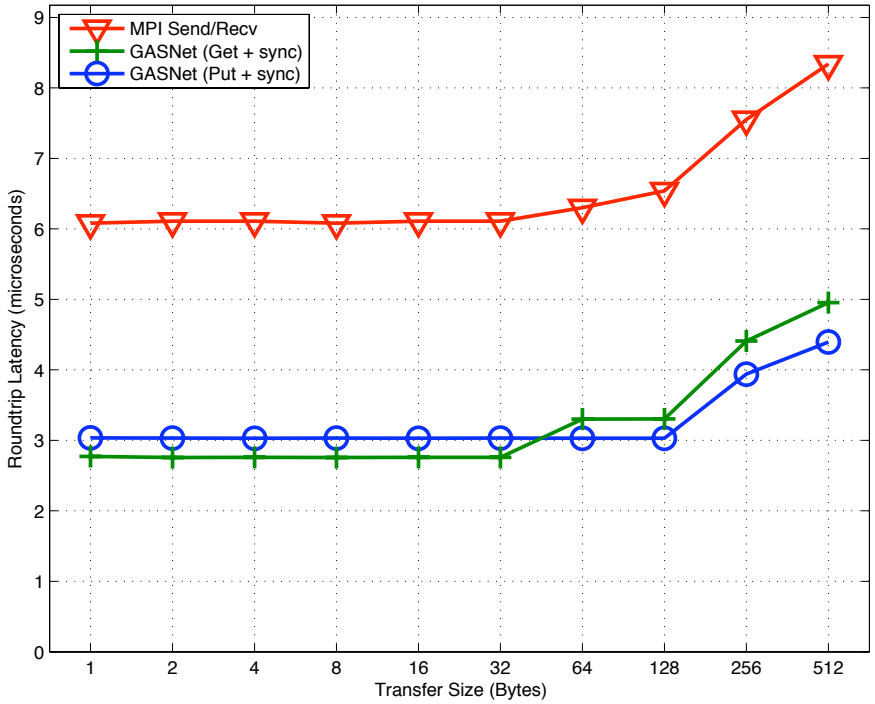


Figure 3.2: Roundtrip Latency Comparison on the IBM BlueGene/P

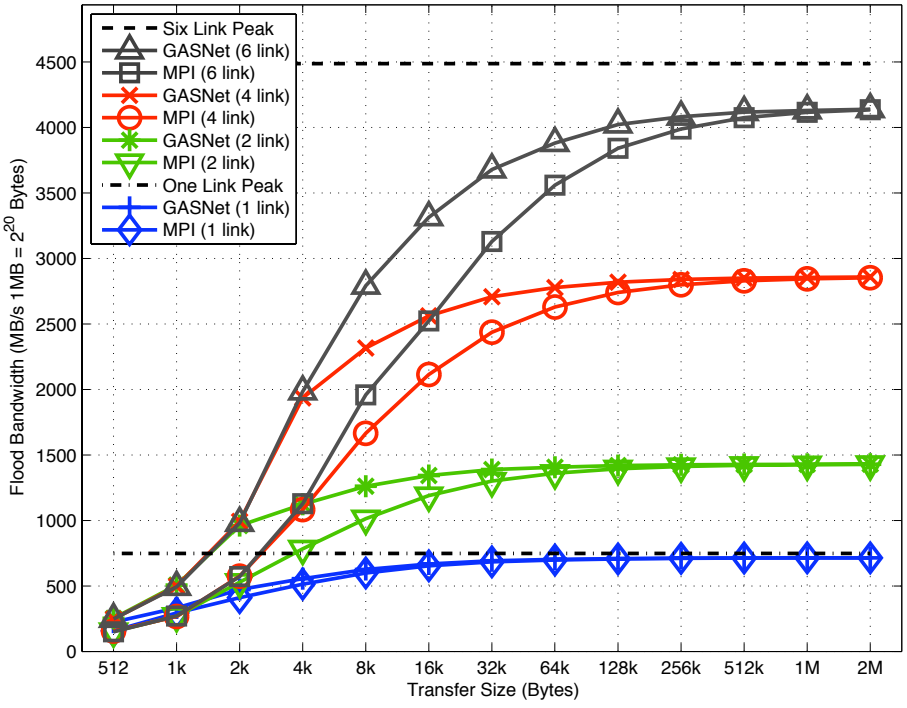


Figure 3.3: Flood Bandwidth Comparison on the IBM BlueGene/P

MPI needs an explicit acknowledgement to implement a roundtrip network traversal (such as those required in applications with fine-grained irregular accesses). Figure 3.2 shows the performance comparison.

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

Multi-link Flood Bandwidth Performance

In our next microbenchmark we analyze the flood bandwidth performance of GASNet and MPI. Each BG/P compute node has six links to neighboring nodes, two links in each of the three dimensions of the torus. When measuring the effective bandwidth of the node, measuring the performance across only one link under-utilizes the bandwidth available at each node. Therefore our bandwidth test, like those presented by Kumar et al. [112], measures the bidirectional bandwidth performance on a varying number of links. Figure 3.3 shows the flood bandwidth performance. The maximum achievable payload bandwidth for one and six links are shown for comparison. For the GASNet tests one core per node initiates a long series of non-blocking puts of the specified size to the neighbors on each of the chosen links (round-robin). For MPI we implement the same pattern of communication using `MPI_Isend()`s and a preposted window of `MPI_Irecv()`s. We have varied the size of this window over powers of two and for each MPI data point report only the highest bandwidth achieved.

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

3.2.2 Active Messages

GASNet also provides a rich *Active Message* library [91]. An active message provides a way to send a payload to a remote thread and invoke a function on the remote node once the data has arrived. For example, we could write a signaling put with an active message in which the payload contains the data which we intend to write and the function invoked on the remote side flips an appropriate bit to signal that the data has arrived and is ready for consumption. These operations map naturally to the IBM BlueGene/P's `DCMF_Send()` operation. Our active messages can be broken into three broad categories:

- **Short:** The active message contains arguments and no payload and is simply just a function invocation with the specified arguments on the target node. A psuedocode prototype for this function is:

  ```
  sendShort(dstnode, function_ptr, num_args, arglist)
  ```

- **Medium:** The active message contains both arguments and a payload that is small enough to be buffered on the target node in an anonymous buffer. The function on the remote node is passed a pointer to the buffer.

  ```
  sendMedium(dstnode, void *src, size_t nbytes,
  function_ptr, num_args, arglist)
  ```

- **Long:** The active message contains arguments, payload, *and* a destination address. The payload is transferred to the remote node at the destination address and the function is invoked once the data has arrived.

  ```
  sendLong(dstnode, void *dst, void *src, size_t nbytes,
  function_ptr, num_args, arglist)
  ```

Chapters 5 will go into detail about how the collectives utilize this functionality to implement the collectives.

Chapter 4

Collective Communication

Alongside the common point-to-point data movement operations in parallel languages and libraries, collective communication operations are important building blocks to writing complex scientific applications. These operations encapsulate common communication patterns that involve more than one processor and provide both productivity and performance advantages to the application writer. In this chapter we outline the basic collective operations and highlight the differences between collectives written for one- versus two-sided programming models.

4.1 The Operations

Even though their exact syntax is language-specific, most parallel languages and libraries provide the following set of collective operations. In all our descriptions below, we will use T to represent the total number of threads involved in a collective and assume that the threads are ranked from 0 to $T-1$. Through out the rest of this chapter we will use the term "thread" to mean a single sequence of instructions and the associated data structures. We assume that all T threads run concurrently and the underlying runtime systems are aware of all the threads and take responsibility for managing the communication amongst them. For example, in MPI this corresponds to an MPI rank and in UPC it corresponds to a UPC thread.

The operations can be broken into two categories. In the first category, listed below, the operations send or receive data from a single root thread. They are typically optimized through a rooted tree and require $O(T)$ messages. A full discussion of the algorithms and implementation can be found in Chapter 5.

- **Broadcast:** A root thread sends a copy of an array to all the other threads.
- **Scatter:** The root thread breaks up an input array into T personalized pieces and sends the i^{th} piece of the input array to thread i.
- **Gather:** This operation is the inverse of scatter. Each thread t sends its contributions to the t^{th} slot of an output array located on the root thread.

- **Reduce:** Every thread sends a contribution to a global combining operation. For example, if the desired result is the sum of a vector $\vec{x}$ of k elements where each thread t has a different value of $\vec{x_t}$, the result $y[j]$ on the root thread is $\sum_{t=0}^{T-1} x_t[j]$.

  The operations sum, minimum, and maximum are usually built-in and the user is allowed to supply more complicated functions. [1]

The operations in the second category require that every thread receive a contribution from every other thread. Naïve implementations of these operations can lead to $O(T^2)$ messages while well tuned can perform the same task in $O(T\ log\ T)$ messages. The implementation of these algorithms is detailed in Chapter 6.

- **Barrier:** A thread cannot exit a call to a barrier until all the other threads have called the barrier.

- **Gather-To-All:** This operation is the same as gather except that all threads get a copy of the resultant array. Semantically it is equivalent to a gather followed by a broadcast.

- **Exchange:** For all $i, j < T$, thread i copies the j^{th} piece of an input array located on thread i to the i^{th} slot of an output array located on thread j.

4.1.1 Why Are They Important?

These operations are designed to abstract common communication patterns so that the user need not worry about how they are implemented. The first and foremost reason to examine collectives is that their performance is a bottleneck in many applications.

A second and equally important reason to encapsulate and abstract these operations is to provide performance portability to the end user. This abstraction enables the following contract with the user: If an application is written with these collectives, the runtime system will guarantee correctness and near hand-tuned performance regardless of the environment it is run in. As a result of this abstraction, the operations and interfaces need to be general enough to suit the needs of a variety of applications while being limited enough to keep the implementation and optimization requirements for the runtime system tractable.

[1] For the computational collectives that involve floating point special care must be done to ensure that the operations are reproducible in the presence of floating-point round-off error. The MPI collectives assume that all reduction operations are associative but allow the user to specify operations that are not commutative. To ensure reproducibility of an operator that is not associative, Reduce can be implemented using a Gather and having the root perform all the computation. While this approach is slower it will yield consistent and reproducible results. As part of the interface special flags can be specified to the collectives to yield the desired behavior.

4.2 Implications of One-Sided communication for Collectives

In a traditional two-sided programming model, a collective is considered complete when that thread has received or contributed its data. Notice that this does not imply that *all* threads have received their data. However, since the only way to send to a remote thread's memory is by having the remote thread perform a corresponding receive operation, there is no way to express the case in which a thread sees a locally complete but globally incomplete collective. In the case of a one-sided programming model, once a thread has received its local contribution from a collective it can start reading/writing data from/to other threads without the remote threads' involvement. Without any additional synchronization, there is no way for the thread performing the communication to know whether the collective has been completed on the remote thread leading to a potential race condition. While this might seem like a drawback to PGAS languages, it actually raises interesting opportunities for optimizations, and along with them, interesting questions about when the data movement for a collective can start and when a collective is considered complete.

Currently, the strictest synchronization mode requires a barrier before and after the collective to ensure that the completion of the collective is globally visible. Thus after the collective and the global barrier, any one-sided operation is considered safe since the collective has been guaranteed to finish. In addition, in the case of these strict synchronization requirements, our collectives will be focused on minimizing latency, i.e. building a communication schedule that minimizes the time it takes for all the nodes to get the data. If we are interested in many collectives with a looser synchronization mode our tuning efforts will focus more on building a communication schedule that maximizes throughput. The synchronization mode is taken into consideration when optimizing the collective. Since the language itself, allows us to intermix these various synchronization modes, our runtime system will need to take the synchronization modes into account when making tuning decisions.

Along with the collective optimization problem, these new issues raise novel questions about collective interfaces that do not occur in the two-sided programming model. Exposing the generality of these synchronization modes has been a hotly debated topic in the PGAS language development community and there is still no clear winner. Part of this dissertation will also focusing on analyzing the various language level interfaces to the collectives in PGAS languages.

4.2.1 Global Address Space and Synchronization

The global address space that PGAS languages offer provide interesting benefits for the implementations of collectives as well as a novel set of performance tuning opportunities and challenges. In this section we will go deeper into how these two features of the languages can aid in the performance of the collectives implementations.

Since the sources of collective communication in two-sided models do not know the final destination of the data, an asynchronous implementation has to deal with the possibility of unexpected messages (due to computational load-imbalance). Two-sided implementations[116,

3, 86] deal with this point-to-point communication problem in one of two ways: either an "eager" or "rendezvous" mechanism. In the eager mechanism every thread has a mailbox in the memory of every other thread it sends data to. Once that target thread is ready to process the data, it then waits for the data to be copied out of the eager buffer space to its final destination. This mechanism has obvious memory scalability problems as machines get larger. Since a second copy is prohibitively expensive for large messages, a second mechanism, called rendezvous, is used. In rendezvous the source thread waits for the target to send it the final destination address so it can perform the write directly into the memory, assuming that the network has the capability to directly write into the remote memory space. This method unfortunately over-synchronizes the transfer.

In PGAS languages, all the threads have global knowledge of the final destinations of all the data. This knowledge can be used to circumvent the problem since the data can be sent directly where it needs to go. However, in order to properly take advantage of this global knowledge we need to ensure that the target thread is not using the data that is about to be over-written by the collective. We could insert a barrier before and after every collective but this again over-synchronizes the problem and will not fully expose the performance advantages of collectives in one-sided programming models. Currently the development committee of Unified Parallel C (UPC) language, a popular PGAS language, is exposing this problem to the end user and having the user decide what the collective synchronization semantics need to be. In the strictest mode, data movement can only occur after the *last* thread has entered the collective and before the *first* thread leaves the collective. In the loosest mode, data movement can start as soon as the *first* thread enters the collective and can continue until the *last* thread leaves the collective.

Related work by Faraj et al. [78] has shown that at large scale it is very difficult to ensure that tasks arrive at a collective at the same time due to the inherent computational load imbalance exhibited by the applications. Thus relying on a collective model that requires this strict synchronization can over synchronize the collectives and lead to performance scalability problems. Thus, by loosening the synchronization semantics of the collective, we can alleviate some of these problems and thereby realize better application performance and scalability.

4.2.2 Current Set of Synchronization Flags

As with the MPI [125] collectives, the current mechanisms are designed such that all the threads participate in the collectives by calling the collective routines together. However, since UPC's point-to-point communication model uses one-sided communication, special care has been taken to carefully describe when source and destination buffers are safe to read and when they are not. The current collectives specify two sets of synchronization flags: one set describes the semantics of when the data movement can start (the UPC_IN_xSYNC flags) and a second set describes when it is safe to modify the buffers that were involved in the collectives (the UPC_OUT_xSYNC flags). These flags affect when the data can be read and modified for both the input *and* output data for the collectives. For the sake of completeness, let us quickly review what these synchronization modes are:

- UPC_IN_NOSYNC: Data movement can start as soon as the *first* thread enters the collective.
- UPC_IN_MYSYNC: Data movement into and out of buffers that have affinity to a certain thread can start only after that thread has entered the collective.
- UPC_IN_ALLSYNC: Data movement can start only after *all* the threads have entered the collective.
- UPC_OUT_NOSYNC: Data movement can continue until the *last* thread leaves the collective
- UPC_OUT_MYSYNC: Data movement into and out of buffers that have affinity to a certain thread can be written or read until that thread leaves the collective.
- UPC_OUT_ALLSYNC: All data movement has to complete before the *first* thread leaves the collective.

These different input and output synchronization flags can be combined in any combinations, leading to a total of 9 synchronization modes. The default synchronization mode is UPC_(IN,OUT)_ALLSYNC, which is the strictest.

4.2.3 Synchronization Flags: Arguments for and Against

These synchronization flags, while powerful, have many flaws. In this section we will summarize a few arguments for and against the use of 9 different synchronization modes.

Arguments For

- They represent a rich set of synchronization modes that encompass a broad range of applications.
- They provide very valuable hints to the runtime system about the application semantics of the collective usage.
- Since the user explicitly specifies the synchronization mode, it saves the compiler from having to perform a dependency analysis of the source and destination buffers to infer the synchronization requirements on these buffers.
- It allows the implementation to optimize the synchronization *and* the collective together.
- They force the user to think carefully about synchronization on the collectives, especially when they are interested in performance.

Arguments Against

- They place a large burden on the programmer to think through the synchronization modes, especially if they are interested in optimal performance. If the programmer (lets say an MPI programmer who wishes to switch to UPC) wishes not to burden himself with analyzing the synchronization modes, and so use the default (strictest mode), then he will pay the costs of over-synchronization on every collective.
- The 9 different synchronization modes for each collective force high-quality implementations to provide different implementations for each of the possibilities.
- Their generality has not been fully utilized by the programmer.

4.2.4 Optimizing the Synchronization and Collective Together

For the strictest synchronization modes, we can optimize the collectives along with the global barrier to save a round of communication. For example, if the user requests an IN_ALLSYNC/OUT_NOSYNC Broadcast, then we can optimize the synchronization with the collective. In a Broadcast the data travels down from the root. Since the user requested that the data movement only start after *all* threads have entered the collective, the root thread has to wait till all threads have signaled their arrival before beginning the broadcast.

One could implement this with a full barrier followed by a broadcast. This would require three rounds of communication: (1) all threads report arrival to the barrier, (2) all threads receive information that all others have arrived, and (3) the data is broadcast down from the root. Since the UPC interface allows the synchronization mode to be expressed as part of the collective, the synchronization can be folded into the collective. Thus the entire operation can be completed with two passes: one pass up with all threads communicating up towards the root indicating they have arrived and one pass for the broadcast. Since the root is the only thread initiating the communication, only the root needs to know all threads have arrived (i.e. half a barrier) before initiating the data movement. Thus we can save an entire round of communication. For collectives over many threads with small message sizes, the latency advantages can be quite significant. Similarly for collectives in which the data travels up to a root (i.e. Gather and Reduce), all the data arriving at the root also signals that the threads have arrived at the collective. Thus the root can send a signal down the tree indicating that the collective and barrier are finished and that the data has arrived.

Performance of Loose Synchronization

Figure 4.1 shows the performance of GASNet collectives compared to their MPI counterparts in the Cray MPI library (version 3.4.1) on 1k cores of the CrayXT4. We use a benchmark that calls a broadcast operation repeatedly in a loop and report the time taken for an average broadcast. For "MPI Throughput" we call an MPI Broadcast repeatedly while "MPI Latency" adds a barrier between each broadcast. As the data show, when the collectives are loosely synchronized, GASNet yields the best performance, especially at mes-

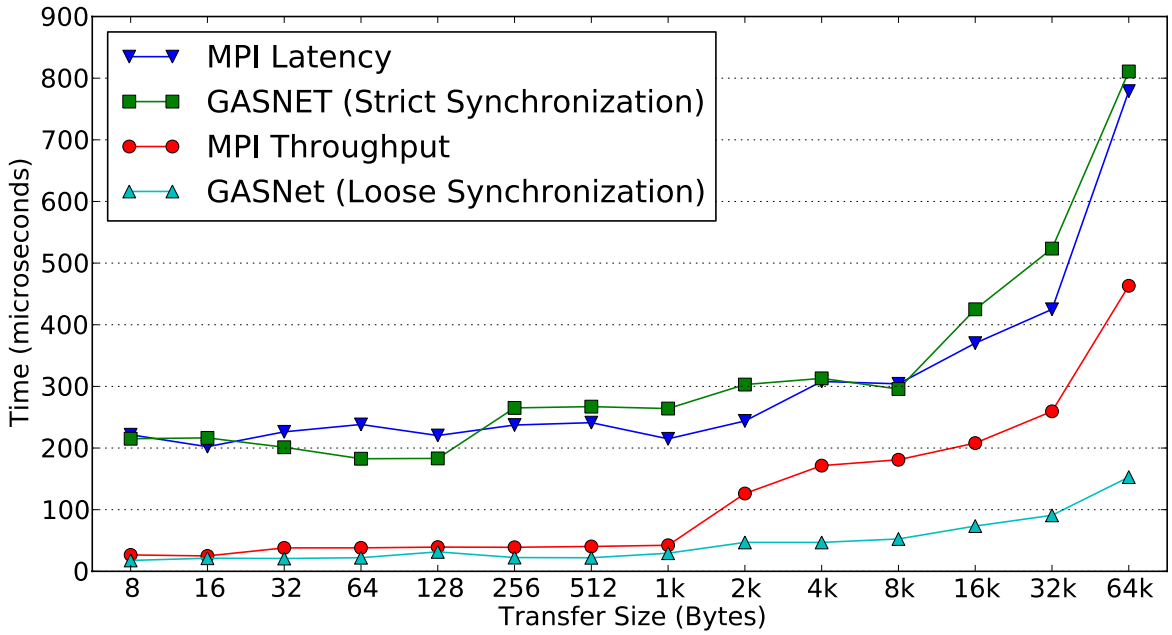


Figure 4.1: Comparison of Loose and Strict Synchronization (1024 cores of the Cray XT4)

sage sizes larger than 1kBytes. The next section goes into much greater detail about how a one-sided communication model is used in the collective communication.

4.3 Collectives Used in Applications

There are a vast number of production parallel applications and each one of them has unique computation and communication requirements. In order to simplify our study of these applications we will use Phil Collela's set of seven "dwarfs"[57] that represent common communication and computation patterns that arise in these applications. In order to get a better sense of which collectives are used in the different dwarfs we first pick a representative set of applications, benchmarks, and tools for each dwarf and analyze the different collectives that are used in each. We augment this list by analyzing the collective communication requirement of the NAS Parallel Benchmarks[19]. Table 4.1 shows the applications that were used for each dwarf while Table 4.2 shows which collectives are used in each of these applications.

From this study we can see that the most widely used collectives are reduce and reduce-to-all. Previous work[163] has also shown that many different applications also rely on these two collectives. In addition, we also notice that two of the dwarfs, Dense Linear Algebra and Unstructured Grids, rely on most of the collectives in some way. However, the Monte Carlo methods do not rely on collectives since they are limited by local computation performance rather than communication performance. Communication is done to distribute the work at the beginning and aggregate results at the end and is outside the critical path. Kamil et

	Dwarf	Applications
1	Dense Linear Algebra	LinPack[71], MADBench[40], ScaLAPACK[147]
2	Sparse Linear Algebra	SuperLU[115], OSKI[164], NAS CG
3	Spectral Methods	FFTW[85], UHFFT[122], NAS FT
4	N-Body Methods	GTC[114]
5	Structured Grids	Chombo[1], NAS MG
6	Unstructured Grids	ParMETIS[4]
7	Monte Carlo	NAS EP

Table 4.1: Representative Applications for Each Dwarf

	Dwarf	Broadcast	Scatter	Gather	Reduce	Reduce To All	Gather To All	All To All
1	Dense Linear Algebra	✓	✓	✓	✓	✓	✓	
2	Sparse Linear Algebra	✓			✓	✓	✓	✓
3	Spectral Methods							✓
4	N-Body Methods			✓	✓	✓		
5	Structured Grids	✓		✓	✓	✓		
6	Unstructured Grids	✓	✓	✓	✓	✓	✓	✓
7	Monte Carlo							

Table 4.2: Collectives in the Dwarfs

al.[108] have also shown that the message sizes that are transferred within the collectives tend to be small (under 2kB) and thus the latency of the collectives is a bottleneck in many applications. This dissertation will analyze how these applications can benefit from the optimizations discussed.

Chapter 5

Rooted Collectives for Distributed Memory

The next two chapters will focus on the implementation strategies for distributed memory. In order to ensure that the collective operations realize the best performance and scale, they must be aware of the network topology and try to minimize the communication amongst nodes that are very far away to avoid crossing the network many times. In addition, as the scale continues to grow, algorithms that rely on every node talking to every other node inducing $O(n^2)$ network operations will realize poor scalability as n grows large. Thus it is often, but not always, useful to route the data through intermediary nodes. These chapters will go more detail into the associated tradeoffs. We divide our discussion of the implementations of the collectives into two classes. We focus on the rooted collectives (Broadcast, Scatter, Gather, Reduce) in this chapter and the non-rooted collectives (Exchange and Gather-to-All) in Chapter 6. As the results will demonstrate, GASNet (and hence the one-sided communication model) can achieve a 27%, 28% and 64% improvement in Broadcast performance on 1024 cores of the Sun Constellation, 2048 cores of the Cray XT4 and 1536 cores of the Cray XT5 respectively. Additionally GASNet is able to achieve a 27%, 44%, 65% on for a Scatter on 512 cores of the Cray XT4, a Gather on 1536 cores of the Cray XT5 and a Reduce on 2048 cores of the Cray XT4 respectively.

This chapter first outlines the different implementation possibilities in Section 5.1 and how the one-sided communication model discussed in Chapter 3 and the synchronization modes shown in Chapter 4 affect the overall implementation. We then go on to discuss how the infrastructure pieces apply to other rooted collectives in Section 5.2. There are many implementation choices and thus, in order to better understand the performance, we create simple performance models in Section 5.3. The main goal of the models will be to provide a guide to what the algorithm space looks like so we can effectively prune the search space when it comes time to automatically tune the operations. Finally we use Dense Matrix Multiplication and Dense Cholesky factorization as examples to show how the collectives can be incorporated into real in applications in Section 5.4.

5.1 Broadcast

When implementing the collectives for distributed memory, there are many components of the infrastructure that are common to all the collective implementations. To provide a context for these features we use Broadcast as a case study to examine different factors that affect the optimal implementation of Broadcast. While our discussion centers on Broadcast, all the collectives in GASNet are able to use the features described in this section.

5.1.1 Leveraging Shared Memory

Most modern systems, including the ones presented in this study, contain multicore processors that have shared memory between all processor cores within a node. 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 a shared memory domain is 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()`. Throughout the rest of this chapter we will use the term "thread" to mean an instruction stream that has a stack and a set of private variables. Our study is conducted with POSIX threads [45] however the techniques discussed in this dissertation are applicable to other threading models as well. In the Berkeley UPC runtime, there is a one-to-one mapping between UPC level thread maps 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. Henceforth we use "thread" to refer to UPC threads and OS level threads interchangeably. We call a collection of threads that share a common address space, system resources, and network endpoint resources a "virtual node." Each virtual node is implemented as an underlying OS process. Since communication amongst processes is handled through the network interface card, the current collective library makes no distinction between virtual nodes that are co-located on the same physical node and virtual nodes on two distinct physical nodes. Thus throughout the rest of this dissertation we will use the term "node" to mean a virtual node. Future work will address this issue and optimize the collectives to further be aware and optimize for the communication amongst virtual nodes that are co-located on a physical node. Specifically the collectives can take advantage of special regions of memory that are managed by the operating system and shared across multiple processes. Chapter 7 will go into more detail about how collectives that target purely shared memory systems can be optimized.

Our collective implementations are aware of this hierarchical model thus, to minimize the network traffic, a representative thread from each node manages the communication on the network with all the other nodes and uses `memcpy` to pack and unpack the data. For a Broadcast, this implies that once the data arrives at a shared memory node, the representative thread will copy the data into the address space of all the other threads directly.

The Sun Constellation machine has 4 quad-core Opteron sockets per node so 16 cores can potentially share the same address space. Figure 5.1 shows the performance of varying

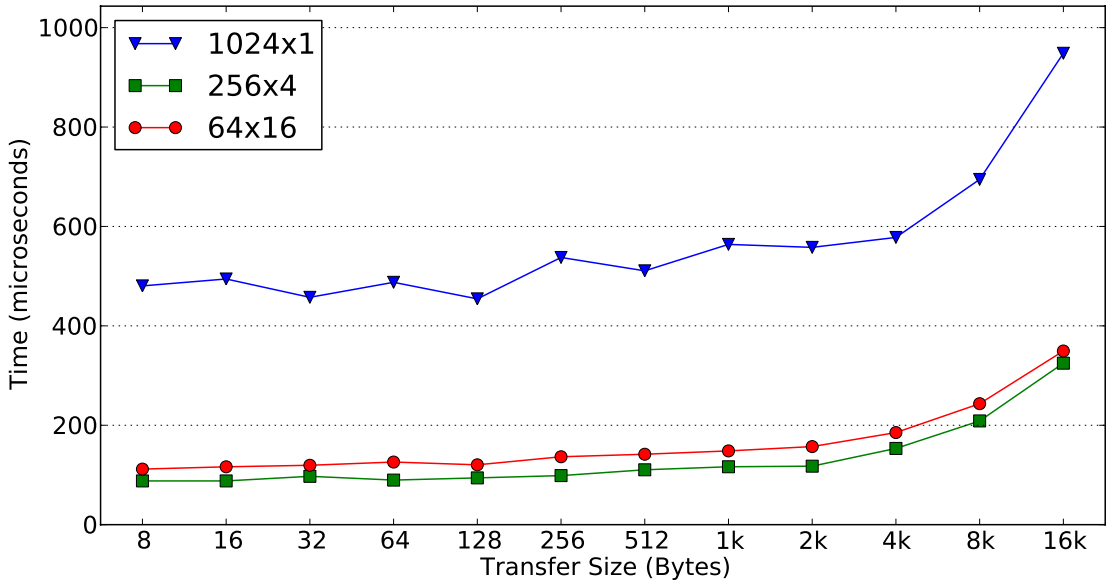


Figure 5.1: Leveraging Shared Memory for Broadcast (1024 cores of the Sun Constellation)

the number of threads per node as a function of message size for 1024 cores. Each of the 1024 cores are arranged into three different models: 1024×1 is 1024 nodes with 1 thread per node (1 virtual node per core), 256×4 is 256 processes with 4 threads per process (1 virtual node per quad-core socket) and 64×16 is 64 processes with 16 threads per process (1 virtual node per physical node). The best performance comes from having one GASNet node per quad-core socket. Since the cores within a socket all share a higher level of cache, their intra node communication costs are drastically reduced by not having to traverse expensive interconnection networks. In addition, GASNet relies on intra-node thread barriers, locks, condition variables, and other synchronization constructs to manage the various threads. Lower thread counts per node reduce the costs of intra-node synchronization. By minimizing the number of cores that are used, we can also minimize the overheads associated with cache coherency. Thus on the Sun Constellation having four virtual nodes (i.e. processes) per physical node yields the best performance.

Throughout the rest of the chapter we have four GASNet nodes with four threads each on Sun Constellation machine, since that yields the best performance. On the CrayXT systems our results have shown that the best performance is when there is exactly one virtual node per physical node. This translates to 4 threads per node on the CrayXT4 and 12 threads per node on the CrayXT5. Since we do not over-subscribe the physical cores with excess threads, each of the threads within a node will always be scheduled and thus we do not concern ourselves with interference from the scheduler in the operating system.

5.1.2 Trees

A broadcast is typically implemented through a *Tree*. We define a tree as a directed acyclic graph constructed amongst the nodes such 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 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 will appropriately route the messages from the source and destination transparently to the collectives library. Constructing the optimal trees for the various collectives on different platforms is a major part of the rest of this chapter.

In order to complete a Broadcast, the root will send the data to its children who will then forward the data on to their children. Thus the children need to know when the data has arrived and when it is safe to send down the tree. By using intermediary nodes to forward data up or down the tree, one can better utilize the available network bandwidth than in a naïve implementation in which the root communicates directly with every node. However this comes at an added latency cost due to the additionally network hops between the root and the leaf nodes. To connect N nodes in a tree there are an exponential number of possible trees thus it is obvious that searching or considering all of them is impractical for any significant value of N. To limit the search space we examine two unique classes of trees that we have found to be useful on a wide variety of platforms: (1) the *K-ary* tree and (2) the *K-nomial* tree.

K-ary Trees

A *K-ary* tree is one where each of the nodes of the tree has at most K children. Thus we define a standard binary tree (two children per node) to have a radix of $K = 2$. Figure 5.2 shows an example set K-ary trees. For a K-ary tree there are $\lceil log_K n + 1 \rceil$ levels where n is the total number of nodes in the tree. We show the algorithm to construct a K-ary tree from a list of nodes in Figure 5.3. Notice that we make an effort to reverse the children so the child with the largest index is always the first to get the data. This optimization is based on the heuristic that nodes with higher index will tend to be farther away than closer ones. The placement and numbering of the different ranks is done by the job scheduling system on most systems [81, 135]. These systems strive to ensure that nodes with similar identifiers are placed close together but it is not always guaranteed. Thus sending to the higher index first implies that we try to initiate communication to peers that are further away first to ensure a fairer balance of the communication. While this heuristic is not always the case, it is common enough from experience that this ordering is good in practice.

Figures 5.4- 5.6 show the performance of the different K-ary trees on three different experimental platforms. As the data show, the oct-ary tree is the best performer on all the platforms at small message sizes except the Cray XT4 where the quad-ary tree is the winner. Section 5.3 goes into much more detail about the selection of the best tree.

K-nomial Trees

One of the main drawbacks of K-ary trees is that once a parent sends the data to the children, it is idle and has to wait until the data propagates to the leaf node. If the user specifies that loose synchronization is allowable, then this is not a major concern since the

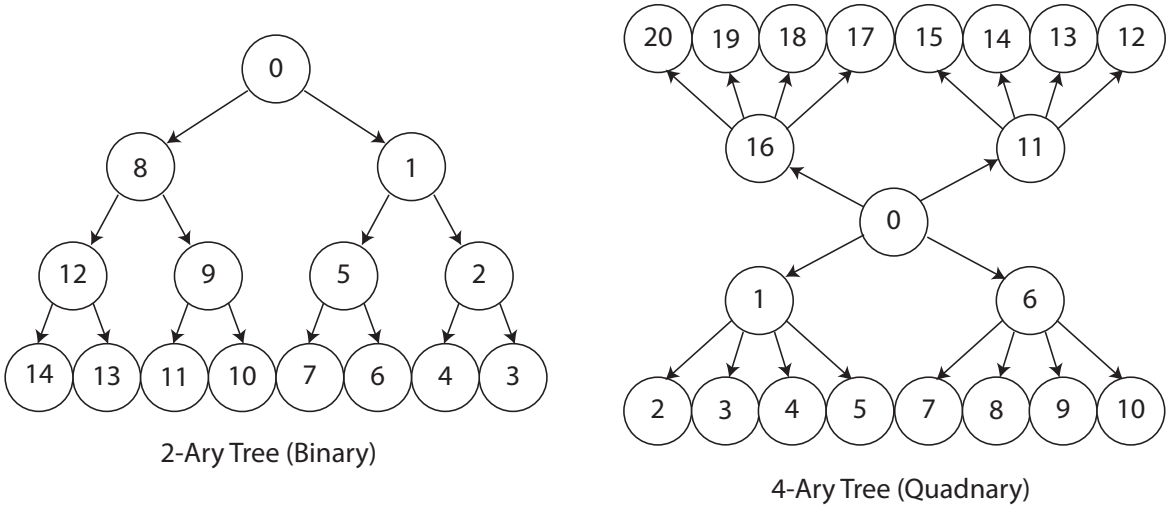


Figure 5.2: Example K-ary Trees

```
MakeKaryTree(nodelist, radix)
 1  if numnodes == 1
 2     then return nodelist[0]
 3  rootnode ← nodelist[0]
 4  rootnode.children = NIL
 5  for i ← 0 to radix − 1
 6  do if i == 0
 7        then start ← 1
 8        else  start ← MIN(len(nodelist), i × (⌈len(nodelist)/radix⌉))
 9
10     end ← MIN(len(nodelist), (i + 1) × (⌈len(nodelist)/radix⌉))
11     if start == end
12        then continue
13     rootnode.children.append(MakeKaryTree(nodelist[start, end), radix))
14
15  rootnode.children = rootnode.children.reverse()
16  return rootnode
```

Figure 5.3: Algorithm for K-ary Tree Construction

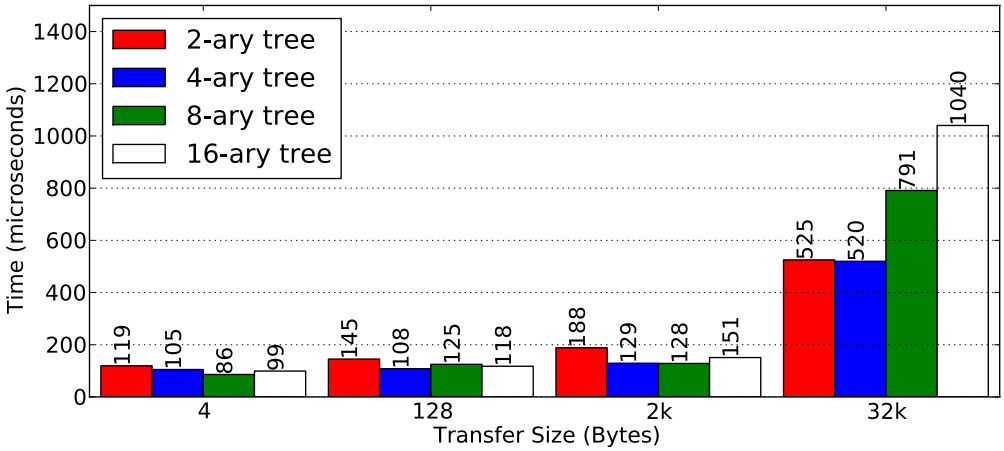


Figure 5.4: Comparison of K-ary Trees for Broadcast (1024 cores of the Sun Constellation)

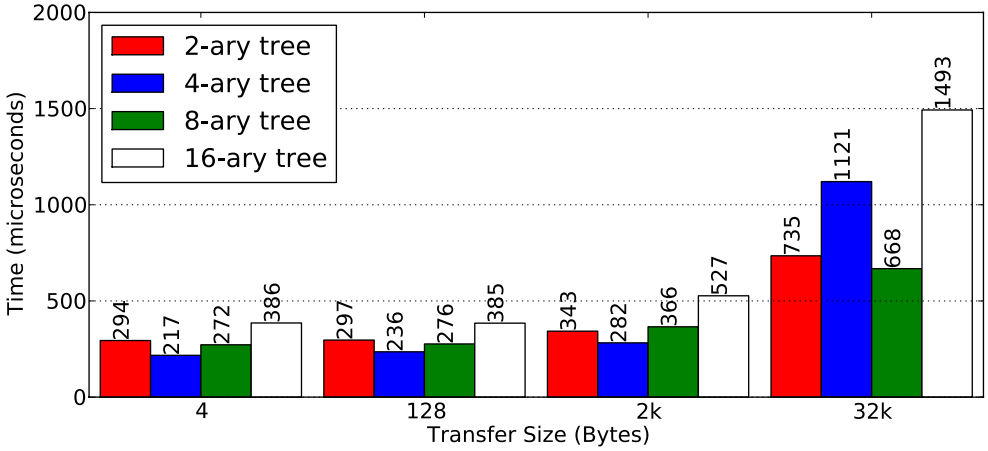


Figure 5.5: Comparison of K-ary Trees for Broadcast (2048 cores of the Cray XT4)

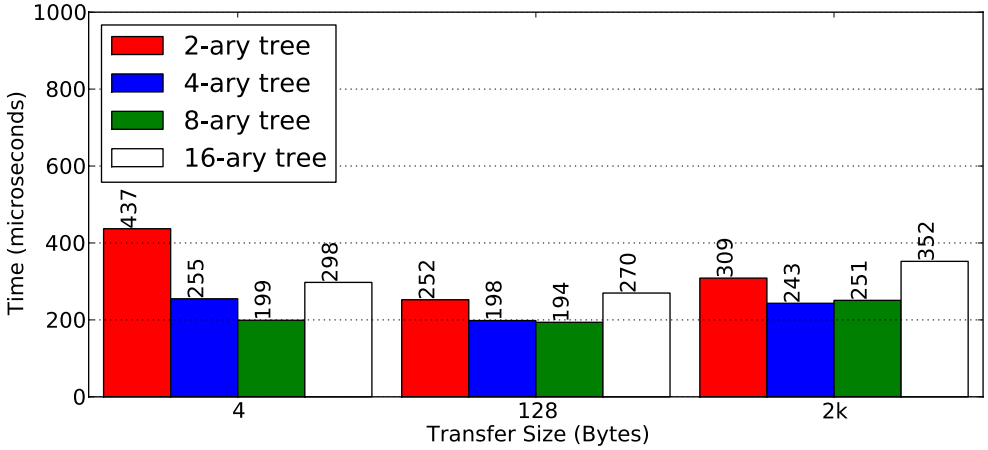


Figure 5.6: Comparison of K-ary Trees for Broadcast (3072 cores of the Cray XT5)

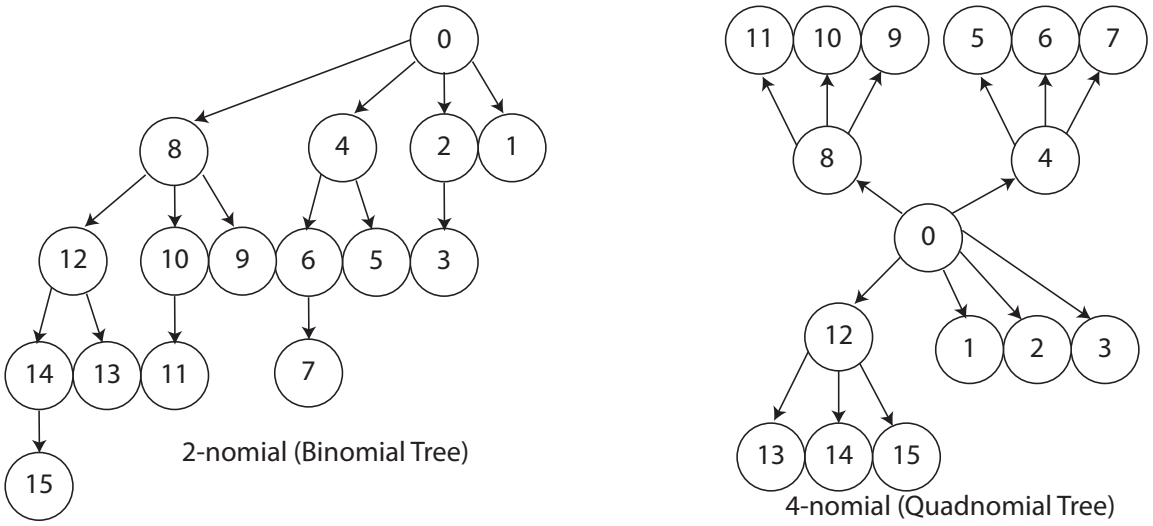


Figure 5.7: Example K-nomial Trees

```
MakeKnomialTree(nodelist, radix)
 1  if numnodes == 1
 2     then return nodelist[0]
 3
 4  done ← 0;
 5  stride ← 1
 6  rootnode ← nodelist[0]
 7  while done < len(nodelist)
 8  do for r ← stride to (stride × radix) − 1  by  stride
 9       do end ← r + MIN(stride, len(nodelist) − done))
10            child ← MakeKnomialTree(nodelist[r, end), radix)
11            rootnode.children.append(child)
12            done ← done + MIN(stride, len(nodelist, done))
13       stride ← stride × radix
14  rootnode.children ← rootnode.children.reverse()
15  return rootnode
```

Figure 5.8: Algorithm for K-nomial Tree Construction

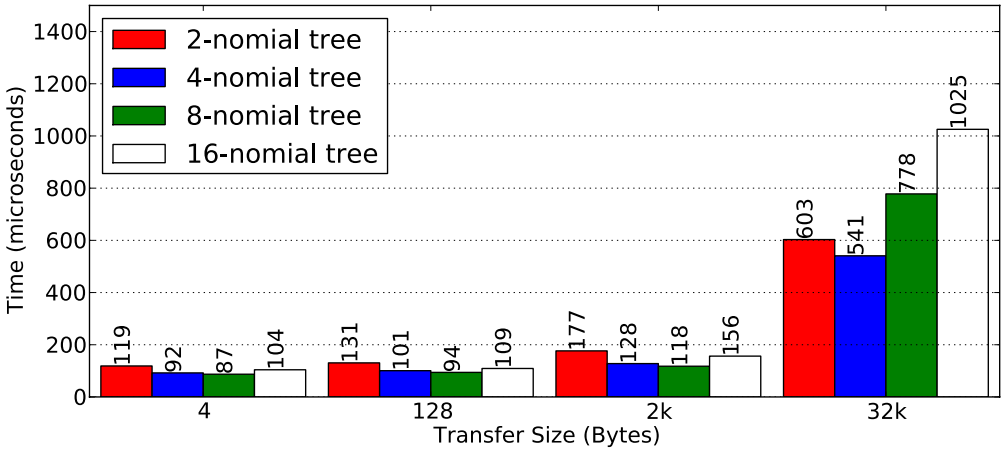


Figure 5.9: Comparison of K-nomial Trees for Broadcast (1024 cores of the Sun Constellation)

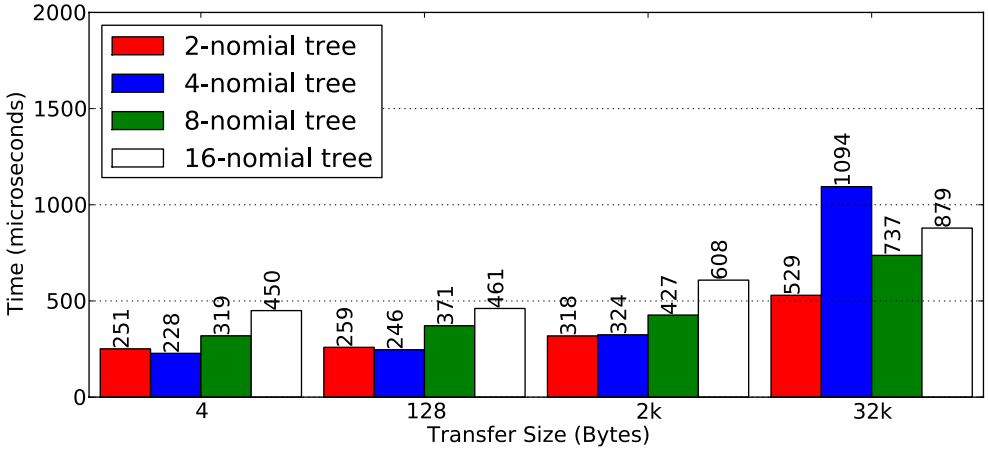


Figure 5.10: Comparison of K-nomial Trees for Broadcast (2048 cores of the Cray XT4)

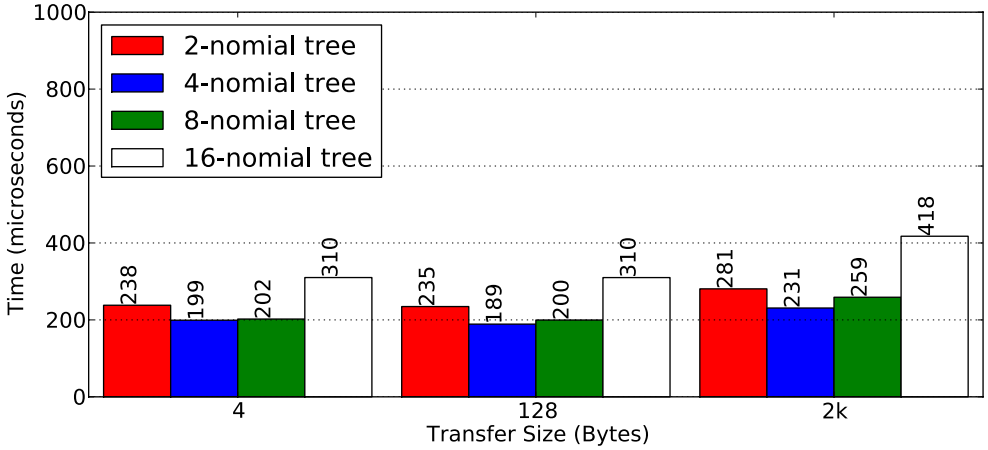


Figure 5.11: Comparison of K-nomial Trees for Broadcast (3072 cores of the Cray XT5)

parents will simply send data to the limited number of children and exit the collective. However, for strictly synchronized collectives, waiting for the data to propagate to the bottom wastes the available processing power at the root of the tree. The root can initiate data to other parts of the tree while the data is propagating down the other parts of the tree. To take advantage of this opportunity, we examine a *K-nomial* tree. Figures 5.7 and 5.8 show example trees and the algorithms used for tree construction. Notice that unlike the K-ary trees, this tree is unbalanced such that the higher ranked children have heavier subtrees. Once a child has received its data it can start sending to its children immediately. One would expect that the root is still actively sending to its children while data is still propagating down to the leaves. In an ideal case where the latency is the same as the overhead to inject a message into the network, the data will reach all the leaves at the same time.

Figures 5.9- 5.11 show the performance of the different k-nomial trees on three different experimental platforms. As the data show, the quad-nomial tree is the best performer on all the platforms at small message sizes. However as the message size grows, the binomial tree becomes the clear winner on the Cray XT4.

Observations

The data show that the platforms prefer the K-ary trees at large message sizes. In fact, at the largest sizes the binary tree is the best performing, suggesting that reducing the in-degree (and thus the computation performed between receiving and sending) is a dominant factor in tuning at these sizes. At lower message sizes there is little difference between the K-ary and K-nomial on the Sun Constellation, but on the CrayXT the K-nomial trees perform better (by as much as 22%). This result suggests that for this platform the increased tree breadth (and corresponding increase in communication parallelism) is the dominant consideration when the cost of the computation is small. These results illustrate two important points. The first is that no single tree shape is universally optimal, while the second is that the optimal choice can be platform dependent.

Fork Trees

There are many other possible tree classes beyond just the standard K-ary and K-nomial trees. There are an exponential number of possible tree shapes to connect a set of nodes. However searching all these possiblitites is intractable and therefore the search space must be carefully defined. Lawrence and Yuan[113] propose a method to automatically discover the optimum topology for a switched ethernet cluster. The K-ary and K-nomial trees are well designed for networks when topology is not taken into account. However, when network topology information is available we can construct trees that are a closer match to this topology. An example of this is the "Fork Tree" which targets a mesh or torus network. This tree ensures that the data is only sent between nodes that are connected together by a physical network link. Figure 5.12 shows some example fork trees and Figure 5.13 shows the algorithm that is employed for fork tree construction.

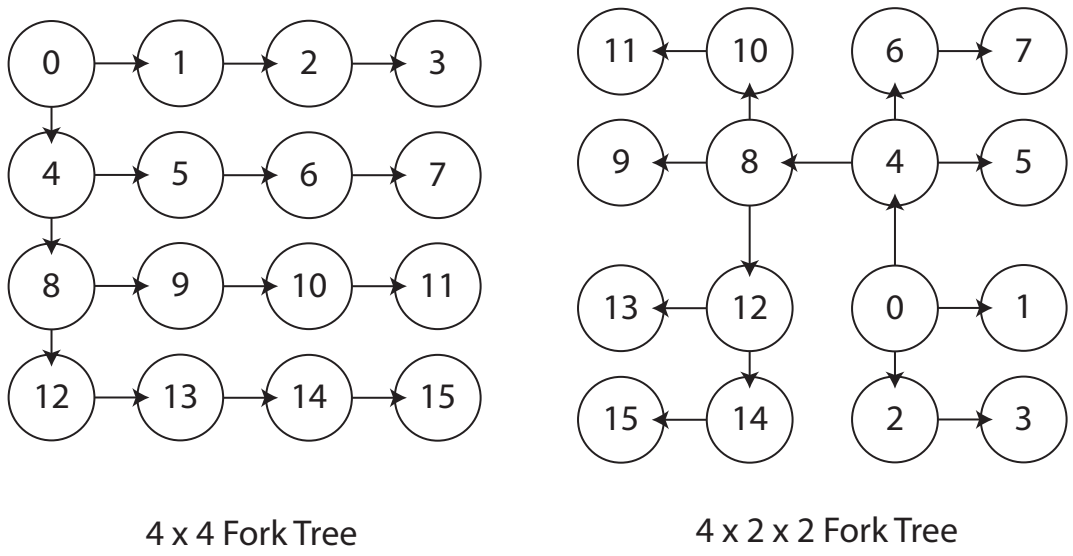


Figure 5.12: Example Fork Trees

```
MakeForkTree(nodelist, dimensions)
 1  if length(dimensions) == 1
 2     then return MakeKaryTree(nodelist, 1)
 3  stride ← 1
 4  for i ← 1 to length(dimensions)
 5  do stride ← stride × dimenions[i]
 6  tmpnodes ←  empty list
 7  for i ← 0 to dimensions[0] − 1
 8  do
 9       child ← MakeForkTree(nodelist[stride ∗ i, stride ∗ i + stride),
10       dimensions[1, length(dimensions)))
11       tmpnodes.append(child)
12  return MakeKaryTree(tmpnodes, 1)
```

Figure 5.13: Algorithm for Fork Tree Construction

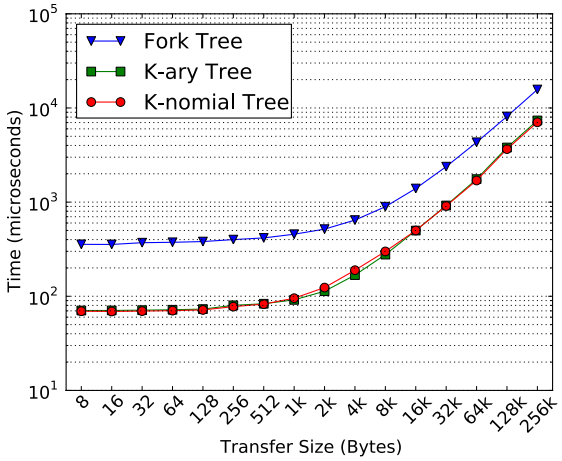


Figure 5.14: Comparison of Tree Shapes for Strict Broadcast (2048 cores of the IBM BlueGene/P)

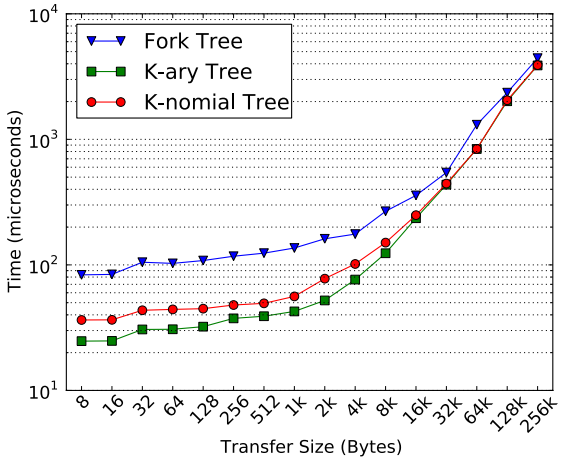


Figure 5.15: Comparison of Tree Shapes for Loose Broadcast (2048 cores of the IBM BlueGene/P)

Notice that the trees are constructed by composing 1-ary trees. While this seems like a natural fit to the network topology, the main drawback of these trees is that they induce many more hops into the collective. For example, given a mesh with $N \times N$ nodes a fork tree that maps to this network will have $O(\sqrt{N})$ hops while a comparable binary or binomial tree will contain $O(log_2\ N)$ hops, which is a much smaller. In the latter case the hardware is responsible for routing the messages over the torus rather than trying to explicitly manage the routing in software. Figures 5.14 and 5.15 shows the performance of the different tree shapes on 512 nodes of the BlueGene/P for a strictly synchronized and loosely synchronized broadcast respectively. The nodes for this configuration are arranged in an $8 \times 8 \times 8$ 3D torus. Thus as the data show the lower number of hops induced by the trees leads to better performance. However, messages destined for different parts of network can overlap. Thus, when bandwidth is the primary consideration, the Fork Tree will minimize the contention since the hops in the tree maps naturally to what the network supports. Thus the Fork Tree, which does not over commit the bandwidth on the links, performs better at larger message sizes. However, in our experiments the Fork Tree never yielded the best performance indicating that the generic trees are a better fit. Future work will validate this at larger scale. The BlueGene/P also supports collectives in the hardware, which will be discussed later in this chapter.

Other Trees

Some networks, such as the SeaStar networks found in the CrayXT systems, use a three dimensional torus to route the data through the network. However, the job schedulers on the systems we use do not guarantee that the nodes are contiguously laid out on the torus, thus enforcing an algorithm that assumes an underlying torus network when it might not be the case has severe performance implications. In this scenario each node is located at

a set of Cartesian coordinates and a spanning tree can be constructed to minimize the number of physical hops between two levels of the tree. The optimal tree that spans these coordinates is called a *Euclidean Steiner Tree* however the exact solution is NP-complete[90]. There are approximation algorithms that can be used to construct these trees that yield good performance in practice[72]. The algorithms we present in this dissertation have been designed to work with arbitrary tree topologies, thus if these custom trees are desired, future work can generate the topology without any additional effort to recode the collectives themselves.

In addition to the tree classes shown above, the software infrastructure in the GASNet collective library supports composing different tree classes together. For example one could build a set of t trees that each span different portions of the network (e.g. different racks in a machine room). The roots of these t trees can then be connected via another tree class that maps to the switching architecture that connects the subcomponents of the networks. However, our experimental results for our benchmarks did not show an improvement with this approach. We believe that these trees will be important when the collectives are applied to whole machine room scale. Future work will explore if and when the tree composition is a beneficial optimization.

5.1.3 Address Modes

One of the key differences between one and two-sided communication is that with one-sided communication the initiator of the communication has to provide both the source and destination addresses for the data, necessitating the addresses on the remote nodes to be known ahead of time. GASNet manages the remote addresses by designating a segment of the address space as sources (for gets) and targets (for puts) of one-sided communication. When the jobs start up all the threads exchange the addresses of the remote segments so that any node can access data in the segment on any other node. Our interface to the collectives allows the user to specify the addresses in the following two ways:

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. Notice that even though there are $O(N)$ addresses passed to the collectives, by leveraging trees only $O(log\ N)$ of them are accessed.

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.

Each of these modes has their advantages and disadvantages. The advantage of specifying all addresses when the collective is spawned is that it circumvents the overhead of discovering addresses. However, it does require that the collective be passed the addresses for all the

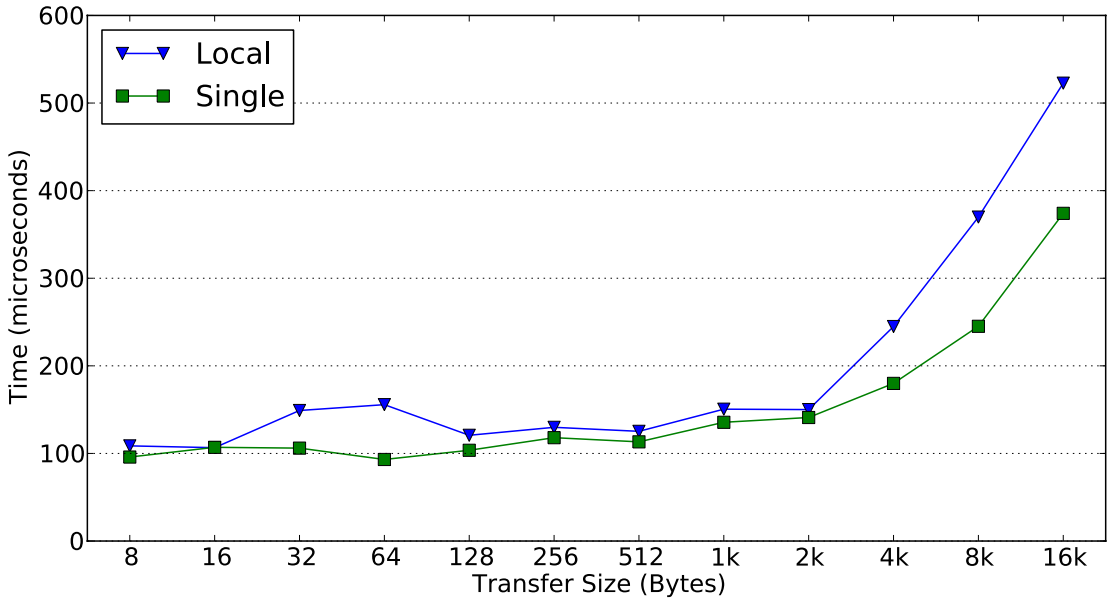


Figure 5.16: Address Mode comparison (1024 cores of the Sun Constellation)

threads involved in the collective, which at scale can be quite large. However, notice that even though a data structure that can hold all the threads is needed, the number of addresses accessed by the nodes is typically the number of peers a node communicates with. This typically scales as $O(log\ N)$ where N is the total number of nodes in the collective assuming the above mentioned trees are used. Depending on the language, these addresses might be easy to calculate. In UPC for example, the collectives are performed over shared arrays. The language also specifies the methodology of calculating the addresses of the different parts of the array on different threads, thus making it very easy to calculate remote addresses with a few arithmetic operations rather than explicit communication.

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 anonymous 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.

Figure 5.16 shows the performance advantages of a knowing all the addresses versus having to discover the addresses on 1024 cores of the Sun Constellation. As the data show, the *Single* address mode can consistently realize better performance than *Local*. One would expect the performance difference to be only significant at lower message sizes. However, as the data show, the cost of the added synchronization also has a significant impact higher 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. The next section goes into more detail about the tradeoffs between the different data transfer

mechanisms that are applicable in each case.

5.1.4 Data Transfer

Once the communication topology has been fixed, the next major issue to worry about is how the data is communicated between the parents and children. Since the intermediary nodes of a tree will need to receive and forward data to their subtree, they need to know when all the data has arrived. Thus along with the one-sided data transfer, we need mechanisms to signal when the data has arrived. In this section we explore three different data transfer mechanisms and explore their tradeoffs.

Signaling Put

For *Single* address collectives, since the parent knows the destination address of where the data needs to go, we use a *Long* active message to transfer the data (see Section 3.2.2). The function that is invoked on the target node sets a state bit indicating that the data has arrived in the target node. For the loosest synchronization mode described in the previous chapter, the data can be transferred to the leaves without point-to-point synchronization since the user has indicated that the synchronization will be handled elsewhere. When a node arrives in the collective, it waits until the parent has sent the data (unless it is a leaf node of a loosely synchronized Broadcast) and then forwards the data down to the children through the same signaling put.

Eager

For *Local* address collectives that would like to avoid the latency costs of the address exchange or *Single* address collectives that require the user buffers not be touched until the children have entered the collective, the data needs to be transferred into an auxiliary buffer on the child's node. For those cases we use a *Medium* active message. The Medium active message contains both message and payload. When the data arrives at the target node it is transferred into a buffer managed by the collectives library. Once the child has entered the collective, the data is then moved into the user space and then passed further down the tree. Notice that the operation is still one-sided in nature since the parent can exit the collective once the active message has been sent.

Rendez-Vous

The main drawback with the *Eager* mechanism is that it requires the collective library to manage memory for the bounce buffers. For machines that have large processor counts with low per node memory (i.e. the IBM BlueGene/P) using a significant percentage of the memory for bounce buffers is impractical. 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 space is not exhausted. Thus to avoid these issues we employ another approach. For larger messages we use a Rendez-Vous. The parent sends a *Short* message that contains only

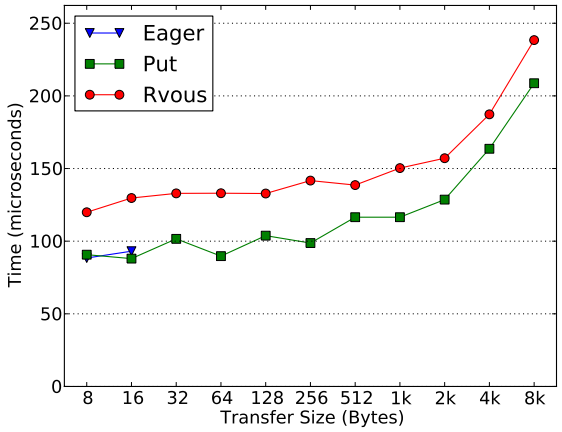


Figure 5.17: Comparison of Data Transfer Mechanisms (1024 cores of the Sun Constellation)

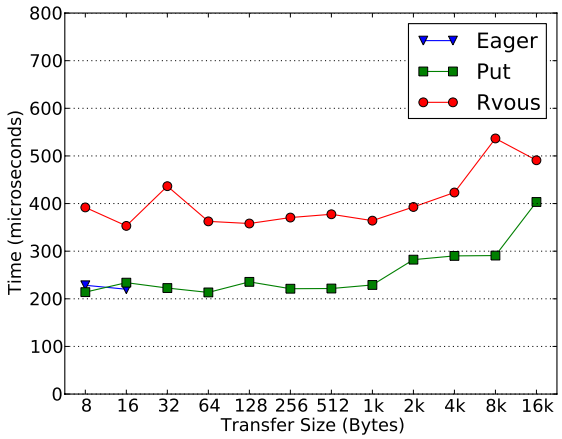


Figure 5.18: Comparison of Data Transfer Mechanisms (2048 cores of the Cray XT4)

the address of the buffer that contains the data. The children initiate get operations once they know the address. Once the gets have completed the children then send another *Short* active message to the parent that increments an atomic counter. The parent spins until all the children have incremented the counter indicating that the transfers have completed at which time they can exit the collective. This collective naturally enforces the requirement that data movement into and out of a specific node be active only while the node is in the collective.

Microbenchmarks

Figures 5.18 and 5.17 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. Notice that when an Eager operation is available the performance is on par with the Signaling Put however it is only applicable at limited scale. From the figures we can draw the conclusion that when applicable, taking advantage of the Global Address knowledge can yield good performance advantages.

5.1.5 Nonblocking Collectives

The performance benefits of overlapping communication with computation have been well studied[128]. 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[43, 97]. The loosest synchronization model in UPC states that data movement can occur until the last processor leaves the collective.

However this forces at least one processor to be active in a collective while the collective is in flight. To alleviate this restriction, we have designed our collectives to be nonblocking. Like a nonblocking memory transfer, the operation returns a handle 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. With the prevalence of heterogeneous multicores, future work can explore devoting a specialized lightweight thread to ensure progress. The issues of network attentivity pose interesting questions for automatically tuning collectives since it affects how quickly intermediary processors can forward the data to other processors. If the intermediary processors are not very attentive, due to external factors outside the collective libraries, the collectives themselves will appear to run very slowly. Section 5.4 describes two important kernels implemented to take advantage of nonblocking collective operations.

Software Architecture

The collectives in GASNet are broken into two distinct phases, collective initiation and a set of state machines that perform the collective itself. The collective initiation marshals the arguments and constructs the operation specific data structures (e.g. allocating the buffers for an *Eager* message transfer). Once the operation has been setup, it is then enqueued onto a queue of active operations. As soon as the operation has been successfully queued the initiation operation returns and the user can perform other operations that are not relevant to the collective. Once the runtime system completes the operation the handle that was returned with the initiation is marked as completed. It is an error to read or modify the collective's source or destination buffers until the operation has been completed.

Based on the synchronization flags explored in previous sections, some of the collective implementations will require a full barrier before and after the collective operation. Thus, to ensure that these collectives can also be split-phased, the barriers must also be divided into a notify and a nonblocking try. In the notify, step a thread advertises that it has entered the barrier. Successive try operations check if all other threads have entered the barrier.

Each collective is implemented as a finite state machine where each state specifies a certain action or phase of the collective. Figure 5.19 shows an example state machine to implement Broadcast using *Signaling Put*. Data arrival or a synchronization event cause a state change and different collectives on the operation queue can be in different stages thus allowing pipelining amongst the operations. A blocking collective call is simply a nonblocking call immediately followed by a wait. While our examples here only show broadcast as an example, we have implemented all the collectives in a similar fashion so they are all nonblocking by default.

```
BROADCAST-SIGNALING-PUT-FSM(op)
 1  switch
 2    case op.state == 0 :
 3         if op.syncflags & IN_ALL_SYNC and
 4           op.barrier_children_reported < op.child_count
 5           then return NOTDONE
 6           else   send short active message to increment
 7                  op.barrier_children_reported on op.parent
 8
 9         op.state++
10    case op.state == 1 :
11         if  op.parent signaled
12           then op.state++
13           else  return NOTDONE
14    case op.state == 2 :
15         for  each child in op.children
16         do  Signaling Put data to child
17         op.state++
18    case op.state == 3 :
19         if op.syncflags & OUT_ALL_SYNC
20           then barrier_notify(op);
21
22         op.state++;
23    case op.state == 4 :
24         if op.syncflags & OUT_ALL_SYNC
25           then if barrier_try(op) returns DONE
26                    then op.state++;
27                    else  return NOTDONE
28  return DONE
```

Figure 5.19: Example Algorithm for Signaling Put Broadcast

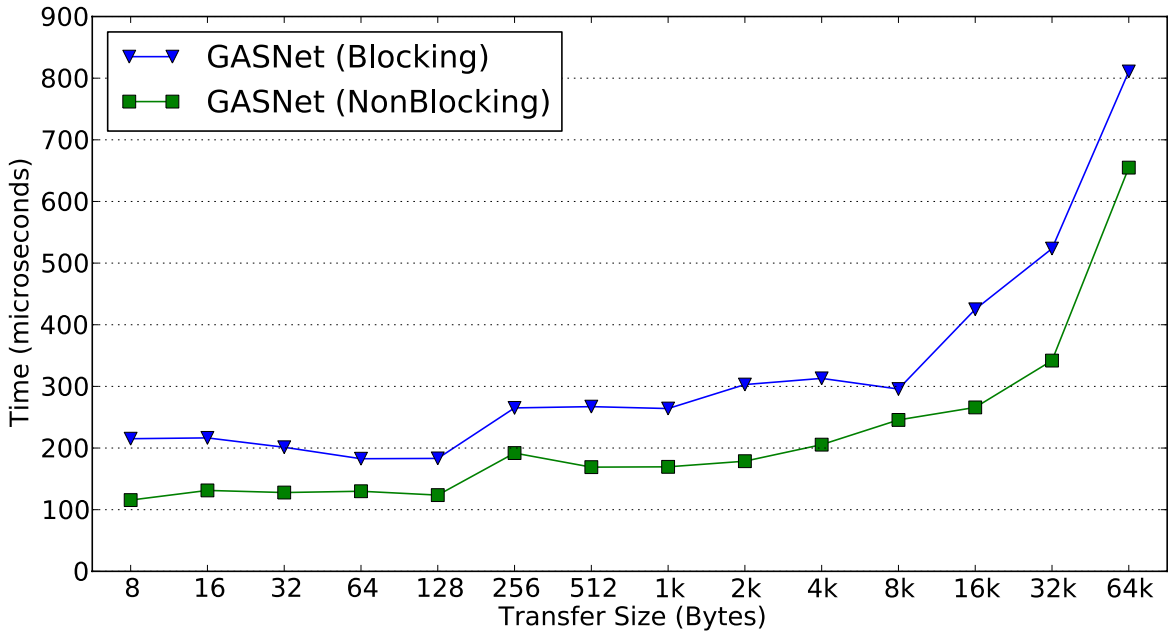


Figure 5.20: Nonblocking Broadcast (1024 cores of the Cray XT4)

Microbenchmarks

To measure the impact of nonblocking collectives we measure the time taken to initiate a set of collectives and sync them after all of the collectives have been initialized. Thus before the first wait is finished all the collectives are simultaneously in flight. We are thus able to leverage communication/communication overlap.

Figure 5.20 shows the performance of a nonblocking broadcast of varying sizes compared to its blocking counterparts. As the data in Figure 5.20 show, the nonblocking collectives are able to realize significantly higher performance than their blocking counterparts. For the nonblocking case, the collectives are allowed to be pipelined behind each other thus increasing the overall throughput of all the operations.

5.1.6 Hardware Collectives

On some platforms, such as the IBM BlueGene/P, the hardware and the communication APIs also provide hardware or vendor collective implementations. GASNet incorporates these collective operations when available so that the vendor provided collectives could be leveraged if applicable. Figure 5.21 shows the performance of three different GASNet Broadcast implementations. The *Point-to-Point* collective is implemented in GASNet and uses only Active Messages, puts, and gets. The *DCMF Tree* collective exposes the BlueGene/P's hardware supported broadcast through the GASNet collective API. *DCMF Torus* shows the performance of the vendor optimized collective that only uses point-to-point operations and does not use the hardware collective features. This is again exposed through the GASNet API. Since the hardware collective is only available when all the threads in the program take

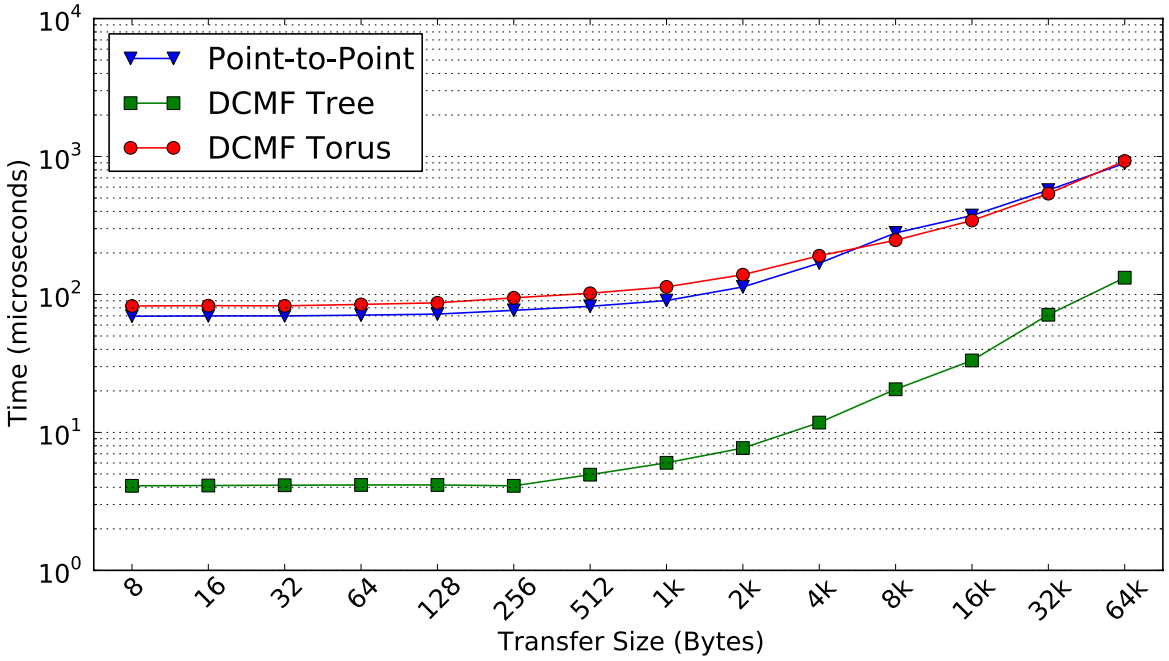


Figure 5.21: Hardware Collectives (2048 cores of the IBM BlueGene/P)

part (i.e. MPI_COMM_WORLD), the Torus implementations provide a proxy for the performance with teams (i.e. only a subset of the nodes are involved in the collective). Teams will be more extensively discussed in Chapter 9. Thus as the data show, the GASNet collective library can deliver better performance than the vendor optimized point-to-point collectives and thus it is not always advantageous to rely solely on vendor-optimized collectives, rather they must be included in the search space and compared alongside the GASNet collectives.

5.1.7 Comparison with MPI

Figures 5.22, 5.23, 5.24 shows the performance of the best Broadcast in GASNet compared against MPI for both loose and strict synchronization on the Sun Constellation, the Cray XT4 and the Cray XT5. The GASNet implementation for Broadcast was chosen by searching over all the aforementioned trees and address modes. On the Sun Constellation system GASNet beats MPI at 128 bytes by 27% but at the other sizes shown MPI yields the best performance. However, MPI's advantage over GASNet is at most 11%. As the also data show, a strict Broadcast in GASNet beats MPI in all cases on the Cray XT systems. GASNet takes 35% of the time that MPI does to complete the 4kByte Broadcast on 1536 cores of the Cray XT5 (a 65% improvement in performance). For the Cray XT4, GASNet achieves a 28% improvement in performance at the same size.

When we compare a loosely synchronized Broadcast on the platforms, the data show that MPI consistently outperforms GASNet at all the small message sizes except for a 128 and 1024 bytes on the Cray XT4. Our hypothesis is that MPI tries to minimize the time

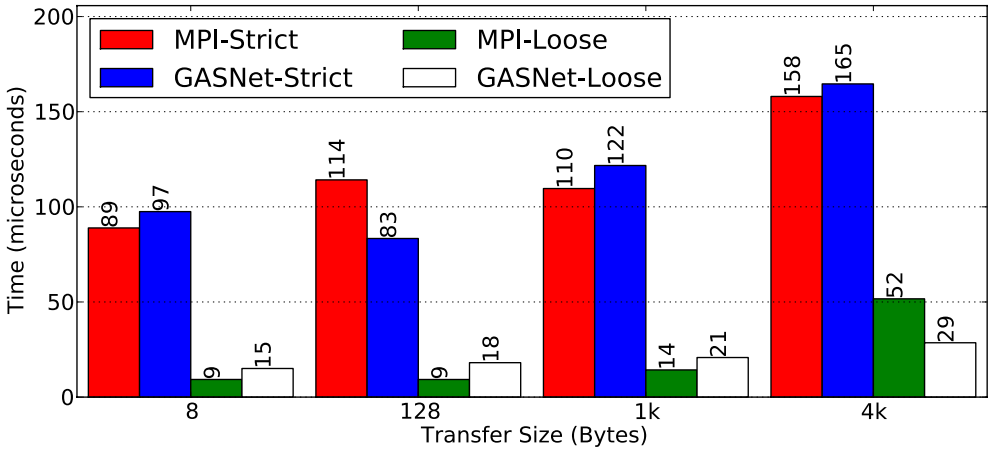


Figure 5.22: Comparison of GASNet and MPI Broadcast (1024 cores of the Sun Constellation)

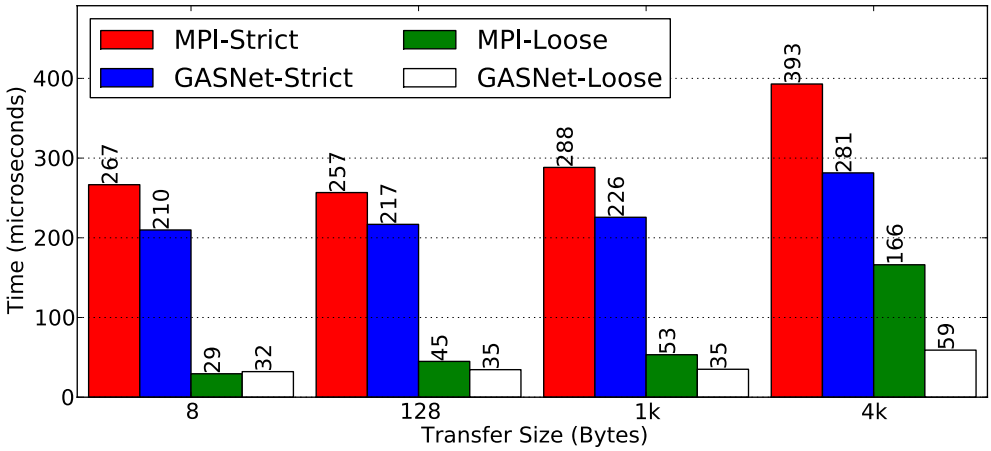


Figure 5.23: Comparison of GASNet and MPI Broadcast (2048 cores of the Cray XT4)

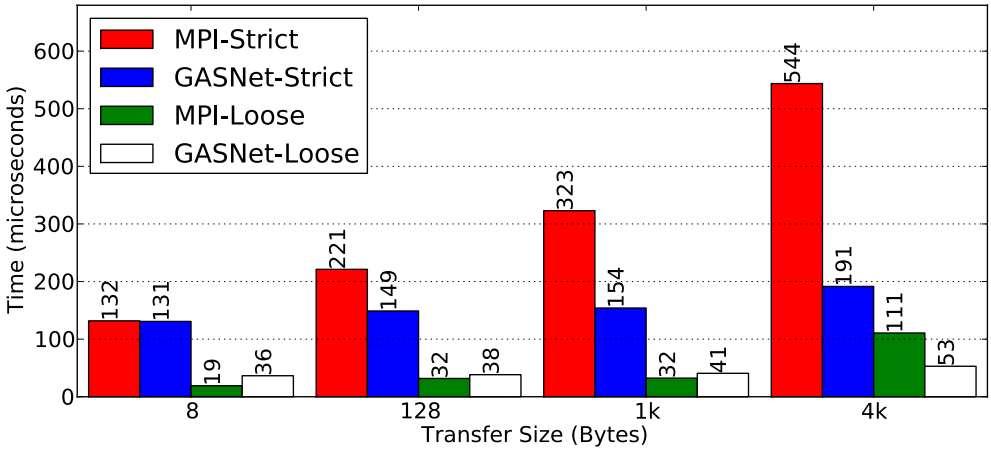


Figure 5.24: Comparison of GASNet and MPI Broadcast (1536 cores of the Cray XT5)

the root spends in a collective call by copying the data to an internal buffer and is thus able to return immediately[1]. The MPI implementations have also been highly optimized to minimize the software overhead of the library [119]. However for 4 kByte messages, the one-sided communication in GASNet yields good performance. The performance differences are consistent with the flood bandwidth microbenchmark results in Section 3.2.1 and our previous work [28]. On the Cray XT4, a 4kByte loosely synchronized Broadcast yields a 64% improvement in performance.

From the data we can draw the conclusion that the performance of the GASNet Broadcast, (a Broadcast written with a one-sided programming model in mind) can realize good performance improvements compared to Broadcast in MPI.

5.2 Other Rooted Collectives

For the rest of the collectives we analyze, each thread sends or receives a globally unique piece of data. In this section we show the performance for the other collectives and discuss some of the tradeoffs associated with their implementations.

5.2.1 Scratch Space

In addition to the infrastructure described thus far, we need to add a new piece of infrastructure to accommodate the other collectives. In the case of Scatter, all of the threads will receive $1/T^{th}$ of the input data where T is the total number of threads and thus each thread is only required to provide a buffer that is $1/T$ of the total buffer space at the root. However, for the reasons cited in Section 5.1.2, it is often desirable to be able to leverage trees. Therefore the intermediary nodes in the tree will have to receive data on behalf of their children to forward down. However, the user space buffers are not wide enough to handle the data for the entire subtree.

We will thus need to designate space in the runtime that is usable for auxiliary space called "scratch space." The scratch space is temporary storage that can be used by the collective for communication. Since GASNet requires that the targets or source of one sided communication be part of the GASNet segment this scratch space needs to be carved out of the existing GASNet segment. Because we will be using space from the segment, minimizing this footprint is an important constraint so that applications can get as large a fraction of the system memory as possible.

We implement a distributed scratch space management system within the GASNet collectives. Each node constructs a scratch space request that contains the peers it will be communicating with, how much scratch space it needs on those peers, and how much scratch space is needed locally. It is up to the different collectives to specify their scratch space requirements and to ensure that they are self-consistent across the nodes. The scratch space manager only performs a minimal amount of error checking. Since the only client

[1]Since the algorithms for the CrayXT systems are not publicly available it was not possible to validate this hypothesis.

of the scratch space management system is the GASNet collectives, the error checking is incorporated into the GASNet collectives library.

The allocator keeps a local copy of the state of the remote node's scratch space locally. If the allocation fits, then the space is allocated and the pointers are updated and thus, without any extra communication, a designated piece of the remote memory is available for the collective to use. However, if the scratch space is misconfigured or the allocation is too large to fit, then the collective is held until the remote nodes indicate that all previous collectives have been drained and that space is again available. Notice that since the space is designed to be temporary, once a collective completes the space can be recycled. There are many possible enhancements that can be made to this system, however we have found that for many cases, this management system yields good performance. The control signals needed for the distributed scratch space management are implemented through *Short* active messages descried in Section 3.2.2. Future work will address how to upgrade this system to use a more aggressive management to minimize the number of drain signals that need to be sent.

5.2.2 Scatter

A Scatter is similar to the Broadcast with one key difference: the data that the root sends are personalized messages. Thus if each of the threads receives B bytes of data from the root, then the root's source array has a length of BT bytes where T is the total number of threads. Even though each thread has a personalized message from the root, it is still valuable to implement the operation through a tree and forward the data through intermediary nodes. Assuming there is a nontrivial amount of overhead o involved in sending a message on the network, a flat scatter would incur a cost of $N \times o$ for the last child to get its data. However, a tree based scatter would incur about $(lg\ N) \times o$ time.

In Broadcast this decrease in latency is almost free since the amount of data sent to the subtrees is the same irrespective of the size of the subtree. However, for Scatter, the intermediary nodes must receive their piece of the data along with the data for the entire subtree they are responsible for. Thus if there are λ levels in the tree (level 0 is the root), then an intermediary node at level i will have to manage data for N/k^i nodes for a K-nomial or K-ary tree of radix k. In the case of the broadcast, where the data was replicated for the children in the intermediary nodes, the Scatter requires extra bandwidth to be implemented through trees. As with Broadcast, we also minimize the number of network communications by having only one representative thread per node manage the communication. Once the data arrives at a node a `memcpy()` is used to transfer the data into the other buffers.

The aforementioned problem with extra space is implemented through the scratch space management system described above. When a scatter operation starts up each thread requests enough space for the data it needs plus the subtree it owns. It also requests the appropriate amount of space on each of the children. Once the scratch space request has been granted and the data arrives from the root, the source data is split into the parts destined for the different subtrees and sent down the subtree[2]. If the tree is rooted at a

[2]This requires that the entire subtree be a contiguous set of nodes. If this is not the case then a special

node $R \neq 0$, then the entire source buffer is rotated left until the first element of the buffer is the root. The rotation can be done with two `memcpy()`s: one to move the first moves the initial R blocks to the end of the buffer and the second moves the remaining $N - R$ blocks to the start.

Figure 5.25 shows the performance of Scatter compared to MPI on 256 cores of the Sun Constellation cluster. For GASNet the best result from an exhaustive search over power of 2 radix tree shapes is reported. As the data show the performance of GASNet for both strictly and loosely synchronized collective is higher than the comparable MPI performance except at small message sizes. As discussed with Broadcast, at the smaller message sizes we suspect that MPI has smaller software overhead to initiate the collectives. Future work will try to improve the GASNet performance to match. As the data also show, the loosely synchronized collectives are able to realize better performance by pipelining the Scatters behind each other. However, unlike Broadcast, where each of the intermediary nodes have the same amount of data to send to their children, the data needed to be sent down to the subtree shrinks by a factor of 2 at each level down the tree. Thus the levels at the bottom of the tree will have less work to do than the levels at the top and be idle for longer. Therefore, the benefits of pipelining the scatters behind each other are not as pronounced at larger message sizes than at smaller ones. The bandwidth out of the nodes at the top of the tree becomes the limiting step.

Figures 5.26and 5.27 shows the same data on 512 cores of the Cray XT4 and 1536 cores of the Cray XT5. As the data show, the GASNet performance is consistently better than MPI's for Loose synchronization. Thus when the synchronization mode can be loosened, the one-sided communication model is well positioned to take advantage of it. When Strict synchronization is desired, both GASNet and MPI realize the same application performance. As the data also show, as the message sizes get larger, the performance difference between the different synchronization modes is a lot less pronounced and thus leading one to conclude that the bandwidth (which is the same for both) is the dominant factor.

5.2.3 Gather

The inverse of Scatter is Gather; each thread has a contribution that it wishes to send to the root thread. If one were to write Gather with a flat tree then the root of the Gather would have to receive messages for all N nodes. Most systems will dedicate some per connection resources to manage the data movement for each peer. However they usually limit the number of possible active peers to a fixed constant to avoid excess usage or allocation of these resources. However, if the number of peers exceeds these resources, the resources are dynamically reallocated to the new peers. This movement of resources can be an expensive process and thus it is often beneficial to limit the number of incoming peers to a particular node. As the message size grows larger, resending the data multiple times in the network can be costly. Thus there is a tradeoff between the per link connection utilization and the overall network bandwidth.

buffer needs to be used to reorder the input data to compensate which can be expensive in terms of extra storage and data movement for large node counts or large message sizes.

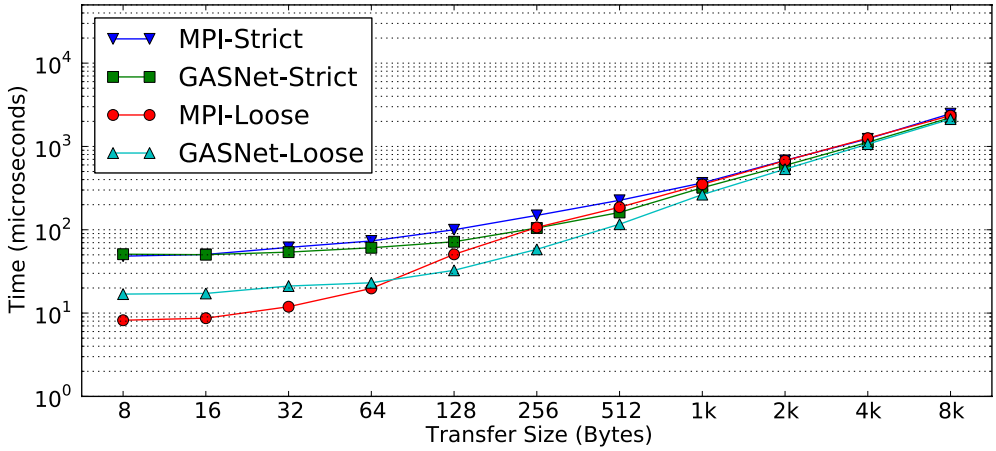


Figure 5.25: Scatter Performance (256 cores of the Sun Constellation)

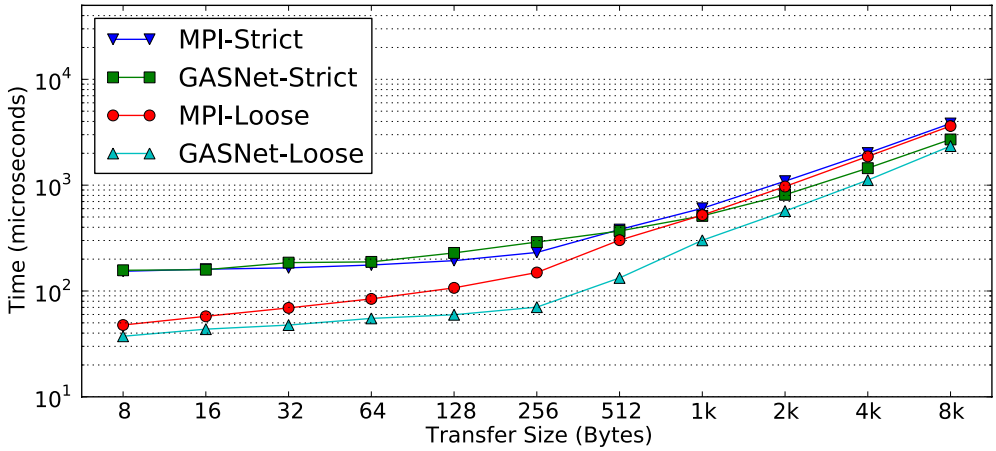


Figure 5.26: Scatter Performance (512 cores of the Cray XT4)

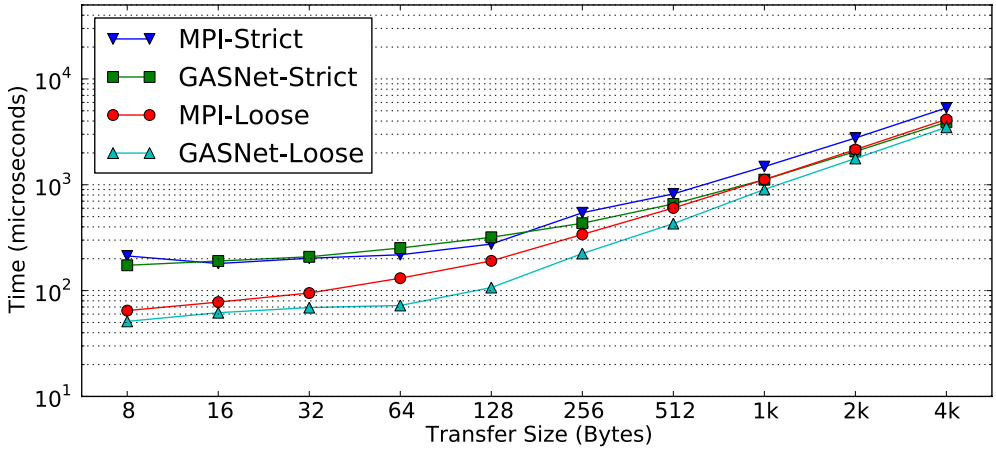


Figure 5.27: Scatter Performance (1536 cores of the Cray XT5)

For these reasons we will again chose to implement Gather through a tree[3]. Like Scatter, each thread will allocate some scratch space large enough to fit the contribution for the entire subtree. Once it has received the data from the subtree the data is forwarded up to the parent. To implement this data transfer for Gather we modify the *Signaling Put* construct above to increment an atomic counter rather than set a state variable and we shall call it a *Counting Put*. Once the counter on the parent is equal to the number of children then we know that it all the data has arrived. If the root R of the Gather is nonzero then the data is rotated right by R to get the data in the correct order. Figure 5.28 shows the performance of a strictly and loosely synchronized Gather. Similar to Scatter, the GASNet performance shown is the best over a search over power-of-two K-ary and K-nomial trees. Like Scatter, the performance, GASNet beats MPI for most cases when a strict synchronization is required, however for loosely synchronized scatters, MPI yields slightly better improvement at lower message sizes which we again suspect is caused by the lower software overhead of MPI to initiate a Gather. Like Scatter, the benefits of pipelining the Gathers behind each other are visible up to 2 kBytes. After which the bandwidth becomes a dominant concern and thus both synchronization modes realize the same performance.

Figure 5.29 shows the performance of Gather on the Cray XT4. As the data show both GASNet and MPI achieve about the same performance at low message sizes, however, as the transfer sizes become larger then the GASNet performance is better. As the data also show for both models, the Gather performance difference between strict and loose synchronization becomes less pronounced at larger transfer sizes where the bandwidth becomes the dominant concern.

5.2.4 Reduce

Reductions are also prevalent in many parallel applications. Each thread has a contribution to a global operation (e.g. summation) whose result will be located on a final root thread. Reductions are also implemented through trees to leverage the parallelism. Intermediary nodes aggregate the results for their subtree to pass onto the parent. Thus the aggregation operations are parallelized over all the nodes. However this does come at the cost of added latency to get to the top of the tree.

To implement a reduction each node will allocate scratch space large enough to fit the contributions from each of the children. Each child first performs a reduction for all the threads within a node so that only one of the threads needs to perform the reduction. Each child sends its contribution to the parent's scratch space through a signaling put. As soon as the parent receives a contribution from a child it applies the reduction operator to the result it already has. With the one-sided communication model the children can be sending data to the parent while it is computing the reduction. Thus, to ensure that live data is not overwritten, each child writes to a separate space on the parent node. Conversely, one could minimize the buffer space by adding an extra level of synchronization between the parent and the children. Our experiments have suggested that the former yields the best performance and hence we favor these algorithms for a wide variety of message sizes. Once

[3]The trees for Gather are exactly like Scatter and Broadcast except that all the edges are reversed.

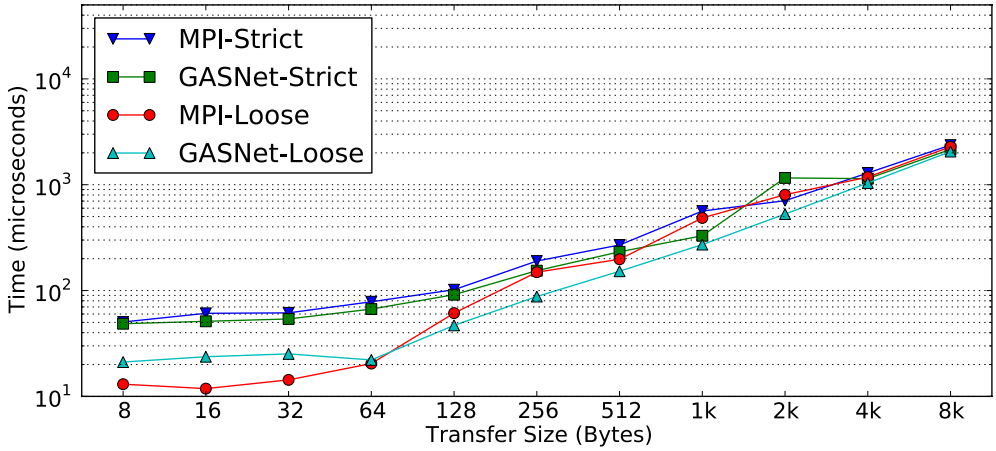


Figure 5.28: Gather Performance (256 cores of the Sun Constellation)

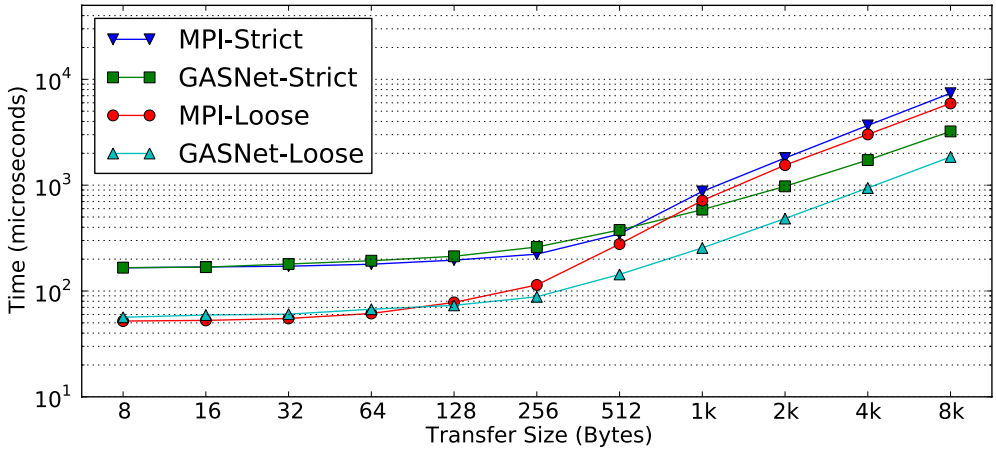


Figure 5.29: Gather Performance (512 cores of the Cray XT4)

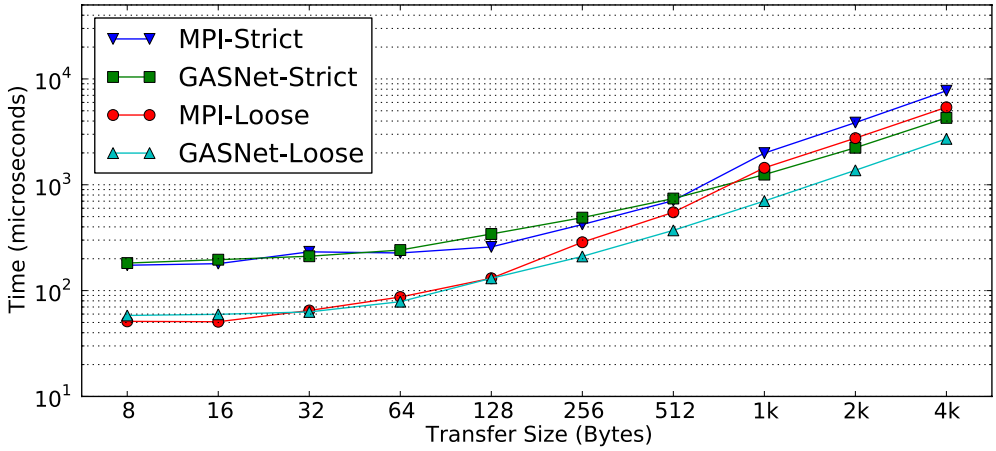


Figure 5.30: Gather Performance (1536 cores of the Cray XT5)

the reduction has been applied to all the children the data is then forwarded up the tree.

If Reduce is implemented through a K-ary tree, then each of the intermediary nodes will have a fixed number of children and therefore each node will incur a constant number of flops. However, with the K-nomial trees then the nodes at the top of the tree will have more children and therefore more reduction operations. Thus two factors come into play for a reduction: (1) the computational overhead associated with applying the operation and (2) the network overhead associated with transmitting the data.

Figures 5.31 and 5.32 show the performance of a 4-byte and 1k-Byte Integer Reduction respectively, on a varying number of cores. For the K-ary and K-nomial data points an exhaustive search over all power of two radices have been searched and only the best is reported. As the data show, the performance difference between the two tree classes is at most 11%. This indicates that while the best tree choice is important the overhead associated with having a varying number of children for this platform is not an important consideration. In addition, the data also shows that the performance of a reduction in GASNet is consistently higher than it's MPI counterpart.

Figures 5.33 and 5.34 show the performance of Strict and Loosely synchronized reductions respectively on 2048 cores of the Cray XT4. As the data show, for the strictly synchronized collectives, the K-nomial trees tend to be consistently better. Since the trees are unbalanced, nodes at the higher portions of the tree will start to get more useful work from their children with smaller subtrees. The heavier children will tend to send the data later in the operation thus staggering the data arrival. From the data, the largest difference in performance, at 4 bytes is more than 20% with an average performance difference of 10%. On the Cray XT4 a loosely synchronized Reduce in MPI yields better performance than GASNet. GASNet's active message implementation on top of the network available for the Cray XT4 (Cray Portals) incurs a performance overhead due to essential features such as flow control [38]. Future work will improve the active message layer underneath the collectives to improve the overall collective performance. Once the improvements have been made we expect that the performance will be comparable to MPI. We suspect that the communication and computation pattern induced by Reduce interacts poorly with the underlying Active Message layer. Future work will validate and address this hypothesis.

Figures 5.35 and 5.36 show the performance of Strict and Loose synchronization on 1536 cores of the Cray XT5. Unlike the CrayXT4, the K-nomial trees have a consistent advantage over K-ary trees for a strictly synchronized Reduce. For a loosely synchronized Reduce the results are consistent with the Cray XT4, the K-nomial trees are the clear winner at smaller message sizes while the K-ary trees are the winner at higher message sizes. However, the MPI implementations outperform the GASNet ones for both synchronization indicating that there are further algorithms that must be added to the tuning space. Again we suspect that this is caused by the overhead of the Active Message layer.

5.3 Performance Models

Throughout this chapter we have motivated many different techniques to implement a collective operation such as the number of threads per node, tree geometry, data transfer

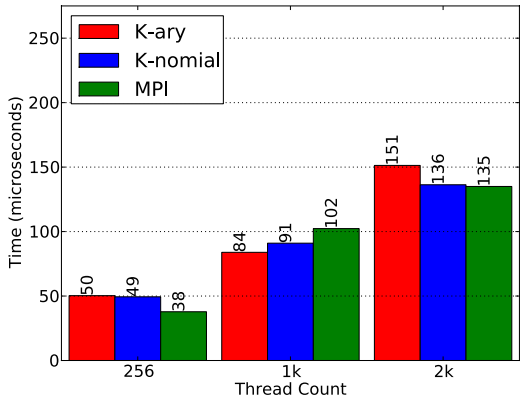


Figure 5.31: Sun Constellation Reduction Performance (4 bytes)

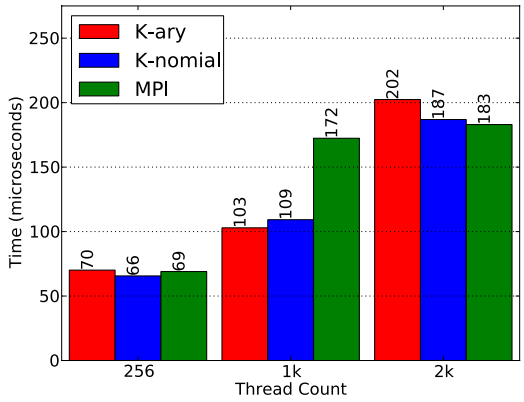


Figure 5.32: Sun Constellation Reduction Performance (1k bytes)

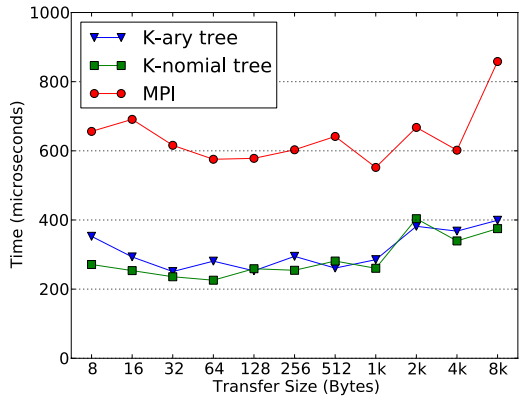


Figure 5.33: Strictly Synchronized Reduction Performance (2048 cores of the Cray XT4)

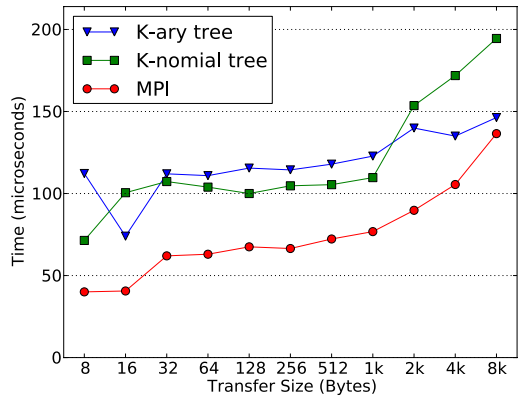


Figure 5.34: Loosely Synchronized Reduction Performance (2048 cores of the Cray XT4)

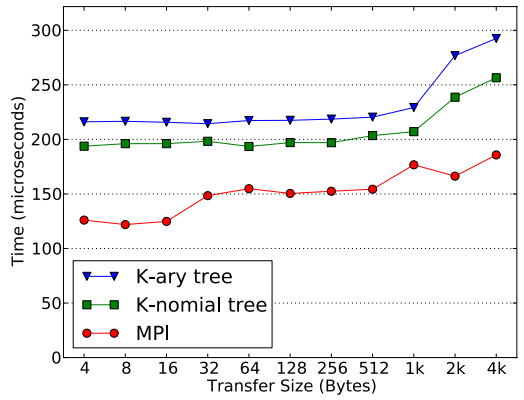


Figure 5.35: Strictly Synchronized Reduction Performance (1536 cores Cray XT5)

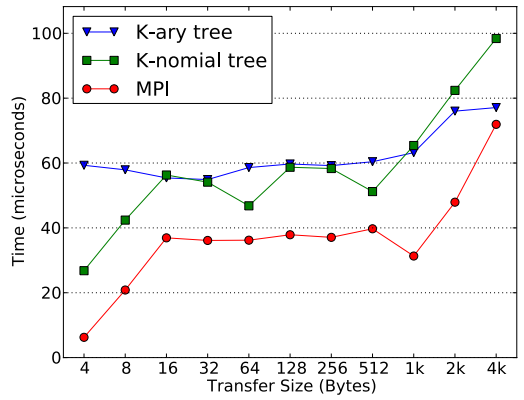


Figure 5.36: Loosely Synchronized Reduction Performance (1536 cores Cray XT5)

mechanism, etc. Due to the large space of possible algorithms, searching over all possibilities will be prohibitively expensive even if the tuning is done outside the critical path of the collectives. One way to prune the search space is by using performance models. Performance models will be able to use static parameters, such as processor count and message sizes, in order to quickly predict the performance of various algorithms. The models need not be perfectly accurate as long as they can predict the implementations that should not be included in the search and be able to give a rough approximation to what the search space will look like.

To simplify the explanation, we only present the models for K-ary trees. The models for K-nomial trees are very similar. In this tree class, every intermediary node sends to at most $(k > 1)$ children and thus there will be $\lceil log_k N \rceil$ levels (where N is the total amount of processors involved). We will focus on modeling the time it takes the *last* leaf processor to receive the data. We will assume that once this last processor gets the data, the entire collective is complete. We can model the performance by analyzing the last processor at every level. We will use the LogGP model [10] which is a variant of the LogP[61] in our analysis. In this model L represents the time taken by the first byte of data to leave the source node and arrive at the destination node. Neither the source nor the destination processor is busy during this time. We will also use o to represent the overhead of sending or receiving a message (i.e. the time the processor is busy receiving a message from the network or sending a message into the network). The gap term, g, represents the amount of time a processor has to wait before injecting another message into the network. And P is the number of processors that are used (throughout this dissertation, we have used N to denote this value). We also add the inverse bandwidth (i.e. microseconds per byte) parameter G from the LogGP. We first present the model for Scatter. The models for the other collectives will be a slight variant on this model.

5.3.1 Scatter

As we stated earlier, Scatter can leverage trees but requires that the same data be sent multiple times on the network to get to the leaf. Thus for small messages we theorize that the regular deep trees, which are able to better parallelize the message injection overhead, are useful since the overhead is the dominant factor. However for large message sizes the cost of sending into the network many times will dominate the total costs. To be able to understand this tradeoff we will demonstrate how a performance model for Scatter can be constructed to show the tradeoffs and guide algorithm selection for the automatic tuner.

For a K-ary tree, a node has at most k children. We can break up the model into two distinct components. We will first consider the latency and then the bandwidth and sum these pieces together. For a tree with N nodes there are $\lambda = \lceil log_k(N) \rceil$ levels in the tree. At each nonroot level in the tree there is a cost of L for the data to transfer across the network and then an additional cost of o at the receiver to receive the data. Then for each child, a node must incur a cost of g to send the data. Thus by the time the data reaches the last leaf there have been λ latency and overhead terms or $\lambda \times (L + o)$. In addition, each of the levels above the leaf had to spend $k \times g$ to send data to their children for a total of $\lambda \times (k \times g)$.

Putting these two terms together we can model the total latency as $\lambda(L + o + kg)$

The next component we analyze is the bandwidth term. We assume that we have t threads per process for a total of Nt threads that participate in the scatter. If each thread is to receive B bytes then at each level we must send B bytes times the size of the subtree under the child node since each child gets a personalized message. The total subtree under a particular node at level i (the root is at level 0) can be approximated as $\frac{Nt}{k^i}$. Thus for each of the k children the bandwidth can be approximated as:

$$T_{scatter\ \ bandiwdth} = \sum_{i=1}^{\lambda} (BG \times \frac{Nt}{k^i})$$

$$= NBGt \times \sum_{i=1}^{\lambda} \frac{1}{k^i}$$

$$= NBGt \times [(\sum_{i=0}^{\lambda} \frac{1}{k^i}) - 1]$$

Using the formula for a geometric series:

$$\sum_{i=0}^{k-1} r^i = \frac{1 - r^k}{1 - r}$$

We can do some algebraic manipulation to get:

$$T_{scatter\ \ bandwidth} = k \times \sum_{i=1}^{\lambda} (BG \times \frac{Nt}{k^i}) = kGBt[\frac{N-1}{k-1}]$$

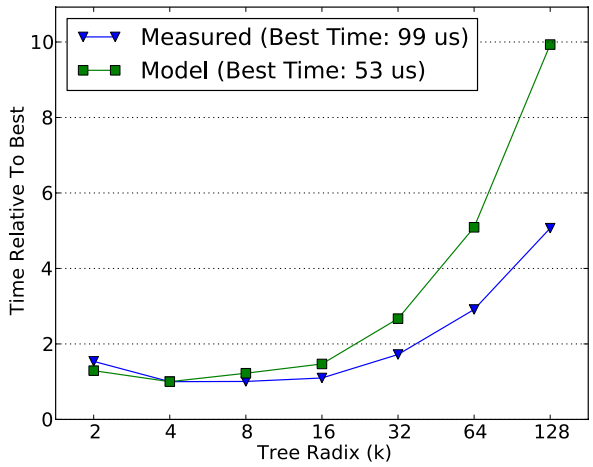
Thus the overall performance model for scatter with a K-ary can be written as:

$$T_{scatter} = \underbrace{\lambda(L + o + kg)}_{latency} + \underbrace{kGBt[\frac{N-1}{k-1}]}_{bandwidth}$$

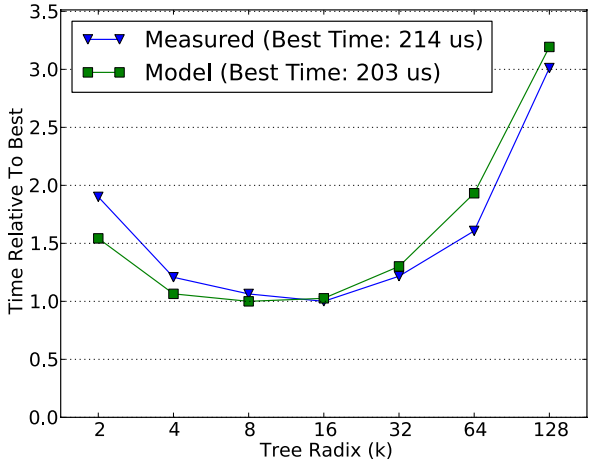
To verify this model we examine a Scatter on 1024 threads (256 Nodes) of the Sun Constellation for three different sizes of Scatters in Figures 5.37(a)- 5.37(c). Using a tester developed by related work [27], we can measure the LogGP parameters and approximate the overall parameters. Our measurements are shown in Table 5.1. The x-axis shows varying tree radices for different K-ary trees. The y-axis shows the time relative to the best tree. A value of one means that corresponding radix on the x-axis is the best. In practice our models do a good job of predicting the actual performance (the value at the minima is shown in the legend). As expected, at lower message sizes, the overhead in message injection is a dominant concern and trees allow the overhead to be spread out across all the nodes rather than at a single node. However, the extra bandwidth costs associated with sending the data multiple times with the trees make them impractical for larger message sizes. The

Name	Term	Value
Latency	L	2.0 μs
Overhead	o	0.5 μs
Gap	g	2.0 μs
Inverse Bandwidth	G	1.0×10^{-3} $\mu s/byte$
Number of Nodes	N	256
Threads Per Node	t	4

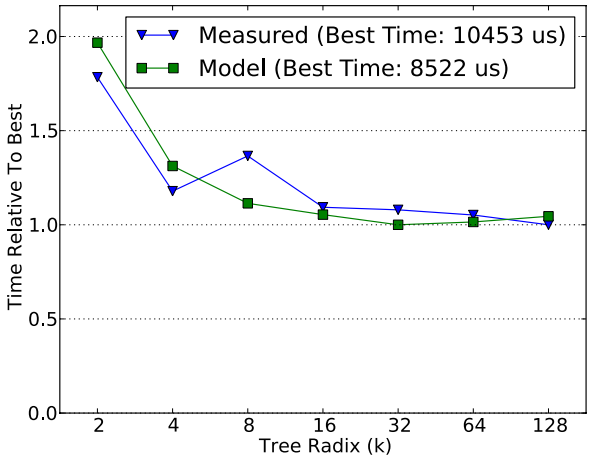
Table 5.1: LogGP Measurements for Sun Constellation



(a) 8 Byte Scatter



(b) 128 Byte Scatter



(c) 8 kByte Scatter

Figure 5.37: Scatter Model Validation (1024 cores of the Sun Constellation)

performance models are able to accurately predict these trends and thus will be a useful tool in mapping the search space.

While the models do well at predicting the trends they under-estimate the time and do not precisely predict the performance ratios implying that there are missing components to our model. Modern systems have a lot of features and components and thus, in order to make the model more accurate, these components need to be taken into account. Using system specific properties make the model less portable and therefore less useful to be incorporated into an automatic tuner. Thus we argue for simpler performance models and combined with limited search will yield the most portable code and performance models.

5.3.2 Gather

Logically Gather is the inverse of Scatter so the messages that flow down the tree in Scatter will flow up the tree in Gather. However, the LogGP model we describe above is insensitive as to the direction of the messages and therefore we can apply the same performance from Scatter to Gather. There are many factors that make the exact performance different based on how the messages can be overlapped and how to handle multiple messages arriving simultaneously.

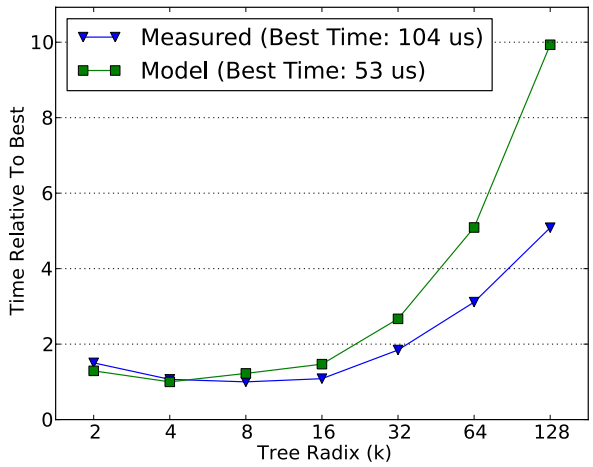
Figures 5.38(a)- 5.38(c) show the performance of the different trees with respect to their predicted performance. As the data show the Gather model does a good job of approximating the search space but is not as accurate as Scatter, especially at 8 kilobytes. The models above assume that there is no contention between the transfers. However, for Gather since the children simultaneously send to the root node, the different transfers could interfere with each other affecting overall bandwidth and thus lower performance. However, the performance models accurately predict the trend and show that the flatter trees are the best candidates for a Gather with a larger message size. Future work will explore making a better model for Gather that will more accurately predict the performance.

5.3.3 Broadcast

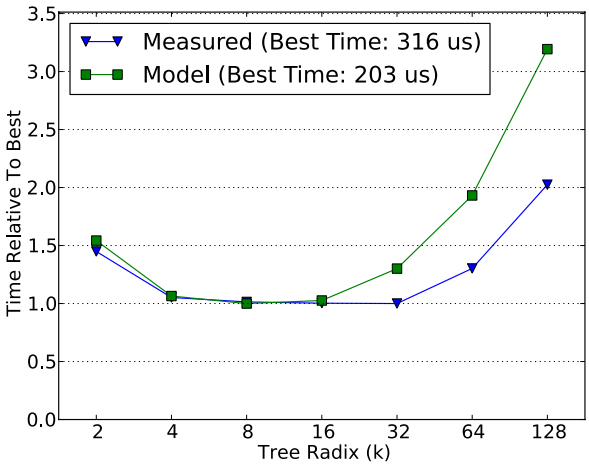
The performance model for Broadcast is a small variation of the Scatter performance model. We account for the latency in Broadcast in the same way we did for Scatter. Since there are no personalized messages for Broadcast, no matter the size of the subtree the bandwidth costs remain the same so the bandwidth per child can be expressed as: $\sum_{i=1}^{\lambda}(BG)$. Combining this with the latency term from above we can approximate the time for Broadcast as:

$$T_{broadcast} = \lceil log_k(N) \rceil [\underbrace{(L + o + kg)}_{latency} + \underbrace{kGB}_{bandwidth}]$$

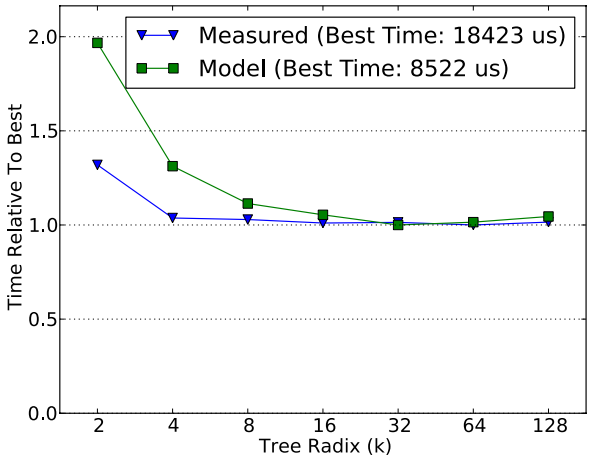
Figures 5.39(a)- 5.39(c) show the verification of the Broadcast model. Because the message sizes in Broadcast do not depend on the size of the subtree and personalized data is not replicated on the network, the best algorithms tend not to change as much with the message sizes. As the data also show the performance model is able to capture the trends and is able to effectively pick the best tree geometry. However, for larger message sizes,



(a) 8 Byte Gather

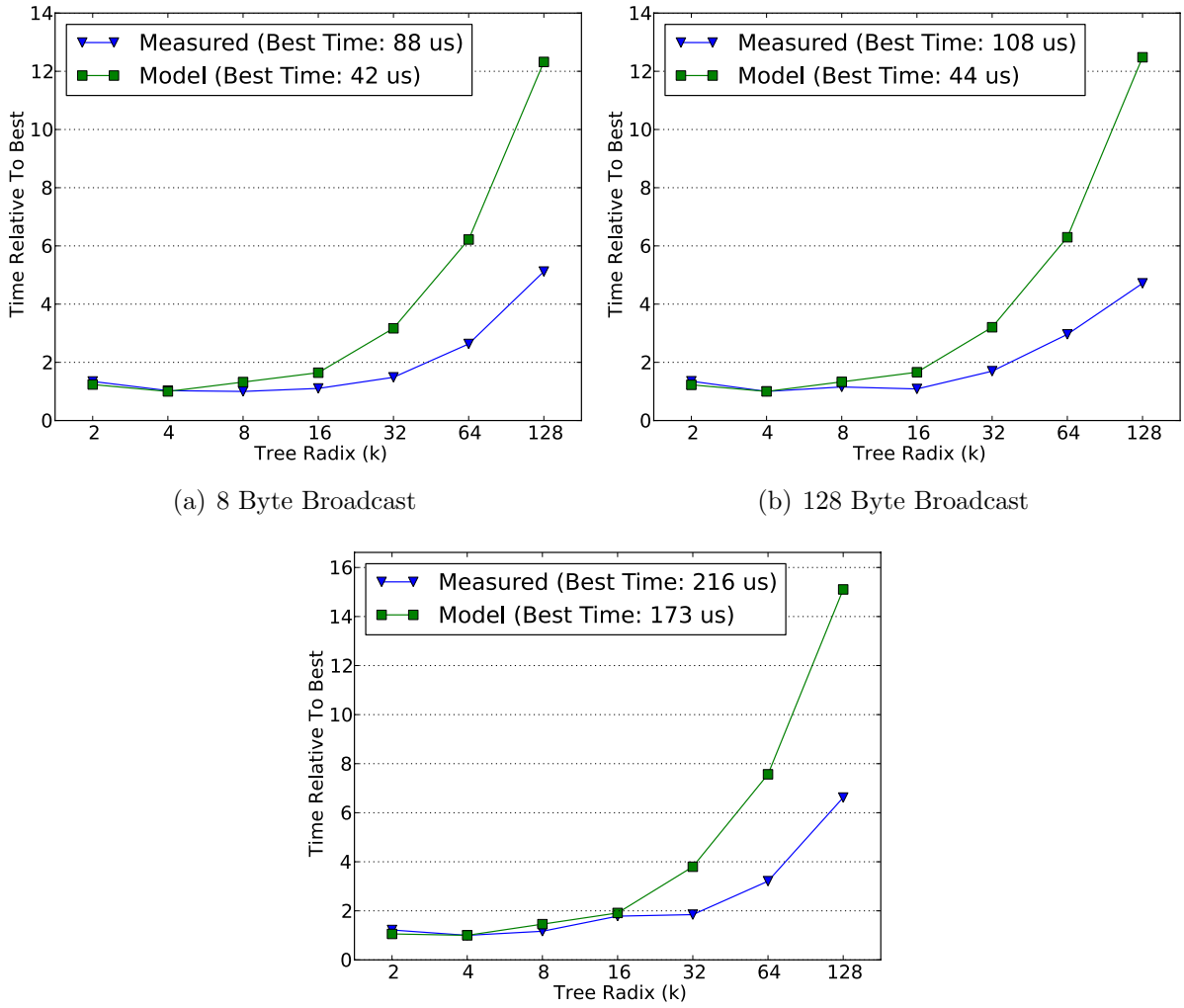


(b) 128 Byte Gather



(c) 8 kByte Gather

Figure 5.38: Gather Model Validation (1024 cores of the Sun Constellation)



(a) 8 Byte Broadcast

(b) 128 Byte Broadcast

(c) 8 kByte Broadcast

Figure 5.39: Broadcast Model Validation (1024 cores of the Sun Constellation)

the model over estimates the time yielding one to conclude that the time for flatter trees is worse than the actual measurements. However, the model does a good job of predicting the performance for the tree geometries that yield the best results. Thus, while the model is inaccurate for all the trees, it is accurate enough to be a useful tool for understanding the search space.

Since the size of the message in Broadcast does not depend on the subtree, leveraging the available parallelism for all message sizes yields performance advantages. However, as the data also shows, the main tradeoff, the added parallelism afforded by deeper trees needs to be compared against the added cost of the added latency of the extra levels. For the Sun Constellation the sweet spot appears to a radix 4 or radix 8 tree.

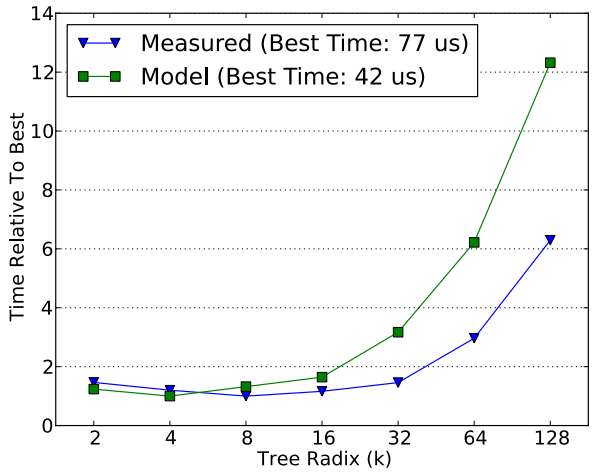
5.3.4 Reduce

Finally we analyze the performance for Reduce. We do not factor in the computation time for the following reasons: (1) for simple reduction operators such as addition the time is dominated by the network operations and (2) for more complicated operators the reduction performance greatly depends on the reduction operator and it is hard to represent the time for an arbitrary operation through a single parameter. From a pure communication standpoint, Reduce is the inverse of Broadcast and thus (using the same reasoning for Scatter and Gather) we can apply the Broadcast model to Reduce as well to yield a first approximation to the performance. Figures 5.40(a)- 5.40(c) show the verification of our model for Reduce. As the data show, the model does a good job of predicting the trends in performance however, the model tends to favor a 4-ary tree when the best performance best performance is an either 8-ary. The models say that either tree will be within 20% of each other. Choosing a 4-ary tree for the reduction will still yield a result that is within 20% of the best. Thus the model is accurate enough to fulfill its role in mapping the search space. Thus we also to complement these models with some form of search to ensure the best algorithms are chosen. For larger reductions the model overestimates the performance at larger tree radices leading one to believe they will yield worse performance than the results show.

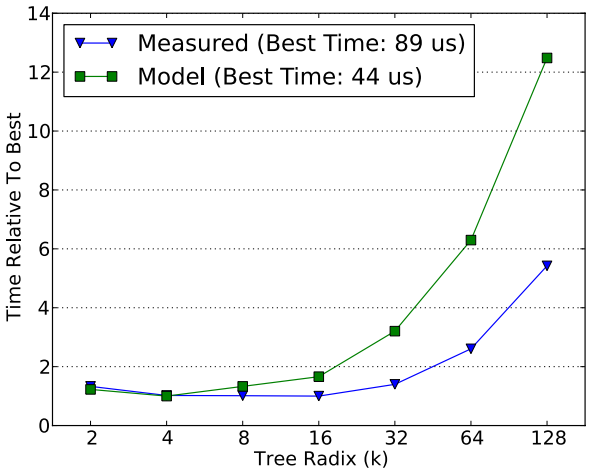
As with Broadcast, the size of the messages sent up to the parent do not depend on the size of the subtree, thus there is no penalty associated with using trees for the reductions. For Reduce, trees allow the network overhead to be parallelized throughout the network as well as the reduction operators. As with Broadcast, a radix 8 tree seems to yield the best universal performance.

5.4 Application Examples

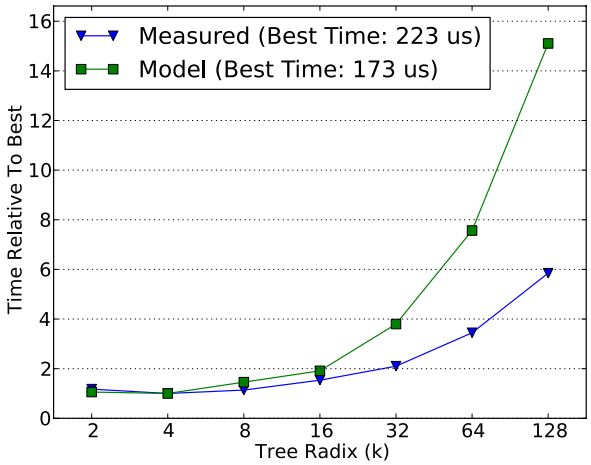
In order to motivate the use of the collectives outside of the microbenchmarks we have rewritten two popular and important mathematical kernels using the optimized nonblocking broadcast outlined in this chapter. We first explore a Dense Matrix Multiplication and then discuss a Dense Cholesky Factorization. Both kernels require that the broadcast be



(a) 8 Byte Reduce



(b) 128 Byte Reduce



(c) 8 kByte Reduce

Figure 5.40: Reduce Model Validation (1024 cores of the Sun Constellation)

performed on a subset of the processors. Chapter 9 goes into much more detailed discussion on the creation of these subset teams.

5.4.1 Dense Matrix Multiplication

One of the most commonly used computational kernels in large scale parallel applications is dense matrix multiplication. In this kernel we perform the operation $C = A \times B$ where A, B, and C are dense matrices of sizes $M \times P$, $P \times N$, and $M \times N$, respectively. In order to effectively parallelize the problem, each of these matrices must be partitioned across the processors.

Figure 5.41 shows an example of the outer product. In the figure the pieces of the matrix are color coded by the processor that owns the piece of the matrix. We can compute a particular block, $C[i][j]$ by performing the operation:

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

Notice that for all blocks in a given row i we need only broadcast the elements of $A[i][k]$ once and store into a temporary array. The subset of the processors that own row i has size $O(\sqrt{T})$, where T is the total number of UPC threads. Next we need to perform a column broadcast of $B[k][j]$ into a separate scratch array. Again notice that column broadcasts occur over a set of $O(\sqrt{T})$ processors in the column dimension. This a blocked implementation of the SUMMA algorithm [161]. 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. Thus we are able to leverage the nonblocking collectives found in GASNet (and hence UPC) to yield communication/computation overlap with a kernel that relies on collective communication.

The data from Figure 5.42 shows the performance of a weakly scaled matrix multiplication (i.e. fixed number of points per node) on the Cray XT4. Our UPC implementation has been written completely from scratch with using the collectives. Our experiments were conducted with one UPC thread per compute node with the local `DGEMM` operations performed through calls to the multi-threaded Cray Scientific Library. UPC is only used to manage the communication amongst the nodes and the math library is allowed to use all four cores to perform the matrix multiplies. The number of nodes in our experiment is always a perfect square so that we can have a square processor grid. 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 when the code leverages the team collectives. In all cases the UPC code significantly outperforms the comparable MPI/PBLAS version. As the data show, effectively using nonblocking communication can lead to good performance improvements for a code that is considered to be dominated by the floating point performance. As the data also show there are still some performance gains possible since the algorithms still do not achieve the peak performance.

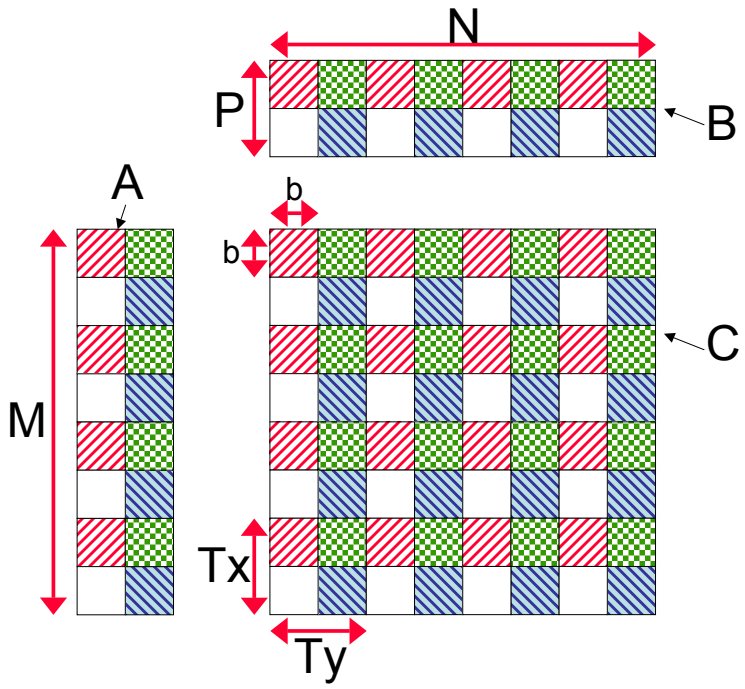


Figure 5.41: Matrix Multiply Diagram

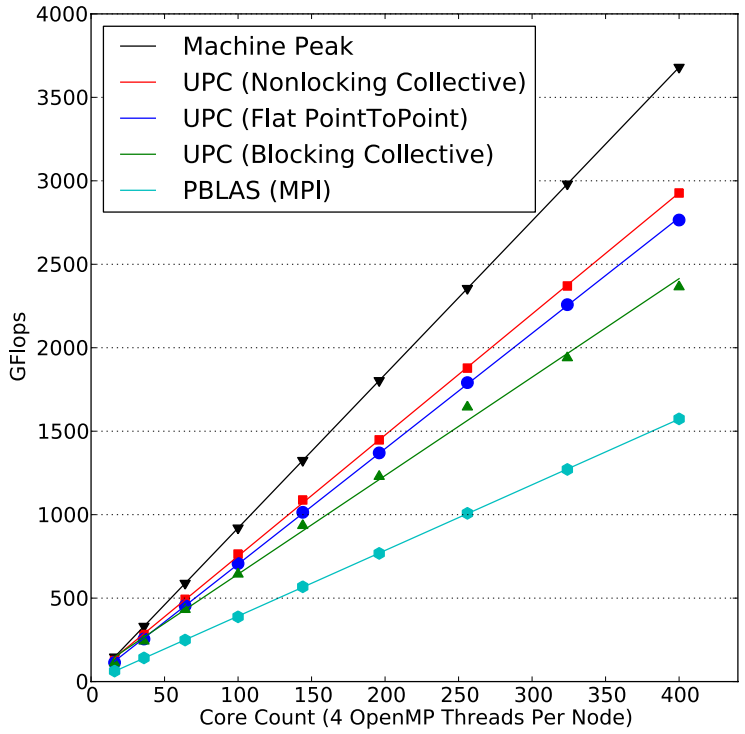


Figure 5.42: Weak Scaling of Matrix Multiplication (Cray XT4)

Better collective algorithms and BLAS library performance will allow us to get a higher fraction of the machine peak.

5.4.2 Dense Cholesky Factorization

In addition to being an important kernel in itself, the Matrix Multiplication is considered the rate limiting step in other dense factorization methods such as the Dense LU decomposition ($A = LU$) or the Dense Cholesky factorization ($A = U^T U$) for a square $M \times M$ matrix A. As shown in Figure 5.43, the popular implementations of the parallel factorization methods of these operations[52, 127] break the matrix 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}). The methods perform serial computation on the upper right corner and update 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. [4] The algorithm continues by recursively factoring the lower right quadrant. Hence 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 amongst a few processors. In order to alleviate this problem, a checkerboard layout[98] of processors is used so that the load is more evenly distributed across the processors. 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. The outer product is implemented with the algorithm from Section 5.4.1, including two rounds of broadcast operations.

Figure 5.44 shows the performance of our Cholesky algorithm compared to the distributed memory implemented in IBM ESSL's [76] ScaLAPACK. Our UPC implementation has been written completely from scratch with using the collectives as a goal. As before, we use one UPC thread per node and use a multithreaded ESSL library to perform the local BLAS operations. We allow ESSL to allow use all 4 cores per node and only use UPC to manage the communication amongst the node. The problem size was also weakly scaled with the number of nodes to keep the memory usage per node roughly constant and still ensure that we use a square problem size. As the data show our implementation of the Cholesky factorization scales as IBM's implementation and at 2k cores slightly out performs it. In addition, as expected, the naïve version of the code that uses point-to-point operations yields significantly lower performance. From the data we can conclude that applying the collective communication discussed in this chapter is essential for delivering good performance and algorithms that are comparable with the current state of the art. As the data for both ScaLAPACK and our implementation show, there is still a large difference between machine peak and the achieved performance. Further improvements in the collective algorithms, load

[4] Since the Cholesky Factorization is only applicable for Symmetric and Positive definite matrices, $A_{LL}^T = A_{UR}$ Thus we do not explicitly compute A_{LL} and simply take the outer product of $A_{UR}^T \times A_{UR}$. However for other Factorization algorithms such as LU, both need to be explicitly computed.

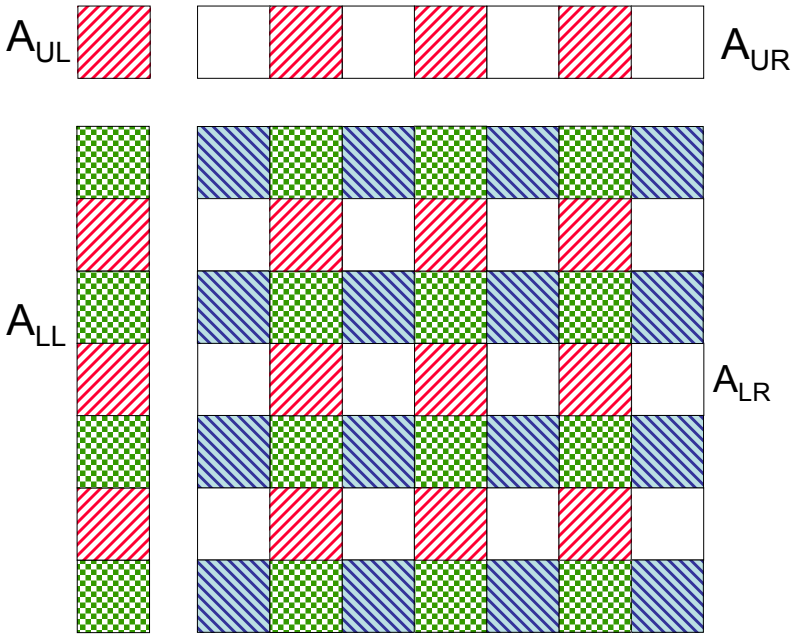


Figure 5.43: Factorization Diagram

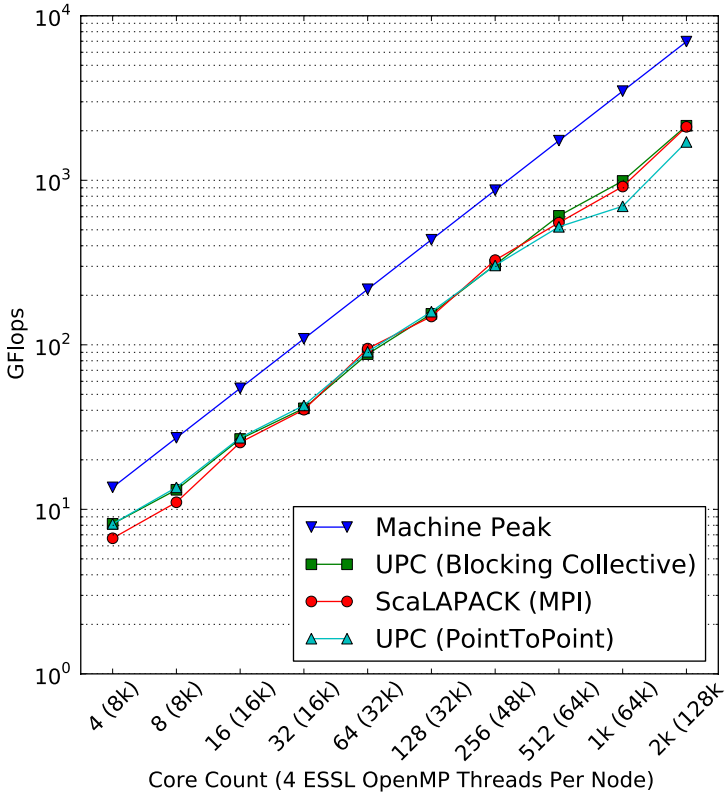


Figure 5.44: Weak Scaling of Dense Cholesky Factorization (IBM BlueGene/P)

	Cray XT4		Cray XT5		Sun Constellation	
Collective	Median	Max	Median	Max	Median	Max
Broadcast (Section 5.1)	22.0%	28.0%	34.0%	71.0%	7.0%	27.0%
Scatter (Section 5.2.2)	-3.0%	28.0%	21.0%	27.0%	14.0%	28.0%
Gather (Section 5.2.3)	-7.0%	53.0%	1.0%	44.0%	15.0%	45.0%
Reduce (Section 5.2.4)	55.0%	65.0%	-5.0%	4.0%	33.0%	42.0%

Table 5.2: Summary of Speedups over MPI for 8 byte through 2kByte Rooted Collectives

balancing, and serial performance can help parallel effeciency.

5.5 Summary

The main focus of this chapter was to outline the implementations and tuning considerations for rooted collectives (i.e. Broadcast, Scatter, Gather and Reduce). The techniques we focus on were leveraging shared memory when available, creating scalable communication schedules through trees, different data transfer mechanisms that the one-sided communication models offer, and showing how the collectives can be made nonblocking so that computation can be overlapped with the communication. We then apply these techniques to the other rooted collectives and show similar performance improvement. The performance results consistently showed that the collectives implemented through the one-sided communication model consistently yielded good performance compared to their MPI counterparts in both microbenchmarks and our application case studies. Table 5.2 shows a summary of our speedup results for the latency of various collectives. The median and best speedups over MPI are reported over a range of 8 bytes through 2 kBytes. A negative value indicates a slowdown compared to MPI. As the data show, our best speedups yeild a 71% improvement in the Broadcast latency (i.e. a speedup of 3.4×) over MPI by leveraging the one-sided communication. From the results for the other collectives also show taht GASNet was able to consistenly outperform MPI for a variety of platforms.

In order to understand the performance tradeoffs associated with the various implementation options we create simple performance models based on the LogGP framework that allow us to analyze what are the important parts of the collective. As we will further discuss in Chapter 8, the main goal of the models is to outline the search space and provide a guide to how to search the various implementation choices and thus as long as the models capture the trends it is good enough for the automatic tuner.

We then applied our collectives to two important application kernels, a Dense Matrix Matrix Multiplication and a Dense Cholesky Factorization. Using 400 cores of the Cray XT4 for DGEMM, we see an 86% improvement in performance over MPI by leveraging the GASNet collectives and communication/computation overlap. On 2k cores of the IBM BlueGene/P our results showed that the algorithms written using the GASNet collectives deliver about the same parallel effeciency as the vendor optimized ScaLAPACK library. From the

microbenchmark and application results we argue that the one-sided communication model found in GASNet is a good choice for implementing high performance and scalable collective communciation for the rooted collectives.

Chapter 6

Non-Rooted Collectives for Distributed Memory

In the previous chapter the focus was on optimizing collectives that specify one thread as a root thread that is the source or target of the communication. The other important class of collective algorithms are the ones in which every thread sends or receives data from all the other threads. These algorithms are inherently more communication intensive since every thread has information that it wishes to communicate with every other thread rather than with a single root thread. If there are T threads the rooted collectives would induce $O(T)$ messages in the network while the non-rooted collectives can induce up to $O(T^2)$. As we will show, we can construct algorithms that perform the same collective with $O(T\ log\ T)$ messages however, as is the case with the trees, this will require the to data be sent through intermediaries and thus potentially consume more bandwidth. Like the previous chapter we will construct performance models to understand the performance tradeoffs and enable the automatic tuner to make a decision on what class of algorithms to use. At the end of the chapter we show how a bandwidth limited application kernel, the 3D FFT, can realize significant performance improvements by leveraging the optimizations discussed in this chapter.

As we will demonstrage the one-sided communication model in GASNet is able to provide upto a 23% improvement in the performance of Exchange on 256 cores of the sun constellation and a 69% improvement in performance for Gather-to-All on 1536 cores of the Cray XT5. By leveraging communication/computation overlap through the nonblocking collectives found in GASNet we are able to deliver 2.98 Teraflops for a communication dominated benchmark, the 3D FFT, on 32,768 processors on the IBM BlueGene/P (a 14% improvement) and 46% improvement in performance for the same benchmark on 1024 cores of the Cray XT4.

6.1 Exchange

In an Exchange, each thread contains a personalized message for every other thread leading to a $O(T^2)$ dependence pattern since every thread depends on information from every

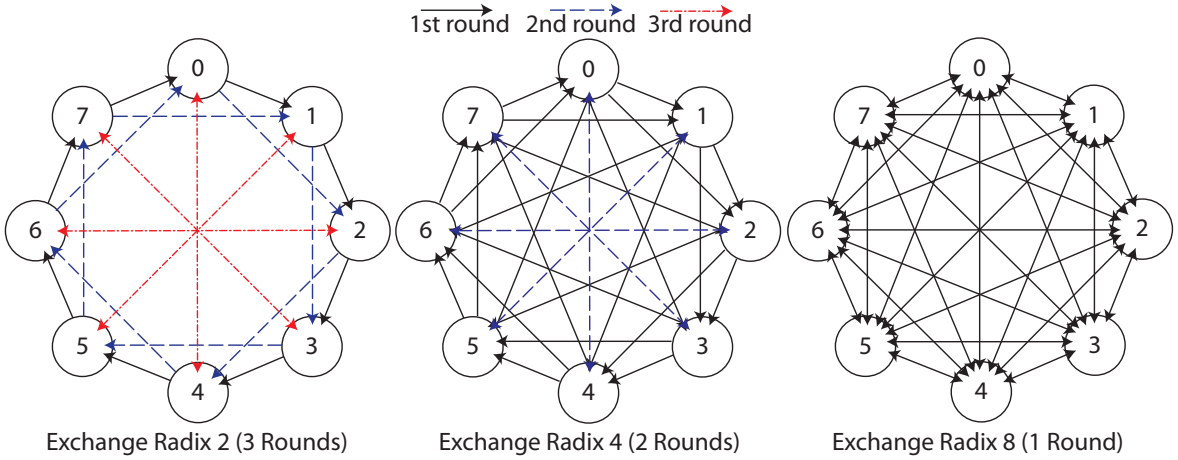


Figure 6.1: Example Exchange Communication Pattern

other thread. However for small message sizes having $O(T^2)$ messages for the collective can be quite wasteful and the algorithm can be significantly improved by routing data through intermediaries to limit the number of messages.

From our results in Section 5.1.1 we observe that choosing a representative thread within a node yields significant performance improvements thus we use a similar strategy for the non-rooted collectives as well. For each node (a collection of threads that share the same shared memory domain), the representative thread will pack the data on behalf of all the t threads it is responsible for and will manage the communication amongst the threads including all the unpacking and unpacking that is needed. For Exchange each thread has Nt blocks of B bytes of data and thus there are Nt^2 blocks in the entire node of which only t^2 blocks are destined for threads within a single node. Henceforth we thus assume the communication is only amongst the N nodes involved in the collective.

The algorithms for Exchange that we examine can be broken up into the following two categories:

- **Flat Algorithms:** In this approach every node packs the data for all the threads within that node that is destined to a remote node and then sends the message directly. Thus there are $O(N^2)$ messages in the network of $O(Bt^2)$ bytes per message, but the data is sent exactly once without intermediaries.

- **Dissemination Algorithms:** The second class of algorithms perform the same operations in $O(N\ log\ N)$ steps by using Bruck's algorithm[44], in which data is sent to its final destination node through intermediaries. Since the same data is sent more than once, this approach requires more bandwidth. Thus we tradeoff reduced message count for additional bandwidth. The most common 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 6.1 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 induces the same number of messages as the Flat Algorithm. At each round, the message size can is $O(\frac{BNt^2}{k})$ where B is the size individualized messages between the threads for Exchange.

Figure 6.2 shows the performance of the different Exchange algorithms on 256 cores Sun Constellation. As the data show at small message sizes the Dissemination algorithms yield the best performance by reducing the overall message count. As the data also show, at small message sizes the Radix 2 and Radix 4 algorithms yield the best performance. Starting at a transfer size of 32 bytes, the radix 8 dissemination algorithm yields the best performance. This thus indicates that there is a tradeoff between the extra bandwidth consumed by sending the data multiple times through intermediaries and the savings in the number of messages. As the data show at 128 bytes the radix 8 Dissemination Exchange outperforms the radix 2 version by 35%. Around 512 bytes the Flat algorithm yields the best performance indicating that the extra bandwidth associated with sending the data multiple times through the network becomes the dominant factor. Section 6.1.1 will go into much more detail about creating a performance model that can accurately capture this tradeoff.

Figures 6.3 and 6.4 show the performance of the different Exchange algorithms on the Cray XT4 and Cray XT5 respectively. As the data show on the XT4 the radix 4 and radix 8 yield the best performance up to 1kBytes where the Flat algorithm dominates. However on the Cray XT5, the Flat algorithm is always the best performer. Since the XT5 has 12 threads per node, each of the representative threads must manage the communication or 12 threads and thus the bandwidth requirements for the algorithms that forward the data are significantly higher (by a factor of 9). This, combined with the network characteristics, makes the Flat algorithm yield the best performance for all message sizes. As the data also show, the MPI performance is consistently better than ours for this algorithm for these two platforms. Our hypothesis is that MPI is using algorithms that are better suited to the 3D torus and mesh found on these networks. Future work will add algorithms to the tuning infrastructure that are better suited towards 3D torus and mesh networks.

6.1.1 Performance Model

As the data show there are clear performance tradeoffs between the different algorithms. We tradeoff the latency and message injection overhead advantages of sending the data to a fewer number of peers against the excess bandwidth of sending the data through multiple intermediaries. In line with our hypothesis, the data show that the dissemination algorithms yield the best performance at low message sizes, however at larger message sizes the flat algorithms perform well. Thus it is important to understand the tradeoffs and predict the best algorithms at the different sizes. Thus, as in Section 5.3 we create performance models using the LogGP framework to analyze the performance. As before, the most important goal of the model is to get relative performance between the algorithms correct. To be able to intelligent decisions about pruning the search space, the absolute performance is not essential.

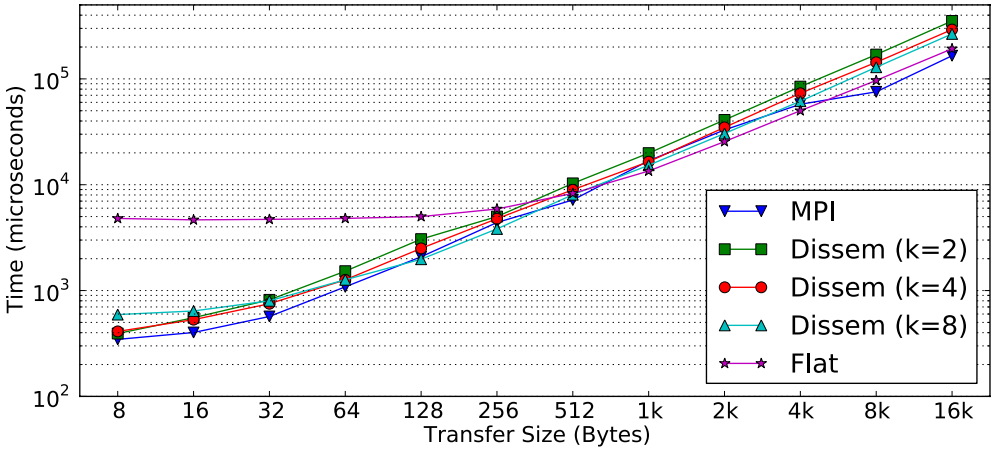


Figure 6.2: Comparison of Exchange Algorithms (256 cores of Sun Constellation)

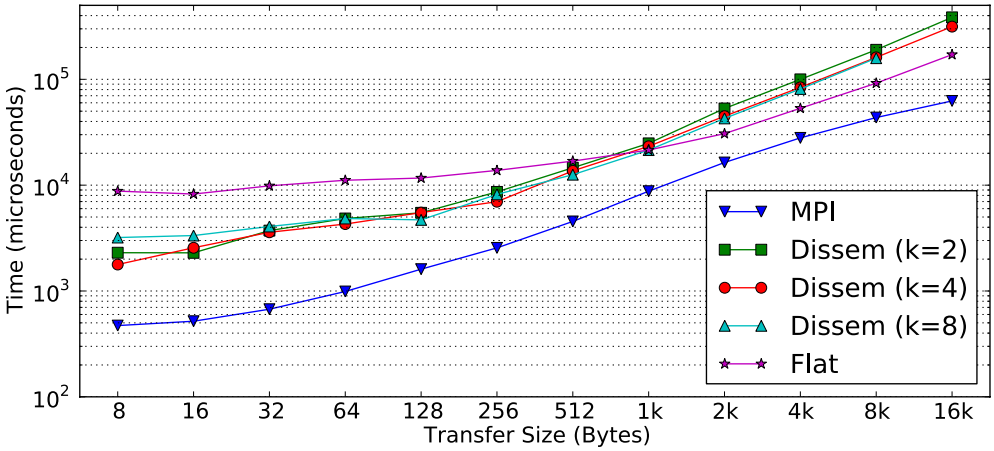


Figure 6.3: Comparison of Exchange Algorithms (512 cores of the Cray XT4)

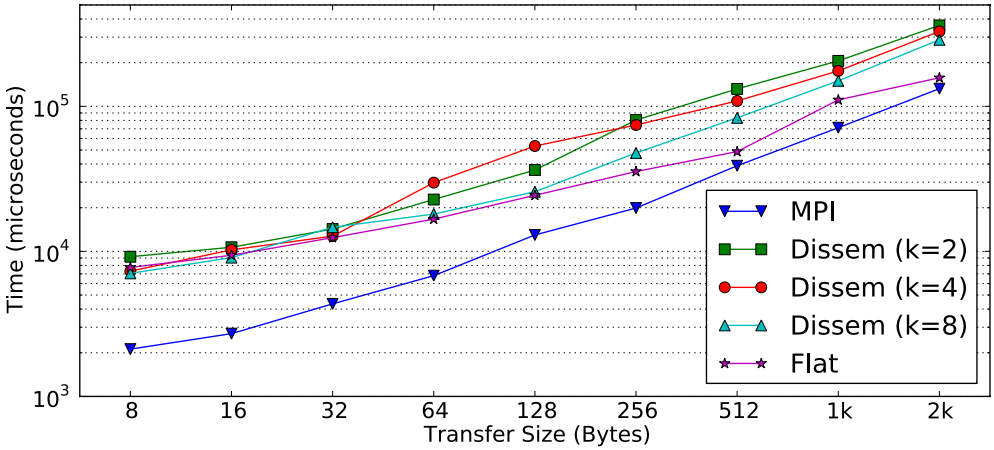


Figure 6.4: Comparison of Exchange Algorithms (1536 cores of Cray XT5)

Given N nodes with t threads per node, there are $\lambda = \lceil log_k\ N \rceil$ rounds of communication. In all, but the last round, each peer will send data to and receive data from $k-1$ peers. In the last round there are $(N/k^{\lambda-1}) - 1$ peers. For simplicity we assume that all the nodes march through the algorithm in lock step. Thus in each of the λ rounds there is a cost of L to receive the first byte of data from the first peer on the network and then a cost of o per peer to receive the data from each of the peers. We are thus assuming that the messages are perfectly pipelined behind each other. For each of the peers a node sends data to it incurs a cost of g to inject the message into then network. Each of the t threads has Nt bytes it wants to exchange. Thus each node has $B \times t^2$ bytes that it needs to send to each of the other remote nodes regardless of however many intermediaries it passes through. In the Bruck's exchange algorithm, a node sends N/k of its data to each of the peers. Thus the entire Bandwidth term to each peer can be summarized as $\frac{GBNt^2}{k}$. Putting the various pieces together we get the following model for the execution time:

$$Time_{Exchange} = \sum_{i=0}^{\lambda} [L + (MIN(k, \frac{N}{k^i}) - 1) \times (o + g + \underbrace{\frac{GBNt^2}{k}}_{Bandwidth})]$$

Figures 6.5, 6.6, and 6.7 show the verification of the performance model on 1024 cores of the Sun Constellation for 8 bytes, 64 bytes, and 1kbytes respectively. The performance results for both 16 threads per node and 4 threads per node are shown. The times have been normalized to show the time relative to the best for each category since there is a dramatic variance in the runtime of the different algorithms on different numbers of threads per node.

As the data show, at 8 bytes the optimal dissemination radix is 8. The model however, picks a radix of 8 which would lead to an algorithm that takes 30% longer. However, if automatic tuner were to use the model to find the top two results then search amongst them, the tuner would find the best algorithm. Notice that model correctly predicts that the Flat tree is a bad choice and thus the tuner would avoid searching over an obviously bad algorithm. When there are 16 threads per node, then the model correctly predicts that the Flat Tree is a viable algorithmic choice that must be considered alongside a radix 8 Dissemination algorithm. Notice that the bandwidth parameter scales as t^2 thus there is a 16× increase in the bandwidth requirements for all the algorithms. This rise makes the flat algorithms, which minimize the number of times the messages are transmitted on the network, viable candidates which the model accurately predicts. For 4 threads per node at 64 bytes the optimal dissemination radix is 8 which the model correctly picks the winner. The model is able to choose the top two performers but incorrectly over penalizes the radix 2 Dissemination algorithm and under penalizes the flat algorithm. At 1024 bytes the models for both 4 and 16 threads per node accurately predict that the Flat algorithm should be the winner.

As the data show, the model does not perfectly sort the data but is able to highlight the algorithms that are potentially good candidates. Thus, using this model, we can avoid wasting valuable time searching over obviously slow candidate algorithms.

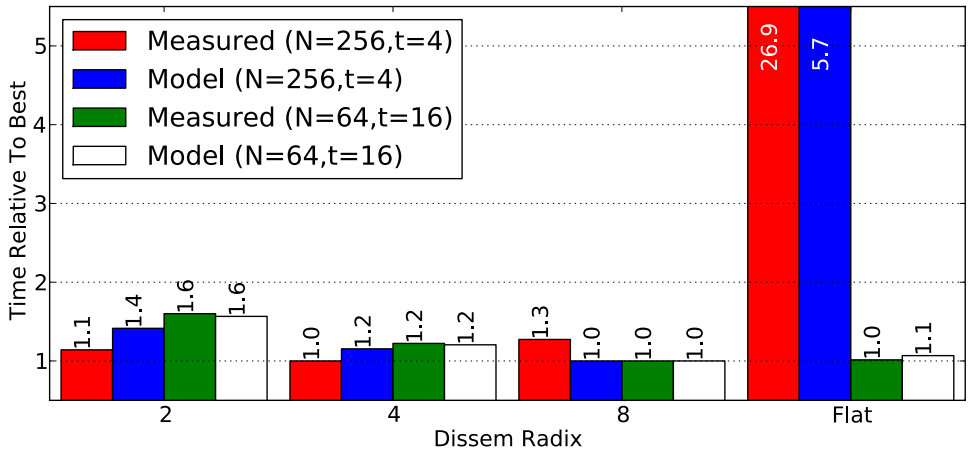


Figure 6.5: 8 Byte Exchange Model Verification (1024 cores of the Sun Constellation)

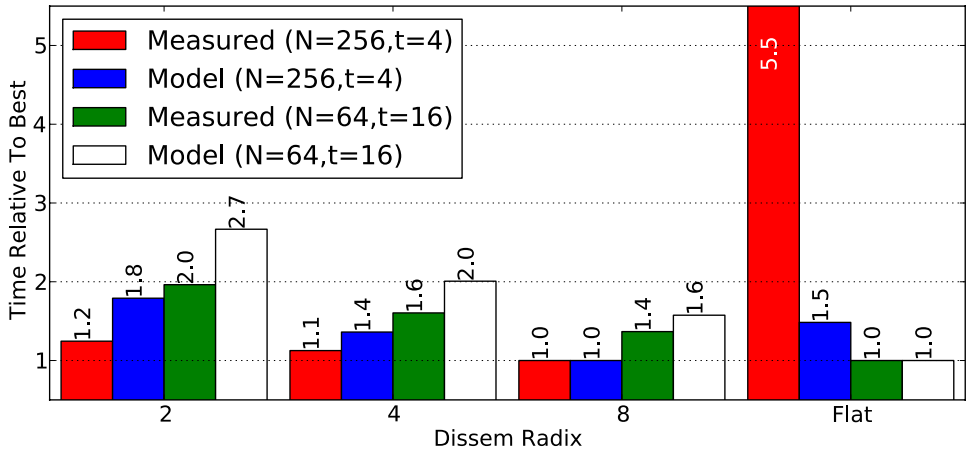


Figure 6.6: 64 Byte Exchange Model Verification (1024 cores of the Sun Constellation)

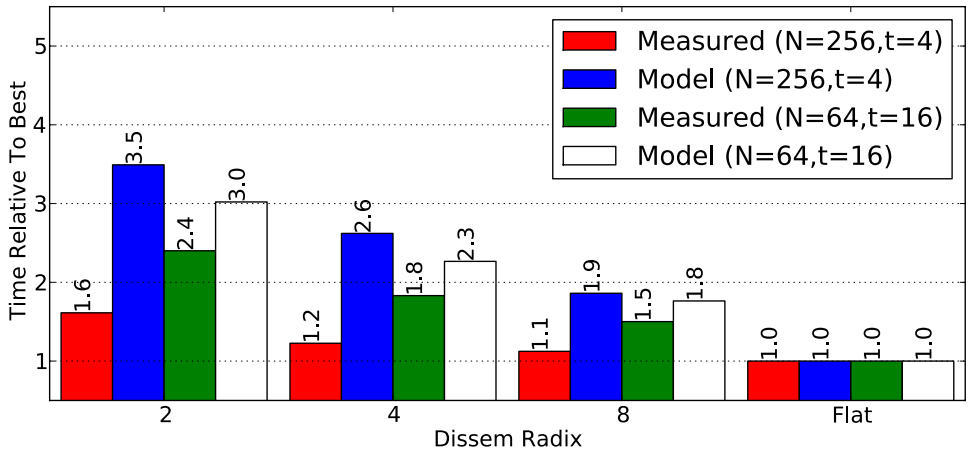


Figure 6.7: 1 kByte Exchange Model Verification (1024 cores of the Sun Constellation)

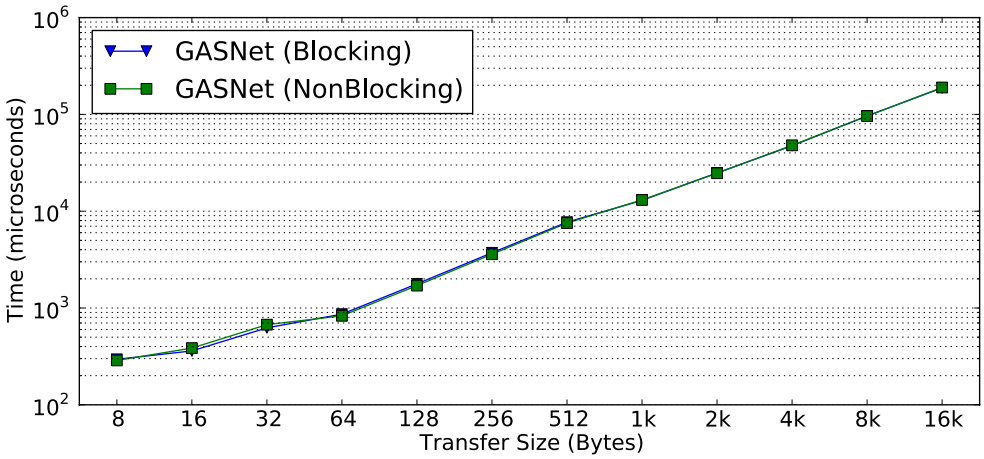


Figure 6.8: Nonblocking Exchange (256 cores of Sun Constellation)

6.1.2 Nonblocking Collective Performance

Figure 6.8 shows the performance of initiating multiple Exchanges back-to-back and then synchronizing them on 256 cores of the Sun Constellation. For the rooted collectives, the nonblocking collectives realized significant performance advantages by allowing the collectives to be pipelined each other. However, for Exchange, there is little speedup by allowing multiple collectives to be in flight simultaneously. By using trees on rooted collectives, different levels of the tree could be working on different collectives ensuring that all nodes were active, thereby increasing the throughput of all the collectives. With blocking collectives, the nodes were only active when the data was at their level of the tree. However, for Exchange all nodes are actively involved in communication every round. Thus, even though we have initiated multiple collectives, each node is active through all stages of the collective and thus the performance gains from nonblocking collectives because the communication pattern forces the collectives to be serialized. However, as we will show in Section 6.3 when the Exchange can be overlapped with computation rather than other collectives, one can realize significantly better performance.

In a radix-2 Dissemination algorithm, $O(BNt^2/2)$ bytes are sent at every round. Thus even for small message sizes the bandwidth component is a significant part of the overall runtime especially for large node counts and platforms with many cores per node. Thus even by exposing multiple transfer opportunities to the system, these messages must get serialized because of network bandwidth limits. This further limits good communication-communication overlap amongst the collectives. As we will show in Section 6.2.2 this is a restriction that is specific to Exchange and not necessarily all rooted collectives.

6.2 Gather-to-All

Unlike Exchange in which every thread has a personalized message for every other thread, every thread in Gather-to-All broadcasts the same data to every other thread. Logically

this can be implemented by t simultaneous Broadcasts, where each of the t broadcasts is rooted at a different thread. However, this will yield a suboptimal implementation since the communication schedules will collide unless one carefully controls the tree shapes and when the data can be sent. Like Exchange, we consider the following two variants of the algorithm:

- **Flat Algorithms:** In this approach every node packs the data for all the threads within that node that is destined to a remote node and then sends the message directly. Thus there are $O(N^2)$ messages of length $t \times B$ bytes where t is the number of threads and B is the number of bytes each thread wishes to Broadcast.
- **Dissemination Algorithms:** The dissemination algorithm works by doubling the message size every round. Thus in the first stage each node exchanges data with one other node and it only has the data for all the threads within that node. However once the first exchange of data is completed, both nodes now have information about two nodes worth of data. And thus with $log_2 N$ stages the operation can be completed. Our implementation follows the one presented by Bruck et al. [44]. As we will demonstrate any radix higher than 2 will likely yield suboptimal performance and hence we only consider radix 2 dissemination algorithms.

A major difference between the dissemination algorithm for Exchange and Gather-to-All is the message size at each round. For example, a radix 2 Dissemination Exchange required that $O(BNt^2/2)$ bytes per round where the radix-2 Gather-to-All will only require $O(Bt \times 2^i)$ bytes at round i. Notice that in the first round ($i = 0$) there is a factor of $Nt/2$ difference in the message sizes between both collectives and in the final round there is a factor t difference in the message sizes. Thus the Gather-to-All, which still requires every thread to communicate with every other, utilizes significantly less bandwidth than a comparable Exchange. This will affect how well the collectives can be overlapped amongst each other. We have modified the algorithms to take advantage of the one-sided communication and the loosened synchronization modes described in the previous Chapter.

Figures 6.9- 6.11 show the performance of Gather-to-All on Sun Constellation, Cray XT4, and Cray XT5 respectively. As the data show the Dissemination algorithms always outperform the Flat algorithms. Section 6.2.1 goes into more detail about why this is the case. In addition, as the data show on the Sun Constellation, the GASNet yields the same performance as MPI up to 8kBytes at which point MPI switches to a different algorithm. Related work by [109] shows a similar increase in performance for the same hardware platform. They attribute the increase in performance to switching to a Ring based Gather-to-All. In this algorithm every node is connected via by a virtual ring and will receive data from each neighbor. Future work will incorporate this algorithm to handle large Gather-to-Alls into GASNet. As the data also show on the Cray XT4 MPI outperforms GASNet consistently at lower message sizes. Our hypothesis is that MPI in this case is able to create a better communication schedule for the torus network. However on the Cray XT5 GASNet consistently outperforms MPI. On this platform, with 12 threads per node, GASNet is able to manage all local communication with memcpy operations rather than going through expensive OS level mechanisms to copy the data across process boundaries.

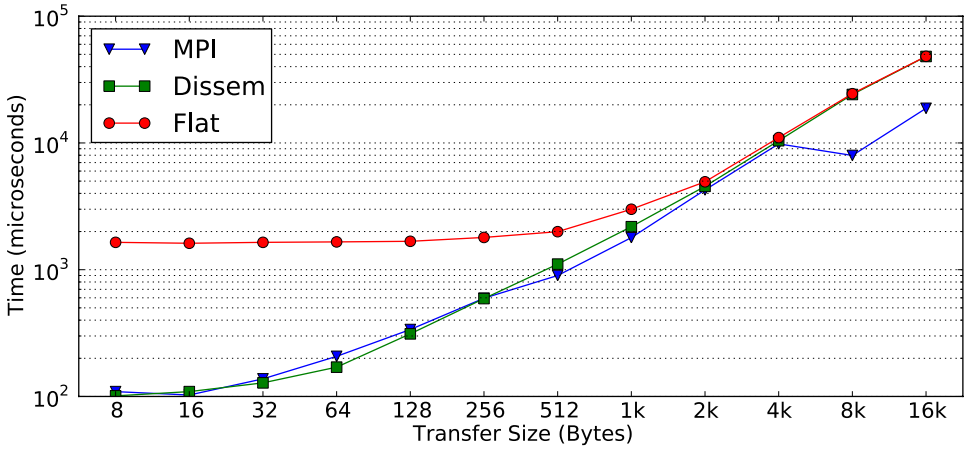


Figure 6.9: Comparison of Gather All Algorithms (256 cores of Sun Constellation)

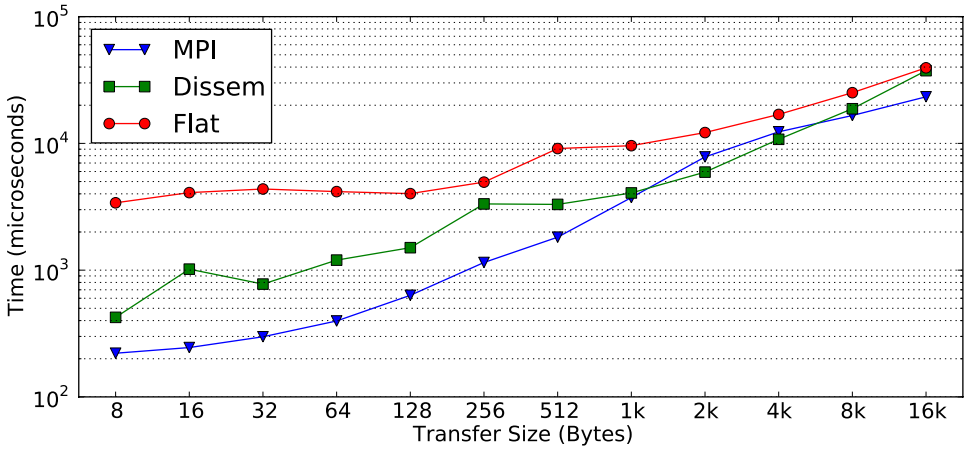


Figure 6.10: Comparison of Gather All Algorithms (512 cores of the Cray XT4)

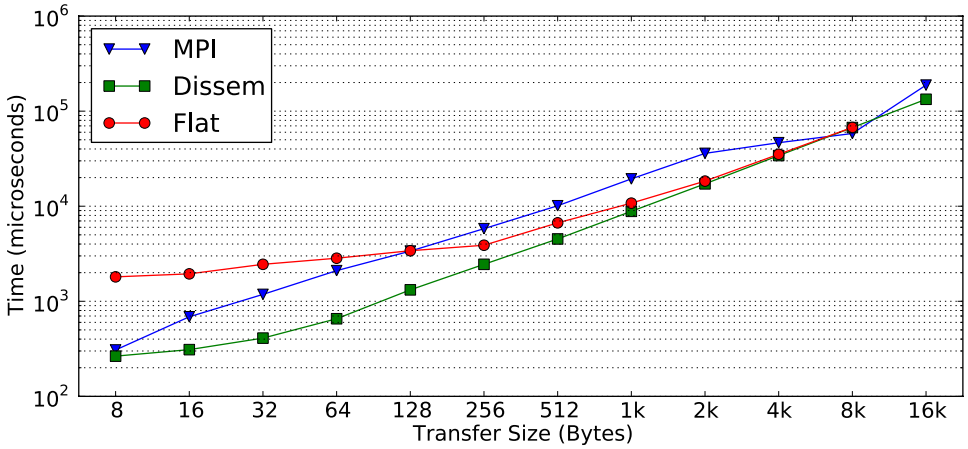


Figure 6.11: Comparison of Gather All Algorithms (1536 cores of Cray XT5)

6.2.1 Performance Model

As the data show the flat algorithm never yields the best performance, thus to prove to ourselves that it is always best to pick the dissemination approach we construct a performance model using the LogGP framework.

With N nodes there will be $\lambda = \lceil log_2 N \rceil$ stages. For each stage there we need to receive a message and simultaneously transmit a message. There will be a cost of o to receive message and g to send the message since we have to wait for the message from the previous round to clear. In addition there will be a cost of L for the message to travel across the network. Thus the total latency component can be calculated as $\lambda \times (L + o + g)$. To calculate the bandwidth component above we notice that at each stage the message size doubles leading to the following expression:

$$T_{Gather-to-All} = \sum_{i=0}^{\lambda} [L + o + g + GBt2^i]$$

Using the formula for a geometric series we can express the total time for Gather-to-All as:

$$T_{Dissemination} = \lambda \times (L + o + g) + GBNt$$

In the flat algorithm each node sends and receives Bt bytes from each of the other nodes leading to the following formula. Thus we can express the time with the following expression:

$$T_{Flat} = L + (N - 1) \times (o + g + GBt)$$

An interesting result of the model is that both algorithms incur similar bandwidth costs ($O(GBNt)$) for any reasonably large values of N. However, where the Dissemination algorithm uses $O(log_2 N)$ messages to send the data the Flat algorithm uses $O(N)$ messages to send the exact same data thus the Dissemination algorithm will be much better at lower message sizes when the overhead and latencies are the dominant concerns. For large message sizes both algorithms incur the same bandwidth costs. Thus the dissemination algorithm will almost always yield the best performance. The only case in which the Flat algorithm will yield better performance is if the L term is the dominant part of the model. The Flat algorithm only incurs one latency cost where the Dissemination algorithm incurs $O(log_2 N)$ latency costs. However for large values of N the cost of injecting $N - 1$ messages and managing those messages in the network will almost certainly be more expensive. Thus, as is consistent with the data, the Dissemination algorithm should and does outperform the Flat algorithms. Choosing a radix of 2 minimizes the number of messages required for the dissemination algorithm. Since sending the fewest number of messages yields the best performance a radix 2 our automatic tuner would always choose the radix 2 dissemination algorithm.

6.2.2 Nonblocking Collective Performance

Figure 6.12 shows the performance of initiating multiple Gather-to-Alls before waiting for all of them to finish. For both the blocking and nonblocking versions we use the radix-2

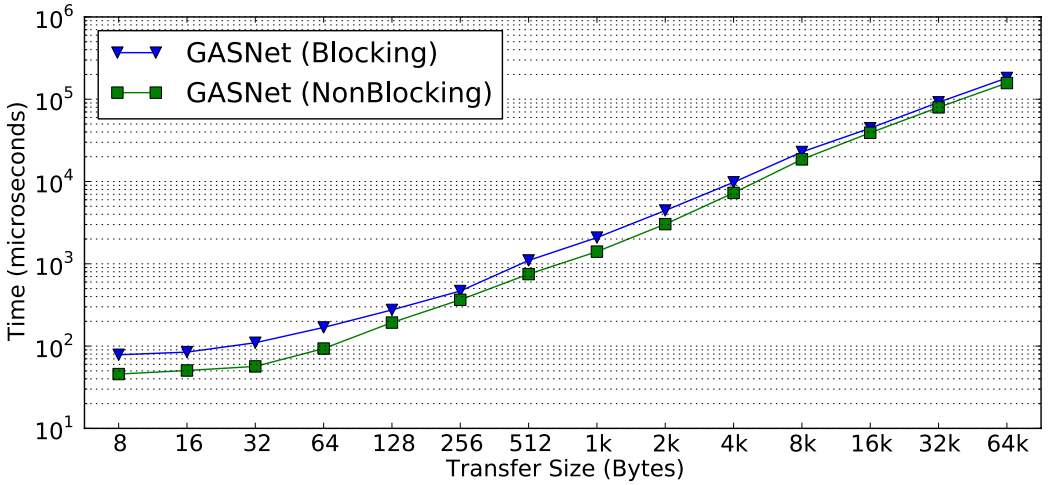


Figure 6.12: Nonblocking Gather-to-All (256 cores of Sun Constellation)

Dissemination algorithm. Thus many Gather-to-Alls are in flight simultaneously. As the data show there is a performance benefit at lower message sizes of being able to overlap the collectives amongst each other, which is a different result than Exchange. At lower message sizes the performance difference is much higher (48% at 32 bytes) than it is at larger message sizes (13% at 16 kBytes). Unlike Exchange, where the message size at each stage was fixed and large at each round, the message sizes for Gather-to-All vary. At lower message sizes the message sizes will be much smaller. For a radix-2 Dissemination Exchange half the final data size ($O(BNt^2/2)$ bytes) are exchanged in every round, where as for a radix-2 Dissemination Gather-to-All only ($O(BNt/2)$ bytes) are exchanged in only the final round. Notice that the message size also only grows by a factor of t rather than t^2. Thus, due to the lower bandwidth costs of Gather-to-All compared to Exchange, Gather-to-All is able to realize consistent speedups when the collectives are overlapped amongst each other.

6.3 Application Example: 3D FFT

To study the performance of the non-rooted collectives in an application setting we use the NAS Parallel Benchmark [19] FT. At the center 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 amongst all the threads and FFTs must be done in each of the dimensions. Depending on how the data is laid out, the prism might need to be transposed in order to re-localize the data to perform the FFTs. We go into further details on this operation later in this section. At large scale this transpose step is often the performance-limiting step since it stresses the bisection bandwidth of the network.

As we have shown in our previous work [28], GASNet (and hence Berkeley UPC) allow effective overlap of communication and computation, enabling large performance gains. In that work we show that we can effectively use hardware features found in modern networks,

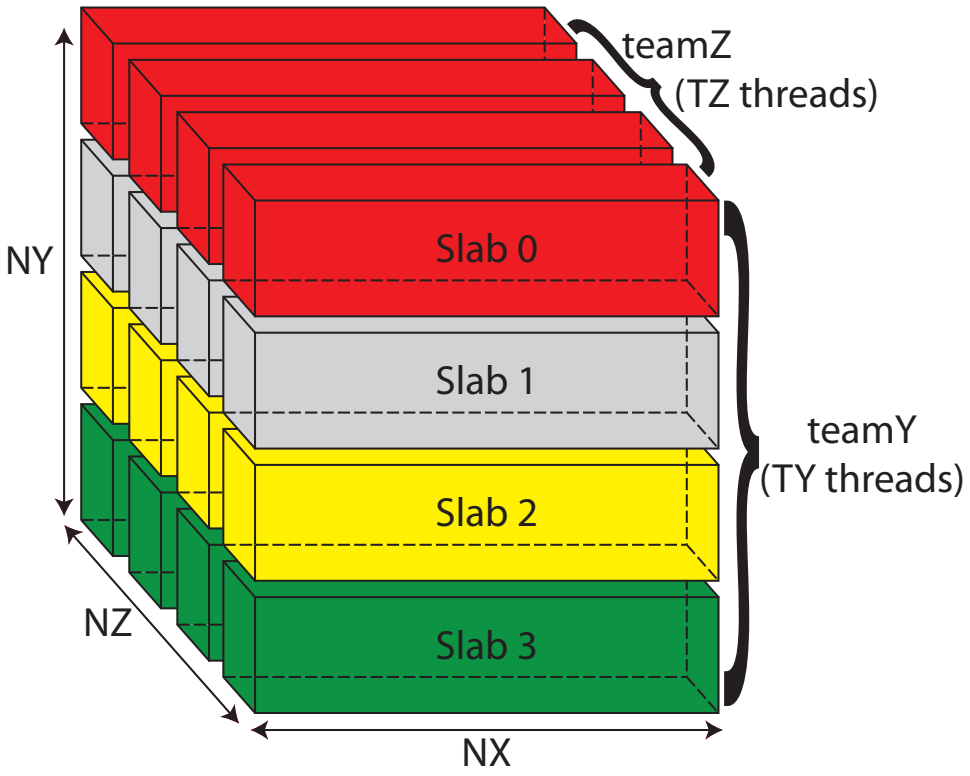


Figure 6.13: 2D decomposition for FFT

such as Remote Direct Memory Access (RDMA), to significantly improve performance for an application that is often considered a bisection bandwidth limited problem. In addition, we show that GASNet is a better semantic match to these hardware features than MPI. One of the most significant results is that breaking up the exchange collective (i.e. `MPI_Alltoall` in MPI parlance) into point-to-point operations that are overlapped with computation leads to significant performance improvements[1].

Our previous work only examined a 1-D decomposition of the problem domain across the threads, which limited the maximum number of threads to be the minimum of NX, NY, and NZ. On BG/P and other systems that use high degrees of parallelism to realize performance, this limitation prohibited scaling to core counts found in typical installations. Thus we have extended our previous work to handle a 2-D thread layout. Figure 6.13 shows the differences between the two thread layouts. In a one-dimensional thread layout a particular thread owns $\frac{NZ}{T}$ planes of $NX \times NY$ points where T is the total number of threads. In a two-dimensional thread layout a particular thread owns $\frac{NZ}{TZ}$ planes and $\frac{NY}{TY}$ rows of NX points where the T threads are laid out in a $TY \times TZ$ thread grid.

In a one-dimensional thread layout two of the three dimensions of the FFT are fully located on one thread thus only one transpose needs to be performed: the one that re-localizes the data in the Z-dimension. In a two-dimensional thread layout only one dimension of the grid is contiguous on a thread and thus two rounds of transposes need to be performed to complete one FFT. The first re-localizes the data to make the Y-dimension contiguous and thus the communication is performed amongst teams of threads in the TY dimension (*i.e.* all the threads within the same thread plane in Figure 6.13). The second one re-localizes the data to make the Z-dimension contiguous amongst teams of threads in the TZ dimension (*i.e.* all the threads within the same thread row).

6.3.1 Packed Slabs

Conventional wisdom suggests the best way to optimize this application is to perform the communication and the computation in two distinct stages, using what we shall refer to as the *Packed Slabs* algorithm. In the first stage this algorithm computes the FFTs for all the $\frac{NZ}{TZ} \times \frac{NY}{TY}$ rows that a thread owns across all the planes. The data is then packed and all the threads within the same thread plane exchange their data in one big communication step. After the first round of exchanges is finished the data is then unpacked and the next $\frac{NZ}{TZ} \times \frac{NX}{TY}$ rows of NY FFTs are performed. The data is then repacked and the final communication step is done. Once the communication is done, the data is unpacked and the final $\frac{NY}{TZ} \times \frac{NX}{TY}$ rows of NZ length FFTs can be performed. This algorithm is more fully detailed in Algorithm 6.14. The Packed Slabs algorithm uses packing to maximize message sizes, in order to achieve the best bandwidth performance for those transfers. However this approach sacrifices the ability to overlap the communication and computation – at any given time either the communication or computational subsystems will be sitting idle.

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

3DFFTPackedSlabs($TY, TZ, MYTHREAD$)

```
1  myPlane ← ⌊MYTHREAD/TY⌋
2  myRow ← MYTHREAD%TY
3  teamY ←  all threads who have same value of myPlane
4  teamZ ←  all threads who have same value of myRow
5  for plane ← 0 to NZ/TZ − 1
6  do for row ← 0 to NY/TY − 1
7      do  1D FFT of length NX
8   Pack the slabs together
9   Do Exchange on teamY
10  Unpack the slabs to make Y dimension contiguous
11 for plane ← 0 to NZ/TZ − 1
12 do for row ← 0 to NX/TY − 1
13     do  1D FFT of length NY
14  Pack the slabs together
15  Do Exchange on teamZ
16  Unpack the slabs to make the Z dimension contiguous
17 for plane ← 0 to NY/TZ − 1
18 do for row ← 0 to NX/TY − 1
19     do  do 1D FFT of length NZ
```

Figure 6.14: FFT Packed Slabs Algorithm

```
3DFFTSlabs(TY, TZ, MYTHREAD)
 1  myPlane ← ⌊MYTHREAD/TY⌋
 2  myRow ← MYTHREAD%TY
 3  teamY ←  all threads who have same value of myPlane
 4  teamZ ←  all threads who have same value of myRow
 5  BARRIER()
 6  for plane ← 0 to NZ/TZ
 7  do for row ← 0 to NY/TY
 8      do  1D FFT of length NX
 9       Pack the data for this plane
10       Initiate nonblocking Exchange for this plane on teamY
11   Wait for all Exchanges to finish
12   Unpack all the data to make Y dimension contiguous
13  for plane ← 0 to NZ/TZ
14  do for row ← 0 to NX/TY
15      do  1D FFT of length NY
16       Pack the data for this plane
17       Initiate nonblocking Exchange for for the slabs on teamZ
18   Wait for all Exchanges to finish
19   Unpack all the data to make Z dimension contiguous
20  for plane ← 0 to NY/TZ
21  do for row ← 0 to NX/TY
22      do  1D FFT of length NZ
```

Figure 6.15: FFT Slabs Algorithm

6.3.2 Slabs

With the goal of overlapping communication and computation in mind we analyze a second algorithm. In the previous algorithm, each thread finishes all $\frac{NZ}{TZ}$ slabs before any communication is started. However, after a thread finishes a single slab there are no further dependencies that prevent the communication from being initiated. Thus in our *Slabs* algorithm, each thread initiates the communication on a slab as soon as the computation on that slab is finished. This overlaps the communication of the current slab with the computation of the next slab. This is more fully detailed in Algorithm 6.15.

6.3.3 Summary

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

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

Table 6.1: Summary of Message Counts and Sizes for the two different 3D FFT algorithms

have fewer larger messages in the network at the cost of the ability to overlap computation with communication[2].

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

The two different algorithms have different impacts on the network hardware. The Packed Slabs approach sends larger and fewer messages while the Slabs approach sends more and smaller messages while simultaneously enabling communication/computation overlap (the total volume of data communicated is the same for all algorithms considered). Table 6.1 shows the difference in communication behavior of the two different FFT algorithms for what a single thread sends in each round. The Packed Slabs approach can have exactly one outstanding collective and that is not overlapped with any of the computation while the Slabs approach can have up to $\frac{NZ}{TZ}$ outstanding nonblocking collectives before needing to wait for the data movement to finish. In the later sections we will highlight the regions on the bandwidth micro-benchmark described in the previous section where these two algorithms reside.

6.3.4 Performance Results

To analyze the performance at scale, the NAS FT benchmark was run upto 32,768 cores of the BlueGene/P and 1024 cores of the Cray XT4. The "UPC Slabs" shows the performance of the Slabs algorithm written in Berkeley UPC with nonblocking collectives found in GASNet. The "UPC Packed Slabs" and "MPI Packed Slabs" shows the performance of the Packed Slabs algorithm using UPC and MPI respectively. The MPI Packed slabs is the pub-

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

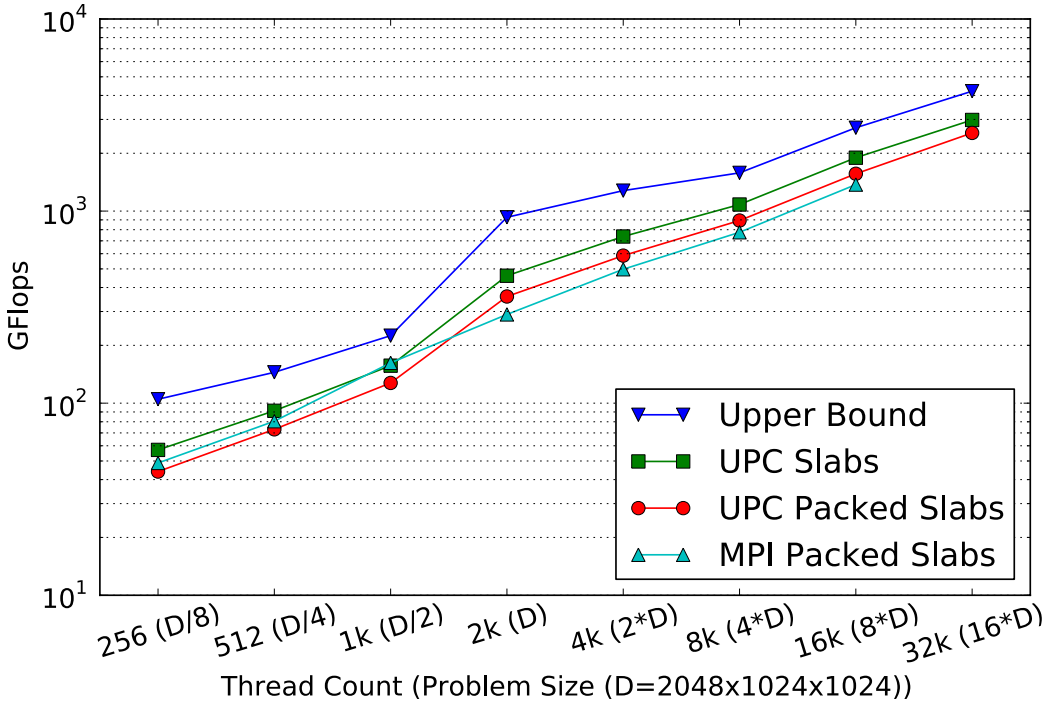


Figure 6.16: NAS FT Performance (Weak Scaling) on IBM BlueGene/P

lically available NAS 2.4 benchmark written in Fortran. The two rounds of communication are done by calls to `MPI_Alltoall()`. The Slabs algorithm is impossible to express in MPI, since the communication library does not yet have an interface to nonblocking collectives. Libraries such as Parallel ESSL [76] and P3DFFT [136] also perform 3D FFTs on the IBM BlueGene/P. Initial experiments at small scale has shown that these libraries yield similar performance to the NAS2.4 Fortran/MPI version so we only compare against this one MPI version at larger scale. Future work will compare our algorithm against these libraries.

IBM BlueGene/P

Figure 6.16 shows the performance of the NAS FT benchmark on power two core counts upto 32,768. The problem size was also scaled as the number of cores grew so that the memory per thread remains constant throughout all the processor 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. Thus it assumes the total time taken by the benchmark is the time needed to perform the Exchanges. 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 minimum number of links that need to be cut to sever the network into two equal halves. This thus provides an approximation for the aggregate Bandwidth a network

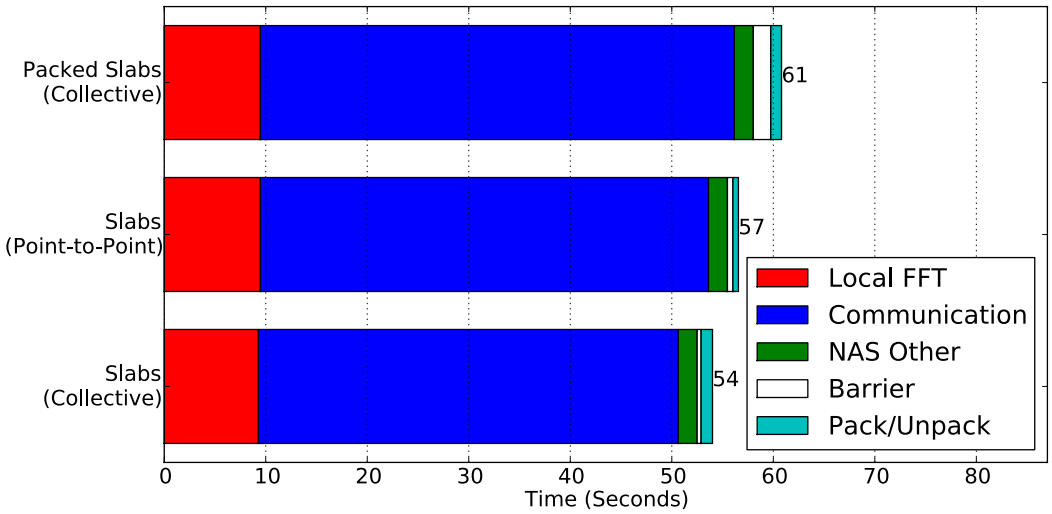


Figure 6.17: NAS FT Performance Breakdown on IBM BlueGene/P

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.

Both programming models utilize the same underlying communication layer, DCMF, for their collective implementation. As the data show overlapping the collective with the computation yields significant performance improvements (17% at 32,768 cores, for example). Thus even though the Exchanges cannot be effectively overlapped amongst themselves (as shown in Section 6.1.2, the collectives *can* be overlapped with other computation to yield good performance.

To further examine where the time is being spent in the weak scaling benchmarks we examine the performance breakdown in Figure 6.17 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, "NAS Other" measures the time for the other parts of the NAS FFT 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 Packed Slabs algorithm induces Exchanges of 128KB messages per thread and the Slabs algorithm 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/computation overlap.

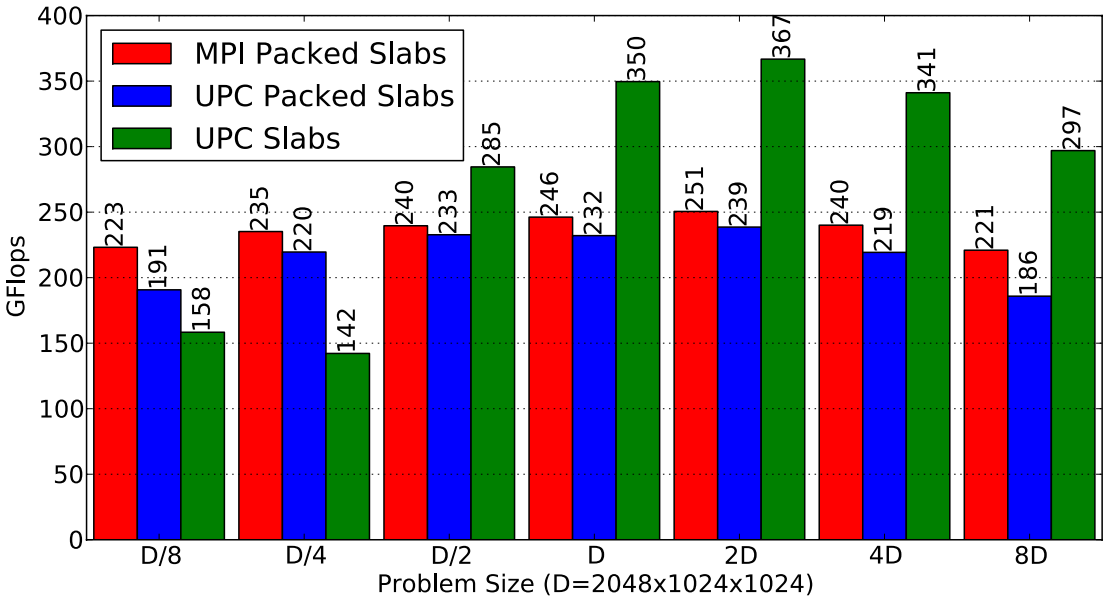


Figure 6.18: NAS FT Performance on 1024 cores of the Cray XT4

Cray XT4

To further understand the impact of overlapping the collectives with the computation we run a variety of problem sizes across a fixed number of processor cores on the CrayXT4. The data is shown in Figure 6.18. The x-axis shows the problem size in ascending order. The y-axis shows the performance in GFlop/s using 1024 cores of the Cray XT4. For this experiment we use FFTW [85] to perform the local FFTs and then use MPI or UPC respectively to perform the communication amongst the cores.

As the data show, for the smallest problem sizes the Slabs algorithm yields the poorest performance. For small problem sizes on a large number of cores the message sizes become so small that the software overheads of message injection and latency of the collective become a dominant factor and thus we are unable to effectively offload the communication. By using the packed slabs algorithm we improve the performance by getting the bandwidth advantages of using larger messages. As the problem sizes grow, the collectives themselves become more bandwidth bound and thus making the offload more useful since the RDMA hardware can effectively manage the communication. This allows the cores to perform FFTs rather than waiting on communication events thereby enabling better communication/computation overlap. At the 2D problem size (a grid size of 2048 x 2048 x 1024 elements) UPC Slabs performs 46% better than MPI Packed Slabs. As the data also show the MPI Packed Slabs also consistently outperforms the UPC Packed Slabs which is consistent with the microbenchmark data from Section 6.1. Future improvements in the Exchange algorithm for this platform will close the performance difference.

Summary

The data from both platforms show a consistent improvement in performance for overlapping the Exchange with the FFT computation through the Slabs algorithm. As a result we argue that the conventional wisdom of packing all the data into one large communication step and dividing the execution into a communication and computation phases can lead to poor performance. By overlapping the collective with the computation one can leverage all the available resources of the network. As a result we argue that nonblocking collectives will be an important part of any future collectives libraries. However, as the data also show, always relying on nonblocking collectives can lead to performance penalties especially when the collectives become too small for small problem sizes. Thus we argue that these factors must also be taken into account when tuning the collective and deciding between which styles of algorithm to use.

6.4 Summary

The main focus of this chapter was to analyze techniques for optimizing the non-rooted collectives Exchange and Gather-to-All. As the data and the performance model show, the optimal algorithm depends heavily on the transfer size for Exchange. There is a tradeoff between sending fewer messages to more intermediaries and using more bandwidth than needed to perform the collective. However for Gather-to-All, the collective is structured so that all variations utilize the same amount of network bandwidth and thus using algorithms that minimize the number of messages leads to the best performance. We are able to construct performance models that accurately map out the search space so that obviously bad algorithms do not need to be searched.

As the microbenchmark results show, the one-sided communication model in GASNet is able to achieve up to a 23% improvement in the latency for an Exchange over MPI on 256 cores of the Sun Constellation. The median improvement is 2% indicating that GASNet is able to deliver comparable performance to the vendor optimized MPI library on a wide range of message sizes. However the results for the Cray XT systems show that further algorithms must be added into the tuning space that are optimized for torus networks. For the Gather-to-All the GASNet collectives are able to realize better improvements over MPI. GASNet gets up to a 69% improvement in latency on 1536 cores of the Cray XT5 with a median improvement of 56%.

These performance benchmarks also translate well to application level benchmarks. We demonstrated the performance of these collectives in the NAS FT benchmark. As the data show, the nonblocking 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 the IBM BlueGene/P and a 46% improvement in performance on 1024 cores of the Cray XT4. As of November 2009, the peak performance of a 1D FFT according to the HPC Challenge Benchmarks [2] is 6.25 TFlops on 224k cores of the Cray XT5 and 4.48 TFlops on 128k cores. We are able

to achieve 2.98 TFlops for a 3D FFT on only 32k cores of the IBM BlueGene/P. From the application and microbenchmark results we can conclude that the one-sided communication and the ability to overlap communication allow GASNet to deliver good performance on a large number of processor cores for the communication intensive non-rooted collectives.

Chapter 7

Collectives for Shared Memory Systems

Current hardware trends show that the number of cores per chip is growing at an exponential pace and we will see hundreds of processor cores within a socket in the near future [17]. However, the performance of the communication and memory system has not kept pace with this rapid growth in processor performance [169]. Transferring data from a core on one socket to a core on another or synchronizing between cores takes many cycles, and a small fraction of the cores are enough to saturate the available memory bandwidth. Thus many application designers and programmers aim to improve performance by reducing the amount of time threads are stalled waiting for memory or synchronization.

So far we have focused on optimizing the collective communication model in PGAS languages (described in Chapter 3) for distributed memory platforms. PGAS languages are also a natural fit for multicore and SMP systems because they directly use their shared memory hardware while still giving control over locality which is important on multi-socket systems. We will show that the one-sided model when extended to collective operations allows for much higher communication bandwidth and better overall collective performance and throughput on shared memory architectures. We will show how the collective communication and techniques outlined in the previous chapters for distributed memory can also be applied to platforms that rely solely on shared memory for inter-thread or inter-process communication. Throughout the rest of this chapter we assume that there is one process running for the entire multicore system and there are t threads per process, one per each hardware execution context. On some platforms, such as the Sun Niagara2, a hardware multiplexer schedules many threads onto the one physical core. The execution state for each thread is kept in hardware to avoid the expensive overheads of relying on the operating systems to manage these threads. On the Sun Niagara2 four threads are multiplexed onto the same physical core while an Intel Nehalem core is shared by two.

Collecting and distributing the data in the previous chapters was done with a flat communication topology (i.e. all threads communicated with a single elected root). The intra-node communication could be implemented with the techniques described in this chapter. However, the two components have not been integrated together for the following reasons: (1) it

would lead to a dramatic increase in the search space, (2) it would be a lot of engineering effort for a very minimal performance improvement because the intra-node communication is rarely the performance bottleneck for collectives on distributed memory especially for large node counts. Future work will revisit this design decision depending on the performance tradeoffs.

As a case study we examine the tuning techniques and considerations for one non-rooted collective, Barrier, in Section 7.1 and two rooted collectives, Reduce and Broadcast, in Section 7.2. We validate our results on four modern multicore systems: the compute node on the IBM BlueGene/P, the Intel Clovertown, the Intel Nehalem, the AMD Barcelona, and the Sun Niagara2. We then analyze the performance of Sparse Conjugate Gradient with and without tuned collectives in Section 7.3.

Our performance results will show that by optimizing the collectives for shared memory we are able to consistently achieve two orders of magnitude improvement in the latency of a Barrier on a variety of modern multicore systems. On the platforms with higher levels of concurrency our results show that by further choosing the best communication topology and communication mechanism we are able to get a further 33% improvement in the latency of a Barrier on 32 cores of the AMD Barcelona and a 46% improvement on 128 cores of the Sun Niagara2. We will also demonstrate that when these tuned collectives are incorporated into Sparse Conjugate Gradient they can deliver up to a 22% improvement in overall application performance by dramatically decreasing the time spent in the communication intensive phases.

7.1 Non-rooted Collective: Barrier

To motivate our work we initially focus on an important collective found in many applications: a barrier synchronization. Having a faster barrier allows the programmer to write finer-grained synchronous code and conversely a slow barrier hinders application scalability as shown by Amdahl's Law. As highlighted in the seminal work by Mellor-Crummey and Scott [121], there are many choices of algorithms to implement a barrier across the threads. One of the critical choices that affects overall collective scalability is the communication topology and the schedule that the threads use to communicate and synchronize with each other. Tree-based collectives allow the work to be more effectively parallelized across all the cores rather than serializing at one root thread, thereby taking advantage of more of the computational facilities available. To limit the search space we only consider K-nomial trees (see Section 5.1.2) of varying radices.

To implement a barrier each thread signals its parent once its subtree has arrived and then waits for the parent to signal it indicating that the barrier is complete. Two passes of the tree (one up and one down) will complete the barrier. All the barriers have been implemented through the use of flags declared as volatile `int`s and atomic counters.

There are two different ways to signal in each direction which we term "Pull" and "Push".

- **Pull:** On the way up the tree a "Pull" signal means that each child sets a boolean flag

Processor Type	GHz	Threads/ Core	Cores/ Socket	Sockets	Total Threads
IBM BlueGene/P	0.85	1	4	1	4
Intel Clovertown	2.66	1	4	2	8
Intel Nehalem	2.67	2	4	2	16
AMD Barcelona	2.30	1	4	8	32
Sun Niagara2	1.40	8	8	2	128

Table 7.1: Experimental Platforms for Shared Memory Collectives

(implemented as a volatile `int` in C[1]) it owns. The parent polls flags of each of the flags belonging to the children until they have been set before setting its own boolean flag. On the way down the tree the parent sets a boolean flag indicating that the signal has arrived. All the children poll a single flag logically belonging to the parent, waiting for it to change.

The flags are written exactly once but can be read an arbitrary number of times. On modern cache-coherent multicore chips this means that each thread polling a flag will get a copy of the cache line. Once the signaler sets the flag, the cache coherency system will automatically throw away stale copies of the flags on different threads and the subsequent read of the flag from memory will yield the correct result.

- **Push:** On the way up the tree a "Push" mechanism means that the children will increment an atomic counter on the parent thread. The parent waits for the atomic counter to reach the number of children that it has. On the way down the tree, the parent sets the flag on each of its children and each of the children will spin on a local flag.

  Since the read and increment of the counters must be atomic, there will be a larger overhead than with simply setting an flag, however the polling for both these mechanisms relies on only polling flags that the threads "own."

Thus there are two different signaling mechanisms on the way up the tree and two on the way down leading to a combination of four different ways to implement the Barrier that is orthogonal to the choice of the Tree. Figure 7.1 shows the performance of the various algorithms on our experimental platforms shown in Table 7.1. For the distributed memory collective we construct the trees over the nodes where as in this case we construct the trees over the nodes. In addition to the Tree algorithms described above, we also implement the Dissemination algorithms from Chapter 6 (a Barrier can also be implemented as a 0 byte Exchange of various radices). For comparison we show the performance of a Barrier using the Pthread library either through condition variables. Some Pthread libraries implement barriers as part of their interface so when available those were used.

[1]Special care has been taking to ensure that the flags for different threads do not share the same cache line to avoid false sharing.

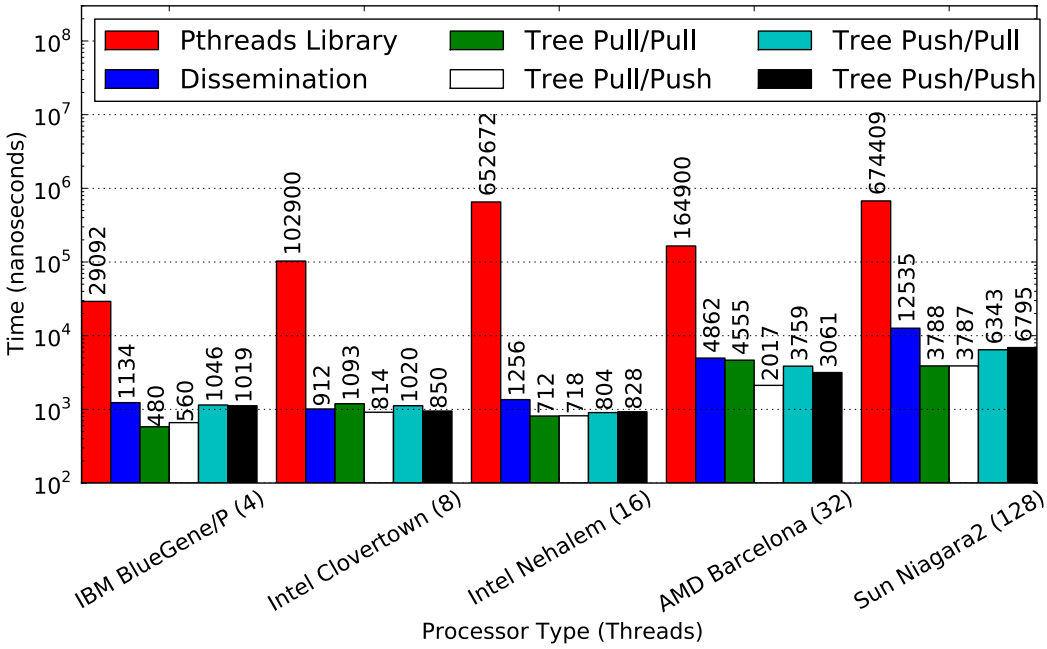


Figure 7.1: Comparison of Barrier Algorithms for Shared Memory Platforms

As the data show the Pthread library yields very poor performance when compared to implementations through signaling. The barriers found in the library are designed to be general purpose such that when there are more threads than hardware execution contexts, the barrier performance will not dramatically fall. However, when the hardware is not oversubscribed with more threads than available hardware contexts, the performance is poor. In addition, as the data show, the performance of the Dissemination algorithm is never optimal and the best performance comes from leveraging trees. A dissemination algorithm would induce $O(n\ log\ n)$ "signals" in the interconnection network amongst the cores whereas the tree based approaches would only incur $O(n)$ signals at a higher latency cost[2]. Since two passes of the trees are required the tree algorithms will incur a latency cost of $O(2 * (log\ t))$. The dissemination algorithms also require all threads to be attentive and active to forward data. Thus on platforms in which multiple threads are multiplexed on the same physical core (i.e. the Intel Nehalem and the Sun Niagara2) the dissemination will pay a latency cost since the threads might not be scheduled in timely fashion when the threads need to send or receive data. Thus on these platforms the dissemination algorithms are sub-optimal. As the data also the different signaling mechanisms yield different relative performance highlighting the differences from their cache coherency systems and the cost of atomic operations; the "Tree Pull/Push" tends to yield close to optimal performance on all our experimental platforms.

Figure 7.2 shows the performance of the *Flat Tree* versus the Best tree geometry. A Flat Tree is defined to be one where all threads except thread 0 are a direct child of thread

[2]We do not consider tree algorithms for Exchange because they would require a non scalable $O(B(Nt)^2)$ bytes of auxiliary storage at the root.

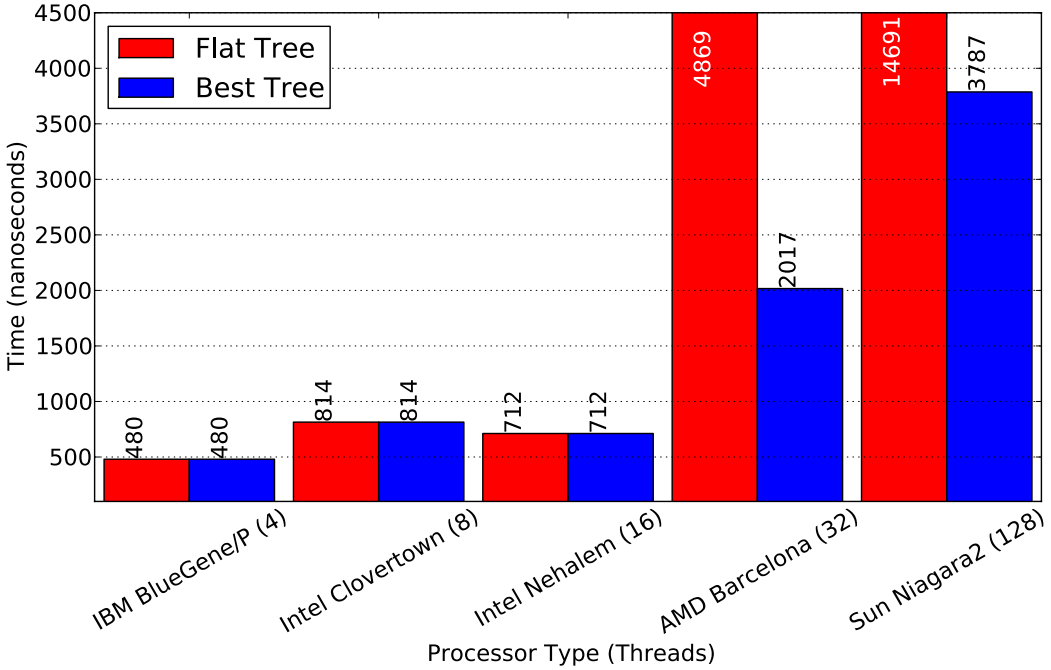


Figure 7.2: Comparison of Barrier Algorithms Flat versus Best Tree

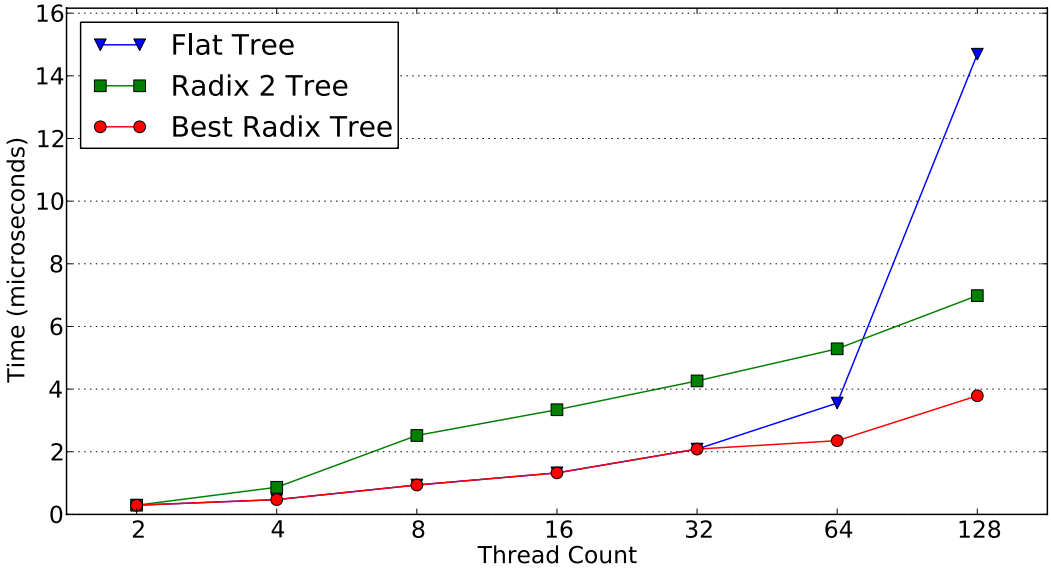


Figure 7.3: Comparison of Barrier Tree Radices on the Sun Niagara2

0. Thus as the data show, as the number of hardware thread contexts increase there is a large performance gain from leveraging trees (60% on the AMD Barcelona and 288% on the Sun Niagara2). Thus as core count continues to grow on future platforms we expect that trees will take on more importance and are an important tuning consideration going forward. Figure 7.3 shows the performance of different tree geometries on the Sun Niagara2 for varying core counts. The experiment first fills up all the hardware contexts within a core, then fills up the cores, and then fills up the sockets. As the data show, the Flat Tree is optimal at small thread counts, however when all the threads within a socket are filled, the Flat Tree no longer yields the best performance. Higher tree radices beat the traditional radix-2 binomial tree presented by Mellor-Crummey and Scott highlighting the necessity for search and automatic tuning.

7.2 Rooted Collectives

In this section we focus on the optimizations for Rooted Collectives. Like Barrier, the tree geometries described in Section 5.1.2 can also be applied to the rooted collectives. To focus our analysis we choose Reduce and Broadcast as a case study and analyze its implementation on two of our experimental platforms: the AMD Opteron with 32 threads and the Sun Niagara2 with 128 threads.

7.2.1 Reduce

Reduce is a very common collective found in many parallel applications and libraries. The collective allows results from different threads working on potentially different tasks to be aggregated. In order to aggregate the results, Reduce typically has to go through the cores' arithmetic units. Thus, as we will show, ensuring that all available arithmetic units in a system are used is an important optimization. We focus on the AMD Opteron and the Sun Niagara2 for our case study on Reduce since our experimental results also showed that the best performance for the IBM BlueGene/P, the Intel Clovertown, and the Intel Nehalem was the trivial case in which one thread handles the computation for all the other threads. As a result of the relatively small number of threads, the benefit of parallelizing the reductions did not outweigh the overhead of the synchronization induced by using trees. However, for the AMD Opteron and the Sun Niagara2, serializing the computation at the root proved to be a nonscalable operation as we will demonstrate. As core counts continue to grow we expect this latter case to be the norm rather than an outlier.

We first analyze the implications of collective synchronization on shared memory platforms (see Section 4.2.1) and show that the performance advantages of loosening the synchronization shown in Chapter 5 also apply to collectives targeted at shared memory. We then discuss how the synchronization affects the optimal tree shape and the performance tradeoffs that exist.

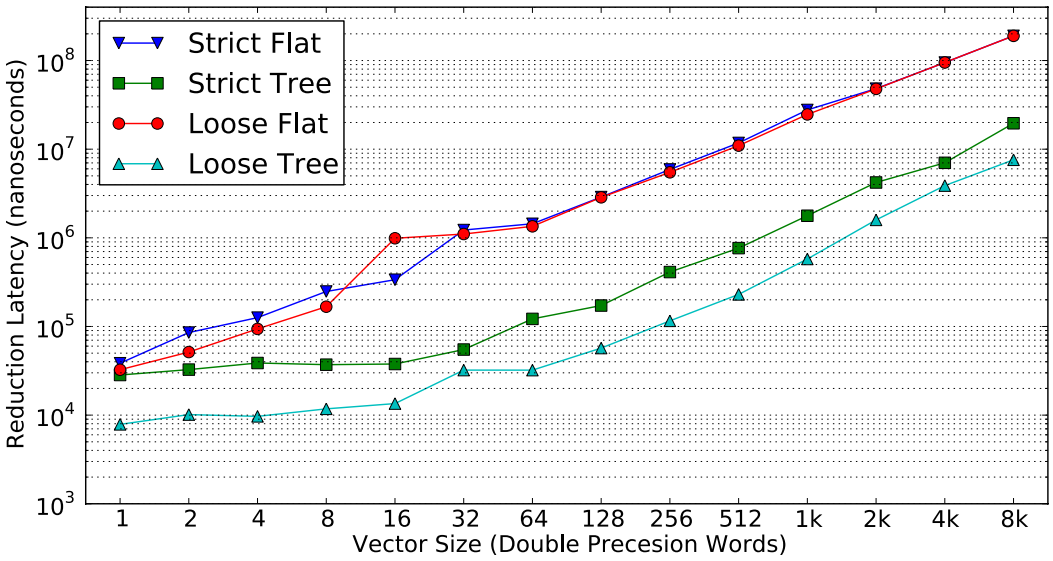


Figure 7.4: Comparison of Reduction Algorithms on the Sun Niagara2 (128 threads)

Collective Synchronization

Each Niagara2 socket is composed of 8 cores each of which multiplexes instructions from 8 hardware thread contexts. Thus, our experimental platform has support for 128 active threads. Due to the high thread count, we consider it a good proxy for analyzing scalability on future manycore platforms. We explore two different synchronization modes: Loose and Strict. In the Loose synchronization mode, data movement for the collective can begin as soon as *any* thread has entered the collective and continue until the *last* thread leaves the collective. In the Strict mode data movement can only start after *all* threads have entered the collective and must be completed before the *first* thread exits the collective. In all our examples the Strict synchronization has been achieved by inserting the aforementioned tuned barrier between each collective. There are many synchronization modes that lie between these two extremes, however for the sake of brevity we will focus on the two extremes.

Figure 7.4 shows the performance of *Reduce* on the Sun Niagara2. The x-axis shows the number of doubles reduced in the vector reduction and the y-axis shows the time taken to perform the reduction on a log scale. We also show the performance of implementing both the synchronization modes with having every thread communicate with the root directly (*Flat*) or every thread communicating through intermediaries (*Tree*). As the data show, the looser synchronization yields significant performance advantages over a wide range of vector sizes. At the lower vector sizes the memory system latency becomes the dominant concern. Thus requiring a full barrier synchronization along with the reduction introduces significant overheads. Thus, by amortizing the cost of this barrier across many operations, we can realize significant performance gains.

However, the data also show that the looser synchronization continues to show factors of 3 improvement in performance over the strict synchronization versions where one would imagine the operations to be dominated by bandwidth. Loosely synchronized collectives

allow for better pipelining amongst the different collectives. At high vector sizes both synchronization modes realize the best performance by using trees. In a strict synchronization approach a particular core is only active for a brief period of time while the data is present at its level of the tree. During the other times the core is idle. Loosening the synchronization allows more collectives to be in flight at the same time and thus pipelined behind each other. This allows the operations to expose more parallelism to the hardware and decrease the amount of time the memory system sits idle. As is the case with any pipelined operation, we have not reduced the latency for a given operation but rather improved the throughput for all the operations. As the data show in Figure 7.4, the median performance gain of the strict execution time compared to the loose execution time is about 3.1× while the maximum is about 4×.

However, there are some cases in which the synchronization cannot be amortized across many operations. In these situations specifying the strict synchronization requirements can also be beneficial to optimize the collective. Let us consider another example in which we need to perform a reduce followed by a barrier. If we assume that the processors are logically organized into a tree, then a reduction would require one pass up the tree to sum the values. The barrier would then require another two passes of the tree, a discovery phase in which all threads advertise that they have arrived a barrier and a notify phase in which all threads learn that all other threads have arrived. If the user were to specify the synchronization requirements of the collective, it can then use this semantic information to reduce the number of passes of the network. In this case, the reduction itself can substitute for the discovery phase thus saving one full traversal of the processor grid. If we again assume that the time to traverse the tree is the dominant factor in the reduction then by reducing the number of traverses from 3 to 2 will reduce the time by 33%.

Tuning Considerations

In previous sections we have seen the effectiveness of both collective tuning and loosely synchronized collectives. In this section we combine the two pieces and show that the collective synchronization must be expressed through the interface to realize the best performance.

To illustrate our approach we show the performance of Reduce on the eight socket quad-core Barcelona (i.e. 32 Opteron cores). The results are shown in Figure 7.5. In the first topology, which we call Flat, the root thread accumulates the values from all the other threads. Thus only one core is reading and accumulating the data from the memory system while the others are idle. In the second topology (labeled Tree) the threads are connected in a tree described above[3]. Once a child has accumulated the result for its entire subtree, it then sends a signal to the parent allowing the parent to accumulate the data from all its children. We search over a set of trees and report the performance for the best tree shape at each of the data points. Unlike the Flat topology the Tree topology allows more cores to participate in the reduction but forces more synchronization amongst the cores. Orthogonally we present the two aforementioned synchronization modes: Loose and Strict.

[3]We perform an exhaustive search over the tree topologies and report the best one. The best tree can vary for each vector size.

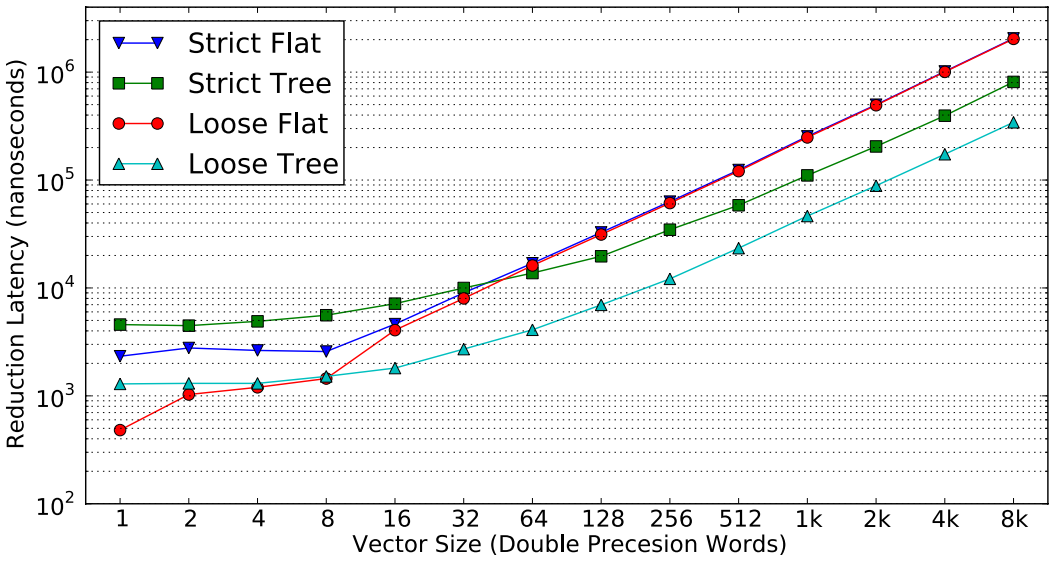


Figure 7.5: Comparison of Reduction Algorithms fon the AMD Opteron (32 threads)

As the data show, the Flat topology outperforms the Trees at smaller vector sizes. Even in the loosely synchronized collectives, the tree based implementations require the threads to signal their parents when they finish accumulating the data for their subtree. Since the Flat topology outperforms the Tree one, this indicates the overheads of the point-to-point synchronizations make the algorithm more costly especially when the memory latency is the biggest consideration. However, as the vector size increases, serializing all the computation at the root becomes expensive. Switching to a tree is a critical for performance in order to engage more of the functional units and better parallelize the problem. Both the Strict and Loose see a crossover point that highlights this tradeoff. As the data also show, the optimal switch-point is dependent on the synchronization semantics. Since the looser synchronization enables better pipelining the costs of synchronization can be amortized quicker, thereby reaping the benefits of parallelism at a smaller vector size. There is a large performance penalty for not picking the correct crossover point. If we assume that the crossover between the algorithms is at 8 doubles (the best for the loose synchronization) for both synchronization modes, then the strict collective will take twice as long as the optimal. If we employ a crossover of 32 doubles then a loosely synchronized collective will take three times as longer. Thus the synchronization semantics are an integral part of selecting the best algorithm.

Tree Selection

Table 7.2 shows the performance of Loose and Strict synchronization on the Barcelona and the Niagara2 as a function of the tree radix. On both platforms a 1-nomial tree (i.e. all the threads are connected in a chain) is the optimal for loosely synchronized collectives and a higher radix tree is optimal for strictly synchronized collectives. The lower radices impose a higher latency for the operation since they imply deeper trees. The higher radices reduce the amount of parallelism but improve the latency since the trees are shallower. Thus we

	Barcelona		Niagara2	
Tree Radix	Loose	Strict	Loose	Strict
1	**46.4**	306	**576**	3,103
2	52.9	**110**	621	2,115
4	60.1	119	710	**1,774**
8	73.8	130	1,316	2,471
16	110	213	2,240	3,998

Table 7.2: Time (in μs) for 8kB (1k Double) Reduction. Best performers for each category are highlighted

tradeoff increased parallelism for increased latency. If the goal is to maximize collective throughput (as is the case with loosely synchronized collectives), then the increased latency is not a concern since it will be amortized over all the pipelined operations and the deep trees do not adversely affect performance. However, if collective latency is a concern then finding the optimal balance between decreased parallelism and tree depth is key. On the Niagara2 trying to force a radix-2 tree has a penalty of 7% in the loosely synchronized case and 16% in the strictly synchronized case. Thus we argue that the synchronization semantics of the collective also determine the optimal communication topology.

7.2.2 Other Rooted Collectives

The other rooted collectives (Broadcast, Scatter and Gather) have also been implemented for shared memory platforms. As with Reduce, we have implemented these algorithms to use a wide variety of trees and loosen the synchronization mode where possible.

Broadcast

As we have shown in Chapter 5, leveraging trees can yield significant speedups with Broadcast. Using a tree one is better able to better leverage the available memory bandwidth. For example, the Niagara2 system that we present has two sockets. In order to send data from one socket to another, it has to cross the interconnection network that connects both the sockets. In a Flat tree, the root thread will redundantly send many copies of the same data for each thread on the remote socket. However by constructing a tree that matches the interconnect topology the root thread needs to only send one message to the remote socket and passes the responsibility of replicating and disseminating the data to the other threads to a thread on the remote socket which can realize significantly better bandwidth.

Figure 7.6 shows the Broadcast latency performance for small message sizes while Figure 7.7 shows the bandwidth performance of Broadcast for larger message sizes. From the data for small message sizes, the data show that leveraging using trees for small message sizes can yield significant performance improvements over flat communication topologies; for small message sizes the tree algorithms take about a third of the time as the flat algorithm.

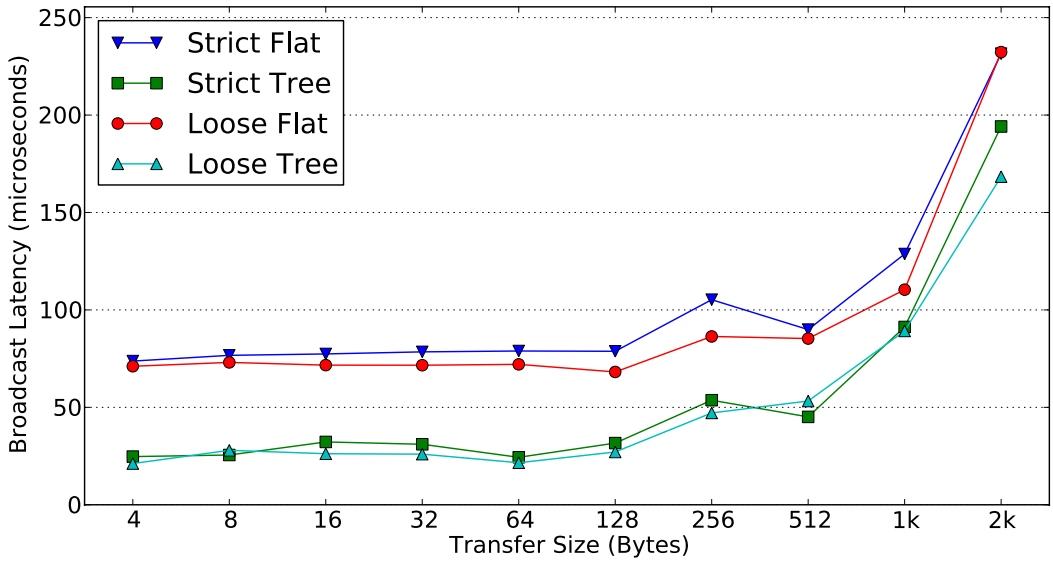


Figure 7.6: Comparison of Broadcast Algorithms on the Sun Niagara2 (128 threads) for Small Message Sizes

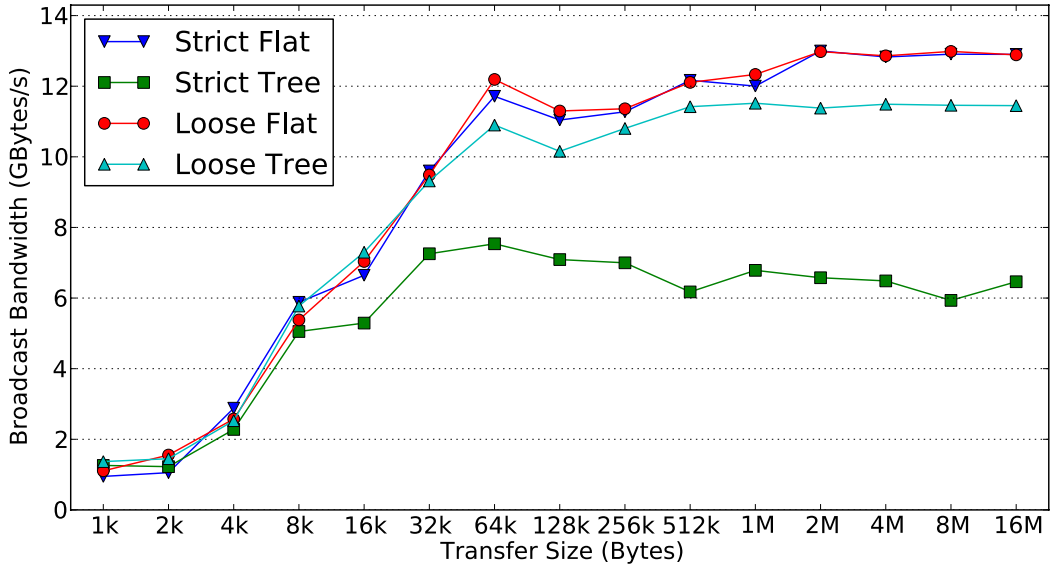


Figure 7.7: Comparison of Broadcast Algorithms on the Sun Niagara2 (128 threads) for Large Message Sizes

However, an interesting difference between Reduce and Broadcast is that loosening the synchronization mode does not yield good performance improvements which is a different result than collectives targeted for distributed memory and thus the benefits of pipelining the collectives behind each other do not provide fruitful results. In the case of Reduce, the pipelining was aided by the fact that each intermediate level of the tree had their own dedicated floating point units (for any radix larger than 4 on the Sun Niagara2) and thus could operate on different collectives simultaneously. However, for the data movement collectives, the resources that the collectives rely on are the various components of the memory system such as the caches, the memory controllers, and the memory busses. Some of these resources are shared amongst all the threads and thus hinders the ability of different threads to work on different collectives simultaneously. However, since the bandwidth to the caches within a socket is much higher than the bandwidth to the caches on remote sockets, leveraging trees to ensure that the data doesn't traverse the critical links numerous times yields the best performance.

However, as the data show, as the message size gets larger the benefit of trees gets diminished. Since the data buffers owned by the threads are too large to fit into cache, the Broadcast must read and write data in the memory system to which the bandwidth is much more restricted than the bandwidth when the data was strictly within the caches. Using trees implies that multiple cores will simultaneously try to read and write data through to the memory across the link that has lower bandwidth. Thus, by allowing only one thread to manage the data movement and mitigating the contention on the memory system, as is done with the Flat tree, the Broadcast can realize the highest performance. Thus unlike the distributed memory collectives, having the data movement resource be shared across all the threads severely hinders the ability for looser synchronization and trees to boost the performance.

Scatter and Gather

Our results show that the Scatter and Gather do not realize speedups from leveraging trees; the optimal performance results from having the root thread directly communicate with every other thread. As our models in Section 5.3 show, the Scatter and Gather, the extra bandwidth costs associated with the data movement are not out weighed by the latency advantages. Due to the much smaller message injection overhead and message latencies the bandwidth becomes the dominant concern and thus any collective algorithm that relies on sending the data multiple times through the memory system do not yield good performance. Since Flat trees yielded the best performance the advantages of pipelining the collectives behind each other were also diminished and therefore the performance advantages by loosening they synchronization modes are not as pronounced. Future work will analyze whether the tree based Scatter or Gather algorithms realize better performance as the number of cores continues to grow.

Matrix ID	Application Area	N	nonzeros	nonzero ratio
finan512	Computation Finance	74,752	596,992	1.07E-04
qa8fm	Acoustics Simulation	66,127	1,660,579	3.80E-04
vanbody	Structural Simulation	47,072	2,329,056	1.05E-03
nasasrb	Structural Simulation	54,870	2,677,324	8.89E-04
Dubcova3	2D/3D PDE Solver	146,689	3,636,643	1.69E-04
shipsec5	Structural Simulation	179,860	4,598,604	1.42E-04
bmw7st_1	Structural Simulation	141,347	7,318,399	3.66E-04
G3_circuit	Circuit Simulation	1,585,478	7,660,826	3.05E-06
hood	Structural Simulation	220,542	9,895,422	2.03E-04
bmwcra_1	Structural Simulation	148,770	10,641,602	4.81E-04
BenElechi1	2D/3D PDE Solver	245,874	13,150,496	2.18E-04
af_shell7	Structural Simulation	504,855	17,579,155	6.90E-05

Table 7.3: Matrices from UFL Sparse Matrix Collection used in Conjugate Gradient Case Study

7.3 Application Example: Sparse Conjugate Gradient

In order to demonstrate the value of tuning collectives we incorporate these tuned collectives into a larger application benchmark, Sparse Conjugate Gradient [68]. This application iteratively solves the equation $A \times x = b$ for x given a sparse symmetric positive definite matrix A and dense vector b. In this method an initial solution for x is guessed and is iteratively refined until the solution converges. At the core of the benchmark is a large Sparse Matrix Vector Multiply (SpMV). To refine the solutions parallel dot products are used which require a scalar *Reduce* followed by a *Broadcast*. In addition, we also employ the *Barrier* for interprocessor synchronization. For the best SpMV we use the routines from the SpMV optimized by Williams et. al. [167] and use the collectives described in this chapter.

Figure 7.8 shows the performance of this benchmark on a dual socket Sun Niagara 2. On the x-axis we show an assortment of matrices from the University of Florida's Sparse Matrix Collection [66] sorted by the number of nonzeros. The nonzero ratio is defined as the number of non-zero elements in the matrix divided by the total number of elements in the matrix. The salient features of the matrices are shown in Table 7.3. The y-axis shows the performance achieved in Gigaflops with and without the tuned collectives. As the data show, the matrices on the right hand side, which have the smallest nonzero counts, tend to benefit more from tuned collectives since more of the runtime is spent in the communication routines. As the number of nonzeros in the matrix grows, the serial computation dominates the total runtime and thus the benefits from the collectives are diminished. As the data show the tuned collectives provide up to a 20% speedup in overall application performance on the smaller matrices.

In order to further analyze performance we show where the time is spent for the algorithm in Figure 7.9. The x-axis again shows the different matrices sorted by the number of nonzeros. The y-axis shows the time spent in the different components of the algorithm

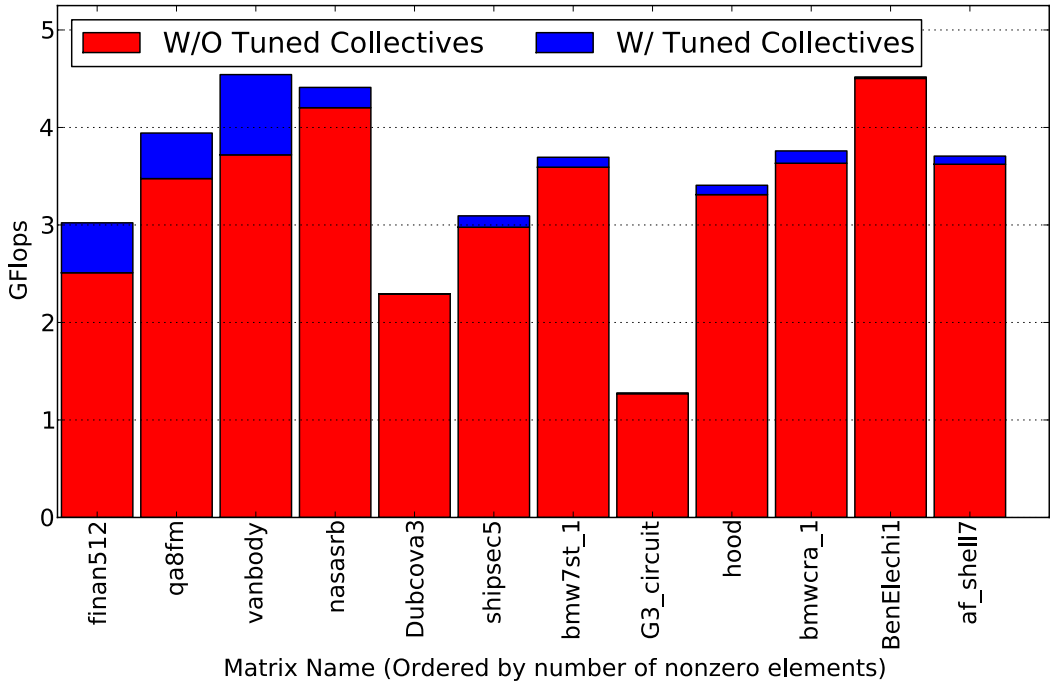


Figure 7.8: Sparse Conjugate Gradient Performance on the Sun Niagara2 (128 threads)

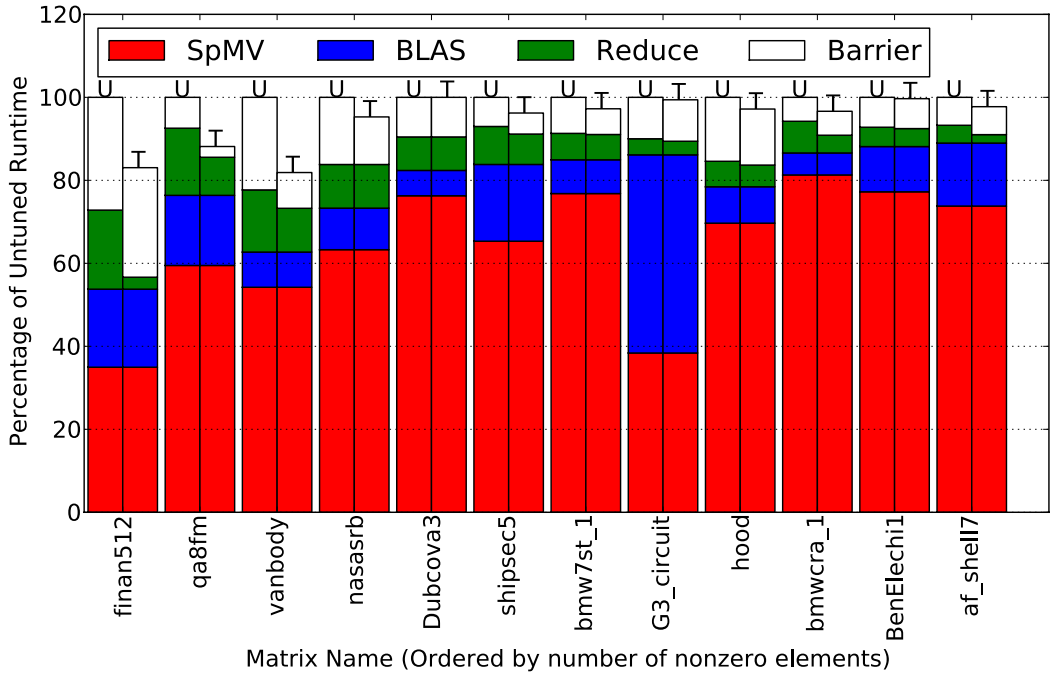


Figure 7.9: Sparse Conjugate Gradient Performance Breakdown on the Sun Niagara2 (128 threads)

normalized by the total time taken by the untuned runtime. The bars marked with "U" represent the performance from the untuned collectives while the bars marked with "T" show the performance of Sparse Conjugate Gradient using tuned collectives. The data has been broken up into four main categories: "SPMV" is the time spent in the SpMV and has been optimized in an external library, "BLAS" is the time spent in BLAS1[33] operations to perform local computation, "Reduce" is the time spent in reductions for the dot-products and "Barrier" is the time spent in the barrier[4]. Since our tuning is done only for the Reduce and Barrier in Conjugate Gradient and not in the SpMVs, the performance of the SpMV and BLAS1 operations is the same for both the tuned and untuned cases. Future work will also optimize the collectives within the Parallel SpMV.

Thus as the data show, for small numbers of nonzeros the tuned collectives can constitute a significant part of the runtime (up to 46% for finan512). For finan512 the data show that the time spent in the reduction went down from 19% of the overall untuned runtime to less than 3% of the overall tuned runtime. However for some of the matrices the tuned collectives did not aid in performance from the flat trees. Our hypothesis is that load imbalance amongst the different threads caused the trees to become less useful since the intermediary threads in the tree were late in arriving at the collective and thus were not able to forward the data in a timely fashion. Future work will validate this hypothesis.

As the number of nonzeros grows, the time spent in the collectives becomes smaller and smaller as the SpMV and BLAS1 operations become the dominant part of the runtime; the collectives only account for 11% of the overall runtime on af_shell7. In this last case the reductions only accounted for 4% of the untuned runtime and 2% of the tuned runtime and thus, even though the reduction performance was greatly improved, the overall impact was smaller since the other parts of the conjugate gradient algorithm were the dominant factors. From the data we can draw two important conclusions: (1) in order for the collective tuning to provide a useful impact the collectives must play a significant part in the overall runtime and (2) the threads must be relatively load balanced in order for the intermediary nodes to be able to forward the data in a timely fashion.

7.4 Summary

As the data show, many of the optimizations and techniques that were employed for collectives targeting distributed memory systems also apply for pure shared memory platforms. One of the key differences, however, is how the various cores and threads share their resources. In distributed memory systems, the communication subsystems are replicated throughout the machine so that each group of threads has dedicated communication resources. However, on the shared memory platforms, more of the resources are shared amongst all the other threads. Thus algorithms that we thought were useful in distributed memory (e.g. a tree for large broadcasts) lead to poor performance on shared memory platforms. Thus, if our aim is to build a collectives library that is tuned for a wide variety

[4]In our measurements the time spent in Barrier also accounts for load imbalance amongst the various threads and thus there is no easy way to tease apart these two components from this measurement.

of platforms, then these differences must be taken into account. Like the results from the previous chapters, the synchronization modes offered by the PGAS languages must be taken into account to yield the best performance.

As we have shown, the tuned collectives can deliver up to two orders of magnitude improvement in Barrier latency. Further tuning suggests another 50% improvement can be gained by choosing the optimal communication topology and signalling mechanisms. In addition we demonstrate how the loosening the synchronization can yield a consistent factor of 3 improvement in performance on the platforms that have the highest levels of parallelism. We also show that these tuned can be incorporated into the Sparse Conjugate Gradient benchmark to realize good performance improvements by leveraging the tuned collectives. In some cases they provide up to a 22% improvement in performance. Thus, like the results from distributed memory, tuning collectives for shared memory is an important optimziation especially as the levels of parallelism inside modern multicore processors continues to grow. The wide variety of multicore processors available and in development for the future will necissitate a system that can automatically tune the collectives for a wide variety of systems.

Chapter 8

Software Architecture of the Automatic Tuner

Thus far our analysis has focused on the different factors that affect the collective performance. The previous chapters outlined the algorithmic and parameter space for the collectives. The wide variety of interconnects and processors that are currently deployed and under development necessitates a system that can automatically tune these operations rather than wasting valuable time hand-tuning these collectives to find the best combination of algorithms and parameters. As we have shown, many of the optimal tuning decisions are dependent on inputs known only at runtime such as the synchronization mode and the message sizes further exasperating the problem.

To address these concerns we have built a system around the collective library that can automatically tune these operations for a wide variety of platforms. The main aim of the automatic tuning system is to provide performance portability. That is, with limited effort from the end user, the collective library will yield optimal or close to optimal solutions for the collectives irrespective of the platform.

The rest of this chapter outlines the software architecture for the automatic tuner. The discussion first starts with an outline of previous efforts in automatic tuning in Section 8.1. The rest of the chapter starting with Section 8.2 provides a general overview of the software architecture of the system. Sections 8.3.2 and 8.3.3 go into much greater depth of when and how the tuning is done.

Our results will demonstrate that by leveraging the aforementioned performance models we can in, the best case, negate the need for search by picking the best algorithm. When search is required we take, on average, 25% of the time needed for an exhaustive search to find the best algorithm. Our performance models do a good job of placing obviously bad candidate algorithms at the end of hte search space so that these algorithms will seldom have to be run.

8.1 Related Work

Automatic tuning has been applied to a variety of fields in scientific and high performance computing. In our work on constructing an automatic tuning system we build upon some of the concepts from prior work.

This is not a complete list of the automatic tuning systems in existence but rather the systems that have influenced our design. Rather than going into the full details of each of the successful systems we highlight the salient parts of the design decisions that are applicable to the collectives.

- **ATLAS:** One of the first and most widely used automatic tuning systems is ATLAS (Automatically Tuned Linear Algebra Software) [166]. These libraries provide a portable and high performance version of the BLAS library [33]. ATLAS targets single threaded performance. For Dense Linear Algebra, the optimal algorithmic choice is easily identifiable by the input parameters to the library (e.g. matrix sizes and access patterns) and the target platform so the tuning is primarily done offline. ATLAS performs a benchmarking run at install time and stores the results that are then used during the runtime. Thus, since all the tuning is done at install time, there is no cost to the end user of the library for the tuning.

- **Sparsity and OSKI:** Two successful automatic tuning efforts for Sparse Matrix Vector Multiplication (SpMV) are Sparsity [99] and OSKI [164]. Unlike Dense Linear Algebra, the input matrix heavily influences the optimal algorithmic choice for SpMV. The sparsity pattern of the matrix and the performance of the memory system dictate the best choice for the algorithm hence requiring tuning during runtime. To minimize the time spent at runtime finding the best algorithm, these systems build performance models [165, 131] to pick the best algorithm. They use a combination of offline heuristics and benchmarks alongside performance models at runtime to determine the best algorithm. OSKI also has an interface that exposes the cost of tuning to the end user so that decisions of how long the tuning process should take are exposed to the library.

- **Spiral and FFTW:** Spiral [144] and FFTW [85] both provide automatically tuned kernels for a variety of spectral methods. Like OSKI, the tuning for FFTW is done at runtime requiring a mechanism to control the time for search. FFTW exposes different levels of tuning (e.g. *Estimate*, *Measured*, and *Exhaustive*) that indicate the level of tuning desired compared. Each of the levels takes longer to tune but yields a more refined and potentially faster solution. Thus, depending on how many calls are needed to amortize the cost of tuning, the user can make decisions based on the level of tuning.

- **Parallel LBMHD, SpMV and Stencils:** The previous works have focused entirely on serial performance. A recent PhD thesis by Sam Williams [169] has shown how two parallel kernels, LBMHD and SpMV can be optimized. One of the major contributions of the work is to show how a performance model called the Roofline Model[168] can highlight the tradeoffs on various architectures of memory bound and compute bound

operations and how the tuning for each category change accordingly. Datta et al. [65] go into further detail how to optimize a stencil (nearest neighbor) computation. The performance of this optimization is bound by the performance of the memory system and hence some of the worries about latency and bandwidth are also important there.

8.2 Software Architecture

In this section we outline the various components of the software architecture and show how the components fit together. We also highlight the differences and similarities between the related automatic tuning projects.

8.2.1 Algorithm Index

Thus far this dissertation has outlined many possible algorithms and parameters that implement the various collectives. The automatic collective tuning system in GASNet stores all these algorithms for the various collectives in a large index implemented as a multidimensional table of C structures. For each type of collective, a collection of possible algorithms is available (e.g. Eager, Signaling Put, or Rendez-Vous Broadcast). 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. For example, an Eager Broadcast advertises that it works with all synchronization and address modes but requires that the transfer size be below the maximum size of a Medium active message. In another case, a Dissemination Exchange can advertise that it can provide an implementation for Exchange that works for only a subset of the synchronization modes but requires a specified amount of scratch space. In the latter case, if the tuning system realizes that the scratch space requirements are too high then the algorithm will not be run.

By allowing different algorithms to have different capabilities, we allow more specialized collectives to be written. That is collectives that work well for certain input parameters but will not work for other input parameters. This relieves the burden of ensuring that all the algorithms work for all the combinations of the input parameters. 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 (see Section 5.1.6) into the tuning framework. We chose to include the hardware collectives in the search space rather than always using them for two reasons. The first is that on some platforms there are many variants of the hardware collectives and it is not often obvious which to use. The second is that some vendor supplied collectives libraries have been designed for MPI. Thus some of the novel considerations that this dissertation raises (such as synchronization semantics and asynchronous collectives) may make the algorithms a bad fit for Partitioned Global Address Space Languages and one of the generic algorithms found in GASNet might perform better. Some network APIs, for example the DCMF library on the IBM BlueGene/P, provide multiple mechanisms to implement the same collective and thus we must still choose one. The choice is not always obvious and thus they must be

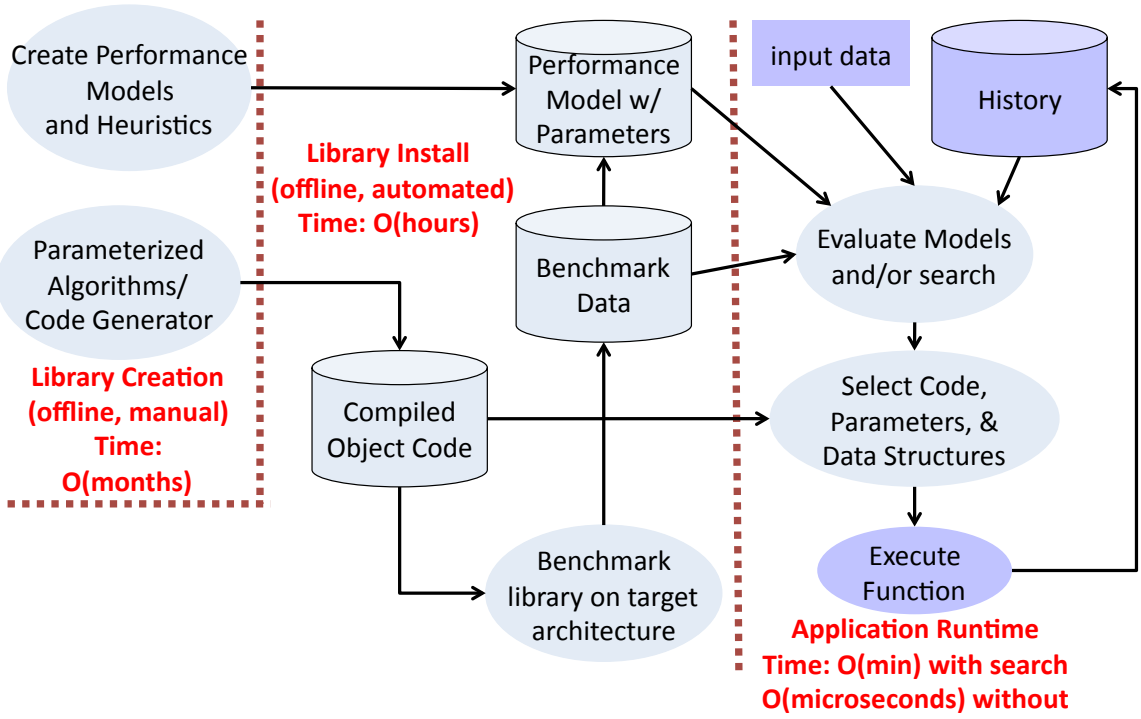


Figure 8.1: Flowchart of an Automatic Tuner

considered in part of the search space. In addition, we expect that as the research in this area moves forward more sophisticated algorithms will be introduced. The extensibility allows these novel algorithms to be added to the automatic tuning system to allow these algorithms without making major changes to the core of the tuning system.

Many other related automatic tuning systems use a combination of code generation and runtime parameters to generate parameterized algorithms. For example, related work for Sparse Matrix Vector Multiplication and Stencil Computations leverage code generators to control the depth of loop unrolling and software prefetch to ensure that the compilers do not generate suboptimal code for these optimizations. For the collectives, however, the overall runtime of the collective on a large distributed system is much larger than any gains made from manually hand optimizations such as loop unrolling. Thus due to the dramatic increase in the engineering effort combined with the minimal performance improvement we have chosen not to pursue this route. Future work will reevaluate these decisions if and when these hand optimizations become relevant for the collectives.

8.2.2 Phases of the Automatic Tuner

In broad terms, all the automatic tuners share many common components. The sophistication and the importance of the various components depends on the kernel they target and the environment. The automatic tuning system can be broken into the following three distinct phases in which different components are constructed and analyzed.

Library Creation

The first step for generating an automatic tuner is to identify all the various algorithmic variants that are available and create parameterized algorithms or a code generator for all the algorithmic variants. For the collective library this involves writing all the aforementioned algorithms and filling out the algorithm index. In order to enter the collectives into the index one must carefully look at the various requirements and outline what the parameter space looks like and what are the restrictions on the algorithms. In addition to writing all the algorithms, performance models and heuristics must be derived and encoded to aid in traversing the search space. Thus far the dissertation has extensively focused on this component of the automatic tuning system and the various tradeoffs between the different algorithmic variants.

Library Installation

Once the library has been created, the next step is the installation process. Unlike a traditional software installation, there are some additional steps that are needed for an automatically tuned library. Compared to traditional software installations the library can take much longer to compile and use more disk space because of the number of algorithmic variants of the same function. Once the library and all the algorithms have been compiled the next step is to benchmark the results on the target system. Section 8.3.2 goes into much greater detail of how the benchmarking is done and how the results are stored. In addition to benchmarking the collectives themselves, microbenchmarks are used to set the parameters for the performance models thus enabling the performance models to give a useful feedback into what the search space looks like. The installation process can take as little as a few minutes if the benchmarking phases are skipped or a few hours if an exhaustive set of benchmarks are run.

Application Runtime

Once the library has been compiled and the benchmark data has been collected the library is ready to be used. Along with the algorithm index a second data structure is used to store the best performing algorithms and parameters. This data structure is indexed on the following *input tuple* to the collective: the number of nodes, the number of threads per node, the address mode, the synchronization flags, the properties of the reduction function (if applicable), and the size of the collective. The data structure, implemented as a series of linked lists, returns the best algorithm and parameters given the input tuple.

This lookup table can initially be empty or preloaded with defaults from the benchmarking phases and previous runs of the application. When a new input tuple that has not been added into this table is seen, the user is given one of two choices. They can either (1) invoke a search to find the best algorithm and parameter combination, which can be expensive, or (2) find the closest match to an existing input tuple and use the entry stored in that slot. In some cases such as the Address Mode and Synchronization modes the algorithmic choices can be fundamentally incompatible in which case a set of heuristic based safe defaults are

used. Details of how the search is done at application runtime are provided in Section 8.3.3 along with how the search space can be pruned with Performance models in Section 8.3.4.

8.3 Collective Tuning

Choosing the best collective algorithm amongst all the choices can be an expensive process. There are many factors that affect the overall performance of the library. A few of them are can be inferred at compile time however some of the other performance influence factors are only known during the actual execution of the application thus necessitating some search during application runtime. To yield the best algorithms a combination of offline tuning is used. We define these two steps as follows:

- **Offline Tuning:** Offline tuning is the process of tuning the collective library outside the users application. This can be done either at install time once the library has been built or periodically by the system administrator. Because the tuning step is outside the critical path of the library can spend more time can be spent finding and searching the best variants of the code. However since the exact collective input parameters are not known from the application a set of input tuples must be searched. The set of input tuples can be adjusted as needed to tailor specific applications but adjusting the input tuple space is a manual process. This is the tuning process 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 the applications. The advantage of this is that the complete information about 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 enough node count), any gains from the collective tuning might be overwhelmed by the cost of the tuning.

Thus 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 will be 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 rest of this section describes the factors that influence performance and how the tuning is done at each of the different stages.

8.3.1 Factors that Influence Performance

There are many factors that influence the optimal collective implementation but some of them are easy to infer such as the processor type and speed and others are much more difficult to measure or even model such as network load and the mix of computation and communication. To highlight the different factors we divide them into three categories: (1) static factors that can be inferred at install time, (2) factors known only at application runtime for which performance models can be constructed, and (3) factors known at application runtime that are difficult to construct a performance model for.

Install Time

During installation time there is a lot of salient information about the nodes' architecture that must be incorporated. Information such as the processor type and speed, the number of hardware execution contexts, the sizes of the caches and the performance of the memory system can be determined. In addition information about the interconnect amongst the various processor cores within a node will also be useful information in determining how best to use the threads. Along with the processors' information it is important to find out the type of network that is available including the network latencies and bandwidths if they are available. These will feed into the performance models. In some cases the platform information is enough to tell you the interconnect topology. If the library targets the BlueGene/P the runtime layer automatically knows that the underlying network is a 3D torus and must schedule the communication accordingly.

Run Time (easy to model)

Once the application is running more information will be available that can be fed back into the models. Information such as the number of threads and cores used for the application, the types of collectives that will be run, the sizes of messages involved and the required synchronization and address modes. This information can be fed back into the performance models to give a useful picture of what the search space looks like.

Run Time (hard to model)

There are some factors that are known only at runtime that are very difficult to construct models for. The first is how the collectives library will interact with the other computation involved, especially if aggressive overlap of collectives and the library is used [128]. The performance of the library and how attentive the processors are to the network affect the choice of algorithm. In addition, many modern computing clusters are not dedicated to particular users and jobs and are shared amongst many disjoint jobs. It is often the case that the nodes themselves are not shared but the network resources and bandwidths are shared. Thus inferring what other jobs are doing and react without any form of search is a very challenging task. For the factors in this category our hypothesis is a minimal search effort amongst collectives that might yield good performance results will yield the best performance.

8.3.2 Offline Tuning

During the offline tuning phase the main goal is to prune the entire algorithm and parameter space into a few candidate algorithms that will yield close to optimal performance and eliminate obviously bad candidate algorithms. A secondary goal is to also be able to assign a set of reasonable algorithmic defaults for various locations in the input tuple space so that no further tuning need be done at runtime to yield collectives that give close to optimal performance.

To perform the offline tuning we construct a dedicated microbenchmark that iterates through the tuple space, fixing the node and threads per node counts. In order to reduce the execution time, the sizes of the collectives explored are powers of two. For each iteration point an extensive search is done that yields the best algorithm. To reduce the search space further, we restrict the search to K-nary and K-nomial trees (see Section 5.1.2) with radices that are powers of two.

Once the benchmark is complete the data is stored in an XML file that will be read in during the application runtime. The iteration space can be augmented by running the benchmark on a different number of nodes. Thus over time the offline tuning data can grow to encompass a large portion of the input iteration space giving the users a much more high fidelity defaults from which to start the online tuning process. The parameters for the models can be set through the results of these runs or be set using a smaller dedicated benchmark. We chose to use the latter method.

8.3.3 Online Tuning

When the application starts it loads in the data and defaults collected from the offline tuning phase to assign some defaults. As mentioned above, all the tuning choices are stored in a lookup table. During the application runtime, when the input tuple is given it is looked up in the table. If the value already exists then that default is used. However, if the slot is empty there is a choice of invoking a search or using the closest match in the table. These decisions can happen either explicitly (the cost of tuning is exposed to the user and done outside of the critical path) or implicitly (the tuning is allowed to be done in the critical path without the user requesting it). GASNet supports both tuning models.

Explicit Tuning

In explicit tuning the cost of tuning is exposed to the user. The user invokes a tuning function with the expected input tuple for the collective. The runtime then invokes a search on that collective and then updates the lookup table with the best information. Thus for any subsequent match to that input tuple an entry in the table is guaranteed to be there. To properly tune collectives for nonblocking communication the user is also allowed to pass an optional function pointer that will be invoked after the collective is initialized and before the collective is synchronized. This is a more accurate representation of the usage of the collective.

The advantage of the explicit tuning is that it exposes the cost of the tuning to the user and ensures that the performance during the critical path is not prone to expensive search operations that might affect overall load balance. However, for many cases the sizes for the collectives change over the course of the run and thus it might be hard to predict the exact sizes to tune for.

Implicit Tuning

Using implicit tuning the application will call the collectives as it normally would presenting the input tuple to the tuning system which would then call the collective on its behalf. If the input tuple is new then a search is invoked and the data is stored. Thus without the application writer having to do any code modifications the collectives will be tuned. This is designed for cases in which there are a small number input tuples that are called many times. Thus the cost of tuning is incurred on the first collective but is amortized over the entire run. The application is also allowed to pass an optional flag indicating that the tuning should not occur.

Because the implicit collective tuning is done without the users knowledge it is important that the collective tuning be done as quickly as possible. Thus it is often useful to tradeoff tuning accuracy for tuning time. Section 8.3.4 will go into greater depth on analyzing this tradeoff.

8.3.4 Performance Models

Sections 5.3, 6.1.1, and 6.2.1 show how performance models can be constructed to map out the search space and understand the various factors that influence performance. In this section we discuss how the performance models are used to prune the search space and we show how they can dramatically reduce the time needed to find the optimal algorithms and parameters for different collectives.

Broadcast

The first collective we analyze is Broadcast. Figures 8.2, 8.3, and 8.4 show the costs of doing search guided by the performance models for an 8 byte Broadcast on the Sun Constellation, Cray XT4, and Cray XT5. The x-axis represents the number of algorithms that are searched. A value of 0 means that the result from the performance model is used without conducting any additional search. In order to guide the search the algorithms are sorted according to their predicted runtime by the performance models. Thus any value along the x-axis $x > 0$ implies that the top $x + 1$ algorithms are searched and the best is chosen. The left y-axis (associated with the blue line with crosses as the markers) shows the performance of choosing an algorithm in this pruned search compared to the best algorithm using exhaustive search. The values shown are the best time found through an exhaustive search divided by the best time found through the pruned search space (i.e. ratios of inverse latencies). A value of 0.75, for example, means that the best algorithm runs in 75% of the time taken for the algorithm found through the pruned search space. Thus once all algorithms are searched over this value will eventually converge to 1.0. The number of algorithms that need to be searched before converging to 1.0 is in an indication of how well the performance models are able to map out the search space. The value on the right y-axis (associated with the red line with triangles as the markers) shows the amount of time spent tuning compared to doing a full exhaustive search. Thus at 0 this value is 0 since we have spent no time doing search since it was a quick calculation with the performance

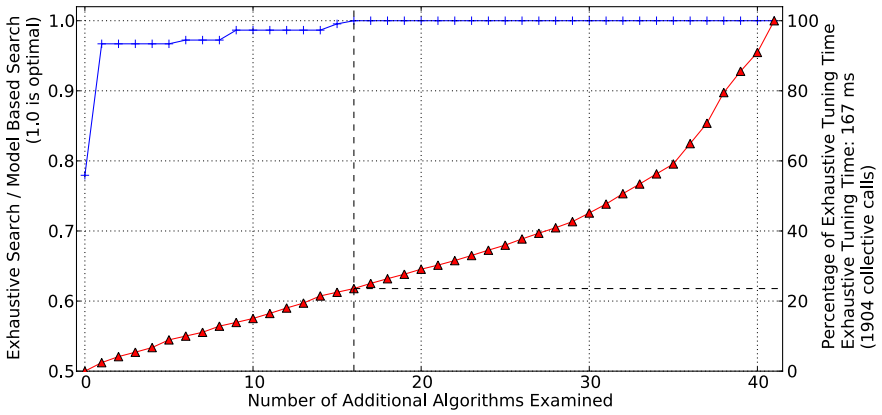


Figure 8.2: Guided Search using Performance Models: 8-byte Broadcast (1024 cores Sun Constellation)

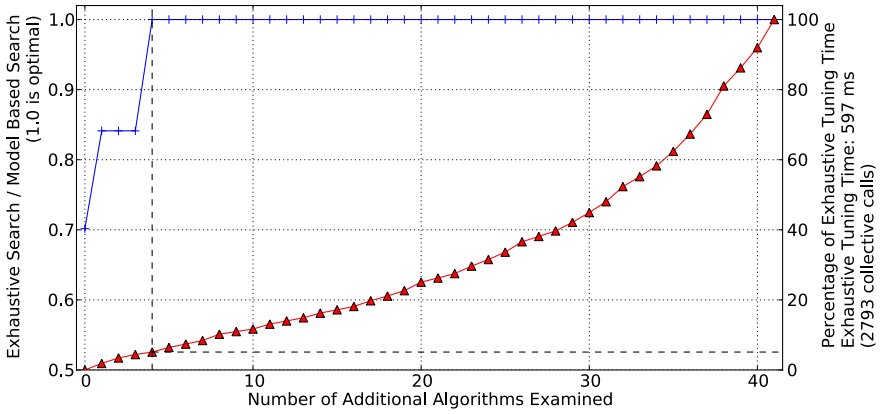


Figure 8.3: Guided Search using Performance Models: 8-byte Broadcast (2048 cores Cray XT4)

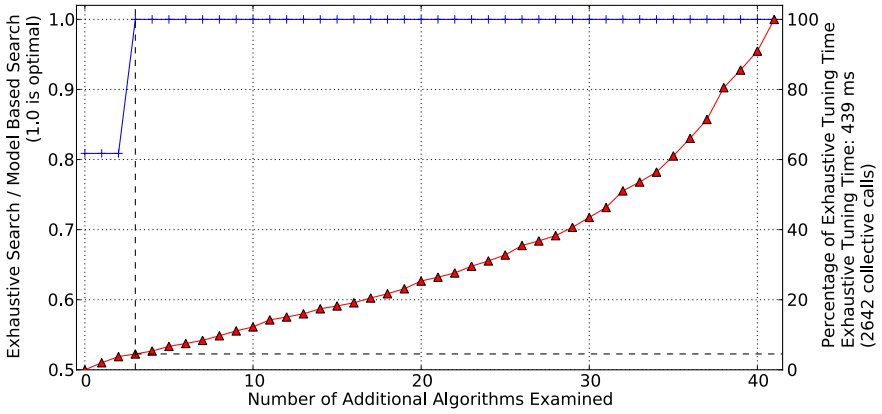


Figure 8.4: Guided Search using Performance Models: 8-byte Broadcast (3072 cores Cray XT5)

model. However each additional algorithm takes time to search. In addition, the plots have been annotated with the time required for an exhaustive search. This leads to a tradeoff between the time spent to search and quality of the resultant algorithm. The idea of trading off the time taken for search with the quality of the resultant algorithm is not a new one. FFTW [85] also successfully employs such heuristics and exposes them to the end user.

The data from Figure 8.2 show that on the Sun Constellation, using the performance model alone for an 8 byte Broadcast will yield an algorithm that takes 1.28 × the optimal solution (i.e. the best algorithm runs in 77% of the chosen algorithm). However by adding just one additional algorithm to the search (so search over the best of two algorithms) can lead to an algorithmic choice that is within 4% of the best. However notice that searching over 17 algorithms is needed to find the optimal solution. Since we have sorted the algorithms based on the performance model, the algorithms that we expect to yield obviously bad performance are placed at the end of the search list. Therefore searching over 17 algorithms (or 40% of the total number of algorithms) takes about a quarter of the time needed for an exhaustive search. If the models were to improve then the search time can be reduced even further.

Figure 8.3 shows the same data collected on the 1024 cores of the Cray XT4. As the data show, using just the performance model yields an algorithm that gets 1.43 × the optimal. However, by searching over just 4 algorithms one finds the global optimal in less than 8% of the time needed for an exhaustive search. Figure 8.4 shows the same data collected on 3072 cores of the Cray XT5. Like the Cray XT4 using the model alone yields a suboptimal algorithm, however searching amongst 4 algorithms or less than 5% of the time needed for an exhaustive search will yield the best algorithm. For both platforms even though the performance model has not accurately predicted the best algorithm, it is able to highlight the algorithms that are likely to succeed such that the best one can be found by searching just a handful of them.

Scatter

Figures 8.5, 8.6, 8.7 show the performance of a 128-byte Scatter on the 1024 cores of the Sun Constellation, 512 cores of the Cray XT4 and 1536 cores of the Cray XT5. Notice that unlike the Broadcast, the three platforms have a different number of algorithms that are included in the search. This difference arises from the fact that some of the Scatter algorithms are not applicable on all platforms due to the limitations in the scratch space size and maximum payloads in the active messages. Thus the automatic tuner will skip over these algorithms in the search.

As the data show in Figure 8.5 on the Sun Constellation, the model yields an algorithm that gives a performance that is within 10% of the best. However, an additional 11 algorithms must be searched (or 32% of the time for an exhaustive search). The results in Figure 8.6 on the Cray XT4 show similar results. The initial performance model yields an algorithm that is within 10% of the best but a further 15 algorithms need to be searched. However, on the Cray XT4, searching over the last 12 algorithms takes about 60% of the overall search time. Thus the poorest performing algorithms (which are accurately placed at the end of the search space) take the bulk of the time for the search on the platform.

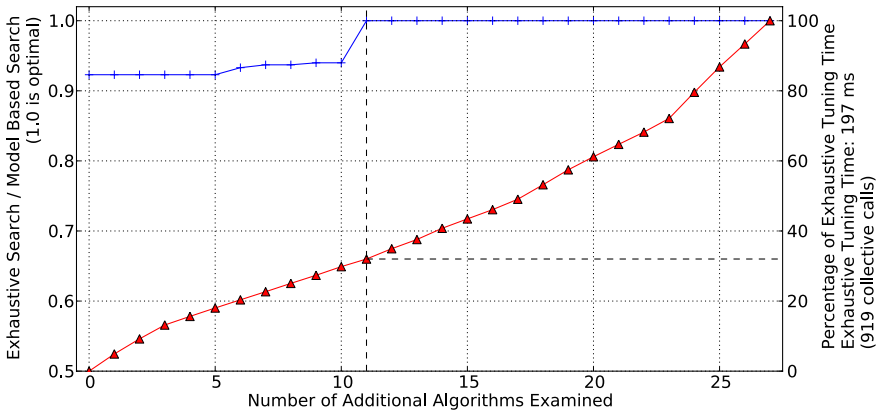


Figure 8.5: Guided Search using Performance Models: 128-byte Scatter (1024 cores Sun Constellation)

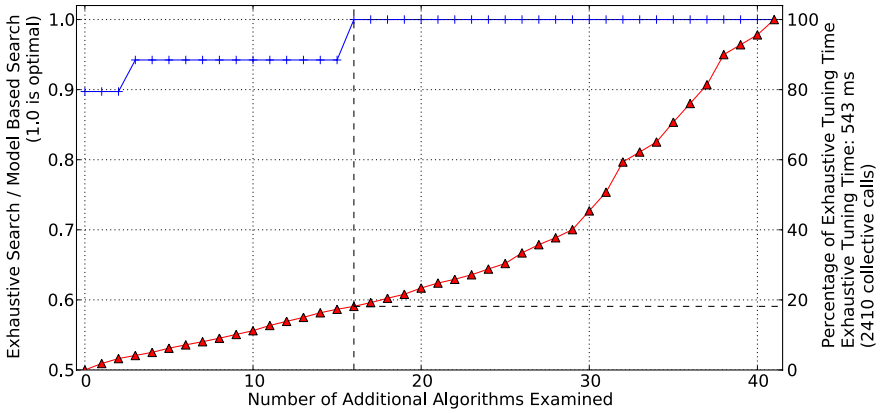


Figure 8.6: Guided Search using Performance Models: 128-byte Scatter (512 cores Cray XT4)

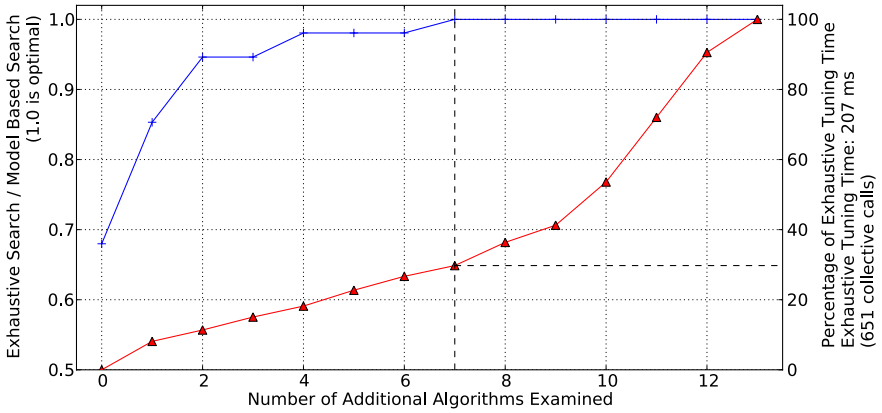


Figure 8.7: Guided Search using Performance Models: 128-byte Scatter (1536 cores Cray XT5)

This steep rise at the end indicates that the difference between the best performing and worst performing algorithms is quite high further arguing the case for intelligently pruning the search space.

On the Cray XT5 the results show that the performance models yield a poorer performing algorithm. The best algorithm can run in just under 70% of the time needed for the algorithm chosen by the models. By adding two algorithms we can get within 10% of the best and a search of 7 algorithms (half the total number of algorithms) is sufficient to find the best one. Similar to the Cray XT4, the last 5 algorithms take more than 60% of the time for an exhaustive search. As the data also show, for Scatter some amount of search is needed to produce a good algorithm.

Exchange

Finally we analyze the performance of Exchange on 1024 cores of the Sun Constellation, 512 cores of the Cray XT4 and 1536 cores of the Cray XT5 in Figures 8.8, 8.9 and 8.10. For the Sun Constellation and the Cray XT4 an 8 byte Exchange was used while on the Cray XT5 a 64-byte exchange was used. Our results showed that for a 64 byte Exchange on the Sun Constellation and the CrayXT4, the performance model picked the best algorithm thus negating the need for search. Similarly the performance model also picked the best algorithm for the 8 byte Exchange on the CrayXT5. Thus for the sake of brevity we show the interesting cases in which a combination of search and performance models were needed.

As the data from both the Sun Constellation show the performance model yields an algorithm that takes about 1.25 times as long to run as the best one. Searching three algorithms is enough to find the best one. Like Scatter, notice that the last two algorithms monopolize more than 90% of the time needed for an exhaustive search and thus by eliminating these candidates the time needed for a search is much more tractable. From Section 6.1 the Flat algorithms, which send $O(N^2)$ messages instead of $O(N\ lg\ N)$, take significantly longer for small message sizes and thus are accurately placed at the end of the search space. On the Cray XT4 there is a similar trend.

On the CrayXT5 for a 64 byte Exchange notice that the model yields an algorithm that is within 10% of the best, the model does not do well at accurately mapping the search space. The best performer is the second to last algorithm to be searched, requiring us to incur more than 80% of the time needed for an exhaustive search to find the best algorithm. This is a good example of when leveraging information from the end user of about what level of performance is "good enough" is useful in pruning the search space. Our hypothesis is that the performance models do not accurately model the underlying network topology and thus the models need to be further refined. Future work will address this issue.

8.4 Summary

In Chapters 5, 6, and 7 many different algorithms are presented to perform the collectives. Due to the large algorithmic space it is infeasible to hand tune and pick algorithms for all current and future platforms. Therefore we have constructed an extensible automatic tuning

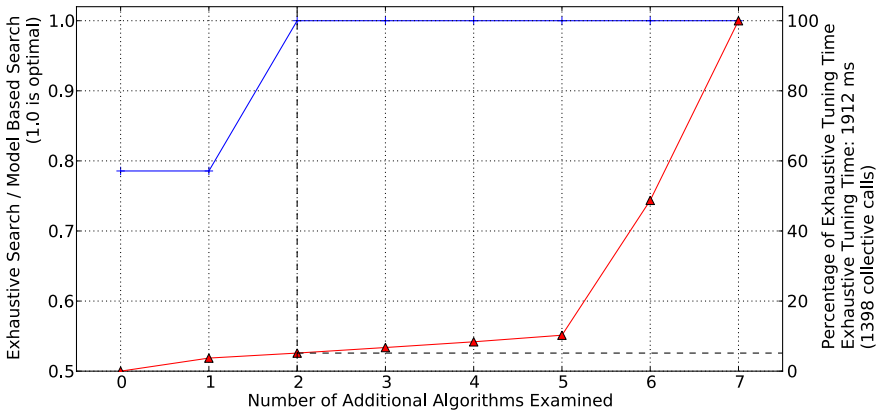


Figure 8.8: Guided Search using Performance Models: 8-byte Exchange (1024 cores Sun Constellation)

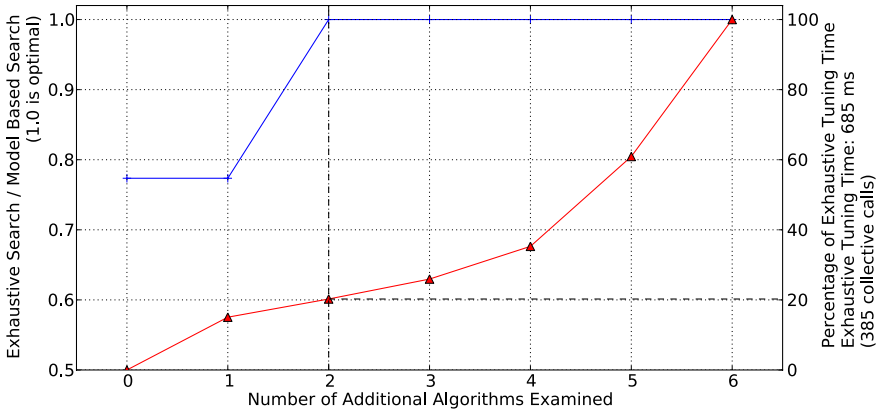


Figure 8.9: Guided Search using Performance Models: 8-byte Exchange (512 cores Cray XT4)

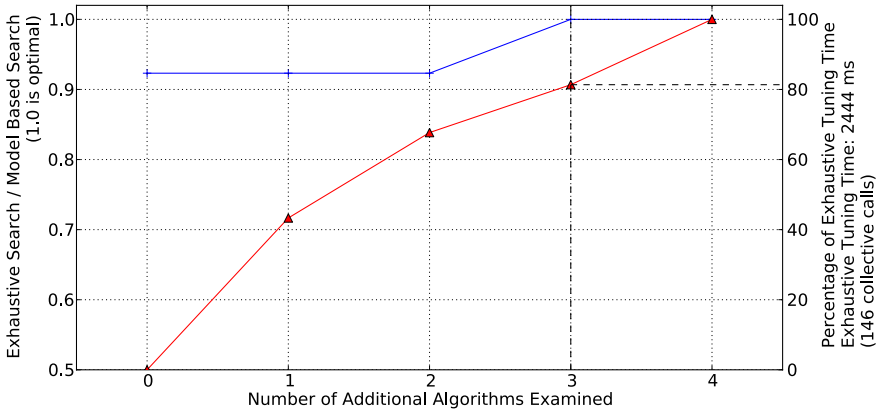


Figure 8.10: Guided Search using Performance Models: 64-byte Exchange (1536 cores Cray XT5)

system whose goal is to provide high performance collectives on a wide variety of platforms. The automatic tuner is broken up into three components: constructing the algorithms and performance models, offline compilation and benchmarking, and online analysis and search. With these components working together we are able to realize portable performance on a wide range of platforms. The automatic tuner also allows tuning data to be saved and restored across different runs so that the tuning information is not lost after the program is complete. In addition, our automatic tuning system is extensible so that new algorithmic or hardware innovations can be added into the search space.

We further showed that a combination of search and performance models is needed to find the best algorithm. The performance models presented in earlier chapters do a good job of mapping out the search space so that with minimal search effort he best algorithms can be found. In most of the cases, the performance models accurately place the worst performing algorithms at the end of the search list so that algorithms that would take the bulk of the time for search are skipped in favor of algorithms that are the likeliest to yield the best results. In many cases the performance models are able to pick the best algorithm and thus negating the need for search. In a lot of our experimental cases, the models pick an algorithm that is already within 10% of the best performance. When search is required the automatic tuner is able to pick the best algorithm within 25% of the time needed for an exhaustive search in most cases. Thus as the data show, our automatic tuning system that builds on previous automatic tuning efforts can deliver portable performance and minimize the time to yield the best algorithm by intelligently navigating the search space.

Chapter 9

Teams

Thus far our discussion has centered on collectives that assume *all* of the threads in the application take part in the collective. However, as we have see with our example applications (Dense Linear Algebra in Section 5.4 and 3D FFTs in Section 6.3) it is often useful to run the collectives over a subset of the processors. In many cases rather than having many large collectives with all T threads, they may run c collectives of T/c threads each. In another usage model, only some of the threads participate in a collective while others perform other tasks. Thus it is essential to build an interface that can construct these different teams of processors. MPI calls these teams subcommunicators however we chose a different name to emphasize that the teams can be constructed differently than MPI and do not necessarily need the full overhead of a communicator in MPI (although if the runtime system requires it they can be built with this overhead).

Our discussion will center around two unique ways to create teams: a thread-centric approach in which the teams are explicitly constructed based on thread identifiers (Section 9.1) and a second more novel data centric approach in which the teams are constructed based on the data that is involved in the collective (Section 9.2). The latter approach relies on the shared arrays found in UPC.

9.1 Thread-Centric Collectives

A common way to construct teams is by identifying the threads within a team. Team construction is hierarchical; the parent of a particular team contains a superset of the members of that team. When the runtime system starts up a primordial team (e.g. `MPI_COMM_WORLD` in MPI[1]). In order to construct a team *all* the threads of the parent team must call into the team construction. The members of the newly constructed team will get a handle to the team while the others will receive an invalid return object. Currently GASNet also implements a prototype of this interface.

There are many different ways to construct these teams. One popular method is a split operation with the following signature: `team_split(team_t parent, int color,`

[1]In a related proposal we have proposed that a similar `UPC_TEAM_ALL` be added to the language specification.

`int rank)`. The team split operation takes a parent as argument and the color and rank as arguments. All threads that call with the same `color` argument will be part of the same subteam with a relative rank of `rank` within that team. Thus for example if all threads call the team creation with (MYTHREAD is the current thread's identifier on in the global team and THREADS is the total number of threads in the global team):

```
team_split(UPC_TEAM_ALL, MYTHREAD%2, MYTHREAD/2);
```

In this example all the even threads will be part of one team and all the odd threads will be part of another. In addition threads 0 and 1 will be relative rank 0 on their new subteams and so forth. Thus with one call we are able to partition the threads into their pieces. This is the method that is needed for all the thread creation needed for our example applications.

Another popular method is by defining teams through *groups*. A group is a lightweight set of ordered set of thread identifiers. Simple set operations such as union, intersect, include, and exclude allow quick construction of these sets which can then be used to construct the team. Again all threads from the parent team are required to participate in the team construction and all the threads are required to pass consistent groups.

9.1.1 Similarities and Differences with MPI

The problem of communication with a subset of the processors is certainly not a new one and has been around in the MPI specification for many years. In MPI parlance the notion of teams is expressed as communicators. Many of the ideas presented in the API for MPI communicators are orthogonal to their use of two-sided communication and their design of the collectives and thus a lot of these ideas are applicable here. However, there are some important differences that will need to be addressed as well.

Similarities

- **Communicators:** A team, like a communicator, is a set of GASNet images along with a preallocated set of buffer space and communication meta data (e.g. a distributed buffer space manager) to allow fast collective communication across the various members of the team.

- **Groups:** We will also adopt the idea of an MPI Group into GASNet. The primary function of a GASNet group is to allow easy team construction. The process of creating a team is an expensive process due to the setup of all the required meta-data. Since a group is merely an ordered set of images without the additional meta data necessary for communication and synchronization their construction and modification can be done using much simpler, and therefore more efficient, operations. Thus the general model, like MPI, will be to build up a group based on the different operations provided in the API and then construct a team around a group once the group has been finalized.

Differences

- **GASNet Images:** The biggest difference (especially in the implementation) that needs to be addressed is the difference between a node and threads within that node (see Section 5.1.1) and how these relate to the teams. MPI does not have this problem since there is no notion of a hierarchical structure between MPI tasks; they are all at the same level.

- **Scratch Space:** GASNet has an explicit segment that allows one to take advantage of some very important communication optimizations (e.g. RDMA) in one-sided operations (see Section 5.2.1). Since we wish to leverage some of these same optimizations in the collectives, the collectives (and thus teams) will need to reclaim some of the space that has been allocated to the GASNet client to provide the best possible performance. The auxiliary scratch space for the initial GASNET_TEAM_ALL will be handled directly within GASNet so that it is not visible to the end user, however further team construction will necessitate the GASNet client explicitly managing these buffers.

- **Usage:** Another important and distinct difference is that MPI allows one to use a communicator for isolation of point-to-point messaging operations such as sends and receives. GASNet is a one-sided communication system that lacks such two-sided message passing operations, and the teams are not relevant for one-sided point-to-point communication. GASNet teams are only relevant for use in the collectives library. Thus each image will have one globally unique name for point-to-point communication. The translation routines that are provided so that one can specify the root for rooted collectives relative to the other images in the team.

Current Status

GASNet currently implements a version of thread-centric collectives across the nodes using the team_split construct described above. At the time of writing of this dissertation, an interface for teams for UPC has not been decided on, therefore, until this has been discussed and incorporated into the language further development has been left as future work to ensure that time is not wasted on an implementation that is not useful.

9.2 Data-Centric Collectives

A drawback of the thread centric teams in Partition Global Address Space languages is that they seem to force a programming model that is applicable to MPI's tasks but it is somewhat awkward when the languages provide the ability to declare large shared arrays. Since the PGAS languages explicitly expose the non-uniform nature of memory access times to the memory of different processors, operations on local data (i.e. the portion of the address space that a particular processor has affinity to) will tend to be much faster than operations on remote data (i.e. any part of the shared data space that a processor does not have affinity

to). Thus, unlike traditional shared memory programming, the languages necessitate global data re-localization operations in order to improve performance that will be served by the collective communication operations.

Since UPC emphasizes global shared arrays as the primary constructs for parallel programming, our goal is to make the collectives use a *data-centric* model rather than the thread-centric model employed by many other parallel programming models.

9.2.1 Proposed Collective Model

Our proposed data-centric collective extensions would allow the users to specify the blocks of data involved and let the underlying runtime system handle the problem of mapping data blocks to threads and packing the data into contiguous blocks. Since the runtime system already has full knowledge of how the shared arrays are mapped onto the threads, the overhead of obtaining this information is relatively small.

For efficiency reasons we require that all threads that have affinity to one of the active blocks of data to make a call into the collective, similar to the current MPI and UPC collective models. If a thread that does not have affinity to any of the data involved in the call, the operation is treated like a no-op. This allows us to build a scalable implementation of the communication schedule by letting the collective build a tree over all threads participating in the collective and leveraging all the techniques discussed in Chapters 5, 6, and 7.

Seidel et al. have also proposed an alternative model for collectives in UPC [146] which would require only one thread to handle the data movement for all the threads involved in the collective, without the need for any of the other threads to participate in the collective. However, this forces the implementations to either (1) always use flat trees, which severely hinders scalability at large processor counts, or (2) have an auxiliary thread on each node that is not part of the runtime that handles the collective communication responsibilities for that node. Setting aside the performance implications of devoting an extra thread to handle the collectives on machines with few cores per node, synchronization of these collectives becomes an issue. The runtime system can not infer that a collective is going to be active within a given barrier phase and therefore the programmer has to either explicitly handle the synchronization, which could get cumbersome, or wait until the next barrier phase to use the data, which could cause over-synchronization and lead to performance penalties. Due to the performance and productivity disadvantages of such an approach we decided to employ a model in which all threads with affinity to data involved in a collective make an explicit call to the collective.

9.2.2 An Example Interface

We are less concerned with the exact formal specification of the collectives in the language. Our goal is to demonstrate their usefulness, and consider syntax to be outside the scope of this dissertation. Future work will formalize the definitions and incorporate them into the UPC language.

We use a Matlab [120] and SciPy [106] style interval notation to specify blocks of data

in each dimension that will be used in the collective. Intervals in each dimension of the array are specified by the first index, the distance between successive indices, and the last element. To specify only the even elements in a UPC shared array, for example, we propose the following notation:

```
shared int A[10];
A<0:2:9>
```

For multidimensional arrays, one can pass a list of intervals (one for each dimension). The application experience in using these collectives motivated the specification of block indices in the interval rather than the array indices themselves. However this decision is not fundamental to the interface. Our interface adheres to the current UPC collective synchronization specification.

To motivate the interface we will discuss one example in each of the two collective categories: (1) one-to-many (e.g., broadcast) and (2) many-to-many (e.g., exchange[2]). Section 9.2.3 goes into further detail on how these operations can be incorporated into real applications.

- **One-To-Many**
  In the first category of collectives, one root block contains the data to be disseminated to other blocks of the shared array. Common collectives in this category include `broadcast()` and `scatter()`. Typical scalable implementations of these operations construct a tree over the threads rooted on the thread that owns the original data. In our interface (as well as the current UPC collective specification), the user specifies a shared pointer rather than explicitly specifying the root thread.

  In addition, we allow the user to specify a list of intervals to the destination which dictate which blocks the broadcast data will be stored into. The number of intervals is dictated by the number of dimensions of the shared array. The proposed prototype for this type of collective is:

  ```
  upc_stride_broadcast(shared void* dst<intervals>,
                       shared void* src,
                       size_t len, int sync_flags);
  ```

  The example in Figure 9.1(a) declares a two-dimensional destination array. The `src` array broadcast the data into every other row and column of the `dst` matrix.

- **Many-To-Many**
  In the next important category of collective operations, every output block involves data from *all* the input blocks. The input blocks are likely to be distributed across many processors. Scalable implementations of these collectives carefully tune the communication schedule to avoid creating hot spots in the network, however the performance is often limited by the bisection bandwidth. Many methods[44] also exist for performing the communication in $O(N\ log\ N)$ rounds rather than $O(N^2)$ rounds.

[2]In MPI parlance this operation is `MPI_Alltoall()`

```
shared [] int src[4];
shared [4][4] int dst[200][200];

upc_stride_broadcast(
dst<0:2:49,0:2:49>, src,
sizeof(int)*4, 0);
```

(a) A code-snippet to perform a broadcast `src` to every other row and column of the `dst` matrix

```
shared [10][10] int src[100][100];
shared [10][10] int dst[100][100];

upc_stride_exchange(dst<0,:>,
src<:,0>, sizeof(int), 0);
```

(b) A code-snippet to exchange data from the first column of `src` into the first row of `dst`

Figure 9.1: Strided Collective Examples

A popular example of a collective in this category is `exchange()`. This collective breaks each block in the input array into k pieces of len bytes each. It is assumed that there are k blocks specified in the each of the source and destination intervals. It then takes the i^{th} slot from block j and places it into the j^{th} slot in block i on the destination. The following prototype illustrates our proposed interface:

```
upc_stride_exchange(shared void* dst<intervals>,
                    shared void* src<intervals>,
                    size_t len, int sync_flags);
```

Figure 9.1(b) shows an example of an exchange operation. In the example all the blocks in the first column of the source matrix are exchanged into the first row of the destination matrix.

Notice that in neither the definition nor the example collectives interface did we require the user to specify the identity or number of threads involved in the communication. The threads involved are implicitly defined by specifying which blocks of data the collective is to be run across. Since there is no explicit mention of how many blocks a particular thread owns, it is up to the implementation to infer this information and make the correct decisions on how to correctly pack the data. Allowing the runtime to make such decisions allows for much greater performance portability.

However, since we do require all the threads with affinity to any part of the memory being communicated call the collective we provide a simple utility function that can query whether the calling thread has affinity to *any* part of the data. Since this information is already stored inside the runtime system such query functions will be fast.

```
int upc_haveaffinity(shared void* arr<intervals>);
```

The function will return a nonzero value if the calling thread has affinity to any of the data in the specified interval or 0 otherwise.

Multiblocked Arrays

Related work by Almasi et al. [23] has shown how to construct multidimensional arrays in UPC which can be naturally extended to having the teams work on them.

The work addresses these problems by means of two techniques. First, it allows the programmer to specify a Cartesian processor distribution for a UPC array. This roughly corresponds to Cartesian topologies in MPI: an ability to denote threads with a tuple $< t_0, t_1, ...t_n >$ instead of a single number $t, 0 \leq t < THREADS$. This has been done by other languages, such as HPF[95] and ZPL[48]. It proposes syntax similar to HPF, in which processor mappings are named and shared arrays are mapped to these distributions. E.g.:

```
#pragma processors MyDistribution(10, 10)
shared [B1][B2] (MyDistribution) int A[N1][N2];
```

The distribution directive above establishes a 10×10 Cartesian distribution, and array A is declared to be of that distribution. The system verifies at runtime whether the distribution is legal and ignores it if it does not match the current running configuration (e.g. not enough running threads to fill up a 10×10 distribution).

Thus as we will show, the data centric collectives and multiblocked arrays can be combined to quickly express complicated data layout and data movement operations.

9.2.3 Application Examples

Thus far we have motivated the use of the global shared arrays and a new collective interface that takes advantage of these arrays. In this section we show how three common and important computational kernels can be written very succinctly in UPC with our proposed additions.

Dense Matrix Multiplication and Dense Cholesky Factorization

The UPC code for matrix multiplication is shown in Figure 9.2 while the Cholesky factorization example can be found in Figure 9.3. Lines 1 to 3 of Figure 9.2 declare the dense matrices with the specified blocksizes and partitioned according to the mapping specified above. Notice that with one simple declaration that UPC offers, the entire matrix is load balanced in the optimum checkerboard pattern. Such a task in MPI is much more cumbersome since the matrix block to processor mapping has to be controlled by the application writer rather than the runtime system. We then allocate a set of scratch arrays in Line 3 that will be used for intermediate results. We iterate through the blocks of the matrix as one would do in a standard blocked implementation of the kernel. Notice all the broadcasts in one dimension occur simultaneously and each processor is only responsible for specifying the portion of the data that it owns.

Three Dimensional Fourier Transform

Figure 9.4 shows the UPC code to implement these operations through data-centric collectives. For sake of brevity, we assume that the cube is $NX \times NY \times NZ$ and that there

```
1. #pragma processors rect(Tx,Ty)
2. shared [b][b] (rect) double A[M][P], B[P][N], C[M][N];
3. shared [b][b] (rect) double scratchA[b*Tx][b*Ty],
                              scratchB[b*Tx][b*Ty];
4. double alpha=1.0, beta=1.0;
5. int myrow=MYTHREAD/Ty; int mycol=MYTHREAD%Ty;
6. for(k=0; k<P; k+=b) {
7.   for(i=0; i<M; i+=Tx*b) {
       /*broadcast across the rows*/
8.     upc_stride_broadcast(scratchA<myrow,:>, &A[i+myrow*b][k],
                            sizeof(double)*b*b, UPC_OUT_MYSYNC);
9.     for(j=0; j<N; j+=Ty*b) {
         /*broadcast down the columns*/
10.      upc_stride_broadcast(scratchB<:,mycol>, &B[k][j+mycol*b],
                               sizeof(double)*b*b, UPC_OUT_MYSYNC);
         /* matmult*/
11.      dgemm('T', 'T', &b, &b, &b, &alpha,
               (double*) &scratchA[myrow*b][mycol*b], &b,
               (double*) &scratchB[myrow*b][mycol*b], &b, &beta,
               (double*) &C[i+myrow*b][j+myrow*b], &b);
12.    }
13.  }
14.}
```

Figure 9.2: UPC Code for Dense Matrix Multiply

are $NX \times TY$ threads. The calls to `fft()` are calls to high-performance FFT libraries such as ESSL [76] or FFTW which provide the appropriate interface.

Observations

We use these benchmarks as a case study to explore the effectiveness of the interface and examine how three of the major hurdles to efficient and scalable parallel programming are addressed.

- **Data distribution and load balancing** In our examples, the user specifies high level properties of how the data should be laid out across processors. For example in the case of the matrix multiply and Cholesky factorization, the user is responsible for specifying the granularity of the checkerboard. In the case of the FFT, the user specifies the TY and NX dimensions to dictate how many processors are involved in each of the exchange rounds. However, notice that once the data distribution directives are given, access to the arrays is straight forward. Mapping the data distribution and array indices to the processors is left to the runtime.

```
1.  #pragma processors rect(Tx,Ty)
2.  shared [b][b] (rect) double A[M][M];
3.  shared [b][b] (rect) double scratchA[Tx*b][Ty*b],
                                scratchB[Tx*b][Ty*b];
4.  int i, j, k;
5.  double alpha=1.0;
6.  int myrow = MYTHREAD/Ty; int mycol = MYTHREAD%Ty;
7.  for(k=0; k<M; k+=b) {
8.    /* dense cholesky on upper left block */
9.    if(upc_threadof(&A[k][k]) == MYTHREAD)
10.      dpotrf('U', &b, (double*) &A[k][k], &b);

11.   /* triangular solve across top row of matrix */
12.   int proc_row = upc_threadof(&A[k][k])/TY;
13.   upc_stride_broadcast(scratchA<proc_row, :>,
                           &A[k][k], sizeof(double)*b*b, 0);
14.   for(j=k+b; j<M; j+=b) {
15.     if(upc_threadof(&A[k][j]) == MYTHREAD)
16.       dtrsm('L', 'U', 'T', 'T', &b, &b,
                &alpha, (double*) &scratchA[myrow*b][mycol*b], &b,
                (double*) &A[k][j], &b);
17    }
18.   /*update (outer product on upper triangular part)*/
19.   for(i=k+b; i<M; i+=Tx*b) {
20.     for(ti=i; ti<i+(Tx*b); ti++)
21.       upc_stride_broadcast(scratchA<(ti/b)%TX,:>,
                               &A[k][ti], sizeof(double)*b*b, 0);
22.     for(j=i; j<M; j+=Ty*b) {
23.       for(tj=j; tj<j+(Ty*b); tj++)
24.         upc_stride_broadcast(scratchB<:,(tj/b)%TY>,
                                 &A[k][tj], sizeof(double)*b*b, 0);
25.       dgemm('T', 'T', &b, &b, &b, &alpha,
                (double*) &scratchA[myrow*b][mycol*b], &b,
                (double*) &scratchB[myrow*b][mycol*b], &b, &alpha,
                (double*) &C[i+myrow*b][j+myrow*b], &b);
26.     }
27.   }
28. }
```

Figure 9.3: UPC Code for Dense Cholesky Factorization

```
1. void fft(complex_t *out, complex_t *in, int len, int howmany,
2.          int instride, int indist, int outstride, int outdist);
3. void main(int argc, char **argv) {
4. #pragma processors (rect)(NX,TY,1)
5.  shared [1][NY/TY][] (rect) complex_t A[NX][NY][NZ], B[NX][NY][NZ];
6.  int myplane = MYTHREAD/TY; int myrow = MYTHREAD%TY;
7.  complex_t *myA = (complex_t*) &A[myplane][myrow*(NY/TY)][0];
8.  complex_t *myB = (complex_t*) &B[myplane][myrow*(NY/TY)][0];
9.  initialize_input_array(A);
10. upc_barrier;
11. fft(myB, myA, NZ, NY/TY, 1, NZ, NY/TY, 1);
12. upc_stride_exchange(A<myplane,:,0>,
                        B<myplane,:,0>,
                        sizeof(complex_t)*(NY*NZ)/(TY*TY), 0);
13. local_transpose(myB, myA, NX, NY, NZ, Ty);
14. fft(myA, myB, NY, NZ/TY, 1, NY, NZ/TY, 1);
15. upc_stride_exchange(B<:,myrow,0>,
                        A<:,myrow,0>,
                        sizeof(complex_t)*(NZ/NX)*(NY/TY), 0);
16. fft(myA, myB, NX, (NZ/NX)*(NY/TY), (NZ/NX)*(NY/TY), 1, 1, NX);
17. upc_barrier;
18.}
```

Figure 9.4: UPC Code for Parallel 3D FFT

- **Constructing an efficient and scalable communication schedule between the processors** After distributing the problem across the processors it is important that these processors work together and communicate as efficiently as possible. Therefore we wrote the three benchmarks with collectives rather than manually controlling the communication. This allows the runtime layer to handle the communication more efficiently by using network features, such as Active Messages, that were unavailable to the UPC programmer. By passing the responsibility of tuning the communication schedule to the runtime layer through a clean interface, the user absolves himself from having to worry about the painstaking task of tuning communication. As we have demonstrated throughout the rest of this dissertation, collective tuning for a variety of architectures is a non-trivial problem and very dependent on the specifics of the target system.

- **Efficient serial computational performance** Once the data has effectively been distributed and communicated the last piece that remains is to perform the serial computation. Serial tuning for many of the popular serial computational kernels have been well studied and serial libraries such as ESSL, FFTW, ATLAS, and OSKI have been well tuned (either by hand or automatically) on many architectures. Any parallel programming language must allow for easy ways to leverage this work to realize optimal serial performance. In our implementations, the data movement is handled in UPC while the computation is handled through optimized serial libraries.

Software libraries, such as PETSc[21] and ScaLAPACK[77], alleviate some of the pain of data distribution and coordinating communication by providing an extensive API to handle these operations. We did not perform a line count amongst the different implementations because the widely available software libraries handle many corner cases that our prototype implementations do not and we do not consider it a fair comparison. However, introducing these features at the language level allows for more expressivity than a library can provide. In order to minimize the complexity of the interface, a library writer must try to keep the interface very simple by making an educated guess about which data layouts to support and which data layouts to omit. By contrast, when these data layouts are incorporated into a language, a simple grammar can lead to a much more rich set of data layouts that are infeasible to efficiently provide at the library level.

By allowing the user to only specify high level language directives regarding the data distribution, letting the runtime handle the details of the communication and allowing the user to use pre-existing libraries we can dramatically reduce the number of lines required to program common and important kernels.

9.3 Automatic Tuning with Teams

Once the teams have been created all the work with automatic tuning shown in Chapter 8 will be critical to delivering performance for the collectives that work on a subset of the teams. Unlike collectives that operate over the threads, collectives that operate on a subset

of the threads will have to be aware of the interaction of collectives across various teams. Thus blindly using the algorithms that work well for the global team might not work well on the subteams. In addition the optimal algorithm might depend on which subset of the processors are involved in the communication which is not known until the team is created. Thus to ensure that the collectives are properly tuned for the teams, the data structures and tuning mechanisms that manage the automatic tuner are attached to the teams so each team will in effect have it's own automatic tuner that makes the decisions for that team. The two different team construction methods will have different requirements for these automatic tuning systems.

- **Thread-Centric Teams:** Thread centric team construction provides a distinct point in the code in which the teams are created and the team handles are explicitly handled. Thus extending the automatic tuning to such an interface is fairly straightforward. When the team is created some initial tuning and performance model creation can be done to minimize the search time when the collectives are finally done.
- **Data-Centric Teams:** With data-centric teams the meta-data for the teams is hidden from the users and stored in the runtime. Under the assumption that the data blocks involved in the collective are not constantly changing, leveraging automatic tuning for data-centric collectives will yield fruitful results again assuming the cost of tuning can be amortized over the run. However, since the team creation is not exposed to the user, spending too much time tuning can lead to undesirable consequences. Thus such an interface would rely much more heavily on the performance models and a minimal amount of search to find the best algorithms. In addition, the teams constructed based on data blocks should be added to a caching system so that the implicit teams can be reused. This would make the runtime system slightly more complex, but having the performance models to yield good initial performance makes delivering good performance tractable.

As an example we analyze the performance of an 8-byte broadcast on 256 cores of the Sun Constellation in Figure 9.5. Recall that this system has 16 threads in one compute node and we organize the threads in a 16 x 16 square grid. The threads have been placed on the nodes such that threads 0-15 are on one node, 16-32 on the next, etc. The teams have been constructed with the team split command as follows:

```
myrow = MYTHREAD/16;
mycol = MYTHREAD%16;
row_team = team_split(GASNET_TEAM_ALL, myrow, mycol)
col_team = team_split(GASNET_TEAM_ALL, mycol, myrow)
```

Thus the row teams will broadcast data within one node only without sending data across the network. In the column teams, all the communication will be outside of the node and across the network. In this case all 16 threads will simultaneously be using the network card. For completeness we also show the performance of the broadcast on `GASNET_TEAM_ALL`. All row broadcasts happen simultaneously and all the column broadcasts

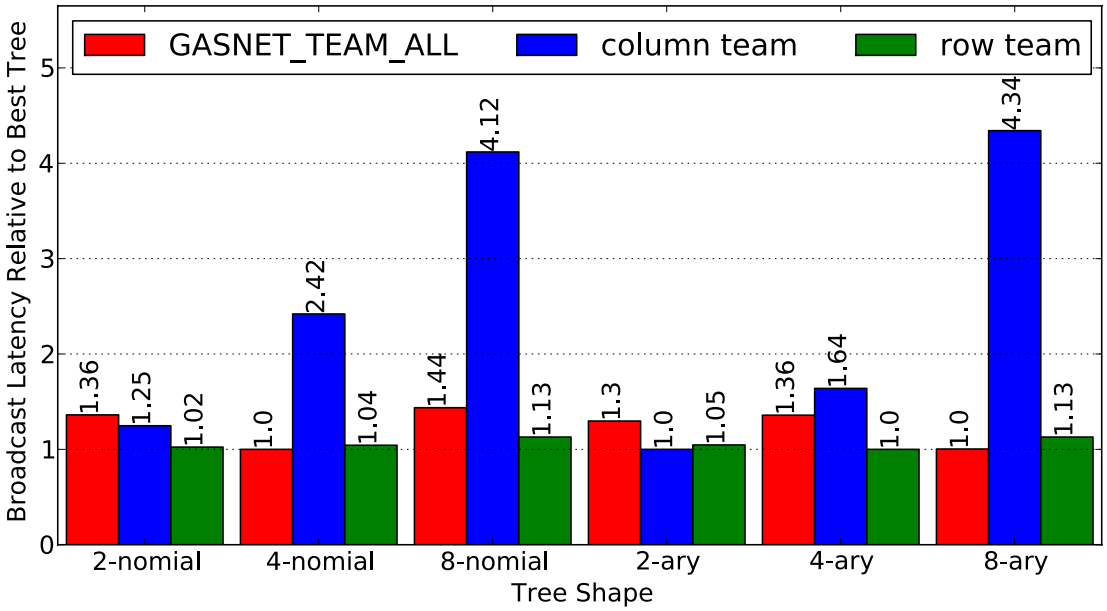


Figure 9.5: Comparison of Trees on Different Teams (256 cores of the Sun Constellation)

happen simultaneously. The benchmark separates these two phases with a global barrier to ensure that the network is quiet.

As the data show, for each team configuration the optimal tree shape is different. They are a 4-nomial, 2-ary, and 8-ary tree for the global, column teams, and row teams respectively. In addition, if we were to naïvely use the optimal tree found for `GASNET_TEAM_ALL` for the column team then broadcast would take 2.42 × longer than it should. Notice that the performance of the 4-nomial and 8-ary trees is about the same for the global team but have drastically different performance characteristics for the column team. Thus one could easily choose the latter leading to algorithm that would be 4.34 × slower. Conversely if we were to use the best algorithm from the column team, a 2-ary tree, the broadcast on `GASNET_TEAM_ALL` would take 1.3 × as long. For the row teams notice that the performance is relatively insensitive to the tree shape, except if one were to use the 8-ary tree (the second best geometry for `GASNET_TEAM_ALL`) which would lead to a 13% degradation in performance. Thus we can conclude, that applying the tuning from `GASNET_TEAM_ALL` will yield suboptimal results for the teams since the communication characteristics of the subteam can be different than using all the threads necessitating tuning for the teams.

9.3.1 Current Status

Due to the implementation overhead of the data-centric approaches only a prototype implementation has been created for the IBM BlueGene/L for related work [129]. If such an interface is adopted by the community then the aforementioned team caching and automatic tuning mechanisms should be added. As our related work has shown, initial overheads associated with data-centric team construction do not adversely affect application scalabil-

ity. Currently GASNet implements the thread-centric collectives. Upon team creation an instance of the automatic tuning system is created and attached to the team so that the collectives on the teams yield optimal performance.

One can imagine a mechanism in which the collective team construction is separated between the language runtime and GASNet. For example, in the case of the data-centric collectives for UPC, the UPC runtime would analyze the shared data structures and arrays to the data-centric collectives and know which threads to place together in a team. Thus the UPC runtime would create a thread-centric team in GASNet as needed. Thus we argue that both models are important and satisfy different application and runtime needs and can be combined through the runtime layer.

Chapter 10

Conclusion

In this dissertation we have presented a new collective communication infrastructure that supports automatically tuned nonblocking collectives. This includes a description of our implementation of the collectives within the GASNet one-sided communication layer. As the data has shown, the performance advantages of leveraging a one-sided communication model translate well when applied to collective communication for two distinct types of networks. The synchronization and memory consistency semantics for programming models that use one-sided communication also present novel tuning and performance operations for the collective library. The following are a few of the highlights of the results.

- In Chapters 3 and 4 we outlined the differences between the one and two sided programming models and showed how the collectives found in the PGAS languages are unique.
- In Chapter 5 we discussed how the rooted collectives (Broadcast, Scatter, Gather, and Reduce) can be implemented. In particular we highlighted how different tree topologies, data transfer mechanisms, and synchronization modes affect the collective. We then went on to outline our software architecture for nonblocking collectives so the collectives can be overlapped with computation. To better understand the performance we created analytic performance models that were good in practice at mapping out the search space.
- As the data from Chapter 5 demonstrated, the one-sided communication model in GASNet is able to consistently deliver good performance improvements in the latency for the rooted collectives. GASNet achieves up to a 27% improvement in performance on 1024 cores of the Sun Constellation, 28% improvement on 2048 cores of the Cray XT4, and up to 71% improvement on 3072 cores of the Cray XT5 for a Broadcast. For the other collectives GASNet achieves a 27% improvement in Scatter latency on 1536 cores of the Cray XT5, a 45% improvement in the Gather latency on 256 cores of the Sun Constellation and a 65% improvement in Reduction latency on the 2048 cores of the Cray XT4.

- We also highlighted the potentially large algorithmic space for collective tuning. The chapter concluded by showing how the optimized collectives can be incorporated into Dense Matrix-Matrix multiplication and Dense Cholesky factorization to see good performance improvements. Our results showed that at 2k cores of the IBM BlueGene/P new UPC programs can deliver comparable performance to a vendor optimized ScaLAPACK library. In addition our results demonstrated an 86% improvement in performance over the traditional PBLAS/MPI implementatin of parallel DGEMM across 400 cores of the Cray XT4.

- Chapter 6 explored different implementation strategies for the nonrooted collectives Exchange and Gather-to-All. We also leveraged performance models written in the LogGP framework to better understand the performance of both collectives and to aid in the tuning. Our results show that GASNet is able to achieve up to a 23% improvement on an Exchange across 256 cores of the Sun Constellation and up to a 69% improvement in latency for a Gather-to-All on 1536 cores of the Cray XT5.

- We then incorporated these optimized collectives into the NAS FT benchmark to realize good application scaling. We slightly reworked the algorithms to enable the use of nonblocking collectives leading to a 17% improvements on 32k cores of the IBM BlueGene/P. At the time of the writing this dissertation the peak performance of a 1D FFT according to the HPC Challenge Benchmarks [2] is 6.25 TFlops on 224k cores of the Cray XT5 and 4.48 TFlops on 128k cores. We are able to achieve 2.98 TFlops for a 3D FFT on only 32k cores of the IBM BlueGene/P. The results show that on the Cray XT4 we are able to improve performance by 46% by using the tuned nonblocking collectives available in GASNet.

- Chapter 7 showed different implementation strategies for collectives that targeted pure shared memory platforms. As the data showed, we can get orders of magnitude better performance for critical collectives such as Barrier. In addition, we showed that the techniques of loosening the synchronization and pipelining the collectives behind each other also led to good performance improvements on these platforms as well.

- Our results showed that by leveraging shared memory we are able to achive two orders of magnitude improvement in the latency of a Barrier on a wide range of modern multicore systems. By further optimizing the communication topology and signalling mechanisms we are able to realize another 33% improvement on 32 cores of the AMD Barcelona and another 46% improvement on 128 cores of the Sun Niagara2.

- We also show how these improvements can be applied to an application targeted for shared memory. We incorporated the tuned collectives into Sparse Conjugate Gradient and realized up to a 22% improvement in overall application performance. Our tuned collectives are able to significantly reduce the amount of time the collectives spend in the communication phases of the program.

- A common theme throughout the dissertation was how the optimal algorithm was dependent on many different factors. In Chapter 8 we outlined a software architecture

that can be used to automatically tune these operations on a wide variety of platforms. The software architecture also allows different algorithms to be easily added so that future algorithmic advances can be easily incorporated. We then showed how the aforementioned performance models could be effectively used to guide the search so that only a quarder of the time spent in the exhaustive search is needed to find the optimal algorithm.

- We proposed two different methodologies for constructing teams in Chapter 9. The first was a more traditional way of having the teams constructed based on thread identifiers. The second was a more novel approach in which the teams are constructed based on the data involved in the collective.

- The automatic tuning system and all of the collective algorithms discussed in this dissertation are freely available in the latest release of GASNet [91] and Berkeley UPC [30].

Future Work

As the number of processors within single-socket systems and large scale distributed systems continues to grow exponentially, optimizing communication will be even more important than it is today. 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. Optimized collectives libraries will therefore be increasingly important, although using power rather than performance as a criteria for optimization. In order to ensure that the collective library can be reused across a wide variety of platforms the collective algorithms must continuously adapt to novel interconnection topologies. Thus one obvious direction for future work is the discovery of new and better collective algorithms adding them into the automatic tuning system. In addition future work can also extend the automatic tuning system and the performance models to tune the operations for other important factors such as power.

In order to facilitate algorithmic development and benchmarking one obvious direction for the future work is to develop more applications in the PGAS programming languages that use the collective communication libraries we have built. A wider variety of algorithms and platforms will provide a richer and more robust collective library. In addition, incorporating these collectives into higher level languages such as Python and Matlab would deliver tuned collective communication to a broader set of applications. In addition by incorporating the collectives into more novel programming languages and environments, improvements can be made to the collective interfaces to allow them to make them a more beneficial abstraction to a wider variety of applications and platforms.

We have shown a variety of different performance models throughout the dissertation, however the performance models are based entirely on static parameters and are unable to adapt to dynamic factors. One way to incorporate more dynamic runtime parameters is to create a set of performance models and use statistical learning to properly weight the different models given runtime conditions. Thus dynamic runtime conditions can be more

accurately captured. The main goal again should be to reduce the time to search.

A wide variety of systems that leverage heterogeneity are becoming popular. These platforms dedicate specialized computational resources to different tasks. Understanding and modifying the PGAS languages and their collectives to effectively leverage these platforms poses interesting problems for the runtime systems as well as the language developers.

In addition to optimizing the communication schedules for parallel and scientific computing workloads the collectives discussed in this work are also good for scheduling the communication for cloud computing applications in a scalable way. To target these platforms future work can focus on how to modify the tuning system to manage the more dynamic workloads found in these environments. In addition, cloud computing platforms are also designed with fault tolerance in mind so future work can also figure out how to tune and schedule the communication in the presence of network faults.

Closing Remarks

Alongside classical experimentation and theory, scientific simulation has become crucial in understanding natural and man-made processes. Some large and important scientific questions, such as the implications of climate change over the course of a century, are only possible to understand through simulation. The results, however, have profound impacts on society as a whole. Scientists are constantly able to solve problems that were once thought intractable and push our level understanding to new heights.

To be able to satiate the scientists' demand for computation power, systems are both becoming increasingly diverse and large. This puts particular pressure on the runtime systems for the programming models to be both scalable and portable. A critical component of this is the collective communication library found in these programming models. We believe that automatically tuning the collective library for performance and scalability is of critical importance so that scientists can continue to effectively leverage the large diverse systems to deliver fundamental scientific breakthroughs.

Bibliography

[1] Chombo: Infrastructure for Adaptive Mesh Refinement. http://seesar.lbl.gov/ANAG/chombo/.

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

[3] MPICH-GM implementation. http://www.myrinet.com.

[4] Parmetis - parallel graph partitioning and fill-reducing matrix ordering. http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

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

[6] The cascade high productivity language. *hips*, 00:52–60, 2004.

[7] Pathways to Open Petascale Computing: The Sun Constellation System-Designed for Performance. Technical report, Sun Microsystems, November 2009.

[8] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high performance parallel algorithm for 1-d fft. In *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, pages 34–40, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[9] Sadaf R. Alam, Jeffery A. Kuehn, Richard F. Barrett, Jeff M. Larkin, Mark R. Fahey, Ramanan Sankaran, and Patrick H. Worley. Cray xt4: an early evaluation for petascale scientific simulation. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[10] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *J. of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[11] Vassil N. Alexandrov, G. Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors. *Computational Science - ICCS 2006, 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part II*, volume 3992 of *Lecture Notes in Computer Science*. Springer, 2006.

[12] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. *The Fortress Language Specification*. Sun Microsystems, Inc., 1.0α edition, September 2006.

[13] George Almási, Charles Archer, Jose G. Casta nos, J. A. Gunnels, C. Chris Erway, Philip Heidelberger, Xavier Martorell, Jose E. Moreira, Kurt Pinnow, Joe Ratterman, Burkhard SteinmacherBurow, William Gropp, and Brian Toonen. Design and implementation of message-passing services for the Blue Gene/L supercomputer. *IBM Journal of Research and Development*, 49(2/3):393–406, March/May 2005. Available at http://www.research.ibm.com/journal/rd49-23.html.

[14] George Almási, Philip Heidelberger, Charles J. Archer, Xavier Martorell, C. Chris Erway, José E. Moreira, B. Steinmacher-Burow, and Yili Zheng. Optimization of mpi collective communication on bluegene/l systems. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 253–262, New York, NY, USA, 2005. ACM Press.

[15] George Almási, Philip Heidelberger, Charles J. Archer, Xavier Martorell, C. Chris Erway, José E. Moreira, B. Steinmacher-Burow, and Yili Zheng. Optimization of mpi collective communication on bluegene/l systems. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 253–262, New York, NY, USA, 2005. ACM Press.

[16] N. Alzeidi, M. Ould-Khaoua, and A. Khonsari. A new modelling approach of wormhole-switched networks with finite buffers. *Int. J. Parallel Emerg. Distrib. Syst.*, 23(1):45–57, 2008.

[17] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[18] Eduard Ayguade, Jordi Garcia, Merce Girones, Jesus Labarta, Jordi Torres, and Mateo Valero. Detecting and using affinity in an automatic data distribution tool. In *Languages and Compilers for Parallel Computing*, pages 61–75, 1994.

[19] 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.

[20] P. Balaji, D. Buntinas, S. Balay, B. Smith, R. Thakur, and W. Gropp. Nonuniformly communicating noncontiguous data: A case study with petsc and mpi. In *IEEE Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[21] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[22] Christopher Barton, Călin Caşcaval, George Almási, Yili Zheng, Montse Farreras, Siddhartha Chatterjee, and José N. Amaral. Shared memory programming for large scale machines. In *Programming Language Design and Implementation (PLDI)*, Ottawa, Canada, 2006.

[23] Christopher Barton, Calin Cascaval, George Almasi, Rahul Garg, and Jose Nelson Amaral. Multidimensional blocking in UPC. Technical Report RC24305, IBM, July 2007.

[24] Christopher Barton, CĆlin Casçaval, George Almási, Yili Zheng, Montse Farreras, Siddhartha Chatterje, and José Nelson Amaral. Shared memory programming for large scale machines. *SIGPLAN Not.*, 41(6):108–117, 2006.

[25] Christopher Barton, Arie Tal, Bob Blainey, and José N. Amaral. Generalized index-set splitting. pages 102–116, Edinburgh, Scotland, April 2005.

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

[27] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick. An evaluation of current high-performance networks. In *The 17th Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

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

[29] Marsha J. Berger. *Adaptive mesh refinement for hyperbolic partial differential equations*. PhD thesis, 1982.

[30] The Berkeley UPC Compiler. http://upc.lbl.gov, 2009.

[31] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almási, Basilio B. Fraguela, María Jesús Garzarán, David A. Padua, and Christoph von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *PPOPP*, pages 48–57, 2006.

[32] Jeff Bilmes, Krste Asanović, Chee whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*, Vienna, Austria, July 1997.

[33] BLAS Home Page. http://www.netlib.org/blas/.

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

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

[36] 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.

[37] D. Bonachea, R.Nishtala, P. Hargrove, and K. Yelick. Efficient point-to-point synchronization in upc. In *Partitioned Global Address Space Programming Models*, 2006.

[38] Dan Bonachea, Paul Hargrove, Michael Welcome, and Katherine Yelick. Porting gasnet to portals: Partitioned global address space (pgas) language support for the cray xt. In *Cray Users Group*, 2009.

[39] Rajesh Bordawekar, Alok Choudhary, and J. Ramanujam. Compilation and communication strategies for out-of-core programs on distributed memory machines. *Journal of Parallel and Distributed Computing*, 38(2):277–288, 1996.

[40] J. Borrill, J. Carter, L. Oliker, D. Skinner, and R. Biswas. Integrated performance monitoring of a cosmology application on leading hec platforms. In *International Conference on Parallel Processing (ICPP)*, 2006.

[41] Dhruba Borthakur. *The Hadoop Distributed File System: Architecture and Design*. The Apache Software Foundation, 2007.

[42] Eric Brewer. *Portable High-Performance Supercomputing: High-Level Platform-Dependent Optimization*. PhD thesis, Massachusetts Institute of Technology, 1994.

[43] Ron Brightwell, Sue P. Goudy, Arun Rodrigues, and Keith D. Underwood. Implications of application usage characteristics for collective communication offload. *Int. J. High Perform. Comput. Netw.*, 4(3/4):104–116, 2006.

[44] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal, and Derrick Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, 1997.

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

[46] Lynn Elliot Cannon. *A cellular computer to implement the kalman filter algorithm*. PhD thesis, Montanat State University, 1969.

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

[48] Bradford L. Chamberlain, Sung-Eun Choi, E. Christopher Lewis, Calvin Lin, Lawrence Snyder, and Derrick Weathersby. ZPL: A machine independent programming language for parallel computers. *Software Engineering*, 26(3):197–211, 2000.

[49] W. Chen, D. Bonachea, J. Duell, P. Husband, C. Iancu, and K. Yelick. A Performance Analysis of the Berkeley UPC Compiler. In *Proc. of Int'l Conference on Supercomputing (ICS)*, June 2003.

[50] W. Chen, D. Bonachea, J. Duell, P. Husband, C. Iancu, and K. Yelick. A Performance Analysis of the Berkeley UPC Compiler. In *Proc. of Int'l Conference on Supercomputing (ICS)*, June 2003.

[51] Wei-Yu Chen, Costin Iancu, and Katherine Yelick. Communication optimizations for fine-grained UPC applications. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 267–278, Washington, DC, USA, 2005. IEEE Computer Society.

[52] Jaeyoung Choi, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, David W. Walker, and R. Clinton Whaley. A proposal for a set of parallel basic linear algebra subprograms. In *PARA '95: Proceedings of the Second International Workshop on Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science*, pages 107–114, London, UK, 1996. Springer-Verlag.

[53] 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.

[54] C. Clos. A study of non-blocking switching networks. *Bell Systems Technical Journal*, 32:406–424, 1953.

[55] Cristian Coarfa, Yuri Dotsenko, Jason Eckhardt, and John Mellor-Crummey. Co-array fortran performance and potential: An npb experimental study*.

[56] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: co-array fortran and unified parallel c. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 36–47, New York, NY, USA, 2005. ACM Press.

[57] Phillip Colella. Defining software requirements for scientific computing. Talk Given in 2004.

[58] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[59] D. E. Culler, A. C. Arpaci-Dusseau, R. Arpaci-Dusseau, B. N. Chun, S. S. Lumetta, A. M. Mainwaring, R. P. Martin, C. O. Yoshikawa, and F. Wong. Parallel Computing

on the Berkeley NOW. In *Proceedings of the 9th Joint Parallel Processing Symposium*, Kobe, Japan, 1997.

[60] David Culler, J. P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, August 1998.

[61] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1993.

[62] Anthony Danalis, Ki-Yong Kim, Lori Pollock, and Martin Swany. Transformations to parallel codes for communication-computation overlap. In *Supercomputing 2005*, November 2005.

[63] DARPA High Productivity Computing Systems. http://www.darpa.mil/ipto/programs/hpcs.

[64] Kaushik Datta, Dan Bonachea, and Katherine Yelick. Titanium performance and potential: an NPB experimental study. In *Proc. of Languages and Compilers for Parallel Computing*, 2005.

[65] Kauskik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures. In *Supercomputing*, November 2008.

[66] Tim Davis. The University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices/.

[67] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. pages 137–150, December 2004.

[68] James W. Demmel. *Applied Numerical Linear Algebra*. SIAM, August 1997.

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

[70] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

[71] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:1–18, 2003.

[72] Derek R. Dreyer and Michael L. Overton. Two heuristics for the euclidean steiner tree problem. *J. of Global Optimization*, 13(1):95–106, 1998.

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

[74] Tarek El-Ghazawi and Francois Cantonnet. Upc performance and potential: a npb experimental study. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–26, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[75] Maria Eleftheriou, José E. Moreira, Blake G. Fitch, and Robert S. Germain. A volumetric fft for bluegene/l. In Pinkston and Prasanna [141], pages 194–203.

[76] ESSL User Guide. http://www-03.ibm.com/systems/p/software/essl/index.html.

[77] L. S. Blackford et al. ScaLAPACK: a linear algebra library for message-passing computers. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing (Minneapolis, MN, 1997)*, page 15 (electronic), Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[78] Ahmad Faraj, Pitch Patarasuk, and Xin Yuan. A study of process arrival patterns for mpi collective operations. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 168–179, New York, NY, USA, 2007. ACM.

[79] Ahmad Faraj and Xin Yuan. An empirical approach for efficient all-to-all personalized communication on ethernet switched clusters. In *ICPP*, 2005.

[80] Ahmad Faraj, Xin Yuan, and David Lowenthal. Star-mpi: self tuned adaptive routines for mpi collective operations. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 199–208, New York, NY, USA, 2006. ACM.

[81] Dror Feitelson, Eitan Frachtenberg, Larry Rudolph, and Uwe Schwiegelshohn. *Job Scheduling Strategies for Parallel Processing: 11th International Workshop, JSSPP 2005, Cambridge, MA, USA, June 19, 2005, Revised Selected Papers (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[82] S. J. Fink and S. B. Baden. Run time data distribution for block structured applications on distributed memory computers. In *Parallel Processing for Scientific Computing*, pages 762–767, Philadelphia, 1995. SIAM.

[83] Franklin system. http://www.nersc.gov/.

[84] Ranger system. http://www.tacc.utexas.edu/.

[85] Matteo Frigo and Steven 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".

[86] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[87] A. Gara, M. A. Blumrich, D. Chen, G. L.-T Chiu, P. Coteus, M.E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-burow, T. Takken, and P. Vranas. Overview of the BlueGene/L system architecture. *IBM Journal of Research and Development*, 49(2/3):195–212, 2005.

[88] J. Garcia, E. Ayguad'e, and J. Labarta. Two-dimensional data distribution with constant topology, 1997.

[89] J. Garcia, E. Ayguad'e, and J. Labarta. Two-dimensional data distribution with constant topology, 1997.

[90] M. R. Garey and D. S. Johnson. The rectilinear steiner tree problem is np-complete. *SIAM Journal on Applied Mathematics*, 32(4):826–834, 1977.

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

[92] GCCUPC website. http://www.gccupc.org/.

[93] M. Gupta and P. Banerjee. Compile-time estimation of communication costs of programs. *Journal of Programming Languages*, 2(3):191–225, 1994.

[94] Manish Gupta, Edith Schonberg, and Harini Srinivasan. A unified framework for optimizing communication in data-parallel programs:. *IEEE Transactions on Parallel and Distributed Systems*, 7(7):689–704, 1996.

[95] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Houston, Tex., 1993.

[96] Paul Hilfinger, Dan Bonachea, David Gay, Susan Graham, Ben Liblit, Geoff Pike, and Katherine Yelick. Titanium language reference manual. Tech Report UCB/CSD-01-1163, U.C. Berkeley, November 2001.

[97] T. Hoefler, A. Lumsdaine, and 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, Nov. 2007.

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

[99] Eun-Jin Im, Katherine A. Yelick, and Richard Vuduc. SPARSITY: Framework for optimizing sparse matrix-vector multiply. *International Journal of High Performance Computing Applications*, 18(1):135–158, February 2004.

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

[101] Intel Math Kernel Library Reference Manual. http://www.intel.com/software/products/mkl/.

[102] International Organization for Standardization. *ISO/IEC 9899-1999: Programming Languages—C*, December 1999.

[103] Intrepid system. http://www.alcf.anl.gov/.

[104] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[105] Jaguar system. http://www.ornl.gov/.

[106] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[107] L. V. Kale, Sameer Kumar, and Krishnan Vardarajan. A Framework for Collective Personalized Communication. In *Proceedings of IPDPS'03*, Nice, France, April 2003.

[108] S. Kamil, J. Shalf, L. Oliker, and D. Skinner. Understanding ultra-scale application communication requirements. In *IEEE International Symposium on Workload Characterization*, 2005.

[109] Krishna Kandalla, Hari Subramoni, Gopal Santhanaraman, Matthew Koop, and Dhabaleswar K. Panda. Designing multi-leader-based allgather algorithms for multi-core clusters. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[110] Ulrich Kremer. Automatic data layout for distributed memory machines. Technical Report TR96-261, 14, 1996.

[111] F. Kuijlman, H.J. Sips, C. van Reeuwijk, and W.J.A. Denissen. A unified compiler framework for work and data placement. In *Proc. of the ASCI 2002 Conference*, pages 109–115, June 2002.

[112] Sameer Kumar, Gabor Dozsa, Gheorghe Almasi, Philip Heidelberger, Dong Chen, Mark E. Giampapa, Michael Blocksome, Ahmad Faraj, Jeff Parker, Joseph Ratterman, Brian Smith, and Charles J. Archer. The Deep Computing Messaging Framework: Generalized scalable message passing on the BlueGene/P supercomputer. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 94–103, New York, NY, USA, 2008. ACM.

[113] J. Lawrence and Xin Yuan. An mpi tool for automatically discovering the switch level topologies of ethernet clusters. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008.

[114] W. W. Lee. Gyrokinetic particle simulation model. *J. Comput. Phys.*, 72(1):243–269, 1987.

[115] X. Li and J. Demmel. Superlu dist: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 2003.

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

[117] Amith R. Mamidala, Sundeep Narravula, Abhinav Vishnu, Gopalakrishnan Santhanaraman, and Dhabaleswar K. Panda. On using connection-oriented vs. connectionless transport for performance and scalability of collective and one-sided operations: trade-offs and impact. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2007.

[118] Amith Rajith Mamidala, Abhinav Vishnu, and Dhabaleswar K Panda. Efficient shared memory and rdma based design for mpi_allgather over infiniband. In *Proceedings of EUROPVM/MPI*, 2006.

[119] A.R. Mamidala, null Lei Chai, null Hyun-Wook Jin, and D.K. Panda. Efficient smp-aware mpi-level broadcast over infiniband's hardware multicast. *Parallel and Distributed Processing Symposium, International*, 0:305, 2006.

[120] T. MathWorks. Using matlab, 1997.

[121] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[122] Dragan Mirkovic. Automatic performance tuning in the uhfft library. In *ICCS '01: Proceedings of the International Conference on Computational Sciences-Part I*, pages 71–80, London, UK, 2001. Springer-Verlag.

[123] Message Passing Interface. http://www.mpi-forum.org/docs/docs.html.

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

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

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

[127] J. C. Nash. *"The Cholesky Decomposition." In Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation*, chapter 7, pages 84–93. Bristol, England: Adam Hilger, 2nd edition, 1990.

[128] Rajesh Nishtala. Architectural probes for measuring communication overlap potential. Master's thesis, UC Berkeley, 2006.

[129] Rajesh Nishtala, George Almasi, and Calin Cascaval. Performance without pain = productivity: data layout and collective communication in UPC. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 99–110, New York, NY, USA, 2008. ACM.

[130] Rajesh Nishtala, Paul H Hargrove, Dan O Bonachea, and Katherine A Yelick. Scaling communication-intensive applications on bluegene/p using one-sided communication and overlap. In *23rd International Parallel & Distributed Processing Symposium*, 2009. Rome, Italy.

[131] Rajesh Nishtala, Richard Vuduc, James Demmel, and Katherine Yelick. When cache blocking sparse matrix vector multiply works and why. In *Proceedings of the PARA'04 Workshop on the State-of-the-art in Scientific Computing*, Copenhagen, Denmark, June 2004.

[132] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[133] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *ACM Fortran Forum*, 17(2):1 – 31, 1998.

[134] OpenMP. Simple, portable, scalable SMP programming. http://www.openmp.org/, 2000.

[135] Openpbs website. http://www.pbsgridworks.com/.

[136] P3dfft. http://www.sdsc.edu/us/resources/p3dfft/.

[137] Yunheung Paek, Angeles G. Navarro, Emilio L. Zapata, and David A. Padua. Parallelization of benchmarks for scalable shared-memory multiprocessors. In *IEEE PACT*, pages 401–, 1998.

[138] UC Berkeley Parallel Computing Laboratory. http://parlab.eecs.berkeley.edu/.

[139] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design: The Hardware/software Interface*. Morgan Kaufmann, 2005.

[140] Radia Perlman. *Interconnections (2nd ed.): bridges, routers, switches, and internetworking protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[141] Timothy Mark Pinkston and Viktor K. Prasanna, editors. *High Performance Computing - HiPC 2003, 10th International Conference, Hyderabad, India, December 17-20, 2003, Proceedings*, volume 2913 of *Lecture Notes in Computer Science*. Springer, 2003.

[142] Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel, and Jack J. Dongarra. Performance analysis of mpi collective operations. *Cluster Computing*, 10(2):127–143, 2007.

[143] Ravi Ponnusamy, Joel H. Saltz, Alok N. Choudhary, Yuan-Shin Hwang, and Geoffrey Fox. Runtime support and compilation methods for user-specified irregular data distributions. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):815–831, 1995.

[144] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[145] Ying Qian and Ahmad Afsahi. Efficient rdma-based m ulti-port collectives on multi-rail qsnetii clusters. In *The 6th Workshop on Communication Architecture for Clusters (CAC 2006), In Proceedin gs of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, 2006.

[146] Z. Ryne and S. Seidel. A specification of the extensions to the collective operations of unified parallel c. Technical Report Technical Report 05-08, Michigan Technological University, Department of Computer Science, 2005.

[147] ScaLAPACK. http://www.netlib.org/scalapack.

[148] T. J. Sheffler, R. Schreiber, J. R. Gilbert, and S. Chatterjee. Aligning parallel arrays to reduce communication. In *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, 1995.

[149] Kuei-Ping Shih, Jang-Ping Sheu, Chua-Huang Huang, and Chih-Yung Chang. Efficient index generation for compiling two-level mappings in data-parallel programs. *Journal of Parallel and Distributed Computing*, 60(2):189–216, 2000.

[150] Carlos Sosa and Brant Knudson. Ibm system blue gene solution: Blue gene/p application development, November 2009. IBM Redbook: SG24-7287-03 .

[151] Matthew J. Sottile, Craig Edward Rasmussen, and Richard L. Graham. Co-array collectives: Refined semantics for co-array fortran. In Alexandrov et al. [11], pages 945–952.

[152] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995.

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

[154] Rajeev Thakur and William Gropp. Improving the performance of collective operations in mpich. In *Proceedings of the 11th EuroPVM/MPI conference*. Springer-Verlag, September 2003.

[155] Vinod Tipparaju and Jarek Nieplocha. Optimizing all-to-all collective communication by exploiting concurrency in modern networks. In *SC*, 2005.

[156] Guillermo P. Trabado and Emilio L. Zapata. Data parallel language extensions for exploiting locality in irregular problems. In *Languages and Compilers for Parallel Computing*, pages 218–234, 1997.

[157] Peng Tu and David A. Padua. Automatic array privatization. In *Compiler Optimizations for Scalable Parallel Systems Languages*, pages 247–284, 2001.

[158] UPC consortium home page. http://upc.gwu.edu/.

[159] UPC language specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[160] Sathish S. Vadhiyar, Graham E. Fagg, and Jack J. Dongarra. Performance modeling for self adapting collective communications for mpi. In *LACSI Symposium*, 2001.

[161] Robert van de Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. *TR-95-13, Department of Computer Sciences, University of Texas*, 1995.

[162] Kees van Reeuwijk, Will Denissen, Henk J. Sips, and Edwin M. R. M. Paalvast. An implementation framework for HPF distributed arrays on message-passing parallel computer systems:. *IEEE Transactions on Parallel and Distributed Systems*, 7(9):897–914, 1996.

[163] Jeffrey S. Vetter and Frank Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *J. Parallel Distrib. Comput.*, 63(9):853–865, 2003.

[164] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, USA, June 2005. Institute of Physics Publishing.

[165] Richard W. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, Berkeley, December 2003.

[166] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[167] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of Supercomputing 2007*, 2007.

[168] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

[169] Samuel Webb Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[170] The X10 programming language. http://x10.sourceforge.net, 2004.

[171] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. Productivity and performance using partitioned global address space languages. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32, New York, NY, USA, 2007. ACM.

[172] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience*, 10(11-13), September-November 1998.

[173] Zhang Zhang, Jeevan Savant, and Steven Seidel. A upc runtime system based on mpi and posix threads. In *PDP '06: Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'06)*, pages 195–202, Washington, DC, USA, 2006. IEEE Computer Society.