Optimizing Collective Communication on Multicores

Rajesh Nishtala
University of California, Berkeley

Katherine A. Yelick
University of California, Berkeley

Abstract

As the gap in performance between the processors and the memory systems continue to grow, the communication component of an application will dictate the overall application performance and scalability. Therefore it is useful to abstract common communication operations across cores as *collective communication* operations and tune them through a runtime library that can employ sophisticated automatic tuning techniques. Our focus of this paper is on collective communication in Partitioned Global Address Space languages which are a natural extension of the shared memory hardware of modern multicore systems. In particular we highlight how automatic tuning can lead to significant performance improvements and show how loosening the synchronization semantics of a collective can lead to a more efficient use of the memory system. We demonstrate that loosely synchronized collectives can realize consistent $3\times$ speedups over their strictly synchronized counterparts on the highly threaded Sun Niagara2 for message sizes ranging from 8 bytes to 64kB. We thus argue that the synchronization requirements for a collective must be exposed in the interface so the collective and the synchronization can be optimized together.

1 Introduction

Current hardware trends show that the number of cores per chip is growing at an exponential pace and we will see hundreds processor cores within a socket in the near future [5]. However, the performance of the communication and memory system has not kept pace with this rapid growth in processor performance [20]. 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.

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 memory 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.

Many communication-intensive problems involve global communication, in which one thread broadcasts to others, one thread combines values from others, or data is exchanged between threads in operations such as a transpose. To coordinate communication operations across cores, it is often useful to think of these as *collective communication* operations, in which a group of threads collectively work to perform the global communication operation. Collective communication is very popular in programming models that involve a fixed set of parallel threads, because multiple threads can combine together to perform the communication efficiently through combining trees or other structures, rather than having a single thread perform all of the work. Collective communication is widely used in message passing (MPI [14]) programs and in global address space models like UPC [18] for both convenience and scalability.

While the collective communication problem has been well studied in the context of message passing on distributed memory clusters [16], we focus on Partitioned Global Address Space Languages (PGAS) such as UPC, Co-Array Fortran [9], and Titanium [21] on multicore and SMP systems. The key feature that distinguishes PGAS languages from message passing is the use of one-sided as opposed to two-sided communication: threads in a PGAS language communicate by reading and writing remote data without the need for a matching communication operation on the other side. One-sided communication decouples data transfer from synchronization and therefore allows for faster communication [7] on clusters, where they are most commonly used. 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.

> **Our Position:** Collective communication operations are useful in programming multicore systems because they encapsulate performance-critical data movement operations. Automatic tuning can significantly improve performance by selecting the right implementation for a given system and communication pattern. Loosening the synchronization requirements for collective operations can also improve performance, and the synchronization requirements should be a parameter in the tuning process.

The techniques and analysis in this paper apply to many common data movement and synchronization patterns such as *Broadcast*, *Scatter*, *Gather*, *Barrier*, *Exchange* and *Reduce*. Due to space considerations, we discuss the following two operations:

Barrier: A thread can not exit a call to a barrier until all the other threads have called the barrier.

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 $\sum_{t=0}^{N-1} x_t[j]$. The operations sum, minimum, and maximum are usually built-in and the user is allowed to supply more complicated functions.

2 Tuning Collectives for Shared Memory

The wide variety of architectures that are currently deployed and under development necessitates a system that can automatically tune these operations rather than wasting valuable time hand-tuning. As the number of cores continues to grow, the performance and scalability of these algorithms will play a more prominent role in the overall application performance.

2.1 The Tuning Space

There have been many related projects that have focused their efforts on hand-tuning collectives including a few

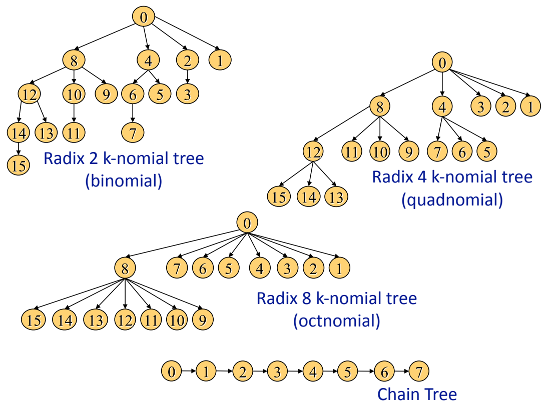

Figure 1: Example Tree Topologies

that have focused on automatically tuning these operations for clusters[16]. From the body of literature it is clear that the tuning space is indeed large. For example, for a rooted collective there are an exponential number of tree shapes that can be used to disseminate the data. Our experience and related work has shown that the following parameters affect the choice of the optimal algorithm. Further research will show whether this list is sufficient to capture all the parameters needed. Factors that influence performance include: processor type/speed, interconnect topology, interconnect latency, interconnect bandwidth, number of threads involved, size of the messages being transferred, synchronization mode, network load, mix of collectives that the user performs, amount of local memory available for collectives.

2.2 Algorithm Selection

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 [13], 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 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. These are extensions of the binomial tree found in [10] where a radix other than 2 is used. The higher the radix, the shallower the tree. Figure 1 shows a diagram of the trees used[1]. While the set of all possible trees is very large, we only focus on a small subset of them to illustrate our argument. Thus to implement a barrier each thread signals its parent once its subtree has arrived

[1]A radix "1" k-nomial tree is a chain tree.

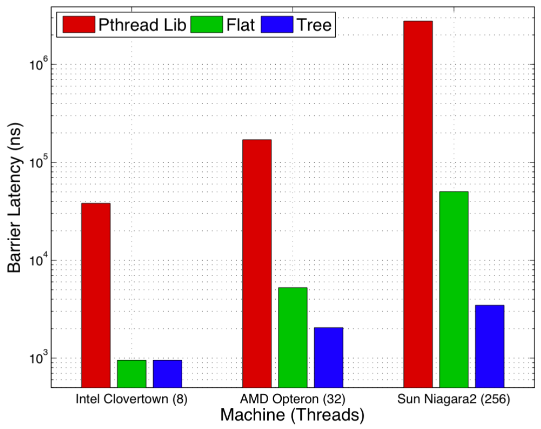

Figure 2: Barrier Performance

Processor	GHz	Cores (Threads)	Sockets
Intel Clovertown[1]	2.66	8 (8)	2
AMD Barcelona[2]	2.30	32 (32)	8
Sun Niagara2[3]	1.40	32 (256)	4

Table 1: Experimental Platforms

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 `ints` and atomic counters.

Figure 2 shows the latency of a barrier on three modern multicore machines shown in Table 1. The group of bars labeled "Pthread Lib" is the performance of the barrier found in the pthread library. The bars labeled "Flat" show the performance of a flat topology to accomplish a barrier, (i.e. all threads communicate directly with the root). The final column, "Tree," shows the performance of the barrier by selecting the best tree geometry. The pthreads library has been designed to handle the case when the number of threads is larger than the number of hardware thread contexts. However as the data show, the pthread library adds a significant amount of unnecessary overhead if the number of software threads do not exceed the number of available hardware thread contexts[2]. Thus even switching to a flat topology in which the threads communicate through the cache can yield almost two orders of magnitude in performance. However as the number of cores continue to rise at a dramatic rate, simply relying on all cores communicating directly with one root thread leads to non-scalable code as shown by the performance data from the Sun Niagara2. Using scalable tree based algorithms can yield another order of magnitude in performance improvement. Thus on the Sun Niagara2 we achieve three orders of magnitude performance gains by tuning the collectives for shared memory.

[2]The comparison of performance of the various approaches within a machine is more valid than comparison across machines due to the wide variation in the amount of available hardware threads.

3 Collective Synchronization

The simplest semantics for a collective communication is to have it appear to execute in isolation after all preceding code on all threads has completed and before any succeeding code starts. However, in a traditional two-sided message passing model, a collective is considered complete on a particular thread when it has received its piece of the data. This does not imply that *all* threads have received their data, but a thread cannot view such asynchrony because all communication is done explicitly, and any lagging thread cannot communicate since it is still tied up executing the collective. In the case of a global address space model, such asynchrony *may be* visible if a thread has received its local contribution from a collective and then reads or writes data on a lagging thread. This raises interesting questions about when the data movement for a collective can start and when a collective is considered complete. Avoiding synchronization can improve performance but complicate programming. We argue that it is essential for the user to specify the synchronization requirements of the collective to achieve maximum possible performance so that the synchronization can be factored into the tuning process.

To illustrate the impacts of collective synchronization we consider *Reduce* on the Sun Niagara2. 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 256 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 3 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. 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 real-

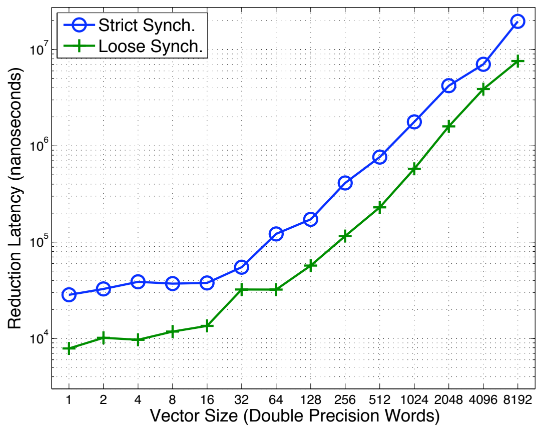

Figure 3: Reduction Performance on Niagara2

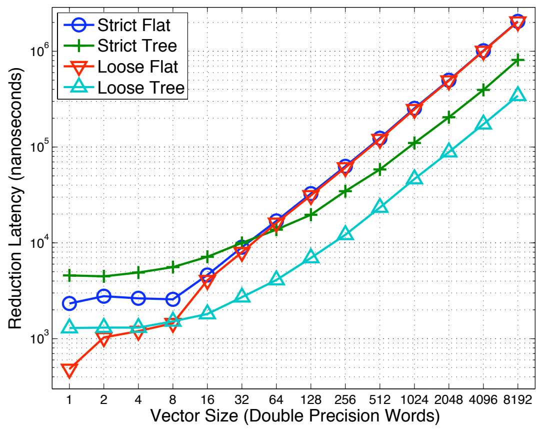

Figure 4: Optimal Algorithm Selection on Barcelona

ize 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 3, the median performance gain of the strict execution time compared to the loose execution time is about $3.1\times$ while the maximum is about $4\times$.

4 Collective Tuning and Synchronization

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 4. 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 in Section 2[3]. Once a child has accumulated the result for all its 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.

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 inte-

[3]We perform an exhaustive search over the tree topologies and report the best one.

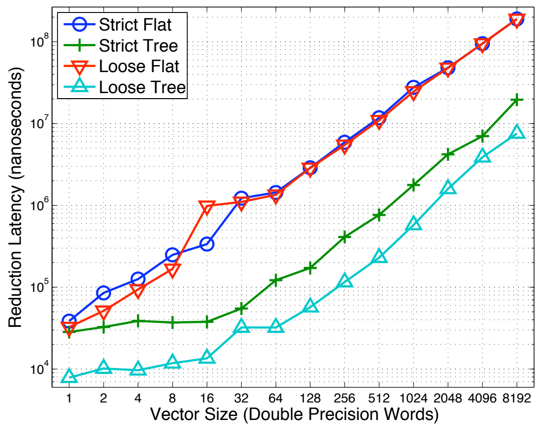

Figure 5: Optimal Algorithm Selection on Niagara2

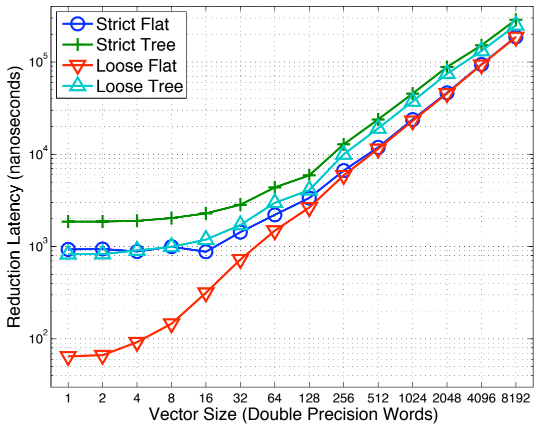

Figure 6: Optimal Algorithm Selection on Clovertown

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

gral part of selecting the best algorithm.

The switch-point is heavily dependent on the target architecture and the concurrency level. The same data is shown for the Niagara2 as well as the Clovertown in Figures 5 and 6. The high concurrency levels for the Niagara2 mandate trees for both synchronization modes since the scalability and added parallelism the trees offer is important at all vector sizes. However, on the Clovertown, the lower thread count implies that the added cost of the point-to-point synchronization associated with the trees outweighs the benefits from parallelism and thus the Flat topology performs the best.

4.1 Tree Selection

Table 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 the Chain tree 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 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.

5 Related Work

Most of the prior work on collective tuning has been done for distributed memory platforms. This includes both hand-tuning for a particular machine[17, 4], optimizations for clusters with shared memory nodes[12] and creating new algorithms[8, 6]. The literature shows that the tuning space is quite large and that automatic tuning (which includes empirical search) is often beneficial[16]. In prior work by one of the authors, we presented a novel interface for UPC collectives on BlueGene/L[15]. In contrast, the work in this paper is focused on collectives within a single shared memory node, including the highly multi-threaded Niagara2 architecture.

Automatically searching over a set of optimized versions of an algorithm has become a popular technique used by compiler and library developers for computational kernels. That literature is too large to cite here, but includes optimizations of dense and sparse linear algebra, spectral transforms, and stencil operations on structured grids. Most of the earlier work on automatic tuning work for computational kernels was done on single processors, but more recent work includes multicore systems[11, 19, 20]. We believe the collective communication problem is at least as important as these computational kernels, since collective patterns are common in user applications and they stress the most limited features of current and future systems.

6 Conclusion

Given the recent limits to clock speed scaling, future performance increases will rely primarily on increasing the number of processor cores. As the numbers of cores grow, communication and memory systems that already limit performance will be increasingly likely to surface as bottlenecks. By focusing on global communication operations encapsulated in a collective communication interface, we measured and tuned some of the most critical execution patterns for future systems. We showed the interface considerations were critical, that loosely synchronized operations are up to 4x faster than strictly synchronized ones for a fixed communication topology. This identifies a clear performance and productivity trade-off, as the strictly synchronized operations are less likely to result in surprising behavior, but are consistently slower. Furthermore, even on relatively small shared memory systems, the choice of topology (the communication pattern within the algorithm) is critical to performance, with differences as high as 4x between the best and worst topology for a given synchronization mode. In general, the choice of optimal algorithm varies with the system, the size of the data being communicated, and the synchronization mode. All of this suggests the need for automatic tuning of multicore collectives on today's multicore SMP systems. As the communication component of applications continues to become a more significant part of the runtime tuning these operations will take center stage to ensure application scalability to the manycore systems of the future.

Acknowledgments

We would like to thank the members of the Berkeley UPC and BeBOP groups for their invaluable advice as well as Intel and Sun for their generous hardware donations. 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.

References

[1] Intel Xeon Quad Processor. http://www.intel.com/products/processor/xeon5000.

[2] Sun Fire X4600 M2 Server Information. http://www.sun.com/servers/x64/x4600/.

[3] Sun UltraSparc T2 Processor Information. http://www.sun.com/processors/UltraSPARC-T2/.

[4] Almási, G., Heidelberger, P., Archer, C. J., Martorell, X., Erway, C. C., Moreira, J. E., Steinmacher-Burow, B., and Zheng, Y. Optimization of mpi collective communication on BlueGene/L systems. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing* (New York, NY, USA, 2005), ACM Press, pp. 253–262.

[5] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. The landscape of parallel computing research: A view from berkeley. Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

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

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

[8] Bruck, J., Ho, C.-T., Upfal, E., Kipnis, S., and Weathersby, D. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Trans. Parallel Distrib. Syst. 8*, 11 (1997), 1143–1156.

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

[10] Culler, D. E., Karp, R. M., Patterson, D. A., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T. LogP: Towards a realistic model of parallel computation. In *Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (1993), pp. 1–12.

[11] Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., and Yelick, K. Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures. In *Supercomputing 2008 (SC08)* (November 2008).

[12] Mamidala, A., Kumar, R., De, D., and Panda, D. K. Mpi collectives on modern multicore clusters: Performance optimizations and communication characteristics. *Int'l Symposium on Cluster Computing and the Grid, Lyon, France* (May 2008).

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

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

[15] Nishtala, R., Almasi, G., and Cascaval, C. Performance without pain = productivity: Data layout and collective communication in upc. In *Principles and Practices of Parallel Programming (PPoPP)* (2008).

[16] Pješivac-Grbović, J. *Towards Automatic and Adaptive Optimizations of MPI Collective Operations*. PhD thesis, University of Tennessee, Knoxville, December 2007.

[17] Qian, Y., and Afsahi, A. Efficient rdma-based multi-port collectives on multi-rail qsnetii clusters. In *The 6th Workshop on Communication Architecture for Clusters (CAC 2006), In Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)* (2006).

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

[19] Williams, S., Carter, J., Oliker, L., Shalf, J., and Yelick, K. Lattice Boltzmann simulation optimization on leading multicore platforms. In *Interational Conference on Parallel and Distributed Computing Systems (IPDPS)* (2008).

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

[21] Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., and Aiken, A. Titanium: a high performance java dialect. In *Proc. of ACM 1998 Workshop on Java for High-Performance Network Computing* (February 1998).