

Benchmark Measurements of Current UPC Platforms

Zhang Zhang and Steven Seidel
Department of Computer Science
Michigan Technological University
Houghton, Michigan 49931
{zhazhang,steve}@mtu.edu

Abstract

UPC is a parallel programming language based on the concept of partitioned shared memory. There are now several UPC compilers available and several different parallel architectures that support one or more of these compilers. This paper is the first to compare the performance of most of the currently available UPC implementations on several commonly used parallel platforms. These compilers are the GASNet UPC compiler from UC Berkeley, the v1.1 MuPC compiler from Michigan Tech, the Hewlett-Packard v2.2 compiler, and the Intrepid UPC compiler. The parallel architectures used in this study are a 16-node x86 Myrinet cluster, a 32-processor AlphaServer SC-40, and a 48-processor Cray T3E. A STREAM-like microbenchmark was developed to measure fine- and course-grained shared memory accesses. Also measured are five NPB kernels using existing UPC implementations. These measurements and associated observations provide a snapshot of the relative performance of current UPC platforms.

1. Introduction

Unified Parallel C (UPC) is an extension of C for programming multiprocessors with a shared address space [5]. UPC provides a common syntax and semantics for explicit parallel programming in C, and it directly maps language features to the underlying architecture. UPC is an example of the *partitioned shared memory* programming model in which shared memory is partitioned among all UPC threads (processes). This partition is formally represented in the programming language. Each thread can access any location in shared memory using the same syntax but the locations in each thread's own partition of shared memory are accessed more quickly.

UPC has been gaining interests from academia, industry and government labs. A UPC consortium

(www.upc.gwu.edu) has been formed to foster and coordinate UPC development and research activities.

As a part of the consortium, the UPC group at Michigan Technological University has developed MuPC [13], an implementation of UPC whose run time system is based on MPI and Pthreads. Also on site are the Berkeley, HP, and Intrepid UPC compilers and three platforms on which one or more of these compilers can run: a 4x8-node AlphaServer SC-40, a 2x16-node Linux Myrinet cluster, and a 48-processor Cray T3E.

All earlier UPC performance reports concentrated on one or two UPC compilers on a single platform [3, 4, 6]. This paper reports performance measurements of six combinations of UPC compilers and platforms on UPC versions of the NAS parallel benchmarks and on the STREAM-like microbenchmark. These data illustrate the strengths and areas needing development for these UPC systems.

The remainder of this paper is organized as follows. Section 2 is a brief introduction to the UPC language. Section 3 describes the MuPC run time system and reviews other UPC compilers that were used in this work. Section 4 describes the benchmarks and the parallel platforms used. Section 5 reports performance measurements and discusses their significance. Section 6 summarizes these findings.

2. UPC Background

UPC is a superset of ANSI C (as per ISO/IEC9899 [11]). UPC programs adopt the SPMD (single program multiple data) execution model. Each UPC *thread* is a process executing a sequence of instructions in a program. Data objects in a UPC program can be either *private* or *shared*. A UPC thread has exclusive access to the private objects that reside in its private memory. A thread also has access to all of the (shared) objects in the shared memory space. UPC provides a partitioned view of shared memory by introducing the concept of affinity. Shared memory is equally partitioned among all threads. The block of shared memory associated with a given thread is said to have *affinity* to that

thread. The concept of affinity captures the reality that on most parallel architectures the latencies of accessing different shared objects are not identical. It is assumed that an access to a shared object that has affinity to the thread that performs the access is faster than an access to a shared object to which the thread does not have affinity.

UPC provides two types of pointers to shared objects. A *pointer-to-shared* is a pointer whose target is a shared data object. The pointer itself resides in the private memory of some thread. A *shared pointer-to-shared* is a pointer whose target is a shared data object. The pointer itself also resides in the shared memory space.

UPC provides a *strict* and a *relaxed* memory consistency model. This choice affects accesses to shared objects. The *strict* model does not allow reordering of shared object accesses. The *relaxed* model allows reordering and coalescence of shared object accesses between synchronization points as long as data dependencies within each thread are preserved. The *relaxed* model offers chances for compiler and run time system optimizations.

3. UPC compilers and run time systems

This paper examines the performance of four UPC compilers: MTU MuPC, Berkeley UPC, HP UPC and Intrepid UPC. MuPC, Berkeley UPC, and Intrepid UPC are open source projects. HP UPC is a commercial product. This section provides some details about MuPC and briefly describes the other compilers.

3.1. MuPC

MuPC [14] is a UPC run time system originally developed with help from the High Performance Technical Computing Division at Hewlett-Packard Company. MuPC implements an API defined by a UPC-to-C translator contributed by HP. The translator is based on the EDG C/C++ front end. The run time system is implemented using MPI and the POSIX standard thread library. MuPC currently supports AlphaServer SMP clusters and Intel x86 Linux clusters. The current version of MuPC is version 1.1; it implements the UPC Language Specification V1.1.1 [7]. Several specific features of MuPC are provided to ensure good performance.

The MuPC run time system implements software caching. Each UPC thread maintains a noncoherent, direct-mapped, write-back cache for remote scalar references made by a thread. Cache invalidation takes place at a fence, with dirty cache lines being written back to their sources using block transfers. Caching greatly reduces the memory latency in the relaxed mode by reducing the number of small messages. The run time system allows multiple outstanding remote writes. This is done by

using MPI's multiple completion capacity. The run time system may issue more than one MPI send and recv during remote writes. It completes them at a later time slot using one MPI test routine. The actual effect of this is similar to message pipelining. Previous studies [3, 8, 4, 6] have observed that an access to a shared variable with which the accessing thread has affinity should be faster than an access to a shared variable with affinity to a different thread. The MuPC front end differentiates these two kinds of accesses. The first kind has been made faster by converting them to regular local memory accesses.

3.2. Berkeley, HP, and Intrepid UPC Compilers

UC Berkeley and the Lawrence Berkeley National Lab [15] built an open source, highly portable UPC compiler. The Berkeley UPC compiler is based on the Open64 open source compiler. Its run time system is based on the GASNet [16] portable networking library, which supports a very wide variety of architectures and high performance networks. As a consequence, it is the most portable UPC compiler. In this work the Berkeley UPC compiler is run on an AlphaServer SMP and a Linux cluster. The current version of this compiler is version 2.0. It supports UPC Language Specification V1.1.1 [7].

Hewlett-Packard produced the first commercial UPC compiler for Tru64 Unix. HP UPC [9] supports any AlphaServer SMP machines and AlphaServer Clusters. Version 2.2 is the current version. It supports UPC Language Specification V1.1.1 [7].

Intrepid Technology provides a UPC compiler [10] as an extension to the GNU GCC compiler. It supports only shared memory platforms and SMP platforms such as the SGI and Cray T3E. It is freely available under the GPL license. It is based on GCC 3.3.2 and supports UPC Language Specification V1.0.

4. Experiments

4.1. The UPC STREAM microbenchmark

The UPC STREAM benchmark measures the sustainable memory bandwidths of a variety of shared memory fine- and coarse-grained accesses. The cost of fine-grained remote memory references is often a dominating factor in the performance of UPC programs [2], while the cost of coarse-grained references, which consist of calls to `upc_memget/put/cpy/set()` functions, reveals whether these message-passing operations are implemented efficiently.

George Washington University developed the first UPC STREAM microbenchmark [4] based on the STREAM microbenchmark for single processors originally designed by

McCalpin [12]. It measures only local shared references and same-thread remote shared references. This paper rewrites the UPC STREAM synthetic microbenchmark by simulating many more kinds of access patterns as described below. All of these access patterns are likely to occur in actual code.

Local shared read Read from shared memory locations with affinity to the reading thread.

Local shared set Write values to shared memory locations with affinity to the writing thread.

Unit stride shared read Read from consecutive shared memory locations with affinity to a remote thread.

Random shared read Read a random collection of shared memory locations with affinity to a remote thread.

Stride-n shared read Read as if doing column-major accesses in C. This operation is supposed to be cache-unfriendly.

Unit stride shared set Like the unit stride shared read, but writes.

Random shared set Like the random shared read, but writes.

Stride-n shared set Like the stride-n shared read, but writes.

Unit stride shared copy A read from shared memory followed by a write to shared memory; both the sink and the source locations have affinity to the same remote thread. Accesses are unit stride.

Random shared copy Like the unit stride shared copy, but the access pattern is random.

Stride-n shared copy Like the unit stride shared copy, but the access pattern is column-major.

Local memget Copy a contiguous chunk of bytes from a shared memory location to a private memory location. The source is with affinity to the current thread.

Local memput Like the local memget, but the opposite.

Local memset Like the local memput, but using the `upc_set()` function.

Local memcopy Copy a contiguous chunk of bytes between two shared memory locations. Both the source and the destination are with the current thread.

Remote memget Like the local memget, but the source is with affinity to a remote thread.

Remote memput Like the remote memget, but the opposite.

Remote memset Like the remote memput, but using the `upc_set()` function.

On-thread memcopy Like the local memcopy, but both the source and the destination are with affinity to the same remote thread.

Remote memcopy Like the on-thread memcopy, but the source and the destination are with affinities to different remote threads.

The cost difference between scalar data types is obscured by the long latency of accesses to shared memory. Because of this the bandwidth is reported in units of “references per second” rather than “bytes per second” for scalar references.

4.2. The UPC NAS benchmarks

The UPC NAS benchmark suite used in this paper was developed by the UPC group at George Washington University [3]. This suite is based on the original MPI+FORTRAN/C implementation of the the NAS Parallel Benchmark suite (NPB 2.4) [1]. There are five kernel benchmarks in the NPB suite. Every benchmark comes with a “naïve” implementation and one or more “optimized” implementations. The naïve implementation makes no effort to incorporate any hand-tuning techniques. The optimized implementations incorporate various extents hand-tuning techniques such as prefetching and privatized pointers-to-shared [3]. This study measures the performance of the naïve version and the most optimized version. The class A workload was used as the input size for all benchmarks. The five benchmarks are conjugate gradient (CG), embarrassingly parallel (EP), Fourier transform (FT), integer sort (IS), and multigrid (MG).

4.3. Platforms

All measurements are made on the following platforms. The Cray T3E is a shared memory platform with 48 PEs. Each PE has a 300MHz Alpha 21164 CPU. The T3E has low communication latency and high bandwidth. The Intrepid UPC compiler runs on this machine. The AlphaServer SC is an SMP cluster consisting of 8 nodes with four 667MHz Alpha 21264 EV67 processors per node. The communication network of the machine is a Quadrics interconnect. Measurements are made using at most two processors from each node. HP UPC, Berkeley UPC and MuPC run on this machine. The Linux cluster has 16 nodes connected with a Myrinet switch. Each node has two Pentium 1.5GHz processors. At most one processor per node is used for UPC jobs. Berkeley UPC and MuPC run on the cluster.

Both MuPC and HP UPC provide caching facilities for remote shared references. Each thread is associated with a cache whose geometry is tunable at compile time. The cache in MuPC is a noncoherent, direct-mapped, write-back cache. In this study, the cache has 256 blocks and each block

has a length of 1024 bytes. The cache in HP UPC is a non-coherent, associative, write-through cache. In this study, the cache is 4-way associative, with 256 blocks per set and 1024 bytes per block.

The version numbers of UPC compilers used in this report, if not otherwise noted, are Intrepid UPC v3.3.2.1, HP UPC v2.2, Berkeley UPC v1.1.1, and MuPC v1.1, respectively.

5. Performance measurement and discussion

5.1. UPC STREAM microbenchmark results

For reference as a baseline, the original STREAM microbenchmarks yielded the following references per second. Cray T3E: $1.7 \times 10^7 \pm 10^6$; AlphaServer SC-40: $6.8 \times 10^7 \pm 10^7$; x86 cluster: $1.7 \times 10^8 \pm 10^7$. These measurements of private scale, copy, add, and triad do not vary much on each platform. The AlphaServer showed the largest relative variation and the T3E the smallest.

The measurements gathered using the UPC STREAM microbenchmark are summarized in Figures 1, 2 and 3.

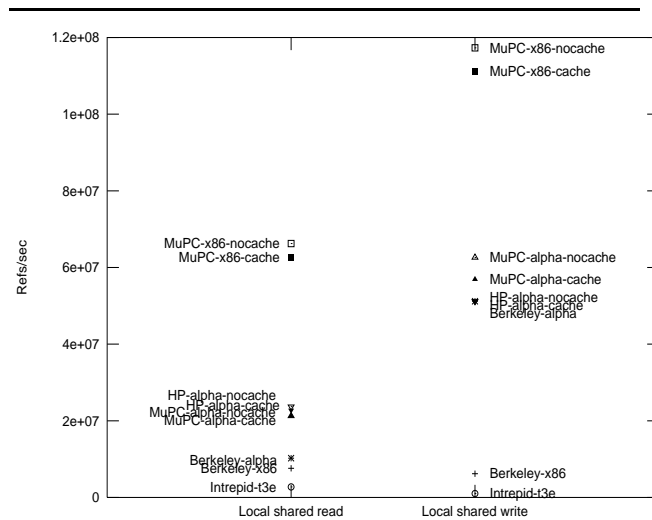


Figure 1. Local shared accesses per second

Local shared accesses A local shared access is an access (read or write) to a scalar that is in shared memory and has affinity to the thread performing the access. The cost of such an access should be the same as that of a private memory access. This performance was measured to be one or two times worse than that of private memory accesses because of the overhead of the UPC run time system. Figure 1 shows that on the x86 cluster, MuPC has very good local shared access performance. All systems running on the Al-

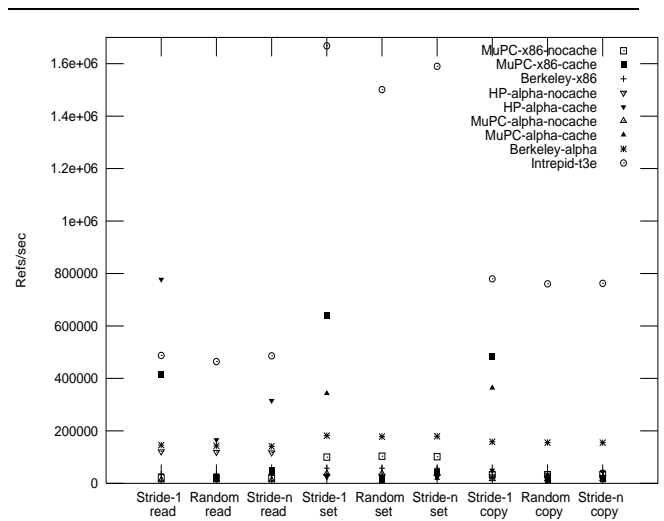


Figure 2. Remote shared accesses per second

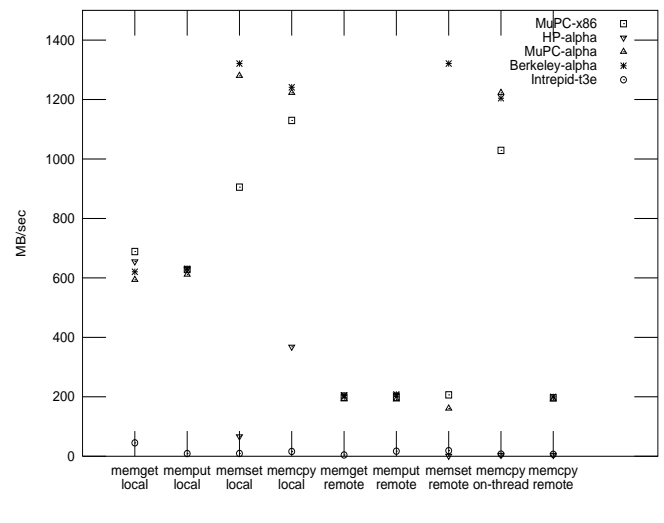


Figure 3. Block shared memory operations, megabytes per second

phaServer exhibit similar performance for local shared accesses. On the T3E, Intrepid UPC handles local shared accesses exceptionally slowly.

Remote shared accesses A remote shared access is an access (read or write) to a scalar that is in shared memory and does not have affinity to the thread performing the access. As Figure 2 shows, Intrepid UPC performs best for this category of shared accesses. The performance is 5 to 10 times better than other systems. This is consistent with the high bandwidth of the shared memory on the T3E. When the

run time cache is not used, HP UPC and Berkeley UPC have comparable performance, with Berkeley UPC being slightly better. Both systems have much better performance than MuPC. On the x86 cluster, Berkeley UPC performs about 50% better than MuPC; on the AlphaServer, HP UPC and Berkeley UPC are 4 to 10 times faster than MuPC. This is because MuPC uses MPI as the communication layer while HP UPC and Berkeley UPC are built on lower level communication layers. When the run time cache is turned on, cache-friendly benchmarks perform better. For MuPC, the cache helps improve the performance by a factor of 15 on the x86 cluster and by factors of 9 to 30 on the AlphaServer. For HP UPC, the cache helps improve the performance of stride-1 reads by a factor of at least 7, but the cache has a negative impact on write and copy. This is because the cache is write-through and remote writes are not cached.

Another important observation is that the performance of a remote shared copy is similar to that of a remote shared read and write. As described in Section 4, a remote shared copy is equivalent to a shared read followed by a shared write, where the source and destination locations have the same affinity to a remote thread. It is reasonable to expect a shared copy to have performance similar to that of local shared accesses because there is actually little inter-thread communication in this case. An optimizer should be able to pick up this fact and convert a shared copy to a local shared read or write. However, none of the current UPC systems do this.

Block memory operations Coarse-grained accesses to shared memory locations involve calls to UPC string handling functions with various source and destination affinities. Nine variations (as described in Section 4) are shown in Figure 3. Intrepid UPC on the T3E shows the lowest bandwidths in all categories. HP UPC, Berkeley UPC and MuPC perform similarly for local memget, local memput, remote memget, and remote memput. Berkeley UPC tops all memset and memcopy categories. HP UPC shows low bandwidths for all memset and memcopy categories.

5.2. NPB 2.4 results

Figures 4 through 8 display the results for the NPB 2.4 benchmarks. The run time cache was used on those compilers that provide one.

CG For the nonoptimized version, this benchmark scales very poorly on all systems. MuPC performs about one order of magnitude better than Berkeley UPC on the x86 cluster and about 3 times better on the AlphaServer. HP UPC performs about 70% faster than MuPC on the AlphaServer. For the optimized version, the scalability is still poor for MuPC, HP UPC and Intrepid UPC, but the overall performance of these systems improves 5-fold. Berkeley UPC shows an order of magnitude improvement on both the x86 cluster and

the AlphaServer. Berkeley UPC also achieves good scalability for the optimized version. This result shows that at the current stage Berkeley UPC is superior in handling unstructured fine-grain accesses. A cross-platform comparison shows that Intrepid UPC running on the T3E is usually slow. This is partially because the T3E machine has the slowest processors. However, since the T3E is very good at handling remote shared accesses, we believe there is ample room for performance improvements.

EP The embarrassingly parallel benchmark requires few remote accesses. MuPC, HP UPC and Berkeley UPC all exhibit nearly identical linear speedups on the x86 cluster and the AlphaServer. Intrepid UPC cannot run this benchmark to completion.

FT On the x86 cluster, Berkeley UPC performs about 50% better than MuPC. They both exhibit good scalability. Optimization increases the performance by 50% for Berkeley UPC and about 100% for MuPC so MuPC and Berkeley UPC have similar performance for the optimized version. On the AlphaServer, the nonoptimized benchmark does not run to completion on Berkeley UPC. HP UPC scales poorly. MuPC has the best performance. The optimized version increases the performance for HP UPC by at least 4 times and doubles the performance for MuPC. Berkeley UPC exhibits similar performance. Intrepid UPC cannot run this benchmark to completion.

IS This benchmark scales well in all cases. On the x86 cluster, MuPC exhibits a 30 – 50% performance edge against Berkeley UPC, for both nonoptimized and optimized implementation. Optimization doubles the performance in both cases. On the AlphaServer, HP UPC performs slightly better than others for the nonoptimized version. HP UPC, MuPC and Berkeley UPC perform very closely for the optimized version. Optimization at least doubles the performance for all three. Again, this benchmark fails Intrepid UPC on the T3E.

MG On the x86 cluster, Berkeley UPC cannot run this benchmark with less than 8 UPC threads because the problem size exhausts the available memory. When the number of threads is 8 or more, Berkeley UPC performs about 50% better than MuPC for the nonoptimized version. Both MuPC and Berkeley UPC benefit tremendously from optimization, with more than one order of magnitude improvements in performance. The two perform similarly for the optimized implementation. On the AlphaServer, HP UPC cannot run with fewer than 8 UPC threads because of memory limitations. HP UPC performs best for the nonoptimized version. Berkeley UPC performs best for the optimized version. All systems benefit by at least an order of magnitude from optimization, but HP UPC scales poorly in this case. MuPC exhibits erratic performance for the nonoptimized versions on the x86 cluster and AlphaServer because when the number of threads is 4 a large proportion of

the shared accesses are local shared accesses. The performance difference between remote shared accesses and local shared accesses is obscured by the huge overall performance improvement after optimization. Intrepid UPC running on the T3E shows good scalability for both the nonoptimized and the optimized code. It benefits from optimization by more than one order of magnitude.

MPI and UPC NAS benchmark performance Measurements of the original NPB [1] MPI-based benchmarks run about twice as fast as the optimized UPC-based NPB codes on all three platforms. Part of this performance difference may be attributed to the maturity of UPC compilers and run time systems. Another part of this difference may be due to MPI's fitness for the NAS benchmarks, though not all of them are coarse-grained. The benchmarks used in this work were not written "from scratch" in the UPC style but were translated from the MPI versions to UPC, so they do not necessarily take advantage of the asynchronousness natural to UPC's relaxed memory access mode but are more attuned to the bulk synchronous style of MPI. Current UPC compilers do not exploit the greatest part of the optimizations available by instruction reordering offered by UPC's relaxed memory access mode. Finally, distributed memory platforms such as the x86 cluster and the SC-40 were not designed to minimize the costs of fine-grained accesses. The T3E can perform low latency, fine grained accesses but the most recently available compiler for that platform is not as mature as those for the distributed shared memory platforms. Thus, the jury is still out on UPC's suitability as a shared memory programming language. Needed to resolve this question is further development of compilers and run time systems, development of benchmarks suitable for fine grained computations, and new parallel architectures to facilitate the investigation of the partitioned shared memory programming model.

6. Summary and conclusion

This work developed a UPC STREAM benchmark to measure the bandwidths of many shared memory access patterns. These bandwidths were measured on a Linux x86 cluster, an AlphaServer SC and a Cray T3E, using four UPC implementations: MuPC, HP UPC, Berkeley UPC and Intrepid UPC. The performance of five UPC NPB 2.4 benchmarks were also measured on these systems.

The UPC STREAM benchmarking results show that there is a significant overhead caused by UPC language features. This overhead reduces the performance of local shared accesses compared to the performance of private memory accesses. UPC implementations should attempt to minimize this overhead. The remote reference caching in MuPC and HP UPC helps improve performance of unit stride accesses but unstructured accesses and non-unit stride

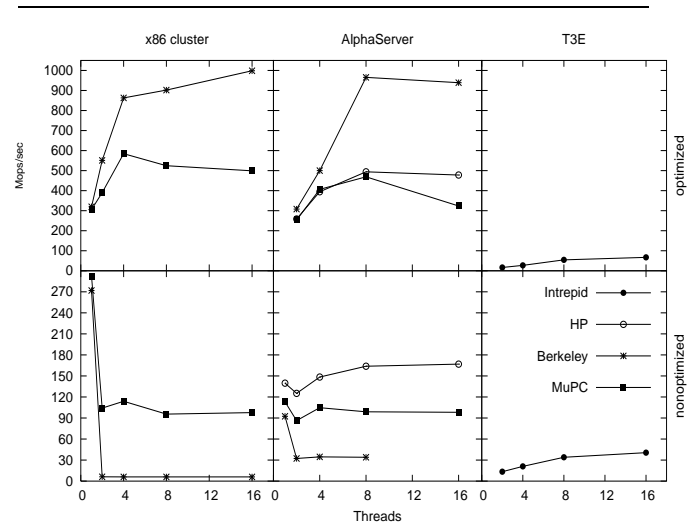


Figure 4. CG benchmark performance. Upper is the fully optimized version, lower is the nonoptimized version. Note: Measurements for Berkeley UPC on the AlphaServer for the nonoptimized version were collected using version 2.0.

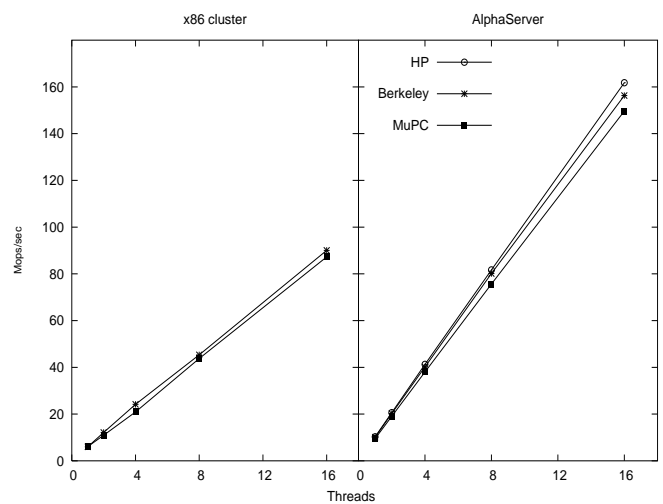


Figure 5. EP benchmark performance. This benchmark failed Intrepid UPC. Note: Measurements for Berkeley UPC on AlphaServer were collected using version 2.0.

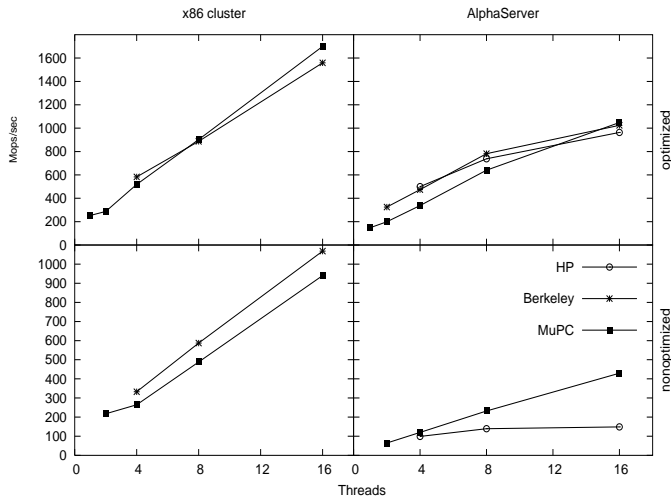


Figure 6. FT benchmark performance. Upper is the fully optimized version, lower is the nonoptimized version. This benchmark failed Intrepid UPC. The nonoptimized version also failed with Berkeley UPC on the AlphaServer. Note: Measurements for Berkeley UPC on the AlphaServer were collected using version 2.0.

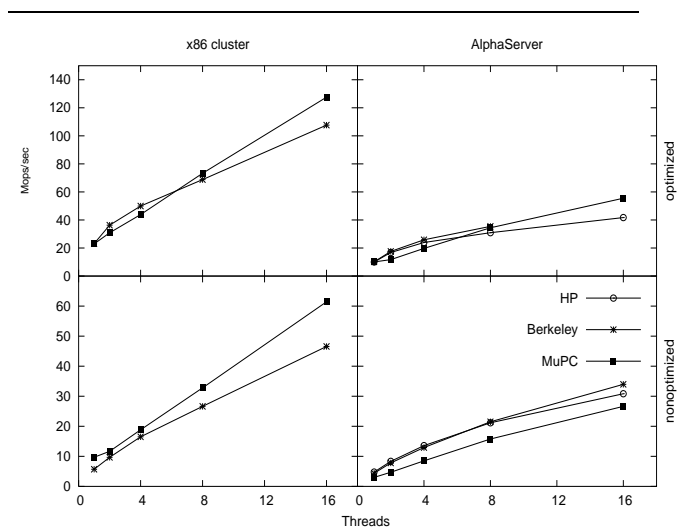


Figure 7. IS benchmark performance. Upper is the fully optimized version, lower is the nonoptimized version. This benchmark failed Intrepid UPC. Note: Measurements for Berkeley UPC on the AlphaServer for the nonoptimized version were collected using version 2.0.

accesses need more sophisticated optimization techniques such as prefetching. The operation of shared copy (copy between two locations with the same affinity to a remote thread) calls for optimization from all UPC implementations. Since this type of shared access needs little inter-thread communication, the performance should be close to that of local shared accesses.

Some implementations fail unexpectedly when running the NBP benchmarks. More stable, reliable and easier-to-use implementations are needed. Hand-tuning optimization techniques such as remote reference prefetching and pointer-to-shared privatization are vital for performance. Almost all kernels studied (except for only the **EP** kernel) benefit greatly from these optimizations.

The **CG** and the **MG** benchmarks show that Berkeley UPC is better than other implementations at handling unstructured, fine-grain shared memory accesses. On the other hand, these benchmarks expose the very ineptness of MuPC in this aspect. HP UPC falls somewhere in between Berkeley UPC and MuPC. The two benchmarks are the only ones that Intrepid UPC runs successfully. Intrepid UPC performs unexpectedly poorly for **CG**, this is against the high-bandwidth nature of the shared memory on the T3E. Intrepid UPC also gives lukewarm performance for **MG**, but it scales well for this application. **MG** is a computation bound kernel, so this performance may be attributed to the slow

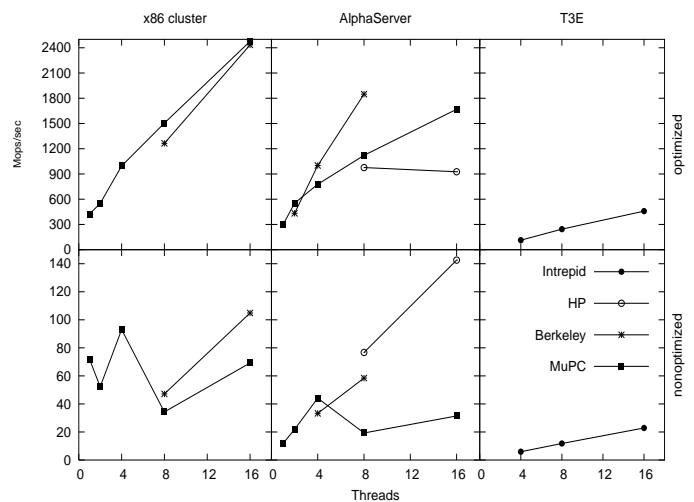


Figure 8. MG benchmark performance. Upper is the fully optimized version, lower is the nonoptimized version.

CPU speed on the T3E.

Although MuPC largely lags behind others in the bandwidths of remote shared accesses, its performance on NPB kernels is comparable to that of others, even better than others in a few cases. This is because in many cases the number of remote shared accesses can be significantly reduced by hand-tuning optimizations or by remote reference caching. This leads us to the conclusion that, while the latency and the bandwidth of the communication layer is usually hard to improve for the current systems, it is more rewarding to invest efforts in developing compiler optimization or run-time optimization techniques to reduce the cost of remote shared accesses.

Acknowledgments

We wish to thank Hewlett-Packard for providing the AlphaServer SC cluster, thank Brian Wibecan and Tanya Klinchina in the UPC group at HP for their many helps on MuPC development, and thank Dan Bonachea at the University of California at Berkeley for much helpful discussion. We also wish to thank the anonymous reviewers who offered constructive comments about this paper.

References

- [1] D. Bailey, E. Barszcz, and J. Barton. The NAS Parallel Benchmark RNR. Technical Report RNR-94-007, NASA Ames Research Center, Mar. 1994.
- [2] K. Berlin, J. Huan, M. Jacob, G. Kochhar, J. Prins, W. Pugh, P. Sadayappan, J. Spacco, and C.-W. Tseng. Evaluating the Impact of Programming Language Features on the Performance of Parallel Applications on Cluster Architectures. In *Languages and Compilers for Parallel Computing (LCPC)*, 2003.
- [3] F. Cantonnet and T. El-Ghazawi. UPC Performance and Potential: A NPB Experimental Study. In *Proceedings, Supercomputing 2002: Baltimore, Maryland*, Nov. 2002.
- [4] F. Cantonnet, Y. Yao, S. Annareddy, A. Mohamed, and T. El-Ghazawi. Performance Monitoring and Evaluation of a UPC Implementation on a NUMA Architecture. In *Proceedings of International Parallel and Distributed Processing Symposium*, 2004.
- [5] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, May 1999.
- [6] W. Chen, D. Bonachea, J. Duell, P. Husbands, C. Iancu, and K. Yelick. A Performance Analysis of the Berkeley UPC Compiler. In *Proceedings of 17th Annual International Conference on Supercomputing (ICS)*, 2003.
- [7] T. El-Ghazawi, W. Carlson, and J. Draper. *UPC Language Specifications*, Oct. 2003. http://www.gwu.edu/~upc/docs/upc_spec_1.1.1.pdf.
- [8] T. El-Ghazawi and S. Chauvin. UPC Benchmarking Issues. In *Proceedings of ICPP (2001)*, 2001.
- [9] Hewlett-Packard. Compaq UPC for Tru64 UNIX, 2004. <http://www.hp.com/go/upc>.
- [10] Intrepid Technology. Intrepid UPC Home Page, 2004. <http://www.intrepid.com/upc>.
- [11] ISO/IEC. *Programming Languages - C, ISO/IEC 9989*, May 2000.
- [12] J. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. <http://www.cs.virginia.edu/stream/>.
- [13] Michigan Technological University. UPC Projects at MTU. <http://www.upc.mtu.edu>.
- [14] J. Savant. MuPC: A Run Time System for Unified Parallel C. Master's thesis, Department of Computer Science, Michigan Technological University, 2002.
- [15] UC Berkeley. Berkeley Unified Parallel C Home Page, 2004. <http://upc.nersc.gov>.
- [16] UC Berkeley. GASNet Home Page, 2004. <http://www.cs.berkeley.edu/~bonachea/gasnet>.