Evaluation of UPC on the Cray X1E

Richard Barrett[1], Yiyi Yao[2], and Tarek El-Ghazawi[2]

[1] Future Technologies Group, Computer Science and Mathematics Division,
Oak Ridge National Laboratory, Oak Ridge, TN 37931.
[2] Department of Electrical and Computer Engineering
and the High-Performance Computing Lab (HPCL),
The George Washington University, Washington, DC 20052.

June 1, 2006

Abstract

Unified Parallel C (UPC) is a parallel programming language that provides the programmer with a syntax for exposing parallelism and managing data locality of an application. Because UPC continues to gain attention as an alternative programming model for distributed memory applications it is important to evaluate how this language interacts with one of today's most powerful, contemporary architectures: the Cray X1E.

Last year we presented an evaluation of UPC on the Cray X1 at Oak Ridge National Laboratory. Since then this X1 has been upgraded to an X1E. In this paper we evaluate UPC on the Cray X1E, examining how the compiler exploits the important features of this architecture's vector processors and multistreaming. Using the NAS Parallel Benchmark Suite, we show that UPC provides a high-performance, scalable programming model that can take advantage of the X1E architecture. In the process we show how users can leverage the power of the X1E for their applications, by identifying areas where compiler analysis can be more aggressive and where there are potential performance caveats.

1 Introduction

Unified Parallel C (UPC) is an explicit partitioned global address space (PGAS) extension of the ISO C programming language[8, 5]. Developed by a consortium of vendors, researchers, and practitioners and leveraging concepts from previous related research, UPC inherits and maintains the efficient syntax of C while providing the programmer low-level access to the underlying system through its abstract programming and memory models. The global address space view is similar to shared memory paradigms such as OpenMP[3], hiding much of the complexity of private and shared memory from the programmer. For example, a simple assignment statement can cause a remote memory read and a remote memory write, which hides much of the underlying data movement from the application developer. Yet like message passing paradigms such as MPI[11, 9], UPC lets application developers co-locate processing potentially in the same node and avoid unnecessary overhead.

The Cray X1E combines a globally-addressable, distributed shared memory architecture with vector, multi-streaming, and scalar processing capabilities. In this study, we examine the ability of the X1E to execute selected kernels from the NAS Parallel Benchmark suite, implemented in UPC. We study the behaviour of these kernels in response to automatic compiler optimizations as well as to our devised emulations that can mimic the effects of automatic optimizations [6]. We also examine compiler output to determine how well the compiler takes advantage of the architectural features of the Cray X1E, and characterize where improvements may still be possible.

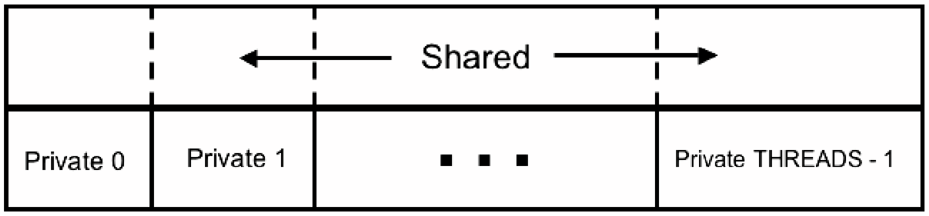

Figure 1: *The UPC Memory and Execution Model*

This paper is organized as follows. Section 2 gives a brief description of the UPC language. Section 3 describes the Cray X1E architecture, focusing on the characteristics of interest that may be exploited by a UPC-based application. Section 4 describes the experimental kernels and strategies used in this report. Section 5 presents the performance measurements, analysis, and observations of our experiments, followed by conclusions in Section 6.

2 Overview of UPC

The most important feature of the UPC language is its memory model. In UPC, application memory consists of two separate spaces: a shared memory space and a private memory space. Figure 1 illustrates the memory and execution model as viewed by UPC applications and programmers. Each UPC thread operates independently, in a single-program-multiple-data (SPMD) fashion, with direct reference to any address in both the shared space and its private space, but not to the private space of any other thread. Two special constants allow a thread to identify its place in the SPMD model: each thread is identified by `MYTHREAD`, a unique integer ranging from 0 to `THREADS`-1, where `THREADS` is the total number of UPC threads. The shared space is logically divided into portions, each with a special association (affinity) to a given thread. With proper declarations, programmers can keep the shared data that is to be primarily manipulated by a given thread (and less frequently accessed by others) associated with that thread. That is, a thread and its pertinent data can be mapped into the same physical memory, exploiting inherent data locality in applications.

Since UPC is an explicit parallel extension of ISO C, all language features of C are already embodied in UPC. In addition, UPC declarations give the programmer control of the distribution of data across the threads. Among the interesting and complementary UPC features is a work-sharing iteration statement, known as `upc_forall`. This statement helps to distribute independent loop iterations across the threads such that iterations and the data processed by them are assigned to the appropriate thread.

UPC defines a number of rich concepts for pointers to both private and shared memory, including dynamic shared memory allocation. Because there is no implicit synchronization in UPC, a broad range of synchronization and memory consistency control constructs are provided. Among the most interesting synchronization concepts is the non-blocking barrier, which allows overlapping local computations and inter-thread communications.

Parallel I/O and collective operation library specifications have been included in the current specification, version 1.2. and are scheduled to be integrated into the next UPC language specifications. For details on these and other language features, the reader is directed to [5] as well as the formal specification[8].

Memory location	*Relative access time*
D-cache	*1X*
E-cache	*2X*
Local (node) memory	*7X*
Remote (off node) memory	*10X-32X*

Table 1: *This table shows the cost of access through the X1E memory hierarchy.*

3 The Cray X1E

The Cray X1E at Oak Ridge National Laboratory (ORNL), named Phoenix, consists of 1024 multi-streaming processors (MSP), each capable of performing 18 GFLOPS, making the system capable of almost 18.5 TFLOPS. Memory bandwidth is very high, roughly half of cache bandwidth. The interconnect functions as an extension of the memory system, offering each node direct access to memory on other nodes at high bandwidth and low latency.

Originally installed for evaluation at ORNL in 2003 as a 256 processor X1, Phoenix was upgraded in 2004 to a production capable 512 processors. In the summer of 2005 Phoenix was again upgraded, this time increasing the clock speed and doubling the number of processors. This upgrade to what is now called the Cray X1E increased the clock speed to 1.13 GHz, up 42% from the X1 800 MHz, effectively increasing peak performance from 12.8 GFLOPS to 18 GFLOPS per MSP. While the density of processors was doubled, the total amount of memory remained the same.

As with the X1, the basic building block of the X1E is a compute module, consisting of MCM, memory, routing logic, and external connectors. Also, four MSPs behave like a traditional SMP. It is here that the processor doubling occurs: an X1 MCM contains one MSP and a module contains one SMP node. An X1E MCM contains two MSPs, meaning each module now contains eight MSPs. Thus two four-MSP-way SMPs comprise a module. The effect is that memory and interconnect bandwidth is now shared by double the number of MSPs on a node. We note that cache performance scales with bandwidth with processor speed, so although the increased clock increases the need for memory bandwidth, it is perhaps less so than imagined.

Most important for partitioned global address space languages like UPC, each processor can directly address memory on any other node. However, unlike the Cray T3E, these remote memory accesses are issued directly from the processors as load and store instructions, transparently executed over the X1E interconnect to the target processor, bypassing the local cache. This mechanism is more scalable than traditional shared memory (costs are listed in Table 1), but is not appropriate for shared-memory programming models, like OpenMP, outside of a given four-MSP node. In contrast to the T3E's E-registers, both scalar and vector loads are blocking primitives, which limits the ability of the system to overlap communication and computation. This remote memory access mechanism is a natural match for distributed-memory programming models, particularly those using one-sided put/get operations, such as UPC and Co-Array Fortran[10].

4 The NAS Parallel Benchmark Suite

The NAS Parallel Benchmarks (NPB) are developed by the Numerical Aerodynamic simulation (NAS) program at NASA Ames Research Center for the performance evaluation of parallel supercomputers [1]. The NPB suite mimics the computation and data movement characteristics of large-scale computation fluid dynamics (CFD) applications, and consists of five kernels (CG, EP, FT, IS, MG) and three pseudo-applications (LU, SP, BT) programs. The bulk of the computation in IS is integer arithmetic. The other benchmarks are floating-point computation intensive, with different communication requirements and intensities. All test problems are defined for strong scaling studies. Problem sets and the number of participating UPC threads or MPI processes are set at compile time.

Our experiments involved the following NAS kernels:

- CG (Conjugate Gradient) computes an approximation to the smallest eigenvalue of symmetric positive definite matrix. This kernel features unstructured grid computations requiring irregular long-range communications.
- EP (Embarrassingly Parallel) can run on any number of processors with little communication. It estimates the upper achievable limits for floating point performance of a parallel computer. This benchmark generates pairs of Gaussian random deviates according to a specific scheme and tabulates the number of pairs in successive annuli.
- FT (Fast Fourier Transform) solves a 3D partial differential equation using an FFT-based spectral method, also requiring long-range communication. FT performs three one-dimensional (1-D) FFT's, one for each dimension.
- IS (Integer sort) is a parallel sorting program based on the bucket sort, requiring a lot of total exchange communication
- MG (MultiGrid) uses a V-cycle multi-grid method to compute the solution of the 3-D scalar Poisson equation, requiring both short and long-range highly structured inter-process communication.

There are several problem sizes (called "classes") in the NPB distribution. We report on the performance of the kernels using the class B problems, with intra-comparisons involving the UPC implementation as well as the official MPI implementation, distributed by NAS, in their NPB 3.2 release.

The UPC NPB implementation used in this study was ported from the MPI and OpenMP[3] versions 2.4. Data that is to be shared is automatically linearly decomposed across the participating threads. Programs distribute their kernels across the participating threads using the UPC construct `upc_forall`. Access to data is controlled using `upc_barrier`[1].

The NPB Fortran/MPI implementation is distributed from `http://www.nas.nasa.gov/Software/NPB`. The UPC implementation is distributed from `http://upc.gwu.edu`.

4.1 Effective use of the X1E

Effective use of the Cray X1E, as with any vector processor based architecture, requires overwhelmingly appropriate use of the vectorization capabilities[2]. The UPC programming language, like C, allows memory pointers, which can result in implicit data dependencies and address aliasing issues, so compilers tend to be conservative in selecting vectorization and multi-streaming in order to assure correctness. This may cause the compiler to miss opportunities for applying vectorization and multi-streaming optimizations. Where possible, this can be overcome by through proper use of compile time directives, i.e. pragmas. Pragmas allow programmers to alert the compiler to optimization opportunities which might not otherwise be attempted. We used two main pragmas, `#pragma _CRI ivdep` and the `#pragma _CRI concurrent`.

[1]This global barrier is an expensive operation on most architectures, and besides, it may be injecting synchronization where none is needed, further degrading performance. Future work calls for replacing the `upc_barrier` with `upc_fence` or `upc_wait` functionality where possible. We expect this to increase performance by allowing for greater asynchronous behaviour and as well as enabling overlap of computation and inter-thread communication. Other modifications to the UPC implementation are also under consideration.

[2]In MSP mode, the vector length should be at least 256 elements.

```
1155.  1 2 r-------<   for ( i1 = d1; i1 <= mm1-1; i1++)
1156.  1 2 r             {
1157.  1 2 r                u((2*i3-d3-1), (2*i2-d2-1), (2*i1-d1-1)) =
1158.  1 2 r                  u((2*i3-d3-1), (2*i2-d2-1), (2*i1-d1-1))
1159.  1 2 r                    + z((i3-1), (i2-1), (i1-1));
1160.  1 2 r------->      }

1155.  m 2        #pragma _CRI concurrent
1156.  m 2        #pragma _CRI ivdep
1157.  m 2 MV---<      for ( i1 = d1; i1 <= mm1-1; i1++)
1158.  m 2 MV            {
1159.  m 2 MV               u((2*i3-d3-1), (2*i2-d2-1), (2*i1-d1-1)) =
1160.  m 2 MV                 u((2*i3-d3-1), (2*i2-d2-1), (2*i1-d1-1))
1161.  m 2 MV                   + z((i3-1), (i2-1), (i1-1));
1162.  m 2 MV--->        }
```

Table 2: *This table compares the loopmark listings of the FT benchmark, before and after the insertion of the pragmas intended to guide the compiler to vectorization opportunities.*

The `ivdep` directive tells the compiler to ignore the possible vector dependencies in the loop and thus to fully vectorize the loop. The `concurrent` directive tells the compiler that there are no data dependencies present in the loop, and thus they may be vectorized and multi-streamed. For more detailed explanations and other pragma directives, refer to the CrayDoc online [2].

Guided by the loopmark listing utility, we inserted pragmas throughout the kernels. Table 2 shows the loopmarking listing for the FT kernel, demonstrating the effects of the pragams: after insertion of the compiler directives, the inner loop has been fully multi-streamed and vectorized. Further, the pragmas allowed the outer loop to be non-partitioned streamed. (Not shown here).

5 Performance

In this section we present the performance results for the NPB kernels on Phoenix. The performance analysis is done from two different perspectives. First we analyze the impact of the hardware upgrade from X1 to X1E. Then we show the benefit of using Cray directives within the NPB code run on the X1E. Finally, we show scaling results in comparison with the Fortran/MPI NPB implementation.

Phoenix is currently running UNICOS/mp 3.1.0.7, and we operated within the latest default programming environment, version 5.5.0.1. Results from the Cray X1, prior to the X1E upgrade, are presented for comparison, and were run under the 5.3.0.2 Programming Environment, with Unicos/mp operating system version 2.5.3.5.

Figure 2 shows the performance of the selected UPC NPB kernels (CG, EP, FT, IS, and MG) on both the X1E and X1. Each graph includes three measurements: the performance of the kernels on X1E with pragma directives inserted, the performance of the kernels on X1E without pragma directives, and the performance of the kernels on the X1[7], also without compiler pragmas..

While we note that UPC programs can have three different levels of optimizations[4] (denoted as O0, the straightforward UPC code, O1, the privatized code and O3, the privatized and software prefetched code), our previous work shows such optimizations do not result in improved performance on this architecture. Our interpretation is that Cray UPC compiler does match the language with the machine architecture. Addressing this issue could result in improved performance for some or all kernels. Thus the performance data presented here use the straightforward UPC implementation.

With regard to the machine upgrade, the CG, FT, IS, and MG kernels all showed marked

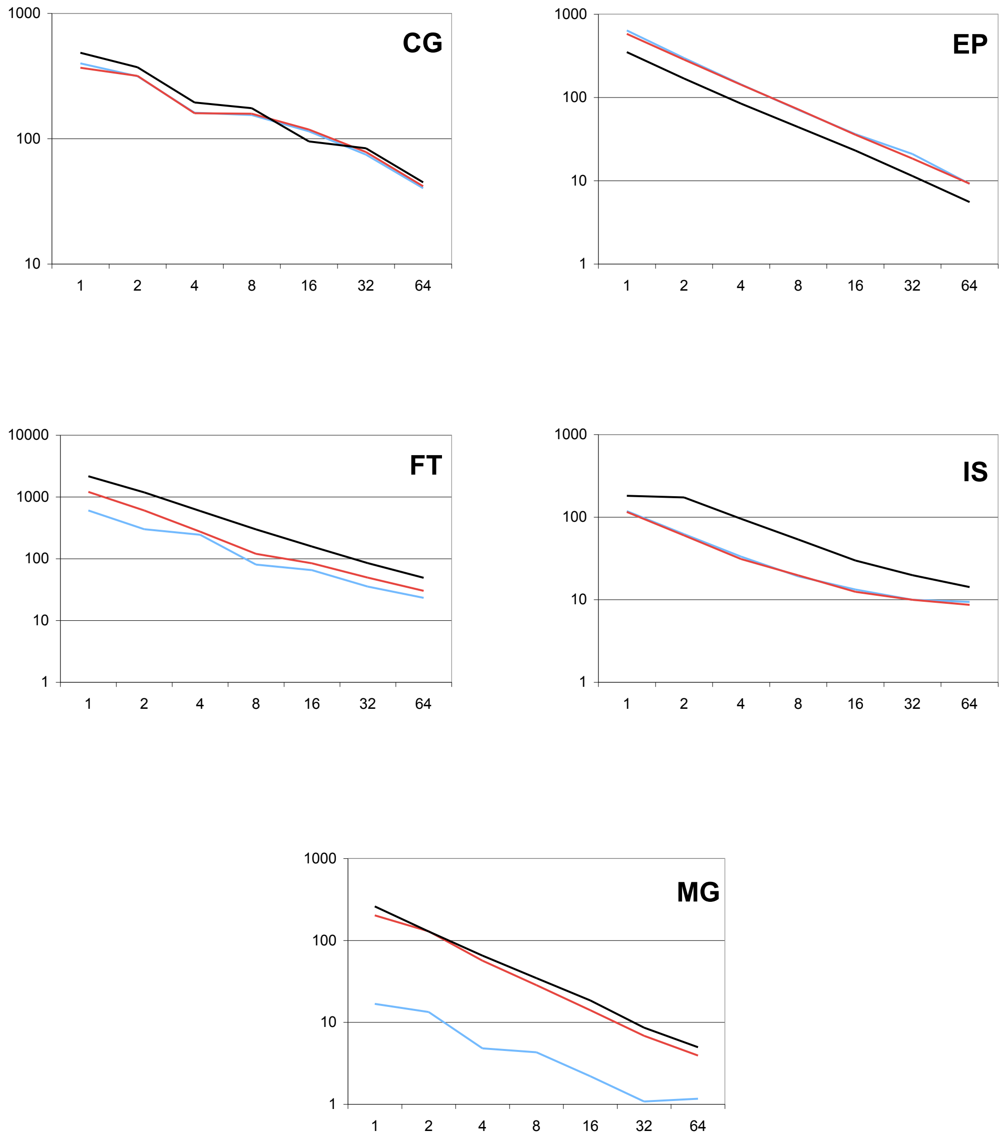

Figure 2: *NPB kernels performance. The x-axis represents the number of processors; the y-axis represents execution time, in seconds. The blue line represents the X1E with pragmas, the red line represents the X1E without the pragmas, and the yellow line represents the data collected last year on the X1.*

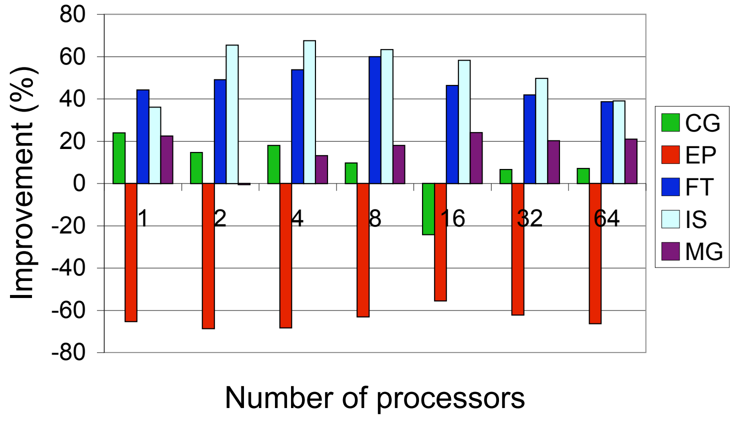

Figure 3: *This graph illustrates the change in UPC NPB kernel performance soley as a function of the machine upgrade from an X1 to an X1E. (The X1E performance does not reflect the use of compiler directives.)*

improvement. (See Figure 3.) The performance of EP degraded significantly, a result not attributable to UPC since the degradation appears for a single thread, and remains consistent as the thread count increases. Anyway, inter-thread data sharing is insignificant, a claim supported by the discussion and results shown in section 5.1. Perhaps the change in EP performance is a result of a change in the random number generator.

The addition of the pragmas increased performance for some of the kernels (FT and most significantly, MG) while the performance of the other kernels were relatively unchanged. The latter is attributed to the fact that some inter-loop dependencies are in fact real, and thus some important loops cannot be vectorized. Table 3 shows runtime statistics for CG and FT as reported by the Cray Performance Analysis Tool (PAT). As expected, FT shows a meaningful increase in vectorization, while CG does not.

5.1 Scalability

There are two major considerations involved in the runtime performance of an application that is executed in a parallel processing environment. First is the computational performance on a processor. Second, and arugably the most important, is scalability of performance as an increasing number of processors is brought to bear on the problem. The former is a function of the compiler's ability to generate effective instructions for use by the processor. The latter is a function of the ability of the architecture to map the semantics of a program to the multi-processor environment, which is the focus of this section.

Since UPC is a superset of the C programming language, it would be best to compare the results shown in section 5 with an analogous NPB version implemented in C. Unfortunately we do not have access to such a version. We can, however, compare our results with the Fortran/MPI version. Since

Counter	Rate	Count	
CG without pragmas			
Cycles	13.848 secs	7823851782	
Instructions graduated	84.143M/sec	1165167166	
Vector instructions	13.931M/sec	192915129	(16.6%)
Scalar instructions	70.211M/sec	972252037	(83.4%)
Vector ops	253.284M/sec	3507355278	
CG with pragmas			
Cycles	13.199 secs	7457371084	
Instructions graduated	83.432M/sec	1101215162	
Vector instructions	14.633M/sec	193135129	(17.5%)
Scalar instructions	68.800M/sec	908080033	(82.5%)
Vector ops	266.792M/sec	3521354134	
FT without pragmas			
Cycles	10.083 secs	5697094082	
Instructions graduated	917.668M/sec	9253173658	
Vector instructions	123.785M/sec	1248170501	(13.5%)
Scalar instructions	793.883M/sec	8005003157	(86.5%)
Vector ops	1614.789M/sec	16282482799	
FT with pragmas			
Cycles	6.808 secs	3846326801	
Instructions graduated	1068.797M/sec	7276004980	
Vector instructions	177.170M/sec	1206111825	(16.6%)
Scalar instructions	891.627M/sec	6069893155	(83.4%)
Vector ops	2384.791M/sec	16234840932	

Table 3: *This table shows the effects of the pragmas described in Section 5. These X1E runtime statistics, as reported by the Cray Programming Analysis Tool (PAT), are for the CG and FT kernels.*

our interest here is on scalability, we normalize the results to the single processor performance, then compare performance as the number of processors increases.

Graphical representations of this view are shown in Figure 4. UPC shows a strong corollation with the Fortran/MPI implementation for CG and EP. FT and MG show a reasonable corollation, which might be improved with careful attention to their long range communication requirements. IS corollation is poor, attributed to the all-to-all communication requirement. The strong performance of the Fortran/MPI implementation is attributable to the much used `MPI_Alltoall`. Again, careful attention to this operation in the UPC implementation should result in a stronger corollation. Work is underway in these areas.

6 Concluding Remarks

Last summer the ORNL Cray X1 was upgraded to an X1E. In addition to doubling the number of MSP processors and increasing the MSP peak performance, the amount of memory available to an MSP was, in effect, reduced by half. In this report we examined the effects of these changes on UPC-based application performance by comparing performance results of the NAS Parallel Benchmark suite on the X1 with the recently upgraded X1E.

We found that in general, the upgrade has no noticeable effect on scaling of the UPC-based NPB kernels. We also showed that vectorization and multi-streaming opportunities can be improved through careful use of compiler directives. In particular, the MG kernel showed a dramatic improvement in performance, while the FT kernel showed meaningful improvement. Little opportunities for such improvement appear to exist in the CG, EP, and IS kernels.

Although we saw a degradation in performance for the EP kernel, we showed that this is not attributable to UPC, but rather to some change in the C programming environment. Unfortunately we can't investigate this further since the X1 is no longer available to us.

Performance of the UPC implementation of the NPB is generally poor in comparison to the Fortran/MPI implementation. However, we attribute this mostly to the difference in capability of the Fortran and C compilers. Yet we do see that improvements in the use of UPC could be made to the FT, IS, and MG kernels, mostly with regard to large scale collective communication requirements. This work is currently being pursued.

References

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. 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. *Intl. Journal of Supercomputer Applications*, Fall 1991.

[2] Cray Inc. Online Cray Documentation. `http://www.cray.com/craydoc`, 2006.

[3] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1), 1998.

[4] T. El-Ghazawi and S. Chauvin. Upc benchmarking issues. In *International Conference on Parallel Processing*, 2001.

[5] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared Memory Programming.* John Wiley and Sons, 2005. ISBN: 0-471-22048-5.

[6] Tarek A. El-Ghazawi and François Cantonnet. Upc performance and potential: A npb experimental study. In *Proceedings of the IEEE/ACM Conference on Supercomputing SC'02*, November 2002.

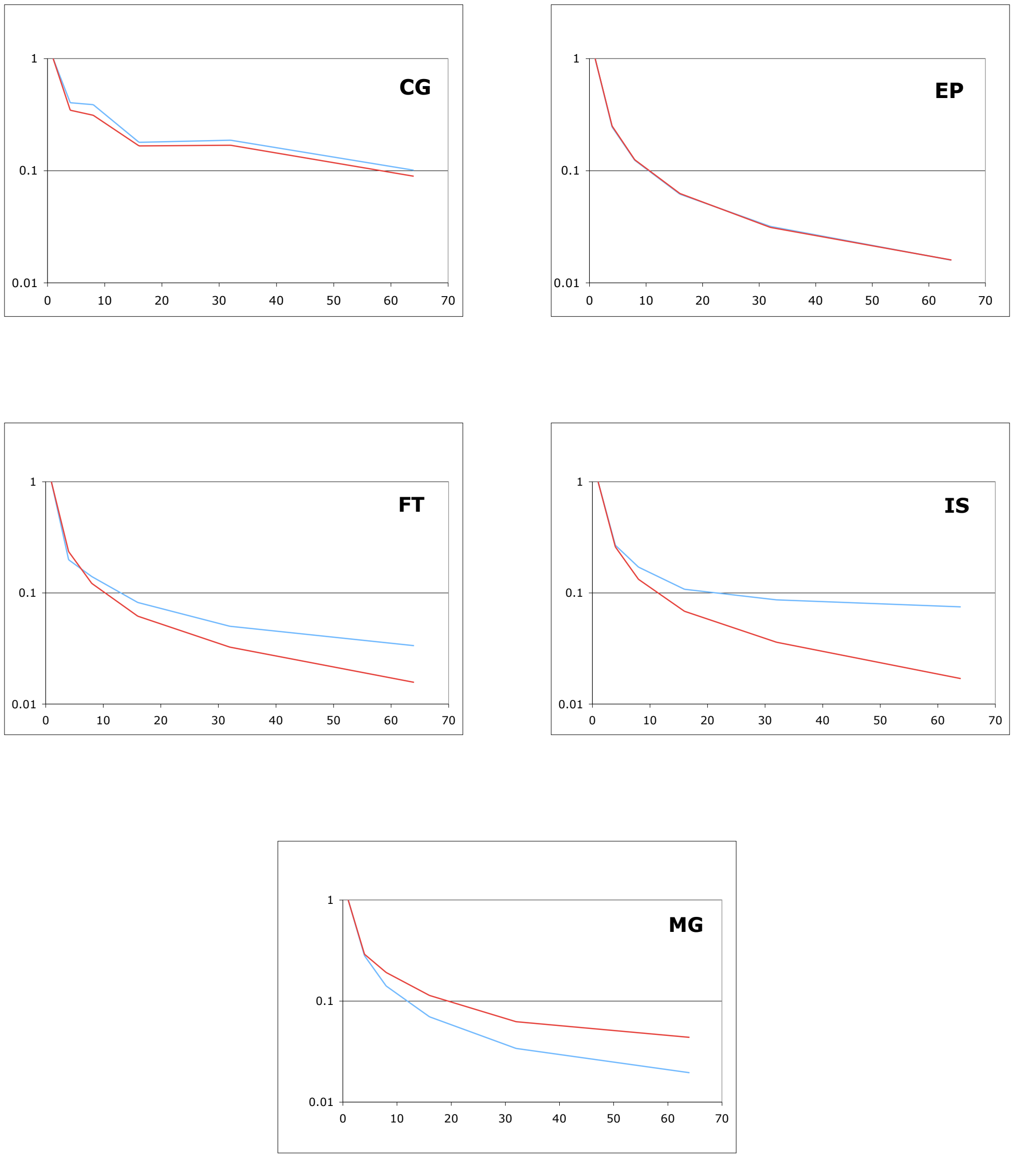

Figure 4: *This figure compares the performance of the UPC and Fortran/MPI implementations of the NPB kernels described in this report, using the class B problem, on the Cray X1E. The x-axis represents the number of processors; the y-axis represents speedup (in relation to the single processor result). The blue line is the UPC implementation and the red line is the Fortran/MPI implementation.*

[7] Tarek A. El-Ghazawi, François Cantonnet, Yiyi Yao, and Jeffrey Vetter. Evaluation of UPC on the Cray X1. In *Cray User Group Meeting*, 2004.

[8] Tarek A. El-Ghazawi, William W. Carlson, and Jesse M. Draper. UPC language specification, version 1.1.1. `http://www.gwu.edu/∼upc/documentation.html`.

[9] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI: The Complete Reference: Volume 2 - The MPI-2 Extentions.* The MIT Press, 1998.

[10] R.W. Numrich and J.K. Reid. Co-Array Fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, 1998. `http://www.co-array.org`.

[11] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference: Volume 1 - 2nd Edition.* The MIT Press, 1998.

About the authors

Richard Barrett is a computational scientist in the Future Technologies Group in the Computer Science and Mathematics Division (CSM) of Oak Ridge National Laboratory (ORNL), where he is also a member of the Scientific Computing Group in the National Center for Computational Science. He can be reached at *rbarrett@ornl.gov.* Yiyi Yao is a Research Assistant in the High Performance Computing Laboratory at The George Washington University. Tarek El-Ghazawi is professor of the Electrical and Computer Engineering department at The George Washington University, and director of the High Performance Computing Laboratory. They can be reached at *gwu-upc@hermes.gwu.edu.*