Early Evaluation of the Cray X1 [*]

T. H. Dunigan, Jr. [†]

M. R. Fahey [‡]

J. B. White III [§]

P. H. Worley [¶]

Abstract

Oak Ridge National Laboratory installed a 32 processor Cray X1 in March, 2003, and will have a 256 processor system installed by October, 2003. In this paper we describe our initial evaluation of the X1 architecture, focusing on microbenchmarks, kernels, and application codes that highlight the performance characteristics of the X1 architecture and indicate how to use the system most efficiently.

1 Introduction

The mainstream of parallel supercomputing in the U.S. has for some time been dominated by clusters of commodity shared-memory processors (SMPs). The Department of Energy's (DOE) Accelerated Strategic Computing Initiative (ASCI), the National Science Foundation's (NSF) Extensible Terascale Facility (ETF), and, with a few exceptions, the Department of Defense's (DOD) High Performance Computing Modernization Program are all following this path. This technology path provides excellent price/performance when measured from the theoretical peak speed, but many applications (e.g., climate, combustion) sustain only a small fraction of this speed. Indeed, over the past decade, the sustained fraction of peak performance achieved by these applications has been falling steadily. Many factors account for this, but foremost are the increasing disparity between the speed of processors and memory and the design of processors for commodity applications rather than high-end scientific simulations. Illustrative of an alternative path is the Japanese Earth Simulator, which attracted attention (including several Gordon Bell prizes at the SC2002 conference) by sustaining a large fraction of its 40 TFlop/sec peak performance on several scientific applications. This performance advantage was realized by employing relatively few (5,120) powerful (8 GFlop/sec) vector processors connected to high-bandwidth memory and coupled by a high-performance network.

The X1 is the first of Cray's new scalable vector systems [6]. The X1 is characterized by high-speed custom vector processors, high memory bandwidth, and a high-bandwidth, low-latency interconnect linking the nodes. The efficiency of the processors in the Cray X1 is anticipated to be comparable to the efficiency of

[*]This research was sponsored by the Office of Mathematical, Information, and Computational Sciences, Office of Science, U.S. Department of Energy under Contract No. DE-AC05-00OR22725 with UT-Batelle, LLC. Accordingly, the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

[†]Computer Science and Mathematics Division, Oak Ridge National Laboratory, P.O. Box 2008, Bldg. 5600, Oak Ridge, TN 37831-6016 (DuniganTHJr@ornl.gov)

[‡]Computer Science and Mathematics Division, Oak Ridge National Laboratory, P.O. Box 2008, Bldg. 5600, Oak Ridge, TN 37831-6008 (FaheyMR@ornl.gov)

[§]Computer Science and Mathematics Division, Oak Ridge National Laboratory, P.O. Box 2008, Bldg. 5600, Oak Ridge, TN 37831-6008 (WhiteJBIII@ornl.gov)

[¶]Computer Science and Mathematics Division, Oak Ridge National Laboratory, P.O. Box 2008, Bldg. 5600, Oak Ridge, TN 37831-6016 (WorleyPH@ornl.gov)

SC'03, November 15-21, 2003, Phoenix, Arizona, USA

the NEC SX-6 processors in the Earth Simulator on many computational science applications. A significant feature of the Cray X1 is that it attempts to combine the processor performance of traditional vector systems with the scalability of modern microprocessor-based architectures.

Oak Ridge National Laboratory (ORNL) recently procured a Cray X1 for evaluation. A 32 processor system was delivered at the end of March, 2003, and delivery of a 256 processor system will be complete by the end of September, 2003. Processor performance, system performance, and production computing readiness are all being evaluated. The evaluation approach focuses on scientific applications and the computational needs of the science and engineering research communities. The primary tasks of this evaluation are to:

1. Evaluate benchmark and application performance and compare with systems from other high performance computing (HPC) vendors.
2. Determine the most effective approaches for using the Cray X1.
3. Evaluate system and system administration software reliability and performance.
4. Predict scalability, both in terms of problem size and number of processors.

In this paper we describe initial results from the first two evaluation tasks. We begin with overviews of the Cray X1 architecture and of the evaluation plan, followed by a snapshot of the evaluation results as of the middle of July, 2003. Most results were collected on the 32 processor system delivered in March, but a few results were collected on the 64 processor system that became available in July.

2 Cray X1 Description

The Cray X1 is an attempt to incorporate the best aspects of previous Cray vector systems and massively-parallel-processing (MPP) systems into one design. Like the Cray T90, the X1 has high memory bandwidth, which is key to realizing a high percentage of theoretical peak performance. Like the Cray T3E, the design has a high-bandwidth, low-latency, scalable interconnect, and scalable system software. And, like the Cray SV1, the X1 design leverages commodity CMOS technology and incorporates non-traditional vector concepts, like vector caches and multi-streaming processors.

The Cray X1 is hierarchical in processor, memory, and network design. The basic building block is the multi-streaming processor (MSP), which is capable of 12.8 GFlop/sec for 64-bit operations. Each MSP is comprised of four single-streaming processors (SSPs), each with two 32-stage 64-bit floating point vector units and one 2-way super-scalar unit. The SSP uses two clock frequencies, 800 MHz for the vector units and 400 MHz for the scalar unit. Each SSP is capable of 3.2 GFlop/sec for 64-bit operations. The four SSPs share a 2 MB "Ecache".

The Ecache has sufficient single-stride bandwidth to saturate the vector units of the MSP. The Ecache is needed because the bandwidth to main memory is not enough to saturate the vector units without data reuse - memory bandwidth is roughly half the saturation bandwidth. This design represents a compromise between non-vector-cache systems, like the SX-6, and cache-dependent systems, like the IBM p690, with memory bandwidths an order of magnitude less than the saturation bandwidth. Because of its short cache lines and extra cache bandwidth, random stride scatter/gather memory access on the X1 is expected to be just a factor of two slower than stride-one access, not the factor of eight or more seen with typical cache-based systems like the IBM p690, HP Alpha, or Intel IA-64. The Cray X1's cache-based design deviates from the full-bandwidth design model only slightly. Each X1 processor is designed to have more single-stride bandwidth than an SX-6 processor; it is the higher peak performance that creates an imbalance. A relatively small amount of data reuse, which most modern scientific applications do exhibit, could enable a very high percentage of peak performance to be realized, and worst-case data access should still provide double-digit efficiencies.

The X1 compiler's primary strategies for using the eight vector units of a single MSP are parallelizing a (sufficiently long) vectorized loop or parallelizing an unvectorized outer loop. The effective vector length of the first strategy is 256 elements, like for the NEC SX-6. The second strategy, which attacks parallelism at a different level, allows a much shorter vector length of 64 elements for a vectorized inner loop. Cray also supports the option of treating each SSP as a separate processor.

Four MSPs and a flat, shared memory of 16 GB form a Cray X1 node. The memory banks of a node provide 200 GByte/sec of bandwidth, enough to saturate the paths to the local MSPs and service requests from remote MSPs. Each bank of shared memory is connected to a number of banks on remote nodes, with an aggregate bandwidth of roughly 50 GByte/sec between nodes. This represents a byte per flop of interconnect bandwidth per computation rate, compared to 0.25 bytes per flop on the Earth Simulator and less than 0.1 bytes per flop expected on an IBM p690 with the maximum number of Federation connections. The collected nodes of an X1 have a single system image.

A single four-processor X1 node behaves like a traditional SMP, but each processor has the additional capability of directly addressing memory on any other node (like the T3E). Remote memory accesses go directly over the X1 interconnect to the requesting processor, bypassing the local cache. This mechanism is more scalable than traditional shared memory, but it is not appropriate for shared-memory programming models, like OpenMP [16], outside of a given four-processor node. This remote memory access mechanism is a good match for distributed-memory programming models, particularly those using one-sided put/get operations.

In large configurations, the Cray X1 nodes are connected in an enhanced 3D torus. This topology has relatively low bisection bandwidth compared to crossbar-style interconnects, such as those on the NEC SX-6 and IBM SP. Whereas bisection bandwidth scales as the number of nodes, $O(n)$, for crossbar-style interconnects, it scales as the 2/3 root of the number of nodes, $O(n^{2/3})$, for a 3D torus. Despite this theoretical limitation, mesh-based systems, such as the Intel Paragon, the Cray T3E, and ASCI Red, have scaled well to thousands of processors.

3 Evaluation Overview

Performance evaluation and application experts at a number of Department of Energy national laboratories recently collaborated to specify an evaluation plan for the Cray X1 [3]. A hierarchical approach to the investigation of the X1 architecture was proposed, including microbenchmarking, application kernel optimization and benchmarking, and application benchmarking. The methodology used here was motivated by this evaluation plan. In this section we give a quick overview of the relevant benchmarking activities, as described in the evaluation plan. The results presented in the paper represent only a subset of this larger evaluation plan. The overview is meant to provide context to the work, as well as to indicate other work and future activities.

3.1 Microbenchmarking

The objective of the microbenchmarking is to characterize the performance of the underlying architectural components of the Cray X1. Both standard benchmarks and customized benchmarks are being used. The standard benchmarks allow component performance to be compared with other computer architectures. The custom benchmarks permit the unique architectural features of the Cray X1 (distributed vector units, and cache and global memory) to be tested with respect to the target applications.

In the architectural-component evaluation we assess the following:

1. Scalar and vector arithmetic performance, including varying vector length and stride and identifying what limits peak computational performance.

2. Memory-hierarchy performance, including cache, local memory, shared memory, and global memory. These tests utilize both System V shared memory and the SHMEM primitives [8], as well as Unified Parallel C (UPC) [5] and Co-Array Fortran [15]. Of particular interest is the performance of the shared memory and global memory, and how remote accesses interact with local accesses.

3. Task and thread performance, including performance of thread creation, locks, semaphores, and barriers. Of particular interest is how explicit thread management compares with the implicit control provided by OpenMP, and how thread scheduling and memory/cache management (affinity) perform.

4. Message-passing performance, including intra-node and inter-node MPI [13] performance for one-way (ping-pong) messages, message exchanges, and aggregate operations (broadcast, all-to-all, reductions,

barriers); message-passing hotspots and the effect of message passing on the memory subsystem are of particular interest.

5. System and I/O performance, including a set of tests to measure OS overhead (context switches), virtual memory management, low-level I/O (serial and parallel), and network (TCP/IP) performance.

3.2 Kernels and Performance Optimization

The Cray X1 has a number of unique architectural features, and coding style and parallel programming paradigm are anticipated to be as important as the compiler in achieving good performance. In particular, early indications are that coding styles that perform well on the NEC vector systems do not necessarily perform well on the X1.

Unlike traditional vector machines, the vector units are independent within the X1 MSP. As such, the compiler must be able to identify parallelism for "multistreaming" in addition to vectorization, and to exploit memory access locality in order to minimize cache misses and memory accesses. As described earlier, an alternative approach is to treat the four SSPs making up an MSP as four (3.2 GFlop/sec peak) vector processors, assigning separate processes to each. This takes the responsbility of assigning work to the vector units away from the compiler and gives it to the user, and may achieve higher performance for certain applications.

Each node in the Cray X1 has 4 MSPs in a shared-memory configuration with a network that supports globally addressable memory. How best to program for the hierarchical parallelism represented by clusters of SMP nodes is typically both system and application specific. The addition of both multistreaming and vectorization makes it even less clear which of the many shared- and distributed-memory programming paradigms are most appropriate for a given application. Cray currently supports the MPI programming paradigm, as well as the SHMEM one-sided communication library and the Co-Array Fortran and UPC parallel programming languages. OpenMP is expected to be supported later in the summer of 2003. System V shared-memory, Multi-Level Parallelism (MLP) [17], and Global Arrays [14] are also possible programming models for the Cray X1.

These usability and optimization issues are difficult to examine with either microbenchmarks, which do not measure the difficulty of implementing a given approach in an application code, and full application codes, in which it is not feasible to modify to try all of the different approaches. The approach we take is to identify representative kernels and use these to examine the performance impact of a variety of coding styles. The goal is not just to choose the best programming paradigm, assuming that one is clearly better than the others, but also to evaluate what can be gained by rewriting codes to exploit a given paradigm. Because of their complexity, many of the important scientific and engineering application codes will not be rewritten unless significant performance improvement is predicted. The evaluation data will also allow us to interpret the fairness of using a given benchmark code implemented using, for example, pure MPI-1 or OpenMP, when comparing results between platforms.

3.3 Application evaluation and benchmarking.

A number of Department of Energy (DOE) Office of Science application codes from the areas of global climate, fusion, materials, chemistry, and astrophysics have been identified as being important to the DOE evaluation of the X1. Here, we examine the performance of three codes, the Parallel Ocean Program (POP) [10], the coupled gyrokinetic-Maxwell equation solver GYRO [4], and Dynamical-Cluster-Approximation Quantum Monte Carlo (DCA-QMC) [11], drawn from global climate, fusion, and materials, respectively. The optimization and performance evaluation is not complete for any of these, but the current status of the work, and the amount of work required so far, provides some insight into the promise and practical considerations of using the Cray X1.

4 Evaluation Results

The data described here were collected primarily during late April and early May of 2003 on the 32 processor system available at the time. Unless stated otherwise, all experiments were run using version 2.1.06 of the

UNICOS/mp operating system, version 4.3.0.0 of the Cray Fortran and C compilers and math libraries, and version 2.1.1.0 of the MPI and SHMEM libraries. For comparison purposes, performance data are also presented for the following systems:

- Earth Simulator: 640 8-way vector SMP nodes and a 640x640 single-stage crossbar interconnect. Each processor has 8 64-bit floating point vector units running at 500 Mhz.
- HP/Compaq AlphaServer SC at Pittsburgh Supercomputing Center (PSC): 750 ES45 4-way SMP nodes and a Quadrics QsNet interconnect. Each node has two interconnect interfaces. The processors are the 1GHz Alpha 21264 (EV68).
- HP/Compaq AlphaServer SC at ORNL: 64 ES40 4-way SMP nodes and a Quadrics QsNet interconnect. Each node has one interconnect interface. The processors are the 667MHz Alpha 21264a (EV67).
- IBM p690 cluster at ORNL: 27 32-way p690 SMP nodes and an SP Switch2. Each node has 2 to 8 Corsair (Colony on PCI) interconnect interfaces. The processors are the 1.3 GHz POWER4.
- IBM SP at the National Energy Research Supercomputer Center (NERSC): 184 Nighthawk (NH) II 16-way SMP nodes and an SP Switch2. Each node has two interconnect interfaces. The processors are the 375MHz POWER3-II.
- IBM SP at ORNL: 176 Winterhawk (WH) II 4-way SMP nodes and an SP Switch. Each node has one interconnect interface. The processors are the 375MHz POWER3-II.
- NEC SX-6 at the Artic Region Supercomputing Center: 8-way SX-6 SMP node. Each processor has 8 64-bit floating point vector units running at 500 MHz.
- SGI Origin 3000 at Los Alamos National Laboratory: 512-way SMP node. Each processor is a 500 MHz MIPS R14000.

The Cray X1 at ORNL has been changing rapidly since its initial installation in March 2003. Since the initial data were collected April and May, the system software has been upgraded a number of times and the size of the system has doubled. Results are likely to change over time, as the system software and the application code optimizations continue to mature. In consequence, none of our evaluations are complete. However, data has been collected that illuminates a number of important performance issues, including:

- Range of serial performance. As in previous vector systems, the Cray X1 processor has a large performance range. Code that vectorizes and streams (uses multiple SSPs) well can achieve near peak performance. Code that runs on the scalar unit demonstrates poor performance.
- Memory performance. Can the memory hierarchy keep the vector units busy? Can processes on separate MSPs contend for memory bandwidth?
- SSP vs. MSP. What are the advantages or disadvantages to running in SSP mode (assigning separate processes to individual SSPs) compared to running in MSP mode?
- Page size. At run time, the user can specify whether to run using 64 KB, 1 MB, 4 MB, 16 MB, or 64 MB pages. Does this make a performance difference?
- Communication protocol. When using MPI, which of the many MPI commands should be used to implement point-to-point communication.
- Alternatives to MPI. What are the performance advantages of SHMEM and Co-Array Fortran?

Note that we have collected (and continue to collect) more data than can be described here. Current and complete data can be viewed on the Evaluation of Early Systems project web pages at
`http://www.csm.ornl.gov/evaluation`.

4.1 Standard Benchmarks

We are running a number of standard low-level and kernel benchmarks on the Cray X1 to identify performance characteristics of the computational units and of the memory subsystem. In this section we describe results from codes drawn from ParkBench 2.1 [9], EuroBen 3.9 [18], STREAMS [12], and HALO [19]. As standard benchmarks, we have not modified them in any way, except to link in vendor libraries when appropriate.

4.1.1 Computation Benchmarks

The Cray X1 is a traditional vector machine in the sense that if a code does not vectorize, then performance is poor, while if it does, then performance can be good. The evaluation issue is understanding the range of performance, and how much work is required to port and optimize a code on the architecture. The first three microbenchmarks measure single MSP performance for some standard, unmodified, computational kernels. As the first two call math library routines that have been optimzed to some degree by the vendors, they represent the promise of the architecture. The third indicates what performance (or lack thereof) can occur if the code does not vectorize. The last two microbenchmarks measure the performance of basic CPU and math intrinsic functions, respectively.

DGEMM. Figure 1 compares the performance of the vendor scientific libraries for a matrix multiply with DGEMM [7] on an X1 MSP with the performance on other IBM and HP systems. From these data it is

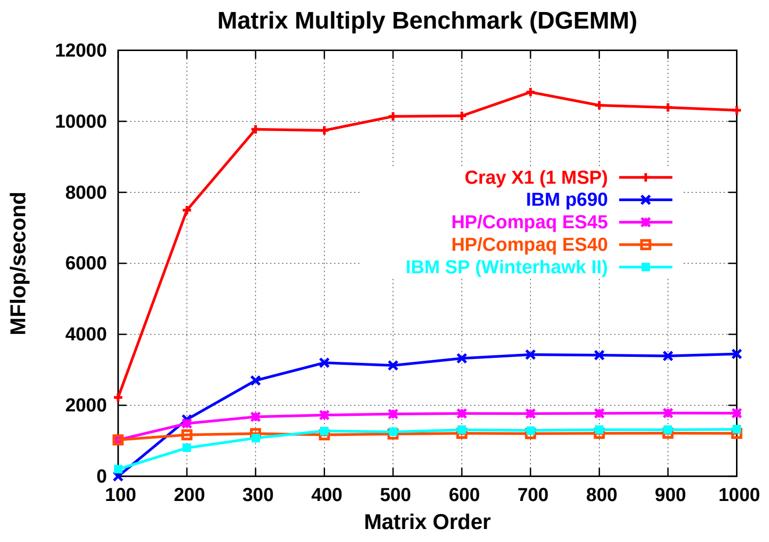

Figure 1: Single processor DGEMM performance

clear that it is possible to achieve near peak (> 80%) performance on computationally intensive kernels.

MOD2D. Figure 2 illustrates the single processor performance of a dense eigenvalue benchmark (EuroBen MOD2D [18]) using the vendor-supplied scientific libraries for the BLAS routines. On the X1, we used a single MSP. For this benchmark, we increased the problem size for the X1 to illustrate that its performance is still improving with larger matrices. Performance is good for large problem sizes relative to the IBM and HP systems, but the percentage of peak is much lower than for DGEMM for these problem sizes. Not all of the BLAS routines in the Cray scientific library (`libsci`) may yet have received the same attention as the matrix multiply routines, and MOD2D performance may improve over time.

MOD2E. Figure 3 describes the single processor performance for the EuroBen MOD2E benchmark [18], eigenvalue analysis of a sparse system implemented in Fortran. Note that the performance metric here is iterations per second (of the iterative algorithm), and that larger is better. The performance of MOD2E on

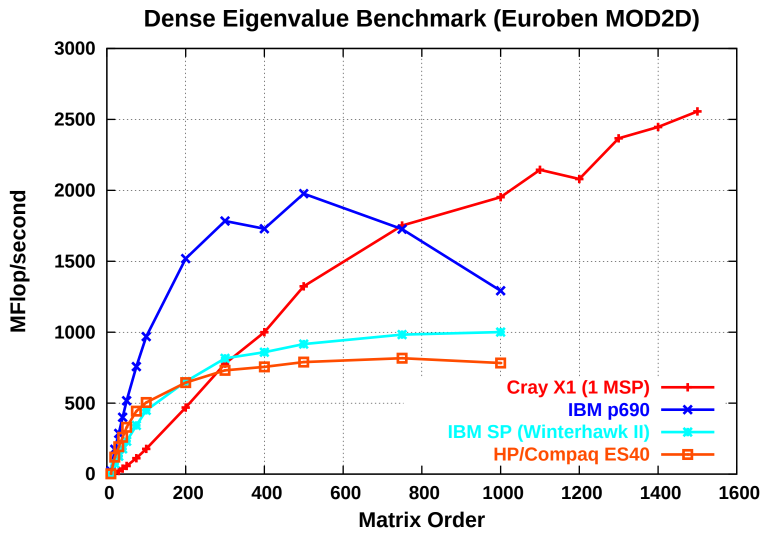

Figure 2: Single processor MOD2D performance

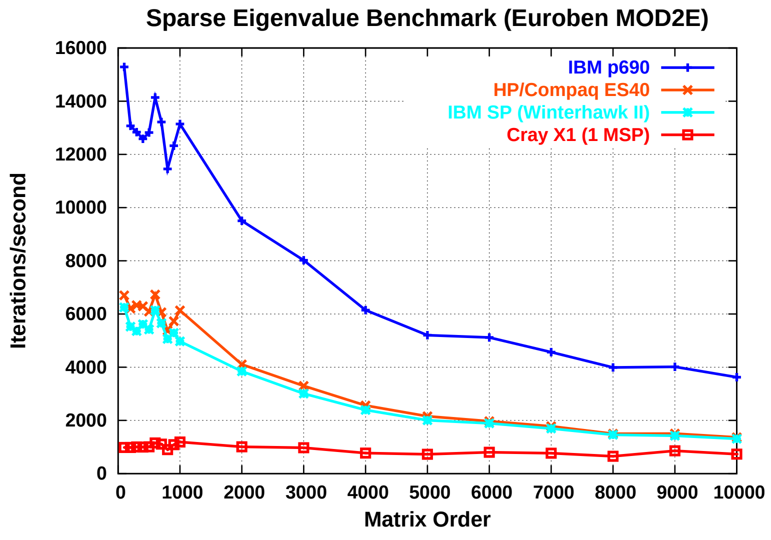

Figure 3: Single processor MOD2E performance.

the X1 is poor compared to the nonvector systems. While aggressive compiler optimizations were enabled, this benchmark was run unmodified, that is, none of the loops were modified nor were compiler directives inserted to enhance vectorization or streaming. Looking at the compiler output, the compiler was able to vectorize only a small number of the loops. Whether this inability to vectorize the code is due to the nature of the benchmark problem or simply the nature of the code is not yet clear. Note that on the Cray X1 the iterations per second metric decreases much slower than the matrix size (and computational complexity per iteration) increases. This implies that performance on the X1 is improving for the larger problem sizes, so some vectorization or streaming is being exploited. The lesson is clear however - it is important to exploit the vector units if good performance is to be achieved.

MOD1AC. Table 1 describes the single processor performance for the first few kernels of the EuroBen MOD1AC benchmark [18], measuring the performance of basic CPU functions. The last kernel (9th degree polynomial) is a rough estimate of peak FORTRAN performance in that it has a high reuse of operands. Timings were taken over loops of length 10000, and performance of the vector units on the Cray X1 is being measured in most of the tests. Both MSP and SSP results are provided for the Cray system.

	HP	IBM	IBM	Cray X1	
	ES40	Winterhawk II	p690	MSP	SSP
addition	285	186	942	1957	601
subtraction	288	166	968	1946	593
multiply	287	166	935	2041	608
division	55	64	90	608	162
dotproduct	609	655	2059	3459	
$\bar{X} = \bar{X} + a \cdot \bar{Y}$	526	497	1622	4134	1158
$\bar{Z} = \bar{X} + a \cdot \bar{Y}$	477	331	1938	3833	1205
$y = x1 \cdot x2 + x3 \cdot x4$	433	371	2215	3713	1218
1st order recursion	110	107	215	48	50
2nd order recursion	136	61	268	46	46
2nd difference	633	743	1780	4960	1822
9th deg. poly.	701	709	2729	10411	2676

Table 1: EuroBen MOD1AC: Performance of Basic Operations (Mflop/sec)

The 9th degree polynomial test demonstrates performance similar to that seen with DGEMM, indicating that the compiler is able to achieve near peak performance with simple Fortran loops. For the simple arithmetic functions (addition, subtraction, multiply), the MSP performance is only three times that of the SSP performance, instead of four times, indicating either that memory contention between the four SSPs or some sort of parallel overhead is limiting performance in the MSP case. As the SSP data for these tests indicates only approximately 20% of peak performance, SSP performance is being limited by main memory performance.

In contrast, for division, MSP performance is approximately four times that of SSP performance. This trend follows for the other functions. If there is sufficient computation per operand load, then the ratio of MSP to SSP performance is approximately a factor of four. Otherwise, the ratio is closer to three. (Note that for the recursion tests, the compiler was unable to use more than one SSP with the benchmark as written, and MSP performance is identical to SSP performance. The compiler was also unable to vectorize the loops, and performance is poor.)

For 6 of the 12 tests, the ratio of IBM p690 performance to Cray SSP performance is approximately the ratio of the peak MFlop rates for the two processors ($5.2/3.2 \approx 1.6$). In contrast, the Cray X1 SSP is a 1.8 times *faster* than the p690 (and the MSP is seven times faster) for the division test, indicating an advantage arising from hardware support for division on the X1. The Cray also has an additional performance advantage over the p690 for the last two tests, which are characterized by larger numbers of operands and greater operand resuse, possibly reflecting an advantage in register number or usage.

MOD1F. Table 2 shows the single processor performance for the EuroBen MOD1F benchmark [18], measuring the performance of a number of different math intrinsic functions. As with MOD1AC, the MOD1F benchmark times a loop of length 10000 for each function, thus measuring vector performance on the Cray X1. Note that a vector intrinsic library is also used in the IBM results (automatically linked in by using -O4 optimization).

	HP	IBM	IBM	Cray X1	
	ES40	Winterhawk II	p690	MSP	SSP
x^y	8.3	1.8	7.1	45.1	12.3
`sin`	13	34.8	64.1	94.4	23.5
`cos`	12.8	21.4	39.6	95.1	23.6
`sqrt`	45.7	52.1	93.9	648	153
`exp`	15.8	30.7	64.3	250	63.8
`log`	15.1	30.8	59.8	183	45.4
`tan`	9.9	18.9	35.7	80.8	19.2
`asin`	13.3	10.4	26.6	110	27.3
`sinh`	10.7	2.3	19.5	86.2	21.1

Table 2: EuroBen MOD1F: Performance of Math Intrinsics (Mcall/sec)

From the data in Table 2, the Cray X1 MSP is always faster than an IBM p690 processor when computing vector intrinsics, with the advantage ranging from factors of 1.5 to 7.6. The largest advantages are for the `sqrt`, for which there is vector hardware support on the X1, and for x^y, for which there is apparently no vector intrinsic on the IBM. As in the MOD1AC results, the MSP performance is approximately four times the SSP performance when the computation per operand load is large, which is true in all of these experiments.

Other EuroBen benchmarks. Data for additional EuroBen benchmarks, including MOD2A, MOD2B, MOD2F, and MOD2G, can be found on links off of `http://www.csm.ornl.gov/evaluation`.

4.1.2 Memory Benchmarks

STREAMS. John McCalpin's STREAM benchmark [12] measures sustainable main memory throughput for several simple operations. Table 3 describes the single processor performance achieved on a number of different architectures. It also includes the aggregate data rate for 4 SSPs running separate copies of the single processor stream test. Data for the ES45 was obtained from the STREAM database maintained by McCalpin.

	copy	scale	add	triad
X1 (1 MSP)	22111	21634	23658	23752
X1 (1 SSP)	6663	6681	8541	8375
X1 (4 SSPs)	24994	25078	31690	31253
HP ES40 (1 proc.)	1339	1265	1273	1383
HP ES45 (1 proc.)	1946	1941	1978	1978
IBM Winterhawk II (1 proc.)	523	561	581	583
IBM Nighthawk II (1 proc.)	486	494	601	601
IBM p690 (1 proc.)	1774	1860	2098	2119

Table 3: STREAM: Sustainable Memory Bandwidth (MByte/sec)

These data show that 4 SSPs achieve approximately four times the performance of 1 SSP in all 4 tests. In contrast, 1 MSP only achieves three times the performance of 1 SSP in the add and triad test, in agreement with the MOD1AC results. The STREAM results for a single X1 SSP are greater than a factor of three faster than on the other systems, and greater than a factor of ten for an MSP.

The single processor STREAM benchmark allows a single processor in an SMP node to exploit all available memory bandwidth. This is not always the relevant metric. Figure 4 shows the aggregate shared memory performance for the STREAM triad operation on a number of different architectures. In this case, all processors in an SMP node are running the benchmark concurrently. The SX-6 data is from the STREAM website. Note that concurrent STREAM on the Cray X1 is implemented with MPI, as OpenMP was not supported at the time of these experiments. X1 processors in this figure refer to MSPs. From these data,

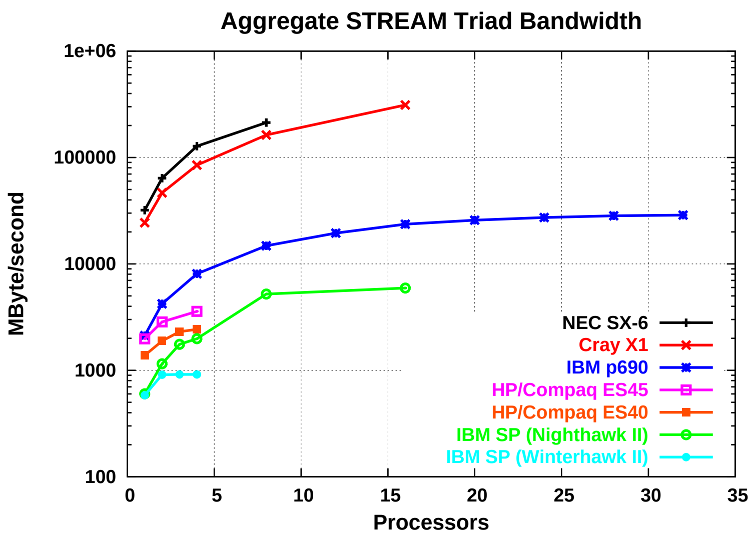

Figure 4: Aggregate STREAM triad bandwidth

the X1 has nearly the same memory bandwidth as the NEC SX-6, and significantly more than the nonvector systems. Note that results are limited to processors within a single SMP node for all systems except the X1. When using more than 4 MSP processors, more than one X1 node is required.

4.1.3 Communication Benchmarks

Latency and bandwidth are often slippery concepts, in that they can be difficult to associate with performance observed in application codes. Using standard point-to-point comunication tests produces the following estimates for the Cray X1:

	latency (microseconds)	bandwidth MB/sec
MPI		
intranode	8.2	13914
internode	8.6	11928

In contrast, SHMEM and Co-Array latency are 3.8 and 3.0 microseconds, respectively. Maximum bandwidth is similar to that of MPI.

Minimum observed time to execute MPI_Barrier, a latency sensitive operation, is described in Table 4. For the X1 data, processors refers to MSPs. From these data, barrier on the X1 is implemented more efficiently than the MPI point-to-point routines, achieving performance analogous to the Co-Array Fortran latency. What is not indicated here is the scalability of the barrier to larger processor counts. We will look at this more closely when the 256 processor system is installed.

Maximum observed MPI bandwidth in a simple bisection bandwidth test ($P/2$ processors sending 1 MByte messages to the other $P/2$ processors) is shown is Table 5. For the X1 data, processors refers to MSPs. The high performance memory and interconnect on the Cray X1 support excellent performance compared to the other systems. Scalability of the network and the system software with respect to the network again needs to examined with a larger system.

Processors	HP AlphaServer SC (ES40)	IBM SP (Winterhawk II)	IBM p690	Cray X1
2	11	10	3	3
4	16	20	5	3
8	18	157	7	5
16	21	230	9	6

Table 4: MPI_Barrier Performance (microseconds)

Processors	HP AlphaServer SC (ES40)	IBM p690	Cray X1
2	195	138	12412
4	388	276	15933
8	752	552	15872
16	1400	1040	32912

Table 5: MPI Bisection Bandwidth (MByte/sec)

The next benchmark attempts to measure communication performance in the context of a common application.

HALO. Wallcraft's HALO benchmark [19] simulates the nearest neighbor exchange of a 1-2 row/column "halo" from a 2-D array. This is a common operation when using domain decomposition to parallelize (say) a finite difference ocean model. There are no actual 2-D arrays used, but instead the copying of data from an array to a local buffer is simulated and this buffer is transferred between nodes.

In Figure 5, we compare the relative performance of a HALO exchange between 16 processors (MSPs) for SHMEM, MPI, and Co-Array Fortran. From these results, there is a clear advantage from using SHMEM

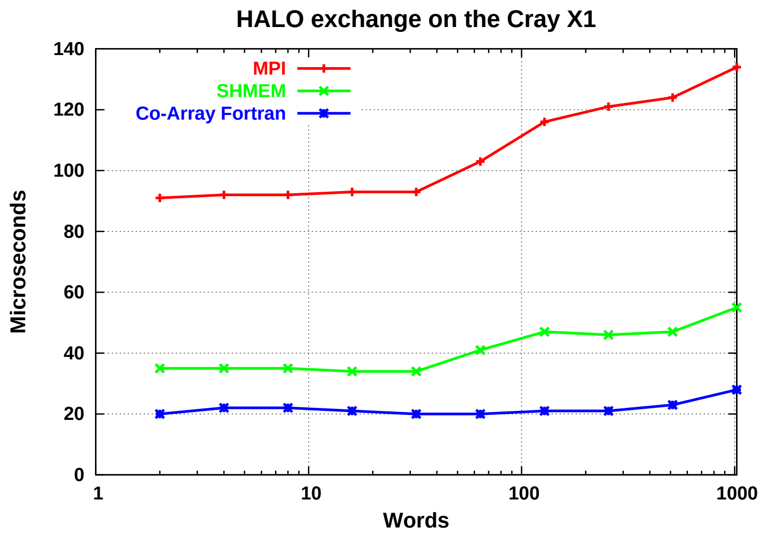

Figure 5: HALO: Comparison of Co-Array Fortran, MPI, and SHMEM performance.

or Co-Array Fortran in such latency-dominated kernels. The MPI data described in Fig. 6 are the best timings from a number of different MPI implementations. Figure 4.6 describes the performance of all of the candidate MPI implementations. The optimal implementation for all halo sizes uses a persistent MPI_ISEND/MPI_IRECV exchange protocol.

Figure 7 compares the HALO performance of the best MPI implementations on the IBM p690 and the

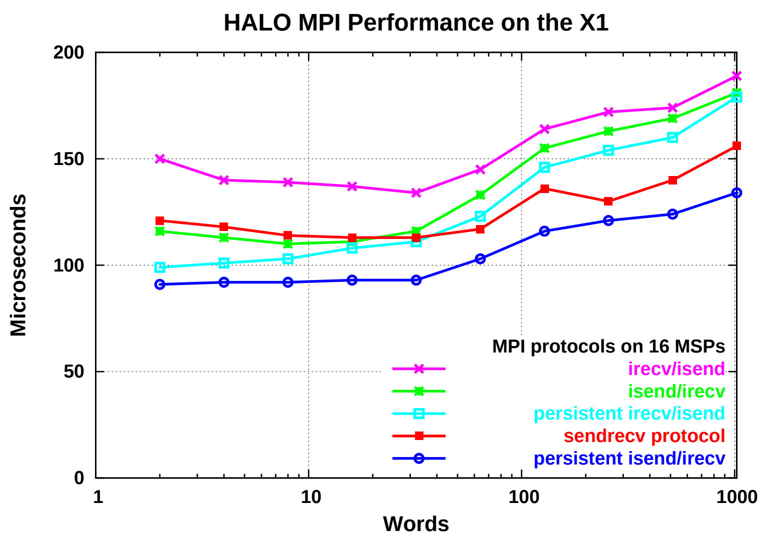

Figure 6: HALO: Performance comparison of different MPI protocols for exchange.

Cray X1. While the Cray X1 MPI performance is much better than that of the IBM p690 for large halos,

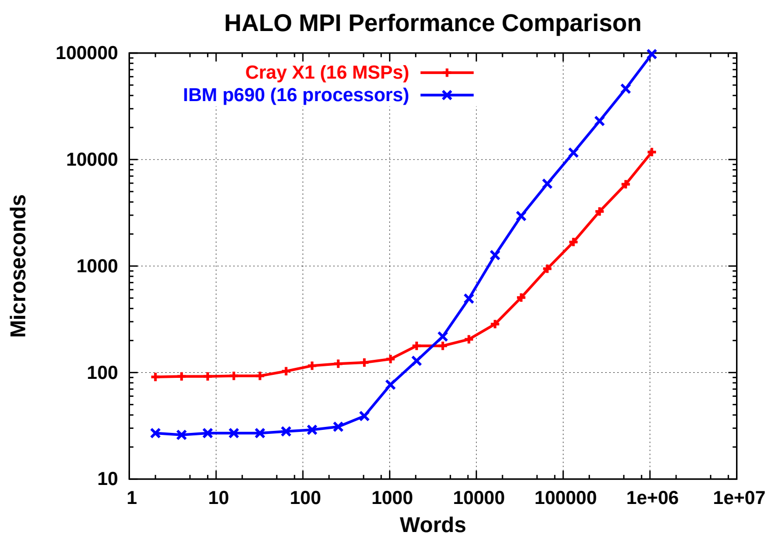

Figure 7: HALO: MPI performance intercomparison.

the opposite is true for small halos. Note, however, that the IBM p690 performance for small halos falls between that for SHMEM and Co-Array Fortran on the X1. It is interesting that the X1 times are much higher than the 8-byte message latency. The performance loss may instead be due to the buffer copying.

4.1.4 Parallel Benchmarks

MG from the NAS Parallel Benchmark [1] (and from the ParkBench benchmark suite [9]) is a kernel that uses multigrid to solve a linear system. Figure 8 shows the aggregate MFlop/sec for a fixed grid of size $256 \times 256 \times 256$. Results are given for both an MPI implementation and a Co-Array Fortran implementation due to Alan Wallcraft. For the two IBM systems, results are given for both MPI and OpenMP implementations. Note that the curves for MPI and Co-Array Fortran on the Cray X1 overlap, and are difficult to distinguish in this figure. The same is true for curves for MPI and OpenMP on the IBM Winterhawk II system.

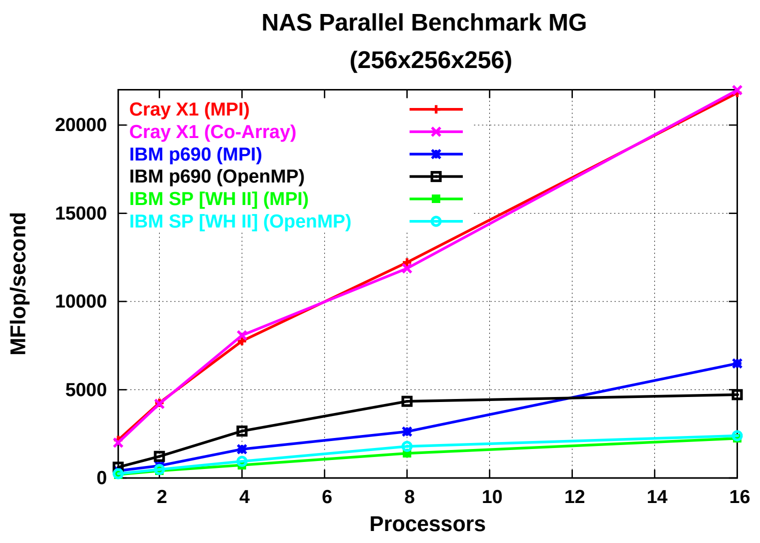

Figure 8: MG: Comparison of Co-Array Fortran, MPI, and OpenMP performance.

Due to the nature of the memory access patterns in the hierarchical multigrid algorithm, MG is typically sensitive to latency, both interprocess and memory. From these data, the X1 performance scales reasonably well, with the only significant problem at the transition from 4 to 8 processors, i.e. the transition from using only one SMP node to using multiple nodes. Co-Array Fortran is not a significant performance enhancer for this code, problem size, and number of processors. This lack of agreement with the HALO results is still being investigated, and may simply be due to the small number of processors applied to a problem of this size making latency less important. However, it does indicate the need for careful studies of alternative programming paradigms in the context in which they will be used before making decisions. While only achieving 10-16% of peak on the X1 in these experiments, this code was not modified to improve vectorization or streaming on the X1.

4.2 Custom Benchmarks

The following custom benchmarks were developed at ORNL, and have been used in a number of performance studies over the past ten years. As these were developed on-site, we are able to experiment with different implementations and to modify them to examine new performance issues. In the first kernel, PSTSWM, we describe our experiences in porting and optimization, look for memory contention, and compare SSP and MSP performance. The second kernel, COMMTEST, looks at MPI and SHMEM message-passing performance in more detail.

PSTSWM. The Parallel Spectral Transform Shallow Water Model (PSTSWM) [20, 21] represents an important computational kernel in spectral global atmospheric models. As 99% of the floating-point operations are multiply or add, it runs well on systems optimized for these operations. PSTSWM also exhibits little reuse of operands as it sweeps through the field arrays, making significant demands on the memory subsystem. By these characteristics, the code should run well on vector systems. However, PSTSWM is also a parallel code that supports a large number of different problem sizes and parallel algorithms, and all array dimensions and loop bounds are determined at runtime. This makes it difficult for the compiler to identify which loops to vectorize or stream.

When porting and optimizing PSTSWM on the X1, five routines were identified as needing modification:

- Forward complex FFT
- Inverse complex FFT

- Forward Legendre transform
- Inverse Legendre transform
- Nonlinear interactions in gridpoint space

All code modifications were local to the specified subroutines. In particular, no global data structures were modified. After modification, the new versions of the Legendre transforms and the nonlinear interactions subroutines still perform reasonably well on nonvector systems. The same is not true for the FFT routines, but on most nonvector systems we use optimized math library routines for these functions. We also examined fixing the vertical problem dimension at compile-time. In each of the above mentioned routines, two of the three problem dimensions (longitude, latitude, vertical) are available for vectorization and streaming, one of which is always the vertical. Specifying the vertical dimension at compile-time gives the compiler additional information for optimization. While this restricts the choice of domain decomposition and parallel algorithms in a parallel run, it is not an unreasonable restriction in many situations.

In a previous evaluation, we ported PSTSWM to the NEC SX-6. Optimizing on the SX-6 led us to modify the same routines. While similar "vector-friendly" versions of the FFT and nonlinear interactions routines were used on the SX-6 and the Cray X1, the code modifications for the Legendre transform routines were very different for the X1 and the SX-6. On the SX-6, long vectors are the goal of any restructuring, even if indirection arrays are then needed due to loop collapsing. Similar techniques are not productive on the X1. Vectors need not be as long on the X1 as long as there is a loop to stream over. Also, the scatter-gather hardware on the X-1 is not efficient for stride-zero or stride-one scatter-gathers, limiting the utility of collapsing loops. The X1 modifications were much closer to the original version of the code.

In the following experiments we examined a number of different alternatives. The code could be run in MSP mode (using both vectorization and streaming) or in SSP mode (running on a single SSP and only exploiting vectorization). As with most compilers, there are a number of different optimization options, as well as compiler directives to provide additional control. We added a small number of directives to the above mentioned routines to indicate to the compiler which loops could be vectorized or streamed. We also looked at two different optimization levels, the default (`def. opt.`) and an aggressive setting (`agg. opt.`):

```
-Oaggress,scalar3,vector3,stream3
```

Finally, at runtime you can specify the page size to use. At the time of these experiments, the choices were 64KB and 16MB. The default at the time of these experiments was 64KB pages when running within a single SMP node, and 16MB pages when using multiple nodes. (At the current time, July 2003, the default is 16MB pages in all cases, and 64KB pages can not be used when using more than one node. The choices have also expanded to include 1MB, 4MB, 64MB, 256MB, 1GB, and 4GB pages.) However, unless noted otherwise, all of the following experiments were run using 16MB pages.

Figures 9-13 display performance for a number of different problem sizes: T5L18, T10L18, T21L18, T42L18, T85L18, T170L18, and T340L18. The "T" number defines the horizontal (longitude, latitude) resolution, while the "L" number refers to the number of vertical levels. So the problem set varies in horizontal resolution, but all with 18 vertical levels. Each problem size requires approximately 4 times as much memory as the next smaller problem size. For example, T5 requires approximately 60KB of data space for each vertical level, while T85 requires approximately 18MB for each vertical level. The performance is described in terms of MFlop/sec The floating point operation count was calculated using an operation count model (which was validated on an IBM p690), and the computation rate should be viewed as a consistently weighted inverse time metric.

Figure 9 compares the serial performance when run on a single MSP. Note that the curves for aggressive and default optimization for the X1 version of the code overlap, and are difficult to distinguish in this figure. From these results, code modification was crucial for performance, and the SX-6 modifications were not appropriate on the X1. The second most important performance optimization was the use of 16MB pages. The default compiler optimizations were as good as the more aggressive optimizations for this code. Stipulating the vertical dimension at compile improved performance for the largest problem sizes. The advantage of compile-time over runtime specification is even greater when the number of vertical levels is smaller.

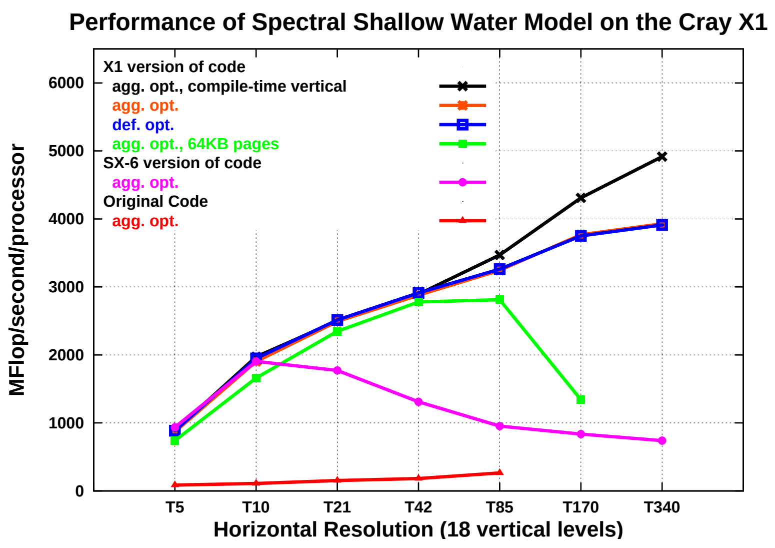

Figure 9: PSTSWM: comparing compiler options and code modification on the Cray X1

Figure 10 compares the performance when running on an MSP with running on an SSP, and when running simultaneously on all processors in an SMP node. For the MSP experiments, this denotes running 4 instances of the same problem simultaneously, while for the SSP experiments, 16 serial instances are running. We also include performance data for running 4 serial SSP instances simultaneously. In this situation, all 4 instances are scheduled on the same MSP, and this is a way of examining contention between SSPs within the same MSP. Note that the curves for 1 MSP and 4 MSPs overlap, and are difficult to distinguish in this figure. From these results, there is no contention when running multiple instances of PSTSWM in MSP

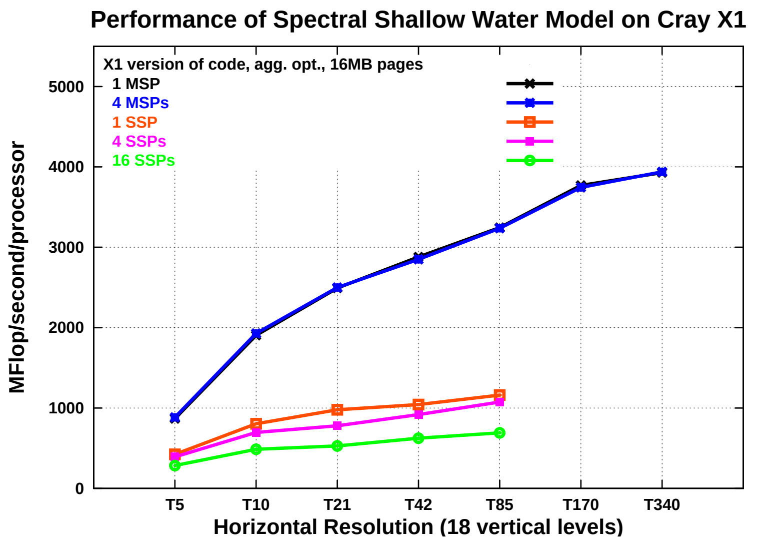

Figure 10: PSTSWM: comparing SSP and MSP processor performance on the Cray X1

mode on an X1 node. This is an excellent result, differing from all other platforms on which we have run the same experiment. Comparing performance between runs on single SSPs and MSPs is complicated. MSP performance can suffer from lack of independent work to exploit with streaming and from contention for memory bandwidth or cache conflicts between the 4 SSPs. In contrast, in the single SSP experiments the other SSPs are idle, and the active SSP has sole possession of all shared resources. In these experiments, MSP performance is twice as fast as SSP performance for the smallest problem size, growing to 2.8 times faster for the largest problem size. When looking at the simultaneous runs, the 16 simultaneous SSP instances, which do share resources, show contention. The performance when running 4 MSP instances is 3.1 to 4.7 times faster than when running 16 SSP instances. Thus, simply from a throughput metric, running simultaneous MSP jobs is more efficient than running simultaneous SSP jobs (for this code) for all but the smallest problem instances. The 4-instance SSP experiment has performance closer to that for the single SSP, but there is still some performance degradation. However, over half of the performance degradation in the 16-instance SSP experiment appears to be due to simple memory contention within the SMP node, not contention within the MSP.

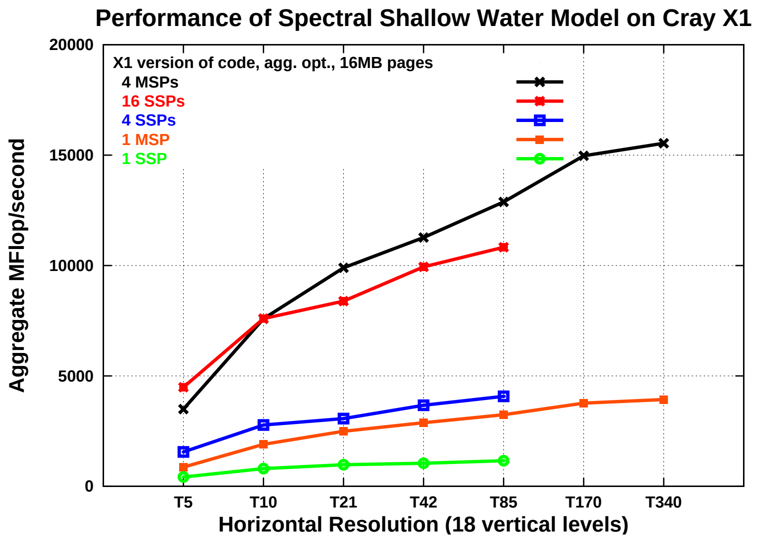

Figure 11: PSTSWM: comparing SSP and MSP aggregate performance on the Cray X1

Figure 11 is an alternative presentation of the same data, graphing the aggregate computational rate. Using this view, it is clear that running 4 serial SSP instances uses the MSP more effectively than running a single MSP instance (for this code, problem sizes, and experiment). Despite the memory intensive nature of the code, the lack of any need to exploit streaming results in higher efficiency, and higher throughput. The same is not true when comparing running on 4 MSPs with running on 16 SSPs, i.e. using the entire SMP node. The locality in the MSP implementation decreases demands on memory, avoiding contention that limits the SSP performance. The comparison of SSP and MSP modes will always be application and problem size dependent, and these experiments do not address important issues that arise when comparing parallel MSP and SSP codes. But these data do indicate some of the memory contention issues.

Figure 12 compares PSTSWM single processor performance across a number of platforms, using the best identified compiler optimizations, math libraries (when available), and versions of the code, excepting that all loop bounds are specified at runtime. Note that the curves for the two IBM SP systems overlap, and are difficult to distinguish in this figure. The p690 processor (5.2 GFlop/sec peak performance) performs reasonably well on this code for small problem sizes, with performance tailing off as problem size (and the memory traffic) increases. As the problem size increases, the vector performance increases, and even the performance on a single SSP (peak performance 3.2 GFlop/sec) exceeds that of the nonvector systems for problem sizes larger than T42L18. The SX-6 performance peaks at 2.6 GFlop/sec and begins decreasing for problem sizes larger than T42. This occurs because the Legendre transforms dominate the performance for the large problem sizes, and they are only achieving approximately 1.5 GFlop/sec on the SX-6. The FFT routines, which determine performance for the small problem sizes, achieve over 4 Gflops/sec on the SX-6. Just the opposite occurs on the X1, where the Fortran implementations of the FFT routines are only achieving 2 GFlop/sec while the Legendre transform performance continues to grow with horizontal resolution, reaching 4.5 GFlop/sec. for the largest problem size described here.

Figure 13 compares PSTSWM SMP node performance across a number of platforms. As before, the same

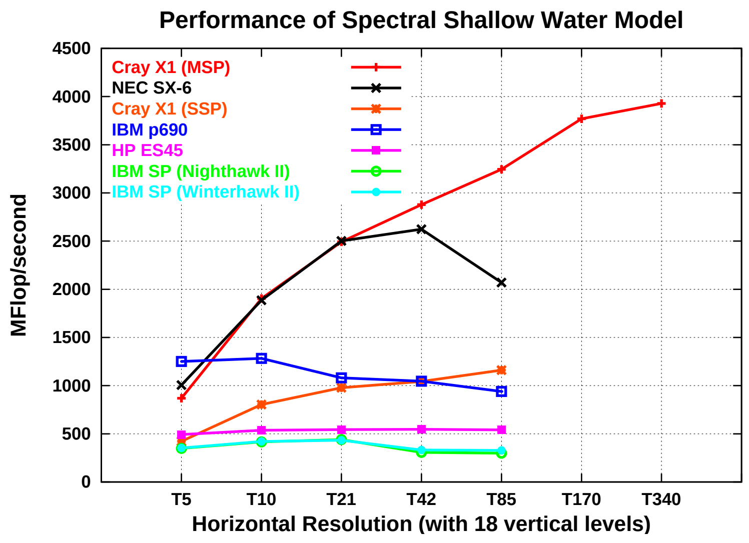

Figure 12: PSTSWM: single processor cross-platform comparison

problem is solved on all processors in the SMP node simultaneously. Performance degradation compared with the single processor performance indicates contention for shared resources and is one metric of throughput capability. The curves for the two IBM SP systems continue to overlap, making them difficult to distinguish in this figure. As noted earlier, the lack of contention in the MSP runs is unique. All of the other platforms

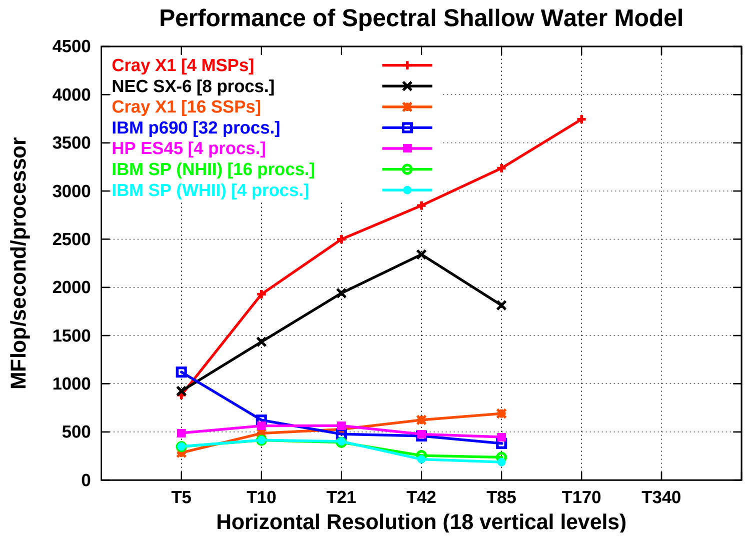

Figure 13: PSTSWM: single SMP node cross-platform comparison

(and the SSP version of PSTSWM on the X1) demonstrate some performance degradation when running the larger problem sizes.

COMMTEST. COMMTEST is a suite of codes that measure the performance of MPI interprocessor communication. COMMTEST differs somewhat from other MPI benchmark suites in its focus on determining the performance impact of communication protocol and packet size in the context of "common usage". However, the performance we report here should be similar to that measured using other interprocessor communication benchmarks. SHMEM performance is also measured in the context of implementing a simple version of the SWAP and SENDRECV operators, where message selection is by source only.

Using COMMTEST, we measured the peak bidirectional SWAP (ping-ping) bandwidth for five experiments:

1. MSP 0 swaps data with MSP 1 (within the same node)
2. MSP 0 swaps data with MSP 4 (between two neighboring nodes)
3. MSP 0 swaps data with MSP 8 (between two more distant nodes)
4. MSP i swaps data with MSP $i+1$ for $i = 0, 2$, i.e. 2 pairs of MSPs (0-1, 2-3) in the same SMP node swap data simultaneously.
5. MSP i swaps data with MSP $i + P/2$ for $i = 0, \ldots, P/2 - 1$ for $P = 16$, i.e. 8 pairs of MSPs (0-8, 1-9, ..., 7-16) across 4 SMP nodes swap data simultaneously.

We measured SWAP bandwidth because the swap operator is more commonly used in performance critical interprocessor communication in our application codes than is the unidirectional send (receive).

The COMMTEST MPI results are displayed in Figures 14-15, using log-log and log-linear plots, respectively. For these experiments, we ran using 16MB pages. We measured performance using approximately 40 different implementations of SWAP using MPI point-to-point commands, including block, nonblocking, and synchronous commands, but not persistent commands. Only the best performance is reported in the graphs. For most of the experiments, the implementation using MPI_SENDRECV was among the best performers.

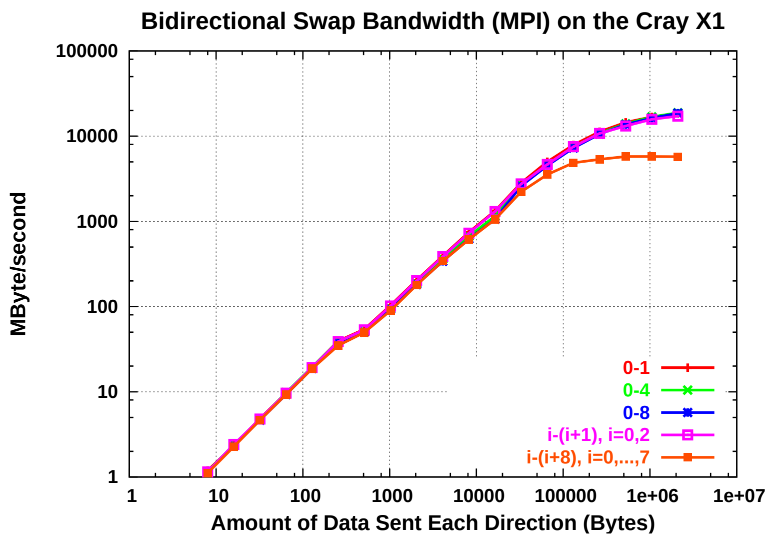

Figure 14: Log-log plot of MPI bandwidth

Performance for small messages (length less than 4 KBytes) is similar across all experiments. For larger messages, performance differs significantly only for the 16 processor simultaneous exchange experiment, where the internode bandwidth must be shared by all pairs. This saturation test reaches a maximum bidirectional bandwidth of 6 GByte/sec for exchanging messages of size greater than 256 KB. The other experiments have not achieved maximum bandwidth by 2MB messages. These same experiments have also been run with explicitly invalidating the cache before each timing, and with multiple iterations of each swap. Neither variant affects the achieved bandwidth significantly.

Figures 16-18 graph MPI and SHMEM SWAP bidirectional bandwidth and MPI and SHMEM ECHO (ping-pong) unidirectional bandwidth for experiment (1), using log-log and log-linear plots. From these data, SHMEM SWAP achieves the highest bandwidth, and SWAP bidirectional bandwidth is close to twice the ECHO unidirectional bandwidth for all but the very largest message sizes. So, nearly full bidirectional bandwidth is achieveable. SHMEM SWAP is only 5% better than MPI SWAP for the largest message sizes,

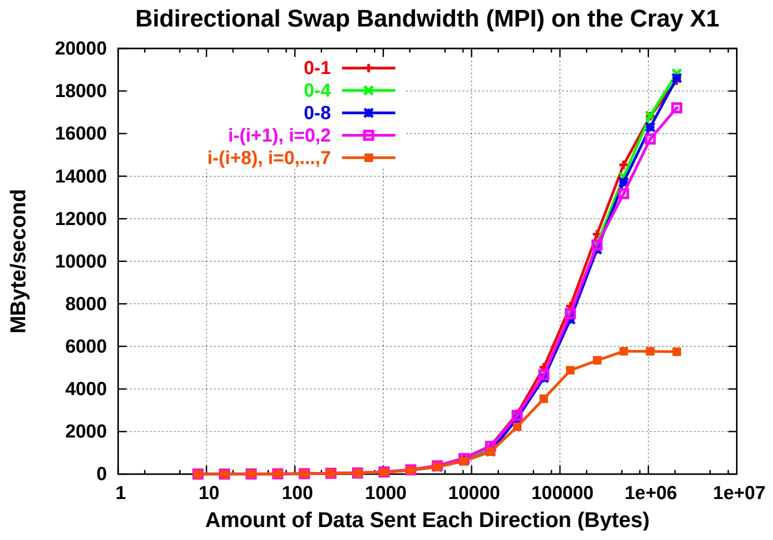

Figure 15: Log-linear plot of MPI bandwidth

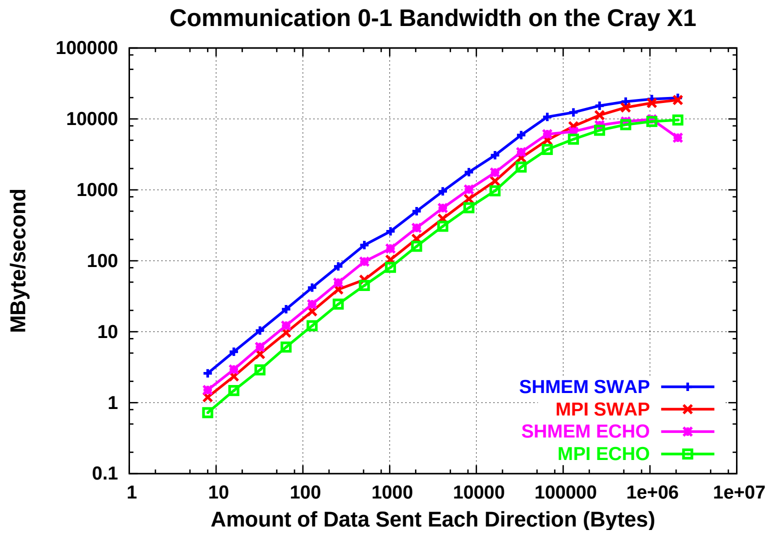

Figure 16: Log-log intranode bandwidth for 0-1 experiment.

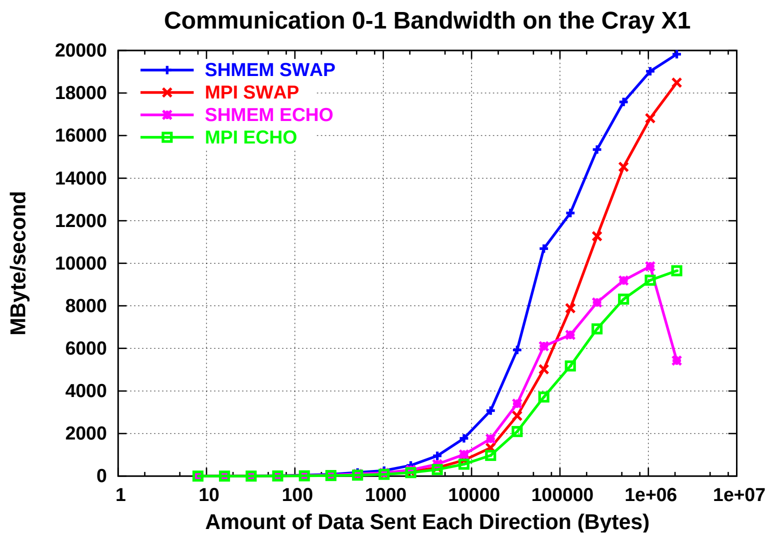

Figure 17: Log-linear intranode bandwidth for 0-1 experiment

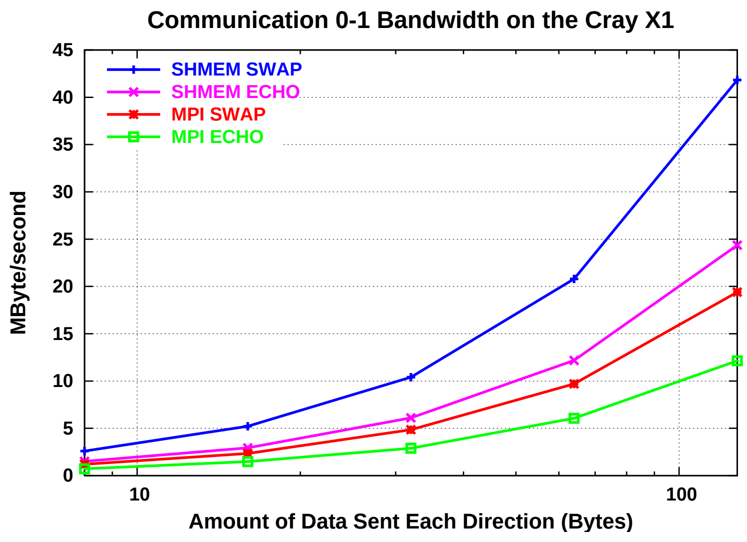

Figure 18: Log-linear intranode bandwidth for 0-1 experiment

but is over twice as fast for small and medium message sizes. Note that the SHMEM echo test suffers a performance problem for the largest message size. This is consistent across the other experiments and is repeatable, with this version of the system software. It did not occur in experiments run with the previous version of system software.

Figures 19-21 graph MPI and SHMEM SWAP bidirectional bandwidth and MPI and SHMEM ECHO (ping-pong) unidirectional bandwidth for experiment (5), again using both log-log and log-linear plots. These data are qualitatively very similar to those from the previous experiment. SHMEM SWAP is

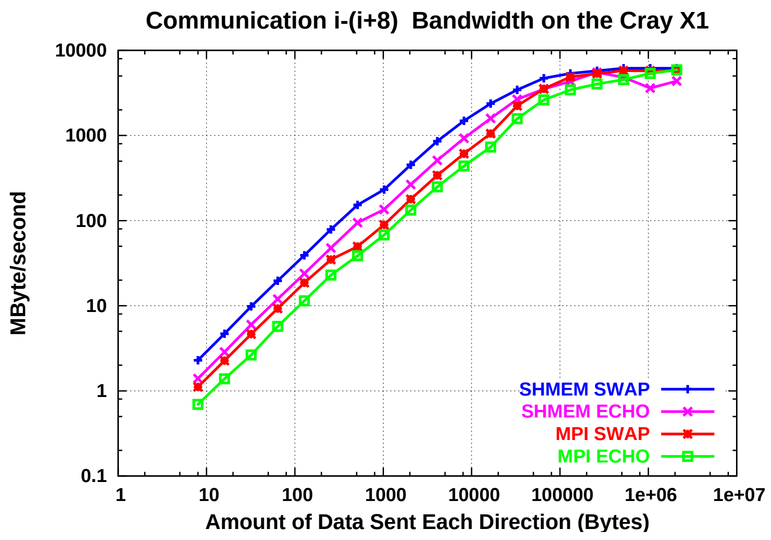

Figure 19: Log-log internode bandwidth for 8 simultaneous pairs experiment.

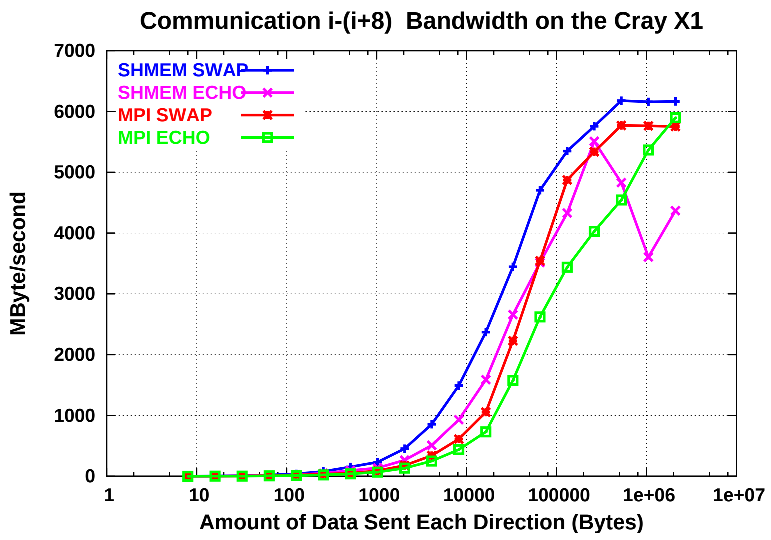

Figure 20: Log-linear internode bandwidth for 8 simultaneous pairs experiment.

again the best performer, and SHMEM ECHO again has a performance problem for large messages. For small messages, the SWAP bandwidth is nearly twice that of the ECHO bandwidth. However, once SWAP bandwidth saturates the network, ECHO bandwidth continues to grow until the two are nearly equal. For the most part, MPI and SHMEM message-passing performance for the optimal protocols are smooth functions of message size.

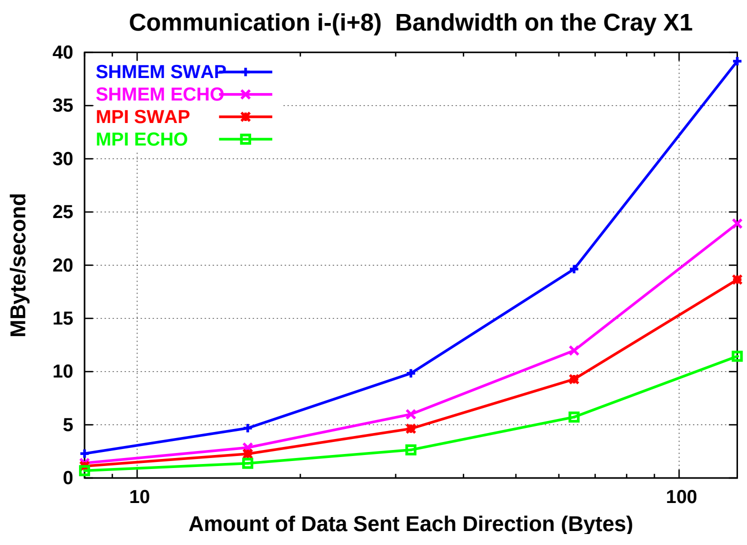

Figure 21: Log-linear internode bandwidth for 8 simultaneous pairs experiment.

4.3 Application Performance

Application benchmarking is the most important aspect of the evaluation. It is also the most difficult. Fairness requires that some effort be made to optimize the applications on a new system. However, the degree to which they can be modified is much less than for the kernel benchmarks. In this section we summarize our progress in porting and optimizing three application codes on the Cray X1.

4.3.1 POP.

The Parallel Ocean Program (POP) is the ocean component of of the Community Climate System Model (CCSM) [2], the primary model for global climate simulation in the U.S. POP is developed and maintained at Los Alamos National Laboratory (LANL). The code is based on a finite-difference formulation of the three-dimensional flow equations on a shifted polar grid.

The performance of POP is primarily determined by the performance of two processes. The ”baroclinic” process is three dimensional with limited nearest-neighbor communication and should scale well. The ”barotropic” process, however, involves a two-dimensional, implicit solver whose performance is very sensitive to network latency. Targeted communication optimizations to minimize latency, such as replacing MPI-1 calls with MPI-2 one-sided calls or SHMEM calls or reimplenting latency sensitive operators in Co-Array Fortran, are expected to be important if good performance is to be achieved.

POP has previously been ported to the Earth Simulator, where restructuring improved the vectorization of the baroclinic process. Cray also has a port of POP, one that uses Co-Array Fortran to minimize the interprocessor communication overhead of the barotropic process. These two codes have similar performance, but represent orthogonal optimizations. A merger is currently being investigated, and preliminary results for the merged code are described here.

Figure 22 compares performance when using 64KB and 64MB pages, as well as performance data for the unmodified Earth Simulator version and for the original version of POP. The metric here is the number of years of simulation that can be computed in one day of compute time, i.e., a weighting of inverse time. The problem is a benchmark specified by the authors of the code that uses a grid with one degree horizontal resolution, referred to as the “by one”, or `x1`, benchmark. The code was compiled with the default optimization level, which, as with PSTSWM, was as effective as more aggressive compiler optimizations.

These data indicate that page size does not impact performance for POP. (The curves for using 16MB and 64KB pages overlap, and are difficult to distinguish in this figure.) Given the earlier PSTSWM results, it is safest to use 16MB pages sizes. The original version of the code again shows poor performance. The Earth Simulator version of the code performs reasonably well. The primary difference between the merged

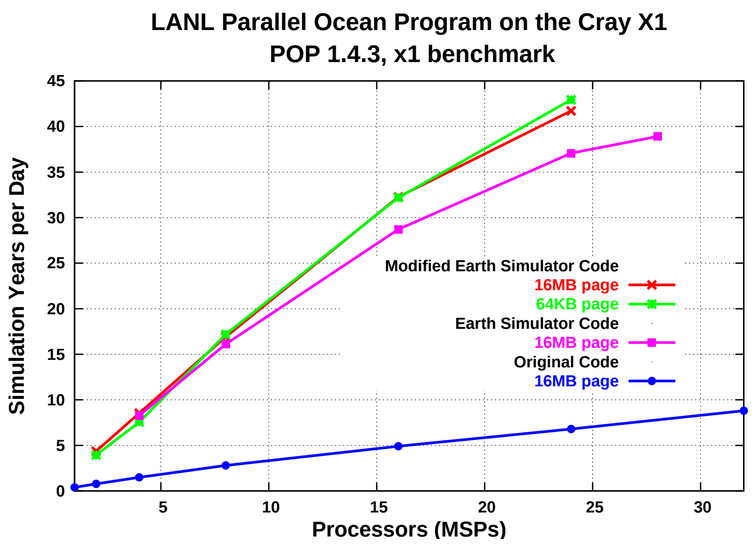

Figure 22: POP: X1 performance results

code and the Earth Simulator version is the use of Co-Array Fortran to implement the conjugate gradient algorithm, an algorithm that is very sensitive to message-passing latency.

Figures 23-27 compare the performance of POP on the X1 with performance on other systems. The first two figures again plot the years per day metric. Figure 23 is a linear-linear plot of performance when using up to 64 processors. Figure 23 is a log-linear plot of performance when using up to 512 processors. Figure 25 is a plot of relative efficiency for a subset of the systems. Efficiency is is defined in terms of performance when using 4 processors:

$$(4 \cdot T_4)/(P \cdot T_P)$$

where T_P is the runtime when using P processors. Figures 26-27 are log-log plots of the seconds per simulation day spent in the baroclinic and barotropic processes, respectively.

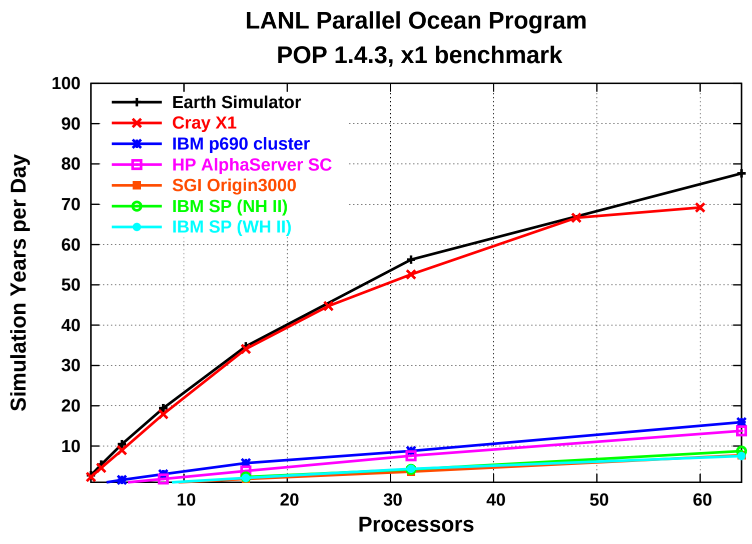

Figure 23: POP: cross-platform performance comparison

From these data, POP on the X1 performs significantly better than it does on the nonvector systems, but performance on the X1 still lags slightly behind performance on the Earth Simulator. The time spent in

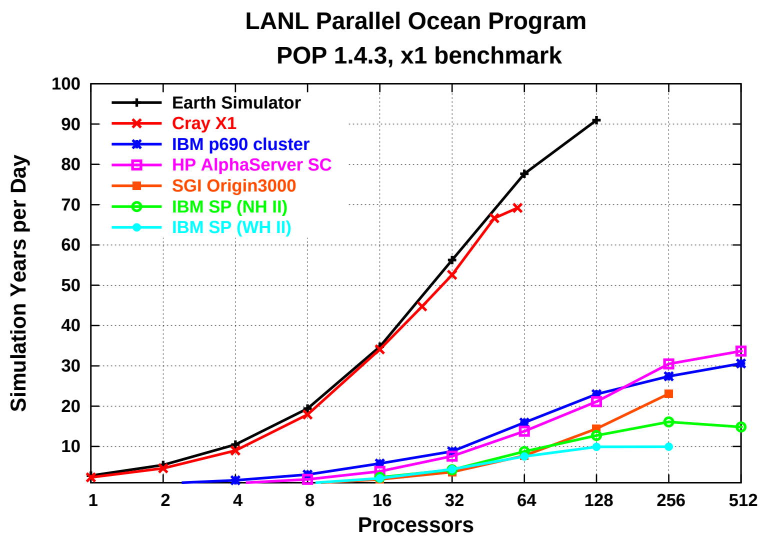

Figure 24: POP: cross-platform performance comparison (log-linear plot)

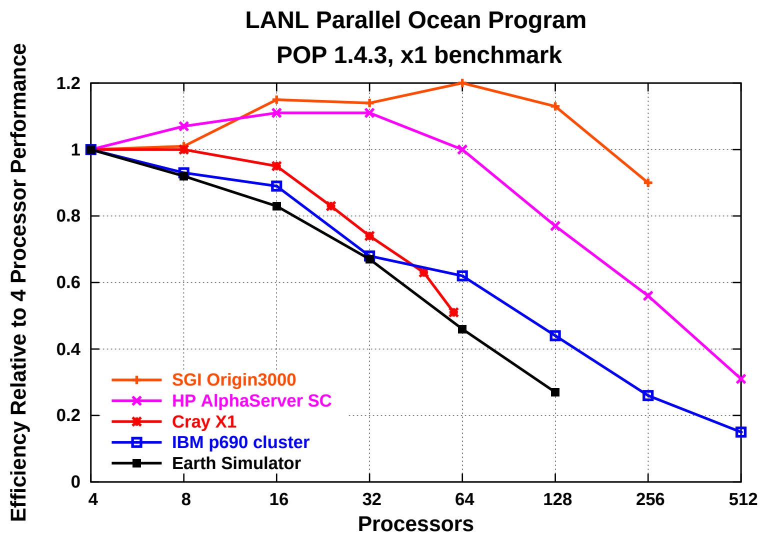

Figure 25: POP: cross-platform efficiency comparison

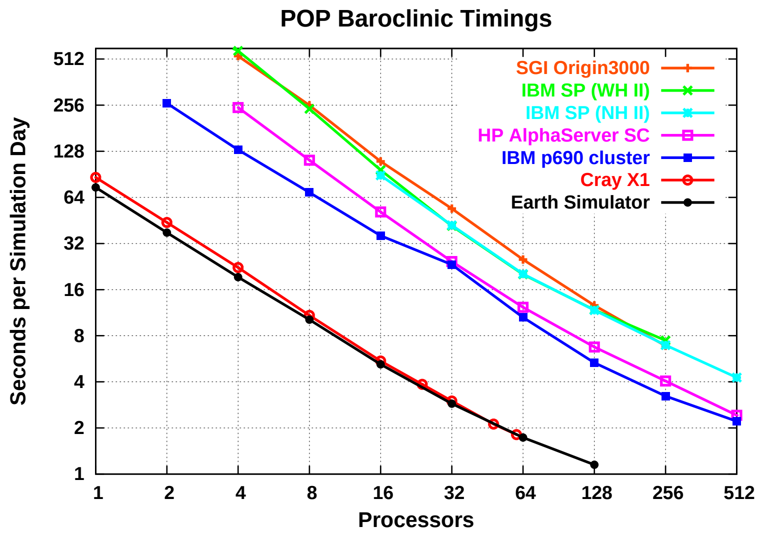

Figure 26: POP: baroclinic performance

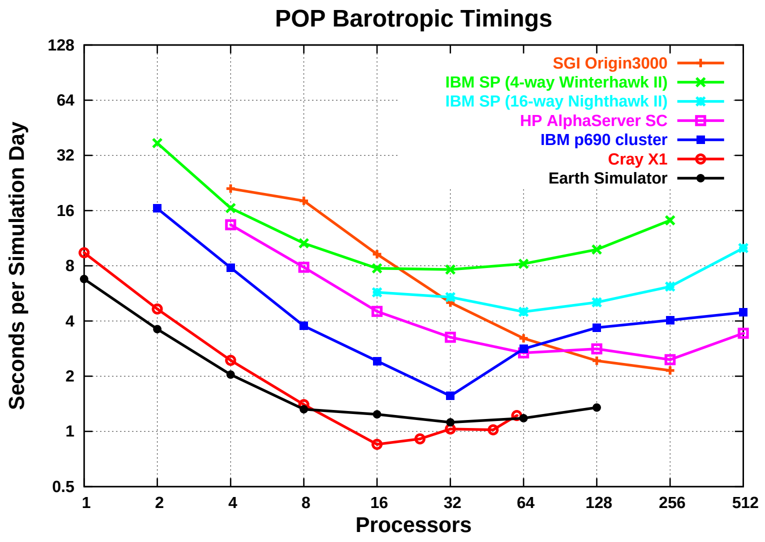

Figure 27: POP: barotropic performance

the baroclinic process is nearly identical on the Earth Simulator and the X1. Performance is better in the barotropic process on the X1 than on the Earth Simulator for moderate numbers of processors, primarily due to the use of Co-Array Fortran in the conjugate gradient solver, but the trend for larger processor counts is inconclusive. As much of the X1 version of POP is still identical to the Earth Simulator version, there is every indication that X1 performance can be improved. This is particularly true in as much as there are routines, which were not part of the baroclinic or barotropic processes, that were not important to vectorize in the Earth Simulator port that appear to be important to vectorize on the X1.

It is interesting to note in Fig. 25 the superlinear speed-up (efficiency greater than one) on the HP and SGI systems. This is undoubtedly due to improved cache locality as the number of processors increase and the per process problem granularity decreases. This effect is not seen on the vector systems or on the IBM p690 cluster, probably due to the smallness of the L2 caches on the X1 and p690 (and the lack of a cache on the Earth Simulator processor).

POP is a community code, one that is expected to run well on many different platforms. One of the challenges of the vector machines is to develop maintainable code that runs well on both vector and nonvector systems. This is an ongoing effort. The Earth Simulator version of the code uses the traditional “ifdef” approach, with separate code for vector and nonvector systems. While the amount of alternative code is relatively small, the code authors have not yet accepted that this is an acceptable approach. The issue will become even more complicated if optimizing for the Cray X1 requires generating even more alternative code. The Co-Array Fortran optimizations developed on the Cray X1 are even less performance portable. However, they are limited to the conjugate gradient solver, and are easily encapsulated in the same way as vendor-specific math library routines are.

4.3.2 GYRO.

”The most aggressively studied and developed concept for power production by fusion reactions is the tokamak” [4]. Advances in understanding tokamak plasma behavior are continuously being realized, but uncertainties remain in predicting confinement properties and performance of larger reactor-scale devices. The coupled gyrokinetic-Maxwell (GKM) equations provide a foundation for the first-principles calculation of anomalous tokamak transport.

GYRO [4] is a code for the numerical simulation of tokamak microturbulence, solving time-dependent, nonlinear gyrokinetic-Maxwell equations with gyrokinetic ions and electrons capable of treating finite electromagnetic microturbulence. GYRO uses a five-dimensional grid and propagates the system forward in time using a fourth-order, explicit, Eulerian algorithm. This application has shown good scalability on large microprocessor-based MPPs, and similar scalability is expected on the X1.

The port of GYRO to the X1 started with version 2.0.2, and was fairly straightforward in that no source code changes required for the code to work. All that was needed was to use the `-sreal64 -dp` compiler options to promote real variables to double precision, as is done on IBM systems, for example.

Performance optimization and evaluation proceeded on version 2.0.10. This involved a cycle of

1. build the code;
2. create an instrumented version;
3. run the code;
4. generate a profile report;
5. optimizing top routines in the profile, using hints from compiler optimization reports;
6. repeat.

The optimizations introduced in version 2.0.10 were then integrated into the 2.1.2 version. Performance evaluation is still ongoing with the current version, but, as with POP,

- aggressive compiler optimization did not increase code performance;
- page size is not a significant performance factor, and the 16MB page size default is a god choice;

- the performance of the unmodified code was poor.

The current performance was reached by modifying only 16 routines out of 140. In many cases the only modifications were the addition of directives to the source code. In a few routines, some minor source code changes were implemented:

- Interchange loops
- Add and/or remove temporary variables
- Promote variables to higher rank
- Rearrange local matrix dimensions in 2 routines

In five instances, loops were ”pushed” down into subroutine calls. This is a more major change, but it is expected to improve performance on all architectures.

A simple bisection search was also replaced with a vector-friendly algorithm in one routine. The function evaluation included the use of `sin` and `cos` which when done in scalar is extremely slow on the X1. A numerical integration routine was modified to use vectors instead of scalars, thereby supporting vectorization. A tridiagonal solve for particle collisions was inlined and simplified, rather than using LAPACK, as in the original version, and suffer the high subroutine overhead and unnecessary pivoting.

The current performance is best described by comparing timings between a couple architectures. Table 6 describes performance for the benchmark problem DIII-D simulation, which simulates the experimental levels of turbulent radial transport. Note that the original code was used for the IBM data, while the code optimized for MSP mode on the X1 were used for the Cray data.

System	Processors Used	Seconds per Timestep
IBM SP (Nighthawk II)	256	2.75
IBM p690	32	13.5
Cray X1	16 MSPs	4.8
Cray X1	32 MSPs	2.74
Cray X1	48 MSPs	2.05
Cray X1	128 SSPs	3.04

Table 6: GYRO: cross-platform performance comparison

From Table 6, the 32 MSP run of the X1 has performance similar to that of the 256 processor run on the IBM SP, and is 5 times faster than the 32 processor run on the IBM p690. Speed-up is reasonably good as well, achieving 78% relative efficiency when increasing the number of MSPs from 16 to 48. While not shown here, the optimized code is approximately 10 times faster than the original version on the X1.

The SSP result can be interpreted as indicating higher MPI communication costs and optimizations that have been solely geared toward the MSP (e.g. streaming). There may be code fragments that can be optimized for SSP mode to obtain better results.

Profiles from the current optimized version on the X1 indicate that the most time consuming routines that may be amenable to further optimization are

- the tridiagonal solve used in particle collisions (performed for thousands of times for each timestep),
- SSUB and TRANSPOSE routines: basically calls to MPI_AlltoAll

In particular, MPI communication is a primary performance bottleneck at this time. For the 48 MSP test on the X1, the top two routines in the profile were MPI calls. Up to now, no tuning effort has been made on the communication in GYRO, but this will be addressed in the near future. Similarly, investigation into the best mode to run GYRO in (SSP or MSP) are ongoing, concentrating on the communication differences and load balancing of SSPs in MSP mode.

In summary, it has been shown that GYRO can benefit from running on the X1. This benefit was the product of a nontrivial, but not unreasonable, optimization effort. (In total, the optimization effort has taken approximately 100 hours over 3 months, not including time spent debugging the code.) While performance optimization is continuing, we believe that a version of GYRO can be implemented that works well on both microprocessor based MPPs and vector MPPs like the X1. At this time, this can be done with the same source for both processor types, except for three routines.

4.3.3 DCA-QMC.

Dynamical-Cluster-Approximation Quantum Monte Carlo (DCA-QMC) [11] is used to simulate high-temperature cuprate superconductors. For large problems, runtime is dominated by the BLAS routines DGER, rank-1 update to a double-precision general matrix, and CGEMM, matrix-matrix multiplication for single-precision complex values.

The problem size is determined by `NC`, the cluster size, and `NL`, the number of time slices. Published results exist for `NC` of 4, and results here are for `NC` of 8. To confirm their theoretical results, scientists want to increase `NC` up to 64. For a given `NC`, scientists perform a series of runs with increasing numbers of Monte-Carlo measurements and increasing values of `NL`, typically from 10 to 72. The size of the matrices used in the BLAS routines comes from the product of `NC` and `NL`, $\texttt{NT} = \texttt{NC} \cdot \texttt{NL}$; the matrices are `NT` by `NT`. Also, DGER is called `NT` times for each iteration. Thus, the computational complexity for each process grows as $\texttt{NT}^3$.

Because of the Monte-Carlo nature of DCA-QMC, much of the computation for each parallel process is independent of the other processes. Scalability is limited by a number of factors, however. A global initialization phase can be distributed across just `NC` processes, and each process has a fixed number of "warm-up" iterations. Also, whereas some Monte-Carlo codes perform a large number of short iterations, DCA-QMC has iterations that are computationally expensive and performs only hundreds of them for each run. The code tested here uses MPI between processes, and each process is singly threaded. Because of the scalability limitations, this version favors a small number of powerful processors.

The Cray X1 is thus well suited for DCA-QMC. The processors are powerful, and the memory bandwidth provides advantages for DGER, a BLAS2 operation. The X1 has less of an advantage for BLAS3 operations, but the vector functional units of the X1 can perform single-precision floating-point operations at twice the rate of double-precision operations. Thus, the X1 has an advantage for CGEMM as well.

The computationally intensive regions of code outside of the BLAS routines vectorized on the Cray X1 without modification or with the addition of "concurrent" compiler directives, which appear as comments to other compilers. The most challenging aspect of adding these directives was determining which loops within a nest are indeed concurrent, for the nests use index arrays that contain some dependences. In a typical nest of loops that require directives, the nesting is four deep, and two loops are safely concurrent. The compiler is then able to vectorize at one level and stream at another.

We compare the performance of the X1 and the IBM p690 on a series of production runs of DCA-QMC. These runs, which must be run in order, have `NC` of 8 and increasing values of `NL`. Figure 28 shows runtimes for 8 and 32 processors of p690 and 8 and 32 MSPs of X1. The lines connecting the points are for clarity and do not imply a direct relation between points. The different runs have different input parameters as dictated by the science and represent a monotonic but not smooth increase in computational complexity. The runs are indicated by their `NL` values.

The figure shows that the X1 has little advantage over the p690 for small problems, but it scales much better to larger problems. Figures 29 and 30 show the contribution to the runtime by DGER and CGEMM for 8 processors/MSPs of the p690 and X1, respectively. As expected, DGER dominates, especially for the p690. Runs of increasing `NC` for fixed `NL`, not presented here, show equivalent behavior.

Adding OpenMP threading to DCA-QMC and linking threaded BLAS routines will allow the single-process work to be distributed across multiple processors, up to 4 MSPs on the X1 and 32 processors on the p690. This threading may help the p690 in particular, since it can reduce the effective memory load per node. It may be necessary to exploit threading on both systems as the computational work grows with increasing values of NC. Results shown here are for NC=8, while the scientific goal is NC=64, which represents an increase of $8^3 = 512$ times in computational complexity. The high performance achievable on the Cray X1, as demonstrated for NC=8, increases the feasiblity of this goal.

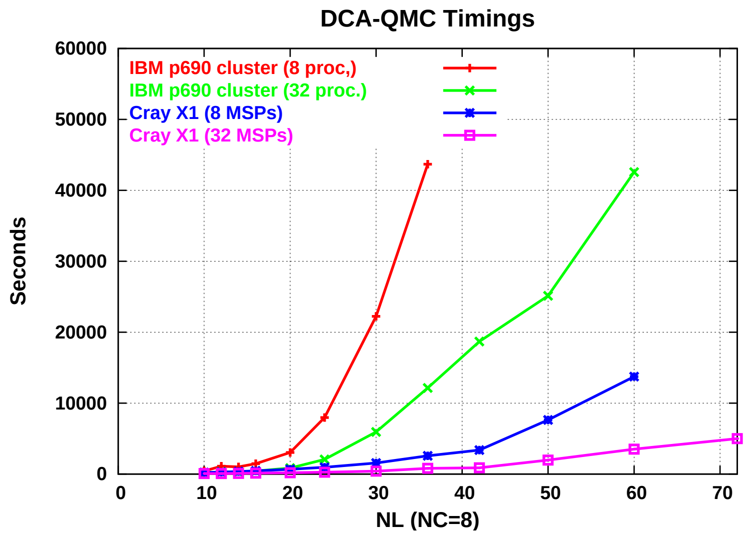

Figure 28: DCA-QMC: cross-platform performance comparison

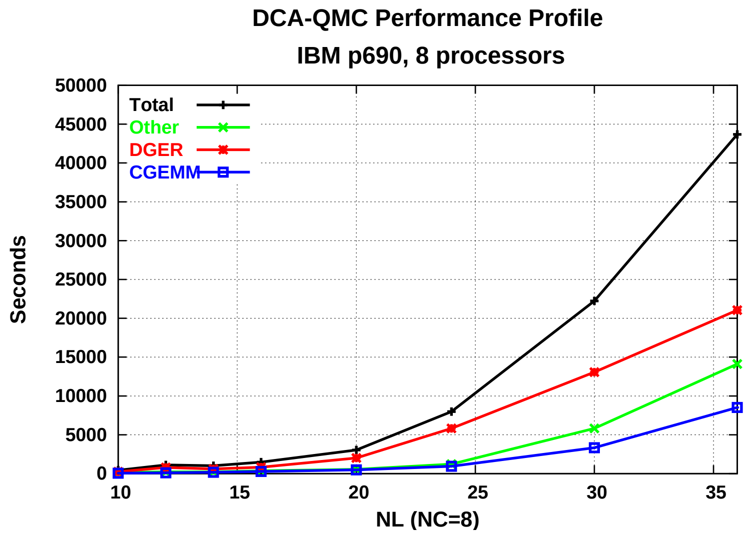

Figure 29: DCA-QMC: performance profile on the IBM p690

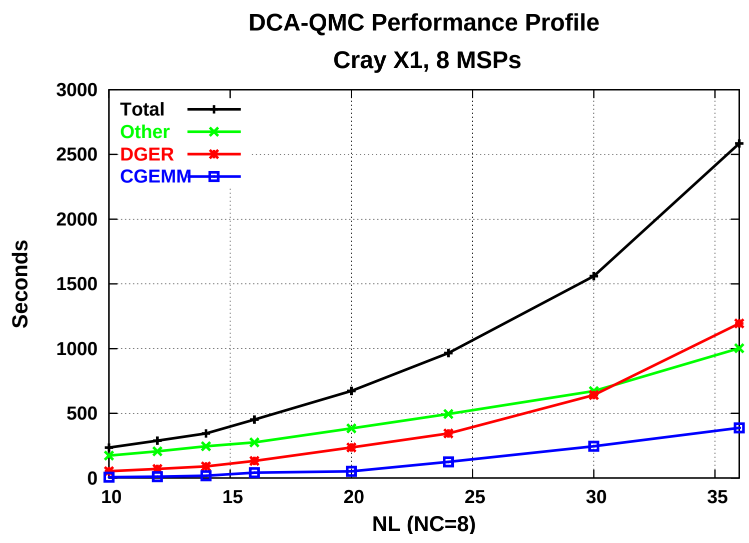

Figure 30: DCA-QMC: performance profile on the Cray X1

5 Conclusions

In this early evaluation of the Cray X1, excellent performance was observed with some of the standard computational benchmarks and with the targeted application codes. Very poor performance was also observed, and it is clear that some tuning will be required of most any code that is ported to the system, including codes that were designed for or optimized on other vector systems. However, when care is taken to optimize the codes, performance has been excellent when compared to the nonvector systems. The codes that have been easiest to port and tune have been those whose computationally intensive processes involved calls to math library routines, especially the BLAS.

The memory bandwidth in the SMP node is excellent, and both SHMEM and Co-Array Fortran provide significant performance enhancements for codes whose performance is sensitive to latency. It is unfortunate that this is required, and we hope that MPI small-message performance can yet be improved. However, MPI large message performance is very good. With regard to SSP and MSP mode, both approaches work. Which is best for a given application is too sensitive to the application, problem size, and number of processors to make any definitive statements. For a fixed size problem, it is typically best to use fewer, more powerful processors (MSPs), in order to minimize parallel overhead. On the Cray X1, this can be offset by inefficiencies in using the SSPs in an MSP, i.e., streaming overhead or inefficiency. In our limited experience, the compilers are quite effective at identifying and exploiting multistreaming, and coding for vectorization and streaming is often easier than for vectorization alone.

When comparing performance between platforms, comparing performance for equal numbers of processors ignores cost. We have not attempted to address this issue in this paper. However, for a number of the codes described here, the vector systems achieve a level of performance that is not possible for any but very large versions of the nonvector systems we have examined. For fixed size problems, such as the POP x1 benchmark, lack of scalability prevents any of the nonvector systems from achieving comparable performance.

With respect to application code performance, we (and others) are porting and optimizing a number of other codes, not all of which are working as well as those reported here. We have not described our experiences with these because of our lack of understanding of the performance issues. It takes longer to determine (accurately) whether a poorly performing code is a poor match for the Cray X1 architecture, needs additional code restructuring, or is limited by a performance bug in the system software. However, even with our limited experience, it is our opinion that many important application can be ported and optimized on the Cray X1 and achieve significant performance. We also expect performance to improve over time, as the system software and the application code optimizations mature.

6 Acknowledgements

The NEC SX-6 version of POP and POP performance data on the Earth Simulator were provided by Dr. Yoshikatsu Yoshida of the Central Research Institute of Electric Power Industry. The modification to POP using Co-Array Fortran was provided by John Levesque of Cray. POP performance data for the SGI Origin3000 were provided by Dr. Phillip Jones of Los Alamos National Laboratory. POP performance data for the IBM SP at the National Energy Research Scientific Computing Center (NERSC) were provided by Dr. Tushar Mohan of NERSC. We also gratefully acknowledge the following institutions for access to their HPC systems:

- Pittsburgh Supercomputing Center for access to the HP AlphaServer SC at PSC
- Center for Computational Sciences at ORNL for access to the IBM p690, IBM SP, and Cray X1 at ORNL
- National Energy Research Scientific Computing Center for access to the IBM SP at NERSC

References

[1] D. H. Bailey, T. Harris, R. F. V. der Wijngaart, W. Saphir, A. Woo, and M. Yarrow, *The NAS Parallel Benchmarks 2.0*, Tech. Rep. NAS-95-010, NASA Ames Research Center, Moffett Field, CA, 1995.

[2] M. B. Blackmon, B. Boville, F. Bryan, R. Dickinson, P. Gent, J. Kiehl, R. Moritz, D. Randall, J. Shukla, S. Solomon, G. Bonan, S. Doney, I. Fung, J. Hack, E. Hunke, and J. Hurrel, *The Community Climate System Model*, BAMS, 82 (2001), pp. 2357–2376.

[3] A. S. Bland, J. J. Dongarra, J. B. Drake, T. H. Dunigan, Jr., T. H. Dunning, Jr., G. A. Geist, B. Gorda, W. D. Gropp, R. J. Harrison, R. Kendall, D. Keyes, J. A. Nichols, L. Oliker, H. Simon, R. Stevens, J. B. White III, P. H. Worley, and T. Zacharia, *Cray X1 Evaluation*, Tech. Rep. ORNL/TM-2003/67, Oak Ridge National Laboratory, Oak Ridge, TN, March 2003.

[4] J. Candy and R. E. Waltz, *An Eulerian gyrokinetic-Maxwell solver*, J. of Computational Physics, 186 (2003), pp. 545–581.

[5] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, *Introduction to UPC and language specification*, Technical Report CCS-TR-99-157, Center for Computing Sciences, 17100 Science Dr., Bowie, MD 20715, May 1999.

[6] Cray Inc., *Cray X1*. http://www.cray.com/products/systems/x1/.

[7] J. Dongarra, J. D. Croz, I. Duff, and S. Hammarling, *A set of level 3 basic linear algebra subprograms*, ACM Trans. Math. Software, 16 (1990), pp. 1–17.

[8] K. Feind, *Shared Memory Access (SHMEM) Routines*, in CUG 1995 Spring Proceedings, R. Winget and K. Winget, ed., Eagen, MN, 1995, Cray User Group, Inc., pp. 303–308.

[9] R. Hockney and M. B. (Eds.), *Public international benchmarks for parallel computers, ParkBench committee report-1*, Scientific Programming, 3 (1994), pp. 101–146.

[10] P. W. Jones, *The Los Alamos Parallel Ocean Program (POP) and coupled model on MPP and clustered SMP computers*, in Making its Mark – The Use of Parallel Processors in Meteorology: Proceedings of the Seventh ECMWF Workshop on Use of Parallel Processors in Meteorology, G.-R. Hoffman and N. Kreitz, eds., World Scientific Publishing Co. Pte. Ltd., Singapore, 1999.

[11] T. A. Maier, M. Jarrell, J. Hague, J. B. White III, and T. C. Schulthess, *Decisive change in the understanding of high-temperature superconductivity through improved computing capability*, 2003. DOE MICS Highlight.

[12] J. D. McCalpin, *Memory Bandwidth and Machine Balance in Current High Performance Computers*, IEEE Computer Society Technical Committee on Computer Architecture Newsletter, (1995). http://tab.computer.org/tcca/news/dec95/dec95.htm.

[13] MPI Committee, *MPI: a message-passing interface standard*, Internat. J. Supercomputer Applications, 8 (1994), pp. 165–416.

[14] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, *Global Arrays: A portable 'shared-memory' programming model for distributed memory computers*, in Proceedings of Supercomputing '94, Los Alamitos, CA, 1994, IEEE Computer Society Press, pp. 340–349.

[15] R. W. Numrich and J. K. Reid, *Co-Array Fortran for parallel programming*, ACM Fortran Forum, 17 (1998), pp. 1–31.

[16] OpenMP Architecture Review Board, *OpenMP: A proposed standard api for shared memory programming*. (available from http://www.openmp.org/openmp/mp-documents/paper/paper.ps), October 1997.

[17] J. R. Taft, *Achieving 60 GFLOP/s on the production CFD code OVERFLOW-MLP*, Parallel Computing, 27 (2001), pp. 521–536.

[18] A. J. van der Steen, *The benchmark of the EuroBen group*, Parallel Computing, 17 (1991), pp. 1211–1221.

[19] A. J. Wallcraft, *SPMD OpenMP vs MPI for Ocean Models*, in Proceedings of the First European Workshop on OpenMP, Lund, Sweden, 1999, Lund University. http://www.it.lth.se/ewomp99.

[20] P. H. Worley and I. T. Foster, *PSTSWM: a parallel algorithm testbed and benchmark code for spectral general circulation models*, Tech. Rep. ORNL/TM–12393, Oak Ridge National Laboratory, Oak Ridge, TN, (in preparation).

[21] P. H. Worley and B. Toonen, *A users' guide to PSTSWM*, Tech. Rep. ORNL/TM–12779, Oak Ridge National Laboratory, Oak Ridge, TN, July 1995.