NAS Experience with the Cray X1

Rupak Biswas, Subhash Saini, Sharad Gavali, Henry Jin, Dennis Jespersen, M. Jahed Djomehri, Nateri K. Madavan, and Cetin Kiris

NAS Division, NASA Ames Research Center, Moffett Field, CA 94035
{*rbiswas, saini, gavali, hjin, jesperse, djomehri, madavan, kiris*}*@nas.nasa.gov*

ABSTRACT:
A Cray X1 computer system was installed at the NASA Advanced Supercomputing (NAS) facility at NASA Ames Research Center in 2004. An evaluation study of this unique high performance computing (HPC) architecture, from the standpoints of processor and system performance, ease of use, and production computing readiness tailored to the needs of the NAS scientific community, was recently completed. The results of this study are described in this article. The evaluation included system performance and characterization using a subset of the HPC Challenge benchmarks and NAS Parallel Benchmarks, as well as detailed performance on scientific applications from the computational fluid dynamics (CFD) and Earth science disciplines that represent a major portion of the computational workload at NAS. Performance results are discussed and compared with the main HPC platform at NAS — the SGI Altix.

KEYWORDS: Cray X1, Benchmarking, Applications, Supercomputing, HPC, CFD

1 Introduction

In mid-2004, a small Cray X1 computer system was installed at the NASA Advanced Supercomputing (NAS) facility at NASA Ames Research Center for evaluation purposes. Although the main computer platforms at NAS in recent years have been based on distributed shared memory machines from SGI, NAS has a rich history in the use of vector machines from Cray such as the XMP, YMP, Cray 2, C-90, J90, and SV1. The Cray X1, introduced in 2003, is a scalable vector system that offers high-speed custom vector processors, high memory bandwidth, and an exceptionally high-bandwidth, low-latency interconnect linking the nodes. A significant feature of the X1 is that it combines the processor performance of traditional vector systems with the scalability of modern microprocessor-based architectures. It is the first vector supercomputer designed to scale up to thousands of processors with a single system image.

The X1 system at NAS was evaluated from the standpoints of processor and system performance, ease of use, and production computing readiness at NAS. This paper describes details of our study focusing on scientific applications and the computational needs of the science and engineering research communities that NAS serves.

The primary goals of our evaluation were to: evaluate kernel benchmarks and application codes of relevance to NAS and compare performance with systems from other HPC vendors; determine effective code porting and performance optimization techniques; determine the most efficient approaches for utilizing the Cray X1; and predict scalability both in terms of problem size and processor counts. While the performance of individual kernels may be predicted with detailed performance models and limited benchmarking, the behavior of full applications and the suitability of this system as a production scientific computing resource can only be determined through extensive experiments conducted on a real system using applications that are representative of the daily workload.

This paper summarizes the results of our evaluation of this unique architecture when using a va-

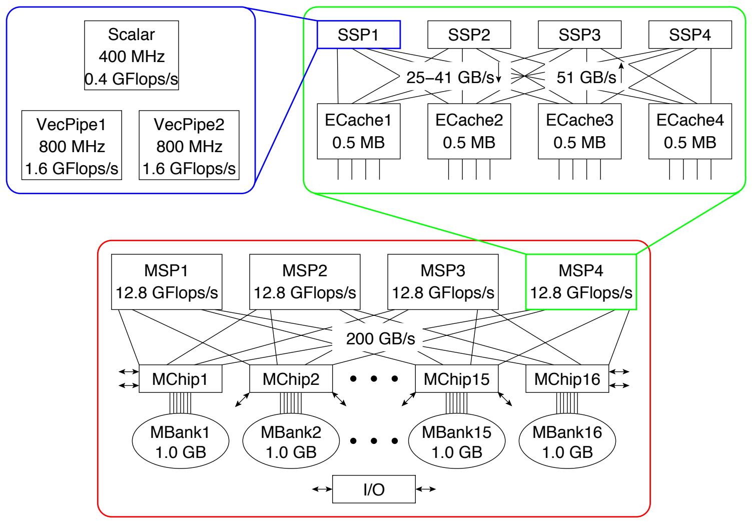

Figure 1: Architecture of the X1 node (in red), MSP (in green), and SSP (in blue).

riety of microbenchmarks, kernel benchmarks, and full-scale application codes from the CFD and Earth science domains. These scientific disciplines account for a substantial portion of the computing cycles at NAS. We also report on our experiences with using the Co-Array Fortran (CAF) programming model on the X1 for one of the kernel benchmarks as well as a full-scale CFD application code.

2 Cray X1 Architecture

The Cray X1 computer is designed to combine the traditional strengths of vector systems with the generality and scalability features of superscalar cache-based parallel architectures (see schematic in Fig. 1). The computational core, called the single-streaming processor (SSP), contains two 8-stage vector pipes running at 800 MHz. Each SSP contains 32 vector registers holding 64 double-precision words, and operates at 3.2 GFlops/s peak. All vector operations are performed under a bit mask, allowing loop blocks with conditionals to compute without the need for scatter/gather operations. Each SSP can have up to 512 addresses simultaneously in flight, thus hiding the latency overhead for a potentially significant number of memory fetches. The SSP also contains a two-way out-of-order superscalar processor running at 400 MHz with two 16 KB caches (instruction and data). The scalar unit operates at an eighth of the vector performance, making a high vector operation ratio critical for effectively utilizing the underlying hardware.

The multi-streaming processor (MSP) combines four SSPs into one logical computational unit. The four SSPs share a 2-way set associative 2 MB data ECache, a unique feature for vector architectures that allows extremely high bandwidth (25–51 GB/s) for computations with temporal data locality. An X1 node consists of four MSPs sharing a flat memory through 16 memory controllers (MChips). Each MChip is attached to a local memory bank (MBank), for an aggregate of 200 GB/s node bandwidth. Additionally, MChips can be used to directly connect up to four nodes (16 MSPs) and participate in remote address translation. To build large configurations, a modified 2D torus interconnect is implemented via specialized routing chips. The torus topology allows scalability to large processor counts with relatively few links although it suffers from limited bisection bandwidth. The X1 is a globally addressable architecture, with specialized hardware support that allows processors to directly read or write remote memory addresses in an efficient manner.

The X1 programming model leverages parallelism hierarchically. At the SSP level, vector instructions allow 64 SIMD operations to execute in a pipeline fashion, thereby masking memory latency and achieving higher sustained performance. MSP parallelism is obtained by distributing loop iterations across the four SSPs. The compiler must therefore generate both vectorizing and multistreaming instructions to effectively utilize the X1. Intra-node parallelism across the MSPs is explicitly controlled using shared-memory directives such as OpenMP or Pthreads, while traditional message passing via MPI is used for coarse-grain parallelism at the internode level. In addition, the hardware-supported globally addressable memory allows efficient implementations of one-sided communication libraries (SHMEM, MPI-2) and implicitly parallel languages such as Unified Parallel C (UPC) and CAF.

3 The NAS Cray X1 System

The Cray X1 system at NAS is a small 4-node configuration that can be expanded as needed. Since one node is reserved for the operating system and other tasks, only three nodes (12 MSPs or 48 SSPs) are available to the user. This limited the size of the application test cases that could be evaluated. The machine has an 800 MHz clock with a peak computation rate of 12.8 GFlops/s per MSP and system peak perfomance of 204.8 GFlops/s. The central main memory is 64 GB and there are 4 TB of FC RAID disk space. The operating system is UNICOS, VMP 2.4.10 with single-system image.

4 Benchmarks

As mentioned earlier, we first evaluated the NAS X1 system using a variety of microbenchmarks and kernel benchmarks. The results of this evaluation are described in this section, followed by the performance behavior on various full-scale application codes.

4.1 Microbenchmarks

The HPC Challenge (HPCC) benchmarks [20] were used for microbenchmarking the X1. These benchmarks are multi-faceted and provide comprehensive insight into the performance of modern high-end computing systems. They are intended to test various attributes and stress not only the processors but also the memory subsystem and system interconnects. They are a good indicator of how modern large-scale parallel systems will perform across a wide spectrum of real-world applications.

The following seven components of the HPCC benchmarks were used to measure the performance of the NAS X1 for comparison with the SGI Altix 3700 (a node of the Columbia supercluster). Note that an Altix contains 512 processors connected with SGI NUMAflex, powered by 1.5 GHz Itanium2 processors with 6 MB L3 cache.

- G-PTRANS implements a parallel matrix transpose given by $A = A + B^T$. It measures network communication capacity using several processor pairs simultaneously communicating with each other.

- G-Random Access measures the rate (in giga (billions) updates per second) at which the computer can update pseudo-random locations of its memory.

- EP-Stream measures sustainable memory bandwidth and the corresponding computation rate for simple vector kernels. All the computational nodes execute the benchmark at the same time and the arithmetic average is reported.

- G-FFTE performs FFTs across the entire computer by distributing the input vector in block fashion across the nodes. It measures the floating-point rate of execution of double precision complex Discrete Fourier Transform (DFT).

- EP-DGEMM measures the floating-point execution rate of double precision real matrix-matrix multiplication performed by the DGEMM subroutine from BLAS. All nodes simultaneously execute the benchmark and the average rate is reported.

- Random Ring Bandwidth reports bandwidth achieved per CPU in a ring communication pattern where the nodes are ordered randomly. The result is averaged over various random assignments of processors in the ring.

- Random Ring Latency reports latency in a ring communication pattern. Again, the communicating nodes are ordered randomly and the result is averaged over various random assignments of processors in the ring.

Table 1 compares performance results for the HPCC benchmarks on the Cray X1 and SGI Altix, both installed at NAS. All results are for a baseline run on 48 CPUs with no tuning or optimization performed for either system. Input data for both systems were identical. From the table, observe

Table 1: Baseline HPCC benchmark performance on 48 CPUs for the X1 and Altix, both at NAS.

Benchmark	Unit	X1	Altix
G-PTRANS	GB/s	0.02513	0.8901
G-Random Access	GU/s	0.00062	0.0017
EP-Stream Triad	GB/s	62.56547	2.4879
G-FFTE	GF/s	0.19217	0.6322
EP-DGEMM	GF/s	9.88915	5.4463
Random Ring BW	GB/s	2.41081	0.7459
Random Ring Lat.	μs	13.71860	4.5552

that the Altix outperforms the X1 for the PTRANS, Random Access, FFTE, and Ring Latency benchmarks, while the reverse is true for the Stream, DGEMM, and Ring Bandwidth benchmarks.

4.2 Kernel Benchmarks

The NAS Parallel Benchmarks (NPB) [1, 2] have been widely used to test the capabilities of parallel computers and parallelization tools. The original NPB suite [1] consisted of five kernels and three compact CFD applications, given as "pencil and paper" specifications. The five kernels mimic the computational core of key numerical methods and the three compact applications reproduce much of the data movement and computations found in full-scale CFD codes. Reference implementations were subsequently provided [2] as NPB 2, using MPI as the parallel programming paradigm. The NPB 2.3 release included sequential code that was essentially a stripped-down version of the MPI implementation.

Recent NPB development efforts have focused on cache performance optimization, inclusion of other programming paradigms such as HPF and OpenMP [8, 9], and the addition of new benchmarks [7, 21]. NPB 3 represents the latest version of the benchmarks and incorporates these recent developments.

For the Cray X1 evaluation, a subset of these benchmarks was selected: two kernels (MG, FT) and three compact applications (BT, SP, LU). This subset includes some of the different types of numerical methods found in various typical applications. Both the MPI and OpenMP versions of these benchmarks in the latest NPB 3.2 distribution [15] were used in the evaluation.

Vectorization issues were given due consideration during the development of NPB 2. As a result, the NPB 2.x code base is expected to run reasonably well on vector machines without much modification. Unfortunately, the cache-friendly optimization for NPB 3.x limited vectorization in some of the benchmarks, notably BT and SP. Although the NPB 2.x code base could have been picked for the X1 evaluation, by choosing NPB 3.2 we were also able to examine the effort needed to port cache-optimized codes to a vector machine like the X1.

Without any changes, both BT and SP (MPI and OpenMP versions) performed poorly on the X1 (performance less than 4% of peak) and indicated the need for additional work to improve performance. The Cray Fortran compiler provides a very useful *loopmark* option that produces a source listing indicating the loops that were optimized (streamed and/or vectorized) and the reasons why other loops were not. Using the loopmark listings, we determined that the main loops in the BT and SP solver routines were not being vectorized, mainly due to complicated loop structures, use of small work arrays, and subroutine calls. We made the following modifications:

– promoted work array dimensions from 3D to 4D so that large, complicated loops could be split into smaller, less complicated loops;
– inlined subroutine calls in BT with the "`!dir$ inlinealways`" directive; and
– added the "`!dir$ concurrent`" directive for loops that are parallel but could not be vectorized by the compiler.

With these changes, the codes vectorized fully.

There are two implementations of the LU benchmark in NPB 3.2: one uses a *pipeline* approach while the other uses a *hyper-plane* technique [9]. The pipeline strategy is cache-friendly, but cannot be easily vectorized; the hyper-plane approach vectorizes easily, but does not utilize cache efficiently. For obvious reasons, we used the hyper-plane version for the X1 evaluation.

The two kernel benchmarks, MG and FT, did not require many changes to run on the X1. In order to improve streaming performance, we used the "`!csd$`" directive for one loop in MG that the compiler failed to stream. We also tried to add the "`!csd$`" directive to loops in FT, but encountered a runtime error that is yet to be resolved. In order to fully utilize the vector capability, a block size of 64 was used for the FFT algorithm in FT.

Recall that on the X1, an application can be compiled and run in either MSP or SSP mode. Since one MSP contains four SSPs, in principle, the performance of an MSP should be four times that of an SSP. We examined the performance of the selected NPBs running in both MSP and SSP modes, and compared them with those obtained on the Altix.

In the following, results obtained for MG, FT, SP, and BT are first discussed. The LU benchmark is worth a special note later. The MPI implementations of SP and BT require the number of processors to be a perfect square, while MG, FT, and LU require the number to be a power of two. On the NAS X1, we can thus run these benchmarks only up to 8/9 MSPs or 32/36 SSPs. Although OpenMP codes can run on any number of processors, they are limited to one shared-memory node, that is, four MSPs or 16 SSPs.

Table 2 lists the measured performance of four benchmarks on the X1 (in both MSP and SSP modes) and the Altix. Since one MSP is equivalent to four SSPs, the results are tabulated accordingly. The following observations can be drawn:

- For MPI codes, SSP mode generally performs better than MSP mode, except for SP where the 9-MSP run outperformed the 36-SSP run.
- For OpenMP codes, the SSP mode is consistently better, indicating streaming under OpenMP is not as efficient.
- Except for SP, streaming seems to be done poorly. Use of the "`!csd$`" directive may be required in some of these cases to improve streaming.
- Through 16 SSPs (the maximum number of threads allowed to run a pure OpenMP code), the OpenMP versions scaled better than their MPI counterparts. This might be related to runtime fluctuations that are discussed later. The situation is reversed on the Altix: the MPI versions scaled better than the OpenMP codes.
- The performance of one SSP is roughly equivalent to one Altix processor, except for BT where the Altix processor shows roughly twice the performance of an SSP.

Because the MPI version of LU uses the pipeline approach (not suitable for the X1) and there is no hyper-plane MPI implementation, we only examined the performance of the OpenMP code. These results are reported in Table 3. For comparison, performance of the pipeline and hyper-plane versions of LU on the Altix are also included in Table 3. Observe that the hyper-plane version performed very well on the X1 and very similar to the pipeline version on the Altix. The performance in MSP mode is about 15% worse than in SSP mode, which seems reasonable. On the other hand, the hyper-plane version did not scale well on the cache-based Altix system and showed no improvement beyond 16 processors.

Table 2: MPI and OpenMP performance (in GFlops/s) of the four NPBs for Class B problem size on the Altix and X1.

		Altix		X1-SSP		X1-MSP		
NPB	CPU	MPI	OMP	MPI	OMP	MSP	MPI	OMP
MG.B	1	1.433	1.610	0.717	0.706			
	2	1.855	2.083	1.405	1.378			
	4	3.463	4.145	2.137	2.657	1	0.940	1.390
	8	7.069	7.274	2.455	4.906	2	1.900	2.720
	16	13.174	11.356	3.475	9.532	4	3.784	5.623
	32	23.759	14.128	6.371		8	4.351	
FT.B	1	0.714	0.879	1.076	1.077			
	2	0.963	1.669	2.064	2.012			
	4	1.912	3.020	3.443	3.916	1	0.962	1.058
	8	4.229	5.580	3.989	5.189	2	1.886	2.033
	16	8.131	10.279	6.188	7.740	4	3.728	4.071
	32	16.737	16.138	10.106		8	7.448	
SP.B	1	0.583	0.960	0.899	0.924			
	2		1.511		1.824			
	4	1.505	2.704	2.381	2.691	1	2.061	2.009
	9	3.971	4.961	4.211	4.317	2		3.860
	16	7.976	7.860	5.874	7.884	4	5.662	7.381
	36	17.944	16.627	9.819		9	11.425	
BT.B	1	1.163	1.672	0.916	0.821			
	2		3.045		1.632			
	4	4.086	5.630	2.930	2.392	1	0.965	0.988
	9	8.781	11.372	4.791	4.561	2		1.955
	16	15.316	16.492	6.967	8.431	4	4.892	3.925
	36	34.821	35.401	12.209		9	10.192	

Table 3: OpenMP performance (in GFlops/s) of the LU NPB for Class B problem size on the Altix and X1.

		Altix		X1-SSP	X1-MSP	
NPB	CPU	pipe-l	hyper-p	hyper-p	MSP	hyper-p
LU.B	1	1.020	0.750	1.122		
	2	2.608	1.663	2.001		
	4	4.658	2.748	3.162	1	2.739
	8	7.406	4.293	6.173	2	4.868
	16	11.522	5.803	11.070	4	8.909

We observed large runtime fluctuations when running MPI processes in SSP mode on the X1. This

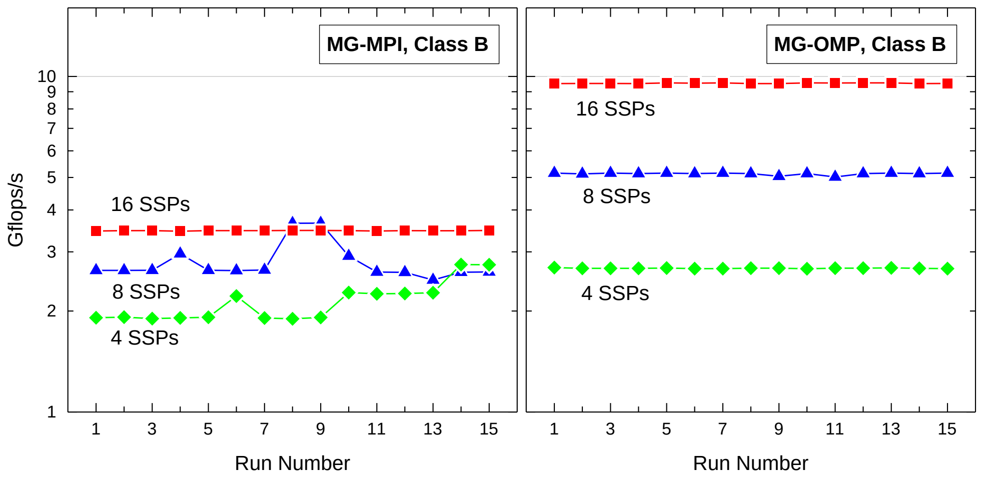

Figure 2: Timing variations observed from running the MG NPB for Class B problem size in SSP mode: MPI (left) and OpenMP (right).

effect can be seen in the left panel of Fig. 2 from 15 separate runs of the MG benchmark on four and eight SSPs. Of the 15 runs using eight SSPs, there are two that achieved an optimal performance of 3.65 GFlops/s, but most were close to a baseline of 2.64 GFlops/s; the performance difference is more than 30%. In contrast, the 16-SSP MPI runs show very little variation (less than 5%). In particular, we do not see a similar variation when running the OpenMP codes, as illustrated in the right panel of Fig. 2. It is also interesting to note that the best 8-SSP MPI runs outperform the 16-SSP MPI runs, and the best 4-SSP MPI runs show similar performance as the corresponding OpenMP runs, although OpenMP scales significantly better.

According to Cray, the large timing variations observed in the SSP runs are related to the design of the X1. Each MSP has 4 SSPs, ECache, and a single high speed path to memory. All SSPs in an MSP share the ECache and the memory path. When running an application in SSP mode, there is no provision for controlling how processes are mapped to the SSPs. For example, a 4-SSP application could end up using SSPs from 1 to 4 MSPs. If all of the processes are mapped to a single MSP, then only one ECache and one memory path would be utilized. The reverse would be true if one SSP each from four different MSPs were used. Both cache and memory performance can be significantly different depending on where processes are mapped. The only way to get consistent multi-processor timings in SSP mode is to use multiples of 16 SSPs, as demonstrated in Fig. 2. This is only an issue for SSP mode and does not apply to MSP mode. Evidently, OpenMP threads are scheduled differently and do not experience the same problem.

We also examined more detailed information from the hardware performance counters reported by the `pat_hwpc` tool. Table 4 shows the results for the five benchmarks running on a single SSP or MSP. The vectorization percentage is above 99% for all the benchmarks. The average vector length ranges from 44 to 64 in SSP mode; however, this value drops in MSP mode by more than 40% for MG and 70% for FT, indicating that some loops were both streamed and vectorized. This is a major factor that limits performance in MSP mode. The LU benchmark is the only one that displays an excellent floating-point-to-load ratio that seems to have a direct impact on performance. Reducing memory loads and enhancing streaming are the keys to further improving NPB performance on the X1.

Table 4: Single-processor performance for the five NPBs reported from the hardware counters on the X1.

	FP Ops	Vec.	Vector Length		% of Peak	
NPB	/Load	%	SSP	MSP	SSP	MSP
MG.B	0.85	99.4	44.65	27.21	24.0	11.8
FT.B	0.94	99.7	64.00	17.49	35.5	8.3
SP.B	1.10	100.0	49.88	35.37	28.9	17.4
BT.B	0.95	100.0	60.87	49.76	25.8	8.3
LU.B	1.75	99.8	55.71	42.80	35.5	21.8

4.3 CAF Version of SP Benchmark

Cray recommends the use of CAF [5, 6] because it has lower overhead than MPI for inter-process communication. CAF is a simple syntactic extension to Fortran95 that converts it into a robust, efficient parallel language. It is a shared-memory programming model based on one-sided communication where data can be directly referenced from any processor rather than by explicit messages.

The co-array extension allows the programmer to distribute the data among memory images using normal Fortran array syntax. A co-array variable is defined by adding a square bracket to denote the image of this variable on a processor indicated within this square bracket. The square bracket can be omitted when referencing a local image of this array. Since most data references in a parallel code are local, the presence of co-array syntax will indicate inter-processor communication. Global barriers for synchronization are also provided.

When starting from MPI code, performance improvements can be obtained by identifying the message-passing calls where most of the computation time is spent and replacing them with CAF constructs. In this evaluation, however, we chose an alternate approach. We developed a CAF version from the serial rather than the MPI implementation in order to evaluate the programming effort involved. We selected the SP benchmark from the NPB suite that deals with the solution of systems of scalar, pentadiagonal equations [1]. Since the latest version of the benchmarks (NPB 3) focus on cache optimization, we chose NPB 2.3 as our starting point as it took vectorization issues into consideration. Note that both serial and MPI implementations are available in NPB 2.3. The code structure was similar for MPI and CAF, and tailored to vector architectures such as the X1.

The SP code involves an initialization phase followed by iterative computations over multiple time steps. At each time step, the right hand sides of the equations are calculated. A banded system is then solved in three computationally intensive bidirectional sweeps along each of the x, y, and z directions. The flow variables are finally updated.

The data structure for this code is a cube of size n which can be represented in arrays with dimension $(imax, jmax, kmax)$, where $imax = jmax = kmax = n$. For class A problem size, n is 64; for class B, it is 102. The data decomposition strategy we chose for the CAF version was to equally distribute planes in the z direction across the processors. Thus, using co-array syntax, a variable Q is declared as $Q(imax, jmax, kmax/np)[*]$, where np is the number of processors. By dividing the last dimension by the processor count, memory storage is reduced as it causes the co-array to be allocated on all np processors. For each processor to have an equal number of z planes, the value of np should be such that $kmax$ is exactly divisible by np.

When sweeping in the x, y, and z directions in subroutines `x_solve`, `y_solve`, and `z_solve`, the process of forward elimination and backward substitutions leads to recursion in the three indices, respectively. For the x and y directions, the recursion does not pose any difficulty for our data decomposition strategy since all the data in these dimensions is available locally on each processor. However, for the z direction sweep in subroutine `z_solve`, recursion creates a problem since the data is distributed. This can be dealt with by swapping the j and k indices, thereby transposing the data. While this may seem to be a considerable overhead, the advantage for architectures like the X1 is that an entire plane of data is available for effective vectorization. The resulting performance gain far outweighs the transpose overhead which is nominal on the X1 due to its high bandwidth shared-memory communication architecture.

Developing a CAF version of the SP code from the serial implementation gave us the opportunity to evaluate this programming model on the X1. The code required many modifications to account for proper data decomposition. All loop limits had to be changed. The most time consuming part involved modifying the `z_solve` subroutine and related routines to accommodate the transposed variables. We developed a utility tool to simplify the tedious tasks. With the CAF syntax, it was relatively simple to write the array transposition routines and distribute the arrays across the various processors.

Figure 3 shows results for the class A and B problem sizes of SP in MSP and SSP modes for both MPI and CAF versions. Results indicate that CAF performed better than MPI for both modes. Note that the results for MSP are shifted by a factor of four since one MSP consists of four SSPs. As the number of processors (MSPs or SSPs) grows, the advantage of CAF over MPI increases. As expected, the performance of both CAF and MPI versions improves as the problem size increases from class A to class B. For class A, the average vector length

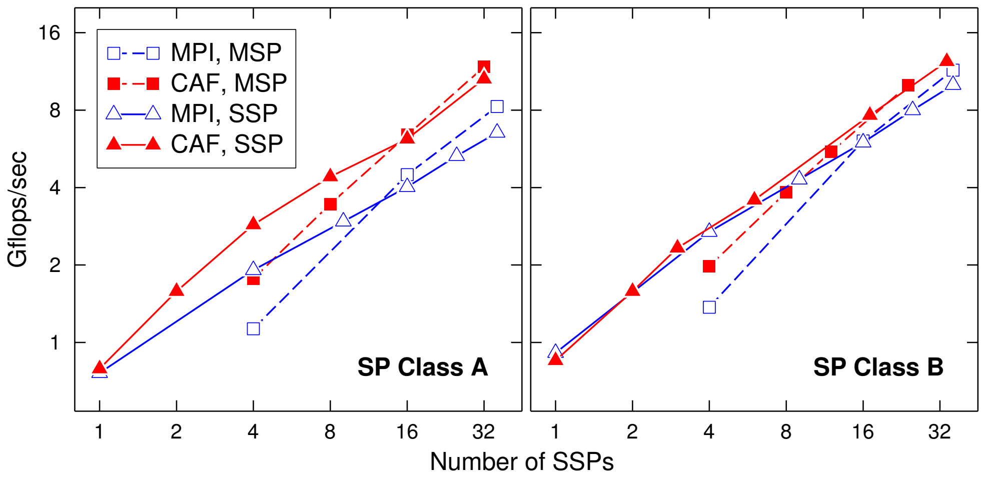

Figure 3: Performance of CAF and MPI versions of the SP NPB: Class A (left) and Class B (right).

in the CAF implementation dropped from 43 to 26 when going from 1 to 32 SSPs, whereas in the MPI version, it dropped from 39 to 10 between 1 and 36 SSPs. The higher vector length is partially responsible for the better performance of CAF. On the other hand, the MPI version uses asynchronous communication while computations are performed. We could not identify this feature in CAF but, even if available, would not be able to accommodate our data partitioning strategy. We conclude that for the SP benchmark, CAF performs better than MPI for classes A and B.

Performance comparisons between MSP and SSP modes also shows similar trends for both CAF and MPI. When the number of processors is small, SSP performance is better than MSP. As the number of processors increases, the situation is reversed. Note that we did not use any code modifications or compiler directives to take advantage of streaming that could potentially improve performance in MSP mode.

5 Scientific Applications

The various application codes chosen for the X1 evaluation were the CFD codes OVERFLOW, ROTOR, and INS3D, and the Earth science code GCEM3D. We address a variety of issues such as application performance using various parallel programming paradigms and execution modes. Detailed results are presented in the subsections below.

5.1 OVERFLOW

OVERFLOW is a large CFD application code for computing aerodynamic flow fields around geometrically complex bodies. Development began in 1990 at NASA Ames [4]; it currently comprises of 1000 subroutines and 100,000 lines of code. OVERFLOW is written in Fortran77 with a small amount of C to handle memory allocation. Upgrades and enhancements continued up through 2003, at which time the code was frozen and effort was directed toward a second-generation version which would combine the features of OVERFLOW and an earlier descendant (called OVERFLOW-D that had a set of capabilities for dealing with bodies in relative motion). The new code is called OVERFLOW-2. Here we consider only the original code, referred to simply as OVERFLOW.

OVERFLOW numerically solves the Navier-Stokes equations of fluid flow with finite differences in space and implicit integration in time. In a data preparation step (performed independent of the code), structured grids are generated about solid bodies with geometric complexity handled via arbitrary overlapping grids. All data access has a regular pattern with the exception of the transfer of information from one zone to another where required by irregular zonal overlap. OVERFLOW is constructed in a highly modular fashion with most subroutines performing just one conceptual operation on a 2D or 3D array. There are a wide variety of options including numerical methods, boundary conditions, and turbulence models.

OVERFLOW was originally developed with vector supercomputers of the time (Cray YMP, Cray C-90) in mind. The numerically intensive subroutines are highly vectorized: hardware performance monitoring of the code typically indicates 99.9% of the floating-point operations are vector operations. Several years later, concurrency was added via explicit microtasking directives. This allowed the use of multiple processors on machines like the Cray C-90. Effective parallelism was limited to about eight processors because of the small degree of available concurrency when computing on a single zone.

The advent of clusters and distributed memory computers led to efforts to exploit other models of parallelism. One path taken was the hybrid approach with explicit message passing at the outer level and loop-level parallelism at the inner level. Here the outer coarse-grained parallelism occurred across the zones of the overset grid system, while the inner fine-grained parallelism was within each individual zone.

The introduction of large distributed shared memory machines allows explicit message passing and its attendant problems (such as software development and maintenance difficulties, and network performance) to be avoided. Instead one can implement a strategy that uses the shared memory to exchange information among the zones of a multi-zone geometry. The finer loop-level parallelism remains. This specific style of hybrid parallelism has been called Multi-Level Parallelism or MLP [17].

The MLP strategy for OVERFLOW was originally implemented on SGI shared-memory machines. Loop-level parallelism was added via explicit OpenMP directives corresponding to the earlier Cray microtasking directives. Explicit OpenMP directives were also added in some subroutines, typically those containing a triply-nested loop which had been autotasked by the C-90 compiler but which other compilers would not automatically parallelize.

The task of implementing the MLP version on the X1 required some system-specific initializations [3]. In addition, OpenMP was replaced by streaming. Subroutines with triply- or doubly-nested loops needed no changes as the Cray compiler automatically streams the outer loop, as a rule. A few explicit streaming and inlining directives were necessary in cases where parallelism occurred across subroutine boundaries and was not visible to the compiler. A handful of other routines also required minor modifications.

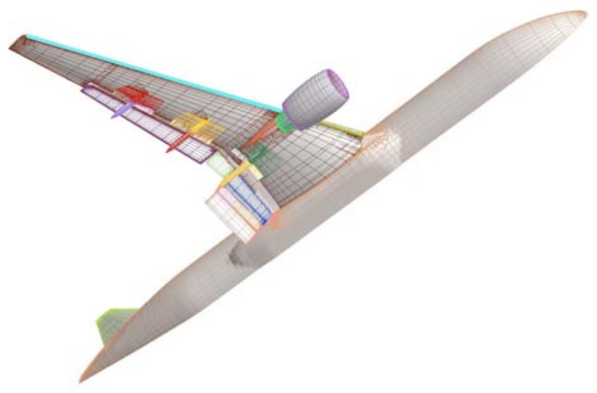

Figure 4: Body surface geometry for a typical transport aircraft showing the plane of symmetry used in the OVERFLOW computations. Different colors denote different computational zones.

In summary, the MLP version of OVERFLOW was run in MSP mode on the X1. Shared memory is used for information exchange between processes while streaming and vectorization are used within each MSP while fine-grained parallelism is via OpenMP. For comparison, this version was also run on the SGI Altix.

A realistic transport aircraft geometry (see Fig. 4) was chosen as a test case. This configuration has 77 zones with almost 23 million grid points and represents a moderately large problem. Performance results are compared in Table 5. All timing data is shown as average wall clock seconds per time step, and ignore startup and shutdown transients. The value directly determines how long a user has to wait for a job to complete once it starts executing. Observe that, for this test case, one X1 MSP is roughly equivalent to 3.5 Altix CPUs (in other words, one X1 SSP is equivalent to about one Altix CPU). Table 5 also shows performance data in microseconds per grid point per time step to give a case-independent measure of machine performance. For reference, a benchmark run of OVERFLOW on a single CPU of the C-90 required 8.455 μs per grid point per time step and operated at 0.47 GFlops/s.

Regression analyses show that the relation between time t and number of CPUs N for the Altix is $t = 48.68 \cdot N^{-0.78}$, while for the X1, the expression is $t = 100.16 \cdot N^{-0.97}$ (here N is the number of SSPs). This indicates that for this set of processor counts, the code scales better on the X1 (exponent nearer -1) than on the Altix.

Table 5: Performance of MLP version of OVERFLOW on the X1 and Altix.

X1			Altix		
MSP	s/step	μs/pt/step	CPU	s/step	μs/pt/step
1	26.205	1.140	4	19.871	0.865
2	13.215	0.575	8	9.893	0.431
4	6.869	0.299	16	5.235	0.228
8	3.481	0.151	32	2.784	0.121
12	2.343	0.102	48	2.152	0.094

Table 6: Hardware performance monitor data for MLP version of OVERFLOW on the X1.

MSP	GF/s	FP Ops /Load	Vec. Len.
1	2.895	1.39	50.20
2	5.462	1.39	50.11
4	9.763	1.39	49.80
8	15.885	1.38	49.23
12	19.666	1.38	48.25

Performance monitoring of this case on the X1 using hardware counters is shown in Table 6. On one MSP, the code attains about 23% of peak speed. The average vector length is reasonable but the code only does about 1.4 floating-point operations per memory load. This is basically due to the highly modular nature of the code as mentioned earlier: most subroutines perform only a small amount of arithmetic operations but access a large amount of memory in the process.

5.2 ROTOR

The ROTOR code was developed at NASA Ames [14, 16] in the late 1980's and early 1990's to accurately simulate the unsteady flow in a gas turbine stage. This code has been widely distributed and forms the basis of several related application codes that are currently in use in industry, NASA, and other government agencies.

A gas turbine stage typically consists of a row of (stationary) stator airfoils and a row of (rotating) rotor airfoils adjacent to each other in an annular region formed between a hub and outer casing. The flow is inherently unsteady due the motion of the rotor row relative to the stator row, the interaction of the rotor airfoils with the wakes and passage vortices generated upstream, and vortex shedding from the blunt airfoil trailing edges. Accurate simula-

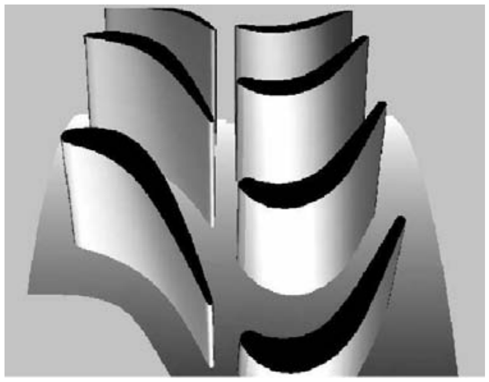

Figure 5: Perspective view of gas turbine stage used in the ROTOR computations. The stator row is on the left and the rotor row is on the right. Flow direction is from left to right, and rotor airfoils rotate toward the viewer.

tions that can capture these unsteady effects can be very useful to gas turbine designers for engine performance optimization, noise minimization, and new design efforts. The ROTOR code accomplishes this by solving the 3D Navier-Stokes equations in a time-accurate manner and accounting for the effects of "stator-rotor interaction" by using a system of computational grids that can move relative to one another.

Figure 5 is a perspective view of the axial turbine stage that is considered in this evaluation study showing the stator (left) and rotor (right) airfoils mounted on the hub (the outer casing is not shown). Figure 6 shows a 2D multiple-zone grid that is used in ROTOR to discretize the flow domain shown in Fig. 5. These 2D grids are wrapped on a cylindrical surface conforming to the turbine hub and several such grids are stacked from the hub outward to form the 3D grid. The governing equations together with appropriate boundary conditions are solved on this grid until a periodic solution is obtained.

To provide a flavor for the capabilities of such simulations, some typical results from an earlier study [14] using the ROTOR code are shown in Fig. 7. The figure depicts the instantaneous pressure distribution on the airfoil, hub, and rotor tip surfaces. On the stator airfoils, the flow is seen to be almost 2D, while on the rotor airfoils strong 3D effects are seen. Note that the pressure distribution shown in Fig. 7 represents merely one snapshot in time — in order to visualize the unsteady effects,

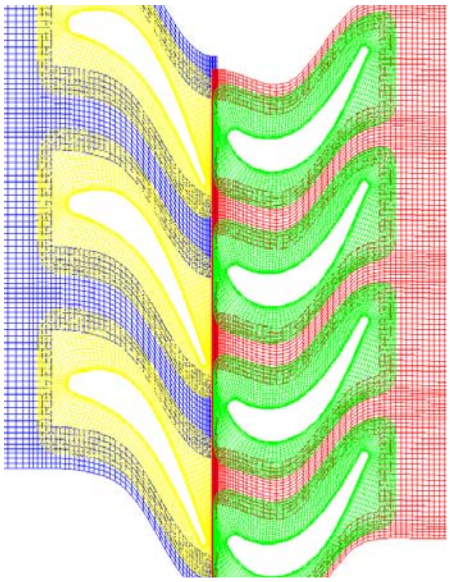

Figure 6: Schematic of 2D patched and overlaid grids typically used to construct the 3D grid. For clarity, a subset of the gridpoints used are shown.

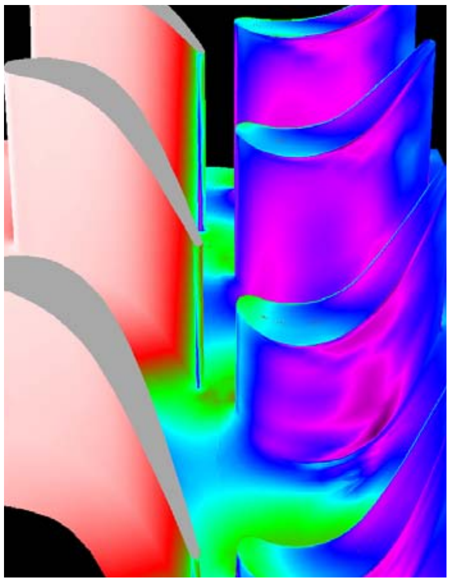

Figure 7: Typical computational results showing the instantaneous pressure distribution in the turbine stage. Darker colors (black, magenta, blue) represent lower pressures while lighter, brighter colors (yellow, red, white) represent higher pressure values.

an animation of such results over multiple time instances is required.

The original ROTOR code targeted Cray vector supercomputers (XMP, YMP, C-90) where it was typically run in serial fashion on a single processor. The code was highly vectorized and generally achieved about 0.5 GFlops/s sustained performance on a single C-90 processor. With the shift in recent years at NAS toward cache-based, distributed shared memory machines such as the SGI Origin and Altix series, it was clear that a multi-level parallel programming paradigm coupled with "cache-friendly" code optimizations was needed to exploit these new architectures. The hybrid MLP approach was therefore adopted to exploit the distributed shared memory feature of these machines in combination with OpenMP directives for loop-level parallelism.

For the X1 evaluation studies reported here, a vectorized version of the MLP-OpenMP code was developed, drawing from past experience with the original Cray serial vector code and the hybrid MLP approach which was used with the "cache-friendly" version of the code. The MLP library routines developed for the SGI Origin were ported to the X1 by Chris Brady of Cray, Inc. Due to time constraints, we did not explore the use of streaming directives (other than those automatically generated by the compiler) and focused mainly on using the machine in SSP mode. (Some results comparing MSP and SSP modes are, however, provided.) This choice was also based on the fact that the test problem requires 12 processors at a minimum. Since the machine at NAS is small (12 MSPs or 48 SSPs available), only one OpenMP thread could be used in MSP mode, while up to four OpenMP threads could be used in SSP mode. As part of this evaluation, a CAF version of ROTOR was also developed for comparison with the shared-memory MLP approach. The CAF implementation also uses OpenMP directives for loop-level parallelism.

The test case chosen for the performance evaluation studies is a 6-airfoil turbine geometry and grid system that is identical to that depicted in Fig. 5 and Fig. 6 with the exception that three airfoils are used in each row (instead of three in the stator and four in the rotor row). This was done to accommodate the problem on the small X1 at NAS. This results in 12 grids or zones, since the region around each airfoil is discretized using two grids. These 12 zones are computed in parallel, and one or several

Table 7: Single-processor performance of ROTOR (optimized for the C-90) on the X1 with only compiler generated automatic streaming.

Test Case	Grid Pnts	Mode	Time (s)	FP Ops /Load	Vec. Len.	Vec. %	GF s	% of Peak
C	0.7M	MSP	12.10	1.15	20.1	99.2	0.77	7.8
		SSP	16.94	1.24	24.9	99.2	0.53	16.6
M	6.9M	MSP	79.42	1.17	30.2	99.5	1.32	10.3
		SSP	135.30	1.26	38.1	99.5	0.75	23.4
F	23.3M	MSP	225.62	1.18	36.6	99.6	1.58	12.3
		SSP	414.24	1.26	42.1	99.6	0.83	25.9

OpenMP threads are used for loop-level parallelism in each zone as needed. Three different grid sizes were used in each zone: coarse (0.7M gridpoints), medium (6.9M gridpoints), and fine (23.3M gridpoints).

Table 7 shows the performance of the original "serial" ROTOR code (optimized for the Cray C-90) on the X1. Results for both MSP and SSP modes are presented. Note that no manual streaming directives were inserted. Observe that the serial code vectorizes well; the average vector lengths range from about 20 to 37 for the coarse to fine grids in MSP mode and from 25 to 42 in SSP mode. In MSP mode, the code achieves 0.77 GFlops/s (7.8% of peak) for the coarse grid, and 1.58 GFlops/s (12.3% of peak) for the fine grid. In SSP mode, it achieves 0.53 GFlops/s (16.6% of peak) and 0.83 GFlops/s (25.9% of peak) for the coarse and fine grids, respectively. The results indicate that the serial code runs more efficiently in SSP mode than in MSP mode. To put these results in historical perspective, we note that the code ran on a single processor of the C-90 at about 0.5 GFlops/s for problem sizes comparable to the medium grid case shown here (where the X1 achieves 1.32 GFlops/s in MSP mode and 0.75 GFlops/s in SSP mode). In all the MSP results presented in this table and others, we emphasize that the use of manually inserted streaming directives could potentially improve performance.

Table 8 compares the performance under MSP and SSP modes for the parallel versions of ROTOR. Results are shown for both the MLP and CAF implementations with one OpenMP thread. Across 12 processors, the MLP code achieves 7.30 and 15.37 GFlops/s for the coarse and fine grids in MSP mode, and 3.60 and 6.41 GFlops/s respectively in SSP mode. The best results are for the fine grid and show the application achieving 10% of peak in MSP mode and 16.69% of peak in SSP mode. Comparison of results between MLP and CAF shows a small improvement (ranging from 1–6%) using CAF in both SSP and MSP modes.

Table 8: Performance of MLP and CAF versions of ROTOR in MSP and SSP modes with one OpenMP thread and 12 processors.

Test Case	Grid Pnts	Mode	Paradigm	Time (s)	GF /s	% of Peak
C	0.7M	MSP	MLP	1.00	7.30	4.75
		SSP	MLP	2.03	3.60	9.38
		MSP	CAF	0.99	7.38	4.80
		SSP	CAF	1.90	3.85	10.03
M	6.9M	MSP	MLP	7.41	12.65	8.24
		SSP	MLP	17.34	5.41	14.09
		MSP	CAF	7.16	13.35	8.69
		SSP	CAF	17.01	5.62	14.64
F	23.3M	MSP	MLP	20.61	15.37	10.00
		SSP	MLP	49.40	6.41	16.69
		MSP	CAF	20.21	16.34	10.64
		SSP	CAF	47.66	6.65	17.32

Table 9: Performance of MLP and CAF versions of ROTOR in SSP mode with multiple OpenMP threads.

				MLP-OMP			CAF-OMP		
Test Case	Grid Pnts	SSP	OMP Thrd	Time (s)	GF /s	Speedup	Time (s)	GF /s	Speedup
C	0.7M	12	1	2.03	3.60	1.00	1.90	3.85	1.00
		24	2	1.06	6.90	1.92	1.02	7.15	1.86
		36	3	0.72	10.10	2.81	0.70	10.51	2.73
		48	4	0.57	12.80	3.56	0.55	13.32	3.45
M	6.9M	12	1	17.34	5.41	1.00	17.01	5.62	1.00
		24	2	8.46	11.12	2.06	8.23	11.61	2.07
		36	3	6.18	15.22	2.81	5.99	15.96	2.84
		48	4	4.81	19.56	3.62	4.60	20.78	3.70
F	23.3M	12	1	49.40	6.41	1.00	47.66	6.65	1.00
		24	2	23.76	13.33	2.08	22.97	13.79	2.07
		36	3	17.24	18.38	2.87	16.49	19.21	2.89
		48	4	13.88	22.82	3.56	12.77	24.81	3.73

Table 9 compares MLP and CAF versions of ROTOR including the effect of multiple OpenMP threads. In this and subsequent tables, we restrict ourselves to SSP mode. Between one to four OpenMP threads are used to spread the test problem across 12 to 48 SSPs. The results for both the MLP and CAF codes are similar, with the CAF version performing slightly better overall. With increasing OpenMP threads, speedups of about 3.6 are noted for both codes for the different grid sizes.

Table 10: Performance comparison of ROTOR: CAF-OpenMP in SSP mode on X1 and MLP-OpenMP on Altix.

Test Case	Grid Pnts	CPU	OMP Thrd	X1 Time (s)	X1 GF /s	X1 Speedup	Altix Time (s)	Altix GF /s	Altix Speedup
C	0.7M	12	1	1.902	3.85	1.00	1.694	4.32	1.00
		24	2	1.023	7.15	1.86	1.035	7.07	1.64
		36	3	0.696	10.51	2.73	0.946	7.74	1.79
		48	4	0.549	13.32	3.45	0.843	8.68	2.01
M	6.9M	12	1	17.010	5.62	1.00	19.292	4.95	1.00
		24	2	8.231	11.61	2.07	11.081	8.62	1.74
		36	3	5.986	15.96	2.84	10.120	9.44	1.91
		48	4	4.598	20.78	3.70	8.889	10.75	2.17

Table 10 compares ROTOR performance between the X1 and the Altix. The comparison is made using the CAF (best performing) implementation on the X1 in SSP mode and using a cache-optimized scalar MLP code on the Altix. It is important to note that the Altix version is still being optimized. Results show that with one OpenMP thread, code performance on the Altix is slightly better than on the X1 (4.32 GFlops/s versus 3.85 GFlops/s). This is probably due to the small problem size and its ability to fit in Altix cache. For larger problem sizes, as evident from the medium grid results, the X1 slightly outperforms the Altix (5.62 GFlops/s versus 4.95 GFlops/s). With increasing OpenMP threads, the code on the X1 also shows much better speedup. In particular, speedups of about 3.6 are achieved with four OpenMP threads on the X1, while the equivalent speedups on the Altix are about 2.1. The reason for this anomalous behavior on the Altix is currently under investigation.

5.3 INS3D

The INS3D code [12], also developed at NASA Ames, solves the incompressible Navier-Stokes equations for both steady-state and unsteady flows and has been used in a variety of applications.

A recent application of the INS3D code has been in performing computations of the unsteady flow through full scale liquid rocket engine pumps and fuel-liners [13]. Liquid rocket turbopumps operate under severe conditions and at very high rotational speeds. The low-pressure-fuel turbopump creates transient flow features such as reverse flows, tip clearance effects, secondary flows, vortex shedding, junction flows, and cavitation effects. Flow unsteadiness that originates from the inducer is considered to be one of the major contributors to the high frequency cyclic loading that results in cycle fatigue. The reverse flow that originates at the tip of an inducer blade travels upstream and interacts with the bellows cavity.

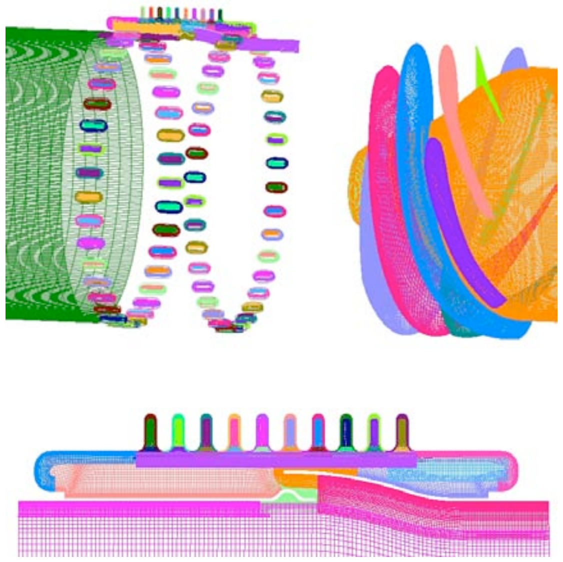
Figure 8: Surface grids used by INS3D for the low pressure fuel pump inducer and the flowliner.

To resolve the complex flow geometry, an overset grid approach is employed in INS3D where the problem domain is decomposed into a number of simple grid components. Connectivity between neighboring grids is established by interpolation at the grid outer boundaries. Addition of new components to the system and simulation of arbitrary relative motion between multiple bodies are achieved by establishing new connectivity without disturbing the existing grids.

A typical computational grid used for pump computations is shown in Fig. 8. This particular grid system has 264 zones and 66 million grid points. Unsteady computations for the flowliner analysis were performed on the Altix platform at NAS. For the purpose of this evaluation, a smaller test case which includes only the S-pipe A1 test section was run on the X1.

Figure 9 displays particle traces colored by axial velocity entering the low pressure fuel pump. The blue particles represent regions of positive axial ve-

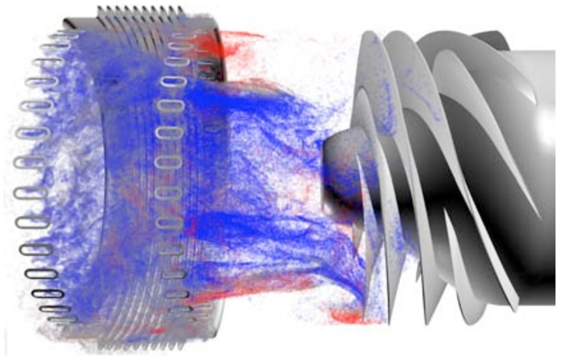

Figure 9: Instantaneous snapshot of particle traces colored by axial velocity values as computed by INS3D.

locity, while the red particles indicate four back flow regions. The gray particles identify the stagnation regions in the flow.

To evaluate the performance of the MPI-OpenMP version of the INS3D code on the X1, a 3D high Reynolds number turbulent flow through the A1-test stand is chosen. The grid for the test case consisted of six zones and a total of 2 million grid points. Each grid contained between 0.3 and 0.4 million grid points allowing for good load balance within each MPI group. For the purpose of testing the scalability of the code in SSP mode with respect to both the number of MPI groups and number of OpenMP threads, we grouped the zones into 1, 2, 3, and 6 MPI groups. For each MPI group size, a number of OpenMP threads ranging from 1 to 8 were used. By varying the number of MPI groups and OpenMP threads, we can determine which methodology scales better on the X1. In MSP mode, we utilized compiler generated streaming and disabled the OpenMP commands; we still ran the test case for all the MPI group sizes in this mode.

Table 11 displays the SSP mode performance using the various MPI groups and OpenMP threads. Note that OpenMP parallelization scales reasonably well up to four OpenMP threads and then begins to level off. On the other hand, the coarse-grained MPI parallelization seems to scale better up to six groups. The INS3D code vectorizes well with average vector lengths of 20 to 45 in SSP mode. The code achieves a high percentage of peak (31%) with one MPI group and one OpenMP thread; the value decreases as the number of MPI groups and OpenMP threads increases.

Table 11: Performance of MPI-OpenMP version of INS3D in SSP mode.

SSP /MPI	OMP Thrd	Time (s)	FP Ops /Load	Vec. Len.	Vec. %	GF /s	% of Peak
1/1	1	66.90	1.90	42.70	99.7	0.994	31.06
2/1	2	45.20	1.87	36.56	99.7	1.472	22.00
4/1	4	37.08	1.78	28.64	99.7	1.792	14.00
6/1	6	32.99	1.70	23.87	99.7	2.020	10.52
8/1	8	30.98	1.64	20.69	99.7	2.148	8.39
2/2	1	34.33	2.11	44.45	99.7	1.928	30.12
4/2	2	26.98	1.89	36.57	99.7	2.471	19.30
8/2	4	21.66	1.85	28.87	99.7	3.078	12.02
12/2	6	20.42	1.74	23.87	99.7	3.262	8.49
16/2	8	18.34	1.68	20.70	99.7	3.613	7.05
3/3	1	29.88	1.94	42.70	99.7	2.236	23.29
6/3	2	18.28	1.90	36.57	99.7	6.647	18.99
12/3	4	15.25	1.84	28.64	99.7	4.305	11.21
18/3	6	13.75	1.77	23.86	99.7	4.852	8.42
24/3	8	13.68	1.69	20.70	99.7	4.875	6.34
6/6	1	17.06	1.93	42.70	99.7	3.909	20.35
12/6	2	11.04	1.92	36.57	99.7	5.858	15.25
24/6	4	8.40	1.88	28.64	99.7	7.968	9.85

Table 12: Performance of MPI version of INS3D in MSP mode with compiler generated automatic streaming.

MSP /MPI	Time (s)	FP Ops /Load	Vec. Len.	Vec. %	GF /s	% of Peak
1/1	40.80	1.91	19.20	99.7	1.623	12.68
2/2	22.10	1.94	19.20	99.7	2.986	11.66
3/3	15.90	1.94	19.20	99.7	6.249	11.04
6/6	8.27	1.95	19.20	99.7	8.265	10.76

Table 12 displays the same runs performed in MSP mode but with OpenMP calls replaced by compiler generated automatic streaming. We observe similar behavior as in the SSP case with four OpenMP threads (compare with Table 11). This demonstrates that automatic streaming is effectively equivalent to having four OpenMP threads, which was the most efficient number to use in SSP mode. However, one difference is in the percentage of peak performance that is achieved by INS3D for this test case. In SSP mode, sustained performance when using four OpenMP threads decreases from 14% for one MPI group to 9.8% for six MPI groups. Instead, in MSP mode, this percentage only decreases from 12.7% to 10.8%. A bigger test case running on a larger X1 is needed to provide a better comparison and more insight.

5.4 GCEM3D

Finally, we evaluate the performance of the parallel GCEM3D (Goddard Cumulus Ensemble Model) application on the X1. GCEM3D is a cloud-resolving model that has been developed and improved at NASA Goddard Space Flight Center over the past two decades [18].

A detailed discussion of the simulations on which the current results are based can be found in [19]. Figure 10, taken from [19], shows 3D simulated cloud hydrometeor mixing ratios for an Atmospheric Radiation Measurement (ARM) case. The white isosurfaces denote the cloud water and cloud ice, blue denotes snow, green denotes rain water, and red denotes hail. Also shown are the simulated surface rainfall rate (mm/hr) corresponding to the same cloud fields. For the X1 evaluation, two parallel versions of the GCEM3D code, based on the MPI and OpenMP paradigms, were used. In both implementations, almost the same geophysical fluid dynamics models are solved, the only exception being the land model which has not yet been integrated into the OpenMP version.

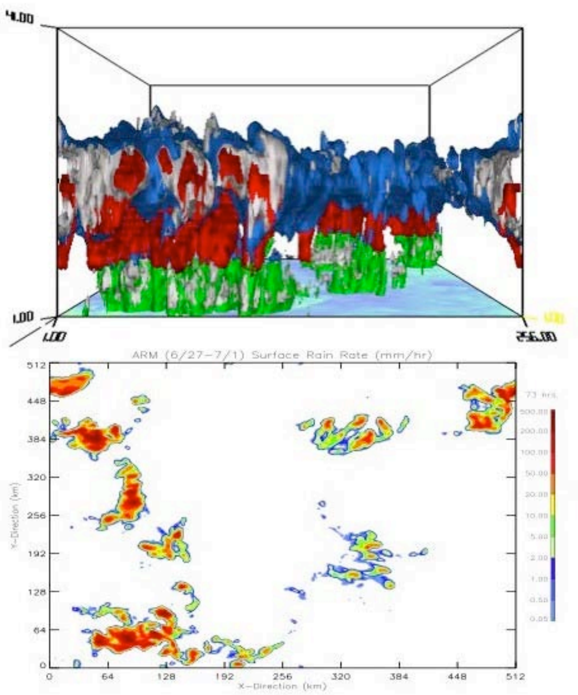

Figure 10: GCEM3D model-simulated cloud isosurfaces (top) and surface rainfall rate (bottom) (courtesy of Ref. [19]).

Table 13: Performance of MPI version of GCEM3D in MSP mode.

MSP	Time (s)	GF /s	Vec. Len.	FP Ops /Load	Mem (MB)
1	1772	1.096	27	2.0	448
2	901	2.125	27	2.0	624
4	675	2.835	15	1.9	976
8	342	5.600	15	1.9	912
12	389	5.048	8	1.7	848

Table 14: Performance of MPI version of GCEM3D in SSP mode.

SSP	Time (s)	GF /s	Vec. Len.	FP Ops /Load	Mem (MB)
1	2519	0.762	47	2.1	384
2	1401	1.371	46	2.1	496
4	920	2.084	45	2.0	720
8	523	3.675	45	2.0	1168
12	524	3.684	24	2.0	1552
16	413	4.674	24	2.0	2064
32	237	8.198	24	1.9	2080

The MPI version of GCEM3D is based on explicit message passing and uses an SPMD style programming model [11, 19]. The paradigm is that of coarse-grain parallelism using a domain decomposition strategy. The initial grid is divided into smaller subgrids in the longitude and latitude directions; one MPI task is then spawned for each subgrid. The OpenMP implementation [10], on the other hand, is based on a fine-grain parallelization strategy applied at the loop-level via compiler directives. In this approach, the initial grid is not partitioned.

Tables 13 and 14 show performance for the MPI version of GCEM3D on the X1 in MSP and SSP modes, respectively. The global 3D grid used for the MPI runs consisted of 104×104×42 grid points. The number of MSPs and SSPs listed in these tables is equal to the total number of MPI tasks, or equivalently, the number of subgrids. All computations were run for a total physical simulated time of 30 minutes.

Table 14 shows performance of the MPI version of GCEM3D when executed in SSP mode. The code scales nicely up to eight SSPs just as in the MSP case (the efficiency is about 60%). Note that the average vector length is reduced by a factor of almost two when the number of SSPs increases beyond eight. Overall, the vector lengths are higher in

Table 15: Performance of OpenMP version of GCEM3D in SSP modeX1.

SSP	Time (s)	GF /s	Vec. Len.	FP Ops /Load	Mem (MB)
1	5872	0.319	56	1.7	880
2	3102	0.604	55	1.7	896
4	1647	1.135	53	1.7	928
8	880	2.116	50	1.7	1008
16	475	3.906	44	1.7	1152

SSP mode due to multistreaming effects as discussed above. For an equivalent number of SSPs, the code always performs better in SSP mode. For instance, the total GFlops/s for four SSPs is twice that for one MSP. The memory usage again increases steadily with the number of SSPs, but at a lower rate for SSP counts beyond 16. The reason for the increased memory usage in SSP mode when using more than eight SSPs is not clear.

Table 15 presents performance results for the OpenMP version of GCEM3D. The global 3D grid used here consisted of 256×256×32 grid points. From total execution times reported in this table, it is evident that the OpenMP code scales almost linearly. The parallelization efficiency is 77% for 16 SSPs. However, sustained performance is relatively low, about half when compared with the MPI code (see Table 14). This may be due to the fact that the vector operations per load is 1.7 for the OpenMP version and 2.1 for the MPI code. Vectorization for the OpenMP code was about 92% for all runs, which was significantly less than the 99.7% achieved for the MPI code. Again, this could be due to the shorter length of the do-loops constructs imposed by the OpenMP directives. As expected, the memory usage increases slowly as SSP count increases due to OpenMP overhead.

6 Summary and Conclusions

The performance of the NAS Cray X1 was evaluated using a variety of microbenchmarks, kernel benchmarks, and CFD and Earth science application codes. The evaluation was somewhat limited due to time and machine size constraints; however, based on our experience, we can draw the following conclusions:

1. The X1 is an interesting architecture and relatively easy to program. The programming environment is user-friendly, the compilers are robust, and various performance tuning, optimizing, and monitoring tools are available.
2. The availability of two different modes, MSP and SSP, may seem confusing at first glance but provides additional flexibility in implementing and subsequently tuning applications to the architecture.
3. On vectorized codes it is relatively easy to achieve reasonable performance (about 25% of peak) in SSP mode. In MSP mode, our experience indicated that compiler generated automatic streaming was not as effective and this was reflected in the performance. The insertion of manual streaming directives could potentially improve application performance in MSP mode.
4. With the availability of the MLP subroutine library on the X1, it is now possible to have the same application codes running on all the high-end computers at NAS, the SGI Origin and Altix, as well as the X1, using the same API.
5. Co-array Fortran was easy to implement on both the NPB SP benchmark as well as the ROTOR CFD application code, and offered slightly improved performance relative to the MPI and/or MLP implementations.
6. OpenMP thread scaling on the X1 was good with reasonably linear speedups as the number of threads was increased to four.
7. Timing variations over multiple repeat runs were observed in the SSP mode for the MPI benchmark codes when the number of SSPs was not a multiple of 16. This phenomenon appears to be related to the X1 design.
8. Comparison of results between the X1 (in SSP mode) and SGI Altix indicates equivalent performance between the two architectures. For smaller grid sizes that can fit in cache, the Altix has a slight performance edge (about 10%), but for larger grid sizes the X1 performs better by roughly the same amount.

Acknowledgment

M. Jahed Djomehri was supported by NASA Ames Research Center under a contract with AMTI/CSC.

References

[1] D. Bailey, J. Barton, T. Lasinski, and H. Simon, 1991. The NAS Parallel Benchmarks. NAS Technical Report RNR-91-002, NASA Ames Research Center, Moffett Field, CA.

[2] D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, and M. Yarrow, 1995. The NAS Parallel Benchmarks 2.0. NAS Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA.

[3] C. Brady and D. Whittaker, 2005. Cray Inc. Personal communications.

[4] P. G. Buning, Personal communication.

[5] Cray Fortran Co-Array Programing Manual, 2004. No. S-3909-42, Cray, Inc.

[6] Co-Array Fortran. URL: *http://www.co-array.org*

[7] H. Feng, R. F. Van der Wijngaart, and R. Biswas, 2005. Unstructured Adaptive Meshes: Bad for Your Memory? Applied Numerical Mathematics, **52**, 153-173.

[8] M. Frumkin, H. Jin, and J. Yan, 1998. Implementation of NAS Parallel Benchmarks in High Performance Fortran. NAS Technical Report NAS-98-009, NASA Ames Research Center, Moffett Field, CA.

[9] H. Jin, J. Yan, and M. Frumkin, 1999. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. NAS Technical Report NAS-99-011, NASA Ames Research Center, Moffett Field, CA.

[10] H. Jin, G. Jost, D. Johnson, and W.-K. Tao, 2003. Experience on the Parallelization of a Cloud Modeling Code Using Computer-Aided Tools. NAS Technical Report NAS-03-006, NASA Ames Research Center, Moffett Field, CA.

[11] H. Juang, W.-K. Tao, X. Zeng, C.-L. Shie, S. Lang, and J. Simpson, 2005. Implementation of a Message Passing Interface into a Cloud-Resolving Model for Massively Parallel Computing. Mon. Wea. Rev., to be published.

[12] C. Kiris, D. Kwak, and S. Rogers, 2003. Incompressible Navier-Stokes Solvers in Primitive Variables and Their Applications to Steady and Unsteady Flow Simulations. Numerical Simulations of Incompressible Flows, (Hafez, M., Ed.), World Scientific.

[13] C. Kiris, D. Kwak, and W. Chan, 2000. Parallel Unsteady Turbopump Simulations for Liquid Rocket Engines. Supercomputing 2000.

[14] N. K. Madavan, et al. 1993. Multipassage Three-Dimensional Navier-Stokes Simulation of Turbine Rotor-Stator Interaction. AIAA J. Propulsion and Power, **9**, 389-396.

[15] NAS Parallel Benchmarks. URL: *http://www.nas.nasa.gov/Software/NPB.*

[16] M. M. Rai, 1989. Three-Dimensional Navier-Stokes Simulations of Turbine Rotor-Stator Interaction; Part I-Methodology; Part II-Results. AIAA J. Propulsion and Power, **5**, 307-319.

[17] J. R. Taft, 2000. Performance of the OVERFLOW-MLP CFD Code on the NASA/Ames 512-CPU Origin System. NAS Technical Report NAS-00-005, NASA Ames Research Center, Moffett Field, CA.

[18] W.-K. Tao, 2003. Goddard Cumulus Ensemble (GCE) Model: Application for Understanding Precipitation Processes. AMS Meteorological Mongraphs - Cloud Systems, Hurricanes and TRMM, pp.103-138.

[19] W.-K. Tao, 2003. Precipitation Processes During ARM (1997), TOGA COARE (1992), GATE (1974), SCSMEX (1998), and KWAJEX (1999): Consistent 2D and 3D Cloud Resolving Model Simulations. Thirteenth ARM Science Team Meeting Proceedings, Broomfield, CO.

[20] The HPC Challenge Benchmark. URL: *http://icl.cs.utk.edu/hpcc.*

[21] R. F. Van der Wijngaart and H. Jin, 2003. NAS Parallel Benchmarks, Multi-Zone Versions. NAS Technical Report NAS-03-010, NASA Ames Research Center, Moffett Field, CA.