Performance Evaluation of the SX6 Vector Architecture for Scientific Computations

Leonid Oliker, Andrew Canning, Jonathan Carter, John Shalf, David Skinner
CRD/NERSC, Lawrence Berkeley National Laboratory, Berkeley, CA 94720

Stephane Ethier
Princeton Plasma Physics Laboratory, Princeton University, Princeton, NJ 08453

Rupak Biswas, Jahed Djomehri*, and Rob Van der Wijngaart*
NAS Division, NASA Ames Research Center, Moffett Field, CA 94035

Abstract

The growing gap between sustained and peak performance for scientific applications has become a well-known problem in high performance computing. The recent development of parallel vector systems offers the potential to bridge this gap for a significant number of computational science codes and deliver a substantial increase in computing capabilities. This paper examines the intranode performance of the NEC SX6 vector processor and the cache-based IBM Power3/4 superscalar architectures across a number of key scientific computing areas. First, we present the performance of a microbenchmark suite that examines a full spectrum of low-level hardware characteristics. Next, we study the behavior of the NAS Parallel Benchmarks using a variety of optimization schemes. Finally, we evaluate the performance of several numerical codes from key scientific computing domains. Overall results demonstrate that the SX6 achieves high performance on a large fraction of our application suite and in many cases significantly outperforms the RISC-based architectures. However, certain classes of applications are not easily amenable to vectorization and would likely require extensive reengineering of both algorithm and implementation to utilize the SX6 effectively.

1 Introduction

The rapidly increasing peak performance and generality of superscalar cache-based microprocessors long led researchers to believe that vector architectures hold little promise for future large-scale computing systems. Due to their cost effectiveness, an ever-growing fraction of today's supercomputers employ commodity superscalar processors, arranged as systems of interconnected SMP nodes. However, the growing gap between sustained and peak performance for scientific applications on such platforms has become well-known in high performance computing.

The recent development of parallel vector systems offers the potential to bridge this gap for a significant number of scientific codes, and to increase computingpower substantially. This was highlighted dramatically when the Japanese Earth Simulator System's [2] results were published [18, 19, 22]. The Simulator, based on NEC SX6[1] *vector technology*, provides five times the LINPACK performance with half the number of processors of the IBM SP-based ASCI White, one of the world's most powerful supercomputers built, using *superscalar technology* [7]. More important than peak performance, however, is what this new capability entails for scientific communities that rely on modeling and simulation. It is therefore critical to evaluate these two microarchitectural approaches in the context of demanding computational algorithms.

We compare the performance of the NEC SX6 vector processor against the cache-based IBM Power3 and Power4 architectures for several key scientific computing areas. We begin by evaluating low-level system characteristics using microbenchmarks. Specifically, we measure memory bandwidth for various data access patterns, interprocessor MPI communication speeds, and OpenMP thread overheads. Next, we evaluate six of the well-known NAS Parallel Benchmarks (NPB) [13]. Finally, we present performance results for numerical codes from scientific computing domains, including astrophysics, fusion energy, materials science, fluid dynamics, and molecular dynamics. Our tests encompass a wide spectrum of algorithms, and multiple programming paradigms and parallelization strategies. Since

*Employee of Computer Sciences Corporation.
[1] Also referred to as the Cray SX6 due to Cray's agreement to market NEC's SX line.

most modern scientific codes are already tuned for cache-based systems, we examine the effort required to port these applications to the vector architecture. We focuse on serial and intranode performance of our application suite, while isolating processor and memory behavior. Future work will explore the behavior of multi-node vector configurations.

2 Architectural Specifications

We briefly describe the salient features of the three parallel architectures examined. Table 1 presents a summary of their intranode performance characteristics. Notice that the NEC SX6 has significantly higher peak performance, with a memory subsystem that features a data rate an order of magnitude higher than the IBM Power3/4 systems.

Node Type	CPU/ Node	Clock (MHz)	Peak (Gflops/s)	Memory BW (GB/s)	Peak Bytes/Flop	MPI Latency (usec)
Power3	16	375	1.5	0.7	0.45	8.6
Power4	32	1300	5.2	2.3	0.44	3.0
SX6	8	500	8.0	32	4.0	2.1

Table 1: Architectural specifications of the Power3, Power4, and SX6 nodes

2.1 Power3

The IBM Power3 was first introduced in 1998 as part of the RS/6000 series. Each 375 MHz processor contains two FPUs that can issue a multiply-add (MADD) per cycle for a peak performance of 1.5 GFlops/s. The Power3 has a short pipeline of only three cycles, resulting in relatively low penalty for mispredicted branches. The out-of-order architecture uses prefetching to reduce pipeline stalls due to cache misses. The CPU has a 32KB instruction cache and a 128KB 128-way set associative L1 data cache, as well as an 8MB four-way set associative L2 cache with its own private bus. Each SMP node consists of 16 processors connected to main memory via a crossbar. Multi-node configurations are networked via the IBM Colony switch using an omega-type topology.

The Power3 experiments reported in this paper were conducted on a single Nighthawk II node of the 208-node IBM pSeries system (named Seaborg) running AIX 5.1 and located at Lawrence Berkeley National Laboratory.

2.2 Power4

The pSeries 690 SMP node is the latest generation of IBM's RS/6000 series. Each 32-way SMP consists of 16 Power4 chips (organized as 4 MCMs). A chip contains two 1.3 GHz processor cores. Each core has two FPUs capable of a fused MADD per cycle, for a peak performance of 5.2 Gflops/s. Two load-store units, each capable of independent address generation, feed the two double precision MADDers. The superscalar out-of-order architecture can exploit instruction-level parallelism through its eight execution units. Up to eight instructions can be issued each cycle into a pipeline structure capable of simultaneously supporting more than 200 instructions. Advanced branch prediction hardware minimizes the effects of the relatively long pipeline (six cycles) necessitated by the high frequency design.

The Power4 storage hierarchy has three levels of cache, plus main memory. Each processor contains its own private L1 cache (64KB instruction and 32KB data) with prefetch hardware; however, core pairs share a 1.5MB unified L2 cache. Certain data access patterns may therefore cause L2 cache conflicts between the two processing units. The directory for the L3 cache is located on-chip, but the memory itself resides off-chip. The L3 is designed as a standalone 32MB cache, or to be combined with other L3s on the same MCM to create a larger interleaved cache of up to 128MB. Multi-node Power4 configurations are currently available employing IBM's Colony interconnect, but future large-scale systems will use the lower latency Federation switch.

The Power4 experiments reported here were performed on a single node of the 27-node IBM pSeries 690 system ("Cheetah") running AIX 5.1 and operated by Oak Ridge National Laboratory. The exception was the OVERFLOW-D code, which was run on a 64-processor system at NASA Ames Research Center.

2.3 SX6

The NEC SX6 vector processor uses a dramatically different architectural approach than more conventional cache-based systems. Vectorization exploits regularities in the computational structure to expedite uniform operations on independent data sets. Vector arithmetic instructions involve identical operations on the elements of vector operands located in the vector register file. Many scientific codes allow vectorization, since they are characterized by predictable fine-grain data-parallelism that can be exploited with properly structured program semantics and sophisticated compilers. The 500 MHz SX6 processor contains an 8-way replicated vector pipe capable of issuing a MADD each cycle, for a peak performance of 8 Gflops/s per CPU. The processors contain 72 vector registers, each holding 256 64-bit words, for a vector length of 256 elements. Thus, algorithms than can be structured to express data parallelism in blocks of 256 elements can most effectively utilize the SX6 vector infrastructure.

For non-vectorizable instructions, the SX6 contains a 500 MHz scalar processor with a 64KB instruction cache, a 64KB data cache, and 128 general-purpose registers. The 4-way superscalar unit has a theoretical peak of 1 Gflops/s and supports branch prediction, data prefetching, and out-of-order execution. Since the SX6 vector unit is significantly more powerful than the scalar unit, it is critical to achieve high vector operation ratios, either via compiler discovery, or explicitly through code (re-)organization.

Unlike most conventional architectures, the SX6 vector unit lacks data caches. Instead of relying on data locality to reduce memory overhead, memory latencies are masked by overlapping pipelined vector operations with memory fetches. The SX6 uses high speed DDR SDRAM, with peak bandwidth of 32GB/s per CPU: enough to feed one operand per cycle to each of the replicated pipe sets. Each SMP contains eight processors that share the node's memory. The nodes can be used as building blocks of large-scale multiprocessor systems. Currently the world's most powerful supercomputer, the Earth Simulator [2], contains 640 SX6 nodes, connected through a single-stage crossbar.

The vector results in this paper were obtained on a single-node (8-way) SX6 system ("Rime") running SUPER-UX at the Arctic Region Supercomputing Center (ARSC) of the University of Alaska.

3 Microbenchmarks

This section presents the performance of a microbenchmark suite that measures a full spectrum of low-level hardware characteristics, including memory subsystem behavior and scatter/gather hardware support (using STREAM [6]); point-to-point communication, network contention, and barrier synchronizations (using PMB [4]); and reduction operation and thread creation overhead (using EEPC [10]).

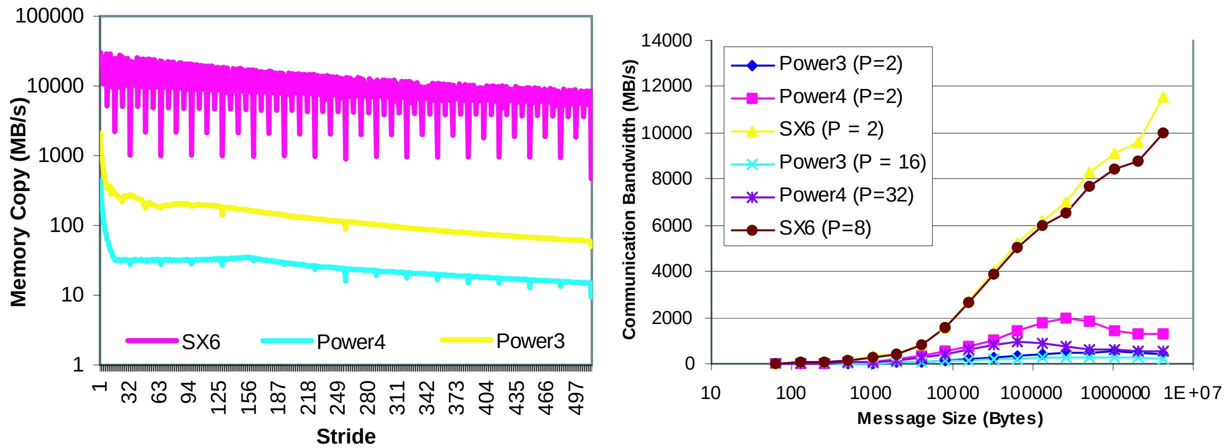
3.1 Memory Access Performance

First we examine the low-level memory characteristics of the three architectures in our study. Table 2 presents unit-stride memory bandwidth behavior of the triad summation using the well-known STREAM benchmark [6], represented as: $a(i) = b(i) + s \times c(i)$. Additionally, the table shows the percentage degradation in performance with increasing numbers of processors, isolating memory contention characteristics. The STREAM benchmark effectively captures the peak bandwidth of the architectures, and shows that the SX6 achieves about 14X and 48X the performance of the Power3 and Power4, respectively, on a single processor. Notice also that the SX6 shows negligible bandwidth degradation for up to eight tasks, while the Power3/4 degrade by almost 50% for fully packed nodes.

Our next experiment concerns the speed of strided data access. Figure 1(a) presents our results for a 64MB memory copy $a(i) = b(i)$ using various memory strides. Once again, the SX6 shows good bandwidth, up to two (three) orders of magnitude better than the Power4 (Power3), while showing markedly less average variation across the range of strides studied. Observe that certain strides impact SX6 memory bandwidth quite pronouncedly, by an order of magnitude or more. Analysis shows that strides containing factors of two worsen performance due to

	Power3		Power4		SX6	
P	MB/s	Degradation	MB/s	Degradation	MB/s	Degradation
1	661	0%	2292	0%	31900	0%
2	661	0%	2264	1.2%	31830	0.2%
4	644	2.6%	2151	6.2%	31875	0.1%
8	568	14.1%	1946	15.1%	31467	1.4%
16	381	42.4%	1552	32.3%		
32			1040	54.6%		

Table 2: STREAM triad performance (in MB/s) and bandwidth degradation



(a) (b)

Figure 1: Performance of (a) 64MB memory copy using various data strides and (b) MPI send/receive using various message sizes.

increased DRAM bank conflicts. On the Power3/4 a precipitous drop in data transfer rate occurs for small strides, due to loss of cache reuse. This drop is more complex on the Power4, due to its more complicated cache structure.

Finally, Table 3 presents the memory bandwidth of indirect addressing through a vector gather operation of various sizes: $a(i) = b(ib(i))$, where $ib(i)$ is a permutation of the indices of b. The unit stride performance is also shown for comparison. For smaller data sizes, the cache-based architectures show better data rates for indirect access to memory. However, for larger data gathers, the SX6 is able to utilize its hardware gather/scatter support effectively, outperforming the cache-based systems.

Data	Power3		Power4		SX6	
Size	Gather	Unit Stride	Gather	Unit Stride	Gather	Unit Stride
16KB	2546	3110	8182	9176	1088	1218
256KB	756	2060	3438	9944	7172	13128
32MB	364	438	2828	3460	7924	30060

Table 3: Memory bandwidth (in MB/s) for irregular and unit stride copy

3.2 MPI Performance

Message passing is the most wide-spread programming approach for high-performance parallel systems. The MPI library has become the de facto standard for message passing. It allows intra- and inter-node communications, thus obviating the need for hybrid programming schemes for distributed-memory systems. Although MPI increases code complexity compared with shared-memory programming paradigms such as OpenMP, its benefits lie in enhanced performance for coarse-grained communication, and implicit synchronization through blocking communication.

	8192 Bytes			131072 Bytes			524288 Bytes			2097152 Bytes		
P	Power3	Power4	SX6	Power3	Power4	SX6	Power3	Power4	SX6	Power3	Power4	SX6
2	143	515	1578	408	1760	6211	508	1863	8266	496	1317	9580
4	135	475	1653	381	1684	6232	442	1772	8190	501	1239	9521
8	132	473	1588	343	1626	5981	403	1638	7685	381	1123	8753
16	123	469		255	1474		276	1300		246	892	
32		441			868			592			565	

Table 4: MPI send/receive performance (in MB/s) for various message sizes and processor counts

Figure 1(b) and Table 4 present bandwidth figures obtained using the Pallas MPI Benchmark (PMB) suite [4], for exchnaging intranode messages of various sizes. The first set of three plots in Fig. 1(b) (the first row in Table 4) shows the best-case scenario, when only two processors within a node communicate. Notice that the SX6 has significantly better performance, achieving more than 27X (8X) the bandwidth of the Power3 (Power4), for the largest messages. The second set of plots shows the effects of network contention when all processors within each SMP are involved in exchanging messages. Once again, the SX6 substantially outperforms the Power3/4 architectures. For example, a message containing 524288 (2^{19}) bytes, suffers 46% (68%) bandwidth degradation when fully saturating the Power3 (Power4), but only 7% on the SX6 (see Table 4).

Table 5 shows the overhead of MPI barrier synchronization (in μsec). As expected, the barrier overhead on all three architectures increases with the number of processors. For the fully loaded SMP test case, the SX6 has 3.6X (1.9X) lower barrier cost than the Power3 (Power4); however, for the eight-processor test case, the SX6 performance degrades precipitously and is slightly exceeded by the Power4.

P	Power3	Power4	SX6
2	17.1	6.7	5.0
4	31.7	12.1	7.1
8	54.35	19.8	22.0
16	79.08	28.92	
32		42.38	

Table 5: MPI synchronization overhead (in μsec)

3.3 OpenMP Performance

Shared-memory parallel programming is often much simpler than message-passing, since each processor has global access to all memory. Shared-memory parallelism—with OpenMP the de facto standard–is generally achieved by inserting compilerdirectives into the code to distribute loop iterations among the processes. This section explores two sources of overhead associated with OpenMP constructs: thread spawning and reductions. The cost of these operations depends on the underlying hardware, as well as on the implementation of the OpenMP run-time library.

Table 6 presents the OpenMP overhead (in μsec) of the three architectures, based on the benchmark suite developed at EEPC [10]. For thread creation, the SX6 has the lowest overhead while suffering the least performance degradation with increasing numbers of processors. The scalar sum reduction experiment shows that the Power4 is fastest for up to eight processors. However, when using all the processors of the node, the SX6 once again outperforms the Power3 and Power4, by factors of 2.5 and 6.3, respectively.

	Thread Spawning			Scalar Reduction		
P	Power3	Power4	SX6	Power3	Power4	SX6
2	35.5	34.5	24.0	37.8	16.3	24.0
4	37.1	35.6	24.3	40.6	17.3	24.3
8	42.9	37.5	25.2	51.4	19.9	25.3
16	132.5	54.9		64.2	38.1	
32		175.5			158.3	

Table 6: OpenMP overheads (in μsec) for spawning threads and sum reduction

Our overall microbenchmarking results demonstrate that for low-level program constructs, the specialized SX6 vector hardware significantly outperforms the commodity-based superscalar designs of the Power3/4 architectures. Additionally, the SX6 consistently showed little degradation in performance with increasing numbers of processors. Unfortunately, the same was not generally true for the Power3/4, where the per-processor performance on fully loaded SMP nodes deteriorated significantly.

4 Scientific Kernels: NPB

The NAS Parallel Benchmarks (NPB) [13] provide a good middle ground for evaluating the performance of compact, well-understood applications. The NPB were created at a time when vector machines were considered no longer cost effective, and although their performance was meant to be good across a whole range of systems, they were written with cache-based systems in mind. Here we investigate the work involved in producing good NPB vector code (Power3/4 results will be included in the final paper). Of the eight published NPB, we select the six most appropriate for our current study: MG, a multi-grid kernel with trivial data dependencies but non-unit stride when exchanging solutions between different grids; CG, a sparse-matrix conjugate-gradient algorithm marked by irregular stride resulting from indirect addressing; FT, an FFT kernel; BT and SP, synthetic flow solvers that feature simple recurrence in a different array index in three different parts of the solution process; and LU, a synthetic flow solver that features a more complicated recurrence in multiple array indices simultaneously.

We present uni-processor performance results for these codes on the SX6 for three (increasing) problem sizes, commonly referred to as Classes A, B, and C, and three variations of vector performance tuning: default compiler flags (FLAG), compiler directives (DIR), actual code changes (MOD). The results of these exercises are presented in Table 7. Dittos (") indicate that performance remained essentially the same after additional optimization.

Tuning	Class	MG	CG	FT	SP	BT	LU
FLAG	A	2.00	0.329	0.411	1.71	0.345	0.481
	B	1.97	0.414	0.423	2.48	0.355	0.743
	C	2.52	0.490	0.398	3.10	0.350	0.741
DIR	A	"	"	"	"	1.92	"
	B	"	"	"	"	2.15	"
	C	"	"	"	"	2.37	"
MOD	A	"	"	1.47	"	2.70	"
	B	"	"	1.97	"	3.12	"
	C	"	"	1.94	"	4.29	"

Table 7: NAS Parallel Benchmarks SX6 uni-processor performance (in Gflops/s)

Evidently, the SP and MG codes vectorize fully without user intervention, despite the fact that some SP loop nests feature recursions in the innermost loop. These get recognized properly by the compiler, which applies a loop interchange. LU could have been improved if a major code restructuring had been attempted that would have turned a triple-loop recurrence into single-loop recurrence (*wavefront* method). Although the CG code fully vectorizes and exhibits fairly long vector lengths, performance is not very good due to many bank conflicts resulting from the indirect addressing. FT did not perform well in its original form, because the computations were blocked, with a fixed block length of 16 words. Adding compiler directives did not help. But once the code was modified to use a blocking length equal to the size of the grid (only three lines changed), performance improved markedly due to increased vector length.

BT is the most interesting case. The base code performed poorly because subroutines in inner loops inhibited vectorization. Also, some inner loops of small fixed length were vectorized, leading to very short vector length. Compiler flags enabling subroutine inlining and expansion of small loops led to complete vectorization, but did not improve performance. This was because the `expand` flag was ignored by the compiler in crucial parts of the code. Inserting an `expand` directive at the proper location did not help either. Expanding the small loops by hand, however, led to long vector lengths throughout the code, and good performance (more than 50% of peak for Class C).

5 Application Performance Metrics

Six applications from diverse areas in scientific computing were chosen to measure and compare the performance of the SX6 with that of the Power3 and Power4. The applications are: Cactus, an astrophysics code that solves Einstein's equations; TLBE, a fusion enery application that performs simulations of high-temperature plasma; PARATEC, a materials science code that solves Kohn-Sham equations to obtain electron wavefunctions; OVERFLOW-D, a CFD production code that solves the Navier-Stokes equations around complex aerospace configurations; GTC, a particle-in-cell approach to solve the gyrokinetic Vlasov-Poisson equations; and Mindy, a simplified molecular dynamics code that uses the Particle Mesh Ewald algorithm.

Performance results are reported in Mflops/s per processor, except where the original algorithm has been modified for the SX6 (these are reported as wall-clock time). Flops are obtained with the `hpmcount` tool on the Power3/4 and `ftrace` on the SX6. Additionally, to characterize vectorization behavior, we show the *average vector length* (AVL) and the *vector operation ratio* (VOR) of the SX6.

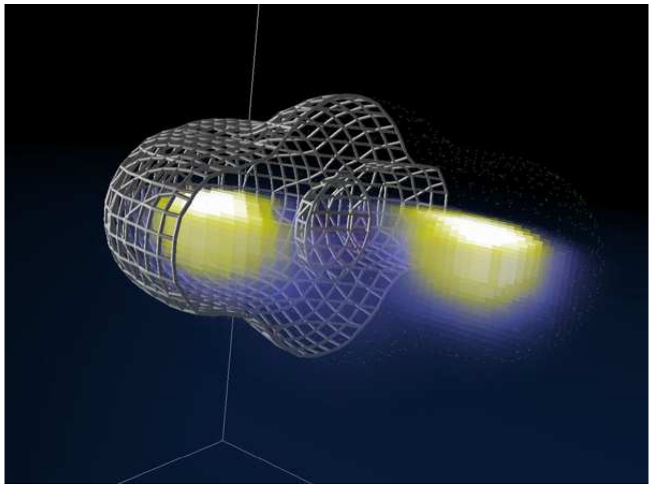


Figure 2: Visualization from a recent Cactus simulation of an in-spiraling merger of two black holes.

6 Astrophysics: Cactus

One of the most challenging problems in astrophysics is the numerical solution of Einstein's equations following from the Theory of General Relativity (GR): a set of coupled nonlinear hyperbolic and elliptic equations containing thousands of terms when fully expanded. The Albert Einstein Institute in Potsdam, Germany, developed the Cactus code [1] to evolve these equations stably in 3D on supercomputers to simulate astrophysical phenomena with high gravitational fluxes, such as the collision of two black holes (see Figure 2) and the gravitational waves that radiate from that event.

6.1 Methodology

The core of the Cactus solver uses the ADM formalism; also known as the 3+1 form. In GR, space and time form a 4D space (three spatial and one temporal dimension) that can be sliced along any dimension. For the purpose of solving Einstein's equations, the ADM solver decomposes the solution into 3D spatial hypersurfaces that represent different slices of space along the time dimension. In this formalism, the equations are written as four constraint equations and 12 evolution equations. The evolution equations can be solved using a number of different numerical methods including staggered leapfrog, McCormack, Lax-Wendroff, and iterative Crank-Nicholson schemes. A "lapse" function describes the time slicing between hypersurfaces for each step in the evolution. A "shift metric" is used to move the coordinate system at each step to avoid being drawn into a singularity. The four constraint equations are used to select different lapse functions and the related shift vectors.

For performance evaluation we focused on two different implementations of the core Cactus ADM solver. The first benchmark application uses the Fortran77-based ADM kernel (BenchADM [8]), written when vector machines were more common; consequently, we expect it to vectorize well. BenchADM is computationally intensive, involving 600 flops per grid point. The second Cactus ADM kernel (BenchBSSN [9]) is a newer solver, written using F90 syntax. BenchBSSN was developed specifically for microprocessor-based architectures; it has features such as more intense use of conditional statements in inner loops that may degrade performance on vector architectures. In both kernels, the loop body of the most numerically intensive part of the solver is large (several hundred lines of code). Splitting this loop provided little or no performance enhancement, as expected, due to little register pressure in the default implementation.

6.2 Porting Details

BenchADM vectorized almost entirely on the SX6 in the first attempt. However, the vectorization appears to involve only the innermost of a triply nested loop (x, y, and z-directions for a 3D evolution). The resulting effective vector length for the code is directly related to the x-extent of the computational grid.

The BenchBSSN kernel was more difficult to port. The size of the inner loop for two loop nests was too large for the flow analysis of the automatic vectorization system. Use of explicit vectorization directives was unsuccessful. Diagnostic compiler messages indicated incorrectly that some scalar temporaries caused an inter-loop dependency. However, the SX6 has no directive to inform the compiler that scalar variables are dependence-free. To circumvent this issue, we converted these temporary scalars into 1D vectors of length equal to the x-extent of the grid. This increased the memory footprint of the code considerably, but allowed the compiler to vectorize the code immediately.

6.3 Performance Results

Table 8 presents serial performance results for BenchADM and BenchBSSN in Mflops/s. For BenchADM, the standard benchmark grid of size 80×80×80 produced results that were only 25% of peak. Increasing the grid size to 128×128×128 doubled AVL and nearly doubled the performance of the benchmark, achieving 4.4 Gflops/s with a VOR of almost 100%. Various experiments with different optimization options, such as more radical expression reorganization (`-C hopt`) or inter-procedural analysis failed to deliver a noticeable boost in performance. To date, SX6's 55% of peak performance is the best achieved for this benchmark on any current computer architecture. In fact, the SX6 uni-processor performance was a remarkable 129X (14X) better than the Power3's (Power4's).

Code	Problem Size	Power3	Power4	SX6		
		Mflops/s	Mflops/s	Mflops/s	AVL	VOR
BenchADM	128×128×128	34	316	4400	127	99.7%
BenchBSSN	128×128×64	186	1168	2350	128	99.3%
	80×80×40	209	547	1765	80	99.0%
	40×40×20	249	722	852	40	98.4%

Table 8: Serial performance of the Cactus BenchADM and BenchBSSN kernels

The SX6 performance for BenchBSSN was considerably lower than for BenchADM, due to the extra level of indirection employed to force the code to vectorize. Additionally, the large number of conditional statements employed in the inner loop, although beneficial for microprocessor-based architectures, degrades vector performance. Notice the strong correlation between AVL and SX6 performance, showing the importance of making the vector lengths as close to the maximum as possible (256 words). Table 8 shows that for the 80×80×40 problem size, the SX6 outperformed the Power3 and Power4 by factors of 8.4 and 3.2, respectively.

7 Plasma Fusion: TLBE

Lattice Boltzmann methods provide a mesoscopic description of the transport properties of physical systems using a linearized Boltzmann equation. They offer an efficient way to model turbulence and collisions in a fluid. The TLBE application [20] performs a 2D simulation of high-temperature plasma using a hexagonal lattice and the BGK collision operator. Figure 3 shows an example of vorticity contours in the 2D decay of shear turbulence simulated by TLBE.

7.1 Methodology

The TLBE simulation has three computationally demanding components: computation of the mean macroscopic variables (integration); relaxation of the macroscopic variables after colliding (collision); and propagation of the macroscopic variables to neighboring grid points (stream). The first two steps are floating-point intensive, the third consists of data movement only. The problem is ideally suited for vector architectures. The first two steps are completely vectorizable, since the computation for each grid point is purely local. The third step consists of a set of strided copy operations. In addition, distributing the grid via a 2D decomposition easily parallelizes the method. The first two steps require no communication, while the third has a regular, static communication pattern in which the boundary values of the macroscopic variables are exchanged.

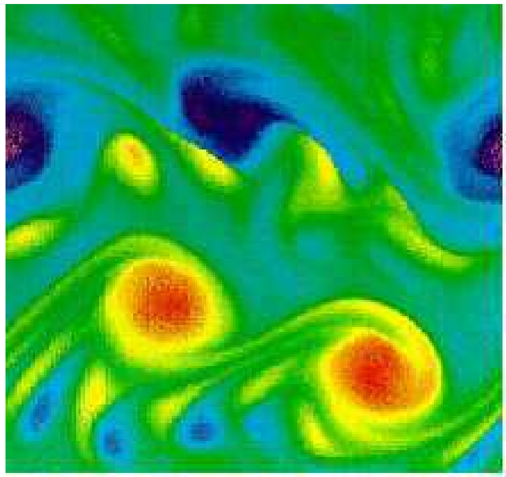


Figure 3: TLBE simulated vorticity contours in the 2D decay of shear turbulence.

7.2 Porting Details

After initial profiling on the SX6 using basic vectorization compiler options (`-C vopt`), a poor result of 280 Mflops/s was achieved for a small 64×64 grid using a serial version of the code. `ftrace` showed that VOR was high (95%) and that the collision step dominated the execution time (96% of total); however, AVL was only about 6. We found that the inner loop over the number of directions in the hexagonal lattice had been vectorized, but not a loop over one of the grid dimensions. Invoking the most aggressive compiler flag (`-C hopt`) did not help.

Therefore, we rewrote the collision routine by creating temporary vectors, and inverted the order of two loops to ensure vectorization over one dimension of the grid. As a result, serial performance improved by a factor of 7, and the parallel TLBE version was created by inserting the new collision routine into the MPI version of the code.

7.3 Performance Results

Parallel TLBE performance using a production grid of 2048×2048 is presented in Table 9. The SX6 results show that TLBE achieves almost perfect vectorization in terms of AVL and VOR. The 2- and 4-processor runs show similar performance as the serial version; however, an appreciable degradation is observed when running 8 MPI tasks, which is most likely due to network contention in the SMP.

	Power3	Power4	SX6		
P	Mflops/s	Mflops/s	Mflops/s	AVL	VOR
1	70	250	4060	256	99.5%
2	110	300	4060	256	99.5%
4	110	310	3920	256	99.5%
8	110	470	3050	255	99.2%
16	110	460			
32		440			

Table 9: Per-processor performance of TLBE on a 2048×2048 grid

For both the Power3 and Power4 architectures, the collision routine rewritten for the SX6 performed somewhat better than the original. In both the Power3/4, parallel TLBE showed higher Mflops/s (per CPU) compared with the serial version. This is due to the use of smaller grids per processor in the parallel case, resulting in improved cache reuse. The more complex behavior on the Power4 is due to the competitive effects of the three-level cache structure and saturation of the SMP memory bandwidth. In summary, using all 8 CPUs on the SX6 gives an aggregate performance of 24.4 Gflops/s (38% of peak), and a speedup of 27.7X and 6.5X over the Power3/4, with minimal porting overhead.

8 Materials Science: PARATEC

PARATEC (PARAllel Total Energy Code) [5] performs first-principles quantum mechanical total energy calculations using pseudopotential and a plane wave basis set. The approach is based on Density Functional Theory (DFT) that has become the standard technique in materials science to calculate accurately the structural and electronic properties of new materials with a full quantum mechanical treatment of the electrons. Codes performing DFT calculations are among the largest consumers of computer cycles in centers around the world, with the plane-wave pseudopotential approach being the most commonly used. Both experimental and theory groups use these types of codes to study properties such as strength, cohesion, growth, catalysis, magnetic, optical, and transport for materials like nanostructures, complex surfaces, doped semiconductors, and others.

8.1 Methodology

PARATEC uses an all-band conjugate gradient (CG) approach to solve the Kohn-Sham equations of DFT to obtain the wavefunctions of the electrons. Part of the calculations is carried out in real space and the remainder in Fourier space using specialized parallel 3D FFTs to transform the wavefunctions. The code spends most of its time (over 80% for a large system) in vendor supplied BLAS3 and 1D FFTs on which the 3D FFTs are built. For this reason, PARATEC generally obtains a high percentage of peak performance on different platforms. The code exploits fine-grained parallelism by dividing the plane wave components for each electron among the different processors. For a review of this approach with applications, see [14, 17].

8.2 Porting Details

PARATEC, an MPI code designed primarily for massively parallel systems, also runs on serial machines. Since much of the computation involves vendor supplied FFTs and BLAS3, an efficient vector implementation of the code requires these libraries to vectorize well. While the BLAS3 routines are well vectorized on the SX6, the standard FFTs (e.g., `ZFFT`) run at a low percentage of peak. It is thus necessary to use the simultaneous 1D FFTs (e.g., `ZFFTS`) to obtain good vectorization. A small amount of code rewriting was required to convert the 3D FFT routines to simultaneous ("multiple") 1D FFT calls.

8.3 Performance Results

The results in Table 10 show scaling tests of a 250 Si-atom bulk system for a standard LDA run of PARATEC with a 25 Ry cut-off using norm-conserving pseudopotential. The runs are for three CG steps of the iterative eigensolver, and include the set-up and I/O steps necessary to run the code.

	Power3	Power4	SX6		
P	Mflops/s	Mflops/s	Mflops/s	AVL	VOR
1	915	2290	5090	113	98%
2	915	2250	4980	112	98%
4	920	2210	4700	112	98%
8	911	2085	4220	112	98%
16	840	1572			
32		1327			

Table 10: Per-processor performance of PARATEC on a 250 Si-atom bulk system

Results show that PARATEC vectorizes well and achieves 64% of peak on one processor of the SX6. The AVL is approximately half the vector register length, but with a high fraction of VOR. This is because most of the time is spent in 3D FFTs and BLAS3. The loss in scalability to 8 processors (53% of peak) are due primarily to memory contention and initial code set-up (including I/O) that do not scale well. Performance increases with larger problem sizes and more CG steps: for example, running 432 Si-atom systems for 20 CG steps achieved 73% of peak on one processor.

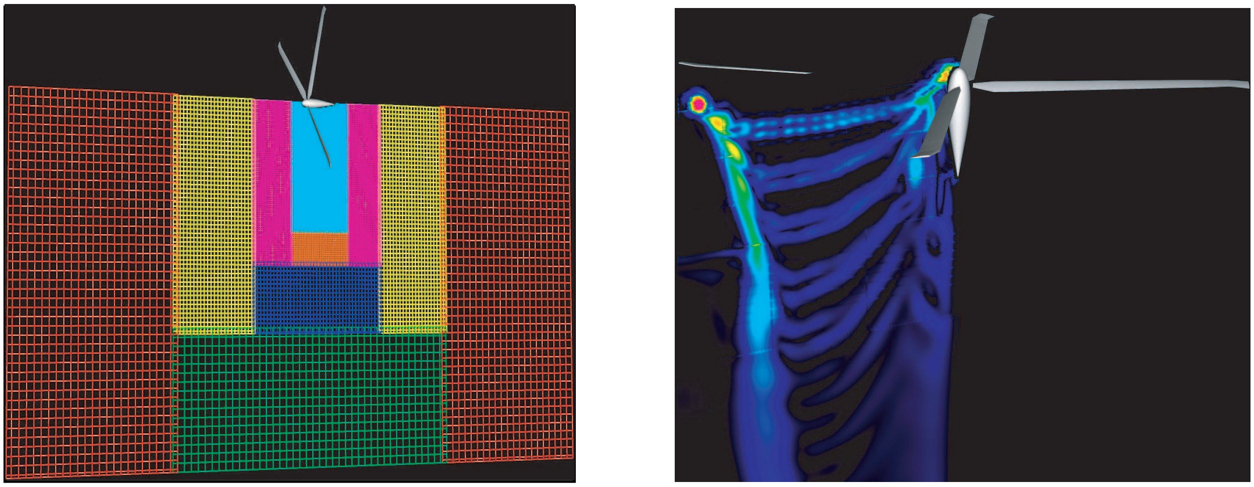


Figure 4: Sectional views of the OVERFLOW-D test grid system and the computed vorticity magnitude contours.

PARATEC runs efficiently on the Power3; the FFT and BLAS3 routines are highly optimized for this architecture. The code ran at 61% of peak on a single processor and an estimable 56% on 16 processors. Larger physical systems—432 Si atoms—ran at 1.02 Gflops/s (68% of peak) on 16 processors. On the Power4, PARATEC runs at a much lower fraction of peak (44% on one processor) due to its relatively poor ratio of memory bandwidth to peak performance. Nonetheless, the Power4 32-processor SMP node achieves high total performance, exceeding that of the 8-processor SX6 node. We conclude that, due to the high computational intensity and use of optimized numerical libraries, these types of codes run efficiently on both scalar and vector machines, without the need for significant code restructuring.

9 Fluid Dynamics: OVERFLOW-D

OVERFLOW-D [21] is an overset grid methodology [11] for viscous flow simulations around aerospace configurations. The application can handle complex designs with multiple geometric components, where individual body-fitted grids are easily constructed about each component. OVERFLOW-D is designed to simplify the modeling of components in relative motion (dynamic grid systems). At each time step, the flow equations are solved independently on each grid ("block") in a sequential manner. Boundary values in grid overlap regions are updated before each time step, using a Chimera interpolation procedure. The code uses finite differences in space, and implicit/explicit time stepping.

9.1 Methodology

The MPI version of OVERFLOW-D (in F90) is based on the multi-block feature of the sequential code, which offers natural coarse-grain parallelism. The sequential code consists of an outer "time-loop" and an inner "grid-loop". The inter-grid boundary updates in the serial version are performed successively. To facilitate parallel execution, grids are clustered into groups; one MPI process is then assigned to each group. The grid-loop in the parallel implementation contains two levels, a loop over groups ("group-loop") and a loop over the grids within each group. The group-loop is performed in parallel, with each group performing its own sequential grid-loop and inter-grid updates. The inter-grid boundary updates across the groups are achieved via MPI.

The hybrid paradigm [12] exploits a second level of parallelism using OpenMP, where explicit compiler directives are inserted at the loop level. The time-loop in the hybrid approach is similar to the pure MPI case. Only the grid-loop, which contains the computationally intensive part of the code, is multithreaded by OpenMP. Currently, the same number of threads are spawned per MPI process; the total number of processors used in the hybrid code is the product of the number of spawned OpenMP threads and the number of MPI processes.

9.2 Porting Details

Both the MPI and hybrid implementations of OVERFLOW-D are based on the sequential version, the organization of which was designed to exploit vector machines. The same basic code structure is used on both the Power4 and the SX6, except for the LU-SGS linear solver that required signifi cant modifi cations to enhance effi ciency. On the Power4, a pipeline [12] strategy was implemented, while on the SX6, a hyper-plane algorithm was used. These changes were dictated by the data dependencies inherited by the solution process, and take advantage of the cache and vector architecture, respectively. A few other minor changes were also made in some subroutines in an effort to meet the specifi c MPI/OpenMP compiler requirements.

9.3 Performance Results

Our experiments involve a Navier-Stokes simulation of vortex dynamics in the complex wake flow region around hovering rotors. The grid system consisted of 41 blocks and 8 million grid points. Figure 4 presents a sectional view of the test application grid and the computed vorticity magnitute contours of the fi nal solution. Table 11 shows execution times (averaged over 10 time steps) on the Power4 and SX6; Power3 results are currently unavailable. The hybrid runs with one OpenMP thread and the pure MPI runs with the same number of processes are conceptually the same; however, due to procedural differences, they have different timings.

P	MPI Tasks	OpenMP Threads	Paradigm	Power4	SX6		
				sec	sec	AVL	VOR
2	2	—	MPI	15.8	5.5	87	80%
2	2	1	Hybrid	18.2	5.6	84	77%
4	4	—	MPI	8.5	2.8	84	76%
4	4	1	Hybrid	10.1	2.8	83	71%
4	2	2	Hybrid	10.5	3.6	—	—
8	8	—	MPI	4.3	1.6	79	69%
8	8	1	Hybrid	6.0	1.6	76	62%
8	2	4	Hybrid	5.9	2.5	—	—
8	4	2	Hybrid	6.2	1.8	—	—
16	16	—	MPI	3.7			
16	16	1	Hybrid	4.5			
16	4	4	Hybrid	3.7			
32	32	—	MPI	3.4			
32	32	1	Hybrid	2.8			
32	8	4	Hybrid	2.7			

Table 11: MPI and hybrid performance of OVERFLOW-D on a 8 million-grid point problem

Results show that the SX6 outperforms the Power4 for both the message-passing and hybrid paradigms. Run times for 8 processors on the SX6 are roughly comparable to the 32-processor Power4 numbers. Scalability is similar for both architectures, with computational effi ciency decreasing for a larger number of MPI tasks, due to load imbalance. On the SX6, the relatively small AVL and limited VOR explain why the code achieves a maximum of only 1.9 Gflops/s on 8 processors. Reorganizing OVERFLOW-D would achieve higher vector performance; however, extensive effort would be required to modify this production code.

In general, the hybrid implementation requires more programmer effort but performs only negligibly better than the MPI version in some cases. Similarly, runs with larger numbers of threads appear to be less effi cient for the same total number of processors. However, the hybrid paradigm for overset grid applications is appropriate when the total number of processors is close to or larger than the number of blocks, when load balancing is diffi cult [12].

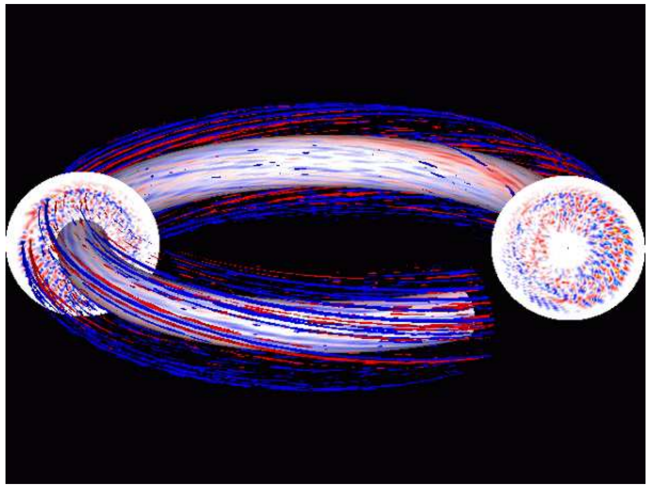


Figure 5: Electrostatic potential fluctuations of microturbulence in magnetically confined plasmas using GTC.

10 Magnetic Fusion: GTC

The goal of magnetic fusion is the construction and operation of a burning plasma power plant producing clean energy. The performance of such a device is determined by the rate at which the energy is transported out of the hot core to the colder edge of the plasma. The Gyrokinetic Toroidal Code (GTC) [16] was developed to study the dominant mechanism for this transport of thermal energy, namely plasma microturbulence. Plasma turbulence is best simulated by particle codes, in which all the nonlinearities are naturally included. Figure 5 presents a visualization of electrostatic potential fluctuations in a global nonlinear gyrokinetic simulation of microturbulence in magnetically confined plasmas.

10.1 Methodology

GTC solves the gyroaveraged Vlasov-Poisson (gyrokinetic) system of equations [15]) using the particle-in-cell approach. Instead of interacting with each other, the simulated particles interact with a self-consistent electrostatic or electromagnetic field described on a grid. Numerically, the PIC method scales as N, instead of N^2 as in the case of direct binary interactions. Also, the equations of motion for the particles are simple ODEs (rather than nonlinear PDEs), and can be solved easily (e.g. Runge-Kutta). The main tasks at each time step are: deposit the charge of each particle at the nearest grid points (scatter); solve the Poisson equation to get the potential at each grid point; calculate the force acting on each particle from the potential at the nearest grid points (gather); move the particles by solving the equations of motion; find the particles that have moved outside their local domain and migrate them accordingly.

The parallel version of GTC performs well on massive superscalar systems, since the Poisson equation is solved as a local operation. The key performance bottleneck is the scatter operation, a loop over the array containing the position of each particle. Based on a particle's position, we find the nearest grid points surrounding it and assign each of them a fraction of its charge proportional to the separation distance. These charge fractions are then accumulated in another array. The scatter algorithm in GTC is complicated by the fact that these are fast gyrating particles where motion is described by charged rings being tracked by their guiding center (the center of the circular motion).

10.2 Porting Details

GTC's scatter phase presented some challenges when porting the code to the SX6 architecture. It is difficult to implement efficiently due to its non-sequential writes to memory. The particle array is accessed sequentially, but its entries correspond to random locations in the simulation space. As a result, the grid array accumulating the charges is accessed in random fashion, resulting in poor cache performance. This problem is exacerbated on vector architectures, since many particles deposit charges at the same grid point, causing a classic memory dependence problem and preventing vectorization. We avoid these memory conflicts by using temporary arrays of vector length (256 words) to accumulate the charges. Once the loop is completed, the information in the temporary array is merged with the real charge data; however, this increases memory traffic and reduces the flop/byte ratio.

Another source of performance degradation was a short inner loop located inside two large particle loops the SX6 compiler could not vectorize. This problem was solved by inserting a vectorization directive, fusing the inner and outer loops. Finally, I/O within the main loop had to be removed in order to allow vectorization.

10.3 Performance Results

Table 12 shows the results of single-processor GTC runs. Only the serial version has been vectorized at this time; however, recent results demonstrate that GTC scales well on massively parallel architectures. The simulation in this study comprises 4 million particles and 301,472 grid points. The geometry is a torus described by the configuration of the magnetic field. The Power3 sustains 174 Mflops/s (12% of peak), while the 304 Mflops/s achieved on the Power4 represents only 6% of its peak performance. The SX6 experiment runs at 716 Mflops/s, or only 9% of it theoretical peak. This poor performance is unexpected, considering the relatively high AVL (180) and VOR (97%). We believe this is because the scalar units need to compute the indices for the scatter/gather of the underlying unstructured grid. However, the SX6 still outperforms the Power3/4 by factors of 2.7 and 5.3, respectively.

Power3	Power4	SX6		
Mflops/s	Mflops/s	Mflops/s	AVL	VOR
174	304	716	180	97%

Table 12: Serial performance of GTC on a 4-million-particle simulation

11 Molecular Dynamics: Mindy

Mindy is a simplified serial molecular dynamics (MD) C++ code, derived from the parallel MD program "NAMD" [3]. The energetics, time integration, and file formats are identical to those used by NAMD.

11.1 Methodology

Mindy's core is the calculation of forces between N atoms via the Particle Mesh Ewald (PME) algorithm. Its $O(N^2)$ complexity is reduced to $O(N \log N)$ by dividing the problem into boxes, and then computing electrostatic interaction in aggregate by considering neighboring boxes. Neighbor lists and a variety of cutoffs are used to decrease the required number of force computations.

11.2 Porting Details

Modern MD codes such as Mindy present special challenges for vectorization, since many optimization and scaling methodologies are at odds with the flow of data suitable for vector architectures. The reduction of floating point work from N^2 to $N \log N$ is accomplished at the cost of increased branch complexity nonuniform data access. These techniques have a deleterious effect on vectorization; two strategies were therefore adopted to optimize Mindy on the SX6. The first severely decreased the number of conditions and exclusions in the inner loops, resulting in more computation overall, but less inner-loop branching. We refer to this strategy as NO_EXCL.

The second approach was to divide the electrostatic computation into two steps. First, the neighbor lists and distances are checked for exclusions, and a temporary list of inter-atom forces to be computed is generated. The force computations are then performed on this list in a vectorizable loop. Extra memory is required for the temporaries and, as a result, the flop/byte ratio is reduced. This scheme is labeled BUILD_TEMP.

Mindy uses C++ objects extensively, hindering the compiler to identify data-parallel code segments. Aggregate datatypes call member functions in the force computation, which impede vectorization. Compiler directives were used to specify that certain code sections contain no dependencies, allowing partial vectorization of those regions.

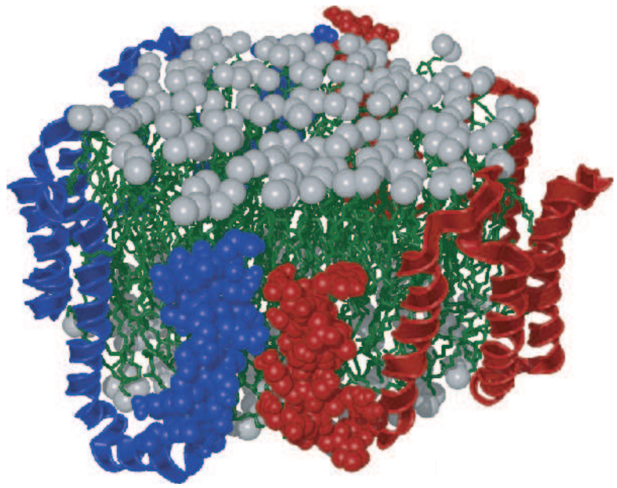


Figure 6: The apolipoprotein A-I molecule, a 92224-atom system simulated by Mindy.

11.3 Performance Results

The case studied here is the apolipoprotein A-I (see Figure 6), a 92224-atom system important in cardiac blood chemistry that has been adopted as a benchmark for large scale MD simulations on biological systems. Table 13 presents performance results of the serial Mindy algorithm. Neither of the two SX6 optimization strategies achieves high performance. The NO_EXCL approach results in a very small VOR, meaning that almost all the computations are performed on the scalar unit. The BUILD_TEMP approach increases VOR, but incurs the overhead of increased memory traffic for storing temporary arrays. In general, this class of applications is at odds with vectorization due to the irregularly structured nature of the codes. The SX6 achieves only 165 Mflops/s, or 2% of peak, slightly outperforming the Power3 and trailing the Power4 by about a factor of two in runtime. Effectively utilizing the SX6 would likely require extensive reengineering of both the algorithm and the code.

Power3	Power4	SX6: `NO_EXCL`			SX6: `BUILD_TEMP`		
sec	sec	sec	AVL	VOR	sec	AVL	VOR
15.7	7.8	19.7	78	0.03%	16.1	134	34.8%

Table 13: Serial performance of Mindy on a 92224-atom system with two different SX6 optimization approaches

12 Summary and Conclusions

This paper presented the performance of the NEC SX6 vector processor and compared it against the cache-based IBM Power3/4 superscalar architecture, across a wide spectrum of scientific computations. Results with a set of microbenchmarks demonstrated that for low-level program characteristics, the specialized SX6 vector hardware significantly outperforms the commodity-based superscalar designs of the Power3 and Power4. Next we examined the NPBs, a well-understood set of kernels representing key areas in scientific computations. These compact codes allowed us to perform the three main variations of vectorization tuning: compiler flags, compiler directives, and actual code modifications. Results enabled us to identify classes of applications both at odds with and well suited for vector architectures, with performance ranging from 5% to 50% of peak.

Several applications from key scientific computing domains were then evaluated. Table 14 summarizes the relative performance results. Since most modern scientific codes are designed for (super)scalar systems, we examined the effort required to port these applications to the vector architecture. Results show that the SX6 achieves high performance for a large fraction of our application suite and in many cases significantly outperforms the scalar architectures. The computationally intensive Cactus BenchADM code showed the best vector performance, achieving a 129X (14X) improvement over the Power3 (Power4), while only requiring recompilation on the SX6.

Name	Discipline	Lines of Code	Power3 % Pk	Power4 %Pk	SX6 % Pk	P	SX6 Speedup vs. Power3	SX6 Speedup vs. Power4
Cactus-ADM	Astrophysics	1200	2.3	6.1	55.0	1	129	13.9
Cactus-BSSN	Astrophysics	8200	13.9	10.5	22.1	1	8.4	3.2
TLBE	Plasma Fusion	1500	7.3	9.0	38.1	8	27.8	6.5
PARATEC	Materials Science	50000	60.7	40.1	52.8	8	4.6	2.0
OVERFLOW-D	Fluid Dynamics	100000	—	10.1	24.3	8	—	3.7
GTC	Magnetic Fusion	5000	11.6	5.9	9.0	1	5.3	2.7
Mindy	Molecular Dynamics	11900	6.3	4.7	2.1	1	1.0	0.5

Table 14: Summary overview of application suite performance

The rest of our applications required the insertion of compiler directives and/or minor code modifications to improve the two critical components of effective vectorization: long vector length and high vector operation ratio. Vector optimization strategies included loop fusion (and loop reordering) to improve vector length; introduction of temporary variables to break loop dependencies (both real and compiler imagined); reduction of conditional branches; and alternative algorithmic approaches. For codes such as Cactus BenchBSSN and TLBE, minor code changes were sufficient to achieve good vector performance and a high percentage of theoretical peak. PARATEC represented a class of applications relying heavily on highly optimized BLAS3 libraries. For these types of codes, all three architectures performed very well due to the regularly structured, computationally intensive nature of the algorithm. For OVERFLOW-D, we compared hybrid programming with a pure message-passing implementation, and showed only a marginal performance improvement over the MPI version at the cost of higher programming complexity.

Finally, we presented two applications with poor vector performance: GTC and Mindy. They feature indirect addressing, many conditional branches, and loop carried data-dependencies, making high vector performance challenging. This was especially true for Mindy, whose use of C++ objects made it difficult for the compiler to identify data-parallel loops. Effectively utilizing the SX6 would likely require extensive reengineering of both the algorithm and the implementation for these applications.

Acknowledgements

All authors from Lawrence Berkeley National Laboratory were supported by Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC03-76SF00098. The Computer Sciences Corporation employees were supported by NASA Ames Research Center under contract number DTTS59-99-D-00437/A61812D with AMTi/CSC.

References

[1] Cactus Code Server. http://www.cactuscode.org.

[2] Earth Simulator Center. http://www.es.jamstec.go.jp.

[3] Mindy — A 'minimal' molecular dynamics program. http://www.ks.uiuc.edu/Development/MDTools/mindy.

[4] Pallas MPI Benchmarks. http://www.pallas.com/e/products/pmb.

[5] PARAllel Total Energy Code. http://www.nersc.gov/projects/paratec.

[6] STREAM: Sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream.

[7] Top500 Supercomputer Sites. http://www.top500.org.

[8] A. Abrahams, D. Bernstein, D. Hobill, E. Seidel, and L. Smarr. Numerically generated black hole spacetimes: Interaction with gravitational waves. *Phys. Rev. D*, 45:3544–3558, 1992.

[9] M. Alcubierre, G. Allen, B. Brügmann, E. Seidel, and W.-M. Suen. Towards an understanding of the stability properties of the 3+1 evolution equations in general relativity. *Phys. Rev. D*, 62:124011, 2000.

[10] J.M. Bull. Measuring synchronisation and scheduling overheads in OpenMP. In *Proc. 1st European Workshop on OpenMP*, pages 99–105, 1999.

[11] P.G. Buning, D.C. Jespersen, T.H. Pulliam, W.M. Chan, J.P. Slotnick, S.E. Krist, and K.J. Renze. *Overflow User's Manual, Version 1.8g*. NASA Langley Research Center, 1999.

[12] M.J. Djomehri and H. Jin. Hybrid MPI+OpenMP programming of an overset CFD solver and performance investigations. Technical Report NAS-02-002, NASA Ames Research Center, 2002.

[13] D.H. Bailey et al. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, 1994.

[14] G. Galli and A. Pasquarello. *First-Principles Molecular Dynamics*, pages 261–313. Computer Simulation in Chemical Physics. Kluwer, 1993.

[15] W.W. Lee. Gyrokinetic particle simulation model. *J. Comp. Phys.*, 72:243–262, 1987.

[16] Z. Lin, S. Ethier, T.S. Hahm, and W.M. Tang. Size scaling of turbulent transport in magnetically confined plasmas. *Phys. Rev. Lett.*, 88, 2002. 195004.

[17] M.C. Payne, M.P. Teter, D.C. Allan, T.A. Arias, and J.D. Joannopoulos. Iterative minimization techniques for ab initio total-energy calculations: Molecular dynamics and conjugate gradients. *Rev. Mod. Phys.*, 64:1045–1098, 1993.

[18] H. Sakagami, H. Murai, Y. Seo, and M. Yokokawa. 14.9 TFLOPS three-dimensional fluid simulation for fusion science with HPF on the Earth Simulator. In *Proc. SC2002*, 2002.

[19] S. Shingu, H. Takahara, H. Fuchigami, M. Yamada, Y. Tsuda, W. Ohfuchi, Y. Sasaki, K. Kobayashi, T. Hagiwara, S. Habata, M. Yokokawa, H. Itoh, and K. Otsuka. A 26.58 Tflops global atmospheric simulation with the spectral transform method on the Earth Simulator. In *Proc. SC2002*, 2002.

[20] G. Vahala, J. Carter, D. Wah, L. Vahala, and P. Pavlo. *Parallelization and MPI performance of Thermal Lattice Boltzmann codes for fluid turbulence*. Parallel Computational Fluid Dynamics '99. Elsevier, 2000.

[21] A.M. Wissink and R. Meakin. Computational fluid dynamics with adaptive overset grids on parallel and distributed computer platforms. In *Proc. Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, pages 1628–1634, 1998.

[22] M. Yokokawa, K. Itakura, A. Uno, T. Ishihara, and Y. Kaneda. 16.4-Tflops Direct Numerical Simulation of turbulence by Fourier spectral method on the Earth Simulator. In *Proc. SC2002*, 2002.