# Computational Fluid Dynamics Applications on the Cray X1 Architecture: Experiences, Algorithms, and Performance Analysis

Andrew A. Johnson
Army HPC Research Center / Network Computing Services, Inc.
Minneapolis, Minnesota
ajohn@ahpcrc.org

April 21, 2003

## ABSTRACT

We present our experiences and performance results of our in-house computational fluid dynamics (CFD) codes on the new Cray X1 parallel/vector/multi-streaming architecture. These codes which solve the time-accurate incompressible Navier-Stokes equations are fully implicit (i.e. a coupled equation system is solved), finite element based, and are built for fully unstructured meshes. The codes are fully parallel based on MPI, incorporating mesh partitioning strategies, and include a GMRES-based iterative equation solver for both matrix-free and sparse-matrix operational modes. Throughout the development of these CFD codes at the Army HPC Research Center, vectorization has never been applied, so new vectorization strategies and additional algorithms that were required to achieve optimal vector and multi-streaming performance on the Cray X1 will be discussed.

We also present a detailed analysis of the performance of this CFD code on the Cray X1 including comparisons with other parallel architectures such as the Cray T3E-1200, as well as raw Giga-Flop rates of various parts of the code. Various factors that may effect the performance of the code on the X1 will be identified. Parallel scalability of the code, as well as inter-processor communication performance, will also be presented.

## INTRODUCTION

The Army High Performance Computing (HPC) Research Center (AHPCRC) is the first non-classified site to take shipment of a Cray X1 system. The Cray X1 is a new computing architecture built to deliver high sustained computational performance for a variety of important numerical simulation and computational modeling applications. The AHPCRC's infrastructure support and system integration contractor NetworkCS, Inc. took shipment of two early-production (EP) air-cooled (AC) systems in September, 2002. Each of these systems contains 16 multi-streaming processors (MSP). A production liquid-cooled (LC) system was installed in February, 2003 (see Figure 1).

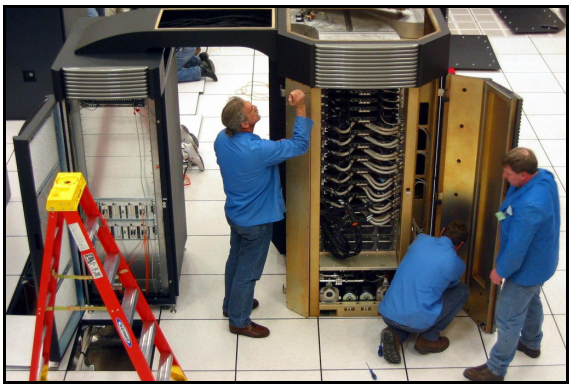


**Figure 1. Cray technicians installing a portion of the AHPCRC's liquid-cooled X1 system at NetworkCS, Inc. (Minneapolis, MN) on February 20, 2003.**

This initial system contains a half populated LC cabinet containing a total of 32 processors. This existing cabinet will be fully populated (i.e. expanded to 64 processors), as well as the addition of another fully-populated LC cabinet, in stages throughout the first half of 2003 to ultimately create an X1 system with a total of 128 processors. The systems are owned by the U.S. Army, but acquired, maintained and operated by NetworkCS, Inc. for use by AHPCRC and DOD researchers (see Figure 2). Both the two AC EP systems, as well as this initial LC system, have passed acceptance tests.

**Figure 2. The AHPCRC's liquid-cooled Cray X1 after instillation at NetworkCS, Inc.**

This new Cray X1 system is the AHPCRC's third large-scale HPC architecture. The center's first HPC system was an 896 processor Thinking Machines CM-5 (Serial Number 1) that was delivered in 1991 and was retired in 1998. The center's second large HPC system is an 1,088 processor Cray T3E-1200 which is still in operation and heavily used by AHPCRC and DOD researchers. This system was installed in 1998. It is expected that the new Cray X1 systems will also be heavily used and augment the numerical simulation and computational modeling capabilities of AHPCRC, Army, and DOD researchers.

The AHPCRC and U.S. Army are targeting research on critical defense applications for the center's Cray X1 such as computational weather modeling and forecasting using applications such as MM5 (see [1]), computational fluid dynamics (CFD) research to be used, in part, to study contaminant dispersion within urban environments [2] as shown in Figure 3, computational solid mechanics simulations such as the prediction of projectile/armor interactions [3,4], as well as computational chemistry and some other areas such as electromagnetics. Good performance for these targeted applications on the Cray X1 will allow AHPCRC and Army researchers to perform larger, more detailed, and more accurate numerical simulations in shorter periods of time that what is currently possible on existing systems. Work is already underway on the AHPCRC's X1 in most of these areas, and the specific topic of computational fluid dynamics performance on the X1 is the focus of this paper.

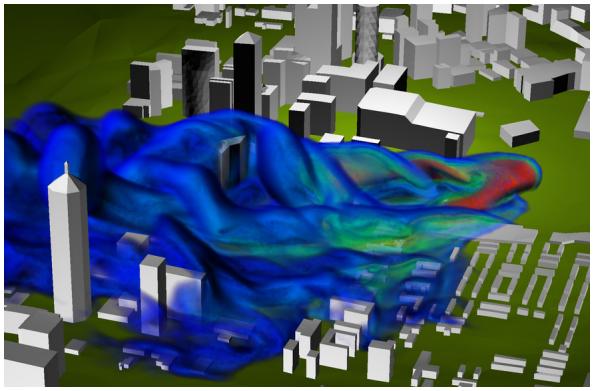


**Figure 3. Numerical simulation of contaminant dispersion in Atlanta, GA. Shown is a volume rendering of contaminant concentration. Simulation performed by S. Aliabadi (AHPCRC-Clark Atlanta University), and visualized by A. Johnson (AHPCRC-NetworkCS, Inc.).**

The porting and performance enhancement work on the CFD codes discussed here will directly apply to similar CFD codes used by our AHPCRC-Clark Atlanta University partners, the US Army Research Laboratory (ARL), the US Army Engineering-Research and Development Center (ERDC), and the US Army Natick Research and Development Engineering Center (Natick RDEC). These CFD codes are also available to researchers at the US Army Military Academy at West Point.

In the next section, details about the exact CFD code being ported-to and tested-on the Cray X1 will be given, followed by an overview of the X1 architecture with a description of porting experience and required code modifications to achieve full vectorization / multi-streaming. Following this are benchmark results including both raw scalar (processor) performance, and multi-processor scalability. We conclude with some final observations about our early Cray X1 experiences.

## UNSTRUCTURED CFD CODE OVERVIEW

Several in-house computational fluid dynamics codes have been developed at the AHPCRC throughout its 13 year history. They all share commonality in the fact that they are finite element based, built for unstructured meshes, fully stabilized using SUPG and PSPG methods [5,6], time accurate, solve the resulting coupled equation system with a GMRES-based iterative solver [7], and are fully parallel based on MPI by incorporating mesh partitioning techniques [8,9]. Some of these details will be explained further in the following paragraphs. Initially, these CFD codes have been developed within the data-parallel programming model on the Thinking Machines CM-5 [10], but have been subsequently ported to the message-passing model based on MPI which is portable to almost all HPC architectures such as the Cray T3E and X1. It is important that these codes run as fast and efficiently as possible because CFD applications such as these take up a significant percentage of time on the AHPCRC's current HPC systems.

The particular CFD code being tested here and discussed throughout this paper is called 'BenchC' which is a trimmed-down version of a more comprehensive CFD code which was used, in part, to perform detailed numerical simulations of fluid-particle applications [8,11]. This code has been in use recently at the AHPCRC for various benchmarking and testing purposes, is written entirely in C, and has no external library dependencies other than MPI. It is fairly representative of most, if not all, finite-element CFD codes in use at the AHPCRC. BenchC has various built-in performance measurements such as Mega-Flop rates, detailed timings of various parts of the code including inter-processor communication times, as well as memory usage statistics. It is a fairly small code with a total of 6,700 lines.

BenchC solves the incompressible Navier-Stokes equations that govern fluid motion for a variety of systems such as aircraft aerodynamics (see Figure 9). The underlying numerical method is the finite element method which can handle unstructured meshes of any element type or even mixed element type meshes, but in general, we commonly use tetrahedral (4-nodded) element meshes generated by our in-house automatic mesh generator DMG [8,12]. Applications using meshes containing anywhere between 1 million and 5 million elements are common, although some of our AHPCRC researchers are starting to use meshes containing up to 40 million elements. In an extreme benchmarking case, we have solved an application on the Cray T3E-1200 with 1 billion tetrahedral elements (850 total equations) using all 1056 processors [13].

BenchC is a time-accurate implicit flow solver, so for each non-linear iteration of each time step, a fully coupled equation system is solved. This equation system solves for the velocity and pressure variables at each nodal point of the mesh. Traditionally, the left-hand-side sparse matrix and the right-hand-side vector are formed based on traditional finite-element numerical integration of linear basis functions, and then the matrix is inverted and multiplied with the right-hand-side vector to generate the solution update. Because the number of equations being solved are typically in the 100s of thousands or millions, a direct solver is far too computationally expensive, so generally, iterative solvers are used. BenchC uses a GMRES-based iterative solver with a diagonal pre-conditioner. For a point of reference, in a typical CFD application that we would perform, anywhere between 100 to 2000 time steps are computed, 4 non-linear iterations are typically used for each time step, and roughly 20 GMRES solver iterations are used to (approximately) solve the equation system at each non-linear iteration.

With iterative solvers, the left-hand-side matrix does not need to be inverted, but its influence is required in the form of matrix-vector multiplications. The GMRES iterative solver will provide various vectors that are normally multiplied with the "user provided" left-hand-side matrix to form a resultant vector, which is then used in the GMRES algorithm for its next iteration. Most of our AHPCRC CFD codes, including BenchC, use a different scheme called matrix-free methods. Instead of forming and storing the left-hand-side sparse matrix to be used for matrix-vector multiplication, we form the matrix-vector resultant vector directly whenever required within the GMRES algorithm. We can

do this since we know the exact formulation that would have been used to create the left-hand-side sparse matrix. While adding somewhat overall to the number of calculations being performed, these matrix-free codes use significantly less memory since the left-hand-side matrix does not need to be stored. The numerical results are exactly the same with matrix-free methods as if we actually went through a matrix-vector multiply procedure using a left-hand-side sparse matrix.

These matrix-free methods were developed in the early 90's on the CM-5 [14], which as with most systems in those days, had very low memory per processor. The AHPCRC's CM-5 had 32 Mega-Bytes of memory per processor.

Because of BenchC's usage of matrix-free methods, roughly 70 percent of its time is spent directly forming these matrix-vector resultant vectors using traditional finite element formation methods (i.e. numerical integrations of linear basis functions). This part of the code is called the 'Block', and it is important that this routine runs at optimal speeds. Another roughly 15 percent of the time is spent in other GMRES routines that typically includes vector dot-products, reductions, scalar-vector multiplies, and other vector-based linear algebra routines. Another 5 percent or so is spent in another routine similar to Block, but is used to form the initial diagonal pre-conditioner and right-hand-side vector. The rest of the code's time is spent in general overhead, vector updates, a few reductions, some I/O, and inter-processor communication. For benchmarking, most I/O is turned off. All floating point calculations in BenchC are double-precision.

BenchC also has a sparse-matrix mode of operation (as opposed to the matrix-free mode) where the left-hand-side is formed and stored in a sparse form, and a normal matrix-vector multiply takes place within the GMRES iterative solver. These other routines were also fully vectorized and multi-streamed, but are not the focus of this paper. Matrix-free operations are much more common for the CFD codes at the AHPCRC, so we chose to initially concentrate on those routines for performance on the Cray X1.

The parallelism of BenchC (and all other AHPCRC finite-element CFD codes) is based on mesh partitioning and fast inter-processor communication "gather and scatter" routines. Basically, the given unstructured mesh is partitioned into contiguous pieces using a mesh partitioner, and a "mesh partition" is assigned to each processor. Typically, each processor will be assigned anywhere between 100 thousand to 1 million mesh elements. We generally use the routines provided by ParMETIS [15,16] to perform the mesh partitioning, but BenchC actually uses its own built-in parallel Recursive Center Bisection (RCB) algorithm to perform this task. An example mesh partitioning provided by ParMETIS (as seen on the surface of the mesh) is shown in Figure 4.

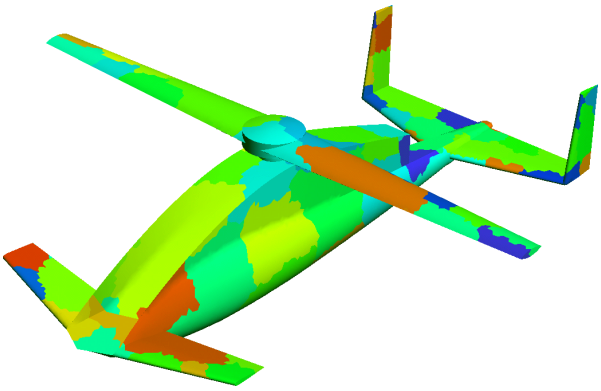


**Figure 4. The surface of an unstructured tetrahedral element mesh of a tactical unmanned aerial vehicle showing the processor assignments of each mesh partition.**

Once the mesh is distributed amongst the processors in an optimal arrangement, inter processor communication paths are built so that mesh nodes, and on-processor node copies, can be kept in sync by using the inter-processor communication procedures. Inter-processor communication is required at each GMRES iteration in order to keep these mesh node variables consistent. Due to the mesh partitioning and efficient distribution, only variables at mesh nodes that lie on partition boundaries need to be transferred to the neighboring processors. Most nodes (in many cases, 90 – 95 percent) lie in the center of mesh partitions and therefore do not need to be communicated amongst any other processors. Also, due to the mesh partitioning, the number of processor neighbors that each

processor may need to communicate with maxes out at around 15 – 20 other processors.

Very efficient non-blocking MPI communication routines are used for the actual data transfers, and these data transfer lists such as which processors need to be communicated with, and how much data is being sent to each processor, as well as the allocation of internal data buffers used for facilitating data transfers, are all set-up during the pre-processing stages and do not need to be re-computed during the actual numerical simulation parts of the code. Typically, only a few percentage of total execution time is spent performing inter-processor communication, and BenchC has built-in procedures to report these timings. Pre-processing time is always excluded from any performance timings.

It is expected that our parallel implementation and communication routines are as efficient as possible using MPI. Possibly using routines provided by Unified Parallel C (UPC) or Co-Array Fortran (CAF) could be even more efficient on architectures that support distributed shared-memory (i.e. globally addressable memory) such as the Cray X1.

## X1 ARCHITECTURE DESIGN FEATURES

The Cray X1 is an entirely new computer architecture that combines both the scalability of a distributed memory, multi-processor system with the computational performance of a specially designed processor (CPU) that incorporates both multi-streaming and vector computing capabilities. Some other features included a fast inter-processor communication network, large memory capacity, high memory-to-processor bandwidth, and a global memory address space (i.e. fully addressable by any processor). All these features are combined in an integrated system to provide a high level of computational performance including a high overall peak performance rate (12.8 Giga-Flops per processor, double-precision) with a high sustained computational capability (10% - 30% observed sustained rates). This level of performance is achieved if all (most) computations are fully vectorized and multi-streamed. Scalar operations on the Cray X1 processor do not perform at nearly the same rates as the vector computing elements, so any significant scalar computations will degrade the overall performance on the Cray X1.

Some of the key Cray X1 hardware features that are important to a user (see Appendix A) are the actual Cray X1 cabinet, and there are two types. The AC cabinet can hold up to 4 node boards, while the LC cabinet can hold up to 16, 8 on each side. Multiple cabinets can be combined together to form larger systems. Each node board within a cabinet holds 4 multi-streaming processors (MSP) and memory. The AHPCRC's X1 systems have 16 Gigia-Bytes of memory on each node board, and that memory is shared by each MSP on the board. Along with the MSPs and memory on each node board, are I/O channels and controllers, as well as the inter-processor network components and controllers. The LC cabinet also includes 4 router boards to help facilitate the inter-processor (inter-node) communications.

A MSP is the user-addressable computational unit (i.e. processor) and has a peak floating point rate of 12.8 Giga-Flops. For example, if a user requests 4 processors with MPI (i.e. 'mpirun –np 4 MY_Application'), they would get 4 MSPs, probably on the same node board. The user is, in general, responsible for breaking up their application amongst MSPs, but in future programming environment releases, OpenMP will be available for single node board applications.

Each MSP contains 4 single-streaming processors (SSP), and the compiler is (in general) responsible for breaking up the work that gets assigned to the MSP amongst its 4 SSPs. Also located on each MSP are 4 cache memory chips. Each SSP has 2 vector registers (vector computing elements) and a scalar computing element. The compiler is, again, responsible for vectorizing the code that gets assigned to each SSP.

To achieve efficiency on the Cray X1, the original problem must be broken up into smaller and smaller pieces of work, and then these smaller pieces of work are performed in parallel. A detailed schematic of this is shown in Appendix A. A typical CFD problem is first broken up into pieces using the mesh partitioning techniques described in the previous section, and each mesh

partition is assigned to a single MSP. The communication routines within BenchC are responsible for communication and coordination amongst the MSPs through the X1's interconnect network. From this point on (i.e. computational work is assigned to an MSP), the compiler takes over and will try to multi-stream and vectorize all loops. For optimal performance, most if not all loops must multi-stream and vectorize. In general, a long loop is both multi-streamed and vectorized at the same time, but for shorter loops, or loops embedded within other loops, the compiler may try to multi-stream one loop (for example, an outer loop) and vectorize another loop (for example, an inner loop). The user can have some control over this process by applying compiler directives strategically.

**PORTING AND CODE MODIFICATIONS**

The porting of BenchC to the Cray X1 was a very straightforward process. The initial port took less than 1 hour and the code was running and providing correct results within that time. However, the code wasn't fully vectorized and multi-streamed initially so performance was initially limited.

The compiler was able to fully vectorize and multi-stream the routines within the GMRES part of the code without any complications since, as stated earlier, that part of the code contains simple vector-based linear algebra routines, and no vector dependencies exist within the loops. The Cray X1 compiler provides many useful loop markings and reporting functions to specify which loops are being vectorized and multi-streamed.

The main part of BenchC, as stated previously, is the 'Block' routine where the code spends roughly 70% of its time. Block contains one single loop with roughly 1000 double-precision floating point operations per iteration, and the compiler couldn't initially vectorize or multi-stream this loop. A pseudo-Fortran outline of Block is shown in Appendix B, which is a very typical finite element method loop. As can be seen, there is one main loop where each iteration corresponds to a single mesh element. Each tetrahedral element consists of 4 mesh nodes, so initially, data associated with these nodes (N1,N2,N3,N4) are "gathered" from global node arrays. Examples of those lines of code are highlighted in blue in Appendix B. After all of the data for this particular element is gathered, all of the calculations are performed using these localized variables (highlighted in green). Once the results are computed, the results are "scattered" back into main memory for each of the 4 mesh nodes of this particular element. Examples of those lines are highlighted in red. This main loop could not be vectorized or multi-streamed because of this memory scattering procedure at the end of the loop. The compiler doesn't have any information about what the 4 indexes N1,N2,N3,N4 could be, and it is possible to generate errors in the results if any of these indices are repeated during a vectorized or multi-streamed loop. We do observe these errors in the results if this looped is forced to be vectorized and multi-streamed without any special modifications.

Our solution to this problem is to re-organize and arrange the elements of the mesh (the ones currently assigned to a particular MSP) into groups. The only restriction we have on a group of elements is that no two elements in a group can be addressing the same node index (i.e. no N1,N2,N3,N4 indices will repeat for a particular element group). With help from David Whitaker from Cray's applications department, we built an element grouping (sometimes called coloring) routine to build these element groups during the code's pre-processing stage. The algorithm is actually quite simple and does not take up much extra time. Element groups are created to contain as many mesh elements as possible, and large element groups can be created. A few of the groups pick-up remainder elements, and may have only a few members.

For a typical problem on the X1 using unstructured tetrahedral element meshes, anywhere between 44 and 47 groups are formed. This number of groups is determined internally by the coloring algorithm, but is roughly proportional to the number of mesh elements attached to each mesh node. A typical distribution of the number of elements in each group is shown in Figure 5. Typically, anywhere between a few thousand to tens of thousands of elements per group can be created for most applications, and that directly corresponds to the length of the vectorizable (and

multi-streamable) loops. The "optimal" vector size on the X1 is 64, add to that the 4 SSPs, to get a size of 256. We have observed that larger vector lengths perform better, so it is recommended that the user try to build as long vector lengths as possible.

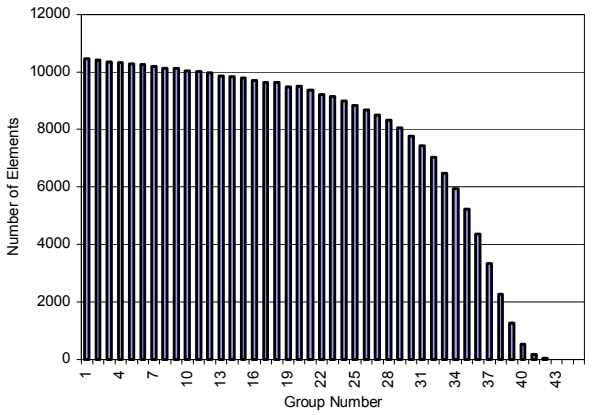



**Figure 5. Number of mesh elements in each vectorized/multi-streamed group for a typical CFD problem.**

Once the element groups are formed in the pre-processing stage, the Block loop shown in Appendix B can be slightly modified by the addition of an outer group loop, with the inner element loop remaining fairly unchanged. Since we now know that each element in a group doesn't have repeated node indices, we can force full vectorization and multi-streaming of the inner element loop by using a "CONCURRENT" compiler directive. This compiler directive replaces the more traditional "IVDEP" directive seen in past Cray systems. A high level of performance of the Block routine is now achieved without introducing any errors due to the memory scatter operations at the end. The loop performs very well on the X1, also due in part to the fact that each iteration contains roughly 1000 floating point operations.

Within this main element loop, there are also a few "IF" statements and square-roots, but the Cray X1 vector processor has been designed to support these features, so they had no effect on vectorization or performance.

Of course, each processor (MSP) has been assigned its own piece of the unstructured mesh, so each processor performs is own element grouping procedure independently. Even though each processor gets assigned roughly the same number of elements, each processor may have a slightly different number of groups with slightly different overall performance based on slight differences in vector sizes. We have observed up to 9% differences in overall run-time for the Block routine on different MSPs. We believe that some of these slight performance differences on each MSP may be contributing somewhat to the overall communication overhead (as described in the following sections) since all processors must by synchronized before a communication procedure can take place.

The inclusion of this element grouping/coloring scheme, as well as the slight modifications to the Block routine as shown in Appendix B, was all that was required to achieve full vectorization and multi-streaming of the BenchC CFD code. The MPI parallel set-up and communication parts of BenchC required no changes.

## PROCESSOR SPEED PERFORMANCE ANALYSIS

For testing of BenchC on the Cray X1, we initially selected 3 test cases. The "Small" data set contains a mesh with 440 thousand tetrahedral elements. The "Medium" data set mesh contains roughly 2 million tetrahedral elements. The "Large" data set mesh contains 4.3 million tetrahedral elements. Our basis of comparison was the performance of BenchC on the Cray T3E-1200 which we have been using as our main HPC system for the past 5 years. In the past, we have performed many simulations, testing, analysis, and optimization of these CFD codes on the T3E. We also compared the performance to some other popular HPC architectures. We initially chose to test the code using 4, 8, and 12 processors (MSPs), but have also performed X1 scalability tests using up to 28 processors, as well as a 60 MSP test performed by Cray themselves on one of their systems.

We measure run times for various parts of BenchC including "Total" time, time spent in the "Block" routine (see Appendix B), time spent in the "GMRES" routine, and time spent performing the inter-processor communication. Set-up time is

not included in any of these measurements and is a relatively small time compared to overall run time.

We have calculated the exact number of floating point operations in the Block routine, and from that, we can derive a Mega-Flop rate for the Block routine. These numbers are based on our own counting, but the Block routine has been made more and more efficient over the years so it would be surprising if the compiler can find any significant floating point operation improvements on its own. Also, through comparisons of performance of Block running on the X1 with Block on the T3E, we get a Mega-Flop rate of around 80 on a T3E processor which we know to be fairly a accurate number, and through extrapolation, we are confident about the Mega-Flop rates we are counting on the X1. We hope to confirm these rates by using Cray Performance Analysis Tools (CrayPAT) in the near future.

The raw benchmark numbers of our 3 data sets on 4, 8, and 12 processors are given in Appendix C. Overall, the performance on the X1 is roughly 42 times faster than the T3E, and the Block routine itself is roughly 53 times faster, on a per-processor basis. The "Block MF" row lists the Mega-Flop rate, so for the Block routine, we consistently are measuring roughly 4 Giga-Flops per MSP, which is almost a third of peak performance, and this number holds all the way up to our largest test case which was run on 60 MSPs. On that many processors, Block was running at a sustained rate of approximately 237 Giga-Flops. Overall (i.e. all of the counted floating point operations divided by the total time) is roughly 3 Giga-Flops per MSP.

The "GMRES" performance increases are not as high as those in Block. The GMRES vectorized/multi-streamed loops are very long, but each iteration contains only one or two floating point operations. We believe that the many operations in each loop iteration of Block contribute to its significant performance.

The percent of time spent performing communication on the X1 is larger than on the T3E. Even though we observe the inter-processor communication on the X1 to be significantly faster than on the T3E, for BenchC, it is not 42 times faster than on the T3E. Therefore, communication is taking up a larger percentage of total time.

Performance comparisons of BenchC to a SGI Origin 3000 (MIPS 14000 at 500 MHz) and an IBM sp690 SP (Power4 at 1.3 GHz) are also provided in Appendix B for the Medium data set. The Cray X1 shows significant better performance than those two systems, on a per-processor basis. The port to these other two systems was a fairly straight port. Some time was spent trying to optimize the Block routine on the IBM, but performance of Block seemed to be fairly insensitive to any code changes or re-structuring.

## PARALLEL SCALABILITY AND COMMUNICATION PERFORMANCE ANALYSIS

Good scalability of BenchC on the X1 can be seen in the tables of Appendix C, and we further tested the scalability of the code all the way to our system's 28 processors (our half populated LC cabinet contains 32 MSPs total, but 4 of them are reserved for the command-node of the system). Those results are shown in Figure 6. In this figure, speed-up is measured based on the performance of BenchC on 4 processors.

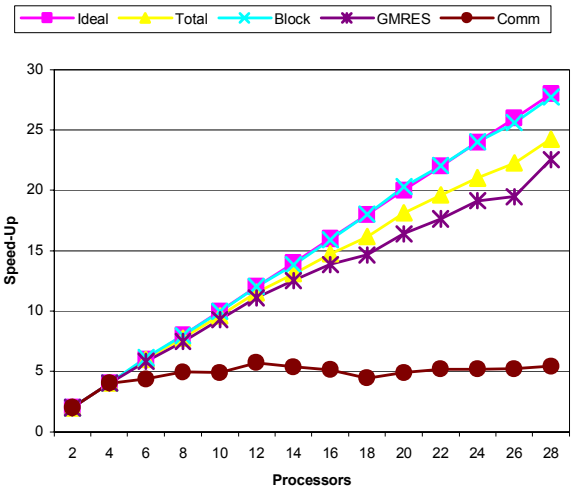



**Figure 6. Overall speed-up of the BenchC code for runs using up to 28 processors (MSPs). Speed-ups of various parts of BenchC are also shown. Speed-ups are based on the performance on 4 processors.**

As can be seen in Figure 6, linear scalability is observed for the Block routine, while overall scalability is still quite good. The scalability of the communication time, however, is flat since we measured consistent communication time (actual seconds) for all processor counts. A graph showing the percentage of total run time spent in the Block routine, the GMRES routine, and in communication is provided in Figure 7.

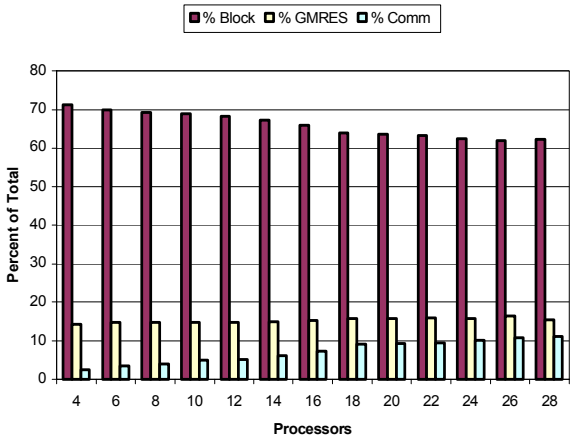



**Figure 7. Percentages of time spent in various parts of the BenchC code, for various runs using up to 28 processors (MSPs).**

Again, since communication time is rather flat, for larger processor jobs, the percentage of total time spent in the Block routine comes down and degrades our scalability somewhat.

Measuring communication time in BenchC is somewhat difficult since a communication procedure can not take place until all processors are synchronized right after the Block routine is called. As explained earlier, we believe that due to the element blocking strategy, each processor achieves slightly different performance through the Block routine, and all processors must wait for the slowest one before communication takes place. This slight out-of-sync time we believe is showing up in our communication measurements. Some of our more detailed measurements of communication time are showing this, since in some measurements, less than one third of the measured communication time is actually spent performing communication. All other time is spent (we believe) in processor synchronization.

In one of our Fortran CFD codes very similar to BenchC, we have replaced the MPI-based communication procedures with new routines based on Co-Array Fortran (CAF). More detailed timings of those routines show an almost insignificant time spent performing actual communication (i.e. for CAF, communication is accomplished by references to global shared arrays), but still, a larger actual time is measured for communication which is due to the processor synchronization time (roughly 3 to 5 percent of our communication measurement is actually spent transferring data within this specific CAF code). With this more precise measurement of inter-processor communication using the CAF code, we see data transfer rates of around 2 to 3 Giga-Bytes per second.

We plan to evaluate these communication procedures, and processor synchronization times, in more detail in the near further. We also plan to add Unified Parallel C (UPC) constructs to the BenchC code to measure its behavior for inter-processor communication.

A test of BenchC on a Cray X1 system with 60 MSPs is shown in Figure 8. For that test, we saw consistently 4 Giga-Flops per processor for the Block routine, and the overall (Total) performance is shown in Figure 8. Again, communication times were rather flat and that limited overall scalability for the higher processor counts. Due to the overall speed of the X1, especially when using up to 60 processors, we believe the problem size (the Large data set test case) was too small and the communication times began to dominate performance.

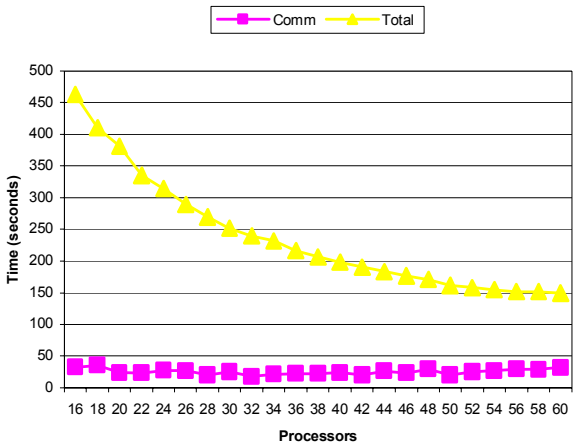



**Figure 8. Overall run times, and time spent performing inter-processor communication, for the BenchC code using up to 60 processors (MSPs).**

Finally, to test the largest application that we could perform on our current 28-processor X1 system, we performed a CFD simulation using BenchC for a test case containing 243 million tetrahedral elements (see Figure 9). This mesh contained roughly 41 million mesh nodes, with an equation system of 160 million unknowns. This example is of airflow past a cargo aircraft in a take-off configuration, and was set-up based on a model and mesh we used several years ago when performing paratrooper/cargo aircraft interaction studies [9]. The aircraft is at roughly 10 degrees angle-of-attack, so large separation regions are observed over the top of the wing. The image was rendered on the AHPCRC's Cray T3E-1200 using 350 processors with our Presto Visualizer's volume rendering capabilities [17,18]. A volume rendering of velocity magnitude (blues are low velocity, reds are high velocity) is shown in Figure 9.

A mesh with 243 million elements was chosen since this was, generally, the larges we could fit into the memory of 28 MSPs. BenchC used roughly 76.8 Giga-Bytes of memory for this application. A total of 2,000 total time steps were computed, with 4 non-linear iterations each time step, and 15 GMRES iterations for each non-linear iteration. Each data file written to disk (we wrote a data file every other time step) is about 1.3 Giga-Bytes in size, and it takes roughly 10 Giga-Bytes to store the tetrahedral element mesh itself.

We could compute a time step in approximately every 3 minutes, and the Block routine ran at a rate of 113.8 Giga-Flops (roughly 91.5 Giga-Flops overall). We believe applications at this scale are now becoming practical on systems such as the Cray X1, since results can be obtained in only a few days.

## CONCLUSIONS

Over the last 6 months, we have been testing the AHPCRC's new Cray X1 for various numerical simulation applications such as the unstructured mesh CFD applications discussed in this paper. Overall, the experience has been positive and the X1 has performed very well for our unstructured mesh CFD codes, achieving overall 43 time faster performance compared to a Cray T3E on a per-processor basis, with the main CFD kernel (i.e. the Block routine) running at almost a third of peak performance. Scalability on the X1 system has also been very good.

We are the first non-classified site to receive this new architecture, both new hardware and new software, and while there were a few software and OS problems that needed to be worked out, they have been fixed fairly quickly and the X1 has proven to be a very stable and productive machine. We are ready to go forward with the upgrades we will be receiving to the system (i.e. the addition of more processors to form a system with 128 MSPs) and begin to use the Cray X1 as our main high-performance, high-capability computing engine. The X1's performance for these CFD codes will be allowing AHPCRC and Army researchers to perform larger, more detailed, and ultimately, more accurate simulations in shorter periods of time.

## REFERENCES

1. T. Meys, "Modeling the weather on a Cray X1", *Cray User Group Conference 2003 Proceedings*, May 2003.
2. S. Aliabadi, "Contaminant propagation in battlespace environments and urban areas", *AHPCRC Bulletin*, Vol. 12, No. 4, 2001.
3. T. Holmquist, G. Johnson, "Response of silicon carbide to high velocity impact", *Journal of*

*Applied Physics*, Vol. 91, No. 9 (2002), 5858-5866.
4. G. Johnson, R. Stryk, S. Beissel, and T. Holmquist, "An algorithm to automatically convert distorted finite elements into meshless particles during dynamic deformation", *International Journal of Impact Engineering*, Vol. 27 (2002), 997-1013.
5. T. Hughes, A. Brooks, "A multidimensional upwind scheme with no crosswind diffusion", *Finite-Element Methods for Convection-Dominated Flows*, Vol. 34 (1979), American Soc. Mechanical Engineers, New York, 19-35.
6. T. Tezduyar, S. Aliabadi, M. Behr, A. Johnson, and S. Mittal, "Parallel finite-element computations of 3D flows", IEEE Computer, Vol. 26, No. 10 (1993), 27-36.
7. Y. Saad, M. Schultz, "GMRES: A generalized minimum residual algorithm for solving non-symmetric linear systems", *SIAM Journal of Scientific and Statistical Computing*, Vol. 7 (1986), 856-869.
8. A. Johnson, T. Tezduyar, "Parallel computation of incompressible flows with complex geometries", *International Journal for Numerical Methods in Fluids*, Vol. 24 (1997), 1321-1340.
9. T. Tezduyar, S. Aliabadi, M. Behr, A. Johnson, V. Kalro, and M. Litke, "Flow simulation and high performance computing", *Computational Mechanics*, Vol. 18 (1996), 397-412.
10. M. Behr, A. Johnson, J. Kennedy, S. Mittal, and T. Tezduyar, "Computation of incompressible flows with implicit finite element implementations on the Connection Machine", *Computer Methods in Applied Mechanics and Engineering*, Vol. 108 (1993), 99-118.
11. A. Johnson, T. Tezduyar, "Methods for 3D computation of fluid-object interactions in spatially periodic flows", *Computer Methods in Applied Mechanics and Engineering*, Vol. 190 (2001), 3201-3221.
12. A. Johnson, T. Tezduyar, "Advanced mesh generation and update methods for 3D flow simulations", *Computational Mechanics*, Vol. 23 (1999), 130-143.
13. S. Aliabadi, "Hydrodynamic simulations using an unstructured mesh with 1 billion tetrahedral elements", *AHPCRC Bulletin*, Vol. 11, No. 1, 2001.
14. S. Aliabadi, T. Tezduyar, "Parallel fluid dynamics computations in aerospace applications", *International Journal for Numerical Methods in Fluids*, Vol. 21 (1995), 783-805.
15. G. Karypis, V. Kumar, "Parallel multi-level k-way partitioning scheme for irregular graphs", *SIAM Review*, Vol. 41 (1999), 278-300.
16. G. Karypis, V. Kumar, "Metis 4.0: Unstructured graph partitioning and sparse matrix ordering systems", Tech. Report, Department of Computer Science, University of Minnesota, 1998, available on the WWW at URL http://www.cs.umn.edu/~metis.
17. A. Johnson, "Large scale scientific visualization on Cray MPP architectures", *Cray User Group Conference 2003 Proceedings*, May 2003.
18. A. Johnson, Q. Quammen, "Presto Visualizer 2.0, Parallel Scientific Visualization of Remote Data Sets: User Guide", Tech Report, Army HPC Research Center / NetworkCS, Inc., 2002.

## ACKNOWLEDGEMENT

The research reported in this document was performed in connection with contract DAAD19-03-D-0001 with the U.S. Army Research Laboratory. The views and conclusions contained in this document are those of the authors and should not be interpreted as presenting the official policies or positions, either expressed or implied, of the U.S. Army Research Laboratory or the U.S. Government unless so designated by other authorized documents. Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

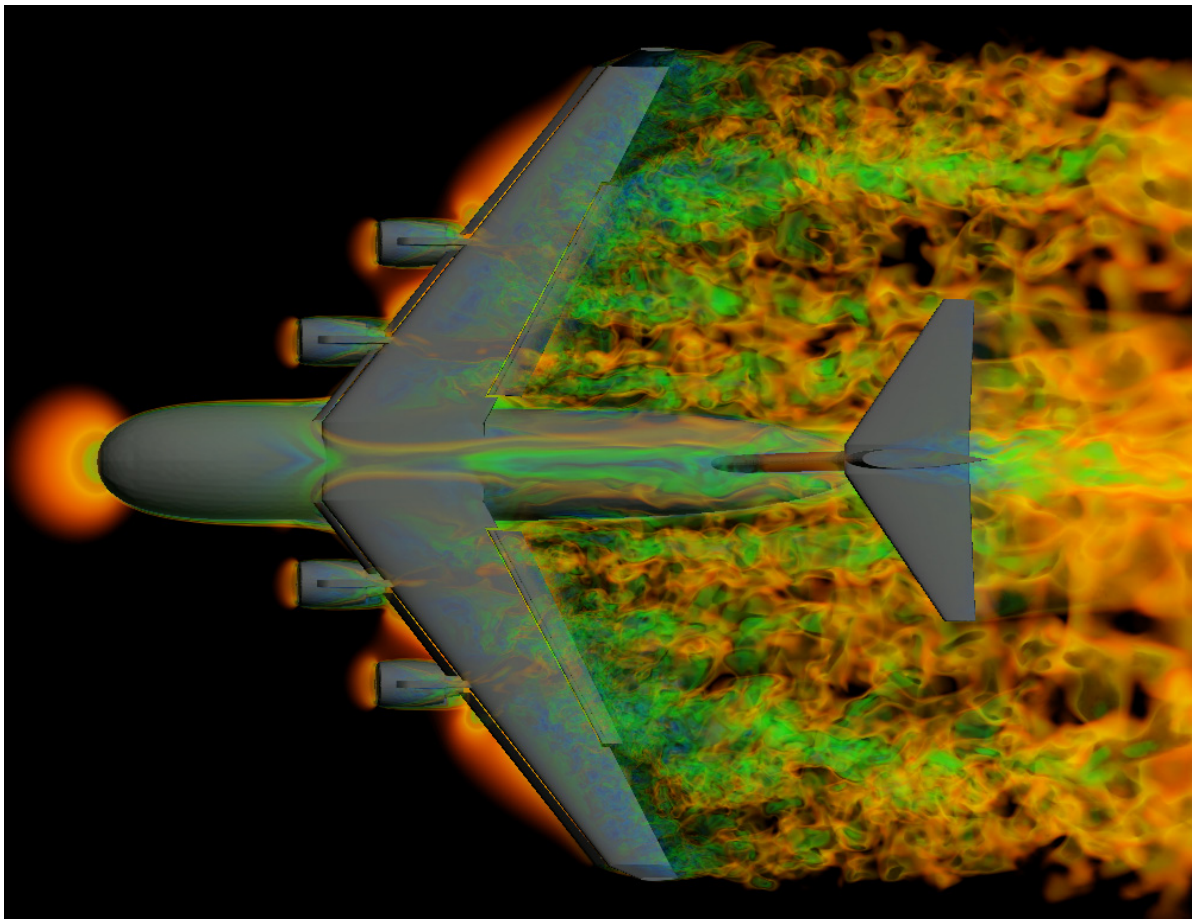
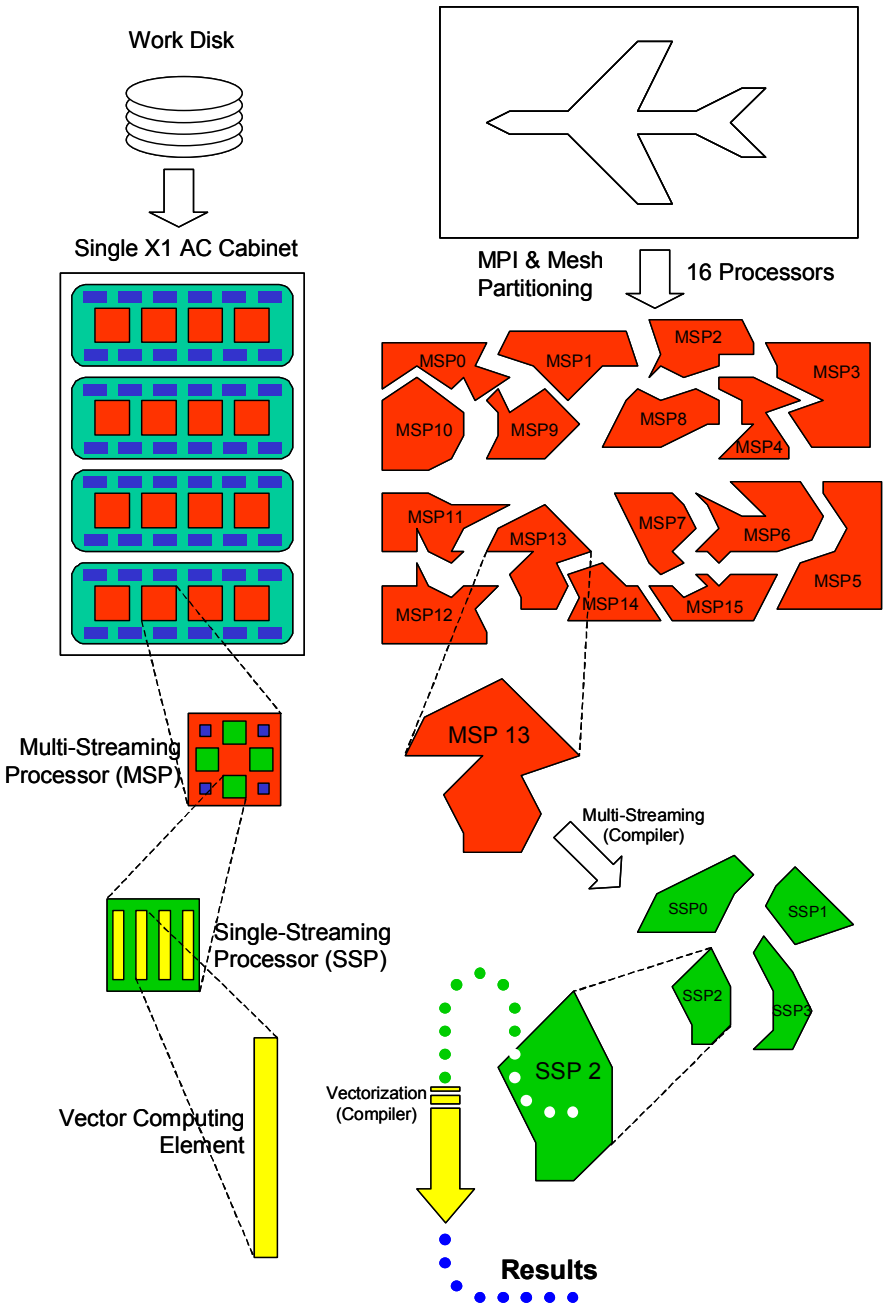


**Figure 9. Large scale simulation of airflow past a cargo aircraft in a take-off configuration. Shown is a volume-rendered image of velocity magnitude. Computed on the Cray X1 using 28 processors. The mesh contains 243 million tetrahedral elements.**

**APPENDIX A**




Schematic of the major Cray X1 hardware components (left) along with a schematic on how a CFD application will be mapped on to the associated X1 hardware. The CFD application is broken up into smaller and smaller pieces until the actual vector computing elements can be engaged by a small sub-set of the original application. Along with the vector computing elements (shown in yellow), other X1 components include the single-streaming processor (SSP) shown in green, the multi-streaming processor (MSP) shown in red, the actual X1 node boards shown in light blue, and an actual X1 AC cabinet. Memory chips (either main memory on the X1 node board, or cache memory on the MSP) are shown in blue. The user's application is responsible for splitting up the application amongst the MSPs, but the compiler (in general) breaks up the problem amongst the SSPs and vector computing elements.

## APPENDIX B

```
DO I = 1, NUMBER_OF_MESH_ELEMENTS
   N1 = IEN(1, I)
   N2 = IEN(2, I)
   N3 = IEN(3, I)
   N4 = IEN(4, I)

   X1 = X(1, N1)
   Y1 = X(2, N1)
   Z1 = X(3, N1)
   X2 = X(1, N2)
   Y2 = X(2, N2)
   Z2 = X(3, N2)
   X3 = X(1, N3)

   …SEVERAL MORE MEMORY "GATHER"…
   …STATEMENTS LIKE THESE…


   UI = SH01*U1 + SH02*U2
      + SH03*U3 + SH04*U4
   VI = SH01*V1 + SH02*V2
      + SH03*V3 + SH04*V4
   WI = SH01*W1 + SH02*W2
      + SH03*W3 + SH04*W4
   PI = SH01*P1 + SH02*P2
      + SH03*P3 + SH04*P4
   UXI = SHX1*U1 + SHX2*U2
       + SHX3*U3 + SHX4*U4
   UYI = SHY1*U1 + SHY2*U2
       + SHY3*U3 + SHY4*U4
   UZI = SHZ1*U1 + SHZ2*U2
       + SHZ3*U3 + SHZ4*U4
   VXI = SHX1*V1 + SHX2*V2
       + SHX3*V3 + SHX4*V4

   …ROUGHLY 1000 FLOATING POINT OPERAIONS…
   …PER ITERATION.  A FEW IF STATEMENTS AND…
   …A COUPLE SQURE ROOTS…


   RES(1, N1) = RES(1, N1) + RESULT_U1
   RES(2, N1) = RES(2, N1) + RESULT_V1
   RES(3, N1) = RES(3, N1) + RESULT_W1
   RES(4, N1) = RES(4, N1) + RESULT_P1
   RES(1, N2) = RES(1, N2) + RESULT_U2
   RES(2, N2) = RES(2, N2) + RESULT_V2
   RES(3, N2) = RES(3, N2) + RESULT_W2

   …SEVERAL MORE MEMORY "SCATTER"…
   …STATEMENTS LIKE THESE…
ENDDO    !…END OF ELEMENT LOOP
```

```
DO IG = 1, NUMBER_OF_GROUPS
   IG_BEG = GROUPS(IG)
   IG_END = GROUPS(IG+1)

   !DIR$ CONCURRENT
   DO I = IG_BEG, IG_END
      N1 = IEN(1, I)
      N2 = IEN(2, I)
      N3 = IEN(3, I)
      N4 = IEN(4, I)

      X1 = X(1, N1)
      Y1 = X(2, N1)
      Z1 = X(3, N1)
      X2 = X(1, N2)
      Y2 = X(2, N2)
      Z2 = X(3, N2)
      X3 = X(1, N3)

      …SEVERAL MORE MEMORY "GATHER"…
      …STATEMENTS LIKE THESE…


      UI = SH01*U1 + SH02*U2
         + SH03*U3 + SH04*U4
      VI = SH01*V1 + SH02*V2
         + SH03*V3 + SH04*V4
      WI = SH01*W1 + SH02*W2
         + SH03*W3 + SH04*W4
      PI = SH01*P1 + SH02*P2
         + SH03*P3 + SH04*P4
      UXI = SHX1*U1 + SHX2*U2
          + SHX3*U3 + SHX4*U4
      UYI = SHY1*U1 + SHY2*U2
          + SHY3*U3 + SHY4*U4
      UZI = SHZ1*U1 + SHZ2*U2
          + SHZ3*U3 + SHZ4*U4
      VXI = SHX1*V1 + SHX2*V2
          + SHX3*V3 + SHX4*V4

      …ROUGHLY 1000 FLOATING POINT OPERATIONS…
      …PER ITERATION.  A FEW IF STATEMENTS AND…
      …A COUPLE SQUARE ROOTS…


      RES(1, N1) = RES(1, N1) + RESULT_U1
      RES(2, N1) = RES(2, N1) + RESULT_V1
      RES(3, N1) = RES(3, N1) + RESULT_W1
      RES(4, N1) = RES(4, N1) + RESULT_P1
      RES(1, N2) = RES(1, N2) + RESULT_U2
      RES(2, N2) = RES(2, N2) + RESULT_V2
      RES(3, N2) = RES(3, N2) + RESULT_W2

      …SEVERAL MORE MEMORY "SCATTER"…
      …STATEMENTS LIKE THESE…
   ENDDO    !…END OF ELEMENT GROUP LOOP

ENDDO    !…END OF GROUP LOOP
```

Pseudo-Fortran code of the main computational kernel for the CFD finite-element benchmark codes. The original code is shown on the left while the slightly modified code that incorporates an element grouping (coloring) strategy is shown on the right. The original code can not be vectorized or multi-streamed due to the memory scatter operations highlighted in red. The inner element loop in the modified code on the right can be fully vectorized and multi-streamed due to the element grouping strategy.

## APPENDIX C

| Small Data Set (0.44M Elements) | | T3E-1200 | Production X1 | |
|---|---|---|---|---|
| | | Seconds | Seconds | Faster |
| **4 CPU** | Block | 3,537.0 | 66.5 | 53.2 x |
| | Block MF | 304.6 | 16,195.0 | 31.6% Peak |
| | GMRES | 360.5 | 16.4 | 22.0 x |
| | Total | 4,203.6 | 103.0 | 40.8 x |
| | % Comm | 1.0 | 7.1 | |
| **8 CPU** | Block | 1,749.0 | 31.3 | 55.9 x |
| | Block MF | 616.0 | 34,423.0 | 33.6% Peak |
| | GMRES | 188.8 | 9.9 | 19.1 x |
| | Total | 2,099.4 | 55.1 | 38.1 x |
| | % Comm | 1.2 | 14.0 | |
| **12 CPU** | Block | 1,133.7 | 22.1 | 51.3 x |
| | Block MF | 950.0 | 48,728.4 | 31.7 % Peak |
| | GMRES | 128.1 | 9.3 | 13.8 x |
| | Total | 1,376.6 | 44.3 | 31.1 x |
| | % Comm | 1.7 | 18.9 | |

| Medium Data Set (2.0M Elements) | | T3E-1200 | SGI Origin | IBM p690 SP | Production X1 | | | |
|---|---|---|---|---|---|---|---|---|
| | | Seconds | Seconds | Seconds | Seconds | Faster T3E | Faster SGI | Faster IBM |
| **4 CPU** | Block | 3,997.7 | 2,049.7 | 1,164.2 | 76.0 | 52.6 x | 27.0 x | 15.3 x |
| | Block MF | 304.1 | 593.1 | 1,044.3 | 15,995.0 | 31.2% Peak | | |
| | GMRES | 388.1 | 209.7 | 115.1 | 14.7 | 26.4 x | 14.3 x | 7.8 x |
| | Total | 4,715.4 | 2,415.6 | 1,375.8 | 107.2 | 44.0 x | 22.5 x | 12.8 x |
| | % Comm | 0.8 | 1.4 | 1.2 | 3.5 | | | |
| **8 CPU** | Block | 1,994.0 | 786.0 | 503.2 | 38.2 | 52.2 x | 20.6 x | 13.2 x |
| | Block MF | 609.7 | 1,546.9 | 2,416.1 | 31,830.3 | 31.1% Peak | | |
| | GMRES | 198.3 | 126.9 | 61.6 | 8.1 | 24.5 x | 15.7 x | 7.6 x |
| | Total | 2,361.5 | 981.1 | 614.4 | 57.0 | 41.4 x | 17.2 x | 10.8 x |
| | % Comm | 0.9 | 1.9 | 2.1 | 6.8 | | | |
| **12 CPU** | Block | 1,335.9 | 441.4 | 374.7 | 25.1 | 53.2 x | 17.6 x | 14.9 x |
| | Block MF | 910.1 | 2,754.4 | 3,244.5 | 48,411.6 | 31.5% Peak | | |
| | GMRES | 132.3 | 66.9 | 50.4 | 5.6 | 23.6 x | 11.9 x | 9.0 x |
| | Total | 1,589.9 | 550.5 | 466.8 | 39.0 | 40.8 x | 14.1 x | 12.0 x |
| | % Comm | 1.1 | 2.5 | 3.0 | 9.5 | | | |

| Large Data Set (4.3M Elements) | | T3E-1200 | Production X1 | |
|---|---|---|---|---|
| | | Seconds | Seconds | Faster |
| **4 CPU** | Block | 4,327.3 | 82.5 | 52.5 x |
| | Block MF | 304.9 | 15,991.6 | 31.2% Peak |
| | GMRES | 438.3 | 17.4 | 25.2 x |
| | Total | 5,120.0 | 117.2 | 43.7 x |
| | % Comm | 0.7 | 2.8 | |
| **8 CPU** | Block | 2,175.5 | 41.5 | 52.4 x |
| | Block MF | 606.5 | 31,791.8 | 31.0% Peak |
| | GMRES | 232.9 | 9.6 | 24.3 x |
| | Total | 2,587.7 | 61.1 | 42.4 x |
| | % Comm | 0.8 | 4.2 | |
| **12 CPU** | Block | 1,466.4 | 27.5 | 53.3 x |
| | Block MF | 899.8 | 47,923.6 | 31.2% Peak |
| | GMRES | 151.9 | 7.0 | 21.7 x |
| | Total | 1,741.8 | 42.2 | 41.3 x |
| | % Comm | 0.9 | 5.8 | |

The numbers in the "Block MF" rows are listing a Mega-Flop rate, all other numbers refer to seconds.