Unified Parallel C within Computational Fluid Dynamics Applications on the Cray X1(E)

Andrew A. Johnson
Army High Performance Computing Research Center,
Network Computing Services, Inc.

ABSTRACT: *We present some recent results of the performance of our in-house computational fluid dynamics (CFD) codes on the Cray X1 and X1E for use in large-scale scientific and engineering applications. Both the vectorization/multi-streaming performance, and the code's implementation and use of Unified Parallel C (UPC) as the main inter-processor communication mechanism, will be discussed. Comparisons of UPC performance and behaviour with the traditional MPI-based implementation will be shown through CFD benchmarking and other special communication benchmark codes.*

KEYWORDS: Cray X1, Unified Parallel C, CFD, vector systems, benchmarking

1. Introduction

The Army High Performance Computing Research Center (AHPCRC) has been using the Cray X1, and now the Cray X1E, for roughly 2 ½ years for various scientific and engineering applications including computational fluid dynamics (CFD). CFD has always been a large application area within the AHPCRC program, and throughout the center's 15 year history, several in-house CFD codes have been developed and used to address various Army applications such as high-speed missile aerodynamics, aerodynamics of both ground and aerial vehicles, both internal and external ballistics simulations, and airborne contaminant transport in urban environments. All of these CFD codes are similar in form and have lots of commonality. They are all built for unstructured meshes, are based on the stabilized Finite Element Method, solve either the time-accurate incompressible or compressible Navier-Stokes equations, solve an implicit equation system using a GMRES-based iterative equation solver, and are fully parallel based on mesh partitioning concepts and fast inter-processor communications. The specific CFD code being benchmarked and tested for this paper is called BenchC which is representative of several other CFD codes in use at the AHPCRC. BenchC is written in C and portable to almost all parallel high performance computing (HPC) systems. BenchC vectorize well on the Cray X1(E) and can achieve high sustained computational rates. Further details of the BenchC CFD code, including its performance benchmarking, are shown in Section 2.

The parallel implementation of the AHPCRC CFD codes, including BenchC, is based on MPI which is portable to almost all HPC systems. The actual evolution of the code's parallel implementation spans more than 1 ½ decades and has been implemented using several other parallel programming models throughout that time. The initial implementation of these CFD codes was on the Cray 2 in 1990, and the parallelization in those days was based on both vectorization and auto-tasking. In 1991 the code was modified to run in parallel on the Thinking Machines CM-5 based on Connection Machine Fortran (CMF) and Scientific Software Libraries (CMSSL) [1]. Those parallel programming models fit well with our CFD code's implementation on that machine, but due to the demise of Thinking Machine Corporation in the 1994 time frame, the support of those parallel programming libraries has ended. Roughly at that time (1994) we switched to a Cray T3D using the Parallel Virtual Machine (PVM) parallel programming model. In those days, PVM was the most widely used message-passing model. A few years after the T3D, the Cray T3E came around and the HPC community started to migrate to the Message-Passing Interface (MPI) parallel programming library, and our CFD codes evolved to support that model. Today, most of our CFD codes at the AHPCRC use MPI as the basis of their parallel implementation.

The Cray X1(E) systems also support newer parallel programming models which are not library-based such as MPI but are more tightly-coupled to the actual programming language themselves. These newer programming models are based on a global address space (GAS) that allows individual processors to read-from or write-to other processors' data which have been declared "shared" within the code. This new type of parallel programming makes code development easier and more efficient than MPI, and also has various performance gains associated with them, at least on the Cray X1(E). For C programs, this new parallel programming "language" is called Unified Parallel C (UPC). A similar concept in Fortran, also supported on the Cray X1(E), is called Co-Array Fortran (CAF). Throughout the 2+ year history of the Cray X1(E), we have been testing the use of UPC within our CFD codes, including BenchC, to observe the advantages of using one of these newer parallel programming models over the traditional MPI-based implementation. Our experiences and results of this analysis are discussed in Section 3.

1.1 Cray X1 and X1E Systems

The Cray X1 was first delivered to the AHPCRC in the Fall of 2002. At that time, two air-cooled systems were delivered, and subsequently, a large liquid-cooled system (pictured in Figure 1) was delivered in early 2003. Network Computing Services, Inc., which holds the Infrastructure Support contract for the AHPCRC, runs and maintains the Cray X1 for the US Army. This X1 system contained 128 Multi-Streaming Processors (MSPs). Each MSP chip is composed of four individual Single-Streaming Processors (SSPs), and the compiler automatically streams loops of a users application code to the 4 SSPs during computation. Generally, MSPs are the user-addressable "processor" within an application, but Cray now supports modes where users can address SSPs directly as a "processor" if they wish to (i.e. to avoid the multi-streaming concept altogether). Each SSP is a vector processor, so codes must vectorize in order to run efficiently on the system. The compiler is generally able to vectorize user application code (i.e. long loops), but in many cases, code porting and optimisation work is required to get the compiler to fully vectorize application code. There is also a serial processor on each SSP, but its performance is orders of magnitude less than the vector computing elements on the SSP, so to achieve good performance on the system, most if not all of the user's code must vectorize.

Figure 1. The AHPCRC's 256-processor Cray X1E at Network Computing Services, Inc. (Minneapolis, Minnesota).

The multi-streaming processor on the Cray X1 has a peak computational rate of 12.8 Giga-Flops (3.2 Giga-Flops for each SSP). Due to the fact that these are vector processors with a large number of vector registers (32 of them) with high bandwidth to memory and complex scheduling if instructions for memory references and computations, users can expect to achieve higher sustained computational rates, as opposed to those observed on "commodity" processors. For example, on the Cray X1, we can achieve roughly 33% of peak performance on an MSP, as compared to 8 to 10 percent we could achieve on an Intel-based processor. Both of these factors, high computational rates and high peak performance, make the Cray X1(E) attractive for critical large-scale numerical simulation applications.

The Cray X1 and X1E systems have other advantages such as a very fast processor inter-connect network, a global and unified memory address space, a single system image, and an advanced programming environment. These features make the Cray X1(E) a very easy-to-use system for the AHPCRC's large user-base.

In early 2005, the AHPCRC's Cray X1 was upgraded to the newer Cray X1E model. This was a simple node-board swap and didn't involve changing the X1 cabinet itself or the system's inter-connect network. The main difference in the X1E compared to the X1 is a re-design of the MSP chip and node board. These differences are highlighted in Figure 2.

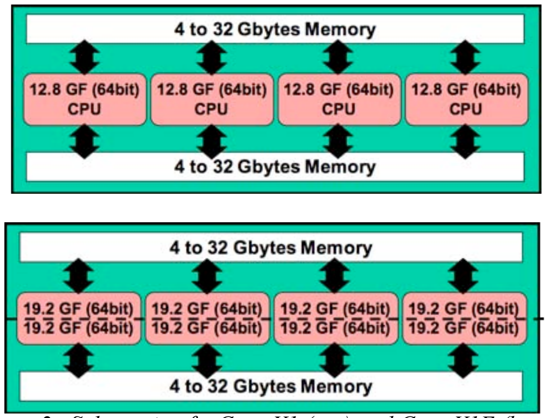

Figure 2. Schematic of a Cray X1 (top) and Cray X1E (bottom) node boards. On the X1, each "physical" node board behaves as a single "logical" node board on the system. On the X1E, due to the multi-core MSP chips, each physical node board behaves as 2 "logical" node boards on the system. Each of the 2 multi-core MSPs on a chip are assigned to separate logical node boards, even though they share cache and memory/network bandwidth.

As seen in Figure 2, the Cray X1 node board (depicted on the top) holds 4 MSP modules. These MSPs act as individual processors with their own cache, but share the large main memory that is located on each node board. Four MSPs share memory on a node board but do not share memory with the MSPs on other node boards. To address that memory, the MSPs would have to go through the processor inter-connect network.

On the newer Cray X1E (depicted on the bottom of Figure 2), each single MSP module is split into a multi-core chip. Each physical MSP chip now has two MSPs on it, and they both share cache and bandwidth to both memory and the network. Overall, this has the effect of doubling the number of processors on the system and now the AHPCRC's Cray X1E has 256 MSPs. Also, each MSP's clock rate went up 45% which increased the theoretical computational rate of each MSP to 19.2 Giga-Flops. Even though each physical node board now holds 8 MSPs, each node board is "logically" split down the middle and behaves as 2 separate node boards, each with 4 MSPs. One of the drawbacks to this design is that the amount of cache and bandwidth to memory associated with each MSP hasn't also increased by a factor of 2. This has the effect of lowering the amount of cache and bandwith associated with each MSP. As will be seen in Section 2, this has had an effect on the performance of our CFD codes on the X1E. We don't see the 45% increase in performance that the increased clock-rate of the MSPs might suggest, but only observe, roughly 10 to 15 percent increase in performance due to these cache and bandwidth limitations on the Cray X1E.

2. Computational Fluid Dynamics

CFD is a large topic with many types of numerical methods, implementations, and codes. The differences in various CFD codes is large and varied enough that conclusions made about the implementation and/or performance of our CFD codes may not hold true for other types of codes which use different numerical methods or implementation schemes. However, we feel that our implementation, performance, and experiences with our unstructured-mesh, implicit CFD codes on the Cray X1(E) will generally apply to other similar codes of this type, which are very common these days due to the necessity for real-world complex geometry applications which require unstructured meshes.

Our CFD codes that are being benchmarked here are built for unstructured meshes based on the Finite Element Method. Generally, we use meshes consisting of 4-nodded tetrahedral elements, even though the codes support several element types or arbitrary combination of element types. The formulations are (generally) the incompressible Navier-Stokes equations, but we also have compressible flow formulations that are not being benchmarked in this paper. The formulations are stabilized using SUPG/PSPG terms for advection-dominated flows, are time-accurate, and implicit using a GMRES-based iterative equation solver to solve the large system of coupled equations at each non-linear iteration of each time step. More information about these codes and formulations can be found in [2, 3]. Although there are several in-house codes based on this general framework in used at the AHPCRC, the specific code being tested and benchmarked for this paper is called BenchC, developed by the author. This code is fairly small (roughly 6,000 lines), is written entirely in C, can use either MPI or UPC as the main communication mechanism, is very portable, has been tested on many types of HPC systems, and has built-in performance monitoring and timing routines. Previous reporting of the vectorization of BenchC and its performance on the Cray X1 has been reported in [4].

One special feature of our CFD codes, including BenchC, is its implementation of both matrix-free and sparse-matrix equation solver modes. In either mode, the GMRES iterative equation solver will require that a matrix-vector multiply takes place, many times, using the Finite Element formulation's large left-hand-side matrix. If this matrix is formed and stored in a sparse form (i.e. using the code's sparse-matrix mode), the matrix-vector multiplications take place normally. In the matrix-free mode, the left-had-matrix is never stored, and the matrix-vector multiplication product is formed directly based on the original Finite Element formulation whenever required by the GMRES solver. We generally use the matrix-free mode due to its use of less memory since the large left-hand-side matrix is never stored. Our CFD codes in the sparse-matrix mode can use, roughly, 2.5 times more memory than when using the matrix-free mode. The matrix-free mode does have an extra cost associated with it because, generally, it requires more calculations to be performed. The sparse matrix-vector product routine has been fully vectorized on the Cray X1(E), but is not the focus of this paper, and we will generally concentrate on the matrix-free mode for performance testing and results.

2.1 Vectorization

The unstructured mesh Finite Element flow solver BenchC (and others in-house CFD codes of this type) vectorize well on the Cray X1(E). In the matrix-free mode, roughly 70% of execution time is spent forming these resultant matrx-vector product vectors within the GMRES solver. Another roughly 15% is spent within the GMRES function itself, and another roughly 10% if execution time is spent forming the right-hand-side vector and diagonal pre-conditioner. Through certain optimisations and some minor changes to the code's algorithms, we can achieve 100% vectorization of our CFD codes. The main addition we had to add was a mesh-element coloring algorithm as part of the pre-processing stages so that the code's element loops could be broken-up into smaller vectorizable loops to avoid the memory conflicts in the Finite Element assembly operations (i.e. the memory scatter operations at the end of each element loop). Comprehensive details about the vectorization of the BenchC CFD code were given in [4] and are not the focus of this paper.

2.2 Performance

We performed several runs of the BenchC flow solver on the Cray X1, Cray X1E, and an AMD Opteron cluster for a CFD application using a mesh containing roughly 2 million tetrahedral elements. The Opteron cluster is from Atipa and consists of 75 nodes, each containing dual AMD Opteron processors running at 2.2 GHz. A high-speed switched Myrinet network connects all of the nodes together for MPI communication. All systems are located at and operated by NetworkCS, Inc. for the AHPCRC and US Army. The Cray X1 numbers were obtained before it was upgraded to the Cray X1E.

In order to make the comparisons of the Cray X1(E) performance to the Opteron cluster performance, SSPs (i.e. Cray's single-streaming processors) are used as the unit of comparisons to AMD Opteron processors. The BenchC flow solver actually runs in MSP mode (i.e. using the multi-streaming processors), and the number of MSPs used for each run was multiplied by 4 for comparison purposes since each MSP contains 4 SSPs within.

We performed exactly the same run of the BenchC flow solver using 16, 32, 48, and 64 processors (or SSP equivalents on the X1 and X1E). The total run time was measured and was converted to a rough estimate of the overall sustained Giga-Flop rate based on our estimation of the total number of calculations performed. Input and output times, as well as simulation set-up times, are not measured in these benchmark runs. The results are shown in Figure 3.

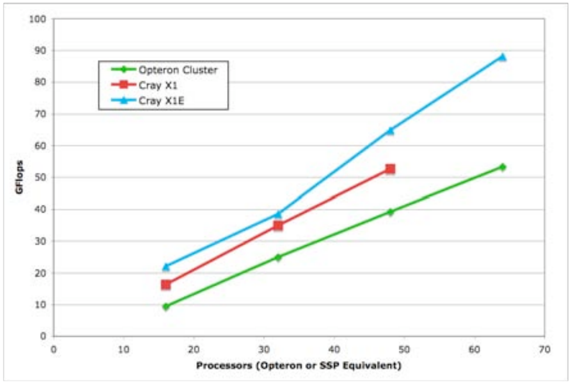

Figure 3. Matrix-free parallel speed-ups (shown as overall sustained Giga-Flop rates) for an application with a fixed mesh size of roughly 2 million tetrahedral elements. Shown are results from the Cray X1 and X1E, as well as an AMD Opteron cluster. Even though MSPs were used for the Cray X1 and X1E runs, shown in this graph are the SSP equivalents (4 SSPs for each MSP).

The performance numbers in Figure 3 show that, as expected, the Cray X1E is the fastest, followed by the Cray X1 and Opteron cluster in that order. The Cray X1E has an overall sustained computational rate of roughly 5.5 Giga-Flops per MSP (28% peak) while the Cray X1 had roughly 4.2 Giga-Flops per MSP (33% peak). Although the X1E is faster than the X1, it has a lower percentage of peak due to the lower bandwith and smaller cache of the X1E since each physical MSP is now dual-core and the two "logical" MSPs contend for resources of the same cache and network. On average, the 2.2 GHz Opterons run at about 60% of a Cray X1E's SSP equivelant's performance.

The smaller cache and contention for resources between the two multi-core MSPs on the Cray X1E has reduced our percentage of peak in the matrix-free mode because these types of CFD codes are very cache dependent and swap memory in-and-out of the vector registers at high rates. In the matrix-free mode however, the Cray X1E is still faster overall than the older Cray X1. This is not the case for the sparse-matrix mode. When these same performance tests are performed in the sparse-matrix mode, the Cray X1E is actually slower than the Cray X1 as shown in Figure 4. This is because the sparse-matrix mode relies even more heavily on cache and memory bandwidth than the matrix-free mode does.

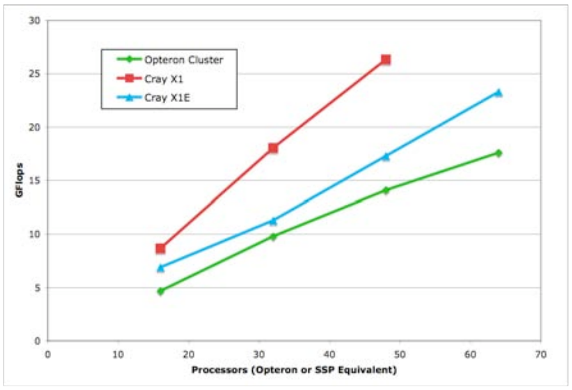

Figure 4. Sparse-mode parallel speed-ups (shown as overall sustained Giga-Flop rates) for an application with a fixed mesh size of roughly 2 million tetrahedral elements.

As stated earlier, the sparse-matrix mode isn't used nearly as often as the matrix-free mode, and therefore, hasn't received as great attention to optimisation as the matrix-free mode, even though the sparse matrix-vector product routines do fully vectorize. Also seen in Figure 4 is that the overall sustained rates are, overall, much slower than the matrix-free mode, and that is another reason (aside from higher overall memory usage), that the sparse-matrix mode isn't used as often. However, with more attention to these algorithms though higher optimisation, the behaviour could change and the performance of the sparse-matrix mode could improve.

Although not highlighted in this paper, the AHPCRC unstructured-mesh CFD codes do achieve high levels of scalability using large numbers of processors. In most cases, almost linear speed-up is observed to high processor number counts. Some of this scalability behaviour of BenchC is discussed in Section 3.1.

2.3 Large-Scale Simulations

The BenchC flow solver has been used to solve many large-scale CFD simulations on the AHPCRC's Cray X1 and X1E systems. We try to test the limits of these CFD codes and HPC systems by determining what types of large applications can be solved, what sorts of challenges are involved in such large-scale simulations (i.e. pre- and post-processing), and what kinds of results are obtained at these large scales. One such application is shown in Figure 5 that involves airflow past an unmanned aerial vehicle (UAV). The mesh used in this simulation contains roughly 450 million tetrahedral elements and was solved in the matrix-free mode. Roughly 100 MSPs were used to compute this time-accurate simulation on the Cray X1.

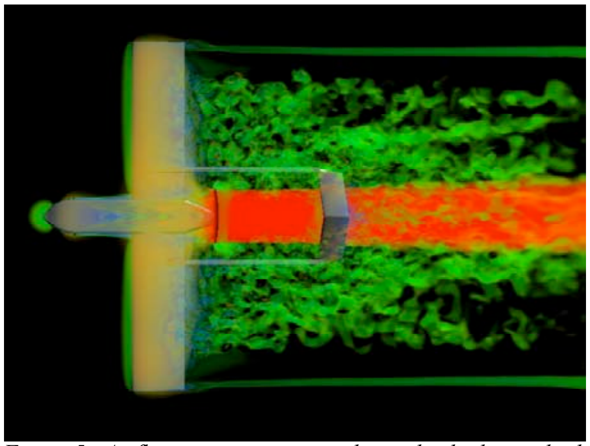

Figure 5. Airflow past an unmanned aerial vehicle at a high angle-of-attack. The mesh used for this simulation contains roughly 450 million tetrahedral elements. Shown is a volume-rendering of velocity magnitude.

One of the largest application we have solved to-date on the AHPCRC's Cray X1E is shown in Figure 6 and simulates the airflow past a military ground vehicle travelling at 60 miles-per-hour. The mesh used in this simulation contains 1.1 billion tetrahedral elements and was computed using 252 MSPs of the AHPCRC's Cray X1E. The overall sustained computational rate for this simulation was measured at 1.25 Tera-Flops.

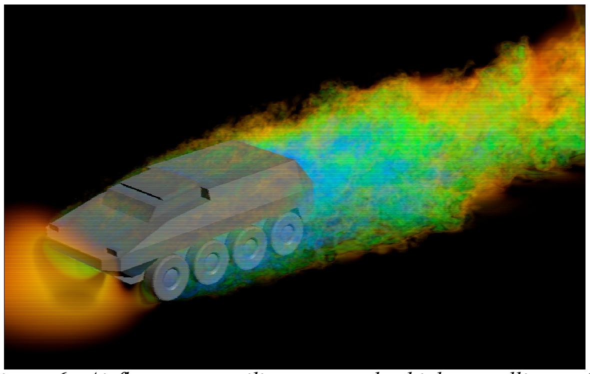

Figure 6. Airflow past a military ground vehicle travelling at 60 miles-per-hour. The mesh used for this simulation contains roughly 1.1 billion tetrahedral elements. The sustained computational rate for this calculation was measured at 1.25 Tera-Flops using 252 processors (MSPs) of the Cray X1E. Shown is a volume-rendering of velocity magnitude.

The Cray X1(E) gives us the ability to perform very detailed and accurate CFD simulation at these scales which would have been impossible only a few years ago. We are currently exploring these more accurate and high-fidelity results in more detail to determine the advantages (both in accuracy and complexity) and challenges (both pre- and post-processing) of performing simulations at this scale.

3. Unified Parallel C

A new type of parallel programming based on a global address space (GAS) is now being supported on several HPC systems, including the Cray X1 and X1E. One of these models is Unified Parallel C (UPC) which is an extension to the C language which gives processors direct access to data located on other processors by simple reads and/or writes to data and/or arrays which are declared to be shared within a user's application code. A similar concept in Fortran is called Co-Array Fortran, which is also supported on the Cray X1(E). UPC and CAF are not library-based programming models such as MPI, but are extension to the languages themselves where inter-processor communication and access to data distributed across the parallel machine are inherent to the language. All memory on the Cray X1(E) is addressable by any processor due to the systems implementation of a global address space which spans the entire system, and the way application codes such as the BenchC flow solver can take advantage of this feature is by using UPC. A detailed description of the UPC language is not the focus of this paper, but more details on it can be found in [5, 6, and 7]. The concepts in UPC and CAF are fairly simple and interested readers can learn these language extensions in only a few hours.

The three main advantages of using a GAS parallel programming model such as UPC or CAF over the traditional MPI library-based model are; 1) higher productivity and easier, more efficient parallel programming; 2) the ability to implement new, more complex algorithms and techniques that would be difficult if not impossible to implement using MPI; 3) higher performance in inter-processor communication, at least on the Cray X1(E). Although we have seen the advantages of UPC over MPI in all three of these areas though various projects underway at the AHPCRC involving GAS programming, we will highlight advantage Number 1 and 3 in the following two sub-sections.

3.1 CFD Communication

The BenchC CFD code discussed in Section 2 can be compiled to either use MPI as its main inter-processor communication mechanism, or can be compiled to use UPC as the main communication mechanism. The UPC version of BenchC was used in the benchmarking numbers reported in Section 2. There are only 3 critical inter-processor communication routines used in the BenchC flow solver. They are called "gather", "scatter", and "reduction/broadcast". UPC versions of all of these three routines were written and compiled into the code if the UPC compilation option is chosen, and this change required the modification of around 100 lines of code. In the UPC version of BenchC, MPI is still used in the main set-up stages of the code since those are not critical performance areas.

The "gather" and "scatter" routines within the BenchC CFD code are used to tie the mesh partitions together during the time-integration stages to make sure that each processor is generating a "compatible" solution along with its neighbouring mesh partitions. The two routines are very similar in form and facilitate communication in two directions. In once sense, the "gather" function sends data from the mesh-elements to the mesh-nodes, and the "scatter" function sends data in the reverse direction from the nodes to the elements. The elements are used to formulate the problem and generate the equation system, and the nodes are used to actually solve the equation system (as stated in Section 2, using a GMRES iterative solver). These inter-processor communication routines are called many times in a simulation and are critical to the scalability performance of the code. While not going into great detail on the operation of these communication routines, we show examples of both the MPI and UPC versions of the "scatter" routine in Appendix A. The actual inter-processor communication parts of the routines are highlighted in blue, while the MPI or UPC communication routines are highlighted in red. As can be seen in these examples, UPC is simply an extension to the C language and more natural and readable over its MPI counterpart which is library based using non-blocking send and receives.

The MPI version of BenchC is probably the most efficient it can be (when implemented using MPI) and was developed/optimised over a number of years, so in most CFD simulations, the BenchC CFD code only spends between 1 and 5 percent of total execution time performing inter-processor communication on systems with a fast inter-processor network such as the Cray X1(E). When using the UPC version of BenchC, the inter-processor communication times are smaller and might bring down communication times between 1/3 or 1/2 of that used with the MPI routines. This has an impact in performance, but by bringing a communication percent down from, for example, 3 percent when using MPI to 2 or 1 ½ percent when using UPC, this has a rather small impact on overall performance.

These differences in inter-processor communication performance are more greatly observed when a small application is scaled to run on a large number of processors. In such cases, the inter-processor communication times begin to grow and can dominate the on-processor calculation times, and thus, limit the scalability of the application. To demonstrate this, we performed a scalability test of the BenchC flow solver on the Cray X1 for a small application consisting of roughly 2 million tetrahedral elements. We performed several runs of the exact same CFD calculation using both the MPI and UPC versions of the BenchC code using different processor counts from 4 MSPs to 124 MSPs. The overall speed of the code is ploted for these runs in Figure 7.

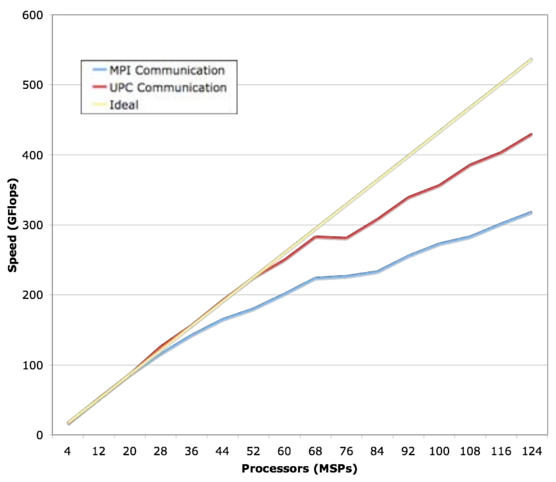

Figure 7. Scalability of the BenchC flow solver for a small application when run using large numbers of processors (MSPs) on the Cray X1. The CFD application uses a mesh containing roughly 2 million tetrahedral elements. The graph shows ideal (i.e. linear) scalability along with the scalability of both the MPI and UPC versions of BenchC. As seen in this figure, the UPC version can hold linear scalability longer than the MPI version due to the smaller communication costs associated with UPC.

As can be seen in this figure, due to UPC's lower communication costs, the UPC version of BenchC can hold linear scalability longer than the MPI version and achieves higher computational rates for larger number of processors. This higher performance of UPC over MPI, at least on the Cray X1(E), as well as the easier programming style as can be seen in Appendix A are reasons we generally have been using UPC over MPI when possible. We are also exploring the use of UPC for other, more complex parallel applications and methods where MPI would either be more expensive or too complicated to use and implement.

3.2 Inter-Processor Communication Benchmark

In order to probe the differences in inter-processor communication times associated with MPI and UPC on the Cray X1(E) in more detail, we wrote a special communication benchmark called "CommBench". There are two versions of the benchmark code, one for MPI called "CommBenchMPI", and a UPC version called "CommBenchUPC". In both codes, each processor sends data to all other processors at the same time. The communication procedure is repeated many times for messages of varying size, and these communication times are measured throughout the run with great precision. This type of communication pattern is the most common one used and models the communication patterns found within the BenchC CFD code itself. See reference [5] for more details on these benchmarks, or contact the author for the actual benchmark codes themselves.

For the MPI benchmark, there are two different ways to facilitate this "all-to-all" communication pattern. One uses MPI non-blocking routines "MPI_Isend" and "MPI_Irecv", and the other uses a more coordinated data transfer called “MPI_SendRecv”. Both of these modes are measured and are called “MPI Non-Blocking” and “MPI Send/Recv”.

For the UPC benchmark, there are four separate ways to facilitate this type of data transfer. First, since UPC uses one-sided communication, each processor can either “put” data into (i.e. write to) the other processors memory, or they could “get” data (i.e. read from) other processors memory. Also, due to the Cray X1(E)’s multi-streaming processor, we could either vectorize and multi-stream together the data transfer statement, or we could stream the loop over the other processors, and then just vectorize over the actual data transfer statement alone. These variations in UPC communication constitutes the four modes, and each one is measured and reported here. Finally, we also run the MPI communication benchmark on the AHPCRC’s AMD Opteron Cluster with a switched Myrinet network. The results of our communication benchmark are reported in Figure 8.

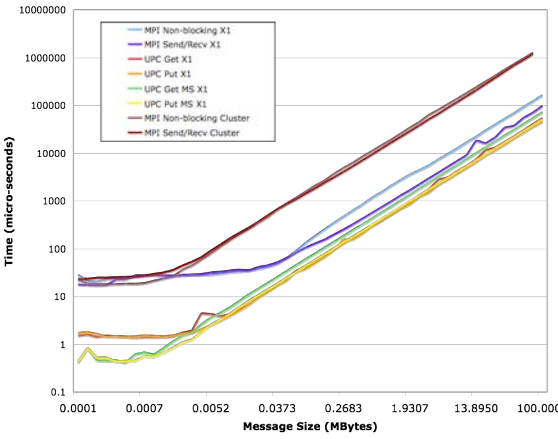

Figure 8. Single message transfer times for the AHPCRC communication benchmark using either MPI or UPC. Shown are results for the Cray X1 as well as the AHPCRC’s AMD Opteron Cluster for benchmark runs using 32 processors.

On the bottom axis of the graph in Figure 8 is the size of the messages being sent, and on the vertical axis shows the data transfer times, measured in micro-seconds, for a single data transfer. Both axises are Log-10. The results shown here are for 32 processors. Many runs were carried out and average times are shown here. Similar behaviour is observed for other processor counts.

Both the bandwidth-dominated region (the large-message region on the right) and the latency-dominated region (the small-message region on the left) can clearly be seen. The UPC communication is somewhat faster than the MPI communication on the X1, and both UPC and MPI on the X1 are roughly an order-of-magnitude faster than the Myrinet network of the Opteron cluster. The communication mode differences are more dramatic in the latency-dominated region. Both MPI on the Cray X1 and Cluster show latency times in the 12-15 micro-second range while the UPC latency times are rather dramatic at either slightly over or slightly under 1 microsecond. The multi-streaming version of the UPC benchmark (i.e. the one where streaming is split from vectorization) has a latency time of under 1 microsecond.

We believe that the extremely small latency times of UPC, at least on the Cray X1(E), will enable the implementation of codes and algorithms which are heavily dominated by the sending of many small messages between processors.

It is important to note that in our benchmark code “CommBenchMPI” and “CommBenchUPC”, all processors are sending messages to all other processors at the same time. This is a more complex procedure and benchmark than other benchmarks codes where only a single processor is sending messages to another single processor. Those single-processor benchmarks may show different bandwidth and latency behaviour than ours, but our benchmark is more representative of actual application codes that use heavy inter-processor communication and will represent real-world behaviour better than other idealized benchmark codes.

4. Conclusion

We have shown that the Cray X1E system has continued to be a very productive tool for the CFD work being carried out at the AHPCRC. Performance gains are observed on the system over those observed on the older X1 system, but this performance analysis has shown that further optimisations might be required to improve the performance even further.

We have also provided details about the use of Unified Parallel C within our CFD codes, and have shown some of the advantages, both in code development and performance, of using these global address-space programming models. We plan to explore these new parallel programming models even further and look into other, newer types of applications that could take advantage of these new features.

References

1. 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*, **108** (1993), 99-118.

2. A. Johnson and T. Tezduyar, “Parallel computation of incompressible flows with complex geometries”, *International Journal for Numerical Methods in Fluids*, **24** (1997), 1321-1340.

3. A. Johnson and T. Tezduyar, “Advanced mesh generation and update methods for 3D flow simulations”, *Computational Mechanics*, **23** (1999), 130-143.

4. A. Johnson, “Computational fluid dynamics applications on the Cray X1 architecture: Experiences, algorithms, and performance analysis”, *Proceedings of the 2003 Cray User Group Conference*, Columbus Ohio, 2003.

5. A. Johnson, “Unified Parallel C in CFD codes on the Cray X1 system”, *AHPCRC Bulletin*, Vol. 14, No. 4, 2004.

6. S. Chauvin, P. Saha, F. Contonnet, S. Annareddy, and T. El-Ghazawi, “UPC Manual”, *The George Washington University High Performance Computing Laboratory*, Version 1.0, available at http://upc.gwu.edu/

7. Berkeley Lab UPC Group; Documentation, Berkeley UPC compiler, and source code, available at http://upc.lbl.gov/

Acknowledgments

This document was developed in connection with contract DAAD19-03-D-0001 with the U.S. Army Research Laboratory. The views and conclusions contained in this presentation 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.

About the Author

Dr. Andrew Johnson is a Senior Scientist at Network Computing Services, Inc. working on the Army High Performance Computing Research Center (AHPCRC) program. He has been with that program since its inception in 1990. Dr. Johnson performs research, development, and support activities in the areas of Computational Fluid Dynamics, High Performance Computing, Automatic Mesh Generation, Geometric Modeling, and Large-Scale Scientific Visualization on parallel architectures. Dr. Johnson holds a Ph.D. in Aerospace Engineering from the University of Minnesota. He can be reached at 1200 Washington Avenue South, Minneapolis, MN 55415 USA. E-mail ajohn@ahpcrc.org.

Appendix A

Shown here are the inter-processor communication "scatter" functions used in the BenchC CFD code. Not shown here are the rather complicated communication pre-processing procedures that calculate the communication patterns and set-up and store the internal memory buffers and values to facilitate these communication routines. On the left is shown the MPI version that is used if BenchC is compiled in the MPI-mode, and on the right is shown the UPC version which is used if UPC is set as the main inter-processor communication mechanism. Highlighted in blue are the actual inter-processor communication stages, and the actual lines of code that involve inter-processor communication are highlighted in red.

```
void nscatter(double *bg, int len, int iflag){
  int i, j, iloc, num, nreq, i1, i2;

  for (i = nreq = 0; i < npnum; i++){
    iloc = nploc[i];
    num  = nploc[i+1] - iloc;
    MPI_Irecv(&buff[iloc*len],len*num,
         MPI_DOUBLE,np[i],MPI_ANY_TAG,
         MPI_COMM_WORLD,&ereq[nreq++]);
  }
  for (i = 0; i < epnum; i++){
    iloc = eploc[i];
    num  = eploc[i+1] - iloc;
    MPI_Isend(&bg[iloc*len],len*num,
         MPI_DOUBLE,ep[i],999,
         MPI_COMM_WORLD,&ereq[nreq++]);
  }
  if (nreq > 0) MPI_Waitall(nreq, ereq, estat);

  for (j = 0; j < npnum; j++){
    i1 = nploc[j+0];
    i2 = nploc[j+1];
#pragma concurrent
    for (i = i1; i < i2; i++){
      iloc = ibuff[i]*4;
      bg[iloc + 0] += buff[i*4 + 0];
      bg[iloc + 1] += buff[i*4 + 1];
      bg[iloc + 2] += buff[i*4 + 2];
      bg[iloc + 3] += buff[i*4 + 3];
    }
  }
}
```

```
void nscatter(double *bg, int len, int iflag){
  int i, j, iloc, iloc1, iloc2, num, ip, i1, i2;

  upc_barrier;

#pragma csd parallel for
  for (i = 0; i < epnum; i++){
    iloc1 = eploc [i]*len;
    iloc2 = eploc2[i]*len;
    num = (eploc[i+1] - eploc[i+0])*len;
    ip = ep[i];

#pragma ivdep
    for (j = 0; j < num; j++)
      buffSH[ip][iloc2+j] = bg[iloc1+j];
  }

  upc_barrier;

  for (j = 0; j < npnum; j++){
    i1 = nploc[j+0];
    i2 = nploc[j+1];
#pragma concurrent
    for (I = i1; I < i2; i++){
      iloc = ibuff[i]*4;
      bg[iloc + 0] += buff[i*4 + 0];
      bg[iloc + 1] += buff[i*4 + 1];
      bg[iloc + 2] += buff[i*4 + 2];
      bg[iloc + 3] += buff[i*4 + 3];
    }
  }
}
```

It is important to note that the internal data transfer buffer is called "`buff`" in the MPI version and "`buffSH`" in the UPC version. In the UPC version, `buffSH` is the only variable that had to be declared "shared" (a UPC variable type declaration) throughout the entire code. By declaring `buffSH` as shared, an extra "processor-dimension" is added to the array so that any processor can read-from or write-to any other processor's version by referencing this processor-dimension. For easier use of this "shared" internal data transfer variable, an on-processor only version of this array is declared and called "`buff`" within the UPC code by the pointer casting statement:

```
double *buff;
buff = (double *) buffSH[MYTHREAD];
```

This statement is called during the set-up procedures after the shared variable is declared and allocated. The value of "`MYTHREAD`" is a UPC-defined value that represents this processor's "number", and is equivalent to the value that would have been returned by the MPI function `MPI_Comm_rank`.