High-Cost CFD on a Low-Cost Cluster *

Thomas Hauser, Timothy I. Mattox, Raymond P. LeBeau,
Henry G. Dietz and P.George Huang

University of Kentucky

Abstract

Direct numerical simulation of the Navier-Stokes equations (DNS) is an important technique for the future of computational fluid dynamics (CFD) in engineering applications. However, DNS requires massive computing resources. This paper presents a new approach for implementing high-cost DNS CFD using low-cost cluster hardware.

After describing the DNS CFD code DNSTool, the paper focuses on the techniques and tools that we have developed to customize the performance of a cluster implementation of this application. This tuning of system performance involves both recoding of the application and careful engineering of the cluster design. Using the cluster KLAT2 (Kentucky Linux Athlon Testbed 2), while DNSTool cannot match the $0.64 per MFLOPS that KLAT2 achieves on single precision ScaLAPACK, it is very efficient; DNSTool on KLAT2 achieves price/performance of $2.75 per MFLOPS double precision and $1.86 single precision. Further, the code and tools are all, or will soon be, made freely available as full source code.

1 Introduction

Computational fluid dynamics (CFD) always has been a discipline in search of greater computer power. As coding techniques and processor speeds have improved, user demands for increased accuracy have met and exceeded each increase in computational ability. This problem has been magnified as the phenomena studied with CFD simulations have expanded from traditional applications used by aerospace engineers and meteorologists to more diverse problems. For instance, the CFD group at the University of Kentucky has worked on noise reduction in inkjet printers [18] and on optimizing the thermal performance of a fan-sink cooling system for high power electronics [17].

High computational cost CFD problems typically are solved at national supercomputer centers or similar facilities. However, access to these facilities is limited, with most organizations

*0-7803-9802-5/2000/$10.00 (c) 2000 IEEE.

outside of government and academia having no access at all. Alternately, large corporations and universities can purchase shared-memory supercomputers such as those built by SGI and HP; however, these machines are expensive, with unclear upgrade paths and relatively high maintenance costs. Smaller companies and research institutions simply cannot afford these systems. Thus, the very high up-front cost of CFD analysis has placed the technology beyond the reach of many of the researchers and engineers whose applications could benefit the most from the new abilities of CFD codes.

This paper presents an alternative approach to the computational challenges of advanced CFD problems: use inexpensive, high-performance, clusters of PCs – "Beowulfs."[1] Over the past few years, many very positive claims have been made in favor of PC clusters, and the hype usually gives better performance than the actual systems do. However, by carefully engineering the cluster design, using (and building) tools to improve application performance, and restructuring the application code for the cluster, it is possible to realize most of the benefits that are so often claimed for clusters.

Since February 1994, when we built the first parallel-processing Linux PC cluster, we have been very aggressively pursuing any hardware and software system technologies that can improve the performance or give new capabilities to cluster supercomputers. DNSTool requires a good network with high bisection bandwidth; we developed a new class of network architectures, and design tools for them, that make it possible to build an appropriate network at minimal cost. DNSTool also requires lots of memory bandwidth and fast floating point math; we provide these by using uniprocessor Athlon PCs and tools to accelerate floating point performance using 3DNow!. Our cluster, KLAT2 (Kentucky Linux Athlon Testbed 2), was not designed to only run DNSTool, but it was designed to run applications like DNSTool exceptionally well.

KLAT2 (figure 1) is a cluster of 64 (plus 2 "hot spare") 700MHz AMD Athlon PCs. Each PC contains 128MB of main memory and four 100Mb/s Fast Ethernet interfaces. Nine (and one spare) 32-way Ethernet switches are used in an unusual network architecture to interconnect the machines with low latency and high bandwidth.

Of course, it was necessary to restructure DNSTool to run more efficiently on KLAT2. Many of the changes were generic and would improve the code performance on any cache-based machine. Some of the changes reduced the total FLOPs needed for the computations, reducing total time somewhat by replacing floating point recomputations with more awkward reference patterns – not really what we should have done if achieving the "peak MFLOPS rate" was our goal. Still other changes were highly specific to the features of KLAT2, such as use of Athlon 3DNow! support. The changes were not easy, but applying them to other CFD codes will now be less difficult.

The combination of DNSTool and KLAT2 yields exceptionally good price/performance: $2.75 per MFLOPS double-precision 80/64-bit operations and $1.86 per MFLOPS using single-precision 3DNow!. Maintenance is made relatively painless by including spares in the initial system (and we have included these "unnecessary" spare parts in the system cost). Upgrading the system can be done incrementally by our tools to design an improved network, replacing the processors with a later (faster) version, etc. Further, although we have not experimented with clusters larger than

Figure 1: Kentucky Linux Athlon Testbed 2 (KLAT2)

KLAT2, given larger CFD problems, there is nothing about the design of KLAT2 or DNSTool that would prevent efficiently scaling the system to many more processors.

Thus, the design tools and implementation of KLAT2 and DNSTool represent a significant step toward moving complex CFD simulations from the supercomputer center to the center of your lab.

The next section of this paper describes the equations that govern DNSTool's CFD simulation. Section 3 describes KLAT2 and, more importantly, the tools that we have created to optimize the system design and to help code use the system more efficiently. In section 4, we detail how DNSTool was restructured and how we used our tools to tune the code for KLAT2. Section 5 presents both the CFD output and computational performance obtained from applying DNSTool to a real engineering problem: flow over a single turbine blade. The complete parts list and itemized cost of KLAT2 also is given in section 5. The final section of the paper summarizes the contributions and suggests directions for future research.

2 The CFD Algorithm

In theory, almost all fluid dynamics problems can be solved by the direct application of the Navier-Stokes equations to a sufficiently fine grid. Direct numerical simulation of the Navier-Stokes equations (DNS) provides a useful tool for understanding the complex physics in engineering processes. Even though with current computer technology, DNS can only be applied to simplified problems, DNS is likely to become the standard practice for engineering simulations in the

21st century. The challenges in applying DNS to practical engineering problems involve: (1) more accurate numerical solutions, (2) high demands on computer memory, and (3) greater computational speed. Traditionally, the focus of CFD research was on improving the numerical accuracy of the solution through more complicated modeling of the physical phenomena. Continual advances in computer architecture and technology have opened the door for focusing on the latter two areas. Meeting the latter two challenges will allow the application of DNS directly to practical engineering problems.

2.1 The Governing Equations for DNS

Direct numerical simulation is based on solving the full, time-dependent Navier-Stokes Equations for an ideal gas which express the conservation of mass, momentum, and energy for a compressible Newtonian fluid. The equations written in curvilinear coordinates are

$$\frac{\partial Q}{\partial t} + \frac{\partial F_\xi}{\partial \xi} + \frac{\partial F_\eta}{\partial \eta} + \frac{\partial F_\zeta}{\partial \zeta} = \frac{\partial G_\xi}{\partial \xi} + \frac{\partial G_\eta}{\partial \eta} + \frac{\partial G_\zeta}{\partial \zeta}, \tag{1}$$

where Q is the vector of the conservative variables multiplied by the volume of the computational cell, V, and is defined by

$$Q = V\,(\rho,\, \rho u,\, \rho v,\, \rho w,\, \rho e)^T\,. \tag{2}$$

The F's and G's denote the convective and viscous fluxes, respectively, and the subscripts ξ, η and ζ represent the directions of the fluxes. The inviscid fluxes are given by

$$F_\xi = \begin{pmatrix} \rho U_\xi \\ \rho u U_\xi + pV\,\partial\xi/\partial x \\ \rho v U_\xi + pV\,\partial\xi/\partial y \\ \rho w U_\xi + pV\,\partial\xi/\partial z \\ \rho(e+p/\rho)U_\xi - pV\,\partial\xi/\partial t \end{pmatrix}, \tag{3}$$

$$F_\eta = \begin{pmatrix} \rho U_\eta \\ \rho u U_\eta + pV\,\partial\eta/\partial x \\ \rho v U_\eta + pV\,\partial\eta/\partial y \\ \rho w U_\eta + pV\,\partial\eta/\partial z \\ \rho(e+p/\rho)U_\eta - pV\,\partial\eta/\partial t \end{pmatrix}, \tag{4}$$

$$F_\zeta = \begin{pmatrix} \rho U_\zeta \\ \rho u U_\zeta + pV\,\partial\zeta/\partial x \\ \rho v U_\zeta + pV\,\partial\zeta/\partial y \\ \rho w U_\zeta + pV\,\partial\zeta/\partial z \\ \rho(e+p/\rho)U_\zeta - pV\,\partial\zeta/\partial t \end{pmatrix}, \tag{5}$$

where ρ is the density, p is the pressure, u, v, w are the cartesian velocity components, $e = E + K$ is the total energy per unit volume and equal to the sum of the internal energy, E, and the kinetic energy, $K = (u^2 + v^2 + w^2)/2$, and the U's are the contravariant velocity components defined by

$$\begin{aligned} U_\xi &= V(\frac{\partial \xi}{\partial t} + \frac{\partial \xi}{\partial x}u + \frac{\partial \xi}{\partial y}v + \frac{\partial \xi}{\partial z}w), \\ U_\eta &= V(\frac{\partial \eta}{\partial t} + \frac{\partial \eta}{\partial x}u + \frac{\partial \eta}{\partial y}v + \frac{\partial \eta}{\partial z}w), \\ U_\zeta &= V(\frac{\partial \zeta}{\partial t} + \frac{\partial \zeta}{\partial x}u + \frac{\partial \zeta}{\partial y}v + \frac{\partial \zeta}{\partial z}w). \end{aligned} \tag{6}$$

The viscous fluxes in the ξ, η and ζ directions are given by

$$G_\xi = V \begin{pmatrix} 0 \\ \tau_{xx}\,\partial\xi/\partial x + \tau_{xy}\,\partial\xi/\partial y + \tau_{xz}\,\partial\xi/\partial z \\ \tau_{xy}\,\partial\xi/\partial x + \tau_{yy}\,\partial\xi/\partial y + \tau_{yz}\,\partial\xi/\partial z \\ \tau_{xz}\,\partial\xi/\partial x + \tau_{yz}\,\partial\xi/\partial y + \tau_{zz}\,\partial\xi/\partial z \\ g_1\,\partial\xi/\partial x + g_2\,\partial\xi/\partial y + g_3\,\partial\xi/\partial z \end{pmatrix}, \tag{7}$$

$$G_\eta = V \begin{pmatrix} 0 \\ \tau_{xx}\,\partial\eta/\partial x + \tau_{xy}\,\partial\eta/\partial y + \tau_{xz}\,\partial\eta/\partial z \\ \tau_{xy}\,\partial\eta/\partial x + \tau_{yy}\,\partial\eta/\partial y + \tau_{yz}\,\partial\eta/\partial z \\ \tau_{xz}\,\partial\eta/\partial x + \tau_{yz}\,\partial\eta/\partial y + \tau_{zz}\,\partial\eta/\partial z \\ g_1\,\partial\eta/\partial x + g_2\,\partial\eta/\partial y + g_3\,\partial\eta/\partial z \end{pmatrix}, \tag{8}$$

$$G_\zeta = V \begin{pmatrix} 0 \\ \tau_{xx}\,\partial\zeta/\partial x + \tau_{xy}\,\partial\zeta/\partial y + \tau_{xz}\,\partial\zeta/\partial z \\ \tau_{xy}\,\partial\zeta/\partial x + \tau_{yy}\,\partial\zeta/\partial y + \tau_{yz}\,\partial\zeta/\partial z \\ \tau_{xz}\,\partial\zeta/\partial x + \tau_{yz}\,\partial\zeta/\partial y + \tau_{zz}\,\partial\zeta/\partial z \\ g_1\,\partial\zeta/\partial x + g_2\,\partial\zeta/\partial y + g_3\,\partial\zeta/\partial z \end{pmatrix}, \tag{9}$$

where

$$\begin{aligned} g_1 &= u\tau_{xx} + v\tau_{xy} + w\tau_{xz} - q_x \,, \\ g_2 &= u\tau_{xy} + v\tau_{yy} + w\tau_{yz} - q_y \,, \\ g_3 &= u\tau_{xz} + v\tau_{yz} + w\tau_{zz} - q_z \,. \end{aligned} \tag{10}$$

The stress tensor τ and the heat flux vector q are defined by

$$\tau_{xx} = \mu\left(2\frac{\partial u}{\partial x} - \frac{2}{3}\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right)\right), \tag{11}$$

$$\tau_{yy} = \mu\left(2\frac{\partial u}{\partial y} - \frac{2}{3}\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right)\right), \tag{12}$$

$$\tau_{zz} = \mu\left(2\frac{\partial u}{\partial z} - \frac{2}{3}\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right)\right), \tag{13}$$

$$\tau_{xy} = \mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right), \tag{14}$$

$$\tau_{xz} = \mu\left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right), \tag{15}$$

$$\tau_{yz} = \mu\left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right), \tag{16}$$

and

$$q_x = -k\frac{\partial T}{\partial x}, \tag{17}$$

$$q_y = -k\frac{\partial T}{\partial y}, \tag{18}$$

$$q_z = -k\frac{\partial T}{\partial z}, \tag{19}$$

where μ and k are molecular viscosity and thermal conductivity, respectively.

The pressure is related to the density and temperature, T, according to the equation of state

$$p = (\gamma - 1)(e - \rho K) \tag{20}$$

where γ is the ratio of the specific heats. The coefficient of viscosity, μ, and thermal conductivity, k, are related by the constant Prandl number Pr.

$$k = \frac{\mu c_p}{Pr} \tag{21}$$

2.2 Numerical Algorithms

There are wide variety of CFD techniques that can solve these governing equations. Of the several CFD codes employed or under development at the University of Kentucky, we have chosen to use DNSTool [9], a code package originating at the University of Technology Munich and undergoing further development at the University of Kentucky. DNSTool is specifically designed to solve three-dimensional flow problems using the direct numerical simulation model for turbulence. It has primarily been used to simulate hypersonic flow around blunt objects [10], but is readily applicable to a wide range of engineering CFD problems. The choice of this particular code is based on it being MPI-ready, having been tested on numerous multiprocessor platforms, and on its clean program organization, making it readily accessible to code-performance enhancements.

The base DNSTool code is over 20000 lines of C, broken into more than 100 separate files. Briefly, DNSTool solves the governing equations as follows. The flux vector, representing the advective and dissipation flux terms of the Navier-Stokes equations, are solved with the AUSM flux-splitting scheme [14]

$$\begin{aligned} \vec{F}^{i}_{i+\frac{1}{2}} = |\vec{S}^{i}|_{i+\frac{1}{2}} & \left(\frac{1}{2} M_{i+\frac{1}{2}} \left(\begin{pmatrix} \rho c \\ \rho c u \\ \rho c v \\ \rho c w \\ \rho c H \end{pmatrix}_L + \begin{pmatrix} \rho c \\ \rho c u \\ \rho c v \\ \rho c w \\ \rho c H \end{pmatrix}_R \right) \right. \\ & \left. -\Phi_{i+\frac{1}{2}} \left(\begin{pmatrix} \rho c \\ \rho c u \\ \rho c v \\ \rho c w \\ \rho c H \end{pmatrix}_R - \begin{pmatrix} \rho c \\ \rho c u \\ \rho c v \\ \rho c w \\ \rho c H \end{pmatrix}_L \right) \right) + \begin{pmatrix} 0 \\ s_x\, p \\ s_y\, p \\ s_z\, p \\ 0 \end{pmatrix}_{i+\frac{1}{2}} \end{aligned} \tag{22}$$

where the geometrical quantities to account for the curvilinear coordinates are contained in $\overrightarrow{S}$ and s_x, s_y, s_z. The L subscript corresponds to the values extrapolated from the left volume center (i) to the face, the R subscript to those extrapolated from the right volume center $(i+1)$ to the face. A key feature of the AUSM technique, which was designed to handle high Mach number flows, is the replacement of the normal velocity through the face, $\left(\overrightarrow{u} \cdot \overrightarrow{S}\right) / \left|\overrightarrow{S}\right|$, with Mc, or the Mach number multiplied by the speed of sound. The term $M_{i+\frac{1}{2}}$ is evaluated as follows

$$M_{i+\frac{1}{2}} = M_L^p + M_R^m, \tag{23}$$

where the superscripts p and m indicate

$$\begin{aligned} M^p &= \begin{cases} M, & M \geq 1 \\ \frac{1}{4}(M+1)^2, & |M| < 1 \\ 0, & M \leq -1 \end{cases} \\ M^m &= \begin{cases} 0, & M \geq 1 \\ -\frac{1}{4}(M+1)^2, & |M| < 1 \\ M. & M \leq -1 \end{cases} \end{aligned} \tag{24}$$

Likewise, the pressure is evaluated as per Liou [13]

$$p_{i+\frac{1}{2}} = p_L^p + p_R^m \tag{25}$$

$$p^p = \begin{cases} p, & M \geq 1 \\ \frac{1}{4}p\left(M+1\right)^2\left(2-M\right), & |M| < 1 \\ 0, & M \leq -1 \end{cases}$$
$$p^m = \begin{cases} 0, & M \geq 1 \\ -\frac{1}{4}p\left(M+1\right)^2\left(2-M\right), & |M| < 1 \\ M. & M \leq -1 \end{cases} \tag{26}$$

The dissipation term evaluation is also M-dependent, although the construct is somewhat different:

$$\Phi_{i+\frac{1}{2}} = (1-\omega)\Phi_{i+\frac{1}{2}}^{VL} + \omega\Phi_{i+\frac{1}{2}}^{modAusm}, \tag{27}$$

with the VL term found from

$$\Phi_{i+\frac{1}{2}}^{VL} = \begin{cases} \left|M_{i+\frac{1}{2}}\right|, & \left|M_{i+\frac{1}{2}}\right| \geq 1 \\ \left|M_{i+\frac{1}{2}}\right| + \frac{1}{2}\left(M_R-1\right)^2, & 0 \leq M_{i+\frac{1}{2}} < 1 \\ \left|M_{i+\frac{1}{2}}\right| + \frac{1}{2}\left(M_L+1\right)^2, & -1 < M_{i+\frac{1}{2}} \leq 0 \end{cases} \tag{28}$$

and the AUSM dissipation term from

$$\Phi_{i+\frac{1}{2}}^{modAusm} = \begin{cases} \left|M_{i+\frac{1}{2}}\right|, & \left|M_{i+\frac{1}{2}}\right| > \tilde{\delta} \\ \frac{\left(M_{i+\frac{1}{2}}\right)^2 + \tilde{\delta}^2}{2\tilde{\delta}}. & 0 \leq M_{i+\frac{1}{2}} < 1 \end{cases} \tag{29}$$

The values of ω are a function of the pressure distribution, while the value of $\tilde{\delta}$ is a function of the eigenvalues, $\lambda^i = \overrightarrow{u} \cdot \overrightarrow{S}^i + c\left|\overrightarrow{S}^i\right|$.

The left and right extrapolated values of the conservative variables are determined by a third-order MUSCL-type extrapolation, in which any given quantity q_L or q_R is defined by

$$\begin{aligned} q_{L,i+\frac{1}{2}} &= q_i + \tfrac{1}{2}\upsilon_i \frac{\left(\Delta_+^2+\epsilon\right)\Delta_- + \left(\Delta_-^2+\epsilon\right)\Delta_+}{\Delta_+^2+\Delta_-^2+2\epsilon}, \\ q_{R,i-\frac{1}{2}} &= q_i - \tfrac{1}{2}\upsilon_i \frac{\left(\Delta_+^2+\epsilon\right)\Delta_- + \left(\Delta_-^2+\epsilon\right)\Delta_+}{\Delta_+^2+\Delta_-^2+2\epsilon}, \end{aligned} \tag{30}$$

where

$$\Delta_+ = q_{i+1} - q_i, \;\; \Delta_- = q_i - q_{i-1}, \;\; \epsilon = \max\left[\left(\kappa_1 \Delta x^n\right),\, \kappa_2\left(\Phi_{i+\frac{1}{2}}^{modAusm} - \overline{M}\right)\right]. \tag{31}$$

The diffusive flux gradients are evaluated using a variation of the circulation method

$$\int_V \overrightarrow{\nabla} q\, dV = \oint_S \overrightarrow{n} q\, dS, \tag{32}$$

or in terms of the finite volume grid

$$\left(\overrightarrow{\nabla} q\right)_{i+\frac{1}{2},j,k} = \frac{1}{V_{i+\frac{1}{2},j,k}} \left(\overrightarrow{S}^{1}_{i+1,j,k}\, q_{i+1,j,k} - \overrightarrow{S}^{1}_{i,j,k}\, q_{i,j,k} \right.$$
$$+ \overrightarrow{S}^{2}_{i+\frac{1}{2},,j+\frac{1}{2},k}\, q_{i+\frac{1}{2},,j+\frac{1}{2},k} - \overrightarrow{S}^{2}_{i+\frac{1}{2},,j-\frac{1}{2},k}\, q_{i+\frac{1}{2},,j-\frac{1}{2},k} \tag{33}$$
$$\left. + \overrightarrow{S}^{3}_{i+\frac{1}{2},,j,k+\frac{1}{2}}\, q_{i+\frac{1}{2},,j,k+\frac{1}{2}} - \overrightarrow{S}^{3}_{i+\frac{1}{2},,j,k-\frac{1}{2}}\, q_{i+\frac{1}{2},j,k-\frac{1}{2}} \right).$$

These spatial calculations are performed in a series of three sweeps in the i-, j-, and k-directions. This trio of sweeps are made each sub-iteration of the multi-step time integration, leading to as many as 12 sweeps in the case of the 4th-order Runge-Kutta method. Both the 2nd-order (for unsteady solutions) and 4th-order (for steady solutions) Runge-Kutta methods were used by DNSTool/KLAT2 machine.

2.3 Parallelization

The parallel structure of DNSTool is based on splitting the computational grid into sub-blocks, which are then distributed to each processor (figure 2). For a complex geometry, this is a nontrivial exercise where an uneven load-balance across different processors could significantly reduce the computational efficiency of the overall code. The partitioning of the grid into sub-blocks is performed independently of the grid generation, using virtual partitions to define the domains corresponding to each processor. The current algorithm governing the partition process is based on spectral recursive bisection in conjunction with a small database of the computational speeds associated with each node. The splitting is performed as a preprocessing task, generating a file that maps the grid sub-blocks onto the processors. This file is the only difference between performing a serial computation and a parallel computation with DNSTool.

Communications between the grid sub-blocks occurs when the sub-blocks exchange data about the flow variables at the boundaries. As show in figure 2, the flow variables on the edge of one grid block are communicated to the dummy points of the neighboring grid block, and vice versa. DNSTool requires such a communication step after each update, or subiteration, of the flow variables. The low-level implementation of the communication between the sub-blocks uses a MPI-based communications system. The communication model is a mailbox algorithm where the data is sent in a non-blocking communication mode as early as possible.

The overall parallel framework of DNSTool is managed by MBLIB, a code library that efficiently controls the domain decomposition and associated communication patterns. On top of this library the efficient compressible Navier-Stokes solver described previously is implemented. Thanks to MBLIB, the Navier-Stokes solver is isolated from the parallel decomposition-the solver code is the same whether the overall computation is performed on a single domain or any number of subdomains.

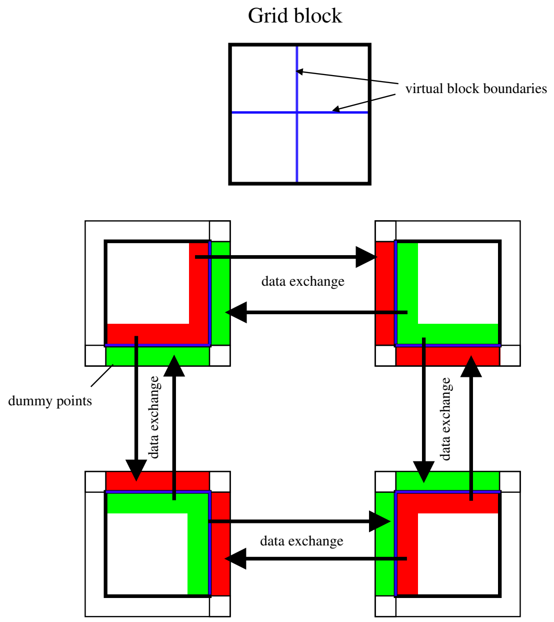

Figure 2: Communication pattern of DNSTool

3 Engineering a Cluster Supercomputer

Since February 1994, when we built the first parallel-processing Linux PC cluster [4], we have been very aggressively pursuing any hardware and software system technologies that can improve the performance or give new capabilities to cluster supercomputers. As the latest in a long sequence of clusters that we have built, KLAT2 continues our tradition of using innovative new technologies. However, unlike all our previous systems, KLAT2's base configuration uses no custom hardware - all of the performance improvements are accomplished by changing the way in which standard hardware components are configured and/or restructuring the software to take better advantage of the hardware.

Without using custom hardware, there are three key approaches that we can use to improve the performance of a CFD code on KLAT2:

1. Optimization and general restructuring of the CFD code to improve performance. Many of these optimizations are not really specific to KLAT2, but improve performance on most computers. We have performed many such optimizations, most of which involve restructuring to change the memory reference behavior and to reorder the calculations so that more redundant computations can be eliminated. Our goal has been to minimize execution time despite the fact that some of the optimizations also reduce the achieved FLOPs rate.

2. Tailoring of the network hardware structure and communication patterns to optimize network performance. The new techniques we used to design KLAT2's network allow us to optimize

network performance in general or even to specialize KLAT2's network for this CFD code. All runs reported here were performed with a generalized network rather than one specialized for this code.

3. Use of the Athlon's 3DNow! vector floating point instructions. Unfortunately, these instructions only support 32-bit precision for floating point arithmetic and also have various performance issues that make them difficult to use. However, we have developed compiler technology and other support that makes use of 3DNow! feasible. This is one of the key reasons that KLAT2 uses AMD Athlon processors.

Achieving high performance from this CFD code requires the first two approaches. The third approach, use of 3DNow!, we viewed with skepticism until we were able to verify that 3DNow! single-precision floating-point arithmetic has sufficient precision to ensure that the CFD results are valid.

The following subsections discuss KLAT2's special architectural characteristics, its network and support for 3DNow!, that we have used to improve the performance of this CFD code. The specific optimizations and general restructuring of the CFD are discussed in section 4.

3.1 Optimizing Network Performance

The cost and baseline performance of AMD Athlon processors is outstanding, and KLAT2's basic performance relies heavily upon that fact, but processors alone are not a supercomputer – a high-performance interconnection network is needed. Further, KLAT2 uses uniprocessor nodes instead of multiprocessor (shared-memory SMP) nodes for two key reasons: (1) using single-processor nodes eliminates inter-processor memory access conflicts, making full memory bandwidth available to each processor and (2) SMP Athlon PCs were not widely available at the time KLAT2 was built. This increases the importance of network performance and also increases the number of nodes, making the network larger and more expensive. Our solution to these network design problems is a new type of network: a "Flat Neighborhood Network" (FNN) [7].

This network architecture came from the realization that the switching fabric need not be the full width of the cluster in order to achieve peak performance. Using multiple NICs (Network Interface Cards) per PC, single-switch latency can be achieved by having each PC share at least one switch with each other PC – all PCs do not have to share the same switch. A switch defines a local network neighborhood, or subnet. If a PC has several NICs, it can belong to several neighborhoods. For two PCs to communicate directly, they simply use NICs that are in the same neighborhood. If two PCs have more than one neighborhood in common, they have additional bandwidth available with the minimum latency.

Before discussing how we design and use FNNs, it is useful to consider a small example. What is the best interconnection network design that can be built using four-way switches for an eight-PC cluster?

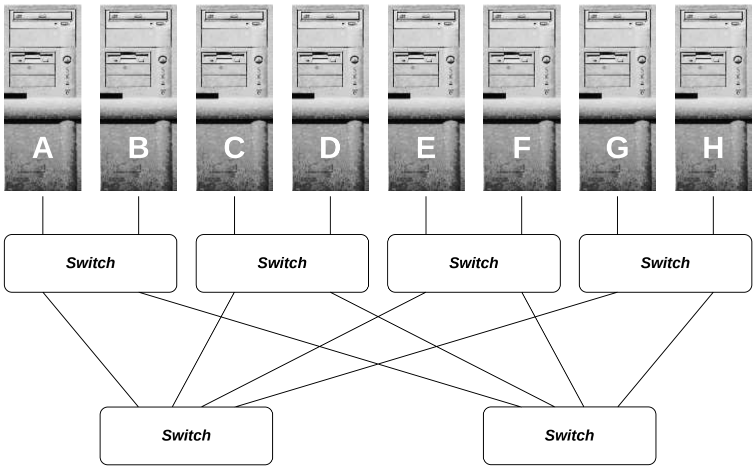

Figure 3: A fat tree network

A very popular answer is to use a fat tree [12], because fat trees easily provide the full bisection bandwidth. Unfortunately, most inexpensive switches cannot handle routing for a fat tree topology. Assuming that appropriate switches are available, the eight-PC network would look like figure 3.

For this fat tree, the bandwidth available between any pair of PCs is precisely that of one link; thus, we say that the pairwise bandwidth is 1.0 link bandwidth units. The bisection bandwidth of a network is determined by dividing the machine in half in the worst way possible and measuring the maximum bandwidth between the halves. Because the network is symmetric, it can be cut arbitrarily in half; the bisection bandwidth is maximal when all the processors in each half are sending in some pattern to the processors in the other half. Thus, assuming that all links are bidirectional, the bisection bandwidth is 8*1.0 or 8.0.

Pairwise latency also is an important figure of merit. For a cluster whose nodes are physically near each other, we can ignore the wire latency and simply count the average number of switches a message must pass through. Although some paths have only a single switch latency, e.g. between A and B, most paths pass through three switches. More precisely, from a given node, only 1 of each of the 7 other nodes can be reached with a single-switch latency. Thus, 1/7 of all pairs will have 1.0 switch latency and 6/7 will have 3.0 switch latency; the resulting average is (1.0 + 3.0*6)/7, or 2.7 switch latency units.

Instead of using a fat tree, suppose that we use a FNN to connect these same eight PCs with four-way switches. Unlike the fat tree configuration, the FNN does not connect switches to switches, so cheap, dumb, switches can be used. However, more NICs are needed. At least for 100Mb/s Ethernet, the cost savings in using dumber switches more than compensates for the larger number of NICs. In this case, each PC must have 3 NICs connected in a configuration similar to that shown by the switch numbers and colors in figure 4.

Unlike the fat tree, the FNN pairwise bandwidth is not the same for all pairs. For example, there are 3.0 link bandwidth units between A and B, but only 1.0 between A and C. Although

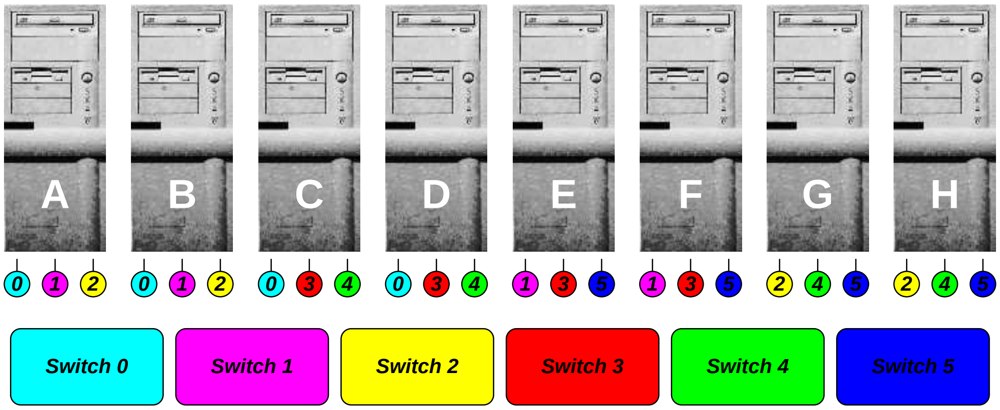

Figure 4: Flat neighborhood network

the FNN shown has some symmetry, FNN connection patterns in general do not have any basic symmetry that could be used to simplify the computation of pairwise bandwidth. However, no PC has two NICs connected to the same switch, so the number of ways in which a pair of connections through an S-port switch can be selected is S*(S-1)/2. Similarly, if there are P PCs, the number of pairs of PCs is P*(P-1)/2. If we sum the number of connections possible through all switches and divide that sum by the number of PC pairs, we have a tight upper bound on the average number of links between a PC pair. Because both the numerator and denominator of this fraction are divided by 2, the formula can be simplified by multiplying all terms by 2. In other words, the average pairwise bandwidth for the above FNN is ((4*3)*6)/(8*7), or about 1.28571428.

Not only does the average pairwise bandwidth of the FNN beat that of the fat tree, but the bisection bandwidth also is greater. Bisection bandwidth of a FNN is *very difficult* to compute because the definition of bisection bandwidth does not specify which communication pattern to use; for FNNs, the choice of pattern can dramatically alter the value achieved. Clearly, the best-case bisection bandwidth is the number of links times the number of processors; 8*3.0 or 24.0 in our case. If we assume that only pairwise permutation communication patterns are used, a *very conservative* bound can be computed as the number of processors times the average pairwise bandwidth; 8*1.28571428 or 10.28571428. However, pairwise permutation patterns do not generally yield the maximum bisection bandwidth for a given cut because they ignore bandwidth available using multiple NICs within each PC to send to distinct destinations at the same time. In any case, bisection bandwidth is significantly better than the fat tree's 8.0.

Even more impressive is the FNN design's pairwise latency: 1.0 as compared with 2.7 for the fat tree. No switch is connected to another, so only a single switch latency is imposed on any communication.

However, the biggest surprise is in the scaling. Suppose that we replace the six 4-way switches and eight PCs with six 32-way switches and 64 PCs? Simply scaling the FNN wiring pattern yields pairwise bandwidth of ((32*31)*6)/(64*63) or 1.47619047, significantly better than the 8 PC value of 1.28571428. FNN bisection bandwidth increases relative to fat tree performance by the same effect. Although average fat tree latency decreases from 2.7 to 2.5 with this scaling, it still cannot match the FNN's unchanging 1.0.

It also is possible to incrementally scale the FNN design in another dimension – by adding more NICs to each PC. Until the PCs run out of free slots for NICs, bandwidth can be increased with linear cost by simply adding more NICs and switches with an appropriate FNN wiring pattern. This is a far more flexible and cheaper process than adding bandwidth to a fat tree.

If FNNs are so great, why has it not been done before? There are four important reasons:

1. It only works well if fairly wide wire-speed switches are available; only recently have such switches become inexpensive.

2. Routing is not trivial; as a minimum, each machine must have its own unique routing table. Optimal routing using multiple NICs as a higher-bandwidth channel is conceptually like channel bonding [2], but requires a much more sophisticated implementation because this bonding is destination-sensitive (i.e., NICs may be used together when sending to one PC, but grouped differently when sending to another PC).

3. The network wiring pattern for a flat-neighborhood network is typically not symmetric and often has poor physical locality properties. This makes everything about the network, especially physical construction, somewhat more difficult.

4. It is not easy to design a wiring pattern that has the appropriate properties. For example, KLAT2's network interconnects 64 PCs using nine 31-way switches. Although 32-way switches would have made the design easier, we needed to reserve the 32nd port of each switch for the cluster's connections to the outside world.

We solved the last three problems by creating a genetic search algorithm (GA) [11] that can design an optimized network, print color-coded wiring labels, and construct the necessary routing tables. Although it was somewhat difficult to create the GA, and the execution time was sufficiently large that we actually run it on a cluster, the GA program is capable of optimizing the network design for any communication patterns or other characteristics specified – an important new capability beyond that of traditional networks. KLAT2's genetically designed FNN is shown in figure 5.

This version of KLAT2's FNN was partially optimized for another code (ScaLAPACK [3]) that uses communication patterns that are completely different from those of the CFD code discussed here – it was not optimized for this CFD. However, KLAT2's FNN uses its nine 31-way switches to yield average pairwise bandwidth of (((31*30)*8)+(8*7))/(64*63) or 1.859 bidirectional 100Mb/s Ethernet links/pair (371.8Mb/s per pair). Multiplying that by 32 PC pairs communicating across the bisection yields a *very conservative* bisection bandwidth of 11.9Gb/s; the uplink switch (which holds the two hot spares and the links to the outside world) adds an additional 1.8Gb/s of bisection bandwidth, for a total of 13.7Gb/s. The upper bound bandwidth on KLAT2's FNN without the

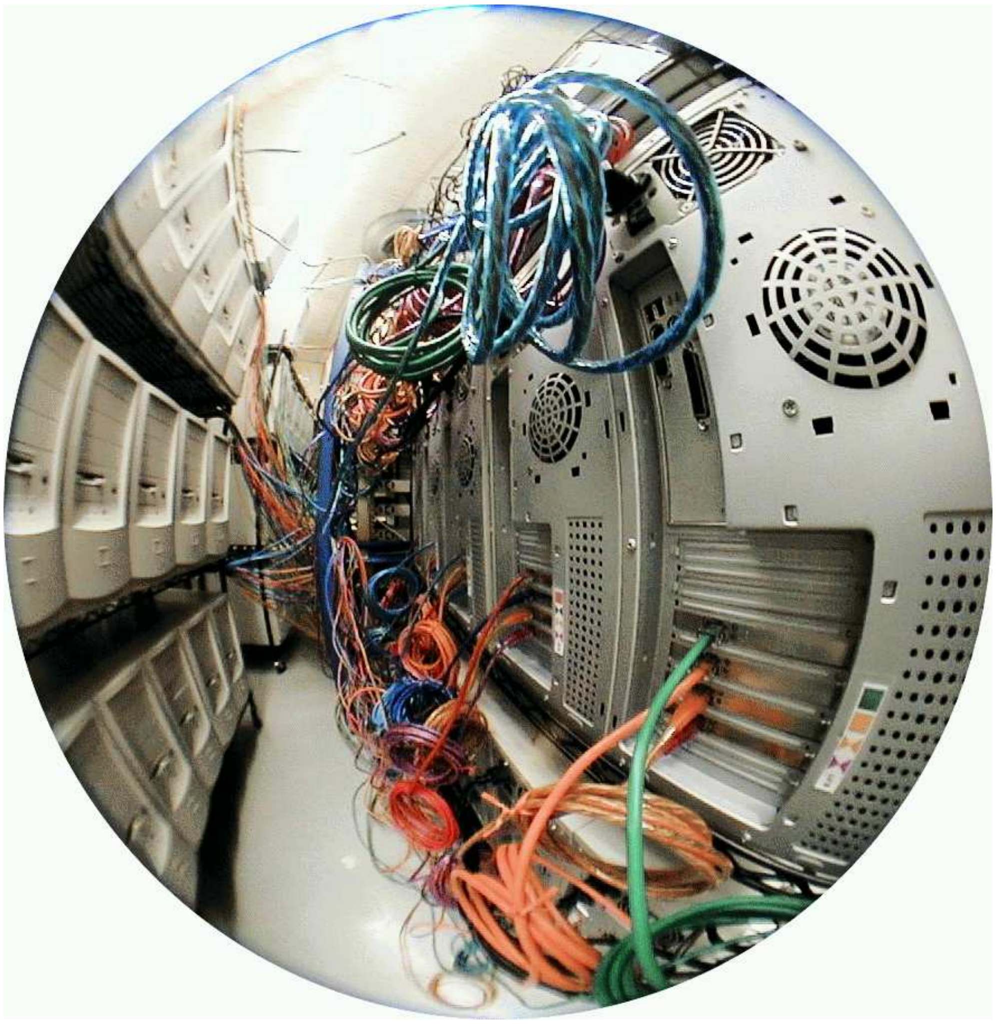

Figure 5: Physical wiring of KLAT2

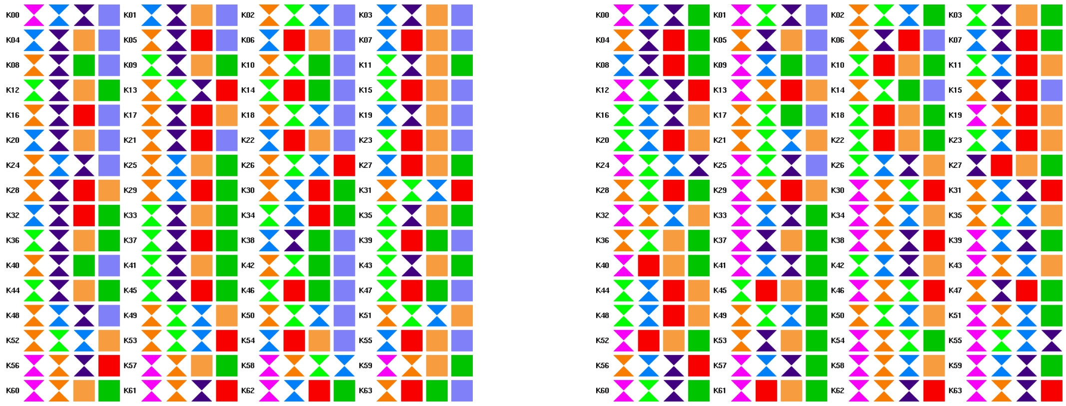

Figure 6: Comparison of FNN for ScaLAPACK (left) and DNSTool (right)

uplink switch is 25.6Gb/s. Our CFD program enqueues multiple communications at a time, making good use of parallel overlap in NIC operation and thus yielding performance much closer to the upper bound. The basic performance of KLAT2's network is sufficient to keep communication overhead under 20% of program runtime – despite the fact that the network was not optimized for this code.

Redesigning and physically rewiring KLAT2's FNN to be optimal for the CFD code is a relatively simple and quick process; we let the GA run for about two hours on a single 1GHz Athlon to design the network and the physical rewiring took two people a total of about 2.5 hours. Figure 6 shows KLAT2's color-coded wiring tags for the network partially optimized for ScaLAPACK (left) and the new network pattern optimized for DNSTool (right). The immediate result of this optimization was an additional speedup of about 1%; much more substantial performance improvements are possible by adjusting DNSTool's order of communications within a timestep and using FNN advanced routing[6]. Certainly, this rewiring and tuning of the communication code is appropriate if KLAT2 (or any other cluster) is dedicated to running this code. However, even with the extra pressure of using uniprocessor nodes, the basic FNN properties proved sufficient to efficiently handle the demands of this communication-intensive CFD code. Thus, throughout this paper, we have quoted the conservative "generic" FNN performance numbers for DNSTool.

At this writing, a version of the FNN design GA has been made freely available at http://aggregate.org/FNN/ via an interactive WWW form. In order to make the GA run fast enough for interactive use, the GA was simplified to use only a generic communication cost function. Once we have completed making the user interface more friendly, we plan to distribute the full GA as public domain source code.

3.2 SIMD Within A Register 3DNow! Optimizations

Over the past five years, the on-chip space available for microprocessor logic has reached the point where adding SIMD (Single Instruction Stream, Multiple Data Stream) or vector parallelism can be very cost effective. However, traditional SIMD and vector implementations rely on high-performance memory interfaces that simply are not compatible with current microprocessor datapaths. Thus, a new flavor of parallelism has been developed to mate SIMD semantics with conventional uniprocessor datapaths: something that we generically call SIMD Within A Register (SWAR) [5]. SWAR differs from SIMD in that SWAR parallelism partitions registers and datapaths into multiple fields that are processed simultaneously. Thus, SWAR requires much more constrained data layout, does not directly support disabling of processing elements, and has parallelism width that is not constant, but a function of the data size (e.g., twice as much parallelism with 16-bit data as with 32-bit data).

The concept of SWAR long predates the microprocessors with hardware supporting this type of execution. For example, population count (important because the population count of the XOR of two values is the Hamming distance) has long been implemented using a SWAR algorithm using bitmasks, shifts, and adds. However, there were no software tools developed for SWAR programming. The first SWAR hardware support, in processors like the Hummingbird PA-RISC and the MMX Pentium and K6, was very tightly focused on speeding a few hand-coded algorithms – especially MPEG decode for playing DVDs – so high-level programming models were not a concern. However, we had extensive experience in building optimizing compilers for SIMD machines, so we saw an immediate need to develop a better model and programming tools.

Thus, about 6 months before Intel began shipping the first Pentium MMX processors, we built our first high-level language and optimizing compiler technology for SWAR. Our high-level language, SWARC, is a vector C dialect supporting first-class arrays with arbitrary precision specified for each variable using the same syntax C uses for specifying precision of bit fields within a struct: int:2 i; declares i as an integer having not less than two-bit precision. To facilitate rewriting only a small portion of a C code in SWARC, the language is designed as a true C superset and is implemented by a module compiler, Scc. The compiler, which is still evolving, has been freely available from our SWAR WWW site, http://shay.ecn.purdue.edu/swar/, for over two years.

So that Scc's SWAR output code integrates cleanly with C code, Scc actually generates C code and invokes GCC on that. Because GCC does not know about any of the new SWAR instructions (i.e., MMX, 3DNow!, etc.), Scc generates GCC-compatible inline assembly macros for these instructions. These macros are actually quite high-level in that they allow direct access to the variables of the ordinary C code with which they are compiled, so they easily can be edited by hand to further tune performance. In fact, our macros for MMX, 3DNow!, and SSE are all freely available and often are used for coding from scratch. For example, to multiply corresponding elements of 2-element float arrays c=a*b, one could use the following 3DNow! macros:

```
movq_m2r(*((mmx_t *) &(a[0])), mm0); /* mm0 = (a[0],a[1]) */
pfmul_m2r(*((mmx_t *) &(b[0])), mm0); /* mm0 = (a[0]*b[0],a[1]*b[1]) */
movq_r2m(mm0, *((mmx_t *) &(c[0]))); /* (c[0],c[1]) = mm0 */
```

Despite the higher-level abstraction, with appropriate care in specifying the access of variables, each of the macros results in precisely one instruction. Of course, the code also will be faster if the float arrays start on 64-bit aligned addresses, the tight dependence structure of the sequence is interleaved with or near other code that the Athlon can use to fill pipeline slots, etc.

Scc uses quite a few very aggressive optimization techniques to ensure that the output code is fast, including a sophisticated pipeline timing model and use of a modified exhaustive search combined code scheduler, register allocator, and addressing mode selector. However, often the Scc-generated code can be improved somewhat by hand editing – for example, by inserting prefetch instructions or applying information about data values (e.g., integer compares can be used for float comparisons if the signs of the values are known). In some relatively rare cases, it is just as easy to write the parallel code directly using the macros as it would be using Scc. To aid in those cases, we also have developed a code rescheduling tool that can automatically reschedule sequences of the macros to improve pipelining.

The choice of Athlons for KLAT2 was largely inspired by our long experience with SWAR in general and with MMX, 3DNow!, and SSE in particular. Although Pentium III SSE theoretically offers comparable performance to that of 3DNow! on an Athlon, there are a variety of minor differences that make our compiler technology and benchmarking strongly favor 3DNow! on the Athlon. Aside from our compiler technology being far better tuned for 3DNow! than SSE, the AMD processors make pipeline bubbles less likely and less harmful. One reason is the difference between a single 128-bit SSE pipeline and two 64-bit 3DNow! pipelines; another is the more aggressive rescheduling done by the K6-2 and especially by the Athlon.

Although we view ScaLAPACK more as a library or benchmark than as a full code, we have created a 3DNow!-aware version that complies with the rules for the LINPACK benchmark as specified at http://www.top500.org/, and that code provides a good basis for performance comparison. Using 32-bit single-precision 3DNow!, KLAT2's 64 700MHz Athlons achieve a very impressive 64.459 GFLOPS on ScaLAPACK for N=40,960. (N1/2 was 13,824; see http://aggregate.org/KLAT2/ScaLAPACK/ for other details.) That translates to just over 1 GFLOPS per processor or 1.44 FLOPs/clock cycle, including all the communication overhead of ScaLAPACK using our FNN-aware version of LAM MPI. It also is less than $0.64 per MFLOPS.

The way we made ScaLAPACK 3DNow!-aware was very simple [8]. ScaLAPACK uses BLAS and BLACS, BLAS uses ATLAS[16], and BLACS uses MPI. Most of the runtime is actually inside a single ATLAS-created routine, SGEMM. ATLAS is a remarkable tool that actually constructs many different variations on that routine and automatically benchmarks them to select the best coding. Using our tools, it took us less than three days to modify the ATLAS-optimized SGEMM to make good use of 3DNow!. The DGEMM/SGEMM performance using a 900x900 element matrix on a single 700MHz Athlon system is shown in table 1.

In fact, if we could get the same performance for ScaLAPACK that we get for single-node

Table 1: DGEMM/SGEMM performance on a single 700MHz Athlon

	MFLOPS	FLOPs/clock
Athlon Legacy IA32 80/64-bit double precision	799.1	1.1416
Athlon Legacy IA32 80/32-bit single precision	926.1	1.3230
Athlon 3DNow! 32-bit single precision	1663.7	2.3767

SGEMM, KLAT2 would be under $0.39 per MFLOPS.

Although cache, TLB, and other effects prevent us from achieving the theoretical 3DNow! peak, the measured performance of the Athlon 3DNow! is substantially better than one can easily achieve with SSE. Even the IA32 legacy floating point performance is quite impressive given IA32's use of a stack model (rather than general registers) and the Athlon's GHz-enabling deep pipelines. The IA32 floating point register stack generally results in serial dependence chains that the processor must aggressively reschedule in order to fill pipeline slots well enough to achieve even one FLOP/clock.

Of course, using 3DNow! to speed-up a full-featured CFD code is much more difficult than a single subroutine of ScaLAPACK because performance depends on many more routines and the nature of the algorithm is substantially less cache-friendly. With our tools, three days of tuning was sufficient for SGEMM; it has taken us weeks to restructure the various CFD routines for 3DNow!.

4 Implementation and Optimization

We employed a variety of coding techniques to improve the computational efficiency of the solver. To guide our tuning, we performed computations on a 60 x 30 x 30 grid for 10 timesteps on a single processor. Using the gprof tool, we were able to exam the amount of CPU time spent on each set of routines. An example of the gprof output for the untuned DNSTool is presented in tables 2 and 3. These gprof usage profiles reveal that 90% of the CPU time is spent in the high-level routines that solve the viscous flux and perform the AUSM flux-splitting computations. Further examination revealed that among the lower-level routines that made up *viscous_flux* and *ausm_plus*, the primary subroutines were the *fill_uvwT_stencil*, *gradient*, and *ausm_plus_flux* routines. These three routines became the focus of our optimization efforts.

As an initial step, all of these routines were aggressively 'cleaned', focusing on removing all redundant calculations that reproduced work performed elsewhere. Repeated calculation of constant terms were removed, and wherever feasible, calculations were shifted outside the innermost loops. These steps created both a more streamlined and a more readable code, improving the usability of DNSTool.

A second key optimization came from restructuring the basic data storage of the variables to better match a cache-based memory system. The solver is employed in a series of sweeps, in the *i*-, *j*-, and *k*-directions, requiring the remapping of the 3-D data arrays into 1-D arrays corresponding

Table 2: Call hierarchy profile for initial double-precison DNSTool

index	%time	self	children	called	name
[1]	99.8	0.00	50.28		*main[1]*
		3.22	34.12	20/20	*viscous_flux[2]*
		2.25	7.76	20/20	*ausm_plus[3]*
		1.33	0.63	20/20	*steady_time_int[14]*
		0.04	0.33	1/1	*calc_metrics [17]*
		0.00	0.29	20/20	*apply_boundary_conditions [19]*
		0.00	0.21	20/20	*set_flux_diff [21]*
[2]	74.1	3.22	34.12	20	*viscous_flux [2]*
		6.51	0.00	36000/36000	*fill_uvwT_stencil2_dir3 [4]*
		5.54	0.00	34800/34800	*fill_uvwT_stencil2_dir2 [5]*
		5.24	0.00	352800/352800	*gradient[6]*
		4.50	0.00	17400/17400	*fill_uvwT_stencil_dir1 [7]*
		3.95	0.00	36000/36000	*fill_metrik_stencil_dir3 [8]*
		2.93	0.00	34800/34800	*fill_metrik_stencil_dir2 [10]*
		2.48	0.00	17400/17400	*fill_metrik_stencil_dir1 [13]*
		2.04	0.00	441000/882000	*add2flux_differences [8]*
		0.94	0.00	264600/705600	*get_1dline_3dfield [12]*
[3]	19.9	2.25	7.76	20	*ausm_plus [3]*
		2.92	0.00	88200/88200	*ausm_plus_flux [11]*
		2.04	0.00	441000/882000	*add2flux_differences [8]*
		1.56	0.00	441000/705600	*get_1dline_3dfield [12]*
		0.92	0.00	88200/88200	*get_1d_metric [15]*
		0.32	0.00	88200/88200	*normalize_1d [18]*
		0.00	0.00	60/60	*init_start_end_indices [53]*
		0.00	0.00	60/60	*get_normal_metric [52]*
		0.00	0.00	20/20	*get_max_dimension [62]*

Table 3: Profile of initial double-precision DNSTool

% time	cum. sec.	self sec.	calls	self ms/call	total ms/call	name
12.92	6.51	6.51	36000	0.18	0.18	*fill_uvwT_stencil2_dir3*
11.00	12.05	5.54	34800	0.16	0.16	*fill_uvwT_stencil2_dir2*
10.40	17.29	5.24	352800	0.01	0.01	*gradient*
8.93	21.79	4.50	17400	0.26	0.26	*fill_uvwT_stencil2_dir1*
8.08	25.86	4.07	882000	0.00	0.00	*add2flux_differences*
7.84	29.81	3.95	36000	0.11	0.11	*fill_metrik_stencil_dir3*
6.39	33.03	3.22	20	161.00	1867.12	*viscous_flux*
5.82	35.96	2.93	34800	0.08	0.08	*fill_metrik_stencil_dir2*
5.80	38.88	2.92	88200	0.03	0.03	*ausm_plus_flux*
4.96	41.38	2.50	705600	0.00	0.00	*get_1dline_3dfield*
4.92	43.86	2.48	17400	0.14	0.14	*fill_metrik_stencil_dir1*
4.47	46.77	2.25	20	112.50	500.38	*ausm_plus*
2.64	47.44	1.33	20	66.50	98.00	*steady_time_int*
1.83	48.36	0.92	88200	0.01	0.01	*get_1d_metric*
1.09	48.91	0.55	5	110. 00	110.00	*compute_local_dts*

to a single direction. The original data layout consisted of a separate 3-D array for each of the flow and geometric variables. When remapping from these 3-D arrays, the full set of the variables at one 3-D index is required to generate the values at one index point in the 1-D arrays. On a cache-based memory system, this would require fetching a distinct cache-line for *each* variable read. In the i-direction, the rest of these fetched cache-lines contain data elements that will be used in subsequent iterations (i+1, i+2, etc.). Unfortunately, in the other two directions (j and k), the rest of the data in the fetched cache-lines will not be used, wasting most of the memory bandwidth.

The new data layout is a 3-D array of structures, with each structure holding the relevant variables for one 3-D index point. Thus, when remapping from the 3-D array of structs, all the variables needed to calculate the values for one index point in the 1-D array are adjacent in memory. This results in only one or two cache-lines being fetched from main memory for the *entire* set of flow and geometry variables per 3-D index, dramatically reducing the required memory bandwidth. Additionally, when using single-precision floats, the fields within the structure were ordered so that pairs of adjacent fields could be loaded into 3DNow! registers with individual load instructions, see figure 7. The pairs were selected so that the calculations on them also could be performed in parallel. As shown in the figure, this new data vector has the form $[\rho \cup \rho u, \rho v \cup \rho w, \rho e]$. Combined with some effort to align the data storage with the 64-byte cache-lines of the Athlon, this more efficient use of the memory and cache effected across-the-board improvements whether using double-precision or single-precision.

The most costly routine in DNSTool was the *fill_uvwT_stencil*, whose purpose was to extract

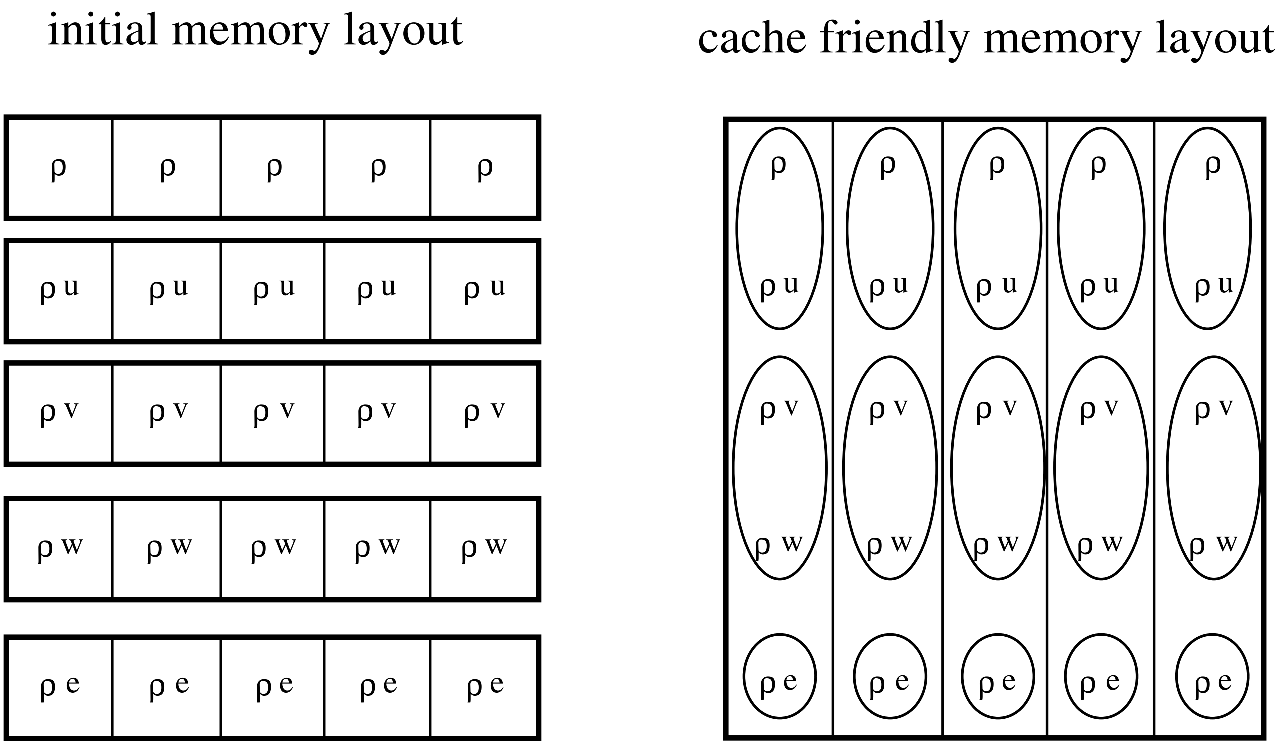

Figure 7: Initial memory layout and improved cache friendly memory layout

the flow variables from 3D arrays into 1D arrays (corresponding to the *i*-, *j*-, or *k*-directions), calculate the primitive variables (ρ, u, v, w, T) from the conservative variables $(\rho, \rho u, \rho v, \rho w, \rho e)$, and compute the intermediate values needed to for the flow gradient calculation on each face as per equation 33. This routine is used repeatedly, with slightly different forms depending on the direction of the computational sweep. We aggressively rewrote the computations in terms of 3DNow!, taking particular advantage of the fast reciprocal for the ρ-division and our new data structure. Efforts to implement pre-fetching of the variables only yielded minimal improvements- the Athlon scheduler is apparently good enough that only extremely aggressive pre-fetching will improve on its performance. Overall, these combined enhancements resulted in a sevenfold reduction in the single processor CPU time in the long-stride directions (*j* and *k*), and a fivefold reduction in the *i*-direction.

Improvement to the subroutine *gradient* was achieved in much the same manner as the *fill_uvwT_stencil* routines, through the combination of the improved data structure and converting the calculation of the finite-volume circulation theorem to 3DNow! macros. Equation 33 illustrates why the volume was included in the flow variable vector of the new data structure, as the volume is a component in many of the flow variable computations.

The final routine that was aggressively optimized with 3DNow! macros was *ausm_plus_flux*. In this routine, two important features were the fast reciprocal square root function (particularly useful for calculating the speed of sound, $c = \sqrt{\gamma RT}$), and replacing if-then structures with bit-masking techniques (for instance in the choice of M^p and M^m definition in equation 24) .

It should be noted that the 3DNow! coding done in DNSTool was strictly in the form of C-macros; no assembler code was explicitly included in the code. Neither were the Athlon extensions to 3DNow! applied to the code; the 3DNow! code we used is compatible with the K6-2 as well as Athlon processors. Although the SWARC compiler was useful as a reference starting point for 3DNow! coding of DNSTool routines, as it turned out, we did not directly use any code generated by it. We did use our code rescheduler to improve some of the 3DNow! code in DNSTool. Throughout the 3DNow code, care was taken to optimize register usage; 3DNow! has only 8 register names, but careful use of these 8 allows the Athlon's register renaming logic to make better use of its 88 physical registers. The new data structure also improves register use by reducing the pressure on the IA32 registers that are used for addressing. The vector-array allows a single base address to be kept in an IA32 register and indexed by short constants instead of forcing the IA32 registers to juggle six or more apparently independent pointers, one for each individual variable array.

The results of this effort, including the speed-up due to switching from double-precision to single-precision floating-point operations and including 3DNow! macros, are show in tables 4 and 5. As can be seen, the *fill_uvwT_stencil* routines have dropped precipitously to the 9, 10, and 11th most CPU-costly routines. The *gradient* subroutine is now the most costly, but its CPU time has fallen from 5.24 seconds to 1.73 seconds. Numerous routines in which no 3DNow! optimizations were used saw speed-ups due to the aggressive removal of redundant calculations, the new data structure, and the improved cache-memory alignment. The overall computational time fell by a factor of 3, from 50.4 to 17.7 seconds.

For additional comparison, we have included in tables 6 and 7 the double-precision results, which eliminate the advantages of moving from 64-bit to 32-bit variables, including making the 3DNow! macros useless. As can be seen, there is considerable improvement in the code simply from 'cleaning', cache-memory alignment, and the new data structure.

Table 4: Call hierarchy profile for optimized 3DNow! DNSTool

index	%time	self	children	called	name
[1]	98.2	0.00	15.82		*main [1]*
		0.06	8.48	20/20	*viscous_flux [2]*
		0.38	3.76	20/20	*ausm_plus [3]*
		1.08	0.31	20/20	*steady_time_int [5]*
		0.76	0.00	176400/176400	*add2flux_difference*
[2]	53.0	0.06	8.48	20	*viscous_flux [2]*
		1.58	0.00	36000/36000	*fill_metrik_stencil_dir3* [4]
		1.23	0.00	34800/34800	*fill_metrik_stencil_dir2 [7]*
		1.16	0.00	88200/88200	*stress_flux [8]*
		0.90	0.00	36000/36000	*fill_uvwT_stencil_dir3 [10]*
		0.87	0.00	17400/17400	*fill_metrik_stencil_dir1 [11*]
		0.82	0.00	17400/1740	*fill_uvwT_stencil_dir1* [12]
		0.81	0.00	34800/34800	*fill_uvwT_stencil_dir2* [13]
		0.76	0.00	352800/352800	*gradient [14]*
		0.33	0.00	264600/264600	*scopy_3dnow [19]*
		0.02	0.00	264600/264600	*get_1dline_3dfield [33]*
[3]	25.7	0.38	3.76	20	*ausm_plus [3]*
		1.37	0.00	88200/88200	*muscl_extrapolation [6]*
		1.13	0.00	88200/88200	*ausm_plus_flux [9]*
		0.51	0.00	88200/88200	*get_1d_metric [16]*
		0.46	0.00	88200/88200	*get_1dline_3dvars [17]*
		0.29	0.00	88200/88200	*vn_extrapolation [20]*
		0.00	0.00	60/60	*init_start_end_indices [58]*
		0.00	0.00	20/20	*get_max_dimension [6*5]

Table 5: Profile of optimized 3DNow! DNSTool

% time	cum. sec.	self sec.	calls	self ms/call	total ms/call	name
9.81	1.58	1.58	36000	43.89	43.89	*fill_metrik_stencil_dir3*
8.50	2.95	1.37	88200	15.53	15.53	*muscl_extrapolation*
7.64	4.18	1.23	34800	35.34	35.34	*fill_metrik_stencil_dir2*
7.20	5.34	1.16	88200	12.81	12.81	*stress_flux*
7.01	6.47	1.13	88200	12.81	12.81	*ausm_plus_flux*
6.70	7.55	1.08	20	54000.00	69500.00	*steady_time_int*
5.59	8.45	0.9	3600	25.00	25.00	*fill_uvwT_stencil_dir3*
5.40	9.32	0.87	17400	50.00	50.00	*fill_metrik_stencil_dir1*
5.09	10.14	0.82	17400	47.13	47.13	*fill_uvwT_stencil_dir1*
5.03	10.95	0.81	34800	23.28	23.28	*fill_uvwT_stencil_dir2*
4.72	11.71	0.76	352800	2.15	2.15	*gradient*
4.72	12.47	0.76	176400	4.31	4.31	*add2flux_differences*
3.17	12.98	0.51	88200	5.78	5.78	*get_1d_metric*
2.86	13.44	0.46	88200	5.22	5.22	*get_1dline_3dvars*
2.61	13.86	0.42	88200	4.76	4.76	*normalize_1d*
2.36	14.24	0.38	20	19000.00	207000.00	*ausm_plus*
2.05	14.57	0.33	264600	1.25	1.25	*scopy_3dnow*
1.80	14.86	0.29	88200	3.29	3.29	*vn_extrapolation*
1.61	15.12	0.26	5	52000.00	52000.00	*compute_local_dts*
1.30	15.33	0.21	20	10500.00	10500.00	*set_flux_diff*

Table 6: Call hierarchy profile for optimized double-precision DNSTool

index	%time	self	children	called	name
[1]	99.5	0.00	26.23		*main [1]*
		0.07	15.39	20/20	*viscous_flux [2]*
		0.43	7.62	20/20	*ausm_plus [3]*
		1.25	0.47	20/20	*steady_time_int [10]*
[2]	58.7	0.07	15.39	20	*viscous_flux [2]*
		2.56	0.00	36000/36000	*fill_uvwT_stencil_dir3 [5]*
		2.01	0.00	34800/34800	*fill_uvwT_stencil_dir2 [7]*
		1.95	0.00	36000/36000	*fill_metrik_stencil_dir3 [8]*
		1.86	0.00	352800/352800	*gradient [9]*
		1.61	0.00	17400/17400	*fill_uvwT_stencil_dir1 [11]*
		1.56	0.00	34800/34800	*fill_metrik_stencil_dir2 [12]*
		1.33	0.00	17400/17400	*fill_metrik_stencil_dir1 [14]*
		1.18	0.00	88200/88200	*stress_flux [15]*
		1.01	0.00	88200/176400	*add2flux_differences [6]*
		0.32	0.00	264600/264600	*get_1dline_3dfield [21]*
[3]	30.6	0.43	7.62	20	*ausm_plus [3]*
		2.83	0.00	88200/88200	*ausm_plus_flux [4]*
		1.40	0.00	88200/88200	*muscl_extrapolation [13]*
		1.01	0.00	88200/176400	*add2flux_differences [6]*
		0.97	0.00	88200/88200	*get_1dline_3dvars [16]*
		0.86	0.00	88200/88200	*get_1d_metric [17]*
		0.33	0.00	88200/88200	*normalize_1d [20]*
		0.22	0.00	88200/88200	*vn_extrapolation [23]*
		0.00	0.00	60/60	*init_start_end_indices [59]*
		0.00	0.00	20/02	*get_max_dimension [65]*

Table 7: Profile of optimized double-precision DNSTool

% time	cum. sec.	self sec.	calls	self ms/call	total ms/call	name
10.74	2.83	2.83	88200	32.09	32.09	*ausm_plus_flux*
9.72	5.39	2.56	36000	71.11	71.11	*fill_uvwT_stencil_dir3*
7.67	7.41	2.02	176400	11.45	11.45	*add2flux_differences*
7.63	9.42	2.01	34800	57.76	57.76	*fill_uvwT_stencil_dir2*
7.40	11.37	1.95	36000	54.17	54.17	*fill_metrik_stencil_dir3*
7.06	73.23	1.86	352800	5.27	5.27	*gradient*
6.11	14.84	1.61	17400	92.53	92.53	*fill_uvwT_stencil_dir1*
5.92	16.40	1.56	34800	44.83	44.83	*fill_metrik_stencil_dir2*
5.31	17.80	1.40	88200	15.87	15.87	*muscl_extrapolation*
5.05	19.13	1.33	17400	76.44	76.44	*fill_metrik_stencil_dir1*
4.74	20.38	1.25	20	62500.00	86000.00	*steady_time_int*
4.48	21.56	1.18	88200	13.38	13.38	*stress_flux*
3.68	22.53	0.97	88200	11.00	11.00	*get_1dline_3dvars*
3.26	23.39	0.86	88200	9.75	9.75	*get_1d_metric*
1.63	23.82	0.43	20	21500.00	402500.00	*ausm_plus*
1.37	24.18	0.36	5	72000.00	72000.00	*compute_local_dts*
1.25	24.51	0.33	88200	3.74	3.74	*normalize_1d*
1.21	24.83	0.32	264600	1.21	1.21	*get_1dline_3dfield*
1.02	25.10	0.27	20	13500.00	13500.00	*set_flux_diff*

5 Results

The primary objective of this paper is to demonstrate the computational improvements yielded from the implementation of the various techniques detailed in the previous sections. The simulation results obtained so far are not sufficient for drawing conclusions about the flow physics in the low-pressure turbine; however, CFD results presented do indicate the complexity and realism of the engineering problem involved in this project and potential for performing CFD simulations on KLAT2-like clusters. The cost of KLAT2 is then itemized. Combining the runtimes obtained with the cost of building KLAT2 demonstrates the cost effectiveness of our approach.

5.1 Simulation Results

The particular simulation we have focused on for this paper is the direct numerical simulation of the flow over a single turbine blade. This research is connected to NASA's Low-Pressure Turbine Physics program, which aims to improve the efficiency of turbomachinery in aircraft engines. Our specific concentration is understanding the turbulent cascade, *i.e.* the process by which the flow over the blade switches from laminar to turbulent flow conditions. This transition is not well-understood, but its properties are often critical to the performance of many aerospace systems, including turbomachinery.

Fundamental understanding of the cascade through numerical simulation requires the use of computationally-intensive models such as LES and DNS, since the more empirically based, less-demanding approaches such as RANS cannot properly model laminar-turbulent transition. For the full turbine cascade, complex geometric effects must be taken into account, requiring dense grids in certain critical regions. For these reasons, the turbine blade flow is an ideal candidate for testing on a cluster using DNSTool.

The selected grid configuration for the CFD computation has been used in both two-dimensional and three-dimensional research at the University of Kentucky on the turbine cascade [15]. As can be seen in figure 8, the grid is curvilinear, representing an inflow region from the nozzle into the turbine, through the turbine blades, and then an outflow region. The overall grid is 400 x 200 x 200, or 16 million grid points. The flow over the blade is considered periodic in both the *j*- and *k*-directions, with the exception of the *j*-direction between the two solid blade boundaries. The flow is subsonic throughout, with a transitional region occurring somewhere on the blade surface depending on the precise flow parameters.

For the purposes of timing, we ran steady-state simulations of the flow. The results of these simulations are presented in figures 9 and 10 for an inflow Mach number of 0.1. This solution combined with homogeneous turbulence is an initial condition for unsteady turbulent simulations. The results for the unsteady simulation are as of yet preliminary; even with the superior performance of KLAT2, a single unsteady simulation would require eight weeks of dedicated usage.

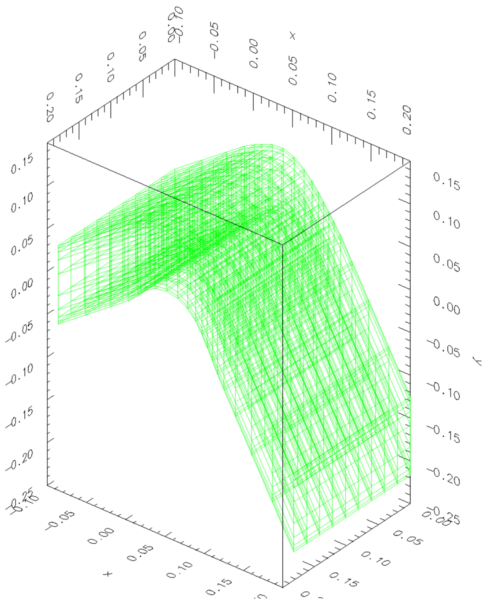

Figure 8: View of the three-dimensional grid

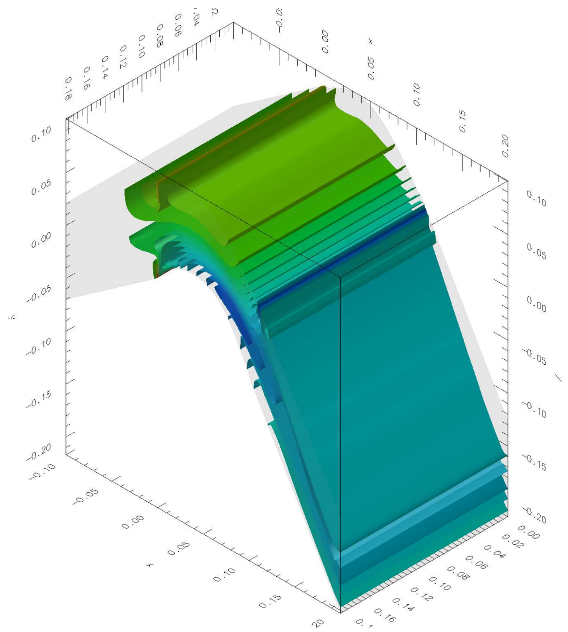

Figure 9: Steady state pressure contours

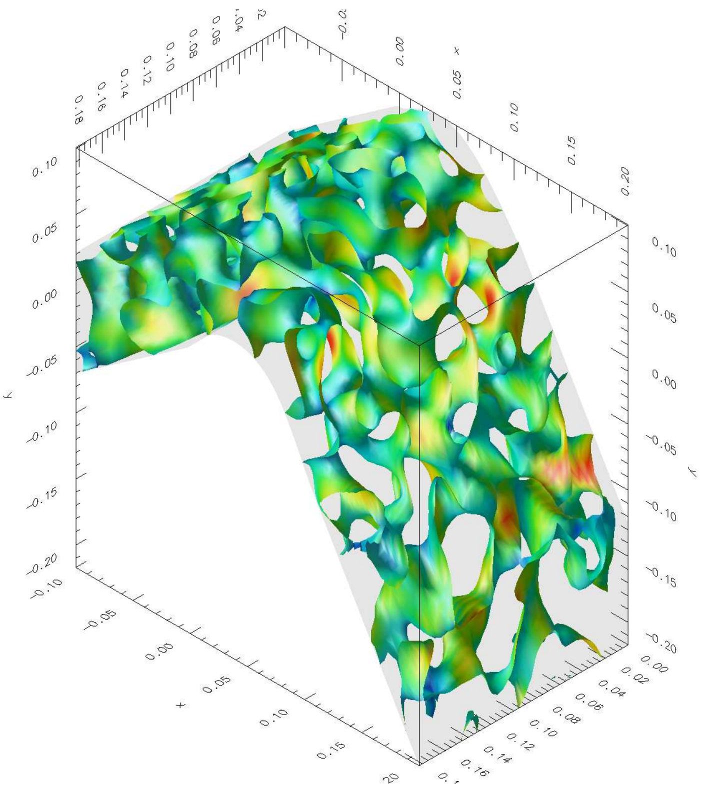

Figure 10: Isosurface of velocity disturbance magnitude colored with pressure

A technical point is that the grid used in these sample calculations is probably not sufficiently fine for a strict DNS; rather, the presented results are more properly QDNS. Evidence suggests that our grids are within a factor of 5 of a strict DNS computation–for comparison, the Flow Physics and Computation Division of Stanford University claims a DNS computation for a similar problem domain using 86 million grid points (http://www-fpc.stanford.edu/).

5.2 The Cost of KLAT2

KLAT2's cost is somewhat difficult to specify precisely because the most expensive components, the Athlon processors, were donated by their manufacturer, AMD (Advanced Micro Devices). In the interest of fairness, we have quoted the retail price for these processors as found on Multiwave's WWW site on May 3, 2000. Similarly, although most applications (including those discussed in this paper) use only 64 nodes, KLAT2 also has 2 "hot spare" nodes and an additional switch layer that are used for fault tolerance and system-level I/O; because we consider these components to be an integral part of KLAT2's design, we have included their cost. We also included 16 spare NICs and several spare surge protectors. Due to University of Kentucky purchasing guidelines and part stocking issues, purchases from the same vendor were sometimes split in odd ways and there were various inconsistencies about how shipping was charged; although the vendor totals are correct, we have had to approximate the component cost breakdown in these cases.

In summary, KLAT2's total value is about $41,200, with the primary costs being roughly $13,200 in processors, $8,100 in the network, $6,900 in motherboards, and $6,200 in memory. We believe the cost breakdown in figure 11 to be a conservative upper bound on the full system cost.

Some people have taken issue with our accounting of the assembly cost of KLAT2 because of the large number of student volunteers. However, people often forget that, as a university, it is our job to inspire and train students. In fact, we could have purchased the PCs as assembled systems without significant extra cost, but buying components allowed us to build precisely the configuration we wanted and, more importantly, was pedagogically the right thing to do. Few projects inspire students like being able to contribute to the construction of a new type of supercomputer, so we encouraged as many people as possible to participate. The construction of the PCs was easily accomplished within a single day (April 11, 2000) with no more than six students working at any time and most of the students having their first experience in PC construction; wiring the cluster took two people only a few hours (thanks to the color-coded cables). The cost of the soda and pizzas may seem a flippant accounting, but are actually a good match for what it would have cost for a well-experienced research assistant to assemble everything without additional help or distractions.

AMD, `http://www.amd.com/`	$13,200
66 Donated 700MHz Athlon OEM processor modules @ ~$200	
MemoryX, `http://memoryx.com/`	$6,182
66 128MB PC100 CAS2 SDRAMs @ $93	
Technology Partners, `http://www.tpi-us.com/`	$5,455
66 Polaris II ATX Mid Towers 300W @ $58	
66 Sony 1.44MB Floppy Drives (for net boot) @ $11	
Multiwave Technology, `http://mwave.com/`	$14,338
66 FIC SD11 Motherboards @ $104	
10 Smartlink 32-port wire-speed 100Mb/s switches @ $527	
28 Smartlink 100Mb/s NIC 10-packs @ $80	
Buy.Com, `http://buy.com/`	$706
32 Hawking 15’ color-coded Cat.5 cable 5-packs @ $9	
32 Hawking 15’ transparent color-coded Cat.5e cable 5-packs @ $12	
Coolerstar, `http://coolerstar.com/`	$607
66 AMD K7/PII Dual Fans (CPU heat sinks & fans) @ $5	
66 DC Fans 80mm (extra case fans) @ $4	
Lowe’s	$256
4 48"x18"x72" black wire-frame shelves @ $64	
Wal-mart	$109
2 WindDance fans (to direct airflow between shelves) @ $15	
20 Surgestrip model 201 surge protectors @ $4	
Various local stores	$352
16 3" diameter threaded-mount wheels for shelves @ $9	
16 Pizzas for student helpers @ $10	
4 Cases of soda for student helpers @ $7	
4 2" diameter threaded-mount wheels for rack @ $7	
Available at no cost/indirectly used items	
1 10-year-old rack & mounting hardware	
1 Surplus 17" monitor used for cluster status	
1 Old PCI video card used for cluster status	
1 18GB EIDE disk drive	
66 Set of inkjet-printed labels for each node	
2 Other clusters for KLAT2’s HW design and SW development	
Total	$41,205

Figure 11: Cost breakdown for KLAT2

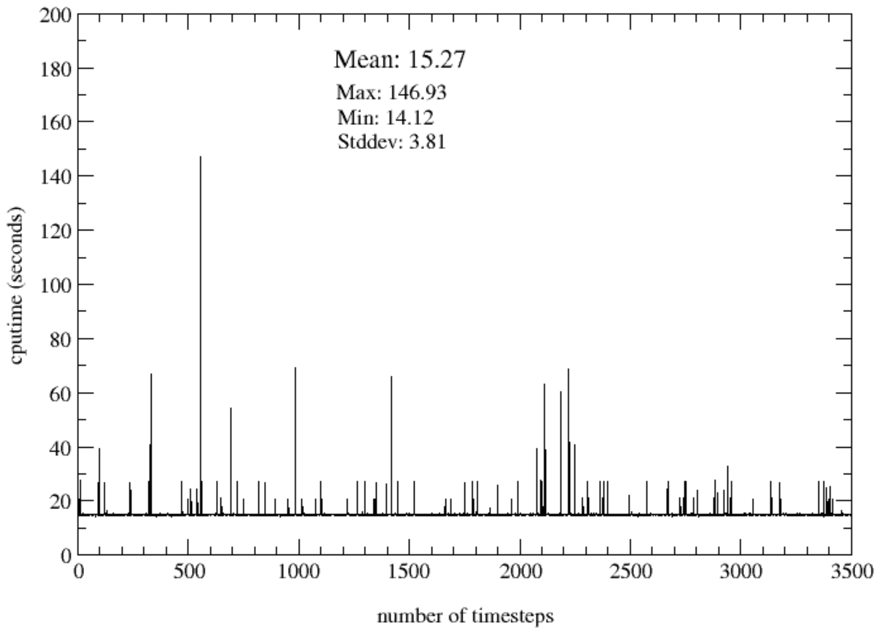

Figure 12: Distribution of the runtimes over a 15h run for the single precision simulation using 3DNow!

Table 8: Final double precision results

I/O	MFLOP count	Walltime	Sustained GFLOPS	$/MFLOPS
with I/O	338237	23398.8 s	14.46	2.85
without I/O	338237	22560.0 s	14.99	2.75

Table 9: Final single precision results (using 3DNow!)

I/O	MFLOP count	Walltime	Sustained GFLOPS	$/MFLOPS
with I/O	338237	16114.9 s	20.9	1.96
without I/O	338237	15273.2 s	22.1	1.86

5.3 Computational Results

The current GFLOPS and $/MFLOPS are given in tables 8 and 9, while figure 12 shows the distribution of the wall-clock times over 3500 timesteps. The relatively rare very slow timesteps occur due to an unfortunate resonance between minor load imbalances and TCP/IP network traffic; it should be possible to eliminate these glitches, but their contribution to total runtime was not sufficient for us to make that a priority.

The presented computations are for a DNSTool run on a 400 x 200 x 200 turbine blade grid (16 million grid points). This grid point count excludes points outside the physical flow region needed for boundary conditions or block overlap. The presented results are the average for 1000 timesteps for a long time integration. The walltime is maximum over the whole cluster for each iteration. All of the computations are double precision on the 64-processor KLAT2. Results are presented for both the isolated numerical computation and the overall computation including I/O.

The GFLOP count is measured by first doing the identical computation on a SGI Origin 2000 and using SGI performance tools; specifically, using *mpirun -np 64 ssrun -ideal* to run the code, followed by applying *prof -archinfo ideal.** to the output, which yields the total FLOP count for the computation. Dividing this number by the walltime expended in the KLAT2 simulation yields the given GFLOP/s results.

We do not think that we have yet achieved optimal performance for this cluster-CFD code combination. In addition to the potential improvements discussed in section 3, we have not done much work on the optimal grid configurations for this cluster, either in terms of the construction and load distribution of the subgrid blocks or the overall best grid density. The I/O also remains a relatively slow part of the code, playing a large role in the difference between the overall and peak performance. Even without these potential improvements, the overall performance of DNSTool on KLAT2 is outstanding.

6 Conclusion

In this paper, we have described the techniques and tools that we created and used to optimize the performance of the combination of DNSTool and KLAT2. This tuning of system performance involved both recoding of the application and careful engineering of the cluster design. Beyond restructuring of the code to improve cache behavior, we used various tools to incorporate 3DNow! into the single-precision version, and even used a GA to design the cluster network.

Using similar techniques for KLAT2 to execute ScaLAPACK yielded better than $0.64 per MFLOPS single precision, but ScaLAPACK is a relatively easy code to speed up. Although ScaLAPACK requires substantial network bandwidth, only a single computational routine (DGEMM or SGEMM) needed to be optimized to obtain that record performance. In contrast, achieving $2.75 per MFLOPS double precision and $1.86 per MFLOPS single precision for DNSTool requires many more optimizations to many more routines and generally taxes the machine design much more severely.

The high computational complexity of the DNS approach to CFD provides important and unique abilities: DNS yields high-quality results for problems that faster, more approximate, CFD techniques cannot yet handle. Thus, the low cost of running DNS on machines like KLAT2 can make a qualitative difference in the range of CFD problems that can be approached, either to directly solve them or to design computationally cheaper models that can suffice for practical engineering applications.

In much the same way, the techniques and tools that we developed and used to engineer KLAT2, and to tune the performance of DNSTool for KLAT2, also represent qualitative advances over previous approaches. For example, 3DNow! was originally intended for 3D graphics in video games, but our tools make it significantly more accessible for scientific and engineering codes. Even if it is not optimized for the particular communication patterns that will be used, the GA-designed FNN provides low latency and high bisection bandwidth; the ability to tune for a specific application's communication patterns, generating deliberately asymmetric designs when appropriate, is a unique additional benefit.

Most importantly, the optimized DNSTool and the tools to design and similarly optimize your own clusters with specific applications either are or will be freely available from http://aggregate.org/. Our primary direction for future work is to take these tools one giant step further, creating easily-replicated personalized turnkey superclusters (PeTS) that will provide scientists and engineers with supercomputer performance for specific applications without requiring them to become experts in computing.

7 Acknowledgements

Thomas Hauser is supported by NASA grant NAG3-2099, sponsored by the NASA-Glenn Research Center and the Center for Computational Science at the University of Kentucky. Ray LeBeau is supported by Kentucky NASA-EPSCoR. George Huang is supported by NASA grant NAG3-2099, sponsored by the NASA-Glenn Research Center. The initial work on DNSTool was supported by SFB 255 at the University of Technology Munich under Rainer Friedrich.

Special thanks to AMD for their generous contributions of the Athlon processors for KLAT2 and to the many students who got their first taste of cluster supercomputing amongst the rubble of discarded packing material from which KLAT2 rose on April 11, 2000.

References

[1] Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings, International Conference on Parallel Processing*, 1995.

[2] Donald J. Becker, Thomas Sterling, Daniel Savarese, Bruce Fryxell, and Kevin Olson. Communication overhead for space science applications on the beowulf parallel workstation. In *Proceedings, High Performance and Distributed Computing*, 1995.

[3] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[4] H. G. Dietz, W. E. Cohen, T. Muhammad, and T. I. Mattox. Compiler techniques for fine-grain execution on workstation clusters using PAPERS. In *Proceedings of the Seventh International Workshop on Programming Languages and Compilers for Parallel Computing*. Cornell University, August 1994.

[5] H. G. Dietz and R. J. Fisher. Compiler optimizations for SIMD within a register. In *Languages and Compilers for Parallel Computing*. Springer-Verlag, New York, 1999.

[6] H. G. Dietz and T. I. Mattox. Compiler techniques for flat neighborhood networks. In *Proceedings of the International Workshop on Programming Languages and Compilers for Parallel Computing, New York*, August 2000.

[7] H. G. Dietz and T. I. Mattox. KLAT2's flat neighborhood network. In *Proceedings of the Extreme Linux track in the 4th Annual Linux Showcase, Atlanta, GA*, October 2000.

[8] Hank Dietz and Tim Mattox. Inside the KLAT2 supercomputer: The flat neighborhood network & 3DNow! Ars Technica, http://arstechnica.com/cpu/2q00/klat2/klat2-1.html, June 2000.

[9] Th. Hauser. *Hyperschallströmung um stumpfe Kegel bei externen wellenartigen Störungen*. PhD thesis, Technische Universität München, 1998.

[10] Th. Hauser and R. Friedrich. Hypersonic flow around a blunt nose cone under the influence of external wave-like perturbations. In *GAMM Jahrestagung, Regensburg*, March 1997.

[11] J. Holland. *Adaptation in Natural and Artificial Systems*. PhD thesis, University of Michigan, Ann Arbor, MI, 1975.

[12] Charles E. Leiserson. Fat-trees: Universal networks for hardware efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, October 1985.

[13] M.-S.L. Liou and Ch.J. Steffen Jr. A new flux splitting scheme. *Journal of Computational Physics*, 107:23–39, 1993.

[14] R. Radespiel and N. Kroll. Accurate flux vector splitting for shocks and shear layers. *Journal of Computational Physics*, 121:66–78, 1995.

[15] Y.B. Suzen, G. Xiong, and P.G. Huang. Predictions of transistional flows in a low-pressure turbine using an intermittency transport equation. In *AIAA Denver Fluids 2000 Conference*, number 2000-2654, June 2000.

[16] R. Whaley and J. Dongarra. Automatically tuned linear algebra software. LAPACK Working Note No.131 UT CS-97-366, University of Tennessee, 1997.

[17] G. Xiong. Optimizing thermal performance for design of advanced power electronics assembly. In *IMAPS2000, Boston*, September 2000.

[18] G. Xiong, P.G.Huang, C.A. Saunders, and P.J. Heink. CFD simultion of laser printhead - a case study on noise reduction. In *Third International Symposium on Scale Modeling (ISSM3)*, September 2000.