Massively parallel visualization on linux clusters with Rocketeer Voyager

Robert A. Fiedler[1] & John C. Norris[1]
[1]*Center for Simulation of Advanced Rockets, University of Illinois at Urbana-Champaign*

Abstract

This paper describes Rocketeer Voyager, a versatile 3-D scientific visualization tool which processes in parallel a series of HDF output dumps from a large scale simulation. Rocketeer reads data defined on many types of grids and displays translucent surfaces and isosurfaces, vectors as glyphs, surface and 3-D meshes, etc. An interactive version is first used to view a few snapshots, orient the image, and save a set of graphical operations that will be applied by the batch tool to the entire set of snapshots. The snapshots may reside on separate or shared file systems. Voyager broadcasts the list of operations, reads the data ranges from all snapshots to determine the color scales, and then processes the snapshots concurrently to produce hundreds of frames for animation in little more time than it would take for the interactive tool to display a single image.

Rocketeer is based on the Visualization Toolkit, which uses OpenGL to exploit hardware graphics acceleration. On a linux cluster the nodes may have different graphics hardware or none at all. To circumvent this difficulty and to eliminate some defects in images rendered using Mesa, a free implementation of OpenGL, CSAR contracted with Xi Graphics, Inc. to develop an efficient and flawless X-server that runs in a "virtual frame buffer" and installs on all nodes in the cluster with little administrator intervention. Because the most time-consuming operation is reading the data files, the total time for generating an individual image is not much longer than it is on the fastest graphics workstations, despite the fact that the virtual frame buffer does not take advantage of graphics hardware acceleration.

1. Introduction

Rocketeer [1] is a powerful interactive 3-D scientific visualization tool developed at the University of Illinois Center for Simulation of Advanced Rockets (CSAR; [2]) by John Norris and Robert Fiedler. Research performed at CSAR involves large coupled multiphysics simulations of solid propellant rockets, as well as a wide variety of detailed subscale simulations, including burning heterogeneous propellants, pressure-driven crack propagation in various materials, turbulent fluid flows with burning aluminum droplets and smoke particles, etc.

Large simulations at CSAR involve the numerical solution of the time-dependent coupled partial differential equations (PDEs) describing a physical system, such as the Space Shuttle booster. The values of the dependent variables representing various physical quantities (e.g., the gas density, velocity, and pressure, and the material displacements and deformation velocities) are determined at sets of points comprising 3-D spatial meshes, and the time evolution of the system is computed by using the PDEs to estimate the solution at the next time level. Typically 10,000 to 100,000 time steps are required to reach a physical time of interest (for rockets, from ignition to steady burn or to propellant burn out).

During a simulation, snapshots of the solution are written to disk at a number (typically hundreds) of equally spaced physical problem time levels. Insight into the behavior of the system can be gained by visualizing the individual snapshots and by displaying a series of snapshots in a scientific animation. Rocketeer can be used to view the snapshots individually and to apply a specified set of graphics operations to an entire series of snapshots to produce a series of frames for animation. Subsequently, these frames may be converted into a GIF animation file, for example, using free software such as ImageMagick [3].

Many simulations are made parallel using domain decomposition (partitioning the 3-D meshes so that each processor works on a different portion of the entire mesh). On distributed memory systems such as linux clusters, the MPI library [4] is commonly used to pass data explicitly between processors when necessary. Parallel simulation codes typically have multiblock grids, with one or more mesh blocks per processor. The most straightforward method for such a parallel application to save a snapshot is to have each processor write its data to a separate file, although in a large simulation hundreds of thousands of files could be produced. With a bit more coding effort, the same data could be combined into a single multiblock file.

Rocketeer is designed to read field data defined on structured or unstructured single block or multiblock grids. It also can display data defined on sets of points in space that do not constitute a grid. Rocketeer automatically merges grid blocks to produce seamless images from multiblock data files. Unstructured grids handled by Rocketeer may consist of tetrahedra, prisms, pyramids, hexahedra, etc. Surface meshes composed of triangles, quadrilaterals, etc., are also supported.

Data files for Rocketeer are written using NCSA's HDF (version 4) library [5] for portability (compared to unformatted binary files) and compactness (compared to text files). An important advantage of HDF files is that their data is self describing, i.e., an application can easily read their data without knowing beforehand exactly what arrays and what data types are stored in the file. HDF allows the user to include additional information about the data in the form of attributes. Rocketeer uses the values assigned to certain attributes to identify the time levels, block names, mesh types, material types, coordinate arrays, scalars and vector fields contained in the file. This helps make Rocketeer an easy-to-use and convenient interactive tool.

Rocketeer is written in C++ and based on the Visualization Toolkit [6], [7], which uses OpenGL to exploit hardware graphics acceleration. The user interface employs wxWindows [8] for portability across platforms. Rocketeer executables for linux, Sun Solaris 2.7, and Microsoft Windows can be downloaded from the User's Guide Web page [1].

To run Rocketeer, Unix-based systems must have OpenGL and the GLX extensions to the X server installed (e.g., Mesa and Xfree86 for linux [9], [10]). The graphics card should support color depths greater than 8 bits (256 colors).

Rocketeer can display field data using a variety of techniques, including surface plots in which values of a scalar variable determine the color, colored isosurfaces, and/or slices along the x, y, and/or z axes (see Figure 1). Both 2-D surface and 3-D interior grids can be shown, with several choices available for selecting a small portion of a 3-D mesh (for speed and clarity), including specifying a bounding box, coordinates of points, or a range of indices (see Figure 2). Clipping planes can be used to cut an image along the x, y, and/or z axis, and the opacity of all objects can be varied to allow the user to see more deeply into a 3-D data set. Multiple windows can be open on the screen at the same time, and the camera position can be saved and loaded to assist the user in comparing similar data sets.

Scalar and vector quantities can be depicted using glyphs (usually spheres for scalars and oriented cones for vectors). The glyphs can represent node- or element-centered data. They can be displayed at all or a small portion of the points, with a choice of two sampling techniques: 1) random over the indices or 2) closest to a set of uniformly distributed points. The sizes of the glyphs can be uniform or they can vary with the value of some variable, such as the radius of aluminum particles or the magnitude of the velocity vectors (see Figure 3). For more details and examples, see the extensive on-line User's Guide [1].

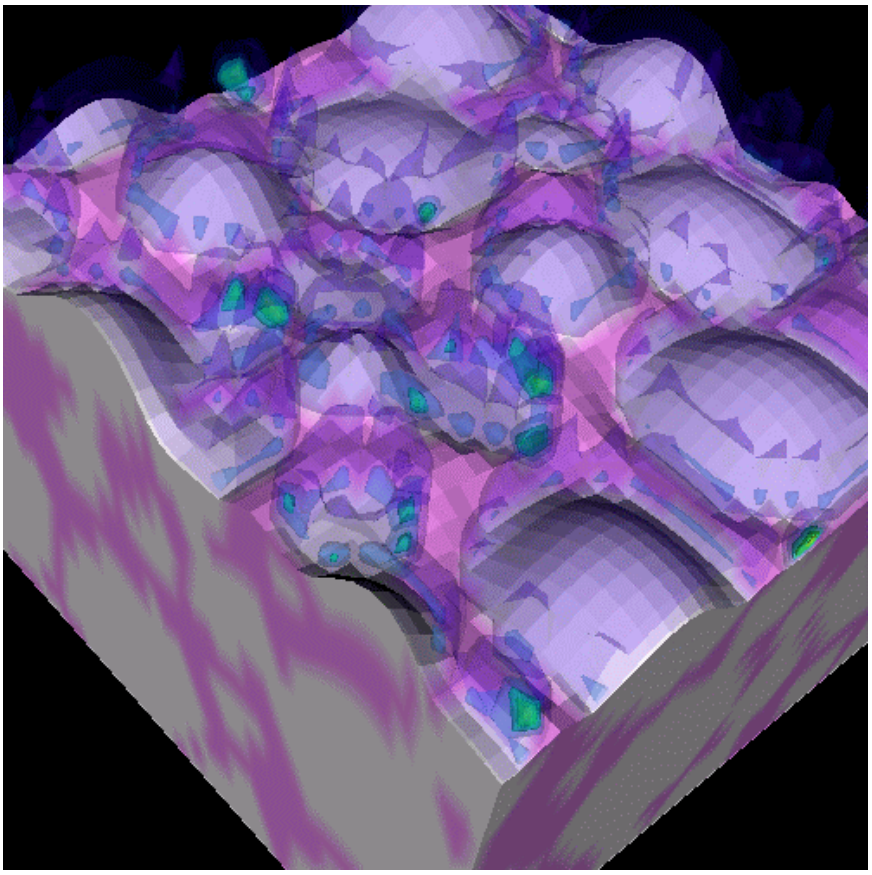

Figure 1: Rocfire [19] computation of the reaction rate for burning 3-D heterogeneous propellant. On the propellant surface, white indicates ammonium perchlorate oxidizer while pink indicates binder (fuel). The reaction rate is indicated by a series of translucent isosurafaces, ranging from low (blue) to high (green) that lie above the bumpy propellant surface. The oxidizer can burn slowly on its own, but the rate is highest near the boundaries between fuel and oxidizer.

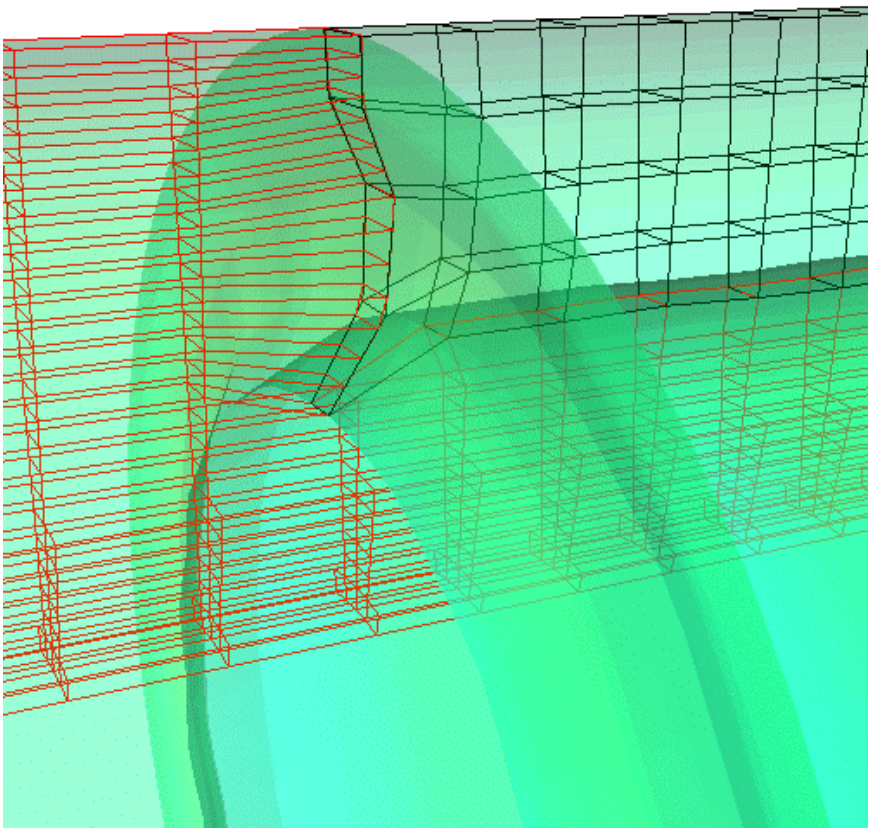

Figure 2: A portion of the interior mesh in a small cylindrical rocket [20]. The black edges show elements of the mesh in the solid propellant, while the red edges show mesh cells in the gas region inside the core of the rocket. The pressure is indicated by a translucent surface (blue is a slightly higher pressure than green). In this calculation, only the surface nodes in the gas mesh were allowed to move as the propellant deformed under pressures approaching 44 atmospheres. The calculation was stopped when the mesh became too skewed to maintain the accuracy of the solution.

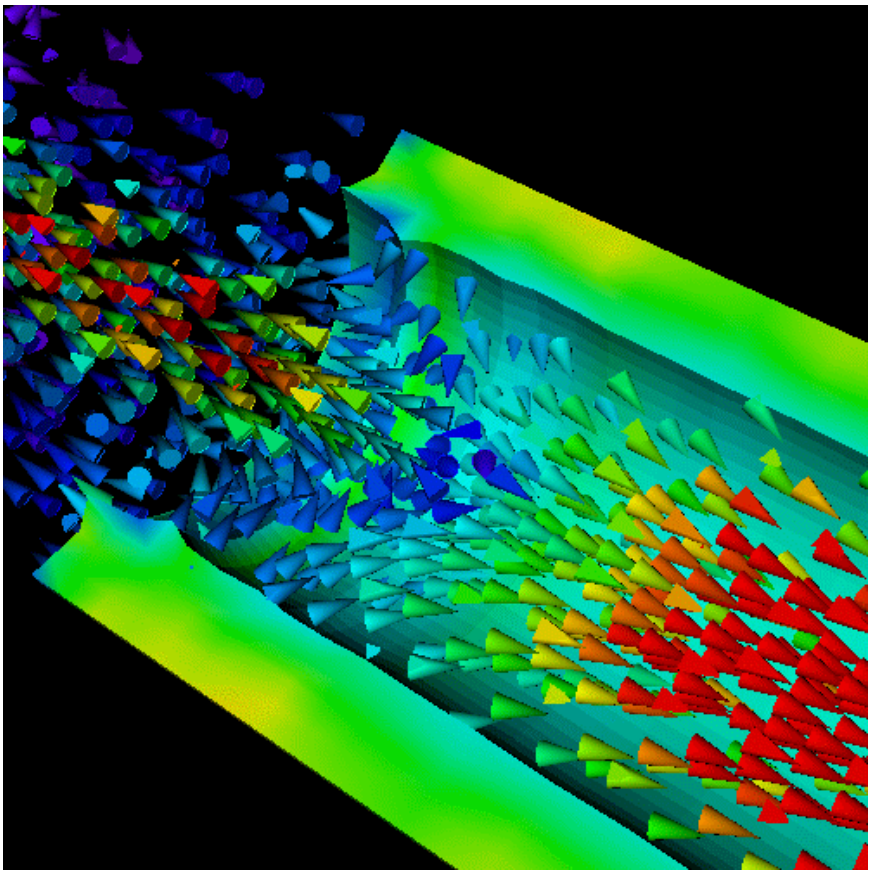

Figure 3: Cutaway view of the gas velocity (cones) and propellant average stress (colored surface) in a small cylindrical rocket [20] just after ignition. Most of the gas moves toward the nozzle (to the lower right out of the frame), but to the upper left (head end of the rocket) there is an empty chamber into which some gas is also flowing. Velocities are indicated by the color and orientation of the cones, with speeds ranging from 0.1 (blue) to 10 (red) meters per second. The gas consists of combustion products originating at the propellant surface.

2. Parallel Visualization

Rocketeer Voyager is a fully featured MPI parallel batch mode version of Rocketeer that shares its code base with the interactive version and is known to run on linux and Solaris systems. The name Voyager commemorates the space probes [11] that provided some of the first detailed images of the outer planets of our solar system. Rocketeer Voyager takes advantage of a parallel platform by processing concurrently a different snapshot on each CPU.

Like the interactive version of Rocketeer, Voyager is an OpenGL application that will attempt to exploit whatever hardware graphics acceleration is available. One could imagine a small linux cluster in which all of the nodes have identical graphics cards and operate perfectly well without a display. However, large clusters are rarely homogeneous, and the nodes may have different graphics cards or none at all. This can make installing OpenGL and an X server with GLX extensions very inconvenient. Moreover, efficient and flawless linux graphics drivers are unavailable for many graphics cards. To circumvent these problems, CSAR contracted with Xi Graphics, Inc. [12] to produce a commercial quality X server that runs in a virtual frame buffer and installs automatically across all nodes in a cluster.

Voyager takes as input a file specifying the camera position, a list of graphics operations to perform, and a list of all HDF files to process. The first two items are text files saved during an interactive session of Rocketeer. If the HDF files reside on a distributed file system (e.g., separate hard drives on each node of a cluster), each processor will make images from all HDF files on the local disk that match an entry in the file list. If the HDF files reside on a shared file system, an additional command line option is required to specify the number of HDF files needed to produce each image. This allows Voyager to determine which files each CPU should process. It is possible to use Voyager on a distributed shared file system (e.g., 2 CPUs share a disk on each node) if the HDF files are distributed among the nodes in the same manner as the MPI tasks are started by the system. The images that are created are saved to disk rather than displayed on a remote system.

Voyager first reads the list of HDF files to determine which files will be processed by each CPU, with the aim of assigning equal numbers of snapshots to each CPU. Next, the list of graphics operations is examined to determine which variables to plot and whether the range of values for each color scale is specified or is to be determined from the maximum and minimum values contained within the entire set of HDF files. If necessary, each CPU extracts the data ranges from its HDF files and the global max and min are determined. Once these steps are complete, there is no more information to be exchanged between CPUs, and the snapshots can be processed independently.

Voyager is potentially scalable to large numbers of processors, provided the I/O system is scalable. A cluster that consists of single-CPU nodes, each of which has its own local hard disk, should be capable of delivering scalable I/O. On such a system, if the HDF files (and the images) are stored on the local disks, Voyager should scale linearly, i.e., it should take N CPUs virtually the same wall clock time to process N snapshots as it would take one CPU to process a single snapshot. If the files are stored on a shared file system, contention for I/O bandwidth could severely degrade scalability.

In practice, it often takes several attempts to produce a set of images suitable for animation. One may have to experiment with the camera position, isosurface values, data ranges, clipping planes, etc., in order to clearly show the physically interesting features of a numerical solution. A parallel tool such as Voyager makes this type of experimentation more practical. Consequently,

Voyager may be run several times in succession using the same HDF data files, but slightly different camera position and graphics operations files.

3. Benchmark Results

Through a generous grant from Intel, the University of Illinois Computational Science and Engineering program [13] operates a linux cluster called turing [14] that consists of 208 two-processor Pentium (II or III) Xeon nodes (400 MHz to 1 GHz) connected with both 100 Mb Ethernet and Myrinet [15]. It also includes a 4-CPU (400 MHz Pentium II) front-end system, which doubles as a file server that all nodes can access via 100 Mb Ethernet. Each node also has an 18 GB internal hard disk, which jointly comprise a distributed shared file system.

The benchmark data set consists of 96 snapshots written by the gas dynamics module of a Space Shuttle booster simulation. The images resemble Figure 4, but do not show the stress in the propellant. Each HDF data file is about 275 MB in size and consists of 287 structured mesh blocks with a total of over 5 million grid points. The of list graphics operations to be applied to each data file includes drawing several temperature isosurfaces and approximately 15,000 glyphs (cones) that represent the velocity field.

To benchmark Voyager, we placed the data files on the shared file system as well as distributed a different snapshot to the local file system on each of 96 nodes (all 1 GHz). We determined the wall clock time for the generation of the images from start to finish using the shell's built-in "time" command. These benchmarks are quite unlike most graphics benchmarks, which deliberately minimize the time spent on tasks other than rendering images from data already stored in memory, in order to focus on the graphics system. Here we are measuring the wall clock time for Voyager to make a specific set of image files from a set of large HDF files.

On a single processor of our linux cluster, the images are generated nearly as quickly as they are on a Sun Ultra 60 with the latest Expert3D graphics card, despite the fact that the virtual frame buffer on turing cannot utilize hardware graphics acceleration. The reason for this is that most of the run time is spent reading the data in the HDF files.

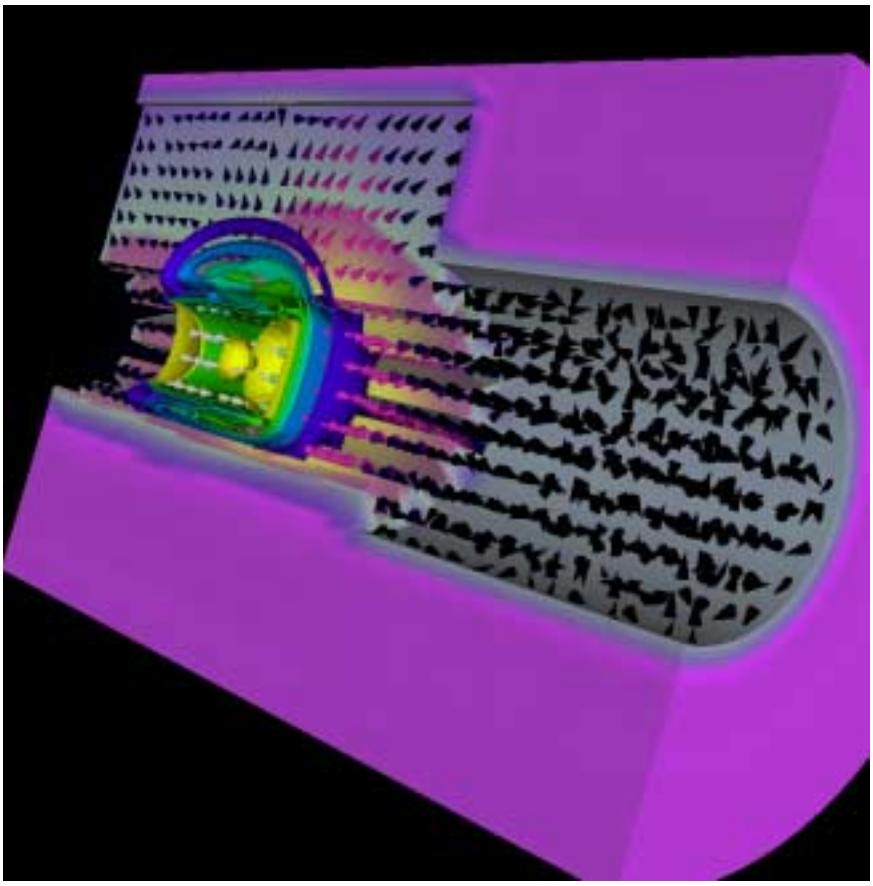

Figure 4: Gas velocity (cones), gas temperature (colored isosurfaces), and propellant average stress (gray/purple colored surface) in the head end of a Space Shuttle booster just after ignition. The igniter fits into the left end of the cylindrical hole down the middle of a region in which slots are cut along the axis of the rocket into the propellant (11 point star). Hot gas, indicated by blue (cool) to green (warm) to yellow (hot) has just begun to flow into the rocket chamber and spread down into the slots. Gas is moving radially away from the igniter inlet, but the flow has not yet reached the right side of the frame, where the propellant has a simple cylindrical geometry and the velocity cones are still pointing in random directions (very small speeds).

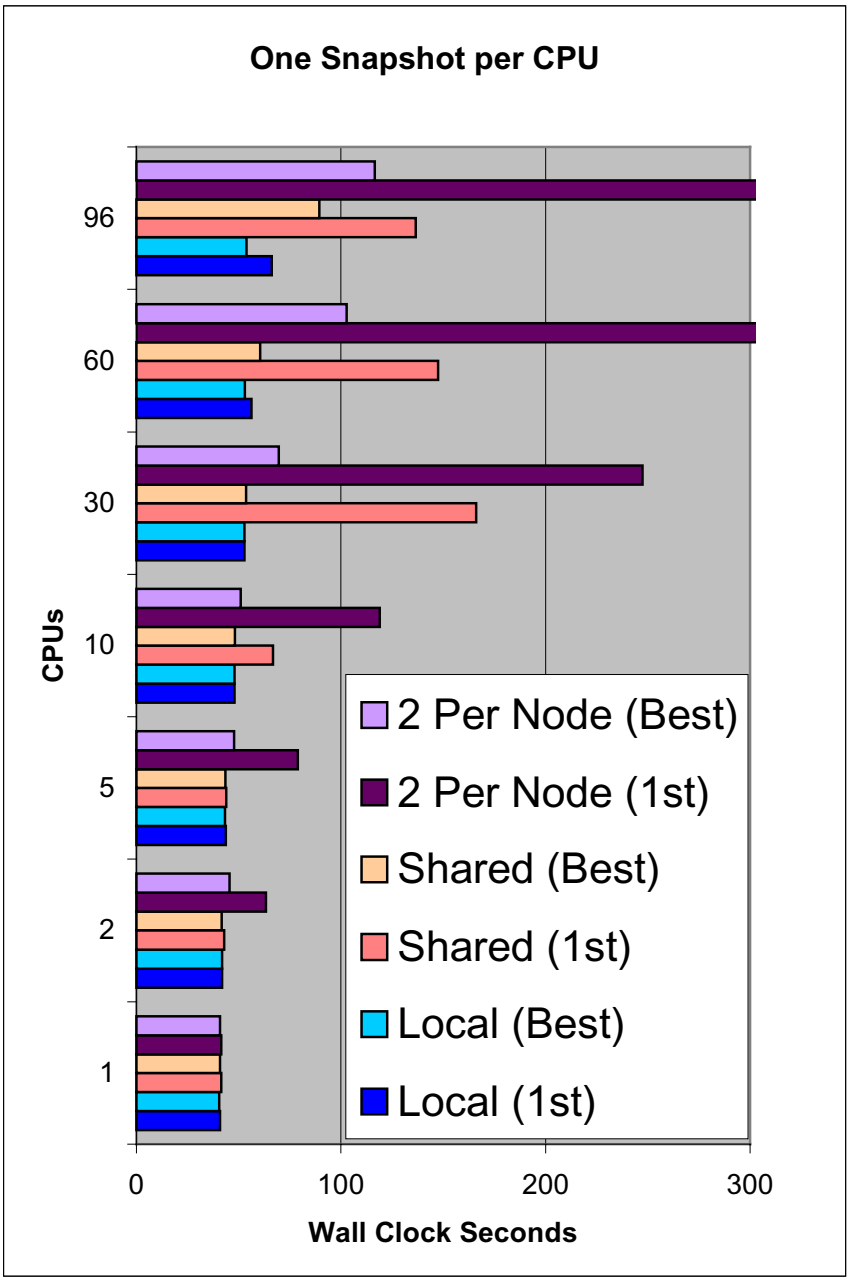

Figure 5: Voyager run times for processing a number of snapshots equal to the number of CPUs.

Figure 5 gives the wall clock time in seconds for runs in which each CPU processes 1 snapshot. The bars labeled "1st" for a given number of CPUs give the time for the first trial, while the bars labeled "Best" give the best time for several trials. The series labeled "Local" corresponds to the data file being read from the hard disk on each node, the "Shared" series is for data files on the shared file system, and the "2 Per Node" series is for data on the shared file system and 2 processors active per node.

If the system were idle and the I/O system were perfectly scalable, all of these runs would complete in less than 41 seconds, the time to process a single snapshot on one CPU. When the data files are on the local disks and only 1 CPU per node is used, I/O contention should be minimal and scalability should be excellent. Our timings show respectable scalability even for large numbers of processors, especially when the run is repeated. The largest speedup we obtained was 72.4 for 96 nodes, which is quite good in light of the facts that the system was not in dedicated mode and all overhead is included in these run times.

The timings in the "Shared" series are surprisingly competitive with those in the "Local" series, which reflects the high bandwidth between the shared file system and the nodes (100 Mb Ethernet plus a generous number of switches). As one might expect, scalability suffers for larger numbers of nodes, particularly on the first trial. However, in subsequent runs in which the same nodes process the same data files again, the run times are much closer to those for runs in which the data as on local disk. This is understandable if each node retains some of its data in a local disk cache.

Since these trials were conducted in order of increasing processor count, the run times for the first test on a new, larger number of processors benefited from the fact that a series of runs on a subset of the processors was just completed and therefore some of the data was already in cache. This situation may occur in practice if one is making animation frames from an incomplete set of snapshots produced by a run in progress. The user might generate an animation with the existing snapshots at one point in time, and then return hours later to make longer animations that include newer snapshots. In this case all of the frames should be redone, because the data ranges could change when the newer snapshots are added to the series.

The run times obtained when 2 CPUs per node are used to make images are significantly longer than run times for 1 CPU per node, especially for large numbers of processors. This is to be expected, since the two processors in a node share I/O bandwidth when accessing files on the shared file system. The two processors in a node also share I/O bandwidth to the local hard disk, but one may expect reasonably good scalability for this situation, since the local disk I/O bandwidth increases linearly with the number of processors. Note that for a fixed number of nodes, allowing both CPUs in each node to process snapshots reduces the run time compared to leaving one processor idle, at least after the first trial. For example, processing 30 snapshots on 30 nodes using 1 processor per node takes about 54 seconds, and therefore it should take about 108 seconds

for 60 snapshots. If we instead use 60 processors in 30 nodes to process 60 snapshots repeatedly, the run time is only 103 seconds.

It took about 3 hours to distribute the data files from the shared file system to the local disks using a serial script running the scp command. If we ran a parallel version of the script to distribute the data files, it would take at least 100 seconds to complete, which could be more time than is saved by running Voyager one time with the data files on local disks rather than on the shared file system. It certainly would not pay to distribute the files if the user is likely to perform multiple runs using the same data, since the data is automatically cached to local disk by the system during the first run.

4. Discussion and Conclusions

Rocketeer Voyager is a powerful tool for producing images in parallel from a series of snapshots output by a wide variety of time-dependent numerical simulations. Voyager has nearly all the features of the interactive version of Rocketeer, although it saves images to files rather than displaying them on a monitor. A commercial quality X server that runs in a virtual frame buffer and installs automatically on all nodes of a heterogeneous linux cluster was developed by Xi Graphics, Inc. to allow such OpenGL-based applications to be used in batch mode on systems without any graphics hardware.

For the set of large HDF data files we used to benchmark Voyager, the run time for processing a set of snapshots is dominated by the time spent reading the HDF data files. Even though all parallel overhead is included in the run time, scalability is respectable for large numbers of processors, especially if the data is on local disk, or if the data is on a shared file system and a run has already been performed using the same data files (but perhaps a different camera position or set of graphics operations). After the files have been read once, some data remains in local disk caches, and therefore during subsequent runs contention for I/O bandwidth to the shared file system is greatly reduced.

If the simulation is run on the same cluster to be used for visualization, the data could be saved to local disk to make visualization more efficient. Note that Voyager requires all of the blocks in a multiblock data set for a given snapshot to reside on a file system accessible to the CPU responsible for processing that snapshot. However, throughout the simulation, each processor works on a portion of the mesh, and therefore if each processor writes its data to local disk, the blocks for each snapshot will be distributed over separate local file systems. Solutions to this problem include modifying the simulation to: 1) write to a shared file system (or a parallel file system) at a cost of some efficiency during the first run of Voyager, 2) pass as messages the data for each snapshot to a single node at a cost of extra communication, or 3) use the PANDA parallel I/O library [16], [17] to collect and combine on extra nodes the data from the compute nodes and migrate it to other nodes that will do the visualization. With this last option, it would be possible for the extra nodes to migrate data from a simulation running on a remote supercomputer to a local cluster for visualization, provided remote logins to individual nodes are allowed.

Writing and reading HDF version 4 files unfortunately takes many times longer than the equivalent operations on binary files. We are considering adding support in Rocketeer for HDF version 5 [18] files, as well as other data formats.

Our future plans include the development of a client/server version of Rocketeer. The server will run on parallel platforms including linux clusters, while the client will run on desktop workstations and laptops. Our focus for the client/server version is on utilizing a parallel server to speed up the generation of individual images, rather than on processing multiple snapshots in parallel. CPUs running the server will adopt the domain decomposition of the simulation and process the mesh blocks in parallel. A subsequent serial step in the visualization pipeline will merge the images and remove artifacts due to the duplication of block boundaries. Final rendering will take place on the client, where hardware graphics acceleration will further enhance performance.

References

[1] Rocketeer: http://www.csar.uiuc.edu/F_software/rocketeer
[2] CSAR: http://www.csar.uiuc.edu
[3] ImageMagick: http://www.imagemagick.org
[4] MPI: http://www-unix.mcs.anl.gov/mpi/index.html
[5] NCSA HDF version 4: http://hdf.ncsa.uiuc.edu/hdf4.html
[6] Kitware: http://www.kitware.com
[7] Schroeder, W. J, Martin, K. M., Avila, L. S. & Law, C. C. *The VTK User's Guide*, Kitware, Inc., 2000.
[8] wxWindows: http://www.wxwindows.org/
[9] Mesa: http://www.mesa3d.org
[10] Xfree86: http://www.xfree86.org
[11] NASA Voyager 2: http://www.jpl.nasa.gov/calendar/voyager2.html
[12] Xi Graphics, Inc.: http://www.xig.com
[13] CSE: http://www.cse.uiuc.edu
[14] CSE's turing cluster: http://turing.cse.uiuc.edu
[15] Myricom, Inc.: http://www.myrinet.com
[16] Rocpanda: http://cdr.cs.uiuc.edu/panda/rocpanda
[17] Lee, J., Winslett, M., Ma, X., &. Yu, S. Tuning High-Performance Scientific Codes: The Use of Performance Models to Control Resource Usage During Data Migration and I/O. To appear in *Proceedings of the 15th ACM International Conference on Supercomputing*, June 2001.
[18] NCSA HDF version 5: http://hdf.ncsa.uiuc.edu/HDF5
[19] Rocfire: http://www.csar.uiuc.edu/~tlj/rocfire.html
[20] Fiedler, R. A., Namazifard, A., Alavilli, P., Tang, K., Parsons, I. D., & Hales, J. Coupled Fluid-Structure 3-D Solid Rocket Motor Simulations. AIAA Paper 2001-3954, *37th AIAA Joint Propulsion Conference*, Salt Lake City, UT, July 2001.