Rendering of Numerical Flow Simulations Using MPI

John Stone
Department of Computer Science
University of Missouri-Rolla
E-Mail: johns@cs.umr.edu

Mark Underwood
Department of Mechanical and Aerospace Engineering
University of Missouri-Rolla
E-Mail: munder@umr.edu

Abstract

Results from a parallel computational fluid dynamics (CFD) code combined with a ray tracing library for run-time visualization will be presented. Several factors make in-place rendering of CFD data preferable to the use of external rendering packages or dedicated graphics workstations. In-place rendering avoids significant I/O to disks or to networked graphics workstations, and provides the ability to monitor simulations as they progress. The use of MPI (Message Passing Interface) in both codes helped facilitate their combination into a single application. Also due to the use of MPI, the two separate applications have been run on several different architectures. The parallel architectures include networks of workstations, the Intel iPSC/860, the Intel Paragon, and the IBM SP2.

1. Introduction

In this paper, a parallel application for the simulation and visualization of high-speed fluid flow is presented. One of the most common problems in visualizing large CFD (Computational Fluid Dynamics) simulations is transferring large amounts of data between the simulation process and the rendering process. In many cases, CFD simulation data is stored in temporary disk files, which are then read into a rendering program or transmitted over a network to another machine for rendering. To improve the overall rendering performance, the simulation and rendering programs can be combined into a single unit thereby avoiding unnecessary I/O. Rendering then can be done in-place on the CFD data while stored in main memory. Along the same lines, an MPEG or JPEG compression engine may be tied to the rendering process to further streamline the compression of animated video. By merging the simulation, rendering, and compression processes, overall runtime is greatly reduced.

2. Parallel Computational Fluid Dynamics Simulation

CFD has gained a very important place in the analysis of complex fluid flows. The CFD code described in this paper numerically simulates compressible fluid flow by solving the full, three-dimensional Navier-Stokes equations. The code has the capability to handle both multiple species and finite-rate kinetics for hydrogen-air combustion. The Navier-Stokes equations are coupled, non-linear partial differential equations which are based on the conservation of mass, momentum, and energy. Due to the very strongly coupled and non-linear nature of these equations, typical CFD solutions require significant computer time and memory. However, recent research has demonstrated the feasibility of using parallel computing to help reduce the computational cost of these simulations [5],[4].

The parallel CFD code PARAFLOW uses a finite difference formulation which relies on the explicit MacCormack method to time-march the solution. The versions of the MacCormack method available in the code are both second-order accurate in time and either second-order or fourth-order (nominally) accurate in space.

For portability, PARAFLOW can be used in either sequential or parallel mode. The code has been validated for both modes and achieves near-linear speedup in the parallel mode for several test cases. The parallel implementation is directed toward a homogeneous domain decomposition on a distributed-memory architecture with point-to-point communication between neighboring processors and a limited amount of collective operations. All message-passing is included explicitly in the code; there is no reliance on parallelizing compilers. Additionally, processor boundaries are overlapped with neighboring processors in order to overlap communication with computation.

Furthermore, the CFD code has the option of either using point-to-point communications to perform the collective operations (with either NX or MPI) or using the MPI collective operations. Currently, the point-to-point collective operations for both MPI and NX require the number of processors to be an integer power of 2 because the code uses multistage pairwise exchanges (butterfly) to perform the collective operations. When using the MPI collective operations, there is no such restriction on the number of processors. As a general rule, only minor performance differences are observed between the different collective operation implementations for typical applications.

Also, MPI allows for portability across different platforms. PARAFLOW has been tested on several sequential and parallel architectures. The parallel architectures include the Intel iPSC/860, the Intel Paragon, the IBM SP2, and various networks of workstations.

3. Parallel Volume Rendering using Ray Casting

Volume rendering is a technique for rendering multidimensional grids of data using the data values to determine visual characteristics such as opacity and color. Volumetric information can be difficult to interpret using standard two dimensional plotting packages. By visualizing the data in three dimensions, and especially in conjunction with animation, it is much easier to grasp important features or trends in the data. In the case of CFD simulations, it is very useful to watch the simulation as it progresses. By visualizing data as it is produced, problems in the setup of a numerical experiment can be detected visually and usually sooner than through inspection of numerical results. The human ability to rapidly interpret graphical information offers a motivation to have run-time visualization capability built into simulation engines.

Volume rendering can be achieved using several different approaches, some based on polygonal surfaces, and others based on ray casting. The volume renderer used in this research is based on ray casting and is part of a parallel ray tracing library currently under development [3]. Volume rendering has many applications, such as visualization of volumetric simulation data and as an enhancement to ray tracing [2] software to provide features such as *Hypertextures* [1]. An advantage in adding volume rendering capability to a general purpose ray tracing library is the ability to render volumetric data along with objects such as spheres, polygons or other surfaces. Through the combination of volumetric rendering and surface rendering it is possible to render scenes containing clouds, fire, fluid flow, MRI scans, and hypertexures alongside more commonplace objects made from polygons and the like.

In order to render the CFD data in-place, the ray tracing library was extended to handle distributed data, and volumes based on irregular grids. These extensions are based on user supplied mapping functions which deal with irregular geometry, value interpolation and any other required interpretation of volume data. The key to making a volume renderer extensible is allowing the user to supply custom mapping and interpretation functions which are optimized for his or her particular data. One of the original goals in the writing of the ray tracing library was to avoid having to recompile the library for custom user programs. This is especially important in the case where a user will be supplying mapping and interpretation functions for their data. The solution was to use C function pointers. Function pointers can be updated at runtime, and allow the user great freedom without the need to recompile the ray tracing library itself. Future work on the ray tracing library will focus on user extensibility. MPI has played a large part in the development of the ray tracing library, allowing it to run on every kind of parallel environment that the authors have had access to. Future enhancements include expanded use of MPI features such as communicators and context to allow greater flexibility when linking with complex simulations consisting of many tasks or processes.

4. Implementation Notes

The ability to supply CFD data in a form usable by the rendering library is the key issue in performing the runtime visualization. By using the MPI message passing capability of both codes, the joining of the two codes was relatively easy. MPI provides features which aid in writing parallel libraries such as the rendering library. The initial implementation uses a single MPI communicator, and maps both the CFD process and the rendering process to the entire set of processors. Although this approach has been adequate for the work to date, future work will require the use of multiple communicators, and will involve several additional computational processes. The current code is able to function with a single communicator because the messages of each sub-process are guaranteed not to overlap. The next phase of the research will introduce a run-time simulation control mechanism, additional I/O options including an MPEG compression process, and a remote graphical user interface. MPI provides key features which will enable these components to be integrated into a single parallel application.

The interface between the CFD code and the rendering library is constructed through the use of mapping functions. These mapping functions are responsible for all aspects of transforming the CFD data into the form required by the rendering library. In the current implementation, the mapping functions perform coordinate system transformations, data interpolation, and retrieval of remote data. Mapping functions are registered with the rendering library at runtime which allows flexibility in both implementation and use.

5. Results

The initial implementation in this investigation has been very encouraging. The rendering and simulation codes have been successfully merged. Several test simulations have been performed with various rendering characteristics. Since both the CFD and rendering codes achieve linear speedup individually, similar speedup was expected and observed in the runtime of the merged code. Since the rendering code depends on user supplied mapping functions for geometric transforms and value interpolation, a high percentage of the rendering runtime involves calling the mapping functions. Overall rendering time is reduced significantly by making mapping functions efficient. For a single rendered image, it is typical to call the user supplied functions one hundred million times. Although the functions are typically very simple, every bit of performance helps.

Overall performance of the merged application is substantially better than could be achieved using more traditional methods. As a quick example, a single time step of the CFD simulation can take more than 30 minutes to write to disk files on a parallel file system. This is just the I/O time involved in storing the data from one particular time step onto a disk. In order to render the data, it must also be read into or transmitted to a typical rendering program which takes additional time. Rendering the same time step in-place, instead of in a separate step, takes about 30 seconds. This is an overall reduction in time of more than 6000 percent! Not only does the overall execution speed greatly out pace the alternative, but the rendering process itself constitutes only a small portion of the entire runtime. For the injector simulation described in this paper, rendering accounted for approximately 10 percent of the overall runtime when rendering one of every 8 time steps. Even though I/O is drastically reduced through the use of in-place rendering, I/O is *still* the main bottleneck in the combined code. Future work will address this through the use of MPEG compressed video output.

The physical layout of the test case which was modeled is a constant cross-sectional area duct with viscous (no-slip) top and bottom walls. The front and back walls are treated as inviscid (no velocity normal to wall). The main flow is Mach 3 air moving from left to right with a flush wall injector on the bottom wall. The injection is Mach 1 air directed 20 degrees up from the horizontal. Figure 1 shows a conceptual schematic of the flow field described. This schematic is not to scale but shows the essential features of the flow field. Also note that the different shocks vary considerably in strength. For the test case, the injector bow shock should be the strongest followed by the recirculation shock. The bottom boundary layer shock and the top boundary layer shock should be of different strengths, however. The reason for the difference in the top and bottom boundary layer shocks is due to the recirculation region which causes the boundary layer to thicken ahead of the recirculation region on the bottom wall. This thickening results in the bottom boundary layer generating a stronger shock than the top boundary layer.

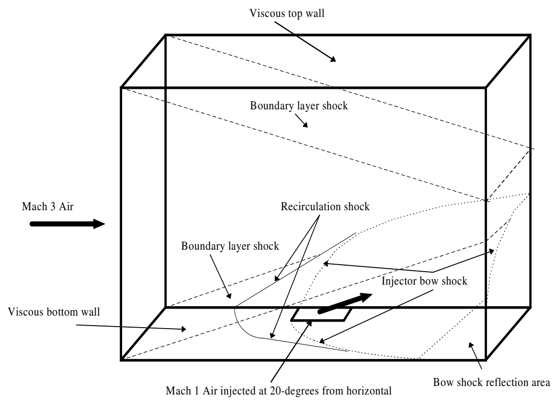

Figure 1. Conceptual Schematic of the Injector Flow Field

Figure 2 shows the pressure contours as predicted by the parallel CFD code and as rendered using the parallel ray tracing library at one particular instant of time.

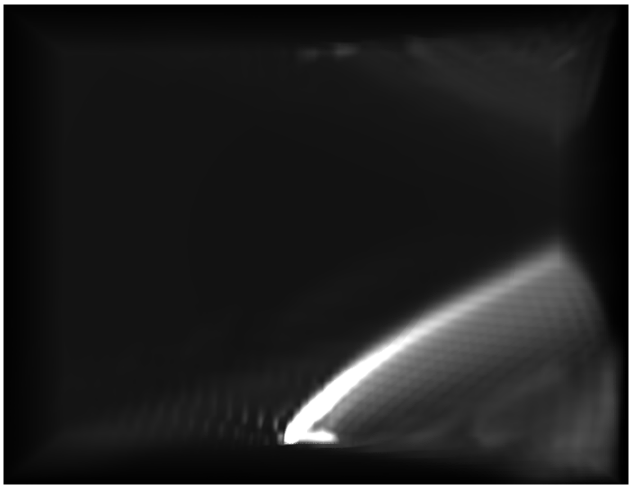

Figure 2. Example Rendering of CFD Data

The injector is the white rectangular region at the middle of the bottom wall. The bow shock due to the injection is the most prominent feature (white and gray region along bottom wall). Note that the conceptual schematic is in an isometric perspective view while the rendering is in one-point perspective view. This difference accounts for the different direction for the curvature of the bow shock on the right hand side. Slightly ahead of the bow shock and near the bottom wall, the triangular recirculation shock can be seen. Due to the normalization of the pressure contours, the weak top boundary layer shock is very faint, while the bottom boundary layer shock is slightly more visible and is seen upstream of the recirculation shock. The important point is that the rendering displays the same major flow features as shown in the conceptual schematic.

The full effect of the volume rendering is best achieved by rendering a frame for every 8 or more iterations of the CFD code. These frames can then be used to create an animation. Starting from the first iteration, one can see the pressure shocks develop and watch them wash toward the right side of the volume. To see an example of this kind of animation, please contact the authors for MPEG compressed video footage.

6. Conclusions and Future Work

MPI is an essential part of this investigation due to its ability to easily merge separate parallel applications. It is clear from this and previous work that writing portable parallel code, and utilizing MPI or its equivalent, will become increasingly important in order to reduce both code development and simulation times. By combining the previously separate tasks of simulation and visualization, overall runtime is greatly reduced.

The results from this work are very promising. In the near future, the authors will explore several different approaches in mapping the volume data, hence potentially improving image quality and rendering performance. This next stage of work will also involve adding MPI safety features and additional MPI communicators in order to further insure reliable message passing, extensibility, and portability. Once the current implementation has been refined and safety features have been incorporated, a graphically interactive simulation application will be built and tested. Other areas of exploration will include the use of threads and shared memory in both codes (currently only implemented in the rendering library). Since clusters of shared memory machines are becoming common, it seems logical to take advantage of their strengths when possible. Initial results from the rendering library's thread implementation have been very promising, and have demonstrated the simultaneous use of threads and message passing for increased performance.

7. Acknowledgments

The authors would like to acknowledge support and computational resources provided by Oak Ridge National Laboratory Center for Computational Sciences, the Hypersonic Vehicles Office at NASA Langley Research Center, NASA NAS, Washington University Computer and Communications Research Center, and the University of Missouri-Rolla. One of the authors (MU) would also like to acknowledge support provided through the NASA Graduate Student Researchers Program (GSRP) by the Hypersonic Vehicles Office at NASA Langley Research Center.

References

[1] Ebert, Musgrave, Musgrave, Peachey, Perlin, and Worley. *Texturing and Modeling, A Procedural Approach*. Academic Press, 1994.

[2] A. Glassner. *An Introduction to Ray Tracing*. Academic Press, 1989.

[3] Intel Supercomputer Users Group. *An Efficient Library for Parallel Ray Tracing and Animation*, 1995.

[4] D. Riggins, M. Underwood, B. McMillin, L. Reeves, and E. J.-L. Lu. Modeling of supersonic combustor flows using parallel computing. *Computing Systems in Engineering*, 3(1-4):217–219, 1992.

[5] M. L. Underwood. The simulation of supersonic flows using parallel computing. Master's thesis, University of Missouri-Rolla, 1993.