The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

REAL-TIME VISUALIZATION OF AEROSPACE SIMULATIONS USING COMPUTATIONAL STEERING AND BEOWULF CLUSTERS

A Thesis in

Computer Science and Engineering

by

Anirudh Modi

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2002

We approve the thesis of Anirudh Modi.

Date of Signature

Paul E. Plassmann
Associate Professor of Computer Science and Engineering
Thesis Co-Advisor
Co-Chair of Committee

Lyle N. Long
Professor of Aerospace Engineering
Professor of Computer Science and Engineering
Thesis Co-Advisor
Co-Chair of Committee

Rajeev Sharma
Associate Professor of Computer Science and Engineering

Padma Raghavan
Associate Professor of Computer Science and Engineering

Mark D. Maughmer
Professor of Aerospace Engineering

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

ABSTRACT

In this thesis, a new, general-purpose software system for computational steering has been developed to carry out simulations on parallel computers and visualize them remotely in real-time. The steering system is extremely lightweight, portable, robust and easy to use. As a demonstration of the capabilities of this system, two applications have been developed. A parallel wake-vortex simulation code has been written and integrated with a Virtual Reality (VR) system via a separate graphics client. The coupling of computational steering of parallel wake-vortex simulation with VR setup provides us with near real-time visualization of the wake-vortex data in stereoscopic mode. It opens a new way for the future Air-Traffic Control systems to help reduce the capacity constraint and safety problems resulting from the wake-vortex hazard that are plaguing the airports today. In another application, an existing computational fluid dynamics code has been integrated with the steering system to enable interactive visualization of complex flow simulations from any remote workstation. Benefits of scalability and dimensional reduction arising from this approach have been imperative in the development of this system. This system makes the visualization of flow simulations easier, efficient, and most-importantly without any delay as and when the results are generated, thus giving the user much greater flexibility in interacting with the huge amounts of data. At a more basic level, the ability to get “immersed” in the complex solution as it unfolds using the depth cue of the stereoscopic display and the near real-time nature of the computational steering system opens a whole new dimension to the scientists for interacting with their simulations.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I am extremely grateful to my advisors, Dr. Lyle N. Long and Dr. Paul E. Plassmann for all their help and guidance during the course of my graduate studies at Penn State. I am especially thankful to Dr. Long for providing a wonderful work environment here at the IHPCA. The autonomy and the resources provided to me have always been unmatched. I have the highest appreciation for the manner in which he has encouraged and supported me at all times. I am equally indebted to Dr. Plassmann for his unflinching confidence in me and for all the valuable time he spared from his busy schedule for the innumerable discussions we had. His wisdom and humour have always been a source of motivation and amazement for me. Together, they have been the most wonderful advising team a graduate student could hope for. Without their constant support, motivation and useful suggestions, this project would never have taken its present shape.

I would like to thank my committee members, Dr. Rajeev Sharma, Dr. Padma Raghavan, and Dr. Mark D. Maughmer for their time to review my thesis. I would like to thank the NSF for the equipment grant EIA–9977526, and the Rotorcraft Center of Excellence (RCOE) at Penn State, for providing the financial support for this work. I would like to thank Zach Vanderveen for his help in the development of the graphical interface used in this work. I would also like to thank my colleagues, Abraham Mathew, Frederic J. Souliez, LTC Robert Hansen, Nilay Sezer-Uzol, and Kelly Corfeld for creating a wonderful work atmosphere. I also thank my friends (and long-time roommates), Anurag Agarwal and Anupam Sharma, for making my stay at Penn State one of the most enjoyable and memorable experiences of my life. I would like to make a special mention of my friend, the late Vernon Mendanha, whose

immortal words always have and always will be the guiding force in my life.

Finally, words cannot describe the love, support, advice and encouragement that I have received from my family. My wife, Sushmita, has been always been a constant source of energy in my life. I thank her for her unfailing and loving support and for her boundless patience during the course of this work. As for my parents, their love, support and motivation have been the very backbone of my existence. All that I have achieved today is a result of the innumerable silent sacrifices they have made over the past decades.

Dedicated to my parents.

Chapter 1

Introduction

1.1 Real-Time Simulations and Virtual Environments

Parallel simulations are playing an increasingly important role in all areas of science and engineering. As the areas of applications for these simulations expand and their complexity increases, the demand for their flexibility and utility grows. Interactive computational steering is one way to increase the utility of these high-performance simulations, as they facilitate the process of scientific discovery by allowing the scientists to interact with their data. On yet another front, the rapidly increasing power of computers and hardware rendering systems has motivated the creation of visually rich and perceptually realistic virtual environment (VE) applications. The stereoscopic display and the enhanced user-interaction provided by these VE systems make them an ideal platform for visualization of complex simulations. The combination of computational steering and VE applications provides one of the most realistic and powerful simulation tools available to the scientific community.

As an example of such an important application here is the problem of maximizing airport efficiency. National Aeronautics and Space Administration (NASA) scientists predict that by the year 2022, three times as many people will travel by air as they do today [1]. To keep the number of new airports and runways to a minimum, there is an urgent need to increase their efficiency while reducing the aircraft accident rate. Today, the biggest limiting factor for airport efficiency is the time delay between aircraft take-offs and landings which are necessary because of the wake-vortices generated by the moving aircraft. Moreover, according to the predictions

by the United States Federal Aviation Administration (FAA), if by the year 2015 the wake-vortex hazard avoidance systems do not improve in any significant way, there is the potential for a significant increase in the number of aviation accidents [2]. One of the goals of this work is to create a wake-vortex hazard avoidance system by realistically simulating an airport with real-time visualization of the predicted wake-vortices. This system can be used to dynamically alter the aircraft take-off and landing trajectories to avoid the collision with wake-vortices. If implemented, such a system has the potential to greatly increase the utilization of airports while reducing the risks of possible accidents. In this work, an easy to use, yet powerful computational steering library is developed to deal with the complexities of real-time wake-vortex visualization. Since the wake-vortex prediction for an entire fleet of aircraft taking-off and landing at a busy airport is a computationally intensive problem, a parallel solution for the same is proposed.

A computational steering system is required to enable such a complex simulation. The output from the parallel code is sent to the VE system via a computational steering framework. A significant amount of work has been done on computational steering over the past few years. Reitinger [3] provides a brief review of this work in his thesis. Some of the well known steering systems are *Falcon* [4] from Georgia Tech, *SCIRun* [5] from Scientific Computing and Imaging research group at University of Utah, ALICE Memory Snooper [6] from Argonne National Laboratory, *VASE* [7] (Visualization and Application Steering) from University of Illinois, *CUMULVS* [8] from Oak Ridge National Laboratory, *CSE* [9] (Computational Steering Environment) from the Center for Mathematics and Computer Science in Amsterdam, and *Virtue* [10] from University of Illinois at Urbana-Champaign. A summary of the environment characteristics of these systems based on their scope and their user interface is shown in Table 1.1. Three different types of scope can be distinguished based on the information extracted

from the target application [11]:

1. *Model exploration*: In this, the user is interested primarily in the application's input and output data.

2. *Algorithm experimentation*: In *algorithm experimentation*, the user is informed about the application's program structure and the program algorithms can be adapted to suite the user's needs during the simulation.

3. *Performance optimization*: In this, the user is provided information about the application's configuration and progress so that the user may improve its performance by manually adjusting some parameters.

	Model exploration	Algorithm experimentation	Performance optimization	User Interface
Falcon			x	2-D visualization; steering user-interface
ALICE	x			steering GUI; MATLAB [12] interface
VASE	x	x		visualization through existing packages; steering through textual input
SCIRun	x	x		visualization modules; steering user interface
CUMULVS	x		x	visualization external through AVS [13]; steering through textual input
CSE	x			graphical editor for multi-dimensional datasets; steering GUI
Virtue			x	Interactive VR visualization and steering

Table 1.1. Summary of environment characteristics of existing steering systems [3]

Computational steering has also been attempted by coupling of a 2-D solver with the

Virtual Reality (VR) environment by Carolina Cruz-Neira et al [14, 15] at the Electronic Visualization Lab (EVL), University of Illinois, in as early as 1993. They worked on very simple simulations of Rayleigh-Taylor instability and gravitational wave components predicted by Einstein's theory of general relativity. The datasets were relatively small and calculations were simplified by approximating theoretical algorithms and decreasing rendering quality. The steering was implemented by providing a menu to change input parameters of the simulation and then restarting the simulation when a change was detected. The computation was simultaneously performed on a separate machine which was networked with a CAVE. Parker et al. have several publications [16, 17, 18] which discuss computational steering software systems and strategies, and applications of their SCIRun system. Juler et al. [19] have implemented a software architecture called Dragon for real-time battlefield visualization. Taylor et al. [20] have done some interesting work which explores the lag time involved in interactive virtual reality applications.

While they are all powerful, the major drawback of these systems is that they are often too complex, are not object-oriented and have a steep learning curve. It is not easy to couple these systems with existing scientific codes. It may take a significant amount of time to be productive with these systems, especially for the large number of computational scientists with little formal education in computer science or software systems. To address these problems, a new lightweight and portable computational steering system based on a client/server programming model has been developed.

1.2 Representative Applications

The ease-of-use and the power of the new computational steering system, POSSE, is illustrated with the help of two major applications. The first application deals with the real-time

visualization of wake-vortex hazards, whereas the second application deals with the interactive visualization of complex computational fluid dynamics simulations. Here, "real-time" means that the physical time simulated by the wake-vortex prediction application is not less than the actual running time of the application by more than a small upper limit ε–that is, t seconds of wake-vortex physics does not take any longer than $t + \varepsilon$ seconds to compute. Both these applications run on parallel machines and use a remote graphics workstation for visualization. Theoretically, however, the steering software described here could be used in traditional real-time systems such as flight simulators and aircraft cockpits. This would require more work to make it fault-tolerant.

1.2.1 Real-Time Simulation of Wake-Vortex Hazards

Just as a moving boat or a ship leaves behind a wake in the water, an aircraft leaves behind a wake in the air. The aircraft wake is generated from the wings of the aircraft and consists of two counter-rotating swirling rolls of air which are termed the "wake-vortex" or "wake-vortices". Figure 1.1 shows a photograph depicting the smoke flow visualization of wake-vortices generated by a Boeing 727. This visualization has been made possible by using wingtip smoke generators [21]. It is to be noted that these are not contrails (i.e., condensation trail left behind by the jet exhausts). These wake-vortex pairs stretch for several miles behind the aircraft and may last for several minutes. The strength of these vortices depends on several factors, prominent amongst which are the weight, size and speed of the aircraft. The strength generally increases with the weight of the aircraft. The rate of decay of the vortices depend on the prevailing weather conditions. Typically, vortices last longer in calm air and shorter in the presence of atmospheric turbulence.

The study of these vortices is very important for aircraft safety [22, 23, 24]. The rapid

Figure 1.1. B-727 vortex study photo (Courtesy: NASA Dryden Flight Research Center)

swirling of air in a vortex can have a potentially fatal effect on the stability of a following aircraft. Past incidents and tests have repeatedly shown that even a commercial aircraft can be thrown out of control if it follows too close behind a large aircraft such as a Boeing 747. What makes the problem worse is that these wake-vortices are normally invisible and the pilots have no way to find out if they are heading into one. The infamous US Air Flight 427 (Boeing 737) disaster [25] which occurred on September 8, 1997 is attributed to this phenomenon, wherein the aircraft encountered the wake-vortices of a preceding Boeing 727. The more recent American Airlines Flight 587 (Airbus A300) disaster [26] on November 12, 2001 in New York is also believed to be, at least partially, a result of wake-vortex encounter from a preceding Boeing 747. For this reason, the FAA and the International Civil Aviation Organization (ICAO) have laid down strict rules about the permitted spacing between aircraft, based on their size. In instrument flying conditions, aircraft may follow no closer than three nautical miles, and a small aircraft must follow at least six nautical miles behind a heavy jet such as a Boeing 747.

These separations are extremely conservative; while they do not always completely avoid the effects of wake-vortices, they are sufficient to be safe in most meteorological conditions.

Nearly all airline pilots have had encounters with vortices, usually on the final approach to airports [27]. These are experienced as a buffeting of the aircraft. While of little concern to passengers and crew who are wearing seat belts at this stage, pilots regularly report minor injuries to crew members who are standing up or moving around the cabin. Although the conservative ICAO regulations on aircraft spacing have greatly reduced the risk of serious accidents, they come at the cost of reduced airport capacity. The spacing between aircraft is very conservative to compensate for the inaccurate prediction of the strength and the position of the wake-vortices, and is often several times more than the actual safety limit, thus reducing the number of take-offs and landings that can be performed safely every hour.

To tackle this problem of reduced airport capacity, which is becoming increasingly important, the NASA researchers have designed a system to predict aircraft wake-vortices on final approach, so that the aircraft can be spaced more safely and efficiently. This technology is termed AVOSS or Aircraft VOrtex Spacing System (AVOSS) [28].

AVOSS determines how winds and other atmospheric conditions affect the wake-vortex patterns of different types of aircraft. The system uses laser radar, or lidar technology, to confirm the accuracy of these forecasts. This information is processed by computers, which can then provide safe spacing criteria automatically. The AVOSS concept integrates the output from a number of subsystems such as the weather, wake prediction, and wake sensors (Figure 1.2). Weather gear at the airport measures the prevailing winds, temperatures and turbulence levels. The equipment also makes short term predictions of potential weather changes. AVOSS then establishes an invisible aircraft approach corridor with multiple wake prediction windows. This information is fed into the wake prediction subsystem which predicts the time it will take for

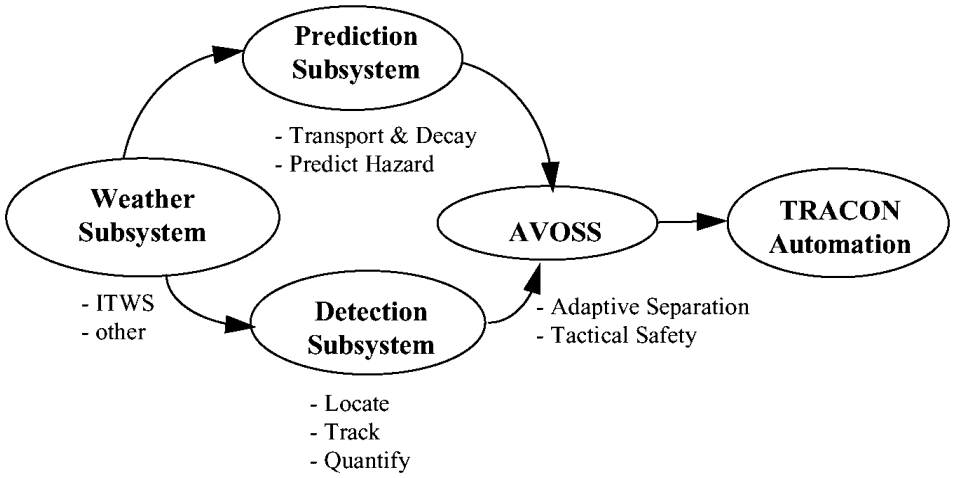


Figure 1.2. AVOSS subsystem architecture [29]

wake-vortices to decay or drift away from the flight path. The prediction subsystem calculates spacing criteria and potential runway capacity. It then provides appropriate aircraft separation distances for the next 30 minutes. The system then predicts the time it will take for the wakes to clear the corridor at multiple windows. The most conservative time is taken as the final spacing time. The end result is thus automatic and is considered sufficiently accurate, safe and efficient for the air-traffic controllers.

AVOSS has been extensively tested at the Dallas/Fort Worth (DFW) airport since 1997. Further details and developments on AVOSS can be found in references [29, 30, 31, 32].

In spite of carrying out a rigorous simulation of the wake-vortices, AVOSS does not implement any system for their visualization. It only provides the Air-Traffic Control (ATC) with the aircraft spacing time for each aircraft which is all the current systems can handle. Thus, at present, it is unable to provide alternate trajectories for the take-off and landing of aircraft. The system presented here not only predicts (computationally) where the vortices will be, but also displays them in the VR system.

1.2.2 Computational Fluid Dynamics Visualization

Computational Fluid Dynamics (CFD) [33] is a simulation technique which allows the numerical prediction of fluid flow based on the governing equations. CFD has become an indispensable tool in aerodynamics for both designing and analyzing flows for which no analytical solutions are available. It constitutes a "third" approach in the philosophical study and development of the whole discipline of fluid dynamics alongside *experiment* and *theory*. However, the sheer complexity and size of today's CFD calculations pose innumerable problems for the visualization of their results. It is common for a typical unsteady run over a full-scale aircraft geometry to take several days or weeks even on a modern state-of-the-art supercomputer. To let such an expensive and time-consuming simulation go unchecked is tantamount to taking a big risk should the solution diverge or give unexpected results due to incorrectly set parameters. Hence, there is a great need to be able to interactively visualize the results from these simulations while they are being performed. This can enable the detection of errors in the simulation earlier which can possibly be rectified by changing the solver parameters. Also, the amount of data these simulations generate can run into hundreds of gigabytes and become prohibitive for storage. An efficient way to retrieve and post-process this data interactively from a remote machine as and when they are generated can result in huge savings of both time and storage.

A computational steering approach not only makes the interactive visualization of CFD simulations possible, but it also makes it more efficient due to the "dimensional reduction" in the data to be visualized. This is due to the fact that the simulation data lives in a higher dimensional space (3D, or 4D when time is included) than the data needed for visualization (2D for iso-surfaces, and 1D for chord plots). This data can be obtained directly from the parallel simulation in a scalable manner as opposed to the traditional way of consolidating the

data into a file and post-processing it. Thus, not only is there an order of magnitude less data to deal with which reduces communication time, but it is also computed faster as it is done in parallel.

To develop such an interactive CFD system [34], the computational steering approach is used to augment an existing parallel flow solver, Parallel Unstructured Maritime Aerodynamics (PUMA) [35], which is written in C using the Message Passing Interface (MPI) library. This enhanced system can be used to monitor and steer several large flow simulations over helicopter and ship geometries, thus providing the user with a fast and simple debugging and analysis mechanism. The flow and convergence parameters can be changed dynamically without having to abort and restart the simulation. Further, this CFD system can be coupled to a VR setup to obtain near real-time visualization of the 3D solution data in stereoscopic mode.

1.3 Thesis Flow

This thesis is divided into 7 chapters. Integrated Virtual Environments and Parallel Computing are introduced in Chapters 2 and 3, respectively. A detailed discussion on computational steering is then provided in Chapter 4, followed by the details of wake-vortex simulations in Chapter 5 and the details of CFD simulations in Chapter 6. Finally, conclusions are drawn in Chapter 7.

Chapter 2

Integrated Virtual Environments

This chapter discusses the basics of Virtual Reality and describes the various hardware and software available to create an Integrated Virtual Environment (IVE). Potential applications of IVE in future Air-Traffic Control systems are also discussed here.

2.1 Virtual Reality

2.1.1 Definition

Virtual Reality (VR) represents a Human-Computer Interface (HCI) technology that is designed to leverage our natural human capabilities. Today's familiar interfaces–the keyboard, mouse, monitor, and the Graphical User Interface force us to adapt to working within tight, unnatural, two-dimensional constraints. VR changes that by allowing the user to interact with real-time 3D graphics in a more intuitive and natural manner. This approach enhances our ability to understand, analyze, create and communicate. VR is a way for humans to visualize, manipulate and interact with computers and extremely complex data.

However, the term Virtual Reality is often considered to be unsuitable since it happens to be an oxymoron consisting of contradictory terms. Some other terms that have been used to describe work in this field include Virtual Environments (VE), Synthetic Environments, Artificial Reality, and Simulator Technology.

A VR system that interactively simulates a particular environment or scenario in real-time is termed an IVE. An IVE lets the user experience the data directly. It generates an

interactive simulation by providing the user a graphics representation of a virtual environment via an output device. Today's advanced interfaces let the user look and move around inside a virtual environment, drive through it, lift items, hear sounds, feel objects, and in other ways experience graphical objects and scenes much as one might experience objects and places in the physical world. As a result, VR serves as a problem-solving tool that lets us accomplish what was previously deemed impossible. It is also a new medium of communication, and ultimately, an artistic tool.

Wickens and Baker [36] have defined VR as incorporating the following five features,

1. The system should be setup for three-dimensional viewing using perspective transforms and/or stereoscopic displays.

2. The display should be dynamic as such a display appears more realistic than a series of static images of the same geometry.

3. The system should support closed-loop interaction where the user is the active navigator as well as the observer.

4. The system should have an inside-out (ego-referenced) frame of reference where the image of the VE displayed is from the point of view of the user's head as it is positioned in the world.

5. The system should support multimodal interaction via several input techniques and feedback through several sensory modalities.

2.1.2 Classification of VR Systems

VR systems are distinguished based on the way they interface with the user. Some of the common VR systems are (adapted from Isdale [37]):

- **Desktop VR** or **Window on World Systems (WoW)**: These systems use a conventional computer monitor to display the virtual world. This is sometimes called "Desktop VR" or a "Window on a World" (WoW). This is one of the earliest concepts that traces its lineage back through the entire history of computer graphics. In 1965, Ivan Sutherland laid out a research program for computer graphics in a paper called "The Ultimate Display" [38] that has driven the field for the past three decades.

- **Video Mapping**: This is a variation of the WoW approach that merges a video input of the user's silhouette with a 2D/3D computer graphic. The user watches a monitor that shows their body's interaction with the world. This concept is used in popular computer games such as Doom and Quake.

- **Immersive Systems**: The ultimate VR systems completely immerse the user's personal viewpoint inside the virtual world. These *immersive* VR systems are often equipped with a Head Mounted Display (HMD). The HMD is a helmet or a face mask that holds the visual and auditory displays. The helmet may be free ranging, tethered, or it might be attached to some sort of a boom armature. A variation of the immersive systems use multiple large projection displays to surround the user in a "cave" or room with the virtual world projected on the walls from the user's perspective. An advantage of this approach is that other viewers can share this environment.

- **Telepresence**: Telepresence [39] is a variation on visualizing complete computer generated worlds. This technology links remote sensors in the real world with the senses of a human operator. The remote sensors might be located on a robot. Fire fighters use remotely operated vehicles to handle some dangerous conditions. Surgeons are using very small instruments on cables to do surgery without cutting a major hole in their patients.

The instruments have a small video camera at the business end. Robots equipped with telepresence systems have already changed the way deep sea and volcanic exploration is done. NASA plans to use telerobotics for space exploration. There is currently a joint US/Russian project researching telepresence for space rover exploration.

- **Augmented/Mixed Reality Systems**: Merging of Telepresence and Virtual Reality systems gives the Augmented Reality systems [40]. Here the computer generated inputs are merged with telepresence inputs and/or the users view of the real world. A surgeon's view of a brain surgery is overlaid with images from earlier CAT scans and real-time ultrasound. A fighter pilot sees computer generated maps and data displays inside his fancy helmet visor or on cockpit displays.

- **Fish Tank Virtual Reality**: The most pervasive display "fixture" is the conventional computer monitor which can be augmented to view stereoscopic, three dimensional environments as well as the more traditional interfaces. Such an immersive effect obtained by using a pair of stereographic Liquid Crystal Display (LCD) shutter glasses attached to a mechanical head tracker to look at the display on the monitor is referred to as "fish-tank" VR, as it is analogous to looking through the screen into another environment. The resulting system is superior to simple stereo-WoW systems due to the motion parallax effects introduced by the head tracker. However, because of its limited field of view and poor sense of immersion, it has been generally ignored by most Virtual Reality researchers.

VR has become a popular field of study in recent years. Books by Stuart [41] and Vince [42] discuss the various intricacies and applications related to VR from the extensive literature that has been developed.

2.2 VR Hardware

2.2.1 Input and Tracking Devices

There are a variety of input and tracking devices available for VR systems. Most can easily be added to an existing VR installation like a CAVE or a RAVE. Various motion trackers, gloves, wands, 6D spaceballs, force/haptic feedback device and 3D digitizers are just a few examples of these devices. Motion tracking devices are an important part of VR hardware as they automatically determine the viewer and eye location and adapt the display perspective accordingly. They also offer an easy and intuitive way of navigation in a virtual environment by providing velocity vectors of the user's motion to the simulation software. Tracking technologies are broadly classified as [41]:

1. **Mechanical tracking systems**: These systems are physically connected to the user and measure the user's position mechanically using the jointed linkages. For example, Goniometers are used in Sutherland's HMD system [43], and FakeSpace Systems uses a BOOM (Binocular Omni-Orientation Monitor) system. Although these systems are responsive and accurate, they restrict the movement of the user due the presence of rigid linkages.

2. **Magnetic tracking systems**: These systems are based on alternating current (AC) emitters or direct current (DC) emitters. The more commonly used AC systems typically have three electromagnetic coils that are wound around mutually perpendicular axes on ferrite cores on both the emitters and the sensors. When a current is sent to the three coils of the emitter in sequence, a magnetic field is generated which in turn generates currents in the coils of the sensor. The strength of the current generated depends on the distance between the sensor and the emitter. The nine currents induced during this three-state

cycle are then processed using an algorithm to calculate the position and orientation of the sensor with respect to the emitter. Since the sensor is usually connected only via a thin wire and the range of the emitter is at least 8-10 feet, there is a lot more room for user movement as compared to the other tracking systems. Magnetic tracking systems are therefore the most widely-used tracking systems for virtual environments. The Flock of Birds from Ascension Technology [44] is an example of one such popular system.

3. **Acoustic tracking systems**: These systems typically use ultrasonic frequencies for determining the position of the sensor. The three main approaches to acoustic tracking are time of flight, time delay, and phase-coherent systems. Acoustic waves, which travel at the speed of sound (approximately 347 m/s), are generated in acoustic trackers by an emitter. The most commonly used time-of-flight systems emit acoustic waves in pulses at known times from the emitter, and measure the time it takes takes for the waves to reach the sensor. This measured time is used to compute the distance between the emitter and the sensor. By using three emitters spread in space, the three different distances of the sensor from each of the emitters is used to compute the exact position of the sensor. However, the line of sight between the emitters and the sensor must be maintained to get the correct position, as reflection of acoustic waves by hard walls and other surfaces can induce significant errors. Power Glove by Nintendo/Mattel and 3D Mouse by Logitech are examples of popular acoustic tracking systems.

4. **Optical tracking systems**: Optical tracking makes use of small markers on the body, either flashing infrared LEDs or small infrared-reflecting dots. A series of two or more

cameras surround the subject and pick out the markers in their visual field. Image processing technology is then used to correlate the marker positions in the multiple viewpoints and use the different lens perspectives to calculate a 3D co-ordinate for each marker. Since the system is computationally intensive, six degree-of-freedom (DOF) systems of this type are not very common as yet. Most of the systems operate in a batch mode, where the trajectories of the markers are captured live, followed by a period of analysis to calculate 3D space positions from the 2D images. Optoelectric tracker at the University of North Carolina is an example of such a system.

2.2.2 CAVE/RAVE

The CAVE (Cave Automatic Virtual Environment) is currently the most popular, as well as sophisticated VR installation for scientific and artistic projects. It was developed in 1992 by a team headed by Tom DeFanti and Dan Sandin at the Electronic Visualization Laboratory (EVL) of the University of Illinois, Chicago [45, 46, 47] and is now being produced commercially by FakeSpace Systems [48].

The CAVE is a projection-based VR system that surrounds the viewer with 1 to 6 screens. The screens are arranged in a cube made up of three rear-projection screens for walls and a down-projection screen for the floor; that is, a projector overhead points to a mirror, which reflects the images onto the floor. A viewer wears stereo shutter glasses and a six DOF head-tracking device. As the viewer moves inside the CAVE, the correct stereoscopic perspective projections are calculated for each wall. A second sensor and buttons in a wand held by the viewer provide interaction with the virtual environment. A diagram of the CAVE environment is shown in Figure 2.1. A screenshot depicting a four-walled CAVE in use is shown

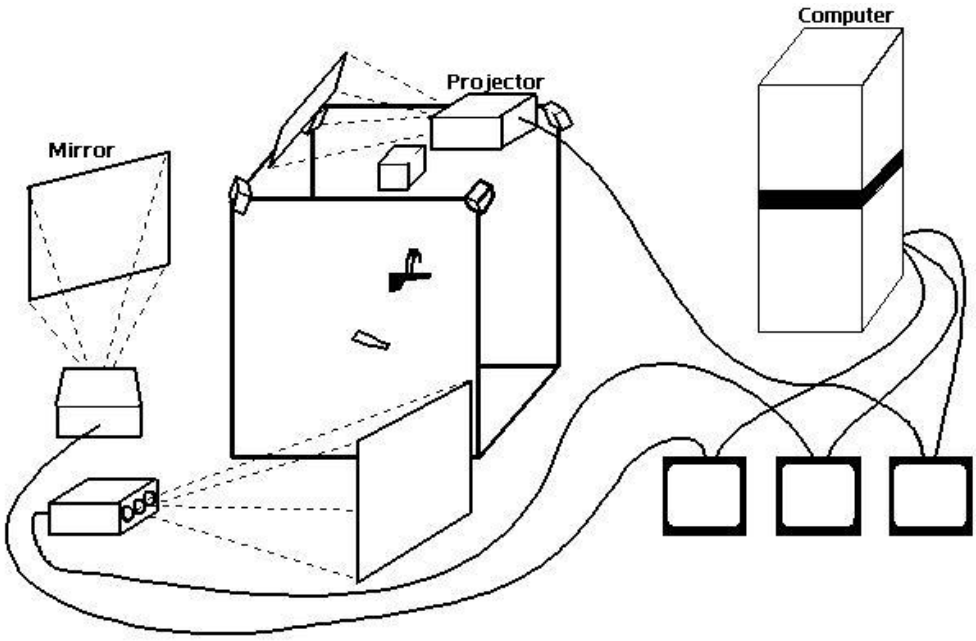


Figure 2.1. CAVE diagram (Courtesy EVL [49])

in Figure 2.2. The CAVE has been used for wide variety of applications, some of them being architectural walkthroughs, cosmic exploration, fractal exploration, viewing the behavior of algorithms implemented on parallel machines, and understanding weather and molecular dynamics [15].

The RAVE (Reconfigurable Automatic Virtual Environment) and Immersadesk are cheaper alternatives to the CAVE. The RAVE is reconfigurable and can consist of one to four projected screens, whereas Immersadesk consists of a single projected screen. HMDs, once very popular devices for VR, are now losing ground to sophisticated displays like RAVEs and CAVEs. Fig. 2.3 shows a snapshot of the Penn State Institute for High Performance Computing Applications (IHPCA) RAVE. The IHPCA RAVE uses the Flock of Birds magnetic tracking system. For the input, a pair of Crystal Eyes shutter glasses and a wand both fitted with an electromagnetic sensor are used (Fig. 2.4). The wand is the equivalent of a mouse with a joystick and three buttons.

Figure 2.2. Screenshot of a CAVE application (Courtesy: SARA, Netherlands)

Figure 2.3. Snapshot of the IHPCA RAVE

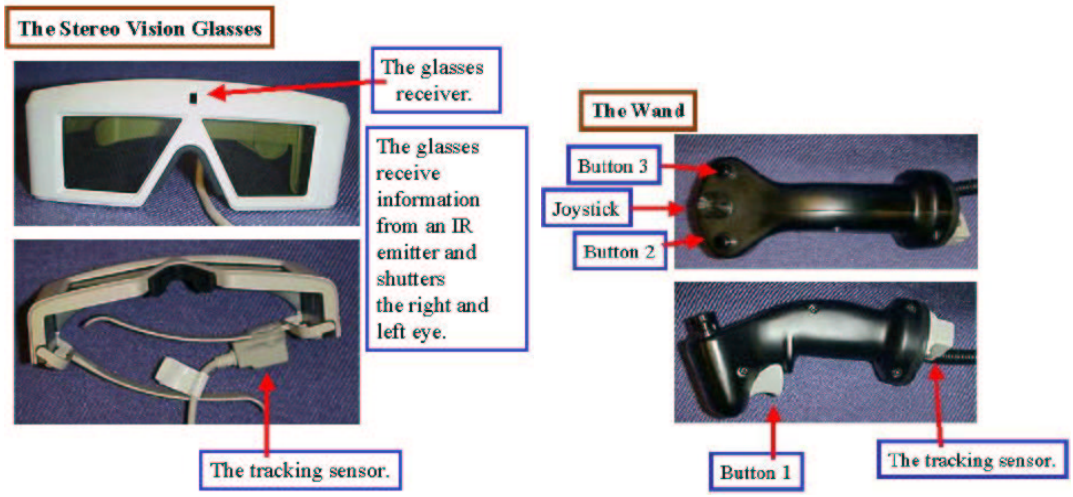


Figure 2.4. Crystal Eyes Shutter Glasses and the Wand (Courtesy Flurchick [50])

2.3 VR Software

Owing to the complexity of VR and the various hardware devices involved in their creation, specialized software packages and Application Programming Interfaces (APIs) are essential to make the development of VR applications easier.

2.3.1 CAVELib

The CAVELib is an API that provides general support for building virtual environments for various types of immersive displays. The CAVELib API forms a building block used to create applications for a variety of virtual environments. It provides functions that can be used by programmers to create robust programs for virtual display devices or desktops. CAVELib was originally developed by Dave Pape of EVL, University of Illinois, and is now marketed commercially by VRCO [51].

CAVELib handles program execution via registered callbacks. CAVE programs run as a collection of processes which are created and managed by CAVELib. The programmer does not have to do any process creation (forking) explicitly. CAVELib configures the display device,

synchronizes processes, draws stereoscopic views, creates a viewer-centered perspective and provides basic networking between remote VEs. CAVELib is available for several hardware platforms and hides all the operating system and display specific code from the user enabling the development of portable applications that can run on a wide variety of hardware and display devices. It uses a resource configuration file that can be modified to change display and input devices. It automatically generates the right viewing frustum for each wall and applies it to the scene and also converts the coordinates from the tracking system (world coordinates) to the coordinate system of the 3D scene (display coordinates). CAVELib uses processes to obtain simple parallelization in the calculation process by using one process for the generation of the projection for each eye of each wall and another for getting data from the tracking device. For example, a standard 4-walled CAVE with stereo support (2 eyes) needs 8 projection matrices and thus 8 processes for rendering each frame of animation. These processes may need access to common data which are provided by allocating shared memory. The programmer needs to explicitly allocate shared memory and set/clear locks.

CAVELib provides an API to control all its functions, which are mainly for the query of the tracking system and the input devices. It provides functions for determining the user's head position and orientation, the wand position and orientation, as well as the state of the wand buttons and joystick. It also provides functions for determining the time and frame-rate. All distributions of CAVELibs are capable of running in simulator mode. The simulator uses the keyboard and mouse to simulate head and wand movement, in a non-stereo display on a workstation monitor. The simulator provides convenience in application development and testing using a workstation without requiring the use of a VR device. Figure 2.5 illustrates a simple CAVELib application written in C. This application draws a sphere whose radius oscillates sinusoidally with time.

```
#include <cave_ogl.h>       // CAVELib functions
#include <GL/gl.h>          // OpenGL functions
#include <GL/glut.h>        // GL Utility Library functions
#include <unistd.h>         // For usleep()

double *radius; // Pointer to shared memory

void draw(void)
{
  glClearColor(0.0, 0.0, 0.5, 0.0);
  glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);

  glColor3f(0.0, 1.0, 0.0); // Set Green color
  glutSolidSphere(*radius, 30, 30); // Draw a sphere
}

void frameUpdate(void)
{
  *radius = 1.5 * sin(*CAVETime); // Compute radius of sphere
}

int main(int argc, char **argv)
{
  CAVEConfigure(&argc, argv, NULL);
  radius = (double *) CAVEMalloc(sizeof(double)); // Allocate shared memory
  CAVEInit();
  CAVEDisplay(draw, 0); // Register drawing function
  CAVEFrameFunction(frameUpdate, 0); // Register frame update function

  // Poll every 10 microseconds to check if ESCAPE key has been pressed
  while(!CAVEgetbutton(CAVE_ESCKEY))
    {
      usleep(10);
    }

  CAVEFree(radius); // Free shared memory
  CAVEExit(); // Exit
}
```

Figure 2.5. A simple CAVELib program written in C

VR Juggler [52], being developed by the Iowa State University, is another API similar to the CAVELib (albeit open-source).

2.3.2 OpenGL

The CAVELib by itself does not incorporate any function for the actual graphics programming. To generate the actual graphics seen in the VR display, a separate graphics API like OpenGL [53] is required (as can be seen in the `draw()` function in Fig. 2.5). OpenGL is the premier environment for developing portable, interactive 2D and 3D graphics applications. Since its introduction in 1992 by Silicon Graphics Inc. (SGI), OpenGL has become the industry's most widely used and supported 2D and 3D graphics API, bringing thousands of applications to a wide variety of computer platforms. OpenGL fosters innovation and speeds application development by incorporating a broad set of rendering, texture mapping, special effects, and other powerful visualization functions. Owing to the standardization of OpenGL, developers can leverage the power of OpenGL across all popular desktop and workstation platforms, ensuring wide application deployment. Among the drawbacks, OpenGL is not object-oriented and is not scene-graph based. With the combination of OpenGL and CAVELib, it is possible to create powerful VR applications in the CAVE environment. The CAVELib does most of the configuration and initialization tasks while the OpenGL handles the display function. Mason et al. [54] is a very popular book to learn OpenGL programming.

2.3.3 Other High-level Software for VR Applications

While CAVELib and OpenGL provide the basic functionality for the development of VR applications, they are usually not sufficient. They lack important software components like

one for hierarchical scenegraph rendering, which is necessary for efficient rendering of complex scenes commonly found in most VR applications. SGI's IRIS Performer is an OpenGL-based scenegraph rendering library and is being extensively used for creating VR applications. However, a limitation of using Performer is that it only works under IRIX (SGI's proprietary operating system) and Linux operating systems.

There are also several high-level software development systems available for building high-performance, real-time, integrated 3D applications for scientific and commercial use, some of them being: WorldToolKit, MR Toolkit, dVISE and DIVE. WorldToolKit (WTK) by Sense8 Corporation [55] is the most popular amongst them all. WorldToolKit has the function library and end-user productivity tools needed to create, manage, and commercialize VR applications. With the high-level API, the programmer can quickly prototype, develop, and reconfigure the applications as required. WorldToolKit also supports network-based distributed simulations, CAVE-like immersive display options, and the industry's largest array of interface devices, such as HMDs, trackers, and navigation controllers. The system started out as a 3D modeling and simulation package in 1990, but has now matured into a full-fledged VR development toolkit. WorldToolKit is an object-oriented software toolkit that greatly simplifies programming of visual simulation and virtual reality by offering an extensive and versatile C/C++ API. WorldToolKit contains more than $1,100$ high-level function calls for developing, configuring, and controlling real-time 3D simulations. WorldToolKit is unique in that it provides a truly cross-platform development and delivery capability. WorldToolKit automatically optimizes graphics performance based on functions unique to each platform, to maximize the graphics hardware performance. WorldToolKit also integrates all of the essential elements, including high-performance graphics, intuitive interface components, 2D or spatialized 3D sound, support for multiple database and media formats, and a comprehensive list of

input/output peripherals. The system is available for a wide range of platforms running UNIX, and has a large and growing user base. All the above-mentioned toolkits work in conjunction with OpenGL for rendering the graphics.

CAVERNsoft (CAVE Research Network software) [56] is a toolkit for creating high-performance tele-immersive applications being developed by EVL, the inventors of CAVE. CAVERNsoft is designed to enable the rapid construction of tele-immersive applications and to equip previously single-user applications with tele-immersive capabilities. It provides networking and database functionality needed in a virtual reality system. CAVERNSoft toolkit allows scientists to steer a program from a virtual reality system without requiring any modification to the program. It enables interactive and immersive analysis of simulations running on remote computers. CAVEStudy [57] is another toolkit built on top of CAVERNSoft which allows interactive and immersive analysis of a simulation running on a remote computer. However, both CAVERNSoft and CAVEStudy depend on Performer for rendering, and are hence limited to SGI IRIX and Linux platforms.

2.4 IVE Systems for Air-Traffic Control (ATC)

According to Federal Aviation Administration (FAA) projections (Table 2.1), the air traffic is expected to almost double by the year 2015. If, by then, the U.S. air transportation system does not change in any significant way, there could be a major aviation accident every seven to ten days. Air-traffic control related delays already cost the airline industry an estimated US $5.5 billion annually. The FAA has for the past two decades been scrambling to replace, modernize, and improve the ATC.

The U.S. air-traffic control system is organized around three types of facilities: *airport towers*, which monitor aircraft on the ground and give take-off and landing clearances; *terminal*

Year	Passengers (Millions)	Passenger Miles (Billions)	Jet Aircraft in Fleet	Domestic Departures (Millions)
2000	639.8	661.8	5610	7.4
2005	767.4	830.3	6903	8.6
2010	931.1	1041.6	8360	9.9
2015	1139.7	1306.2	10034	11.5

Table 2.1. Commercial Air Carriers FY 1999-2010 (Source: FAA [58])

radar approach control (TRACON) facilities, which handle aircraft ascending and descending to and from airports; and *enroute centers*, which handle aircraft flying between airports at the higher altitudes [2]. ATC is a highly demanding human-machine system. With the rapid progress in real-time processing of speech and images, together with database management, an intelligent system combining satellite navigation, automated speech synthesis and recognition is expected to be introduced to replace some human functions in the foreseeable future [59]. The next logical step for the ATC would be to transform itself into an IVE which simulates a complete airport.

The FAA has declared that the existing ATC system will transition to a new system known as “free flight”. While free flight has not been precisely defined in a way that is universally accepted, the basic concept is to reduce the centralized control in the existing system to allow pilots greater freedom in choosing and altering routes, leading to reduced costs and increased capacity. In today’s system, controllers issue commands and pilots follow them. Pilots wishing to change their routes must issue requests and receive clearance from controllers. The control and responsibility for safe operation is centralized in the controllers. In contrast, free flight will allow pilots to change their routes in real-time, with controllers intervening only when necessary to ensure adequate separation. In some definitions of free flight, pilots themselves are responsible for avoiding conflicts in simple situations [60]. Azuma et al. [60, 61, 62] have worked extensively on advanced human-computer interfaces and visualization tools to assist

in the free flight air-traffic management. They have created a VR-based test-bed using C and WorldToolKit 7 for the visualization of conflicts that may occur in a free flight scenario. While their work may look similar to a part of the proposed work in this thesis, the fact that they do not deal with any vortex-wake problem whatsoever, makes it actually different.

To facilitate this and to ease the pressure on the ATC, NASA has also been working with a team of seven industry partners to help develop the "Highway In The Sky" (HITS) system [63]. With this IVE system, pilots will follow a preprogrammed destination on a "virtual highway" in the sky, drawn on a highly intuitive, low-cost flat panel display. This display might even be a part of the cockpit window, thus augmenting the real display seen through the glass with the computer-generated path superimposing it. In addition to transforming the cockpits, the technology developed by the team will redefine the relationship between pilots and the ATC and fundamentally change the way future general aviation pilots fly. This technology is expected to significantly increase freedom, safety and ease-of-flying by providing pilots with affordable, direct access to information needed for future free flight air-traffic control systems. Pilots will have the ability to safely determine their routes, speeds and proximity to dangerous weather, terrain and other airplanes. This display system and other equipment will provide intuitive situational awareness and enough information for a pilot to perform safely, with reduced workload, in nearly all weather conditions. A multifunction display of position navigation, terrain map, weather and air-traffic information is expected, in addition to the digital radios which will send and receive flight data in real-time.

NASA's FutureFlight Central [64] is a national ATC test facility dedicated to solving the present and emerging capacity problems of the nation's airports. Designed in collaboration with the Air Transportation Association (ATA), FAA, National Air-Traffic Controllers Association (NATCA) and Supervisors Committee (SUPCOM), NASA FutureFlight Central is operated and

Figure 2.6. An IVE system for ATC: NASA FutureFlight Central's tower [64]

managed by NASA personnel, including experts in ATC, computer graphics, human factors and large-scale simulations. The two-storey facility is an IVE offering a 360-degree full-scale real-time simulation of an airport (Figure 2.6), where controllers, pilots and airport personnel can interact to optimize operating procedures and test new technologies.

Chapter 3

Parallel Computing and Beowulf Clusters

Parallel Computing is a strategy for performing large, complex computational tasks faster. Large tasks can either be performed serially, one step following another, or if possible can be decomposed into smaller tasks to be performed simultaneously—that is, in parallel. Thus, the terminology "parallel computing" refers to the use of multiple computers or processors working together on a common task.

3.1 Taxonomy of Parallel Computer Architectures

Currently, the most popular nomenclature for the classification of parallel computer architectures is that proposed by Flynn [65] in 1966 (see Fig. 3.1). Flynn chose not to examine the explicit structure of the machine, but rather how instructions and data flow through it. Specifically, the taxonomy identifies whether there are single or multiple "streams" for data and for instructions. The term stream refers to a sequence of either instructions or data operated on by the computer. Single instruction refers to the fact that there is only one instruction stream being acted on by the Central Processing Unit (CPU) during any one clock tick; single data means, analogously, that only one data stream is being employed as input during any one clock tick. The four possible categories of Flynn's taxonomy are:

1. **Single Instruction Single Data (SISD)**: SISD machines are conventional serial computers that process only one stream of instructions and one stream of data. A SISD machine is equivalent to a von Neumann machine. Instructions are executed sequentially but may be overlapped by pipelining—most SISD systems are now pipelined. SISD computers

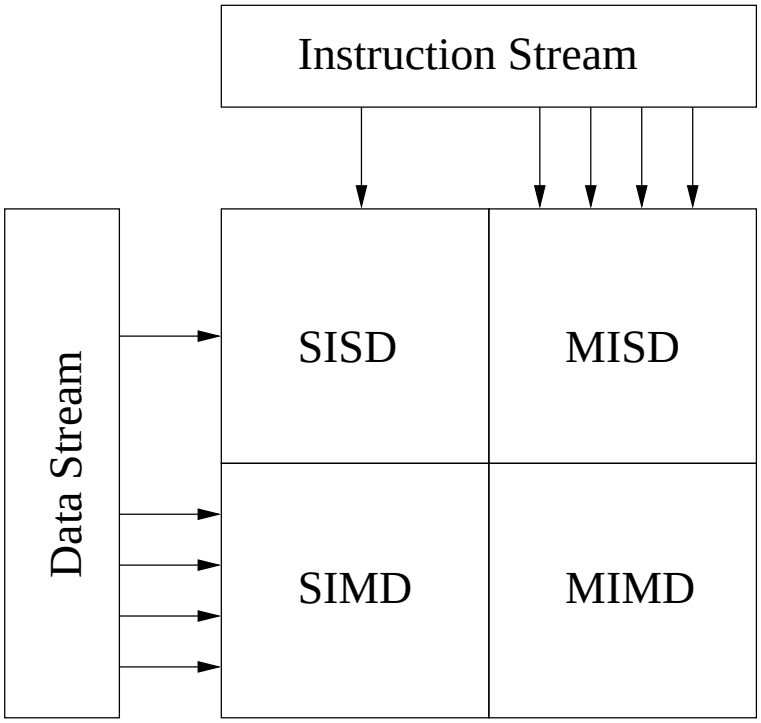


Figure 3.1. Flynn's Classification

may have several functional units, examples include extra mathematics coprocessors, vector units, graphics processors and I/O processors. As long as there is only one CPU the architecture remains SISD. Examples of typical SISD machines are the widely-used single-processor desktop PCs and workstations.

2. **Single Instruction Multiple Data (SIMD)**: This category encompasses computers which have many identical interconnected processors under the supervision of a single control unit. The control unit transmits the same instruction, simultaneously, to all processors. The topology of the communications network is not addressed by Flynn's taxonomy. All the processing elements simultaneously execute the same instruction and are said to be "lock-stepped" together. Each processor works on data from its own memory and hence on distinct data streams. Every processor must be allowed to complete its instruction before the next instruction is taken for execution. Thus, the execution of instructions is

said to be synchronous. Examples of SIMD machines are the ILLIAC-IV, Connection Machine (CM-1 and CM-2) and MasPar.

3. **Multiple Instruction Single Data (MISD)**: MISD machines have many processing elements, all of which execute independent streams of instructions. However, all the processing elements work on the same data stream. There are two ways in which this can be done: firstly the same data item can be fed to many processing elements each executing their own stream of instructions. Alternatively, the first processing element could pass on its results to the second processing element and so on, thus forming a macro-pipeline. The only known example of a computer capable of MISD operation is the C.mmp [66] built by Carnegie-Mellon University, which is reconfigurable and can also operate in SIMD and MIMD modes.

4. **Multiple Instruction Multiple Data (MIMD)**: Most multiprocessor systems and multiple computer systems can be placed in this category. A MIMD computer has many interconnected processing elements, each of which have their own control unit. The processors work on their own data with their own instructions. Tasks executed by different processors can start or finish at different times. They are not lock-stepped, as in SIMD computers, but run asynchronously. Examples of MIMD machines are Intel Paragon, IBM SP2, Cray C90, and clusters of workstations running programs based on Parallel Virtual Machine (PVM) or Message Passing Interface (MPI).

   MIMD systems can be further divided coarsely based on memory organization:

   (a) **Shared Memory**: Shared-memory systems can be divided into the following three sub-categories.

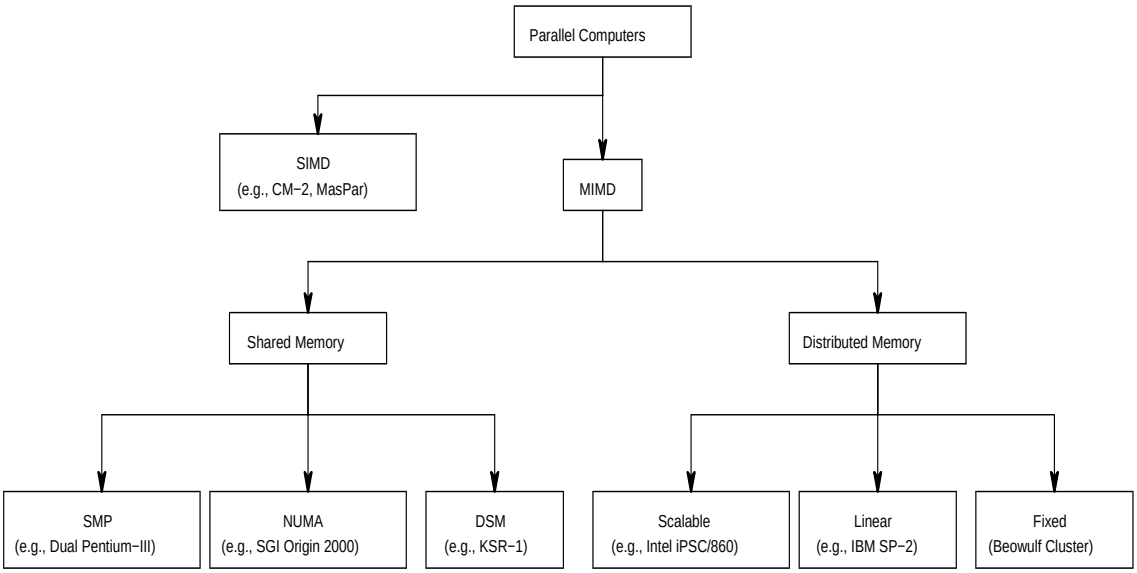


Figure 3.2. Classification of Parallel Computers

i. **Symmetric Multi-Processing (SMP)**: In SMP systems, the memory is physically shared, and all processors access the memory equally at equal speeds.

ii. **Non-uniform Memory Access (NUMA)**: In NUMA systems, the memory is physically shared, but not distributed in a 1:1 relation with the processors. Due to the non-uniform distribution, the access to different portions of the memory may require significantly different times.

iii. **Distributed Shared Memory (DSM)**: In DSM systems, the memory is distributed among the processors, but the system gives the illusion that it is shared.

(b) **Distributed Memory**: Distributed memory systems can be divided into the following three sub-categories based on the network connection between the multiple processors.

i. **Fixed**: In fixed distributed memory systems, the number of connections is fixed as more processors are added (e.g., Ethernet-connected workstations, since the Ethernet interconnection is a single resource that all processors share).

ii. **Linear**: In linear distributed memory systems, the number of connections grows linearly with the number of nodes (e.g., mesh-connected multicomputers such as the Intel Paragon).

iii. **Scalable**: In scalable distributed memory systems, the number of connections grows as $P \log P$ or greater where P is the number of processors (e.g., hypercubes such as the Intel iPSC/860).

3.2 Parallelization Strategies

Parallel computing paradigms fall into two broad classes: *functional decomposition* and *domain decomposition*. These classes reflect how a problem is allocated to processes. Functional decomposition divides a problem into several distinct tasks that may be executed in parallel; one field where this is currently popular is that of Multidisciplinary Design Optimization (MDO) [67]. In MDO, the objective function involves numerical simulations from more than one physical discipline which can be efficiently implemented in parallel. On the other hand, domain decomposition distributes data across processes, with each process performing more or less the same operations on the data. Domain decomposition is to be distinguished from the SIMD model: in the SIMD model, each process executes the same instructions at the machine level–that is, all processes proceed in lockstep, doing exactly the same operations on different data. This kind of parallelism (sometimes called fine-grain parallelism) is usually found at the loop level, whereas domain decomposition is applied at the problem level. In the *Single Program Multiple Data* (SPMD) parallelism, the same code is replicated to each process. One process acts as the master and collects the results from all other processes, thus doing slightly more work than the other processes. Each processor works on its section of the problem and is allowed to exchange information (data in local memory) with other processors.

There are several reasons for the choice of parallel computing over a conventional single processor computation. Single processor computers have a limited amount of memory and cannot process data faster than their rated speed (which is also limited by the current semiconductor technology). Parallel computing not only allows us to solve problems that do not fit on a single processor machine, but is also capable of obtaining the solution in a short amount of time (dependent on the size of the problem and the number of processors employed). Thus, one can run larger problems than were previously possible, and obtain solutions to these problems much quicker. Parallelism can be achieved by:

- breaking up the task into smaller tasks;
- assigning the smaller tasks to multiple workers to work on simultaneously; and
- coordinating the workers.

Traditionally, programs that have been written for serial computers execute only one instruction at a time, use one processor, and process instructions dependent on how fast data can move through hardware. The current fastest processors execute approximately 2.5 billion instructions per second [68]. One might think that this is fast enough, but there are several classes of problems that require faster processing such as various simulation and modeling problems, problems dependent on computations/manipulations of large amounts of data, and grand challenge problems such as climate modeling, fluid turbulence, human genome, etc. Computers that support parallel computing are broadly referred to as High Performance Computing (HPC) systems.

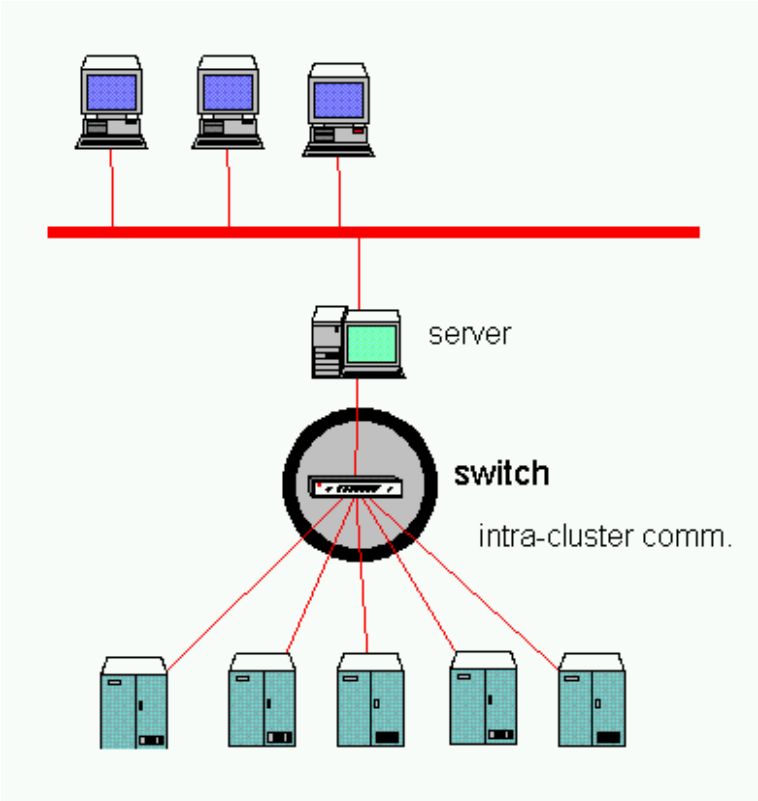


Figure 3.3. Schematic of a typical Beowulf cluster

3.3 Beowulf Clusters

The Beowulf cluster is a multi-computer MIMD architecture which can be used for parallel computations. The word “Beowulf” was borrowed from one of the earliest surviving epic poems written in English, which tells the story about a hero (in this case the “commodity computer”) of great strength and courage who defeats a monster (in this case the “existing supercomputers”). Based on the previously discussed taxonomy, it is a fixed distributed memory MIMD machine. A typical Beowulf cluster consists of one server node, and one or more client nodes connected together via some network. A schematic of a typical Beowulf cluster is shown in Figure 3.3.

Figure 3.4. COst effective COmputing Array-2 (COCOA-2)

The first Beowulf cluster was built by NASA in 1994 [69] and consisted of 16 486DX4-100 MHz machines each with 16 MB of memory—the best available commodity hardware of the time. Beowulf is a system built using commodity personal computers, standard ethernet adapters, and switches. It uses Cost-effective Off-The-Shelf (COTS) components. It does not contain any custom hardware components and is trivially reproducible. Beowulf also uses commodity software such as the Linux operating system [70], PVM and MPI, and other widely available open-source software. A Beowulf system behaves more like a single machine rather than many workstations as the server node controls the client nodes transparently. Today, the price-performance (or dollars/MFLOPs) for these cheap clusters is far better (often 10 times or more) than for a typical supercomputer from IBM, SGI or Hitachi, making them extremely attractive for small organizations and research groups.

The COst effective COmputing Array (COCOA) [71] is a Penn State Department of

Aerospace Engineering initiative to bring low cost parallel computing to the departmental level. COCOA is a 50-processor cluster of off-the-shelf PCs connected via fast-ethernet (100 Mbit/sec). The PCs are running RedHat Linux version 7.0 with MPI for parallel programming and DQS [72] for queuing the jobs. Each node of COCOA consists of Dual 400 MHz Intel Pentium II Processors in SMP configuration and 512 MB of memory. A single Baynetworks 24-port fast-ethernet switch with a backplane bandwidth of 2.5 Gbps was used for the networking. The whole system cost approximately $\$100,000$ (1998 US dollars). COCOA-2 [73] (Fig. 3.4), a successor to COCOA was also recently built at the cost of $\$60,000$, and is a 40-processor rackmounted cluster (dual PIII-800s) with 20 GB memory connected via two fast-ethernet adaptors (using Linux channel bonding). Each of the clusters have 100 GB of usable disk space. COCOA and COCOA-2 were built to enable the study of complex fluid dynamics problems without depending on expensive external supercomputing resources. To get even 50,000 hrs of CPU time in a supercomputing center is difficult, while COCOA can offer more that 400,000 CPU hours annually, and the commodity processors are often similar in performance to special purpose processors used in the supercomputers.

3.4 Message Passing Interface

The Message Passing Interface (MPI) [74] is a standard industry/academic API for message passing libraries, designed to be a standard for message passing in parallel computing. Message passing is a paradigm used widely on certain classes of parallel machines, especially those with distributed memory.

The message passing paradigm distributes all the data among several processes with access to local data only. To access non-local (remote) data, a message must be explicitly sent and received, i.e., both the sending and receiving process must participate in the communication.

This paradigm has been extremely successful for two main reasons [75]:

1. *Generality*: The message passing paradigm is applicable to all kinds of parallel architectures from shared-memory supercomputers to distributed-memory computers with high-speed communication subsystems to heterogeneous workstation clusters on slow networks.

2. *Performance*: Because non-local data must be accessed by the non-trivial and expensive way of setting up and receiving messages, the programmer is painfully aware of the cost of using such data and is compelled to consider better ways to distribute their data. Also, this paradigm gives the programmer maximum control over how and when their data is sent, which is lacking in the other paradigms. This makes it possible to write code that is more tolerant of slow networks or heterogeneous processors.

Although there are many variations, the basic concept of processes communicating through messages is well understood. Over the last ten years, substantial progress has been made in casting significant applications in this paradigm and each vendor has implemented its own variant. In 1994, an MPI standard was adopted which attempts to establish a practical, portable, efficient, and flexible standard for message passing [74]. The main advantages of having a message-passing standard are portability and ease-of-use. In a distributed memory communication environment in which the higher level routines and/or abstractions are build upon lower level message passing routines, the benefits of standardization are particularly apparent. Furthermore, the definition of a message passing standard provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases provide hardware support for, thereby enhancing scalability.

There are several open-source MPI libraries available that run on Beowulf clusters, the

most prominent being MPI CHameleon (MPICH) [76] being developed by Argonne National Lab, and the Local Area Multicomputer MPI (LAM-MPI) [77] being developed by the University of Notre Dame. Most MPI libraries are written in C, and have C++ and FORTRAN 77/90 bindings. One of the major flaws of most existing MPI implementations (including MPICH and LAM-MPI) is that they are not thread-safe. Thus, it is not possible to simultaneously perform more than one set of communications by using MPI in a multi-threaded application. In a program using MPI, all parallelism is explicit and the programmer is responsible for correctly identifying parallelism and implementing the resulting algorithm using MPI functions (of the approximately 155 that are available). The number of tasks dedicated to run a parallel program is static and new tasks can not be dynamically spawned during run time. Pacheco's book on MPI [78] is a good reference to start with for parallel programming using MPI.

Figure 3.5 shows a sample MPI program written in C++. As is typical of any MPI program, it uses `if` statements based on the process rank assigned at runtime to distribute the work between different processes. Figure 3.6 shows the output when this program is run on 8 processors using the command `mpirun -np 8 ./hellompi`. The program sends a character string "Hello World" from the master processor (with rank 0) to all the slave processors (ranks 1 to 7) waiting to receive it. On receipt, all the processors print the string along with their process ids. Figure 3.7 shows a sample MPI code written in C++ to communicate a custom C++ structure containing dynamically allocated members. Here, each dynamically allocated member has to be sent and received separately as sending the entire structure by using its pointer will only send the pointers to the allocated data and not the data itself.

While there are several other parallel programming paradigms such as PVM and OpenMP available, they are not currently widely used and are not considered here.

```
#include <string.h>
#include <iostream>
#include <mpi++.h>            // For MPI:: functions

int main(int argc, char *argv[])
{
  char msg[100];

  MPI::Init(argc, argv);      // Initialize MPI
  int numCPUs = MPI::COMM_WORLD.Get_size();
  int myRank = MPI::COMM_WORLD.Get_rank();

  if (myRank == 0)            // Master processor
    {
      strcpy(msg, "Hello World");
      // Send message to each processor
      for (int i = 1; i < numCPUs; i++)
      {
        int target = i;
        int tag = i;
        MPI::COMM_WORLD.Send(msg, 100, MPI::CHAR, target, 2*tag);
      }
    }
  else                        // Slave processors
    {
      // Receive message from master processor
      int source = 0;            // Master
      int tag = myRank;
      MPI::COMM_WORLD.Recv(msg, 100, MPI::CHAR, source, 2*tag);
    }

  cout << "Proc " << myRank << " says \"" << msg << "\"" << endl;

  MPI::COMM_WORLD.Barrier();      // Synchronize communication

  if (myRank == 0)
    cout << "Bye." << endl;

  MPI::Finalize();            // Finalize MPI before exiting
  return 0;
}
```

Figure 3.5. A simple “Hello World” MPI program written in C++

```
Proc 0 says "Hello World"
Proc 1 says "Hello World"
Proc 2 says "Hello World"
Proc 3 says "Hello World"
Proc 4 says "Hello World"
Proc 5 says "Hello World"
Proc 6 says "Hello World"
Proc 7 says "Hello World"
Bye.
```

Figure 3.6. Output produced by MPI program in Fig. 3.5 when run on 8 processors

```
#include "alloc.h"            // For ALLOC* and FREE* calls
#include <mpi++.h>            // For MPI:: functions

class Sample
{
public:
  Sample(int size) { Alloc(size); }
  ~Sample() { Destroy(); }
  void Alloc(int size) {
    n = size;
    ALLOC1D(&array1, n); ALLOC1D(&array2, n); // Allocate memory for "n" elements
  }
  void Destroy() {
    if (n == 0) return;
    FREE1D(&array1, n); FREE1D(&array2, n); // Free allocated memory
    n = 0;
  }
  int     n;
  char   *array1;  // Pointer to char (for dynamic memory allocation)
  double *array2;  // Pointer to double (for dynamic memory allocation)
};

void SendSample(Sample *S, int source, int target) {
  int myRank = MPI::COMM_WORLD.Get_rank();
  if (myRank == source) {    // Source (sending) processor
      // Send the size defining the structure to the target
     MPI::COMM_WORLD.Send(&S->n, 1, MPI::INTEGER, target, 101);
      // Send each dynamically allocated member separately
     MPI::COMM_WORLD.Send(S->array1, S->n, MPI::CHAR, target, 102);
     MPI::COMM_WORLD.Send(S->array2, S->n, MPI::DOUBLE, target, 103);
    }
  else if (myRank == target) { // Target (receiving) processor
      S->Destroy();  // Clear the structure
      // Receive the size defining the structure from the source
      MPI::COMM_WORLD.Recv(&S->n, 1, MPI::INTEGER, source, 101);
      S->Alloc(S->n); // Initialize the structure based on the received size
      // Receive each dynamically allocated member separately
      MPI::COMM_WORLD.Recv(S->array1, S->n, MPI::CHAR, source, 102);
      MPI::COMM_WORLD.Recv(S->array2, S->n, MPI::DOUBLE, source, 103);
    }
}
```

Figure 3.7. MPI code written in C++ depicting communication of custom structure containing dynamically allocated members

Chapter 4

Computational Steering

While running a complex parallel program on a HPC system, one often experiences several major difficulties in observing output or information about the ongoing state of the simulation. Usually, the way the simulation is implemented severely limits the interaction with the program during the execution and makes the visualization and monitoring slow and cumbersome (if at all possible). This is especially the case if the visualization needs to be carried out on a different system (such as the RAVE). In particular, some of the questions pertaining to visualization that arise in the simulation of the wake-vortices are the following.

- How do we get the current location of the wake-vortices shed by each aircraft for visualization on the RAVE while the parallel simulation is running on a remote Beowulf Cluster?
- How do we keep the simulation on the Beowulf Cluster updated about the constantly changing weather conditions and the trajectories of the aircraft?

These questions are of critical importance to this research as the predictions by the wake-vortex code need to be known in real-time for the ATC system to take appropriate action. The activity of periodically obtaining the location of the wake-vortices is referred to as "monitoring," which is defined as the observation of a program's behavior at specified intervals of time during its execution. On the other hand, the weather condition at the airport is continuously changing and both the number and the trajectories of aircraft change as they take-off and land.

Thus, there is a need to update the simulation based on the state factors. This process of updating ongoing simulation is referred to as “steering”. The above questions can thus be rephrased as “How do we monitor and steer our wake-vortex simulation?”.

Software tools which support these activities are called (computational) steering environments. These environments typically operate in three phases: instrumentation, monitoring, and steering [4].

- Instrumentation is the phase where the application code is modified to add monitoring functionality. This phase involves “registration” of relevant application data so that it can be accessed by the remote monitoring and steering client.

- Monitoring phase requires the program to run with some initial input data, the output of which is observed by retrieving important data about the program’s state change. Analysis of this data gives more knowledge about the program’s activity.

- During the steering phase, the user modifies the program’s behavior (by modifying the input) based on the knowledge gained during the previous phase by applying steering commands, which are injected on-line, so that the application need not be stopped and restarted.

4.1 Portable Object-oriented Scientific Steering Environment (POSSE)

The steering software developed here, *Portable Object-oriented Scientific Steering Environment* (POSSE) [79], is a general-purpose, lightweight, portable software system based on a simple client[1] /server[2] model. It uses an approach similar to Falcon [4] and ALICE Memory

[1] A program that sends queries over the network and accepts corresponding responses
[2] A program that provides a network service by answering related queries

Snooper [6]. Falcon was one of the first systems to use the idea of threads and shared memory to serve registered data efficiently. ALICE is an API designed to help in writing computational steering, monitoring and debugging tools developed at Argonne National Lab. POSSE consists of a steering server on the machine running the simulation that controls registered application data, and coupled to any number of steering clients that provides the user interface and control facilities to interact with the steering server to access the data. The steering server is created as a separate thread of the application to which local monitors forward only those registered data that are of interest to steering activities. A steering client receives run-time information from the application, conveys this information to the user, accepts steering commands from the user, and transmits these commands back to the application. The communication between a steering client and a server are implemented using UNIX sockets and the threading is implemented using POSIX[3] threads. POSSE has been written completely in C++, using several of C++'s advanced object-oriented features making it fast and powerful, while hiding most of the code complexities from the user.

Figure 4.1 shows the inheritence diagram for the POSSE class structure and Fig. 4.2 shows a UML[4] diagram depicting the software design of the POSSE code. `TCPSocket` is the base class that handles all the low-level communication consisting of contiguous byte segments using UNIX sockets. `RemoteSocket` is an inherited class that extends the `TCPSocket` functionality by providing functions for higher-level communication of variables, arrays and structures. The inherited classes `DataClient` and `DataServer` then add client and server functionalities, respectively. `DataClient` provides client-side functions for accessing registered data, while `DataServer` adds server-side functionality for data registration and coherence.

[3]Portable Operating System Interface standard
[4]Unified Modeling Language: used for modeling object-oriented applications

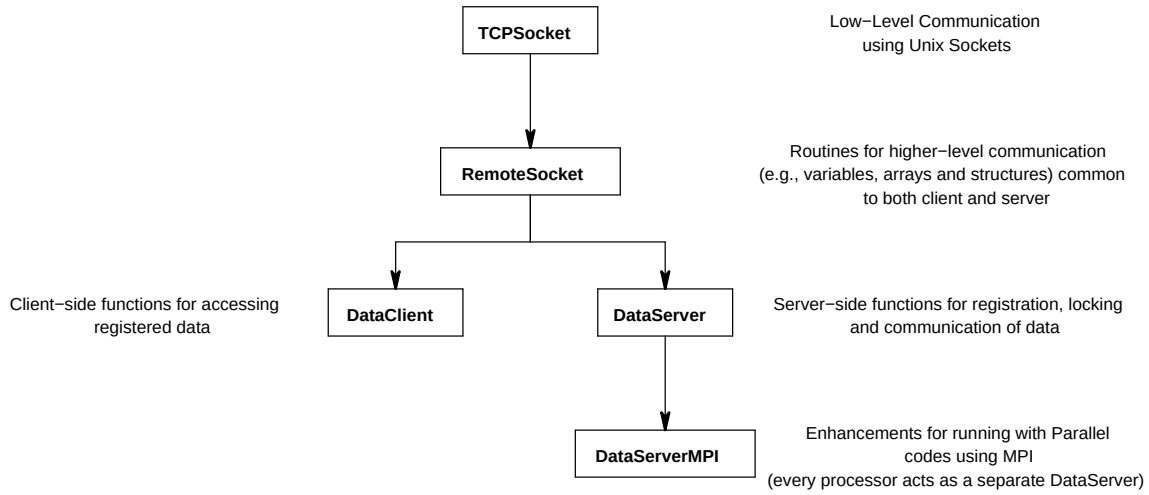


Figure 4.1. Inheritance diagram for POSSE classes

`DataServerMPI` inherits from `DataServer` and provides enhancements for parallel codes that use MPI.

Figure 4.3 shows a schematic view of how POSSE may be used. An on-going scientific simulation is running on a remote Beowulf computing cluster. Any number of remote clients can query/steer registered data from the simulation from the DataServer thread. Two clients are depicted: (1) a visualization client, and (2) a GUI client that provides a simple user interface to all registered simulation data. The visualization application can be used to interactively monitor a dataset at various time intervals. The GUI application can be used to steer the simulation by changing registered parameters.

POSSE is designed to be extremely lightweight (has very low computational overhead), portable (runs on all Win32 and POSIX-compliant UNIX platforms including Linux) and efficient. It deals with byte-ordering and byte-alignment problems internally and also provides an easy way to handle user-defined classes and data structures. It is also multi-threaded, supporting several clients simultaneously. POSSE supports parallel simulations that use the Message Passing Interface (MPI) [74] library. The biggest enhancement of POSSE over existing steering

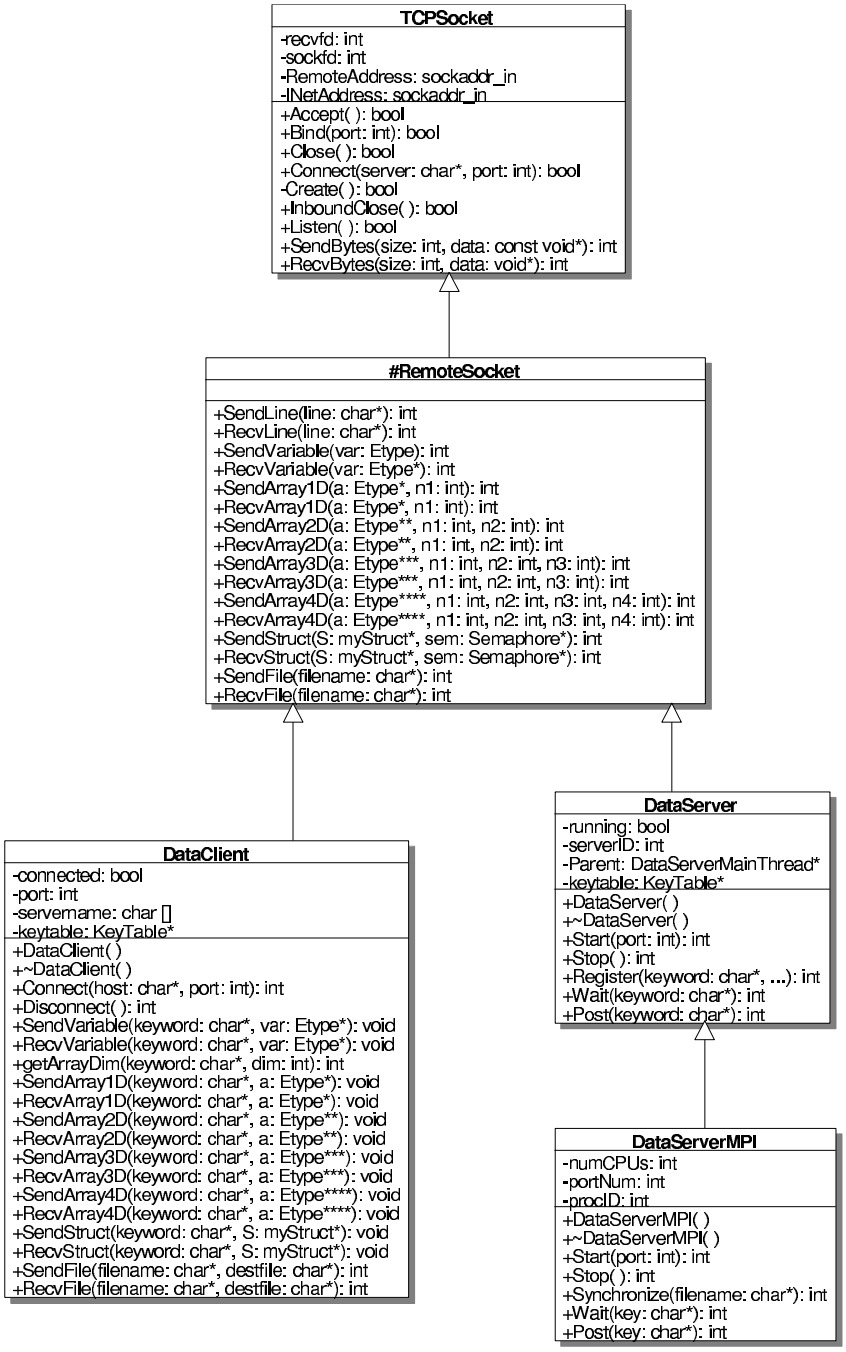


Figure 4.2. UML Diagram for POSSE classes

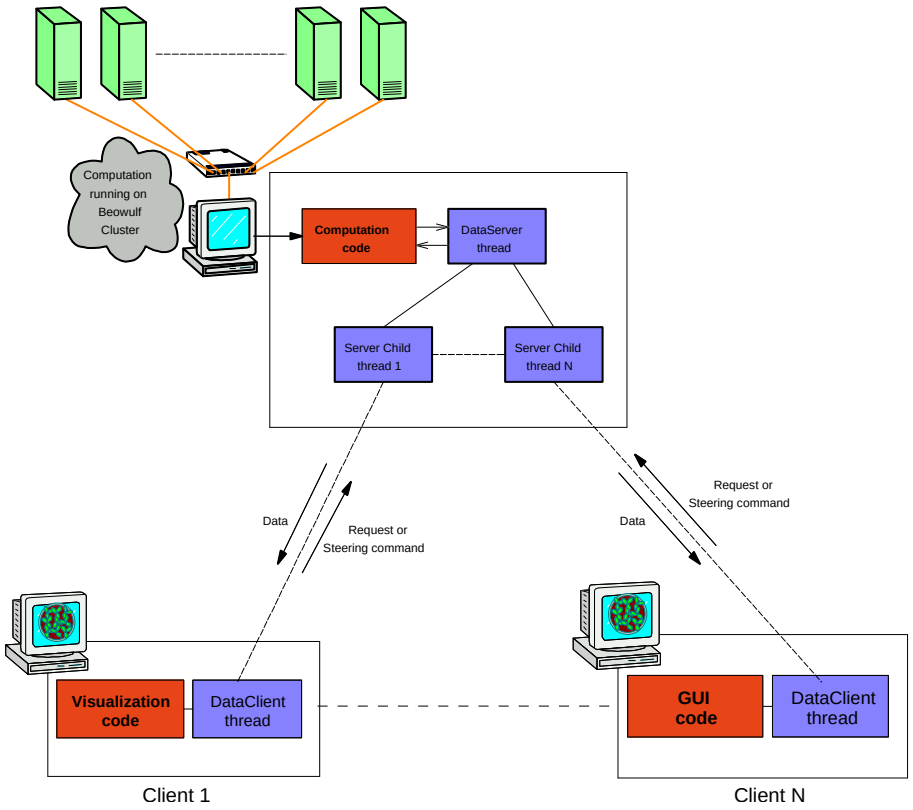


Figure 4.3. A schematic view of POSSE

systems is that it provides the same features of these systems, yet is extremely easy to use, making augmentation of any existing C/C++ simulation code possible in a matter of hours. POSSE makes extensive use of C++ classes, templates and polymorphism to keep the user Application Programming Interface (API) elegant and simple to use.

Figures 4.4 and 4.5 illustrate a simple, yet complete, POSSE client/server program in C++. As seen in Fig. 4.4, registered data on the steering server (which are marked *read-write*) are protected using binary semaphores when they are being updated in the computational code. Also, due to the automatic compile-time type-inferencing capability of C++ templates, the type of the data being registered need not be specified, thus making it easier and less error-prone for the programmer. User-defined data structures are handled by a simple user-supplied pack and unpack subroutine that call POSSE data-packing functions to tackle the byte-ordering and byte-alignment issues.

The programmer need not know anything about the internals of threads, sockets or networking in order to use POSSE effectively. POSSE also allows for a simulation running on any parallel or serial computer to be monitored and steered remotely from any machine on the network using a cross-platform Graphical User Interface (GUI) utility.

4.2 Real-Time Implementation Issues

Real-time systems consist of several networked machines, often with differing hardware and software communicating with each other via messages. For message communication to work correctly, the message formats have be defined unambiguously. In many systems this task is achieved simply by using arrays and/or structures to implement the message format. While this may be a simple approach, it has its own pitfalls. Some of the portability problems plaguing this are:

```
#include "dataserver.h"

const int MAX_ITER = 2000;
int dummyInt = 0, n1, n2;
double **dyn2D;

REGISTER_DATA_BLOCK()   // Register global data
{
  REGISTER_VARIABLE("testvar", "rw", dummyInt);
  REGISTER_DYNAMIC_2D_ARRAY("dyn2D", "ro", dyn2D, n1, n2);
}

int main(int argc, char *argv[])
{
  DataServer *server = new DataServer;

  if (server->Start(4096) != POSSE_SUCCESS) // Start Server thread
      {
       delete server;
       exit(-1);
      }
  n1 = 30; n2 = 40;
  ALLOC2D(&dyn2D, n1, n2);

  for (int iter = 0; iter < MAX_ITER; iter++) {
         server->Wait("dyn2D"); // Lock DataServer access for dyn2D
           /*
             Compute(dyn2D);    // Update dyn2D with new values
           */
         server->Post("dyn2D"); // Unlock DataServer access for dyn2D
   }
  FREE2D(&dyn2D, n1, n2);
  delete server;
}
```

Figure 4.4. A simple, complete POSSE server application written in C++

```
#include "dataclient.h"

int main(int argc, char *argv[])
{
  DataClient *client = new DataClient;
  double **dyn2D;

  // Connect to DataServer
  if (client->Connect("cocoa.ihpca.psu.edu", 4096) != POSSE_SUCCESS)
       {
        delete client;
        exit(-1);
       }
  client->SendVariable("testvar", 100); // Send new value for "testvar"
  int n1 = client->getArrayDim("dyn2D", 1);
  int n2 = client->getArrayDim("dyn2D", 2);
  ALLOC2D(&dyn2D, n1, n2);
  client->RecvArray2D("dyn2D", dyn2D);
  /*
    Use(dyn2D);  // Utilize dyn2D
  */
  FREE2D(&dyn2D, n1, n2);
  delete client;
}
```

Figure 4.5. A simple, complete POSSE client application written in C++

1. byte-ordering or endianness, and

2. byte-alignment issues.

Another problem that arises when periodically exchanging data that is continually evolving at either end is that of “data coherence.” Also, if the system is to be run on a parallel machine, we have to worry about “computational overhead and scalability.” The sections below describe each of these issues in detail and discusses ways to resolve them.

4.2.1 Endianness

When transporting data over a computer network from one machine to another, one has to ensure that the represention of this data is uniform across both the machines. However, this is not always the case due to differing endianness between the machines. Endianness is the attribute of a hardware system that indicates the order of the bytes representing basic data types (e.g., integers, floats and their variants). Endianness are of two types: *big-endian* and *little-endian*. A big-endian system represents the data types in a left to right order with the most significant byte (MSB) present on the left. Conversely, a little-endian system represents the data types in a left to right order with the least significant byte (LSB) present on the left. Examples of big-endian machines are Cray, HP, IBM, Mac, SGI and Sun, and those of little-endian machines are PC (x86), Alpha and VAX.

Consider the hexadecimal representation of the 4-byte (32-bit) integer 2030005539:

$$2030005539 \rightarrow 78FF6D23$$

In the big-endian representation, the byte-ordering is:

$$\text{Big-endian:}\quad 78FF6D23 \rightarrow \overbrace{78}^{\text{MSB}}\quad FF\quad 6D\quad \overbrace{23}^{\text{LSB}}$$

In the little-endian representation, the byte-ordering is reversed and is:

$$\text{Little-endian:}\quad 78FF6D23 \rightarrow \overbrace{23}^{\text{LSB}}\quad 6D\quad FF\quad \overbrace{78}^{\text{MSB}}$$

Hence, a memory dump for the code-segment below:

```
char  a = 1;
char  b = 2;
short c = 255;   // 0x00FF
int   d = 23245; // 0x01EBD405
```

will look like the following under the different endian systems:

Little-Endian:	01	02	FF	00	05	D4	EB	01
Big-Endian:	01	02	00	FF	01	EB	D4	05

Figure 4.6 illustrates a simple C++ function to determine the endianness of any given system. The function works by initializing a 2-byte data field (`short`) with a known hex value (0x4142) with differing values for each of the byte locations. A character pointer is then attached to the beginning of its memory location. If this pointer, which points to the first and left-most byte for the 2-byte data contains the hex value 0x41 then the system is determined as big-endian, and if it contains the hex value 0x42 then it is determined as little-endian. Figure 4.7 illustrates another general C++ function to reverse the byte-ordering of any data type by swapping the

bytes at opposite ends. This can be used to change the representation of any data type from little-endian to big-endian and vice versa.

As can be concluded from the above explanation, the endianness issue is only associated with binary data comprising of any multibyte data type. Since a `char` or an `unsigned char` are represented using 1 byte on all known systems, they are independent of byte-ordering. Hence, ASCII data comprising of characters are considered universally portable.

Endianness issues have to be resolved efficiently and unambiguously when dealing with machines with differing byte-order. For this, the computer network defines its own byte-order, namely the *network byte order*, which is big-endian. All data transported over any computer network is converted from the host byte-order to the network byte-order. On the receiving end, the data is converted from the network byte-order to the target host's byte-order. This can lead to conversion overhead for little-endian machines which can reduce the effective network bandwidth by as much as $10 - 15\%$. If majority of the machines participating in the communication are big-endian, then this is not of concern. However, if majority of the machines participating in the communication are little-endian, then the overhead can become significant as there will be a conversion required from big-endian to little-endian (or vice-versa) for every machine on the network that is little-endian. To avoid this overhead, POSSE can use either little-endian or big-endian as the network byte-order, thus reducing the overall conversion overhead if knowledge about the systems communicating are known a priori. For example, using little-endian as the network byte-order among a network of Intel PCs (little-endian) reduces overhead greatly as no byte re-ordering of data needs to be performed.

```
bool isBigEndian()
{
  short s = 0x4142;      // 2-byte data-type
  char *leftmostbyte = (char *) &s;

  if (leftmostbyte[0] == 0x41 && leftmostbyte[1] == 0x42)
      return true;      // Big-endian
  if (leftmostbyte[0] == 0x42 && leftmostbyte[1] == 0x41)
      return false;     // Little-endian
// Should never reach here!
}
```

Figure 4.6. A simple C++ function to determine the endianness of any system

```
template <class Etype>
Etype swapbytes(Etype in)
{
  Etype out;
  char *p_in = (char *) &in;
  char *p_out = (char *) &out;
  static int size = sizeof(Etype);

  for (int i = 0; i < size; i++)
       p_out[i] = p_in[size-1-i];

  return out;
}
```

Figure 4.7. A simple C++ function to reverse byte-ordering of any data type

4.2.2 Byte-alignment

The byte-alignment problem crops up due to the fact that different processors/compilers might define the same structure differently, thus causing incompatibility in the interface definition. Here, alignment is defined as the placing of data and code in the computer memory in addresses that are more efficient for the hardware to access (at the price of wasting some memory).

32-bit microprocessors typically organize memory in 32-bit (4-byte) increments. Memory is accessed by performing 32-bit bus cycles. Since memory is organized in 4-byte increments, these 32-bit bus cycles can only be performed at addresses that are divisible by 4. Table 4.1 shows a snapshot of memory with two 4-byte data *A* and *B*, where *A* comprising of bytes A_1, A_2, A_3 and A_4 is aligned with the 4-byte boundary, and *B* comprising of bytes B_1, B_2, B_3 and B_4 is not.

	Byte 1	**Byte 2**	**Byte 3**	**Byte 4**
0x00000000				
0x00000004	A_1	A_2	A_3	A_4
0x00000008				
0x0000000B		B_1	B_2	B_3
0x00000010	B_4			

Table 4.1. Byte-alignment problem

The reasons for not permitting misaligned data reads and writes are clearly seen from the table. For example, the aligned data *A* can be read by the processor in a single bus cycle starting at address 0x00000004. If the same processor now attempts to access the data *B* at address 0x0000000C, it cannot do it in a single bus cycle. The processor will have to issue two different reads at address 0x0000000B and 0x00000010 to read the complete 4-byte data. Thus it takes twice the time to read a misaligned 32-bit data. Often times, misaligned addresses may generate bus errors and abort the program. Older computers (e.g., VAX systems), can

handle misaligned memory accesses more gracefully, but by degrading the performance. This approach was mainly due to the fact that memory was once very costly, and computer designers could not afford to waste even a small amount on alignment.

Code alignment is usually controlled by the compiler. Let us consider the following user-defined data-structure in C:

```
struct myStruct
{
  short v1;   // 2-byte
  char  v2;   // 1-byte
  int   v3;   // 4-byte
  char  v4;   // 1-byte
  short v5;   // 2-byte
};
```

For a 32-bit processor, the compiler will likely use the following padded definition for the data-structure:

```
struct myStruct
{
  short     v1;  // 2-byte
  char      v2;  // 1-byte
  char pad1[1];  // Pad to start the following "int" 4-byte boundary
  int       v3;  // 4-byte
  char      v4;  // 1-byte
  char pad2[3];  // Pad to start the following "short" at a 4-byte boundary
  short     v5;  // 2-byte
  char pad3[2];  // Pad to complete aligning at a 4-byte boundary
};
```

In the above example, the compiler has added pad bytes to enforce byte alignment rules of the target processor. If the above data-structure was used in a different compiler/microprocessor combination, the pads inserted by that compiler might be different. Thus, two applications using the same data-structure definition may have different sizes and may be incompatible with each other.

POSSE tackles the byte-alignment issue (and the byte-ordering issue) by having the user provide a custom packing and unpacking function for every user-defined data-structure that needs to be communicated over the network. These function individually pack and unpack each field in the structure using macros provided by POSSE. Figure 4.8 illustrates this with a simple user-defined data-structure `Trajectory` along with the functions `packStruct()` and `unpackStruct()` to pack and unpack the structure during the communication. The POSSE macros also convert the host byte-ordering to the network byte-ordering before packing the data, and back from the network byte-ordering to the host byte-ordering after unpacking the data. Thus the resulting contiguous data-segment (`unsigned char *data`) containing the data-structure is unambiguous and portable. An advantage of this approach is that the entire structure need not be packed; only parts of the structure of interest to the client can be packed and unpacked, thus reducing time to convert and transport the data.

4.2.3 Data coherence

Whenever there is an exchange of constantly changing data between networked machines, one has to worry about the problem of maintaining data coherence. There are several situations that can lead to data inconsistencies.

1. A request for data arrives while it is being modified. If the request is answered immediately, the data sent may be incoherent, especially if the data is an array or a structure which may require several milliseconds to update. A part of the data sent will be in the new state after modification while the rest will be in the old state before modification.

2. Multiple clients may want to modify the same portion of a *read-write* data on a server simultaneously. If this is not controlled by some means, parts of data modified by one

```
typedef struct {
   int i;
   double d;
   float a[200][400];
} Trajectory;

int packStruct(Trajectory *T, unsigned char **dataptr, int &totsize)
{
  totsize = sizeof(T->i) + sizeof(T->d) + sizeof(T->a);
  unsigned char *data = new unsigned char[totsize];
  *dataptr = data;
  int ptr = 0;

  PACK_VARIABLE(T->i, data, &ptr);
  PACK_VARIABLE(T->d, data, &ptr);
  PACK_2D_ARRAY(T->a, data, &ptr);

  return ptr;
}

int unpackStruct(Trajectory *T, unsigned char *data, int size)
{
  int ptr = 0;

  UNPACK_VARIABLE(&(T->i), data, &ptr);
  UNPACK_VARIABLE(&(T->d), data, &ptr);
  UNPACK_2D_ARRAY(T->a, data, &ptr);

  return ptr;
}
```

Figure 4.8. Packing and unpacking of a custom C++ structure

client can be overwritten by the other client leading to inconsistent data on the server-side. If another client requests this data immediately thereafter (or worse, during the write operation), it will receive incoherent data.

3. A server is utilizing a particular *read-write* data for a computation in a loop. It expects this data to remain unchanged during any given computation loop (e.g., time step iteration) to yield correct results. If during any particular iteration, a client tries to modify the data and succeeds, then the rest of the computation for that iteration will use the modified data and may produce inconsistent results.

These data coherence problems are dealt with in POSSE through the use of binary semaphores. A semaphore is a classic method for restricting access to shared resources in a multi-processing or multi-threading environment [80]. Since each client request in POSSE is handled by a separate thread on the server-side (see Fig. 4.3), chances of conflict between threads is likely to occur. On the server-side, binary semaphores are used to lock and unlock *read-write* data (also known as *critical data*) when they are being modified. This has to be done by the user using the lock associated with that data handle. This is illustrated in Fig. 4.4 with the `Wait("dyn2D")` and `Post("dyn2D")` commands used in the C++ server code. At any given time, at most one thread can acquire the lock by issuing the `Wait()` command. While the lock is held by the user code, no DataServer thread can acquire the lock which is required in order to allow modification to the critical data. Once the user releases the lock by issuing the `Post()` command, any DataServer thread may acquire the lock and proceed to modify the data. Modification for server-side data is first received into a buffer and then copied to the target location after a lock is acquired for the target data. This is done to minimize the holding period of the lock by excluding the communication time required to receive the data over the network which may be significant. Long holding periods for a lock can lead to poor response times for competing

threads waiting for the same lock.

4.3 Overhead and Scalability

One of the most important issues with any steering software that is used in conjunction with parallel programs is that of scalability. It is essential that the processor overhead due to the steering thread be small relative to the primary computational thread. When this is not the case, the primary computational thread will compete with the steering thread for CPU cycles slowing down the entire simulation. Also, it is expected that the computational overhead be uniform across all the compute nodes. If not, due to the nature of SPMD parallelism, the node having the highest overhead will dominate slowing down all the other nodes along with it.

In POSSE, the blocking socket `accept()` call is used to wait for a client connection and a blocking socket `recv()` call is used to wait for requests from an established client connection. Since these calls are blocking, they work directly with the network (TCP/IP) layer and hence do not use polling[5] (which can be very inefficient and can lead to high processor overhead). Thus, when no `DataClient` requests are being processed by the `DataServer`, the overhead is limited to that of the blocking `accept()` and/or `recv()` calls, which is extremely low ($< 0.1\%$ of total processor time). When there are requests being handled, the overhead varies with the number of requests and the type of requests (*read* vs. *write*). This overhead is primarily due to the spawning of new server threads to answer requests from each client, and due to the constant locking and unlocking of critical data (if any). A detailed performance analysis of this overhead is presented in Section 4.4.

[5] Periodic checking in an infinite loop

4.4 Experimental Results

POSSE has been extensively tested using various platforms for stability and performance. In this section, tests demonstrating both the single and multiple client performance of POSSE are presented.

4.4.1 Single Client Performance

Fig. 4.9 shows a plot of the effective network bandwidth achieved by varying the size of a dynamic 1-D array requested by a steering client. These tests were carried between machines connected via a Fast Ethernet connection having a peak-theoretical network bandwidth of 100 Mbps. The communication time used to calculate the effective bandwidth includes the overheads for byte-ordering, data packing and other delays introduced by the querying of the registered data. The average latency for query of any registered data by the client has been found to be 38 ms. As can be seen, there is a noticeable decrease in the bandwidth (about 10 Mbps) when communicating between machines with different byte-ordering (i.e., little endian vs. big endian) as opposed to machines with the same byte-ordering. This difference reflects the overhead involved in duplicating the requested data and converting it into the byte-order of the client machine for communication. In the same byte-order case, as the size of the requested data increases to about 5 MB, the effective bandwidth approaches 80 Mbps, which is 80% of the peak-theoretical bandwidth. The source code for the programs used to carry out the test can be found in Appendix C.1.

4.4.2 Multiple Client Performance

Fig. 4.10 shows a plot of the effective bandwidth achieved by varying the number of clients simultaneously requesting data. In this test, both the clients and the server were machines with the same byte-ordering. The server had a registered 4-D array with $200,000$ *double* elements (1.6 MB of data) but did not perform any computation in the main thread. There was an idle loop included in the `main()` function. All the clients were then run simultaneously from two remote machines on the same network and were programmed to request the 1.6 MB 4-D array from the server. The effective bandwidth in this case is obtained by dividing the total amount of data served by the server with the total wall-clock time required to serve all the requests. It can be seen that the network performance of POSSE is very good (84 Mbps) even when dealing with over 500 client requests simultaneously. The source code for the programs used to carry out the test can be found in Appendix C.2.

For the wake-vortex simulation system, the amount of data communicated to the client after every update is 420 bytes for every aircraft and 56 bytes for every vortex element. For 10 aircraft each having $2,000$ elements tracked, this amounts to 1.12 MB of data. From Fig. 4.9, we can see that this corresponds to a data-rate of approximately 62 Mbps, or 145 ms of communication time. Thus, we can get updated data at a rate of almost 7 fps from the server. The wake-vortex simulation runs with a Δt of 0.2 seconds which can be maintained for up to $2,000$ vortex elements per aircraft on COCOA-2. The parallel code has very good scalability for up to 15 processors (tracking 15 aircraft), after which it linearly deteriorates due to the overhead borne by the master for distributing and collecting data from the slave nodes. At this point, the simulation has only been qualitatively checked and seems to be consistent with the theory. The simplification of using k neighbors for induced-velocity computation works very well with an error of less than 1% when compared to the original $O(N^2)$ case. Since the weather conditions

play a substantial role in the determination of the vortex decay, a more sophisticated weather model like the one used in AVOSS will definitely improve the accuracy of the simulations.

4.4.3 Symmetric Multi-Processor (SMP) vs. Uni-Processor Server

Figure 4.11 shows the effect of Symmetric Multi-Processor (SMP) vs. Uni-Processor (UP) for the machine running the server-side application. This was obtained by running a variation of the server-side code used for the multiple client performance test in the section above. The server was run on an dual PIII-400 Mhz SMP machine running Linux. For the UP tests, the Linux kernel without SMP support was used to boot the machine resulting in the detection of only one processor. In the server for this test, an actual computation loop was included instead of an idle loop in the `main()` function. All the clients were then invoked simultaneously. Once all the client requests were answered by the server, the number of computation loops performed per second were noted. As can be seen clearly, the 2-processor SMP machine performs approximately 48% more computations per unit time as compared to the UP machine while answering the same number of client requests. This is due to the multi-threaded nature of the server application which scales well on a multi-processor architecture. A single processor machine cannot cope up with several simultaneous client requests without slowing down the computation. The source code for the programs used to carry out the test can be found in Appendix C.3.

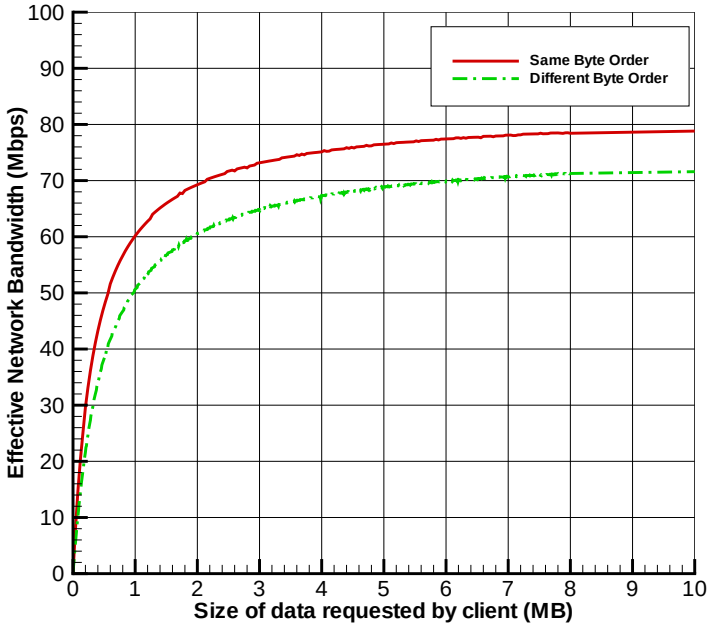


Figure 4.9. Effective Network Bandwidth vs. Size of data requested by client

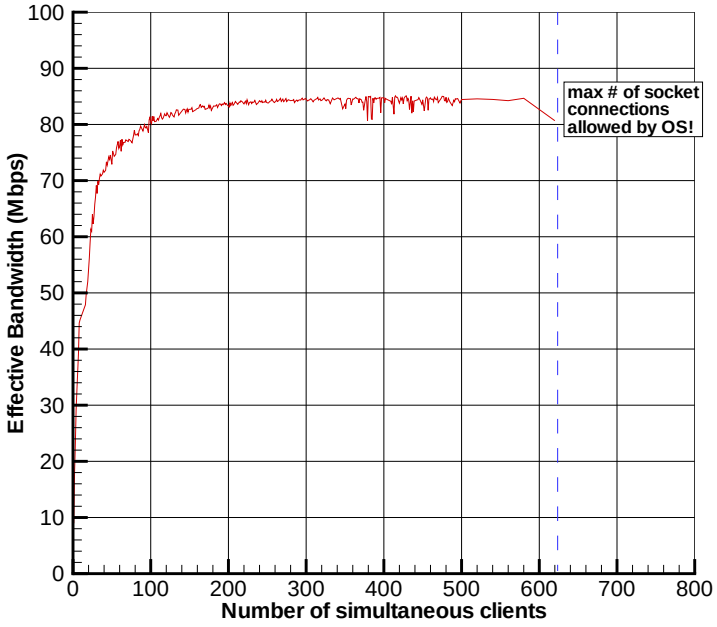


Figure 4.10. Effective Network Bandwidth vs. Number of simultaneous clients

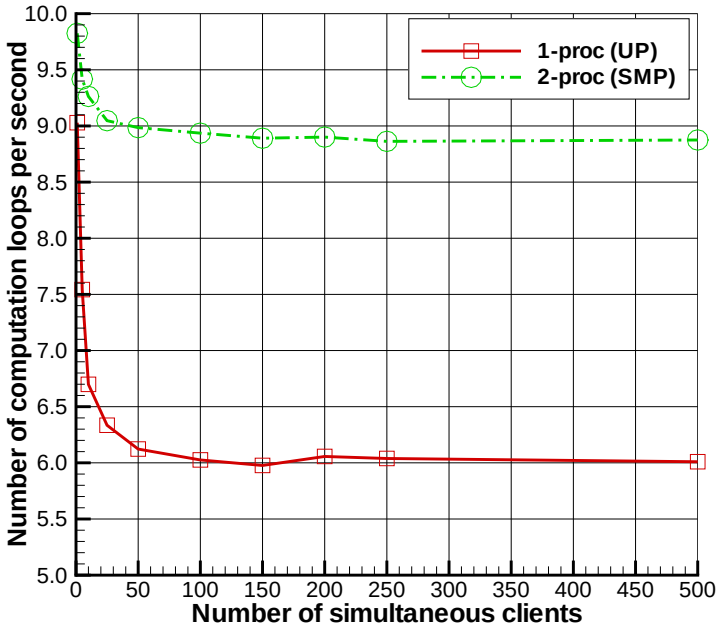


Figure 4.11. Effect of SMP vs UP machine on the server

Chapter 5

Wake-Vortex Simulations

5.1 Wake-Vortex Theory

For the wake-vortex simulations described in this thesis, aerodynamic potential flow theory is used to predict the strength of the wake-vortex elements [82]. The circulation produced by the wing is assumed to be contained in two vortices of opposite signs trailing from the tips. Figure 5.1 shows the schematic of a wake-vortex pair. The wake is assumed to consist of a pair of vortices which are parallel and the longitudinal axis of the tracked airplane is assumed to be parallel to the vortex pair. The centers of the vortices are on a horizontal line separated by a distance of $b_s = \frac{\pi}{4} b_g$, a result of assuming an elliptic distribution, where b_s is the separation of the vortices in the wake-vortex pair, and b_g is the span of the airplane wing generating the wake vortex [83]. The magnitude of the circulation of each vortex element is approximately

$$|\Gamma| = \frac{4}{\pi} \frac{L_g}{\rho V_g b_g}, \tag{5.1}$$

where L_g and V_g are the lift and the velocity of the aircraft, respectively. References [84, 85] deal with more details on the numerical simulation of these aircraft vortices. The velocity $\vec{V}_{1,2}$ induced by a 3D, straight vortex segment $(1,2)$ of strength Γ at point $P(x,y,z)$ outside the vortex core as shown in Fig. 5.2 is:

$$\vec{V}_{P_{(1,2)}} = \frac{\Gamma}{4\pi} \frac{\vec{r_1} \times \vec{r_2}}{|\vec{r_1} \times \vec{r_2}|^2} (\vec{r_1} - \vec{r_2}) \cdot \left(\frac{\vec{r_1}}{|\vec{r_1}|} - \frac{\vec{r_2}}{|\vec{r_2}|} \right). \tag{5.2}$$

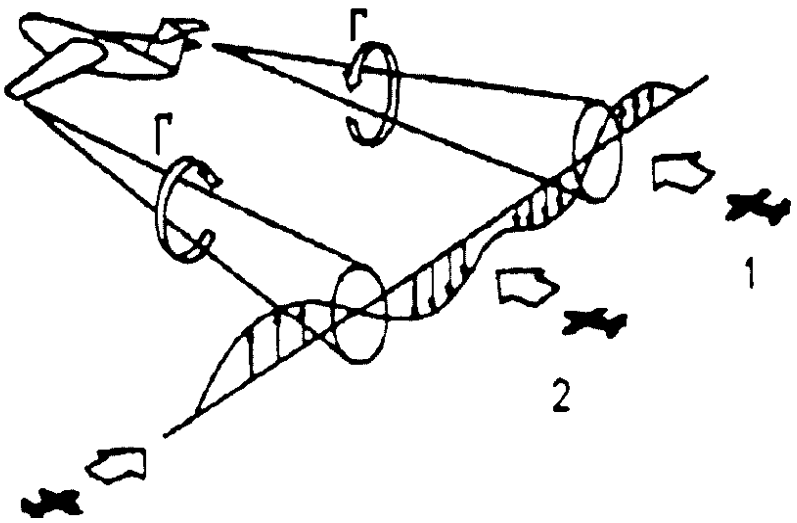


Figure 5.1. Schematic of a wake-vortex pair

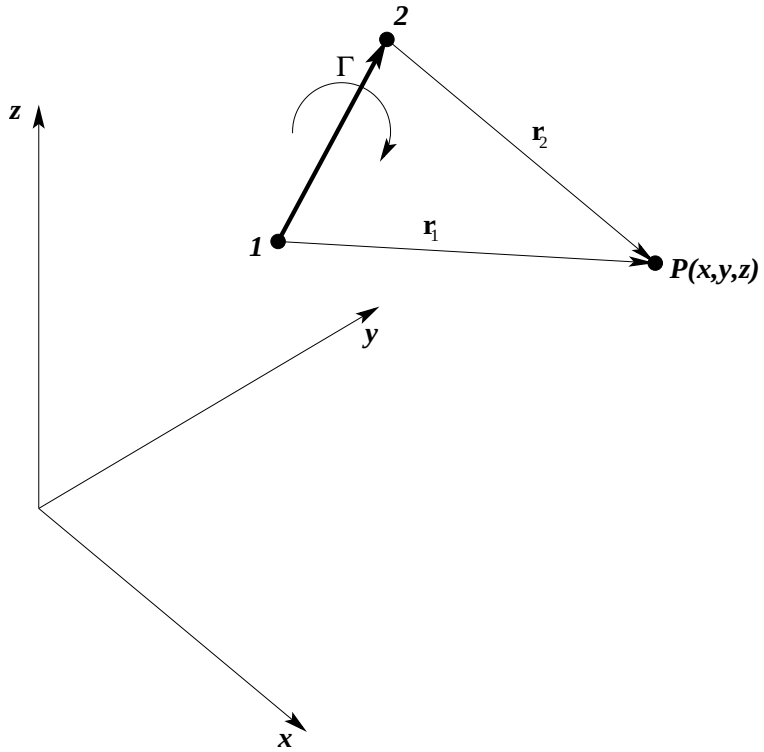


Figure 5.2. Nomenclature used for the velocity induced by a 3D, straight vortex segment (Courtesy Katz and Plotkin [81])

If the point P is within the vortex core of radius ε, the induced velocity is

$$\vec{V}_{P_{(1,2)}} = \frac{\Gamma}{2\pi\varepsilon^2}\frac{\vec{r_1}\times\vec{r_2}}{|\vec{r_1}-\vec{r_2}|}. \tag{5.3}$$

ε is typically 10% of the chord length of the airplane wing.

After the strength of these vortices are computed, the effect due to the prevailing weather data is applied to the prediction. The vortex filaments propagate with the freestream wind conditions and the induced velocity due to the other vortex elements. A standard turbulent atmospheric boundary layer profile is used to compute the wind velocities close to the ground. The typical height of the atmospheric boundary layer is taken to be 400 meters above the ground. Now, in the wake-vortex pair, the velocity induced by the rotating flow from one vortex pushes its neighbor down. And the velocity induced by the rotation from the other vortex, in turn, pushes its neighbor down. Consequently, the vortices descend toward the ground. Near the ground, a boundary condition has to be implemented to ensure that the downward velocity of the vortices goes to zero at the ground plane. This boundary condition is satisfied by imagining a pair of virtual vortices under the ground with equal but opposite circulation to the real vortices. Typically, this boundary condition is enforced only when the wake-vortices are within 300 meters of the ground. These boundary conditions ensure that both the wind velocities and the induced velocities smoothly go to zero as the vortices approach the ground.

The decay of the vortex strength is based on a simplified version of the model suggested by Greene [86]:

$$\Gamma_{t+\Delta t} = \Gamma_t(1-\frac{\Delta t V_t}{8b_g}), \tag{5.4}$$

where V_t is the vortex velocity at time t and is given by

$$V_t = \frac{\Gamma_t}{2\pi b_g}.$$

Here Γ_t represents the strength of the vortex element at time t and $\Gamma_{t+\Delta t}$ represents the strength of the vortex at time $t + \Delta t$ (next time-step).

5.2 Simulation Complexity

The wake-vortex prediction for an entire fleet of aircraft taking-off and landing at a busy airport is an extremely computationally intensive problem. Not only does the wake-vortex simulation have to run fast enough to mimic the actual physical phenomena in real-time, but also the time required to communicate the computed data to the remote visualization client has to be kept to a minimum. As such, a POSSE-based parallel solution for the same is required to maintain a real-time response of the simulation. For example, a typical metropolitan airport in the USA is extremely busy with several take-offs and landings occurring every few minutes.

Dallas/Fort Worth, USA's third busiest airport, has seven runways that handle nearly $2,300$ take-offs and landings every day. For the wake-vortex code to track the vortices shed by an aircraft for 5 miles after take-off, assuming that a vortex core is stored every 5 meters, $5 \times 1,600/5 \times 2 = 3,200$ vortex filaments have to be tracked. For $2,300$ take-offs and landings every day, it implies that $3,200 \times 2,300/24 = 306,667$ vortex filaments have to be tracked every hour. Since the vortices may take as long as 15 minutes to decay significantly, vortices due to typically half the take-offs and landings every hour need to be tracked at any given time. This amounts to roughly $153,333$ vortex filaments. While this may not seem to be a very large number on its own, the problem gets complicated by the presence of an $O(N^2)$ calculation for

the induced velocity of every vortex element on every other vortex element, where N represents the number of vortex elements. Even if the induced velocity effect due to vortices from the other aircraft are ignored, this still amounts to as much as $3,200 \times 3,200 = 10.24$ million calculations for each airplane at every timestep.

For $2,300$ planes/day, this comes out to $10.24 \times 2,300/24/2 = 490.7$ million calculations per timestep for the induced velocity, a very large number indeed for a conventional uniprocessor system. And with each timestep being, say 0.2 seconds, this amounts to 2.45 billion calculations per second. This number can be reduced by as much as a factor of 100 by making a simplifying assumption for the induced velocity calculations. It is assumed that any vortex element is only affected by a fixed number of neighboring elements, say k, on each stream of the wake-vortex pair rather than all the N elements in each stream. This reduces the simulation complexity by a factor of $N/4k$. Here, k is chosen as the smallest acceptable value such that the error introduced by the approximation on a sample set of runs is a maximum of 1% when compared to the original $O(N^2)$ solution. The typical value of k used in these simulations is 10, which for an N of $4,000$, is a factor of 100 lower. However, even a 100-fold reduction still amounts to a large computation considering that each induced velocity calculation consists of $200-300$ floating point operations. This takes the net computational requirement to approximately $5-8$ Gigaflops, necessitating the need of a parallel computer. To address the problem of maintaining real-time response, the wake-vortex prediction code, based on the potential flow theory described above, is written in C++ with MPI for parallelization.

5.3 Pseudocode

In Fig. 5.3, we present the pseudocode for the wake-vortex simulation. Every continuous wake-vortex is represented by several discrete "vortex elements" within the computation. Each

vortex element has two main properties associated with it, strength, position and orientation. The initial strength (Γ) is calculated based on the potential flow theory (Eq. 5.1) and the initial position is based on the the position of the aircraft. The vortex strength then decays as a function of time (Eq. 5.4) and the prevailing weather conditions, and the position and orientation changes because of the velocity induced by neighboring vortex elements (Eq. 5.2) and the prevailing wind velocity. Fig. 5.4 depicts a diagram of the complete client/server simulation system. The simulation system consists of the *Wake-Vortex Server*, *Airport Data Server* and the *Sound Server*. The Wake-Vortex Server is the simulation code enhanced using POSSE. This code can be found in Appendix A.

The Airport Data Server is another POSSE server that serves the positions of the aircraft in the vicinity of the airport as well as the prevailing weather conditions. The Sound Server is an optional component in the the system for simulating the noise-level at the airport. The wake-vortex code has been parallelized to track vortex elements from each aircraft on a different processor in such a way that (if it is possible) nearly real-time solution to this problem is obtained with tolerable lag no more than the time-step Δt in the simulation.

In the parallel simulation, one processor (usually the first) acts as the master doing a round-robin scheduling of any new aircraft to be tracked among the available processors (including itself). The master, therefore, does the additional work of distributing and collecting vortex data from the slave nodes. Hence, it is ensured that the master is always running on a Symmetric Multi-Processor (SMP) node with at least two processors so that the POSSE server thread runs on an idle processor and does not slow down the master node because of the constant monitoring of the vortex data by the visualization client.

$V \leftarrow \emptyset$
$t \leftarrow 0$
Foreach aircraft A on a different processor
  While (A in specified range from airport) **do**
    read updated aircraft position from **airport data server**
    read updated weather condition from **airport data server**
    $V \leftarrow V + \{$newly created vortex element from wing using potential theory$\}$
    Foreach vortex element ($v_i \in V$)
      $v_i.inducedvel \leftarrow 0$
      Foreach vortex element ($(v_j \in V) \neq v_i$ & $|j-i| \leq k$)
        $v_i.inducedvel \leftarrow v_i.inducedvel +$ InducedVelocity(v_i, v_j)
      Endforeach
      $v_i.position \leftarrow v_i.position + \Delta t \times v_i.inducedvel$
      $v_i.position \leftarrow v_i.position + \Delta t \times$ (prevailing wind velocity)
      $v_i.strength \leftarrow v_i.strength -$ DecayFunction$(\Delta t,$ Weather Conditions$)$
      If ($v_i.strength < threshold$) **then**
        $V \leftarrow V - v_i$
      Endif
    Endforeach
    $t \leftarrow t + \Delta t$
  Endwhile
Endforeach

Figure 5.3. Algorithm for Wake-Vortex prediction

5.4 Implementation

To tackle the issues arising when dealing with such a complex simulation, the wake-vortex application is run as four separate components: *Airport Data Server*, *Sound Server*, *Wake-Vortex Server* and *Visualization Client*. The airport data server is written using POSSE and provides the current weather data and aircraft trajectory to any wake-vortex client that requests it. It can either read the data from a file or generate random data. For the sound server, an already existing application, the *Bergen sound server* [87] is used. Like POSSE, Bergen too is written in C++, is based on UNIX sockets and is portable. It comes with a C++ library with functions to communicate with the server. The third component, the wake-vortex server

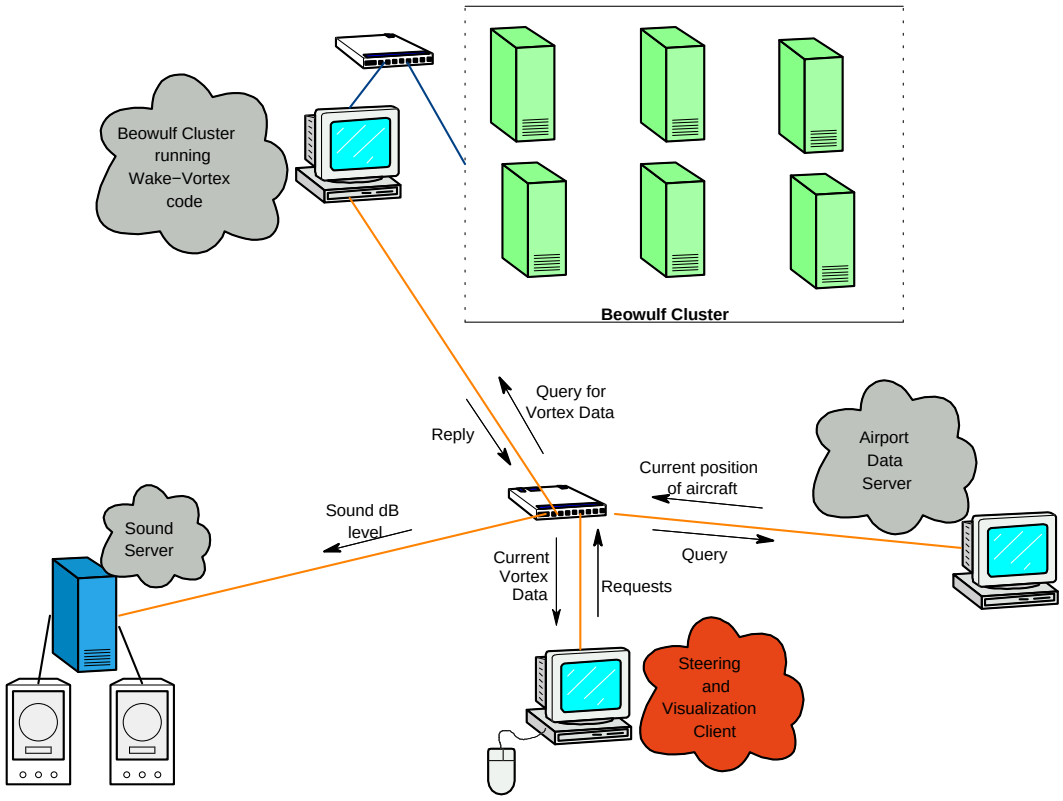


Figure 5.4. Wake-Vortex Simulation System

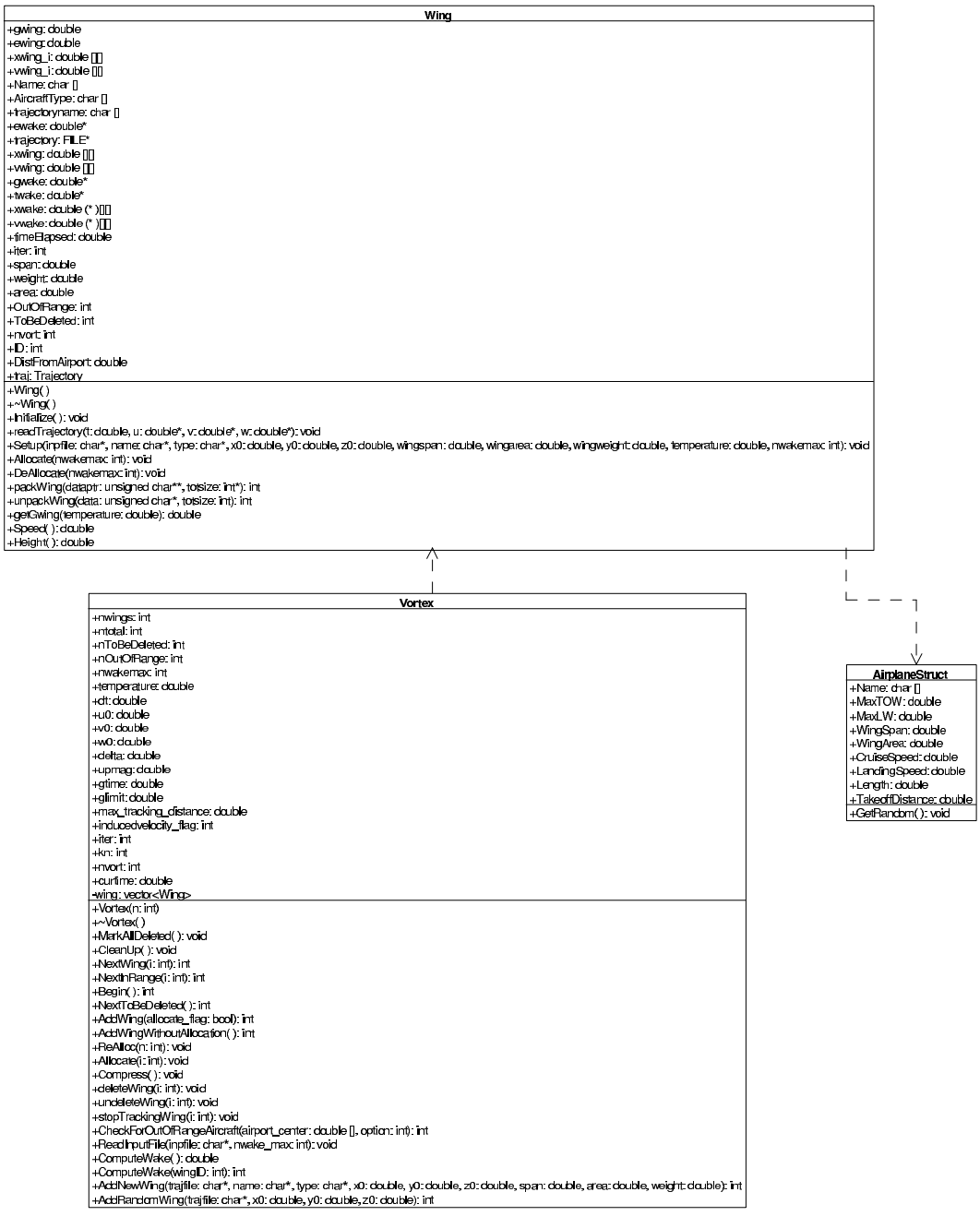


Figure 5.5. UML Diagram for Wake-vortex classes

is broken into three separate parts:

1. *Numerical*: The numerical portion of the wake-vortex simulation code is written as a separate library which is independent of POSSE calls. A UML diagram for this library can be seen in Fig. 5.5. It consists of two main classes: `Wing` and `Vortex`. The `Wing` class manages all the simulation data for any single aircraft which includes their physical attributes, their trajectory information and all the wake-vortex simulation data. The `Vortex` class consists of a collection of `Wing` objects (using the C++ STL[1] vector) and contains methods for performing all the wake-vortex computations, as well as for the addition and deletion of aircraft from the structure as and when they appear or disappear around an airport. In the absence of the optional airport data-server, an input file specifies the weather conditions, the number of aircraft in the simulation, their physical attributes, their location and their trajectories. Appendix A.1 contains a sample input and trajectory file.

2. *Parallel*: An MPI application is written using the wake-vortex library to parallelize the simulation based on the pseudocode described in Fig. 5.3.

3. *Steering*: The POSSE server component (DataServerMPI) is then added and relevant simulation data required by the visualization client is registered. Any locking of registered data is done as required to avoid data coherence problems. A POSSE client component (DataClient) is also added to periodically retrieve the trajectory and weather data from the *airport data server*.

The visualization client is broken into three separate parts:

[1] Standard Template Library

1. *Steering*: The POSSE client component (DataClient) is used to periodically receive the updated `Vortex` structure from the wake-vortex server. The `Vortex` structure contains the updated trajectory and wake-vortex data for all the aircraft being tracked. This component is run as a separate thread to maintain reasonable response times for the graphics component of the application, as the receipt of wake-vortex data is dependent on the amount of data to be communicated and the network speed, and could thus vary from a few milliseconds to several seconds.

2. *Graphics*: This component deals solely with the rendering of the display and the user-interaction. The graphics calls are written using OpenGL and user-interaction is managed by GLUT when running on a commodity (non-VR) hardware and by CAVELib when running on the RAVE. The CAVELib support adds the stereoscopic mode to the application and also the ability to navigate using the wand. The data for the display is obtained from the `Vortex` structure which is interpreted using the functions provided by the wake-vortex library. The vortex elements are connected using a 4th-order B-Spline curve [88] to give a smooth connectivity while maintaining the local control of the individual vortex elements. The aircraft models are stored as Wavefront [89] .OBJ files and are loaded at run-time using a public domain OBJ file loader written in C. The land, sea and the sky are drawn using OpenGL mipmaps.

   The airport is modeled as a separate class with its own file format. An example airport input file for San Francisco airport is provided in Appendix A.2. The airport class consists of functions to simulate some intelligence for automatic landing and take-off of aircraft by queueing requests from aircraft in the vicinity of the airport and providing them with pre-computed take-off and landing paths in First-In First-Out (FIFO) order.

The user-interface consists of the arrow-keys and/or the mouse movements for navigation of the airport. The mouse mode can be used to fly around the airport at varying speeds. The viewer can also optionally attach oneself to the aircraft frame of reference by pressing a hot-key. A compass on the top-right portion of the screen depicts the direction of the viewer. All important information is displayed on the screen as text and can be toggled using a hot-key. The top-left corner of the application window displays the name of the wake-vortex server along with the number of processors it is running on. It also provides information on the prevailing wind velocity, the location of the viewer, the number of aircraft being tracked and their distance from the center of the airport and the jet noise due to each of them. It also gives the type and location of each aircraft and the strength of the vortex-element being formed at its wing-tips. The bottom-left portion of the screen displays the information on the performance of the application. It displays the frame-rate for the graphics, the current time-step of the simulation, the elapsed simulation time, the number of iterations performed, the computational time required per iteration, and the effective network bandwidth for the monitoring of wake-vortex data.

3. *Sound*: This component calculates the sound pressure level due to the jet-noise from the aircraft engines at the viewer location. This is calculated by a simple empirical relationship which is inexpensive to compute. Figure 5.6 depicts the distribution around a commercial jet that is used to calculate the noise level based on the angle from the jet nozzle [90]. Figure 5.7 depicts the behavior of noise amplitude as the distance from the jet increases. This is based on the following relation:

$$dB(\text{distance}) = 120 - 20\log_{10}\left(\frac{\text{distance in feet}}{100}\right).$$

According to the above relation, the noise level is 120 dB[2] at a distance of 100 feet from the jet and decreases by 6 dB ($\approx 20 \log_{10} 2$) every time the distance doubles. The final noise level in decibels is then sent to the Bergen sound server using Bergen library calls which then outputs the corresponding audio via the speakers. Figure 5.8 lists the client code in C++ used to initialize and communicate with the Bergen sound server.

For the real-time simulation, the parallel wake-vortex code runs on COCOA-2 and the visualization code runs on the RAVE. A screenshot of the program depicting the wake-vortex simulation for a single aircraft is shown in Fig. 5.9. Another screenshot (Fig. 5.10) shows several aircraft flying above a model of the San Francisco International airport (SFO). Another screenshot (Fig. 5.11) shows aircraft flying over a model of the New York–John F. Kennedy International airport (JFK). The colors of the wake-vortices represent the relative strength of the vortices with respect to a base value, with red being maximum and blue being minimum.

[2] decibel: a logarithmic unit of sound intensity

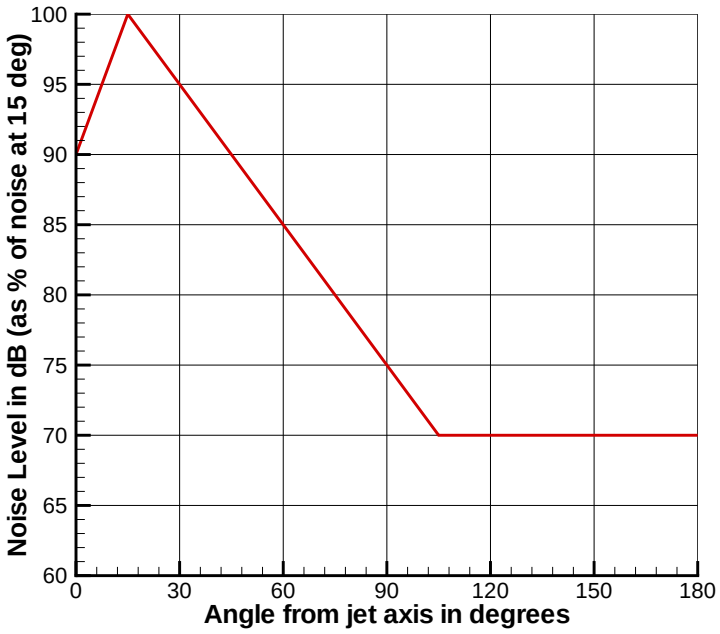


Figure 5.6. Sound dB level as a function of angle from the jet axis [90]

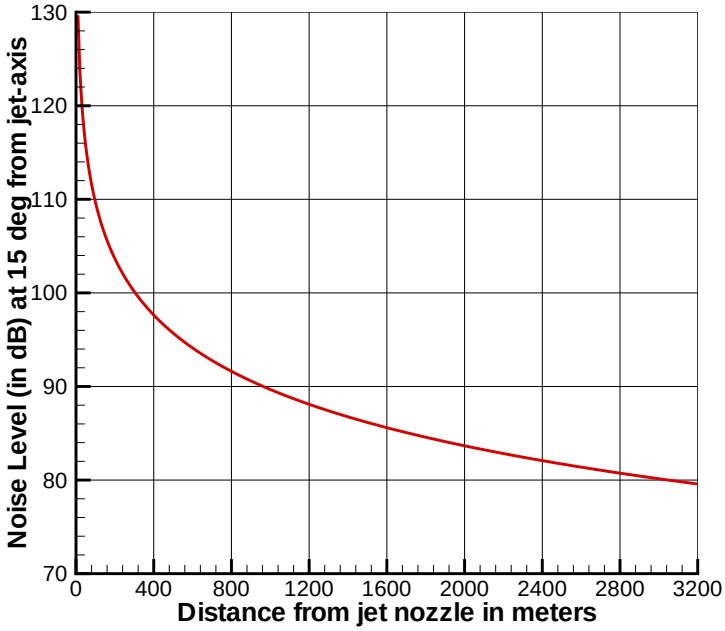


Figure 5.7. Sound dB level as a function of distance from the jet nozzle

```
//***************************************************************************
#include "bergen.h"
#include "vortex.h"
//***************************************************************************
bergenSample *sound;  // Global variable "sound"
//***************************************************************************
// Initialize the connection to the Bergen Sound Server
void InitSound(char *SoundServer)
{
  char s[100];
  sprintf(s, "BERGEN_SERVER=%s", SoundServer);
  putenv(s);  // Set the name of the sound server in the environment

  // Connect to the Bergen sound server specified by "BERGEN_SERVER" above.
  bergenServer *server = new bergenServer;

  // Play the "jet.wav" file in an infinite loop.
  sound = new bergenSample("jet.wav", server);
  sound->setLoop(1);
  sound->play();
}
//***************************************************************************
// Compute and send the current jet-noise level to the Bergen server
void PlaySound(Vortex * V)
{
  double SPL = 0.0;            // Sound Pressure Level

  // Loop over all aircraft present in the structure
  for(int i = V->Begin(); i < V->nwings; i = V->NextWing(i))
    {
      // Get angle and distance of observer from the "i"th aircraft
      AngleAndDistFromPlane(V, i, &angle, &dist);
      SPL += dB2Pressure(SoundDB_distance(dist) * SoundDB_angle(angle));
    }

  // Send the modified amplitude to the Bergen sound server
  sound->setAmplitude(Pressure2dB(SPL));
}
//***************************************************************************
```

Figure 5.8. Bergen client code for simulating noise level

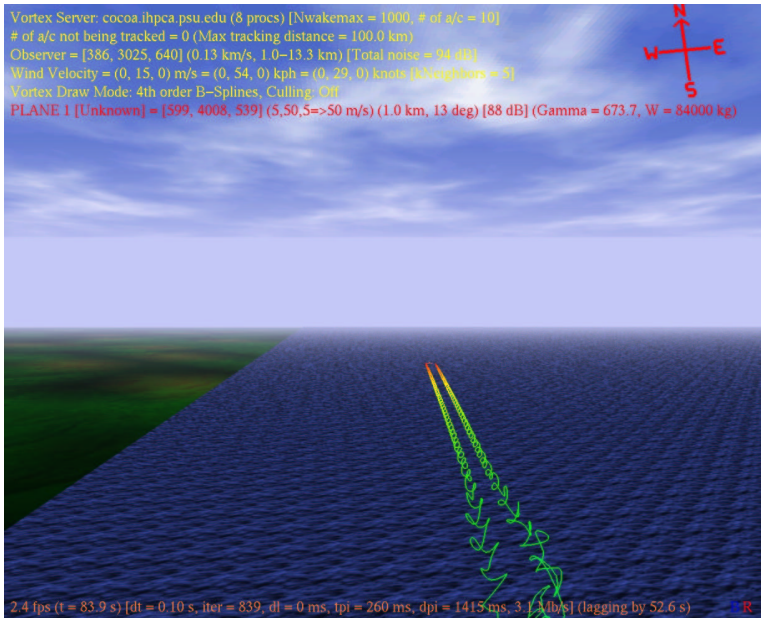


Figure 5.9. Screenshot of the Wake-Vortex simulation for a single aircraft from a visualization client

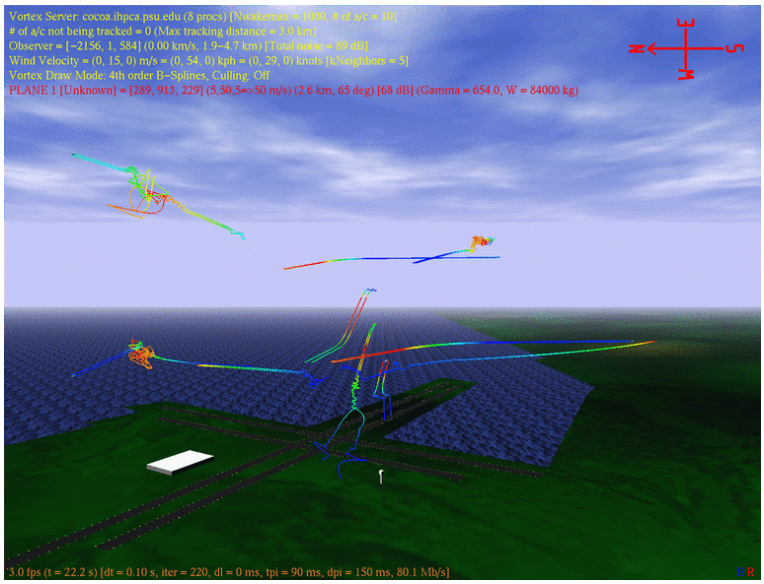


Figure 5.10. Screenshot of the Wake-Vortex simulation for several aircraft flying above the San Francisco (SFO) airport

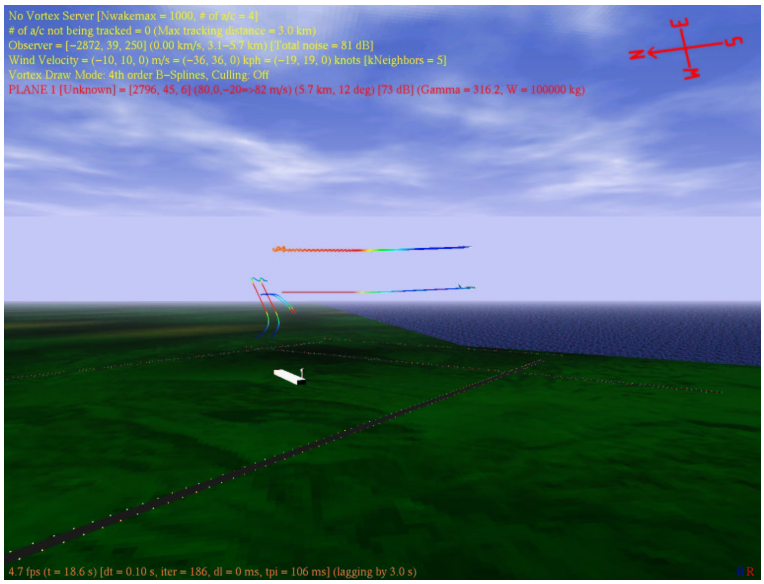


Figure 5.11. Screenshot of the Wake-Vortex simulation for several aircraft flying above the New York–John F. Kennedy (JFK) airport

Chapter 6

Computational Fluid Dynamics Simulations

6.1 Parallel Unstructured Maritime Aerodynamics (PUMA)

Parallel Unstructured Maritime Aerodynamics (PUMA), is a computer program for the analysis of internal and external non-reacting compressible flows over arbitrarily complex 3D geometries. It was originally written by Dr. Christopher W.S. Bruner as a part of his doctoral thesis [35]. The program is written entirely in ANSI C using MPI (Message Passing Interface) libraries for message passing. The resulting code is portable and exhibits good performance. The discretization of the fluid dynamics equations is based on the Finite Volume Method (FVM) that solves the full three-dimensional Navier-Stokes equations. The code supports mixed topology unstructured grids composed of tetrahedra, wedges, pyramids and hexahedra (bricks). A cell-centered finite-volume approach is used with the flow variables represented by values at the center of every cell.

PUMA may be run in a mode which preserves time accuracy or as a pseudo-unsteady formulation which enhances the convergence to a steady-state solution. The code uses dynamic memory allocation; thus, problem size is limited only by the amount of memory available on the machine. The finite-volume formulation requires approximately 582 bytes/cell and 634 bytes/face using double precision variables (not including message passing overhead). The development of PUMA was motivated by the need of work to compare the performance of several different algorithms on the same problem. Hence, PUMA implements a range of time-integration schemes like *Runge-Kutta, Jacobi* and various *Successive Over-relaxation Schemes*

(SOR), as well as both *Roe* and *Van Leer* numerical flux schemes. It also implements various monotone limiters used in second-order computations (*Venkatakrishnan, Barth, Van Albada, Superbee*). PUMA requires anywhere between $25000 - 30000$ floating point operations/iteration/cell, depending very much on the nature of the grid and cell connectivity, for a second order accurate computation running in double precision using SSOR time-integration and Roe fluxes. Several modifications and extensions have been incorporated into PUMA during the past several years [91, 92, 93, 94, 95].

6.1.1 Finite Volume Formulation

PUMA solves the Navier-Stokes equations for an ideal gas, expressing the conservation of mass, momentum and energy for a compressible Newtonian fluid in absence of external forces. In a FVM formulation, these equations are written in integral form for a bounded domain Ω within the surface boundary S, as shown in the equation below

$$\frac{\partial}{\partial t}\iiint_{\Omega}\vec{Q}\,dV + \iint_{S}\left(\vec{F}\cdot\hat{n}\right)\,dS = 0, \tag{6.1}$$

where the state vector $\vec{Q}$ is

$$\vec{Q} = \{\rho \;\; \rho u \;\; \rho v \;\; \rho w \;\; \rho e_0\},$$

and the normal component of the flux vector $\vec{F}\cdot\hat{n}$ is

$$\vec{F}\cdot\hat{n} = \{\rho V_n \;\; \rho u V_n \;\; \rho v V_n \;\; \rho w V_n \;\; \rho h_0 V_n\}.$$

Here, $\hat{n}$ denotes the unit surface normal, ρ denotes the air-density, u, v and w denote the speed of the flow in x, y and z directions, respectively, e_0 denotes the stagnation energy per unit mass,

h_0 denotes the stagnation enthalpy per unit mass, and V_n (=$\vec{V} \cdot \hat{n}$) denotes the magnitude of the normal component of the flow velocity. In the FVM formulation, the physical domain is broken up into many small control volumes of simple shape referred as *cells*. These cells are usually tetrahedra (4 faces) or bricks (6 faces). Once discretized for a constant volume, equation 6.1 becomes:

$$\Omega_i \frac{d}{dt}\overline{Q_i} + \sum_{j=1}^{\text{Nfaces}_i} \left[(F \cdot \hat{n}S)_j\right]_i = 0 \qquad i = 1, ..., \text{Ncells} \tag{6.2}$$

where $\overline{Q_i}$ is the volume average of $\vec{Q}$ over the cell i and $\left[(F \cdot \hat{n}S)_j\right]_i$ is the average flux through face j of cell i. As one can see from equation 6.2, the time rate of change of state vector $\vec{Q}$ within the domain Ω of a cell is balanced by the flux vector $\vec{F}$ across the cell boundary surface S. Between two consecutive time-steps, new flux values are calculated from updated state variables, forming a new residue which stands on the right hand side of equation 6.3 below:

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Omega_i} \sum_{j=1}^{\text{Nfaces}_i} \left[(F \cdot \hat{n}S)_j\right]_i. \tag{6.3}$$

For a steady-state calculation, the solver attempts to bring the residue to zero, so that the solution converges to a constant set of values. During an unsteady (time-accurate) calculation, the residue continuously changes over time.

6.1.2 Domain Decomposition

PUMA uses the *Single Program Multiple Data* (SPMD) parallelism–that is, the same code is replicated to each process. One process acts as the master and collects the results from all other processes, thus doing slightly more work than the other processes.

Domain decomposition is generally done with the aim to minimize communications cost

(i.e., time), by distributing the cells in the domain in some optimal way.

$$\text{communication time} = \text{latency} + \frac{\text{message size}}{\text{communications bandwidth}}$$

Following are the two commonly used approaches for domain decomposition:

1. **Minimization of inter-domain boundaries**: This approach is ideal for slow networks (lower communications bandwidth) or whenever many small messages are passed. It neglects the latency in the above expression. In this approach, the total length of inter-domain boundaries is minimized (which is length of all shared data). In other words, it minimizes the shared part of perimeter/surface area of the domain. It requires a-priori knowledge of the number of domains (compute nodes) and hence has to be done every time the problem is run on a different number of nodes. Popular techniques to implement this approach need the eigenvalues of the connectivity matrix, and thus can be expensive for large problems. However, this approach produces the best grids for large problems.

2. **Minimization of communications**: This approach is ideal for fast networks or whenever few big messages are passed. It neglects the second term in the above expression as communication bandwidth is assumed large. This approach minimizes the total number of messages exchanged. Since the above minimization corresponds to minimization of bandwidth of the cell connectivity matrix, this approach is independent of the number of domains (compute nodes). A large knowledge base exists in this area and is readily accessible so implementation is sound and easy.

These domain decomposition techniques are absolutely essential since we are dealing with unstructured grids in which the ordering of grid cells is generally arbitrary (especially, if say, Advancing Front Method was used to create the 2D/3D grids or Delaunay Triangulation for

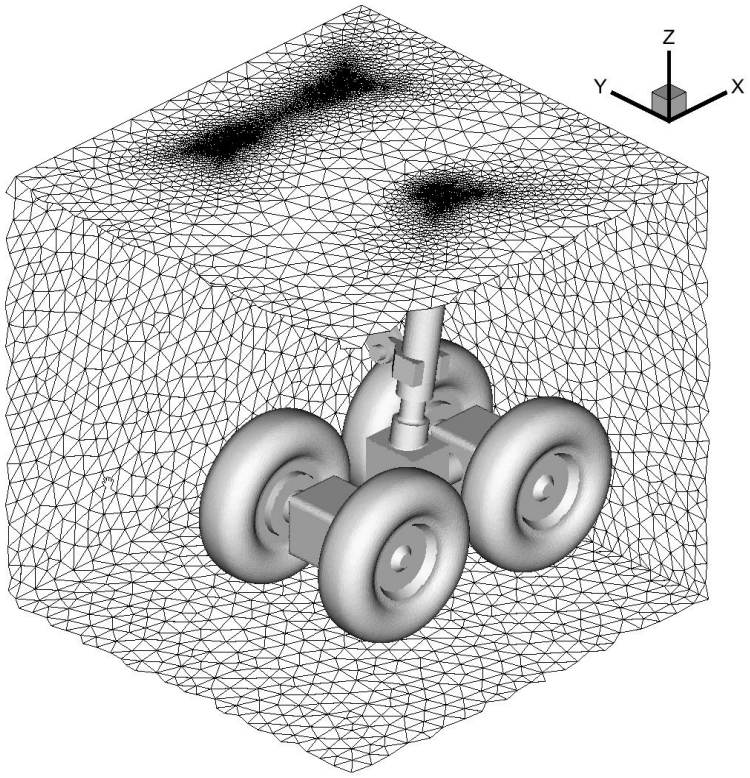


Figure 6.1. Section depicting unstructured grid for aircraft landing gear geometry (Courtesy Souliez [95])

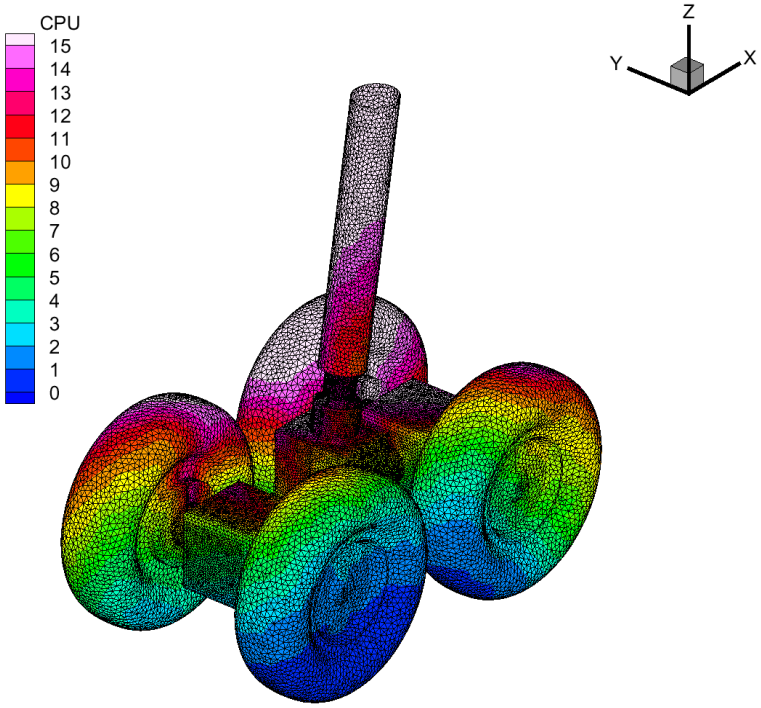


Figure 6.2. 16-way partitioning of the landing gear grid (figure 6.1) from the Gibbs-Poole-Stockmeyer reordering (Courtesy Souliez [95])

the 2D grids). Either approach can be used with PUMA, as domain decomposition is done by a separate program at the pre-processing stage. However, the latter approach is preferred as it is aimed at machines with high communications bandwidth, and also since the resulting decomposition is independent of the number of processors. Currently, the Gibbs-Poole-Stockmeyer (GPS) [96] algorithm is used to achieve this. For the first approach, a publicly available graph partitioning software package called METIS [97], which is being actively developed by the Department of Computer Science at University of Minnesota, can be used. A 16-way partitioning generated by the GPS approach as applied to a landing gear GRID (Fig. 6.1) is shown in Fig. 6.2.

6.1.3 Parallelization in PUMA

PUMA implements explicit and implicit time integration schemes on distributed memory computers. The solution procedure for the parallel implementation can be divided into six major steps.

1. Each compute node reads its own portion of the grid file at startup.

2. Cells are divided among the active compute nodes at runtime based on cell IDs, and only faces associated with local cells are read.

3. Faces on the interface surface between adjacent computational domains are duplicated in both domains. Fluxes through these faces are computed in both domains.

4. Solution variables are communicated between domains at every timestep which ensures that the computed solution is independent of the number of compute nodes.

5. Communication of the solution across domains is all that is required for first-order spatial accuracy, since Q_L and Q_R are simply cell averages to the first order. (Q being the

conservative set of flow variables).

6. If the left and right states are computed to higher-order, then Q_L and Q_R are shared explicitly with all adjacent domains. The fluxes through each face are then computed in each domain to obtain the residual for each local cell.

6.1.4 Boundary Condition Implementation

Implicit boundary conditions are used in PUMA when any of the implicit time-integration schemes are used (i.e., the boundary condition is evaluated at the next timestep as opposed to the current timestep). The boundary condition Jacobian ($\frac{\partial Q_{BC}}{\partial Q}$) is obtained numerically using central differences, which is completely general and does much to improve the maintainability of the code. There are several types of BCs implemented in PUMA, and more can be added without much difficulty as and when needed. The existing list of boundary conditions is given in table 6.1 [98]. A sample input file for PUMA is present in Appendix B.1.

Type	**Description**	**Variable values specified**
0	Do nothing	
1	Fixed at freestream	
2	Fixed at given values	ρ,u,v,w,p
3	First order extrapolation from the interior	
4	Second order extrapolation from the interior	
5	Inflow/outflow (Riemann)	
6	Specified p_0, T_0	p_0, T_0
7	Fixed back pressure pressure	p
8	Tangency	
9	No slip adiabatic wall	
10	No slip fixedtemperature wall T_{wall}	T_{wall}
11	Zero u velocity	
12	Zero v velocity	
13	Zero w velocity	

Table 6.1. PUMA boundary conditions

6.2 Real-Time Visualization

6.2.1 Modifications to PUMA

To achieve interactivity, several modifications were made to the PUMA code. The POSSE server component, DataServerMPI, was added to the `main()` function of PUMA. This modification was done by registering the cell-centered flow vector q and various important flow parameters in the code (refer to Appendix B.2). Several new global variables were added to receive iso-surface requests and store resulting iso-surfaces. An iso-surface extraction routine also had to be added to PUMA. Since we are dealing with unstructured mesh data consisting of tetrahedra, a variation of the classic "marching cubes" algorithm [99] is used for iso-surface extraction. The implementation is closely related to the marching cube algorithm except that the fundamental sampling structure here is a tetrahedron instead of a cube [100]. Since this implementation expects the flow data at the nodes of every tetrahedron, a subroutine to interpolate the flow data from cell-centers to the nodes also had to be added to PUMA.

The planar facet approximation to the iso-surface is calculated for each tetrahedron independently. The facet vertices are determined by linearly interpolating where iso-surface cuts the edges of the tetrahedron. This gives rise to 8 different cases, 7 of which are illustrated in Fig. 6.3. The hollow and filled circles at the vertices of the tetrahedron indicate that the vertices are on different sides of the iso-surface. The case not illustrated is where all the vertices are either above of below the iso-surface, and no facets are generated in this situation.

In PUMA, when the request for an iso-surface reaches the master processor (in the form of an update to a *read-write* variable specifying the flow parameter of interest), it is broadcast to all the processors, which act on their local tetrahedral elements to compute their portion of the iso-surface in parallel before sending it back to the master for consolidation. Once

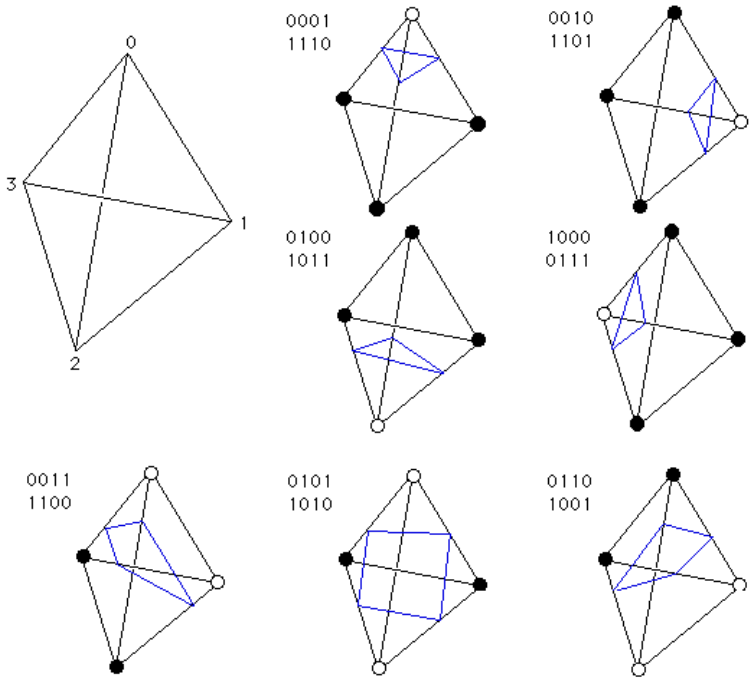


Figure 6.3. Different cases when iso-surface cuts the face of the tetrahedron (Courtesy Bourke [100])

the data is consolidated by the master, it is send to the client which is waiting to receive it. Alternatively, the client can also choose to collect the data individually from each processor in the computation.

6.2.2 Graphical User Interface

A graphical user interface (GUI) client is used to connect to the computational steering server. Figures 6.4 and 6.5 show a screenshot from the POSSE GUI client application. A PUMA specific client is used to extract and visualize iso-surfaces. The client is written in C++ with the cross-platform FLTK [101] API for the GUI and VTK [102] for the 3D visualization. Since VTK is written on top of OpenGL, the resulting application can benefit from the OpenGL hardware acceleration available on most modern graphics chipsets. Figure 6.6 illustrates a simple “Hello World” FLTK application written in C++ to draw a cylinder, and Fig. 6.8 shows

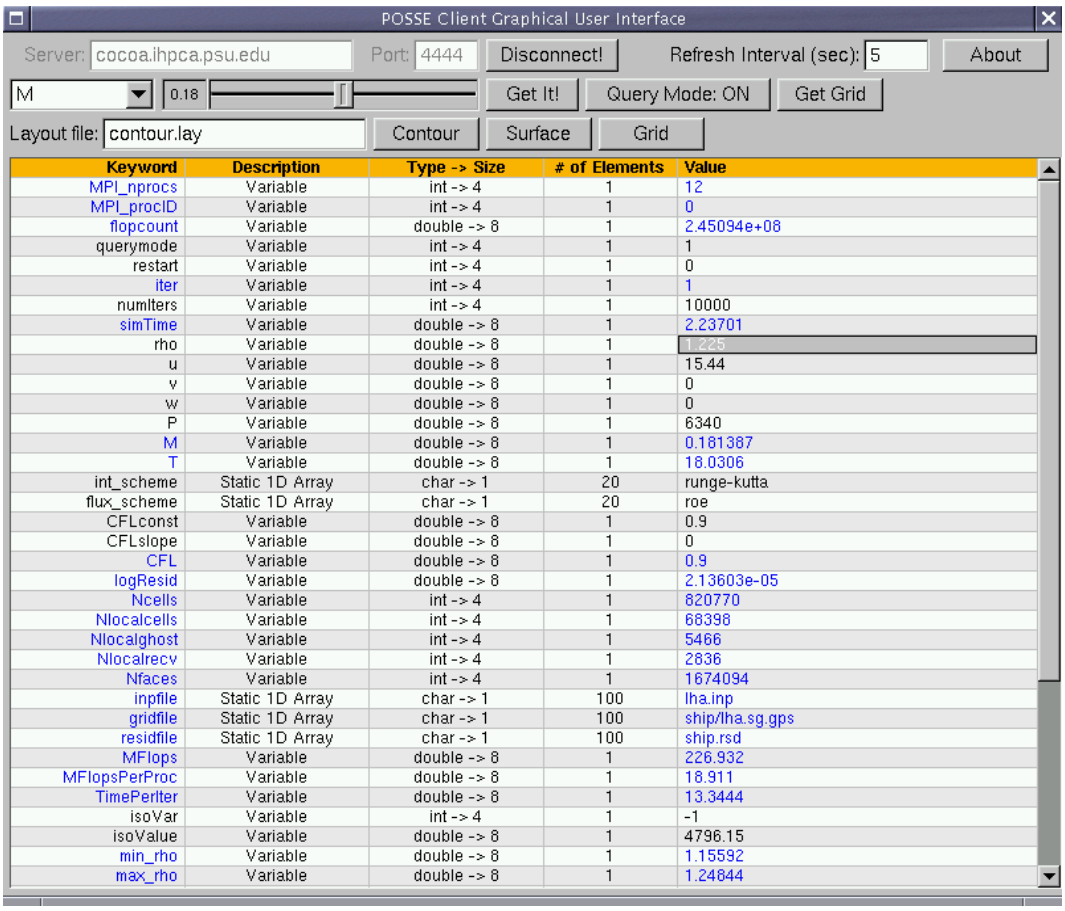
its output. Figure 6.7 illustrates a simple VTK application written in C++ to draw a cylinder, and Fig. 6.9 shows its output. Figure 6.10 shows the UML diagram for the GUI design.

The GUI provides information on the application data registered by the POSSE `DataServer` application. For every registered data present in a given row, the following information is displayed by the columns:

1. *keyword*: the string handle identifying the registered data.
2. *description*: the data type stating whether the registered data is a variable, a dynamic array, static array or a custom structure.
3. *variable type and size*: whether the data is of a standard system type (char, int, float, double, etc.) or a custom user-defined type.
4. *number of elements*: the number of elements if the data is an array.
5. *value*: the current value of the data if it is a variable or a string (character array).

A blue-colored value field indicates read-only data which cannot be modified by the client, whereas the black-colored value field indicates read-write data which can be modified by clicking and changing the value in the text-box. The “Refresh Interval” field at the top-right corner is used to specify the client refresh interval which determines the rate at which a new copy of the data is obtained from the server.

A drop-down menu is provided to choose the flow variable for which iso-surfaces are requested. After the numerical value for the iso-surface has been selected, a request is sent to the flow solver which responds by extracting the iso-surface for the given flow parameters on each of the processors; then collecting it on the master processor and sending the final iso-surface to the client. The iso-surface is then displayed in a separate window in stereo along



Keyword	Description	Type -> Size	# of Elements	Value
MPI_nprocs	Variable	int -> 4	1	12
MPI_procID	Variable	int -> 4	1	0
flopcount	Variable	double -> 8	1	2.45094e+08
querymode	Variable	int -> 4	1	1
restart	Variable	int -> 4	1	0
iter	Variable	int -> 4	1	1
numIters	Variable	int -> 4	1	10000
simTime	Variable	double -> 8	1	2.23701
rho	Variable	double -> 8	1	1.225
u	Variable	double -> 8	1	15.44
v	Variable	double -> 8	1	0
w	Variable	double -> 8	1	0
P	Variable	double -> 8	1	6340
M	Variable	double -> 8	1	0.181387
T	Variable	double -> 8	1	18.0306
int_scheme	Static 1D Array	char -> 1	20	runge-kutta
flux_scheme	Static 1D Array	char -> 1	20	roe
CFLconst	Variable	double -> 8	1	0.9
CFLslope	Variable	double -> 8	1	0
CFL	Variable	double -> 8	1	0.9
logResid	Variable	double -> 8	1	2.13603e-05
Ncells	Variable	int -> 4	1	820770
Nlocalcells	Variable	int -> 4	1	68398
Nlocalghost	Variable	int -> 4	1	5466
Nlocalrecv	Variable	int -> 4	1	2836
Nfaces	Variable	int -> 4	1	1674094
inpfile	Static 1D Array	char -> 1	100	lha.inp
gridfile	Static 1D Array	char -> 1	100	ship/lha.sg.gps
residfile	Static 1D Array	char -> 1	100	ship.rsd
MFlops	Variable	double -> 8	1	226.932
MFlopsPerProc	Variable	double -> 8	1	18.911
TimePerIter	Variable	double -> 8	1	13.3444
isoVar	Variable	int -> 4	1	-1
isoValue	Variable	double -> 8	1	4796.15
min_rho	Variable	double -> 8	1	1.15592
max_rho	Variable	double -> 8	1	1.24844

Figure 6.4. A POSSE GUI to connect to the flow solver

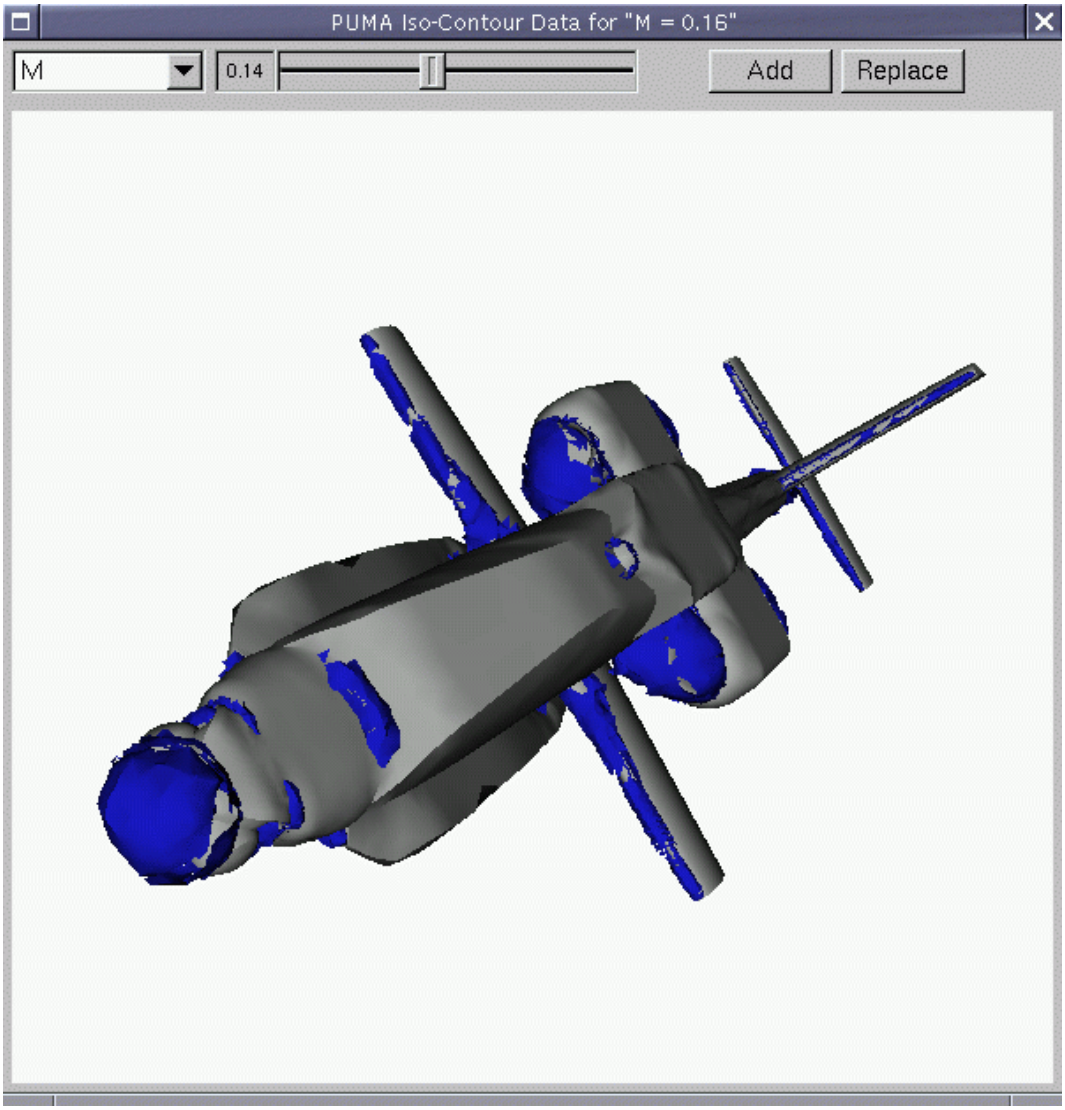


Figure 6.5. A POSSE client application depicting iso-surfaces for a flow solution over the Apache helicopter

```
#include <iostream>
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Button.H>

void clickAction(Fl_Button *o, void *v)
{
  cout << "Hello, World!" << endl;
}

int main(int argc, char **argv)
{
  // Create new FLTK window of size 300x180
  Fl_Window *window = new Fl_Window(300, 180);

  // Create a button of size 260x100 with upper-left corner at point (20,40)
  Fl_Button *button = new Fl_Button(20, 40, 260, 100, "Click Here!");
  // Register a callback function to be called when button is clicked
  button->callback((Fl_Callback*) clickAction);

  // Set visual properties of the button
  button->box(FL_UP_BOX);
  button->labelfont(FL_HELVETICA_BOLD_ITALIC);
  button->labelsize(36);
  button->labeltype(FL_SHADOW_LABEL);

  // Add the button to the window
  window->add(button);

  // Finish setting the window
  window->end();
  // Display the window
  window->show(argc, argv);

  return Fl::run(); // Run the main FLTK interaction loop
}
```

Figure 6.6. A simple FLTK application written in C++

```
#include "vtkCylinderSource.h"
#include "vtkPolyDataMapper.h"
#include "vtkActor.h"
#include "vtkRenderer.h"
#include "vtkRenderWindow.h"
#include "vtkRenderWindowInteractor.h"

int main(int argc, char *argv[])
{
  // This creates a polygonal cylinder model with 50 circumferential facets.
  vtkCylinderSource *cylinder = vtkCylinderSource::New();
  cylinder->SetResolution(50);

  // The mapper pushes the geometry into the graphics library
  vtkPolyDataMapper *cylinderMapper = vtkPolyDataMapper::New();
  cylinderMapper->SetInput(cylinder->GetOutput());

  // The actor is a grouping mechanism: besides the geometry (mapper), it
  // also has a property, transformation matrix, and/or texture map.
  vtkActor *cylinderActor = vtkActor::New();
  cylinderActor->SetMapper(cylinderMapper);
  cylinderActor->GetProperty()->SetColor(0.0, 0.0, 1.0); // Set blue color

  // Create the graphics structure. The renderer renders into the
  // render window. The render window interactor captures mouse events.
  vtkRenderer *ren = vtkRenderer::New();
  vtkRenderWindow *renWin = vtkRenderWindow::New();
  renWin->AddRenderer(ren);        // Add the renderer to the window
  vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
  iren->SetRenderWindow(renWin);   // Add the window to the interactor

  ren->AddActor(cylinderActor);    // Add the actor to the renderer
  ren->SetBackground(1.0,1.0,1.0); // Set background to white
  renWin->SetSize(600, 600);       // Set the background size
  renWin->Render();                // Render the display window
  iren->Start();                   // Start the event loop

  // Exiting by deleting the instances that have been created.
  cylinder->Delete();
  cylinderMapper->Delete();
  cylinderActor->Delete();
  ren->Delete();
  renWin->Delete();
  iren->Delete();
  return 0;
}
```

Figure 6.7. A simple VTK application written in C++

Figure 6.8. Output from the FLTK application shown in Fig. 6.6

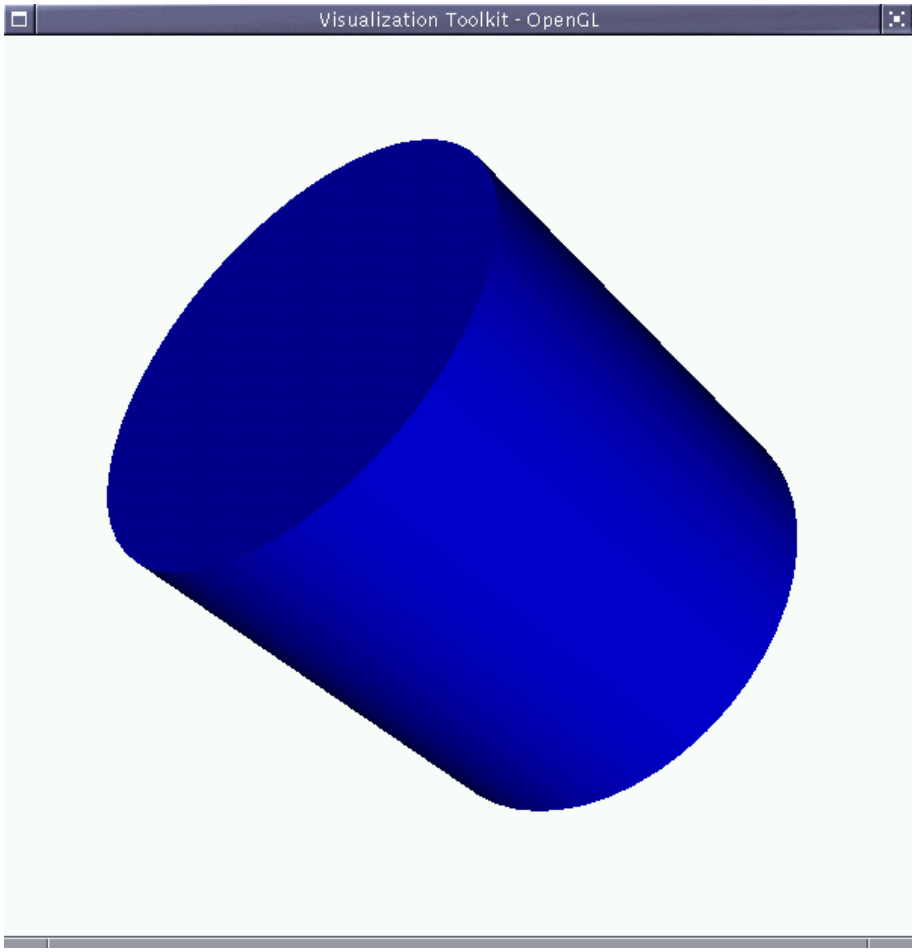


Figure 6.9. Output from the VTK application shown in Fig. 6.7

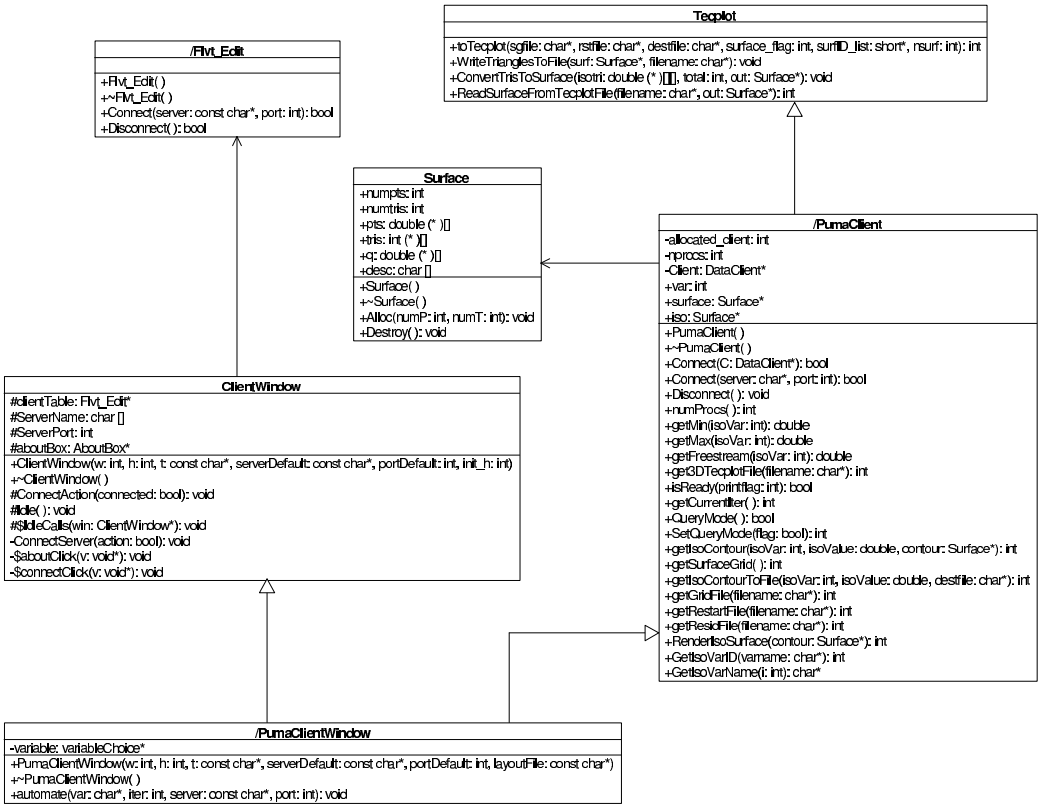


Figure 6.10. UML Diagram for client GUI classes

with the surface of the object of interest (i.e., helicopter, ship, etc.). There are two modes provided for querying iso-surfaces. In the default mode, the user queries for iso-surfaces while the flow-solver is computing the solution for the next iteration. This mode can be slow if the user wants to query several iso-surfaces one after another, as the flow solver cannot answer the client requests until the current iteration is over. For greater responsiveness, the user can enable the "Query" mode which temporarily halts the PUMA computations, so that the flow-solver can exclusively devote all its CPU cycles to answering the client iso-surface requests without any lag. There is also an option "Get Grid", which will download the entire grid and the updated solution file and construct a Tecplot [103] volume grid file for the user to browse locally using Tecplot. Several iso-surfaces can be layered on top of each other to compare the differences between two iterations. Also, as seen in the Fig. 6.11, all the retrieved data can be optionally filtered through a user-specified Tecplot layout file to create a powerful and highly-customized visualization display on a separate Tecplot window.

6.2.3 Scalability and Dimensional Reduction

Traditionally, visualization of CFD data has been done by consolidating the 3D data from a parallel simulation into a single file, transporting it over the network to a single or dual-processor graphics workstation, and post-processing it to extract the required data. This is both slow and cumbersome.

A significant advantage arising from the use of POSSE within PUMA is that of "scalability". For an evenly distributed grid, the number of grid faces on each processor of a parallel computation is N/P where N is the total number of grid faces and P is the total number of processors. Here, the scalability comes from the fact that the extraction of iso-surfaces is done on a parallel machine rather than the traditional and non-scalable way of consolidating the data

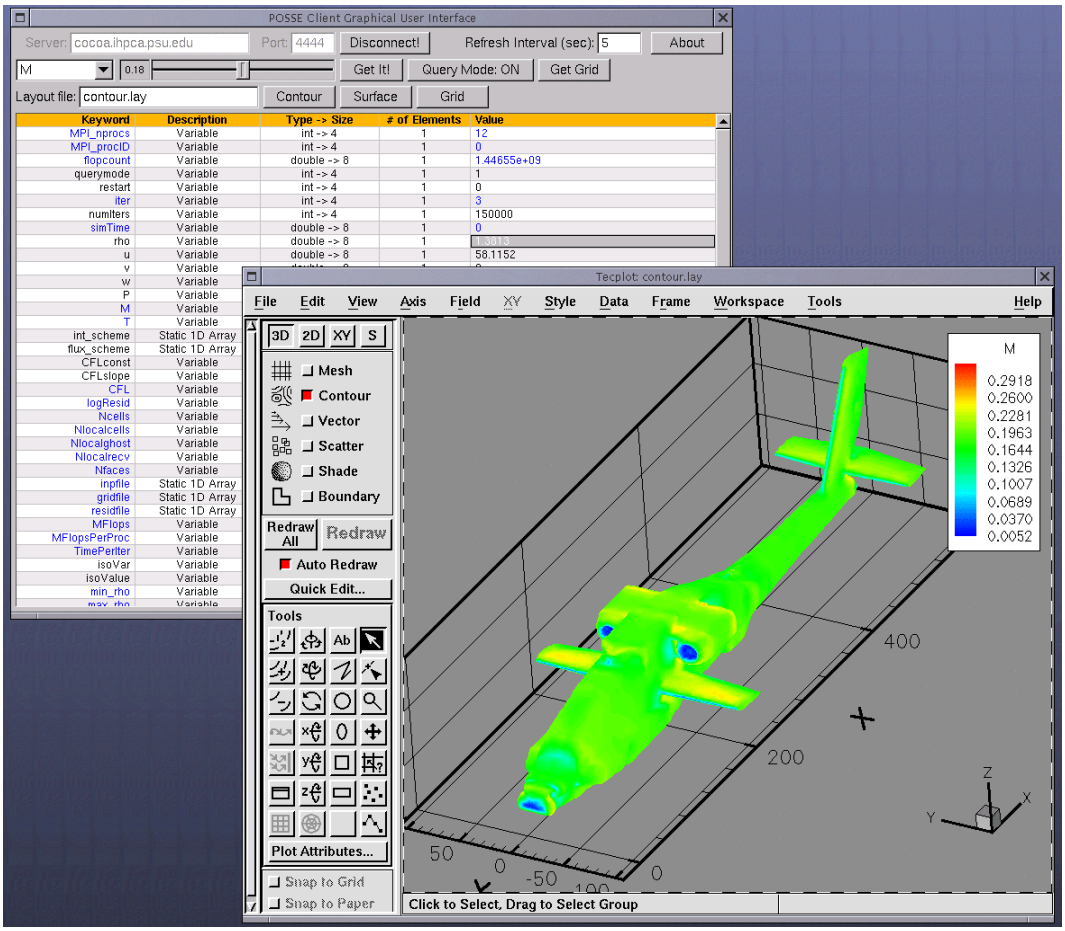


Figure 6.11. Screenshot of PUMA GUI with Tecplot filter

into a file and post-processing it. Thus, the computational time for extraction of an iso-surface is $O(N/P)$ as compared to the sequential algorithm which takes $O(N)$ for the same procedure.

Another significant advantage arising due to the nature of data typically used for flow visualization is that of "dimensional reduction." The dimensional reduction comes from the fact that the data required for the CFD simulation lives in a higher dimensional space (3D) than the data that is required for visualization (which are in 2D and 1D space for iso-surfaces and chord plots, respectively). The total number of grid faces, N, are expected to be $O(n^3)$ where n is the average number of grid faces in each direction. The total number of triangles in an iso-surface obtained from this grid are expected to be only $O(n^2)$ or $O(N^{2/3})$. Thus, the average time required for the communication of this iso-surface data between the client and the server is $O(N^{2/3})$. Scalability and dimensional reduction combine to give an expected $O(N^{2/3}/P)$ data coming from each processor during the parallel computation of an iso-surface.

This approach is also scalable both in "space" and "time." By monitoring a time-dependent simulation, the entire time history can be accessed, whereas it would be prohibitive to store the time history in files which could possibly take tens or even hundreds of gigabytes of disk space even for a moderately complex case. Figures 6.12, 6.13 and 6.14 show the relative size of the iso-surfaces with the total size of the grid for varying X-coordinate, Mach number and C_p values for the flow over an Apache helicopter geometry. It can be clearly seen that the average number of triangles in an iso-surface is less than 1% of the total number of grid faces for this case. Figure 6.15 depicts another case where two grids with vastly varying sizes are used for extraction of iso-surfaces with varying values of X-coordinate. Here, although the larger grid (with 2.3 million faces) is more than twice as large as the smaller grid (with 1.1 million faces), the average number of triangles in its iso-surfaces is only about 55% more. The theoretical average difference is expected to be $(2.3/1.1)^{2/3} = 1.6352$ or 63.52% more, which is close to

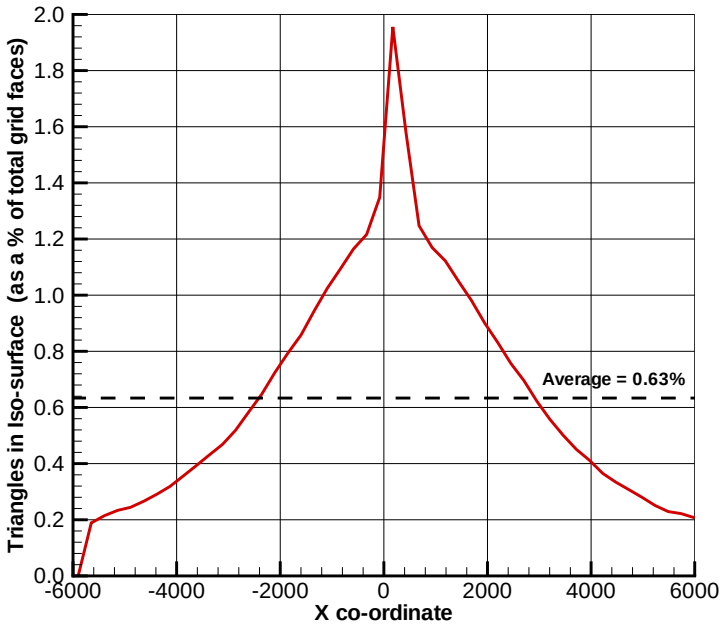


Figure 6.12. Relative size of X-coordinate iso-surfaces for the Apache case

the result obtained above.

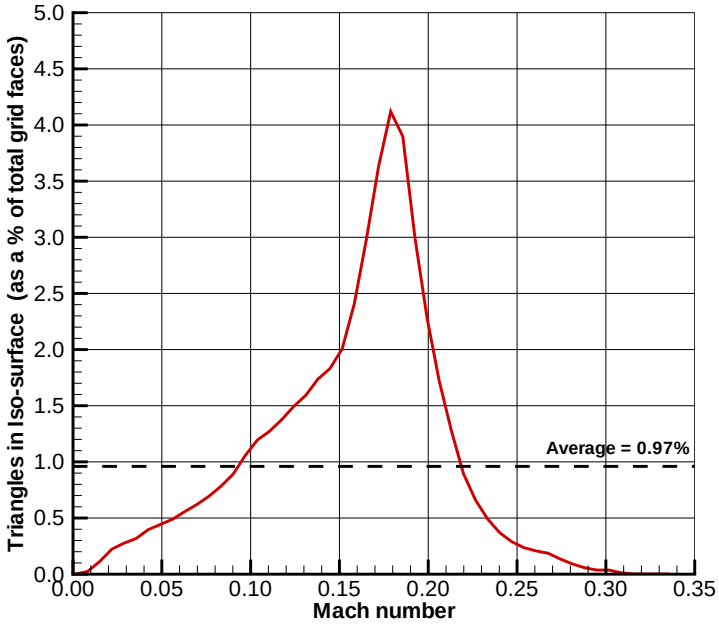


Figure 6.13. Relative size of Mach number iso-surfaces for the Apache case

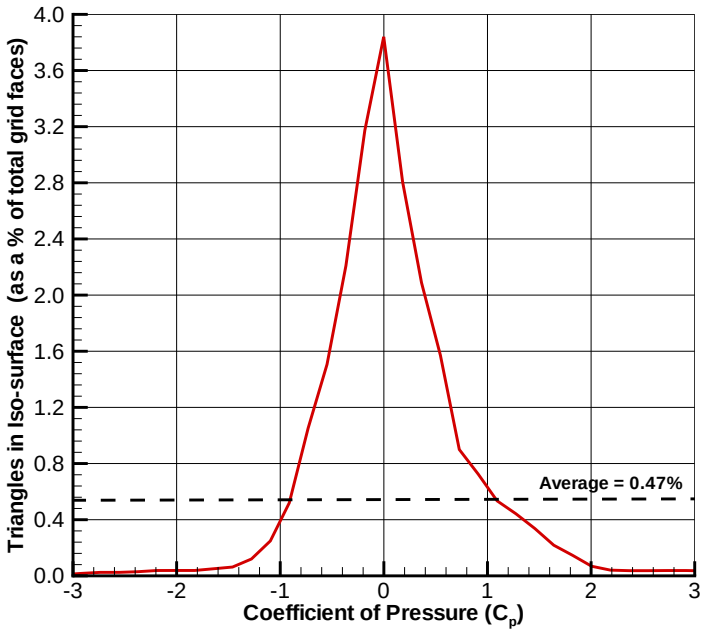


Figure 6.14. Relative size of C_p iso-surfaces for the Apache case

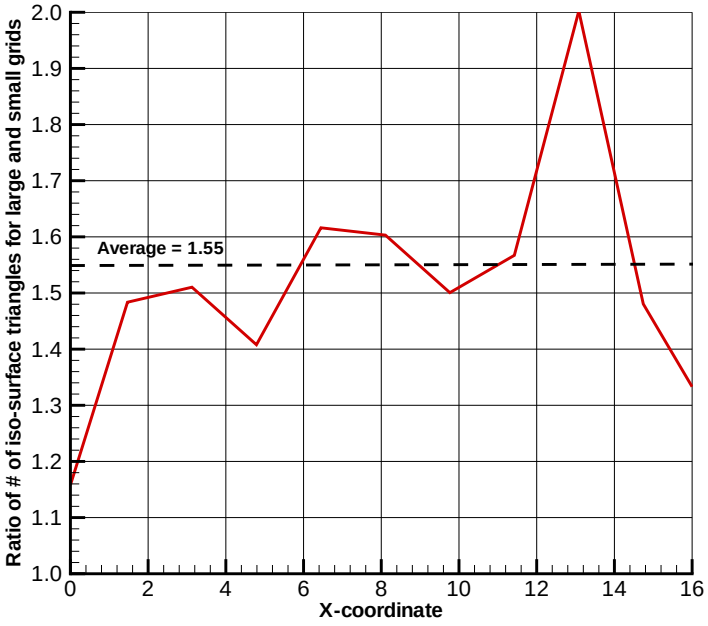


Figure 6.15. Grid sensitivity analysis for iso-surfaces

Chapter 7

Conclusions and Future Work

During the course of this research effort, several contributions have been made to enable remote real-time visualization of parallel simulations. A general-purpose, lightweight, portable and efficient computational steering system, POSSE, has been developed to achieve this objective. POSSE has been completely written in C++, making it fast and powerful, while hiding most of the code complexities from the user. The advanced object-oriented features of C++ have given POSSE an edge over existing steering systems written in older languages. POSSE has proven to be a very powerful, yet easy to use software with a high rate of acceptance and approval at Penn State. If scientists and engineers are given an easy to use software system with a mild learning curve, they will use it.

As a demonstration of the steering approach, two important applications using POSSE have been developed. The first application, a wake-vortex hazard avoidance system, enables the real-time simulation and visualization of wake-vortex hazards. For this, an MPI-based parallel wake-vortex simulation code has been developed and integrated with POSSE. Further integration with CAVELib enabled the operation of this software system in an integration virtual environment providing the benefits of improved input and display devices. This system opens a new way for the ATC to effectively deal with the wake-vortex hazard problem and to improve the capacity and safety of large airports.

The second application, an interactive Computational Fluid Dynamics (CFD) system, makes the on-line visualization and steering of complex flow simulations possible. For this, POSSE has been successfully coupled to a C-based flow solver, PUMA. Benefits of scalability

and dimensional reduction arising from this approach have been imperative in the development of this system. This system makes the visualization of flow simulations easier, efficient, and most-importantly without any delay as and when the results are generated, thus giving the user much greater flexibility in interacting with the huge amounts of data.

At a more basic level, the ability to interact and visualize a complex solution as it unfolds using the depth cue of the stereoscopic display and the real-time nature of the computational steering system opens a whole new dimension to the scientist and engineer for interacting with their simulations.

7.1 Future Work

While a significant amount of work has been done during the course of this research, there are several aspects that need further improvement. Currently, POSSE only supports periodic polling as a way to monitor server-side data. While this does not adversely affect the applications described in this thesis, polling may introduce substantial overhead for applications that need to retrieve data from the server at a high frequency. To avoid this, a publish-subscribe methodology can be implemented. Also, POSSE can be enhanced to support more than one data registration block per application enabling the use of several `DataServer`s in a single program.

For the wake-vortex simulation system, fault-tolerance issues have been completely neglected. Currently, the failure of any single compute node can lead to the failure of the entire application. The real-time steering is a great proof-of-concept, but further research needs to be carried out before this system is to be ever considered for serious use for ATC purposes. Stricter specifications for maintaining real-time nature of the simulation have to be arrived upon and mechanisms for implementing them have to be researched. Dedicated networks with

quality-of-service guarantees may need to be utilized to maintain reliable and responsive network connections which are unaffected by other network traffic. The user-interface of the visualization client also needs to be greatly improved to support better navigation. Support for better aircraft, airport and trajectory models also needs to be included in the visualization client.

For the interactive CFD system, sampling in time along a lower dimensional space has not yet been implemented. If included, a 2D surface or a 1D slice can be selected and time-averaged flow data on that lower-dimensional surface can be obtained. This information would be of great interest for unsteady flow simulations. This time-dependent data can also be archived on the local disk of every node of the parallel simulation making it possible to do time-dependent queries from stored time slices in a more scalable way. These archives can also be used for efficiently restarting the simulation to get data from any intermediate time. This would enable jumping around in time with much less overhead than would be incurred if the simulation had to be started from scratch. This would open up a whole new “dimension” in the monitoring and visualization of unsteady flow data.

REFERENCES

[1] Hypotenuse Research Triangle Institute. Wake Vortex Detection System: Engineered for Efficiency and Safety. http://www.rti.org/hypo_etc/winter00/vortex.cfm, 2001.

[2] Perry, T. S. In Search of the Future of Air Traffic Control. *IEEE Spectrum*, **34(8)**, pp. 18–35, August 1997.

[3] Reitinger, B. On-line Program and Data Visualization of Parallel Systems in a Monitoring and Steering Environment. Dipl.-Ing. Thesis, Johannes Kepler University, Linz, Austria, Department for Graphics and Parallel Processing, http://www.gup.uni-linz.ac.at/thesis/diploma/bernhard_reitinger/thesis.pdf, January 2001.

[4] Gu, W., Eisenhauer, G., Kraemer, E., Schwan, K., Stasko, J., Vetter, J., and Mallavarupu, N. Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs. *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pp. 433–429, February 1995.

[5] Parker, S., Miller, M., Hansen, C., and Johnson, C. An Integrated Problem Solving Environment: The SCIRun Computational Steering System. *IEEE Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, **7**, pp. 147–156, 1998.

[6] Ba, I., Malon, C., and Smith, B. Design of the ALICE Memory Snooper. http://www.mcs.anl.gov/ams, 1999.

[7] Jablonowski, D., Bruner, J., Bliss, B., and Haber, R. VASE : The Visualization and Application Steering Environment. *Proceedings of Supercomputing '93*, pp. 560–569, 1993.

[8] Geist, II, G. A., Kohl, J. A., and Papadopoulos, P. M. CUMULVS: Providing Fault Tolerance, Visualization, and Steering of Parallel Applications. *The International Journal of Supercomputer Applications and High Performance Computing*, **11(3)**, pp. 224–235, Fall 1997.

[9] van Liere, R., and van Wijk, J. J. CSE : A Modular Architecture for Computational Steering. In Göbel, M., David, J., Slavik, P., and van Wijk, J. J., editors, *Virtual Environments and Scientific Visualization '96*, pp. 257–266. Springer-Verlag Wien, 1996.

[10] Shaffer, E., Reed, D., Whitmore, S., and Schaeffer, B. Virtue: Performance Visualization of Parallel and Distributed Applications. *IEEE Computer*, **32(12)**, pp. 44–51, December 1999.

[11] Mulder, J. D., van Wijk, J. J., and van Liere, R. A Survey of Computational Steering Environments. *Technical Report, Faculty of Mathematics and Computer Science, University of Technology, Eindhoven*, September 1998.

[12] Mathworks. MATLAB: The Language of Technical Computing. `http://www.mathworks.com/products/matlab/`, 2001.

[13] Lin, F., Larkin, S., Grant, A., and Slinger, R. AVS: Advanced Visual Systems. `http://www.avs.com/`, 2001.

[14] Goldman, J., and Roy, T. M. The Cosmic Worm. *IEEE Computer Graphics and Applications*, **14(4)**, pp. 12–14, July 1994.

[15] Cruz-Neira, C., Leigh, J., Papka, M., Barnes, C., Cohen, S. M., Das, S., Engelmann, R., Hudson, R., Roy, T., Siegel, L., Vasilakis, C., DeFanti, T. A., and Sandin, D. J. Scientists in Wonderland: A Report on Visualization Application in the CAVE Virtual Reality Environment. *IEEE Proceedings on Virtual Reality*, pp. 59–66, 1993.

[16] Johnson, C. R., and Parker, S. G. A Computational Steering Model Applied to Problems in Medicine. *Proceedings of the IEEE Conference on Supercomputing*, November 1994.

[17] Johnson, C., Parker, S. G., Hansen, C., Kindlmann, G. L., and Livnat, Y. Interactive Simulation and Visualization. *IEEE Computer*, **32(12)**, pp. 59–65, December 1999.

[18] Parker, S. G., R., J. C., and D., B. Computational Steering Software Systems and Strategies. *IEEE Computational Science and Engineering*, **4(4)**, pp. 50–55, December 1997.

[19] Julier, S., King, R., Colbert, B., Durbin, J., and Rosenblum, L. The Software Architecture of a Real-Time Battlefield Visualization Virtual Environment. *Proceedings of IEEE Virtual Reality*, pp. 29–36, 1999.

[20] Taylor, V., Chen, J., Disz, T., Papka, M., and Stevens, R. Interactive Virtual Reality in Simulations: Exploring Lag Time. *IEEE Computational Science and Engineering*, **3(4)**, pp. 46–54, 1996.

[21] Richwine, D. M., Curry, R. E., and Tracy, G. V. A Smoke Generator System for Aerodynamic Flight Research. *NASA-TM-4137*, January 1989.

[22] McCormick, B. W. Aircraft Wakes: A Survey of the Problem. *Keystone Presentation at FAA Symposium on Aviation Turbulence*, March 1971.

[23] Mikhailov, Y., and Vyshinsky, V. Vortex Wake Safety: The Problem of Encountering Aircraft. *Third Seminar on RRDPAE '98*, 1998.

[24] Reimer, H. M., and Vicroy, D. D. A Preliminary Study of a Wake Vortex Encounter Hazard Boundary for a B737-100 Airplane. *NASA TM-110223*, April 1996.

[25] Airdisaster.com. Investigation: US Air Flight 427. `http://www.airdisaster.com/investigations/us427/usair427.shtml`, 1997.

[26] AirDisaster.com. American Airlines Flight 587 Disaster. `http://www.airdisaster.com/photos/aa587/photo.shtml`, November 2001.

[27] Stewart, E. C. A Piloted Simulation Study of Wake Turbulence on Final Approach. *AIAA 97-0057*, August 1998.

[28] Hinton, D. A. Aircraft Vortex Spacing System (AVOSS) Conceptual Design. *NASA TM-110184*, August 1995.

[29] Perry, R. B., Hinton, D. A., and Stuever, R. A. NASA Wake Vortex Research for Aircraft Spacing. *AIAA 97-0057*, pp. 56–74, January 1997.

[30] Riddick, S. E., and Hinton, D. A. An Initial Study of the Sensitivity of Aircraft Vortex Spacing System (AVOSS) Spacing Sensitivity to Weather and Configuration Input Parameters. *NASA TM-110184*, January 2000.

[31] Hinton, D. A., Charnock, J. K., Bagwell, D. R., and Grigsby, D. W. NASA Aircraft Vortex Spacing System Development Status. *AIAA 99-0753*, January 1999.

[32] Hinton, D. A., Charnock, J. K., and Bagwell, D. R. Design of an Aircraft Vortex Spacing System for Airport Capacity Improvement. *AIAA 2000-0622*, January 2000.

[33] Anderson, J. D. *Computational Fluid Dynamics: The Basics with Applications*. McGraw Hill, 6th edition, 1995 (ISBN: 0070016852).

[34] Modi, A., Sezer, N., Long, L. N., and Plassmann, P. E. Scalable Computational Steering System for Visualization of Large Scale CFD Simulations. *AIAA 2002-2750*, June 2002.

[35] Bruner, C. W. S. Parallelization of the Euler equations on unstructured grids. PhD Dissertation, Virginia Polytechnic Institute and State University, Department of Aerospace Engineering, `http://scholar.lib.vt.edu/theses/public/`, May 1996.

[36] Wickens, C., and Baker, P. Cognitive issues in virtual reality, 1995.

[37] Isdale, J. What is VR? `http://www.isdale.com/jerry/VR/WhatIsVR.html`, September 1998.

[38] Sutherland, I. E. The Ultimate Display. In *Proc. 3rd IFIP Congress*, volume **2**, , pp. 506–508, Washington D.C., U.S.A., 1965. Spartan Books.

[39] Steuer, J. Defining Virtual Reality: Dimensions Determining Telepresence. *Journal of Communication*, **42(4)**, pp. 73–93, 1992.

[40] Azuma, R. T. A Survey of Augmented Reality. *SIGGRAPH '95 Proceedings*, pp. 1–38, August 1995.

[41] Stuart, R. *The Design of Virtual Environments*. McGraw-Hill, 1996 (ISBN: 0070632995).

[42] Vince, J. *Virtual Reality Systems*. SIGGRAPH Series, Addison-Wesley, 1995 (ISBN: 0201876876).

[43] Sutherland, I. E. A head-mounted three-dimensional display. *AFIPS Conference Proceedings*, **33**, pp. 757–764, 1968.

[44] Ascension Technology. Flock of Birds. http://www.ascension-tech.com/products/flockofbirds/, 2002.

[45] Cruz-Neira, C. Virtual Reality Based on Multiple Projection Screens: The CAVE and its Applications to Computational Science and Engineering. Ph.D. Dissertation, University of Illinois at Chicago, Department of Electrical Engineering and Computer Science, May 1995.

[46] Cruz-Neira, C., Sandin, D. J., DeFanti, T. A., Kenyon, R. V., and Hart, J. C. The CAVE: Audio Visual Experience Automatic Virtual Environment. *ACM Communications*, **35(6)**, pp. 64–72, June 1992.

[47] Cruz-Neira, C., Sandin, D. J., and DeFanti, T. A. Surround-screen Projection-based Virtual Reality: The Design and Implementation of the CAVE. *Proceedings of the 20th annual conference on Computer graphics*, pp. 135–142, August 1993.

[48] CAVE. Fake Space Systems. http://www.fakespacesystems.com, 2001.

[49] Pape, D., Cruz-Neira, C., and Czernuszenko, M. CAVE User's Guide. http://www.evl.uic.edu/pape/CAVE/prog/CAVEGuide.html, May 1997.

[50] Flurchick, K. RAVE documentation. http://www.ecu.edu/rave/Doc/Description.html, 2000.

[51] VRCO. CAVELib Users Manual. http://www.vrco.com/CAVE_USER/caveuser_program.html, 2001.

[52] Bierbaum, A. VR Juggler Homepage. http://www.vrjuggler.org, 2000.

[53] Silicon Graphics Inc. OpenGL Graphics Library. http://www.opengl.org/, 2001.

[54] Woo, M., Neider, J., Davis, T., and Shreiner, D. *OpenGL Programming Guide*. Addison-Wesley, 3rd edition, August 1999 (ISBN: 0201604582).

[55] Sense8. WorldToolKit. `http://www.sense8.com`, 2001.

[56] Park, K., Cho, Y., Krishnaprasad, N., Scharver, C., Lewis, M., Leigh, J., and Johnson, A. CAVERNsoft G2: A Toolkit for High Performance Tele-Immersive Collaboration. 2001.

[57] Renambot, L., Bal, H. E., Germans, D., and Spoelder, H. J. W. CAVEStudy: an Infrastructure for Computational Steering in Virtual Reality Environments. *Proceedings of the Ninth International Symposium on High-Performance Distributed Computing*, pp. 239–246, August 2000.

[58] Federal Aviation Agency. `http://www.faa.gov/`, 2001.

[59] Young, W., Tsai, M.-T., and Chuang, L.-M. Air Traffic Control System Management. *Proceedings of the IEEE National Aerospace and Electronics Conference (NAECON)*, pp. 494–498, 2000.

[60] Azuma, R., Neely, H., Daily, M., and Correa, M. Visualization of Conflicts and Resolutions in a “Free Flight” Scenario. *Proceedings of IEEE Visualization ’99*, pp. 433–557, 1999.

[61] Azuma, R., Daily, M., and Krozel, J. Advanced Human-Computer Interfaces for Air Traffic Management and Simulation. *Proceedings of 1996 AIAA Flight Simulation Technologies Conference*, pp. 656–666, July 1996.

[62] Azuma, R., Neely, H., Daily, M., and Geiss, R. Visualization Tools for Free Flight Air-Traffic Management. *IEEE Computer Graphics and Applications*, pp. 32–36, September 2000.

[63] NASA. Highway In The Sky (HITS). `http://nctn.hq.nasa.gov/innovation/Innovation72/aviation.htm`, 2001.

[64] NASA. FutureFlight Central. `http://ffc.arc.nasa.gov/`, 2001.

[65] Flynn, M. J. Some computer organizations and their effectiveness. *IEEE Transactions in Computers*, **C-21**, pp. 948–960, 1972.

[66] Wulf, W. A., Levin, R., and Harbison, S. P. *Hydra/C.mmp: An Experimental Computer System*. McGraw-Hill, June 1981 (ISBN: 0070721203).

[67] Cramer, E. J., Dennis, J. E., Frank, P. D., Lewis, R. M., and Shubin, G. R. Problem Formulation for Multidisciplinary Optimization. *SIAM Journal of Optimization*, **4**, pp. 754–776, 1994.

[68] SPEC. Standard Performance Evaluation Corporation Benchmarks. `http://www.spec.org/`, June 2002.

[69] Sterling, T., Savarese, D., Becker, D. J., Dorband, J. E., Ranawake, U. A., and Packer, C. V. BEOWULF: A Parallel Workstation for Scientific Computation. *Proceedings of the 24th International Conference on Parallel Processing*, pp. 11–14, 1995.

[70] Torvalds, L. The Linux Operating System. `http://www.linux.org/`, 1994.

[71] Modi, Anirudh. COst effective COmputing Array. `http://cocoa.ihpca.psu.edu`, 1998.

[72] Pasko, J. L. DQS-Distributed Queueing System. http://www.scri.fsu.edu/ pasko/dqs.html, January 1996.

[73] Modi, Anirudh. COst effective COmputing Array-2. http://cocoa2.ihpca.psu.edu, 2001.

[74] Forum, M. P. I. MPI: A Message-Passing Interface Standard. (UT-CS-94-230), 1994.

[75] Gropp, W., Lusk, E., and Skjellum, A. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, MA, USA, 1999 (ISBN: 0262571323).

[76] Argonne National Lab. MPI Chameleon. http://www.mcs.anl.gov/mpi/, 2001.

[77] University of Notre Dame. LAM-MPI. http://www.lam-mpi.org/, 2001.

[78] Pacheco, P. S. *Parallel programming with MPI*. Morgan Kaufmann Publishers, 1997 (ISBN: 1558603395).

[79] Modi, A. POSSE: Portabale Object-oriented Scientific Steering Environment. http://posse.sourceforge.net, 2001.

[80] Stallings, W. *Operating Systems: Internals and Design Principles*. Prentice Hall, 4th edition, 2001 (ISBN: 0130319996).

[81] Katz, J., and Plotkin, A. *Low-Speed Aerodynamics*. Cambridge Univeristy Press, 2nd edition, 2001 (ISBN: 0521665523).

[82] Proctor, F. H., and Switzer, G. F. Numerical Simulation of Aircraft Trailing Vortices. *Ninth Conference on Aviation, Range and Aerospace Meteorology,* September 2000.

[83] Stewart, E. C. A Comparison of Airborne Wake Vortex Detection Measurements With Values Predicted From Potential Theory. *NASA TP-3125*, November 1991.

[84] Switzer, G. F. Validation Tests of TASS for Application to 3-D Vortex Simulations. *NASA CR-4756*, October 1996.

[85] Shen, S., Ding, F., Han, J., Lin, Y., Arya, S. P., and Proctor, F. H. Numerical Modeling Studies of Wake Vortices: Real Case Simulations. *AIAA 99-0755*, January 1999.

[86] Greene, G. C. An approximate model of vortex decay in the atmosphere. *Journal of Aircraft*, **23(7)**, pp. 566–573, July 1986.

[87] Pape, D. Bergen Sound Server and Library. `http://www.evl.uic.edu/pape/sw/bergen/`, 2000.

[88] Foley, J. D., van Dam, A., Feiner, S. K., and Hughes, J. F. *Computer Graphics: Principles and Practice in C*. Addison-Wesley, 2nd edition, 1996 (ISBN: 0201848406).

[89] Silicon Graphics Inc. Alias Wavefront/Maya. `http://www.aliaswavefront.com/`, 2001.

[90] H. H. Hubbard, ed. Aeroacoustics of Flight Vehicles, Theory and Practice–Volume 1: Noise Sources. *NASA RP-1258, WRDC TR-90-3052*, August 1991.

[91] Modi, A. Unsteady Separated Flow Simulations using a Cluster of Workstations. Master's Thesis, The Pennsylvania State University, Department of Aerospace Engineering, `http://www.anirudh.net/thesis/thesis.pdf`, May 1999.

[92] Savary, N. A Higher-Order Accurate Finite Volume Method Using Unstructured Grids

for Unsteady Fluid Dynamics. Master's Thesis, The Pennsylvania State University, Department of Aerospace Engineering, December 1999.

[93] Sezer-Uzol, N. High Accuracy Wake and Vortex Simulations using a Hybrid Euler/Discrete Vortex Method. Master's Thesis, The Pennsylvania State University, Department of Aerospace Engineering, May 2001.

[94] Sharma, A. Parallel Methods for Unsteady, Separated Flows and Aerodynamic Noise Prediction. Master's Thesis, The Pennsylvania State University, Department of Aerospace Engineering, August 2001.

[95] Souliez, F. J. Parallel Methods for the Computation of Unsteady Separated Flows around Complex Geometries. Ph.D. Dissertation, The Pennsylvania State University, Department of Aerospace Engineering, August 2002.

[96] Duff, I. S., Erisman, A. M., and Reid, J. K. *Direct methods for sparse matrices*. Oxford University Press, 1986.

[97] Karypis, G. Family of Multilevel Partitioning Algorithms. `http://www-users.cs.umn.edu/~karypis/metis/`, 1997.

[98] Bruner, C. W. S. Parallel Unstructured Maritime Aerodynamics (PUMA) Manual. `http://cocoa.ihpca.psu.edu/puma/manual.pdf`, 1996.

[99] Lorensen, W. E., and Cline, H. E. Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH '87 Proceedings*, **21**, pp. 163–169, July 1987.

[100] Bourke, P. Polygonising a Scalar Field Using Tetrahedrons. `http://astronomy.swin.edu.au/~pbourke/modelling/polytetra/`, June 1997.

[101] Spitzak, B. The Fast Light Toolkit (FLTK). http://www.fltk.org, 1998.

[102] Schroeder, W., Martin, K., and Lorensen, W. E. The Visualization Toolkit (VTK). http://public.kitware.com/VTK/, 1998.

[103] Amtec Engineering. TECPLOT. http://www.amtec.com/Product_pages/tecplot.html, 2001.

Appendix A

C++ Source for Wake-Vortex Program

A.1 Sample Server Input File

A.1.1 vortex.inp

```
! freestream velocity (m/sec),  B.L. thickness (m)
     -10.0   10.0  0.0                    30.0
! ground temperature (in Celsius)
      5.0
! nwings   nwakemax (no. of a/c & max no. of segments in tip vortices)
      3        2200
!   Wing center (x,y,z) (m) wingspan (m)   file        area (m^2) weight (kg)
    200.0   44.8  250.0      29.6   "trajectory1.inp"      10.0  100000.0
   -500.0    0.0   50.0      59.6   "trajectory2.inp"      10.0  76000.0
    569.8  -40.0  250.0     139.6   "trajectory3.inp"      10.0  5800.0
```

A.1.2 trajectory1.inp

```
#dt (sec)       u       v       w       (m/s)
1               250     0       0
5               transition
5               50      30      34
4               transition
1               200     10      -5
4               transition
0               190     1       10      # Final velocity (lasts forever)
```

A.2 Sample Airport Input File

A.2.1 San-Francisco Airport (SFO.inp)

```
/ SFO.apt
/ The San Francisco International airport
/ Slashes denote comments. The file must be written in blocks of
/ comments, information, comments, information, comments...
/ Blank lines can be inserted anywhere.
/ Each file specifies an airport.
/ There can be only one control tower, but any number of runways or
/ terminals.
/ The positive Y direction is considered North. Positive X is East.
/ Units should be in meters, pounds, and degrees.

/ Airport extents information
/ Format: negative x, negative y, positive x, positive y
```

```
-10000
-10000
10000
10000

/ Airport center
/ Format: x, y
0 0

/ Control tower information
/ Format: center x, center y, x radius, y radius, height
0 0
10 10
55

/ Runway information
/ Format: handle, directional type, threshold weight,
/ starting x, starting y, ending x, ending y, width/2,
/ (anchor x, anchor y)
/ anchor x = starting x or ending x
/ anchor y = starting y or ending y
/ The anchor point specifies the start of each runway. This
/ only matters in one-directional runways, so you only need
/ to specify anchor points on one-directional runways.
28L
TWOWAY
710000
-549 915
2250 -701
31

28R
TWOWAY
710000
-726 1200
2409 -610
31

19L
TWOWAY
710000
91 -732
1449 1619
31

19R
TWOWAY
710000
61 -366
1128 1483
31

/ Terminal information
/ Format: center x, center y, x radius, y radius, height, z rotation
```

```
244 976
92 122
20
60

/ Water information
/ Format: number of coordinate pairs in the polyline, x coordinate,
/ y coordinate, x coordinate, y coordinate, x coordinate... (a list of
/ coordinates specifying the polyline representing the coast). Polylines
/ must be closed. If there is no coast, you must put a '0' (for 0
/ coordinate pairs). Put the coordinates in counterclockwise order.
26
-1000 50000
-1000 5000
-500 4000
-400 3600
-1050 3250
-1050 2600
-950 2400
-500 2300
0 2400
950 1800
1000 1900
1300 1700
1450 1900
1650 1650
900 450
2700 -600
2400 -900
675 0
200 -800
1900 -1700
2200 -1900
4000 -1900
4100 -2100
50000 -2100
50000 50000
-1000 50000

4
0 -50000
-50000 50000
-50000 -50000
0 -50000
/ End of airport information
```

A.3 Simulation Code (Server-side)

```
//*************************************************************************
#include <mpi++.h>      // For MPI:: functions
#include <math.h>
#include <string.h>
#include <stdio.h>
```

```
#include <assert.h>
#include <stdlib.h>
#include <unistd.h>
#include "wakevortex.h"
#include "dataservermpi.h"
#include "alloc.h"
#include "main.h"
//***************************************************************************
double ComputeWakeVortexParallel(Vortex *V, int iter);
void ComputeDistanceFromAirport(int i, Vortex *V);
void SendWing(Vortex *V, int wingID, int target);
void RecvWing(Vortex *V, int wingID, int source);
int computeDelay();
void UpdateVortex(Vortex *V);
//***************************************************************************
#define VORTEX_INP_FILE "vortex.inp"
Vortex *vort = NULL;              // For wake-vortex calculations
int delay = 1000; // delay in millisecond
int run_flag = 0;
int time_per_iter = 20;      // in millisecond
double global_dt = 0.0;
int global_nwakemax = 1000;
int numCPUs = 1, myRank = 0;
int exit_flag = 0;
double time_elapsed = 0.0;
double timer_start = 0.0;
int isLagging = 0;
int kn = 5;
int iv_flag = 1;
double max_tracking_distance = 1000.0;
double simtime = 0.0;
int global_iter = 0;
double airport_center[3] = {0.0,0.0,0.0};
//***************************************************************************
void Update_dt(Vortex *V)
      { V->dt = global_dt; }
void Update_kn(Vortex *V)
      { V->kn = kn; }
void Update_ivflag(Vortex *V)
      { V->inducedvelocity_flag = iv_flag; }
void Update_maxdist(Vortex *V)
      { V->max_tracking_distance = max_tracking_distance; }
//***************************************************************************
REGISTER_DATA_BLOCK()      // Register Global Data
{
REGISTER_VARIABLE("run_flag", "rw", run_flag);
REGISTER_VARIABLE("numCPUs", "ro", numCPUs);
REGISTER_VARIABLE("dt", "rw", global_dt);
REGISTER_VARIABLE("iter", "ro", global_iter);
REGISTER_VARIABLE("kneighbors", "rw", kn);
REGISTER_VARIABLE("iv_flag", "rw", iv_flag);
REGISTER_VARIABLE("delay", "rw", delay);
REGISTER_VARIABLE("time_per_iter", "ro", time_per_iter);
REGISTER_VARIABLE("isLagging", "ro", isLagging);
```

```
REGISTER_VARIABLE("simtime", "ro", simtime);
REGISTER_VARIABLE("realtime", "ro", time_elapsed);
REGISTER_VARIABLE("exit_flag", "rw", exit_flag);
REGISTER_VARIABLE("maxdist", "rw", max_tracking_distance);
REGISTER_VARIABLE("airportx", "rw", airport_center[0]);
REGISTER_VARIABLE("airporty", "rw", airport_center[1]);
REGISTER_VARIABLE("airportz", "rw", airport_center[2]);
REGISTER_VARIABLE("nwakemax", "rw", global_nwakemax);
REGISTER_STRUCTURE("vortex", "rw", *vort);
}
//***************************************************************************
#define MASTER (myRank == 0)
//***************************************************************************
int main(int argc, char *argv[])
{
MPI::Init(argc, argv);
numCPUs = MPI::COMM_WORLD.Get_size();
myRank = MPI::COMM_WORLD.Get_rank();

if (MASTER)
      {
       cout << "MPI Version with " << numCPUs << " processor(s)." << endl << flush;
      }

char hostname[200];
gethostname(hostname,200);
cout << "This is processor " << myRank+1 << " of " << numCPUs << " [" <<
      hostname << "]" << endl << flush;

DataServerMPI *S = new DataServerMPI;
int port = 4096;

while (S->Start(port) != CSL_SUCCESS) // Start DataServerMPI
      mySleep(1000*1000);      // Sleep for 1 sec

while (!S->Update<int>(&run_flag)) // Main loop for each set of calculations
      {
       if (S->Update<int>(!S->isRunning()))
            break;      // Exit because DataServerMPI is dead!

       while (!S->Update<int>(&run_flag) && !S->Update<int>(&exit_flag))
            mySleep(1000);      // Wait loop (Sleep for 1 ms)

       if (S->Update<int>(&exit_flag))
            break;      // Exit

       vort = new Vortex;
       myprintf("CPU %d: Allocating vortex\n", myRank+1);

       S->RegisterCallback("vortex", (void(*)(void*)) UpdateVortex, vort);
       S->RegisterCallback("dt", (void(*)(void*)) Update_dt, vort);
       S->RegisterCallback("kneighbors", (void(*)(void*)) Update_kn, vort);
       S->RegisterCallback("iv_flag", (void(*)(void*)) Update_ivflag, vort);
       S->RegisterCallback("maxdist", (void(*)(void*)) Update_maxdist, vort);
```

```
        S->Wait("vortex");
             if (MASTER)
                  vort->ReadInputFile(VORTEX_INP_FILE, global_nwakemax);
             UpdateVortex(vort);
        S->Post("vortex");


        while (IS_ZERO(S->Update<double>(&global_dt)))
             mySleep(1000);      // Sleep for 1 ms

        if (MASTER)
             myprintf("MPI: Starting iterations:\n");

        time_elapsed = 0.0;
        timer_start = myTimer()/1000.0;
        // Calculation loop
        for (int i = 0; i >= 0 && S->Update<int>(&run_flag); i++)
             {
              int time1 = myTimer();
              simtime = vort->curtime;
              global_iter = vort->iter;
              S->Synchronize();       // Synchronize DataServer data

              S->Wait("vortex");
                   ComputeWakeVortexParallel(vort, i+1);
              S->Post("vortex");

              mySleep(10*1000); // Sleep for DataServer Thread
              time_per_iter = myTimer() - time1;
              delay = computeDelay();
              mySleep(delay*1000); // Sleep to keep RT nature
             }

        // Client has exited
        global_dt = 0.0;
        S->Wait("vortex");
             delete vort;
             vort = NULL;
        S->Post("vortex");
        myprintf("CPU %d: Cleaning vortex\n", myRank+1);
       }

MPI::Finalize();
delete S;      // Clean-up DataServerMPI
}
/*****************************************************************************/
double ComputeWakeVortexParallel(Vortex *V, int iter)
{
double time;
int i;
int wingList[100];      // max number of aircrafts per node

// VWING_DELETE or VWING_STOP_TRACKING
```

```
int result = 0;
if (MASTER)
      result = V->CheckForOutOfRangeAircraft(airport_center, VWING_STOP_TRACKING);
BroadcastToSlaves(&result);
if (result)
      {
       for (i = 0; i < V->nwings; i++)
            {
             BroadcastToSlaves(&V->wing[i].ToBeDeleted);
             BroadcastToSlaves(&V->wing[i].OutOfRange);
            }
       BroadcastToSlaves(&V->nToBeDeleted);
       BroadcastToSlaves(&V->ntotal);
       if (MASTER)
            myprintf("Aircraft %d goes out of range!\n", result);
      }

static int flag1, flag2;
if (V->iter == 0) flag1 = flag2 = 1;

if (flag1 && V->curtime > 5.0)
      {
       flag1 = 0;
       V->AddRandomWing("trajectory3.inp", 0.0, 0.0, 300.0);
      }
if (flag2 && V->curtime > 10.0)
      {
       flag2 = 0;
       V->AddRandomWing("trajectory2.inp", 100.0, -30.0, 0.0);
      }

int count = 0;
for (i = V->Begin(); i < V->nwings; i = V->NextWing(i))
      {
       if (i%numCPUs == myRank)
            {
             wingList[count] = i;
             ++count;
            }
      }
MPI::COMM_WORLD.Barrier();

++V->iter;

// All the slaves compute the vortex-wake and send the data to the master
for (i = 0; i < count; i++)      // For every a/c assigned to each CPU/node
      {
       int wingID = wingList[i];

       V->ComputeWake(wingID);      // Main Vortex-Wake Computation

       if (!MASTER)      // Slaves send computed data to Master
            {
             SendWing(V, wingID, 0);
```

```
            }
      }

if (MASTER)      // Master collects computed data from slaves
for (i = V->Begin(); i < V->nwings; i = V->NextWing(i))
      {
       int source = i%numCPUs;
       if (source != 0)      // If source is not the MASTER
            {
             RecvWing(V, i, source);
            }
      }
MPI::COMM_WORLD.Barrier();

V->curtime += V->dt;

time_elapsed = myTimer()/1000.0-timer_start;
char plusminus = '+';
double dt = V->curtime - time_elapsed;
if (dt < 0.0) plusminus = '-';

if (MASTER)
      myprintf("MPI: Iter = %d, Sim Time = %.2f s, Clock Time = %.2f s "
      "(%c%.2fs)\n", iter, V->curtime, time_elapsed, plusminus, fabs(dt));

if (dt < -vort->dt)
      isLagging = 1;
else
      isLagging = 0;

return (V->iter*V->dt);
}
/*****************************************************************************/
// Compute sleep delay required to maintain RT nature
// if computation is faster than needed
int computeDelay()      // in ms
{
double realtime = myTimer()/1000.0-timer_start;
double extra_dt_required = (realtime - vort->curtime)/vort->dt;
int temp = (int) (1000*vort->dt - (1.3+extra_dt_required)*time_per_iter);
int sleepdelay = temp > 0 ? temp : 0;

return sleepdelay;
}
/*****************************************************************************/
// Update Vortex structure among all procs if modified by client
void UpdateVortex(Vortex *V)
{
unsigned char *data;
int size;

if (MASTER)
      {
       cout << "Updating Vortex Structure!" << endl << flush;
```

```
       packStruct(V, &data, &size);
      }
BroadcastToSlaves(&size);

if (!MASTER)
      ALLOC1D(&data, size);

BroadcastToSlaves(data, size);

if (!MASTER)
      unpackStruct(V, data, size);

FREE1D(&data, size);
}
//*************************************************************************
// Pack and send specified wing data from master proc to the specified proc
void SendWing(Vortex *V, int wingID, int target)
{
unsigned char *data;
int totsize;

V->wing[wingID].packWing(&data, &totsize);
int tag = wingID*10;
MPI::COMM_WORLD.Send(&wingID, 1, MPI::INTEGER, target, tag);
++tag;
MPI::COMM_WORLD.Send(&totsize, 1, MPI::INTEGER, target, tag);
++tag;
MPI::COMM_WORLD.Send(data, totsize, MPI::BYTE, target, tag);
delete data;
}
//*************************************************************************
// Master proc receives and unpacks specified wing data from specified proc
void RecvWing(Vortex *V, int wingID, int source)
{
int tag = wingID*10;
int totsize;
int dupwingID = wingID;

MPI::COMM_WORLD.Recv(&dupwingID, 1, MPI::INTEGER, source, tag);
assert(wingID == dupwingID);
++tag;
MPI::COMM_WORLD.Recv(&totsize, 1, MPI::INTEGER, source, tag);
++tag;
unsigned char *data = new unsigned char[totsize];
MPI::COMM_WORLD.Recv(data, totsize, MPI::BYTE, source, tag);
V->wing[wingID].unpackWing(data, totsize);
delete data;
}
//*************************************************************************
```

A.4 Header file for wake-vortex library (vortex.h)

```
//*************************************************************************
```

```
#ifndef WAKEVORTEX_H
#define WAKEVORTEX_H
//*************************************************************************
#include <vector>
#include <iostream>
#include "alloc.h"
#include "wing.h"
#include "airplane.h"
#include "packvortex.h"
using namespace std;
//*************************************************************************
enum { VWING_STOP_TRACKING = 0, VWING_DELETE };
//*************************************************************************
class Vortex
{
private:
 friend ostream &operator << (ostream &Out, Vortex &V);
 void movewings(int wingID);
 void shiftwake(int wingID);
 void movewakes(int wingID);
 void wakevel(int wingID);
 void agewake(int wingID);
 void findUVW(double u0, double v0, double w0, double delta,
   double upmag, double x, double y, double z, double *uu,
   double *vv, double *ww);
 inline void inducedvel(Wing *wing, int i, int j, double *vx,
      double *vy, double *vz, int sign);
 inline void inducedvel(Wing *wing, int i1, int j1, int i2,
      int j2, double *vx, double *vy, double *vz, int sign, int sign2);
 void inducedvel(double x1, double y1, double z1, double x2,
      double y2, double z2, double x, double y, double z,
      double *velx, double *vely, double *velz, double gamma,
      double eps);
 void Initialize();

public:
 //! Total number of aircraft (wings) in the structure
 int nwings;
 //! Total number of aircraft (wings) in the structure
 //! including wings that are not being tracked
 int ntotal;
 //! Number of aircraft marked for deletion
 int nToBeDeleted;
 //! Number of aircraft that contain useful vortex data
 //! but are not actively being tracked
 int nOutOfRange;
 //! Maximum number of vortex-elements to be tracked per aircraft
 int nwakemax;
 //! Current atmospheric temperature (in Celsius)
 double temperature;
 //! Current time-step for simulation (in seconds)
 double dt;
 //! Current atmospheric wind velocity in x-direction (in m/s)
 double u0;
```

```
//! Current atmospheric wind velocity in y-direction (in m/s)
double v0;
//! Current atmospheric wind velocity in z-direction (in m/s)
double w0;
//! Height of atmospheric boundary layer above the ground (in meters)
double delta;
//! Atmospheric fluctuation levels (%)
double upmag;
//! Time required for vortex-strength gamma to decay by 50% (in seconds)
double gtime;
//! Vortex strength threshold below which it is considered harmless
double glimit;
//! Maximum tracking distance for aircraft from the
//! center of the airport (in Kilometers)
double max_tracking_distance;
//! Flag specifying where induced velocity calculation is to be performed
int inducedvelocity_flag;
//! Current number of iterations since the simulation started
int iter;
//! Number of neighboring vortex elements to be used for induced
//! velocity calculation over every vortex element
int kn;
//! Current number of vortex elements being tracked (<= nwakemax)
int nvort;
//! Current time of the simulation since the start (in seconds)
double curtime;
//! Vector containing data for individual aircraft (wing)
vector<Wing> wing;

//! Constructor (with space for n aircraft being allocated)
Vortex(int n = 0);
//! Destructor
~Vortex();
//! Mark all aircraft in the vector for deletion
void MarkAllDeleted();
//! Free all memory (called by destructor)
void CleanUp();

//! Index of the next aircraft in the vector which is i
//! not marked for deletion (but may be out-of-range)
inline int NextWing(int i);
//! Index of the next aircraft in the vector within airport-range
//! which is not marked for deletion
inline int NextInRange(int i);
//! Index of the first active aircraft in the vector
inline int Begin(void);
//! Index of the next aircraft in the vector which is marked for deletion
inline int NextToBeDeleted();
//! Assignment overloading to copy a vortex structure
Vortex &operator= (const Vortex &);
//! Add new aircraft to the vector and
//! allocate memory for it only if "allocate_flag" is true
int AddWing(bool allocate_flag = true);
//! Add new aircraft to the vector without allocating memory
```

```
 int AddWingWithoutAllocation();
 //! Allocate memory for "n" aircraft from the current number of aircraft
 void ReAlloc(int n);
 //! Allocate memory for the "i"th aircraft in the vector
 void Allocate(int i);
 //! Compress vector by removing all aircraft that are marked deleted
 void Compress();
 //! Free memory for "i"th aircraft in the vector and mark it for deletion
 void deleteWing(int i);
 //! Undo the effect of the "deleteWing(i)"
 void undeleteWing(int i);
 //! Stop tracking aircraft "i" (mark it out-of-range)
 void stopTrackingWing(int i);
 //! Check for out-of-range aircraft
 int CheckForOutOfRangeAircraft(double airport_center[3], int option);
 //! Read intial inputs for the simulation from "inpfile"
 void ReadInputFile(char *inpfile, int nwake_max);
 //! Read intial inputs for the simulation from "inpfile"
 //! Compute the wake-vortex for all aircraft for the next iteration
 double ComputeWake();
 //! Compute the wake-vortex for the specified aircraft for the
 //! next iteration
 int ComputeWake(int wingID);
 //! Write TECPLOT output for the specified iteration
 void WriteTecplotFile(int iter);
 //! Add a new aircraft at location (x0,y0,z0) with trajectory information
 //! from file "trajfile" and specified wing-span, area and weight
 int AddNewWing(char *trajfile, char *name, char *type,
      double x0, double y0, double z0, double span,
      double area, double weight);
 //! Add a randomly selection aircraft from database at location (x0,y0,z0)
 int AddRandomWing(char *trajfile, double x0, double y0, double z0);
 //! Copy data from the current vortex structure to "dest"
 int Copy(Vortex *dest);
};
//***************************************************************************
#endif // WAKEVORTEX_H
/***************************************************************************/
```

A.5 Header file for wake-vortex library (wing.h)

```
//***************************************************************************
#ifndef _WING_H
#define _WING_H
//***************************************************************************
#include <vector>
#include <iostream>
#include "alloc.h"
using namespace std;
//***************************************************************************
typedef struct
{
 //! Maximum time specified in trajectory file
```

```
 double time_max;
 //! Local tracking element
 int firsttime;
 //! Is the aircraft trajectory in transition from one velocity to
 //! another?
 int isTransition;
 //! Elapsed time since the trajectory has been executed
 double t1;
 //! Time period for specified velocity (uu2, vv2, ww2)
 double dt1;
 //! Initial trajectory velocity (uu1, vv1, ww1)
 double uu1;
 double vv1;
 double ww1;
 double uu2;
 double vv2;
 double ww2;
} Trajectory;

class Wing
{
private:
 friend ostream &operator << (ostream &Out, Wing &V);
 void getWingTrajectoryUVW(double t, double *u, double *v, double *w);
 void GetWingCoordinates(double u, double v, double w,
        double x_mid, double y_mid, double z_mid, double span,
        double *x1, double *y1, double *z1,
        double *x2, double *y2, double *z2);
 double getDensity(double temperature);

public:
 //! vortex-strength (gamma) at wing-tip
 double gwing;
 //! size of vortex core (meters)
 double ewing;
 //! Initial location of wing-tips (meters)
 double xwing_i[2][3];
 //! Initial velocity of wing-tips (i.e., aircraft) (m/s)
 double vwing_i[2][3];
 //! Identification label for aircraft
 char   Name[30];
 //! Type of aircraft (e.g., Boeing 777, Airbus 320, etc.)
 char   AircraftType[30];
 //! Name of file containing trajectory information
 char   trajectoryname[100];
 //! core size (meters) of array of tip vortices
 double *ewake;
 //! File-pointer for trajector file "trajectoryname"
 FILE   *trajectory;

 // Out
 //! Current location of wing-tips (meters)
 double xwing[2][3];
 //! Current velocity of wing-tips (i.e., aircraft) (m/s)
```

```
double vwing[2][3];
//! Vortex strength (gamma) of each segment of tip vortices
double *gwake;
//! Age of each segment of tip vortices (seconds)
double *twake;
//! Coordinates of points along tip vortices (meters)
double (*xwake)[2][3];
//! Velocity of points along tip vortices (m/s)
double (*vwake)[2][3];

//! Time elapsed since vortex-elements for aircraft being tracked (sec)
double timeElapsed;
//! Number of iterations
int iter;
//! Wing-span (meters)
double span;
//! Weight of aircraft (kg)
double weight;
//! Area of wing (meters^2)
double area;
//! Out-of-range flag
int OutOfRange;
//! Deletion flag
int ToBeDeleted;
//! Current number of vortex elements being tracked (<= nwakemax)
int nvort;
//! ID for the aircraft
int ID;
//! Distance from the center of the airport (Kilometers)
double DistFromAirport;

//! Trajectory associated with the aircraft
Trajectory traj;

//! Constructor
Wing();
//! Destructor
~Wing();
//! Initialization routine (resets variables)
void Initialize();
//! Assignment operator overloading for copying
Wing &operator= (const Wing &);
//! Read trajectory info for time "t" and return velocity (u,v,w)
void readTrajectory(double t, double *u, double *v, double *w);
//! Setup the aircraft at location (x0,y0,z0) with supplied parameters
void Setup(char *inpfile, char *name, char *type, double x0,
     double y0, double z0, double wingspan, double wingarea,
     double wingweight, double temperature, int nwakemax);
//! Allocate space for vortex-elements (max elements: nwakemax)
void Allocate(int nwakemax);
//! Free space allocated for vortex-elements
void DeAllocate(int nwakemax);
//! Pack wing for efficient communication from one processor to another
int packWing(unsigned char **dataptr, int *totsize);
```

```
 //! Unpack pack after communication completed
 int unpackWing(unsigned char *data, int totsize);
 //! Vortex strength (gamma) for current element at wing-tip
 double getGwing(double temperature);
 //! Current speed of the aircraft
 inline double Speed();
 //! Current height of the aircraft above the ground
 inline double Height();
};
//***************************************************************************
#endif // _WING_H
/***************************************************************************/
```

Appendix B

C++ Source for the CFD Client Application

B.1 Sample input file for PUMA

```
Gamma      R         rhoRef    Vref      muRef
1.4        287.04    1.3813    320.46    1.831e-5
rho_inf     u_inf      v_inf      w_inf      p_inf         fsMult    xyzMult
1.3813      58.1152    0.0        0.0        101325.0      1.0       1.0
mu_inf   T_Suth  PrLam   PrTurb
1.831e-5 110.0   0.72    0.9
numIts      maxMinutes      wrRestart      wrResid      relResid      absResid
150         50000           5a             1            1.0e-6        1.0e-16
CFLconst      CFLslope
26            15
localStepping  spatial_order    inviscid_flux     integ_scheme    limiter
1               2                "roe"             "ssor"          "none"
innerIters    innerTol      omega     stages    K        commScheme
50            1.0e-8        1.0       2         5.0e+2   "delta q"
gridName                  restartFrom      restartTo        residName
"grids/apache.sg.gps"     ""               "apache.rst"     "apache.rsd"
implicitBCs    viscousModel      artDiss      VIS-2   VIS-4   M_CR
1               "inviscid"        0            1.00    1.0     0.85
Num_surfaces
7
surface BC_type rho u v w p
1       8
2       5
3       5
4       5
5       5
6       5
7       5
```

B.2 Additions to PUMA (Server-side)

```
/*************************************************************************/
#include "puma.h"
#include "global.h"
#include "dataservermpi.h"
#include "steering.h"
/*************************************************************************/
// Global variables for Computational Steering
int Ncells, Nghost, Nrecv;
int numCPUs, iter = 0, numIters = 0;
int Ntotal = 0;
double CFL, CFLconst, CFLslope;
double(*q)[5];
```

```
double GV_freestream[5];
char GV_fluxscheme[20];
char GV_intscheme[20];
char GV_gridfile[100], GV_inpfile[100], GV_residfile[100];
int NglobalCells = 0;
int NglobalFaces = 0;
double GV_logResid = 0.0;
double GV_MFlops = 0.0;
double GV_MFlopsPerProc = 0.0;
double GV_time_per_iter = 200.0;
int GV_querymode = 0;
NumericalFlux fscheme;
IntMethod tscheme;
double GV_min[12];
double GV_max[12];
double GV_mach, GV_temperature;
int GV_getrstflag = 0;
double *GV_rst = NULL;
int GV_num_rst = 0;
Surface isoContTri;
Surface surfaceTri;
double(*GV_isoContTri)[3][12] = NULL;
double(*GV_surfaceTri)[3][12] = NULL;
int GV_num_surfTri = 0;
int GV_num_isoContTri = 0;
int GV_isovar = -1;
int GV_restartflag = 0;
double GV_isovalue;
/*****************************************************************************/
REGISTER_DATA_BLOCK()
{
  REGISTER_VARIABLE("flopcount", "ro", dFlops);
  REGISTER_VARIABLE("querymode", "rw", GV_querymode);
  REGISTER_VARIABLE("restart", "rw", GV_restartflag);
  REGISTER_VARIABLE("iter", "ro", iter);
  REGISTER_VARIABLE("numIters", "rw", numIters);
  REGISTER_VARIABLE("simTime", "ro", simTime);
  REGISTER_VARIABLE("rho", "rw", GV_freestream[0]);
  REGISTER_VARIABLE("u", "rw", GV_freestream[1]);
  REGISTER_VARIABLE("v", "rw", GV_freestream[2]);
  REGISTER_VARIABLE("w", "rw", GV_freestream[3]);
  REGISTER_VARIABLE("P", "rw", GV_freestream[4]);
  REGISTER_VARIABLE("M", "ro", GV_mach);
  REGISTER_VARIABLE("T", "ro", GV_temperature);
  REGISTER_1D_ARRAY("int_scheme", "rw", GV_intscheme);
  REGISTER_1D_ARRAY("flux_scheme", "rw", GV_fluxscheme);
  REGISTER_VARIABLE("CFLconst", "rw", CFLconst);
  REGISTER_VARIABLE("CFLslope", "rw", CFLslope);
  REGISTER_VARIABLE("CFL", "ro", CFL);
  REGISTER_VARIABLE("logResid", "ro", GV_logResid);
  REGISTER_VARIABLE("Ncells", "ro", NglobalCells);
  REGISTER_VARIABLE("Nlocalcells", "ro", Ncells);
  REGISTER_VARIABLE("Nlocalghost", "ro", Nghost);
  REGISTER_VARIABLE("Nlocalrecv", "ro", Nrecv);
```

```
  REGISTER_VARIABLE("Nfaces", "ro", NglobalFaces);
  REGISTER_1D_ARRAY("inpfile", "ro", GV_inpfile);
  REGISTER_1D_ARRAY("gridfile", "ro", GV_gridfile);
  REGISTER_1D_ARRAY("residfile", "ro", GV_residfile);
  REGISTER_VARIABLE("MFlops", "ro", GV_MFlops);
  REGISTER_VARIABLE("MFlopsPerProc", "ro", GV_MFlopsPerProc);
  REGISTER_VARIABLE("TimePerIter", "ro", GV_time_per_iter);
  REGISTER_VARIABLE("isoVar", "rw", GV_isovar);
  REGISTER_VARIABLE("isoValue", "rw", GV_isovalue);
  REGISTER_VARIABLE("min_rho", "ro", GV_min[0]);
  REGISTER_VARIABLE("max_rho", "ro", GV_max[0]);
  REGISTER_VARIABLE("min_u", "ro", GV_min[1]);
  REGISTER_VARIABLE("max_u", "ro", GV_max[1]);
  REGISTER_VARIABLE("min_v", "ro", GV_min[2]);
  REGISTER_VARIABLE("max_v", "ro", GV_max[2]);
  REGISTER_VARIABLE("min_w", "ro", GV_min[3]);
  REGISTER_VARIABLE("max_w", "ro", GV_max[3]);
  REGISTER_VARIABLE("min_Cp", "ro", GV_min[4]);
  REGISTER_VARIABLE("max_Cp", "ro", GV_max[4]);
  REGISTER_VARIABLE("min_M", "ro", GV_min[5]);
  REGISTER_VARIABLE("max_M", "ro", GV_max[5]);
  REGISTER_VARIABLE("min_T", "ro", GV_min[6]);
  REGISTER_VARIABLE("max_T", "ro", GV_max[6]);
  REGISTER_VARIABLE("min_P", "ro", GV_min[7]);
  REGISTER_VARIABLE("max_P", "ro", GV_max[7]);
  REGISTER_VARIABLE("min_Entropy", "ro", GV_min[8]);
  REGISTER_VARIABLE("max_Entropy", "ro", GV_max[8]);
  REGISTER_VARIABLE("min_X", "ro", GV_min[9]);
  REGISTER_VARIABLE("max_X", "ro", GV_max[9]);
  REGISTER_VARIABLE("min_Y", "ro", GV_min[10]);
  REGISTER_VARIABLE("max_Y", "ro", GV_max[10]);
  REGISTER_VARIABLE("min_Z", "ro", GV_min[11]);
  REGISTER_VARIABLE("max_Z", "ro", GV_max[11]);
  REGISTER_VARIABLE("getrstflag", "rw", GV_getrstflag);
  REGISTER_DYNAMIC_1D_ARRAY("q", "ro", q, Ncells);
  REGISTER_DYNAMIC_1D_ARRAY("isoCont", "ro", GV_isoContTri, GV_num_isoContTri);
  REGISTER_DYNAMIC_1D_ARRAY("SurfTri", "ro", GV_surfaceTri, GV_num_surfTri);
  REGISTER_DYNAMIC_1D_ARRAY("rstdata", "ro", GV_rst, GV_num_rst);
  REGISTER_STRUCTURE("isoContSurf", "ro", isoContTri);
  REGISTER_STRUCTURE("SurfTriSurf", "ro", surfaceTri);
}
/*****************************************************************************/
void
UpdateCallback(char *key)
{
  myprintf("UpdateCallback: Keyword \"%s\" modified!!\n", key);
}
/*****************************************************************************/
void
UpdateIntScheme()
{
  myprintf("UpdateIntScheme() called!\n");
  tscheme = getIntScheme(GV_intscheme);
}
```

```
/*****************************************************************************/
void
UpdateFluxScheme()
{
  myprintf("UpdateFluxScheme() called!\n");
  fscheme = getFScheme(GV_fluxscheme);
}
/*****************************************************************************/
```

B.3 Graphical User Interface (Client-side)

```
//*****************************************************************************
#include "PumaClientWindow.h"
//*****************************************************************************
PumaClientWindow::PumaClientWindow(int w, int h, const char *l,
                           const char *serverDefault,
                           int portDefault, const char *layoutFile):
ClientWindow(w, h, l, serverDefault, portDefault, 95)
{
  window->user_data(this);
  window->end();

  int width;
  int pos_h = 25;
  int pos_w = 5;
  int start_h = 35;
  int height = 25;
  int gap = 5;

  width = 110;
  variable = new variableChoice(pos_w, start_h, width, pos_h, this);
  pos_w += width + gap;
  variable->down_box(FL_BORDER_BOX);
  for(int i = 0; i < NumVars(); i++)
    variable->add(GetIsoVarName(i), 0,(Fl_Callback *) variableUpdate);
  variable->value(GetIsoVarID("M"));
  window->add(variable);

  width = 240;
  value = new Fl_Value_Slider(pos_w, start_h, width, pos_h);
  pos_w += width + gap;
  value->type(5);
  window->add(value);

  width = 70;
  getIt = new Fl_Button(pos_w, start_h, width, pos_h, "Get It!");
  pos_w += width + gap;
  getIt->callback((Fl_Callback *) getItClick);
  window->add(getIt);

  width = 140;
  queryMode = new Fl_Button(pos_w, start_h, width, pos_h, "Query Mode: OFF");
  pos_w += width + gap;
```

```
  queryMode->callback((Fl_Callback *) queryModeClick);
  window->add(queryMode);

  width = 80;
  getGrid = new Fl_Button(pos_w, start_h, width, pos_h, "Get Grid");
  pos_w += width + gap;
  getGrid->callback((Fl_Callback *) getGridClick);
  window->add(getGrid);

  start_h += height + gap;
  pos_w = 5;

  width = 200;
  pos_w += 70;
  layoutInp = new Fl_Input(pos_w, start_h, width, pos_h, "Layout file:");
  pos_w += width + gap;
  layoutInp->value(layoutFile);
  window->add(layoutInp);

  width = 80;
  getTecIso = new Fl_Button(pos_w, start_h, width, pos_h, "Contour");
  pos_w += width + gap;
  getTecIso->callback((Fl_Callback *) getTecIsoClick);
  window->add(getTecIso);

  width = 80;
  getTecSurf = new Fl_Button(pos_w, start_h, width, pos_h, "Surface");
  pos_w += width + gap;
  getTecSurf->callback((Fl_Callback *) getTecSurfClick);
  window->add(getTecSurf);

  width = 80;
  getTecGrid = new Fl_Button(pos_w, start_h, width, pos_h, "Grid");
  pos_w += width + gap;
  getTecGrid->callback((Fl_Callback *) getTecGridClick);
  window->add(getTecGrid);

  ConnectAction(false);

  Fl::add_idle((void(*)(void *)) IdleCalls, this);
}

//****************************************************************************
PumaClientWindow::~PumaClientWindow()
{
  delete variable;
  delete value;
  delete getIt;
  delete queryMode;
  delete getGrid;
  delete layoutInp;
  delete getTecIso;
  delete getTecSurf;
  delete getTecGrid;
```

```
}
//***************************************************************************
void PumaClientWindow::getItClick_i(Fl_Button * o, void *v)
{
  getIsoContour(variable->value(), value->round(value->value()));
  RenderIsoSurface(iso);
}
//***************************************************************************
void PumaClientWindow::getTecIsoClick_i(Fl_Button * o, void *v)
{
  char datafile[] = "contour.dat";
  getIsoContour(variable->value(), value->round(value->value()));
  WriteTrianglesToFile(iso, datafile);
  char cmd[200];
  sprintf(cmd, "tecplot %s %s &", layoutInp->value(), datafile);
  system(cmd);
}
//***************************************************************************
void PumaClientWindow::getTecSurfClick_i(Fl_Button * o, void *v)
{
  getSurfaceGrid();
  char datafile[] = "surface.dat";
//WriteTrianglesToFile(surface, datafile);
  char cmd[200];
  sprintf(cmd, "tecplot %s %s &", layoutInp->value(), datafile);
  system(cmd);
}
//***************************************************************************
void PumaClientWindow::getTecGridClick_i(Fl_Button * o, void *v)
{
  char datafile[] = "grid.dat";
  get3DTecplotFile(datafile);
  char cmd[200];
  sprintf(cmd, "tecplot %s %s &", layoutInp->value(), datafile);
  system(cmd);
}
//***************************************************************************
void PumaClientWindow::queryModeClick_i(Fl_Button * o, void *v)
{
  if (strcmp(queryMode->label(), "Query Mode: OFF") == 0)
    {
      SetQueryMode(true);
      queryMode->label("Query Mode: ON");
    }
  else
    {
      SetQueryMode(false);
      queryMode->label("Query Mode: OFF");
    }
}
//***************************************************************************
void PumaClientWindow::variableUpdate_i(Fl_Button * o, void *v)
{
  double minv = getMin(variable->value());
```

```
  double maxv = getMax(variable->value());
  value->minimum(minv);
  value->maximum(maxv);
  value->value(0.5 *(minv + maxv));
  clientTable->take_focus();
  variable->set_changed();
}
//****************************************************************************
void PumaClientWindow::getGridClick_i(Fl_Button * o, void *v)
{
  get3DTecplotFile("tecgrid.out");
}
//****************************************************************************
void PumaClientWindow::ConnectAction(bool connected)
{
  if (connected)
    {
      variable->activate();
      value->activate();
      getIt->activate();
      queryMode->activate();
      getGrid->activate();
      layoutInp->activate();
      getTecIso->activate();
      getTecSurf->activate();
      getTecGrid->activate();
      PumaClient::Connect(clientTable->GetClient());
      variableUpdate_i(NULL, NULL);
      if (QueryMode())
      queryMode->label("Query Mode: ON");
      else
      queryMode->label("Query Mode: OFF");
    }
  else
    {
      variable->deactivate();
      value->deactivate();
      getIt->deactivate();
      queryMode->deactivate();
      getGrid->deactivate();
      layoutInp->deactivate();
      getTecIso->deactivate();
      getTecSurf->deactivate();
      getTecGrid->deactivate();
      PumaClient::Disconnect();
    }
}
//****************************************************************************
void PumaClientWindow::Idle2()
{
  // enter critical section ++++++++++++++++++++++++++++++++++++++++++++
  if (!PreventRefreshSem.isBusy())
    {
      if (clientTable->GetClient())
```

```
      {
        clientTable->update();
        if (QueryMode())
          queryMode->label("Query Mode: ON");
        else
          queryMode->label("Query Mode: OFF");
      }
    }
  // exit critical section ---------------------------------------
}
//***************************************************************************
int variableChoice::handle(int event)
{
  switch(event)
    {
    case FL_PUSH:
      if (window)
      window->PreventRefreshSem.Wait();
      Fl_Choice::handle(event);
      break;
    case FL_LEAVE:
      if (window)
      window->PreventRefreshSem.Post();
      break;
    }
}
//***************************************************************************
void PumaClientWindow::automate(char *var, int iter, const char *server, int port)
{
  cout << "-------------------" << endl << flush;
  cout << "Starting Automation" << endl;
  cout << "-------------------" << endl << flush;
  Connect((char *) server, port);
  int isoVar = GetIsoVarID(var);
  double minv = getMin(isoVar);
  double maxv = getMax(isoVar);
  cout << "Range for \"" << var << "\" = {" << minv << "," << maxv << "}" <<
    endl << flush;
  double dv =(maxv - minv) * 1.0 /(iter - 1);
  for(int i = 0; i < iter; i++)
    getIsoContour(isoVar, minv + dv * i);
  Disconnect();
  cout << "-----------------" << endl << flush;
  cout << "Ending Automation" << endl;
  cout << "-----------------" << endl << flush;
}
//***************************************************************************
```

Appendix C

C++ Source for POSSE Test Programs

C.1 Single Client Performance Test

C.1.1 Server-side program

```
//*****************************************************************************
#include <math.h>
#include <string.h>
#include <iomanip.h>
#include "alloc.h"
#include "misc.h"
#include "dataserver.h"
//*****************************************************************************
// Define all data that has to be registered for being served as "global"
int run_flag = 1;
double *dyn1D;
int n1;
int dim1;
//*****************************************************************************
// Callback function to change the dimension of 1D array to "dim1".
void UpdateDimension()
        { FREE1D(&dyn1D, n1); n1 = dim1; ALLOC1D(&dyn1D, n1); }
//*****************************************************************************
// Register all the variables/data here along with their handles/keywords
REGISTER_DATA_BLOCK()
{
REGISTER_VARIABLE("run_flag", "rw", run_flag);
REGISTER_VARIABLE("dimension", "rw", dim1);
REGISTER_DYNAMIC_1D_ARRAY("dyn1D", "ro", dyn1D, n1);
}
//*****************************************************************************
int main(int argc, char *argv[])
{
int port = 4096;
if (argc > 1)
      port = atoi(argv[1]);

DataServer *Server = new DataServer;

n1 = 1;
ALLOC1D(&dyn1D, n1);

// Fill the Array with some values
for (int i = 0; i < n1; i++)
      dyn1D[i] = i*1.0;
```

```
// Start the server
while (Server->Start(port) != CSL_SUCCESS) { }
Server->RegisterCallback("dimension", (void(*)(void*)) UpdateDimension, NULL);
int start_time = myTimer();

// Sleep while run_flag is true
while (run_flag) mySleep(10*1000);            // Sleep for 10 ms

// Stop the server
Server->Stop();

double bytes_sent = Server->BytesSent();
delete Server;
int uptime = myTimer()-start_time;      // in ms

cout << "Effective network bandwidth = " << setprecision(4)
      << 8.0*bytes_sent/(1000.0*uptime) << " Mbps." << endl;

FREE1D(&dyn1D, n1);
}
/*****************************************************************************/
```

C.1.2 Client-side program

```
//*****************************************************************************
#include <math.h>
#include <iomanip.h>
#include "dataclient.h"
//*****************************************************************************
int parseArg(int argc, char *argv[], char *tag);
//*****************************************************************************
int main(int argc, char *argv[])
{
char servername[100];
int port = 4096;
if (argc < 2)
      {
       cout << "Usage: client <server-name>" << endl;
       exit(-1);
      }
strcpy(servername, argv[1]);

if (int index = parseArg(argc, argv, "-port"))
      port = atoi(argv[index]);

DataClient *client = new DataClient;

// Connect to the server
if (client->Connect(servername, port) != CSL_SUCCESS)
      {
       cout << "Connection to " << servername << ":" << port << " failed!" << endl;
       delete client;
       exit(-1);
      }
```

```
int start = 0;
int end = 1000000;
int interval = 1000;      // in ms

if (int index = parseArg(argc, argv, "-start"))
      start = atoi(argv[index]);
if (int index = parseArg(argc, argv, "-end"))
      end = atoi(argv[index]);
if (int index = parseArg(argc, argv, "-interval"))
      interval = atoi(argv[index]);

// Loop to receive Array with different sizes ranging from "start"
// to "end" elements with "interval" interval.
for (int i = start; i <= end; i += interval)
      {
       double *dyn1D;
       // Set dimension of the 1D array
       client->SendVariable("dimension", i);
       // Receive dimension for verification
       int n1 = client->getArrayDim("dyn1D", 1);
       // Make sure that dimensions match
       assert(n1 == i);

       // Allocate appropriate memory and receive array and
       // time the communication.
       ALLOC1D(&dyn1D, n1);
       int starttime = myTimer();
       client->RecvArray1D("dyn1D", dyn1D);
       int commtime = myTimer()-starttime;
       FREE1D(&dyn1D, n1);

       cout << n1*8 << " " << commtime << " " << setprecision(4)
            << 8.0*8.0*n1/(1000.0*commtime) <<  endl << flush;
      }
// Print statistics
cout << "Total Bytes sent = " << setprecision(5) << client->BytesSent()
      << " at the rate of " << client->SendRate() << " Mbps." << endl;
cout << "Total Bytes recd = " << setprecision(5) << client->BytesRecd()
      << " at the rate of " << client->RecvRate() << " Mbps." << endl;

delete client;
}
/*****************************************************************************/
int parseArg(int argc, char *argv[], char *tag)
{
for (int i = 1; i < argc; i++)
      if (strstr(argv[i], tag) == argv[i] && strlen(tag) == strlen(argv[i]))
                return i+1;
return 0;
}
/*****************************************************************************/
```

C.2 Multiple Client Performance Test

C.2.1 Server-side program

```
//*************************************************************************
#include <math.h>
#include <string.h>
#include <iomanip.h>
#include "alloc.h"
#include "misc.h"
#include "dataserver.h"
//*************************************************************************
// Define all data that has to be registered for being served as "global"
int run_flag = 1;
double ****dyn4D;
int n1, n2, n3, n4;
//*************************************************************************
// Register all the variables/data here along with their handles/keywords
REGISTER_DATA_BLOCK()
{
REGISTER_VARIABLE("run_flag", "rw", run_flag);
REGISTER_DYNAMIC_4D_ARRAY("dyn4D", "rw", dyn4D, n1, n2, n3, n4);
}
//*************************************************************************
int main(int argc, char *argv[])
{
int port = 4096;
if (argc > 1)
      port = atoi(argv[1]);

DataServer *Server = new DataServer;

n1 = 8;
n2 = 66;
n3 = 55;
n4 = 7;

// Allocate memory to the 4D array
ALLOC4D(&dyn4D, n1, n2, n3, n4);

// Fill the array with some values
for (int i = 0; i < n1; i++)
for (int j = 0; j < n2; j++)
for (int k = 0; k < n3; k++)
for (int l = 0; l < n4; l++)
      dyn4D[i][j][k][l] = (i+1)*100+(j+1)*10+(k+1)*1+(l+1)/10.0;

// Start the server
while (Server->Start(port) != CSL_SUCCESS) { }
int start_time = myTimer();

// Sleep while run_flag is true
while (run_flag) mySleep(10*1000);            // Sleep for 10 ms
```

```
// Stop the server
Server->Stop();

double bytes_sent = Server->BytesSent();
delete Server;
int uptime = myTimer()-start_time;      // in ms

cout << "Effective network bandwidth = " << setprecision(4)
      << 8.0*bytes_sent/(1000.0*uptime) << " Mbps." << endl;

FREE4D(&dyn4D, n1, n2, n3, n4);
}
/*****************************************************************************/
```

C.2.2 Client-side program

```
//*****************************************************************************
#include <math.h>
#include <iomanip.h>
#include "dataclient.h"
//*****************************************************************************
int main(int argc, char *argv[])
{
char servername[100];
int port = 4096;
if (argc < 2)
      {
       cout << "Usage: client <server-name> [port-#]" << endl;
       exit(-1);
      }
strcpy(servername, argv[1]);
if (argc > 2)
      port = atoi(argv[2]);

DataClient *client = new DataClient;

// Connect to the server
if (client->Connect(servername, port) != CSL_SUCCESS)
      {
       cout << "Connection to " << servername << ":" << port << " failed!" << endl;
       delete client;
       exit(-1);
      }

double ****dyn4D;
// Get the dimensions of the 4D array and allocate memory for it
int n1 = client->getArrayDim("dyn4D", 1);
int n2 = client->getArrayDim("dyn4D", 2);
int n3 = client->getArrayDim("dyn4D", 3);
int n4 = client->getArrayDim("dyn4D", 4);
ALLOC4D(&dyn4D, n1, n2, n3, n4);

int numrequests = 1;
int interval = 500;      // in ms
```

```
// Request data for "dyn4D" array numrequests times with a wait
// of "interval" ms between consecutive requests
for (int i = 0; i < numrequests; i++)
      {
       int oldtime = myTimer();
       client->RecvArray4D("dyn4D", dyn4D);
       int commtime = myTimer()-oldtime;
       if (numrequests > 1)
            {
              int delay = interval-commtime;
              if (delay > 0) mySleep(delay);
            }
      }

// Free the memory for thr 4D array
FREE4D(&dyn4D, n1, n2, n3, n4);

delete client;
}
/*****************************************************************************/
```

C.3 SMP vs. UP Performance Test

C.3.1 Server-side program

```
//*****************************************************************************
#include <math.h>
#include <string.h>
#include <iomanip.h>
#include "alloc.h"
#include "misc.h"
#include "dataserver.h"
//*****************************************************************************
// Define all data that has to be registered for being served as "global"
int run_flag = 0;
double ****dyn4D;
int n1, n2, n3, n4;
//*****************************************************************************
// Register all the variables/data here along with their handles/keywords
REGISTER_DATA_BLOCK()
{
REGISTER_VARIABLE("run_flag", "rw", run_flag);
REGISTER_DYNAMIC_4D_ARRAY("dyn4D", "rw", dyn4D, n1, n2, n3, n4);
}
//*****************************************************************************
int main(int argc, char *argv[])
{
int port = 4096;
if (argc > 1)
      port = atoi(argv[1]);

DataServer *Server = new DataServer;
```

```
n1 = 8;
n2 = 66;
n3 = 55;
n4 = 7;

// Allocate memory to the 4D array
ALLOC4D(&dyn4D, n1, n2, n3, n4);

// Fill the array with some values
for (int i = 0; i < n1; i++)
for (int j = 0; j < n2; j++)
for (int k = 0; k < n3; k++)
for (int l = 0; l < n4; l++)
      dyn4D[i][j][k][l] = (i+1)*100+(j+1)*10+(k+1)*1+(l+1)/10.0;

// Start the server
while (Server->Start(port) != POSSE_SUCCESS) { }

int count = 0;
while (!run_flag) mySleep(1000); // Sleep while run_flag is false

int start_time = myTimer();
while (run_flag)
      {
       ++count;
       for (int i = 0; i < n1; i++)
       for (int j = 0; j < n2; j++)
       for (int k = 0; k < n3; k++)
       for (int l = 0; l < n4; l++)
            dyn4D[i][j][k][l] = sin(dyn4D[i][j][k][l]*1.0)*
                              (i+1)*100+(j+1)*10+(k+1)*1+(l+1)/10.0;
      }

// Stop the server
Server->Stop();

double bytes_sent = Server->BytesSent();
delete Server;
int uptime = myTimer()-start_time;      // in ms

cout << "Number of computations = " << count << endl;
cout << "Uptime = " << uptime << " ms" << endl;
cout << "Effective network bandwidth = " << setprecision(4)
      << 8.0*bytes_sent/(1000.0*uptime) << " Mbps." << endl;

FREE4D(&dyn4D, n1, n2, n3, n4);
}
/*****************************************************************************/
```

C.3.2 Client-side script

```
//*****************************************************************************
#!/bin/sh
if [ $# != 2 ]
```

```
then
        name=‘basename $0 .base‘
        echo Usage: ${name} \<server-name\> \<number-of-simultaneous-clients\>
      exit
fi
server=$1
n=$2

# Start the computation
printf "run_flag = 1 \n quit \n" | key_client ${server}

# Spawn "n" client requests simultaneously
while test $n -gt 0; do
      n=‘echo $n | awk ’{print $1-1}’‘
      ./client ${server} &
done

# Wait till all client requests are completed
flag=1
while test $flag -gt 0; do
      flag=‘ps auwx | grep client | grep -v grep | wc -l‘
done

# Stop the server
printf "run_flag = 0 \n quit \n" | key_client ${server}
```

VITA

Anirudh Modi was born in Ranchi, India on 3rd April, 1975. He did his schooling with Shrimati Sulochanadevi Singhania High School in Thane, Maharashtra. He graduated from high school at the Ruparel College, Bombay, India in 1993. He obtained his Bachelor of Technology in Aerospace Engineering from the Indian Institute of Technology, Bombay in April 1997. After a short summer stint as a Project Engineer at his alma mater, he enrolled in the M.S. program at the Department of Aerospace Engineering at the Pennsylvania State University, USA in August 1997. There he was a recipient of the Institute for High Performance Computing Applications (IHPCA) Fellowship and the Rotorcraft Center of Excellence (RCOE) Fellowship. He received his Master of Science degree in Aerospace Engineering with emphasis on Computational Fluid Dynamics in May 1999 and then went on to enroll in the Ph.D. program at the Department of Computer Science and Engineering at the Pennsylvania State University. Anirudh's research interests are in the field of software engineering with special interests in computational steering and high performance computing. He is a member of the AIAA and the IEEE. He is married to Sushmita Poddar.