Real-Time Visualization of Wake-Vortex Simulations using Computational Steering and Beowulf Clusters *

Anirudh Modi[1], Lyle N. Long[2], and
Paul E. Plassmann[3]

[1] Ph.D. Candidate, Department of Computer Science and Engineering,
Pennsylvania State University, University Park, PA 16802
`anirudh@anirudh.net`
`http://www.anirudh.net/phd/`
[2] Professor, Department of Aerospace Engineering,
Pennsylvania State University, University Park, PA 16802
`lnl@psu.edu`
`http://www.personal.psu.edu/lnl/`
[3] Associate Professor, Department of Computer Science and Engineering,
Pennsylvania State University, University Park, PA 16802
`plassman@cse.psu.edu`
`http://www.cse.psu.edu/~plassman/`

Abstract. In this paper, we present the design and implementation of POSSE, a new, lightweight computational steering system based on a client/server programming model. We demonstrate the effectiveness of this software system by illustrating its use for a visualization client designed for a particularly demanding real-time application—wake-vortex simulations for multiple aircraft running on a parallel Beowulf cluster. We describe how POSSE is implemented as an object-oriented, class-based software library and illustrate its ease of use from the perspective of both the server and client codes. We discuss how POSSE handles the issue of data coherency of distributed data structures, data transfer between different hardware representations, and a number of other implementation issues. Finally, we consider how this approach could be used to augment AVOSS (an air traffic control system currently being developed by the FAA) to significantly increase airport utilization while reducing the risks of accidents.

1 Introduction

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

* This work was supported by NSF grants EIA–9977526 and ACI–9908057, DOE grant DG-FG02-99ER25373, and the Alfred P. Sloan Foundation.

perceptually realistic virtual environment (VE) applications. The combination of the two 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 wait 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]. The ultimate goal of the work presented in this paper is to create a wake-vortex hazard avoidance system by realistically simulating an airport with real-time visualization of the predicted 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, we utilize an easy-to-use, yet powerful computational steering library to deal with the complexities of real-time wake-vortex visualization.

To enable such a complex simulation, we will require a computational steering system. 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* from Georgia Tech [4], *SCIRun* from Scientific Computing and Imaging research group at University of Utah [5], ALICE Memory Snooper from Argonne National Laboratory [6], *VASE* (Visualization and Application Steering) from University of Illinois [7], *CUMULVS* from Oak Ridge National Laboratory [8], *CSE* (Computational Steering Environment) from the Center for Mathematics and Computer Science in Amsterdam [9], and *Virtue* from University of Illinois at Urbana-Champaign [10]. 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. To be productive with these systems by using them in existing scientific codes is not an easy task, and may take a significant amount of time, especially for the large number of computational scientists with no formal education in computer science or software systems.

To address these problems, we have developed a new lightweight computational steering system based on a client/server programming model. In this paper, we first discuss computational steering in section 2, then the details of wake-vortex simulations in section 3, and finally some experimental results in section 4.

2 Computational Steering

While running a complex parallel program on a high-performance computing system, one often experiences several major difficulties in observing computed results. Usually, the simulation severely limits the interaction with the program during the execution and makes the visualization and monitoring slow and cumbersome (if at all possible), especially if it needs to be carried out on a different system (say a specialized graphics workstation for visualization).

For our simulations, it is very important for the predictions by the wake-vortex code to be known in real-time by the Air-Traffic Control (ATC) in order for it to take appropriate action. This activity 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 conditions at the airport may keep changing and both the number and the trajectories of the aircraft can change as they take-off and land. Thus, there is a need to modify the simulation based on these factors by manipulating some key characteristics of its algorithm. This activity is referred to as "steering," which is defined as the modification of a program's behavior during its execution.

Software tools which support these activities are called computational steering environments. These environments typically operate in three phases: instrumentation, monitoring, and steering. Instrumentation is the phase where the application code is modified to add monitoring functionality. The 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 does not need to be stopped and restarted.

Our steering software, the *Portable Object-oriented Scientific Steering Environment* (POSSE) [11], is very general in nature and is based on a simple client/server model. It uses an approach similar to Falcon [4] (an on-line monitoring and steering toolkit developed at Georgia Tech) and ALICE Memory Snooper [6] (an application programming interface designed to help in writing computational steering, monitoring and debugging tools developed at Argonne National Lab). Falcon was one of the first systems to use the idea of threads and shared memory to serve registered data efficiently. POSSE consists of a steering server on the target machine that performs steering, and a steering client that provides the user interface and control facilities remotely. The steering server is created as a separate execution thread of the application to which local monitors forward only those "registered" data that are of interest to steering activities. A steering client receives the application run-time information from the application, displays the information to the user, accepts steering commands from the user, and enacts changes that affect the application's execution. Communication between a steering client and server are done via UNIX sockets and threading is done using POSIX threads. POSSE has been completely written in C++, using several of C++'s advanced object-oriented features, making it fast and powerful, while hiding most of the complexities from the user. Fig. 1 shows a schematic view of how POSSE can be used. An on-going scientific simulation is running on a remote Beowulf computing cluster. Any number of number of remote clients can query/steer registered data from the simulation from the DataServer thread. Two clients are shown, a visualization client and a GUI client that provides a simple user interface to all registered simulation data.

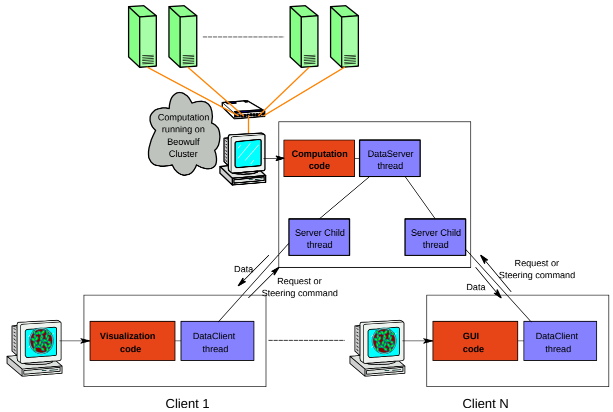

Fig. 1. A schematic view of POSSE

POSSE is designed to be extremely lightweight, portable (runs on all Win32 and POSIX-compliant Unix platforms) 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. It can also be easily incorporated into parallel simulations based on the Message Passing Interface (MPI) [12] library. The biggest enhancement of POSSE over existing steering systems is that it is equally powerful, yet extremely easy to use, making augmentation of any existing C/C++ simulation code possible in a matter of hours. It makes extensive use of C++ classes, templates and polymorphism to keep the user Application Programming Interface (API) elegant and simple to use. Fig. 2 and Fig. 3 illustrate a simple, yet complete, POSSE client/server program in C++. As seen in the figures, 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. 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 does not need to know anything about the internals of threads, sockets or networking in order to use POSSE effectively. POSSE also allows 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. Among other applications, we have successfully used POSSE to enhance our existing parallel Computational Fluid Dynamics (CFD) code to perform visualizations of large-scale flow simulations [13].

3 Wake-Vortex Simulation

One of the main problems facing the ATC today is the “wake-vortex” hazard. Just as a moving boat or a ship leaves behind a wake in the water, an aircraft leaves behind a wake in the air. These wake-vortex pairs are invisible to the naked eye and stretch for several miles behind the aircraft and may last for several minutes. The aircraft wake is

```
#include "dataserver.h"

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;
}
```

Fig. 2. A simple, complete POSSE server application written in C++

```
#include "dataclient.h"

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

  if (client->Connect("cocoa.ihpca.psu.edu", 4096) != POSSE_SUCCESS) // Connect to DataServer
       {
        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;
}
```

Fig. 3. A simple, complete POSSE client application written in C++

generated from the wings of the aircraft and consists of two counter-rotating swirling rolls of air which are termed “wake-vortices”. In Fig. 4, we show a photograph depicting the smoke flow visualization of wake-vortices generated by a Boeing 727. It is to be noted that these are not contrails (i.e., condensation trail left behind by the jet exhausts). The strength of these vortices depends on several factors, including weight, size and velocity of the aircraft. The strength increases with the weight of the aircraft. The life of the vortex depends 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 [14]. The rapid swirling of air in a vortex can have a potentially fatal effect on the stability of a following aircraft. Currently, the only way to deal with this problem is the use of extremely conservative empirical spacing between consecutive take-offs and landings from the same runway, which has been laid down by the International Civil Aviation Organization (ICAO) and FAA. 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. But, despite these spacings being extremely conservative, they are not always able to prevent accidents owing to the several unknowns involved, primarily the exact location and strength of the vortices. The US Air Flight 427 (Boeing 737) disaster which occurred on September 8, 1997 near Pittsburgh is attributed to this phenomenon, wherein the aircraft encountered the wake-vortices of a preceding Boeing 727 [15]. The more recent Airbus crash 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.

Fig. 4. B-727 vortex study photo (Courtesy: NASA Dryden Flight Research Center)

To tackle this problem of reduced airport capacity which is a direct fallout of these overly conservative spacing regulations, and to address the concerns of the aircraft in circumstances when these regulations fail to meet the safety requirements, 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) [16]. AVOSS, in spite of performing a rigorous simulation of the wake-vortices, does not implement any system for their visualization. It only provides the ATC with the aircraft spacing time for each aircraft which is all the current ATC systems can handle. Thus, at present, it is unable to provide alternate trajectories for the take-off and landing of aircraft.

This work attempts to fill in the gaps left by AVOSS by creating a wake-vortex hazard avoidance system by realistically simulating an airport with real-time 3D visualization of the predicted wake-vortices generated by the moving aircraft. Aircraft will be able to adjust their flight trajectory based on the information obtained from the visualization system to avoid the wake-vortices and operate more safely and efficiently.

3.1 Wake-Vortex Theory

For the wake-vortex simulations described in this paper, we use potential theory to predict the strength of the wake-vortex elements [17]. The circulation generated by the lift is assumed to be contained in two vortices of opposite signs trailing from the tips of the wing. 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 [18]. The magnitude of the circulation of each vortex is approximately

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

where L_g and V_g are the lift and the velocity of the aircraft, respectively. References [19, 20, 17] deal with more details on the numerical simulation of these aircraft vortices.

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. The decay of the vortex strength is based on a simplified version of the model suggested by Greene [21]:

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

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).

3.2 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. As such, a parallel solution for the same is required to maintain a real-time response of the simulation. For example, a typical metropolitan airport in the US is extremely busy with several take-offs and landings occurring every few minutes. Dallas/Fort Worth, the country'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 computations 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. Although this number can be reduced by as much as a factor of 100 by making simplifying assumptions for the induced velocity calculations (wherein, we say that any vortex element is only affected by a fixed number of neighboring elements, say k, rather than all the other elements), this still amounts to a large computation considering that each induced velocity calculation consists of $200-300$ floating point operations. This takes our net computational requirement to approximately $5-8$ Gigaflops, necessitating the need of a parallel computer. Hence, our wake-vortex prediction code, based on the potential flow theory described above, is written in C++ with MPI for parallelization.

Pseudocode for the simulation is given in Fig. 5. Each vortex element has two main properties associated with it, strength and position. The initial strength (Γ) is calculated based on the potential flow theory and the initial position is based on the position of the aircraft. The strength then decays as a function of time and the prevailing weather conditions, and the position changes due to the velocity induced by neighboring vortex elements and the prevailing wind velocity. Fig. 6 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 actual simulation code enhanced using POSSE. 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 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 we get an almost real-time solution to this problem with tolerable lag no more than the time-step Δt in our simulation. The first processor 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. It is also 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 $((v_j \in V) \neq v_i$ & $|j-i| \leq k)$
        $v_i.inducedvel \leftarrow v_i.inducedvel + \text{InducedVelocity}(v_i, v_j)$
      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)
      $v_i.strength \leftarrow v_i.strength - \text{DecayFunction}(\Delta t, \text{Weather Conditions})$
      If ($v_i.strength < threshold$) **then**
        $V \leftarrow V - v_i$
      Endif
    Endforeach
    $t \leftarrow t + \Delta t$
  Endwhile
Endforeach

Fig. 5. Algorithm for Wake-Vortex prediction

For the real-time simulation, the parallel wake-vortex code has been augmented with POSSE, so that it can remotely run on our in-house 40 processor PIII-800 Mhz *Beowulf Cluster*[1] [22], the COst-effective COmputing Array 2 (COCOA-2) [23]. For a steering client, a visualization tool has been written in C++ using the OpenGL API for graphics and CAVELib [24] API for stereo-graphics and user interaction. The monitoring code runs as a separate thread in the visualization client retrieving new vortex data whenever the simulation on the remote cluster updates. A screenshot of the program depicting the Wake-Vortex simulation for a single aircraft is shown in Fig. 7. Another screenshot (Fig. 8) shows several aircraft flying above the San Francisco International airport (SFO). The colors represent the relative strength of the vortices with red being maximum and blue being minimum. The Reconfigurable Automatic Virtual Environment (RAVE) from FakeSpace Systems [25] driven by an HP Visualize J-class workstation is then used as the display device.

[1] A cluster of commodity personal computers running the LINUX operating system.

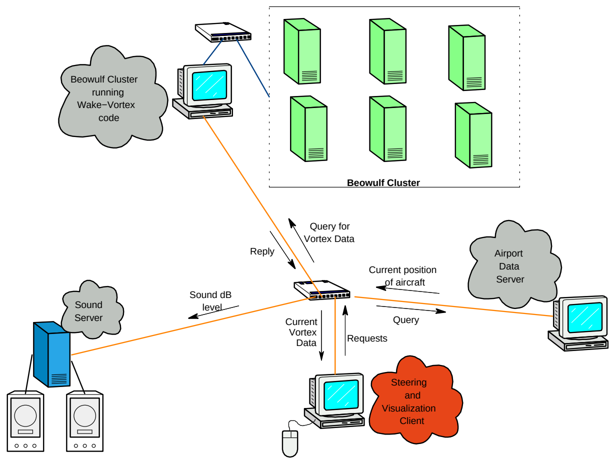

Fig. 6. Wake-Vortex Simulation System

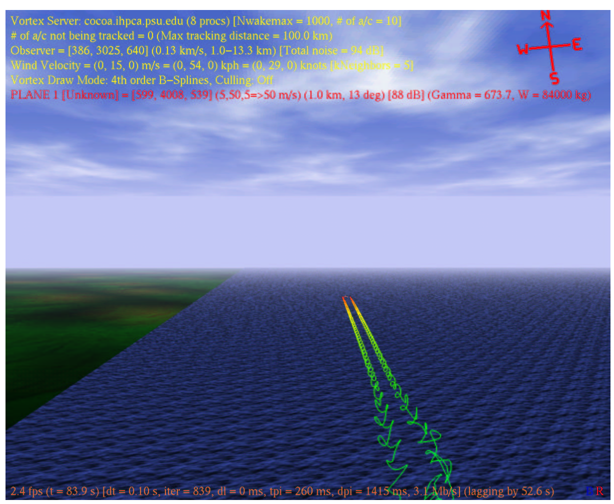

Fig. 7. Screenshot of the Wake-Vortex simulation for a single aircraft from a visualization client

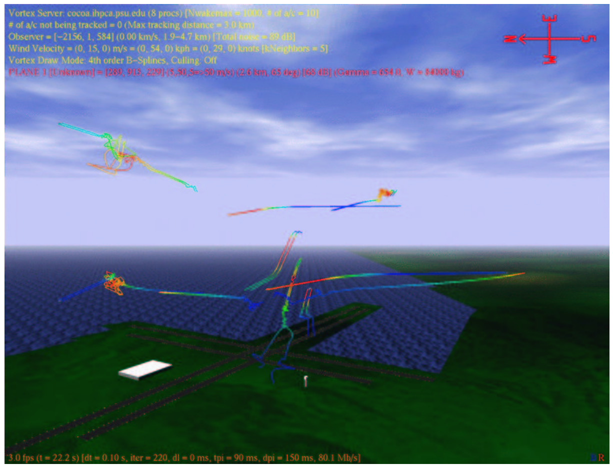

Fig. 8. Screenshot of the Wake-Vortex simulation for several aircraft flying above the San Francisco (SFO) airport

4 Experimental Results

POSSE has been extensively tested using various platforms for stability and performance. Tests demonstrating both the single and multiple client performance for POSSE are discussed here.

4.1 Single Client Performance

Fig. 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 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.

4.2 Multiple Client Performance

Fig. 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). 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.

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 aircrafts each having $2,000$ elements tracked, this amounts to 1.12 MB of data. From Fig. 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 aircrafts) 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 strength, a more sophisticated weather model like the one used in AVOSS will definitely improve the accuracy of the simulations.

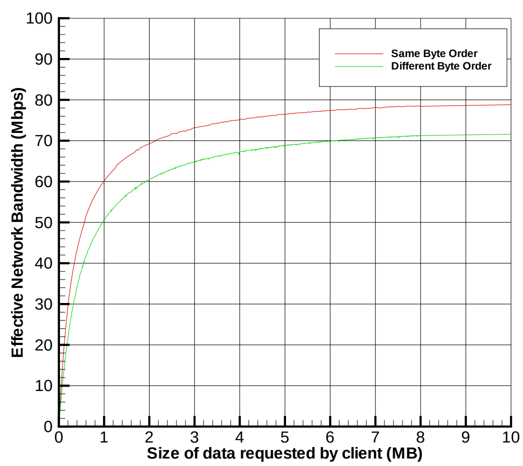

Fig. 9. Effective Network Bandwidth vs. Size of data requested by client

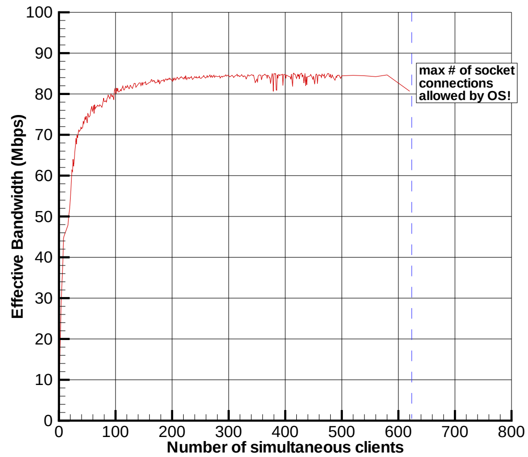

Fig. 10. Effective Network Bandwidth vs. Number of simultaneous clients

5 Conclusions

The coupling of computational steering to our parallel simulation makes the real-time visualization of the wake-vortex simulations possible. It 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. Our steering system, POSSE, has proven to be a very powerful, yet easy to use software with a high rate of acceptance and approval in our research group. If scientists are given an easy to use software system with a mild learning curve, they will use it. At a more basic level, this ability to interact and visualize a complex solution as it unfolds and the real-time nature of the computational steering system opens a whole new dimension to the scientists for interacting with their simulations.

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. T. S. Perry. In Search of the Future of Air Traffic Control. *IEEE Spectrum*, 34(8):18–35, August 1997.
3. Bernhard Reitinger. 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://eos.gup.uni-linz.ac.at/thesis/thesis.pdf, January 2001.
4. W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, J. Vetter, and N. Mallavarupu. Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs. *Proceedings of*

the Fifth Symposium on the Frontiers of Massively Parallel Computation, pages 433–429, February 1995.

5. S.G. Parker, M. Miller, C.D. Hansen, and C.R. Johnson. An Integrated Problem Solving Environment: The SCIRun Computational Steering System. *IEEE Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, 7:147–156, 1998.
6. I. Ba, C. Malon, and B. Smith. Design of the ALICE Memory Snooper. http://www.mcs.anl.gov/ams, 1999.
7. D. Jablonowski, J. Bruner, B. Bliss, and R. Haber. VASE : The Visualization and Application Steering Environment. *Proceedings of Supercomputing '93*, pages 560–569, 1993.
8. G. A. Geist, II, James Arthur Kohl, and Philip M. Papadopoulos. CUMULVS: Providing Fault Tolerance, Visualization, and Steering of Parallel Applications. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):224–235, Fall 1997.
9. Robert van Liere and Jarke J. van Wijk. CSE : A Modular Architecture for Computational Steering. In M. Göbel, J. David, P. Slavik, and J. J. van Wijk, editors, *Virtual Environments and Scientific Visualization '96*, pages 257–266. Springer-Verlag Wien, 1996.
10. E. Shaffer, D.A. Reed, S. Whitmore, and B. Schaeffer. Virtue: Performance Visualization of Parallel and Distributed Applications. *IEEE Computer*, 32(12):44–51, December 1999.
11. Anirudh Modi. POSSE: Portable Object-oriented Scientific Steering Environment. http://posse.sourceforge.net, 2001.
12. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. (UT-CS-94-230), 1994.
13. A. Modi, N. Sezer, L. N. Long, and P. E. Plassmann. Scalable Computational Steering System for Visualization of Large Scale CFD Simulations. *AIAA 2002-2750*, June 2002.
14. Barnes W. McCormick. Aircraft Wakes: A Survey of the Problem. *Keystone Presentation at FAA Symposium on Aviation Turbulence*, March 1971.
15. Airdisaster.com. Investigation: USAir Flight 427. http://www.airdisaster.com/investigations/us427/usair427.shtml, 1997.
16. D. A. Hinton. Aircraft Vortex Spacing System (AVOSS) Conceptual Design. *NASA TM-110184*, August 1995.
17. Fred H. Proctor and George F. Switzer. Numerical Simulation of Aircraft Trailing Vortices. *Ninth Conference on Aviation, Range and Aerospace Meteorology*, September 2000.
18. Eric C. Stewart. A Comparison of Airborne Wake Vortex Detection Measurements With Values Predicted From Potential Theory. *NASA TP-3125*, November 1991.
19. G. F. Switzer. Validation Tests of TASS for Application to 3-D Vortex Simulations. *NASA CR-4756*, October 1996.
20. S. Shen, F. Ding, J. Han, Y. Lin, S. P. Arya, and F. H. Proctor. Numerical Modeling Studies of Wake Vortices: Real Case Simulations. *AIAA 99-0755*, January 1999.
21. G. C. Greene. An approximate model of vortex decay in the atmosphere. *Journal of Aircraft*, 23(7):566–573, July 1986.
22. T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A Parallel Workstation for Scientific Computation. *Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14, 1995.
23. Anirudh Modi. COst effective COmputing Array-2. http://cocoa2.ihpca.psu.edu, 2001.
24. VRCO. CAVELib Users Manual. http://www.vrco.com/CAVE_USER/caveuser_program.html, 2001.
25. CAVE. Fake Space Systems. http://www.fakespacesystems.com, 2001.