pV3: A Distributed System for Large-Scale Unsteady CFD Visualization

Robert Haimes*

Department of Aeronautics and Astronautics
Massachusetts Institute of Technology, Cambridge, MA 02139

Abstract

This paper presents a distributed architecture for the visualization of large steady and transient data-sets. Because one of the goals is to not have to store away the 10s to 100s of Gigabytes necessary to perform a post-processing animation of the data, this system must be able to run concurrently with the task producing the data. This requirement forces portions of the visualization system to run on the same hardware where the data is generated.

What follows is a discussion of current computer architectures used for CFD calculations and how these systems affect the model selected for the visualization software. Also data and data movement considerations must be an integral part of any project that requires the distribution of a compute task.

1 Introduction

Scientific visualization is an area that has grown greatly in size and importance in the last five years. This growth has been driven by a number of factors. One is the change in the types of calculations that are being done by Computational Fluid Dynamics (CFD) researchers. Ten years ago, the standard CFD calculation would be of steady, 2D inviscid flow, which could be easily understood by straightforward 2D computer graphics (e.g. monochrome contour plots, line plots of surface pressure, etc.). Now, 3D, unsteady, and/or viscous calculations are common, and these are very much more difficult to imagine or visualize in our minds, let alone on a two-dimensional computer screen. Therefore, the development of tools for the visualization of such flows is very important if we are to properly understand the results of our calculations.

A related driving force is the huge increase in computer power over the last decade. A numerical simulation can be split into three phases: pre-processing (defining the geometry and constructing the computational grid), computation and post-processing (trying to understand the computational results). Ten years ago, most of the time and cost was in the computation phase. Due to faster computing speeds and much more complex geometries and flows, most of the time is now spent in the pre- and post-processing phases for steady-state problems.

The final most important factor is the emergence in the last five years of reasonably priced computer hardware (and associated software) for visualization.

Five years ago our group at MIT was involved in the development of flow algorithms for unstructured 'finite-element' grids, using triangles and quadrilaterals in two dimensions, and hexahedra in three dimensions. Also we worked on both unsteady flow calculations and grid adaptation in two dimensions. Other people at MIT were working on grid adaptation and unstructured flow solvers in three dimensions using tetrahedra, and were beginning to do unsteady three dimensional calculations. We realized that there was no single workstation visualization software available at that time to view these results. Therefore the research into visualization and the development of **Visual2**[1] and **Visual3**[1][2], visualization packages for two and three dimensions, respectively. From the start, this effort was designed to handle unstructured grids with a variety of different cell types, and to view steady or unsteady data.

Currently, the computer horsepower of high-performance workstations is not up to the task of pseudo-real-time unsteady visualization. This may always be true. In general, the more the compute capabilities, the larger the CFD problem that is attacked. With the use of super computers and Massively Parallel Processors (MPPs) more data can be generated than can be digested by traditional workstation architectures. Clearly some other compute model is required to give the CFD investigator a high degree of interactivity for viewing unsteady data-sets.

2 Design Goals

The following are the design goals for the parallel version of **Visual3** (**pV3**):

- High Performance

  Take advantage of the proper hardware to get the best performance out of the entire compute arena. It is viewed as foolish to have overall compute performance of gigaflops and only the ability to do 2D graphics. **pV3** requires graphics hardware so that scene rendering will not be a limitation and the data presented to the investigator is of high quality and timely.

*Principal Research Engineer, Member AIAA

- Interactive

  This goal is of utmost importance. Interactivity is a requirement that facilitates interrogation of the data. If batch processing of a 'movie' is all that is done, then all that can be viewed and understood is what has been 'story-boarded' *a priori*.

  The goal of any scientific visualization package should be to allow the assimilation of the vast amounts of data produced by the models and solvers in order to better understand the underlying physics. The ultimate goal, with this new knowledge, is to affect design and produce a better car, aircraft, gas-turbine engine, etc. This can only be done by poking and probing into the data to interrogate areas of interest. Feature extraction helps to isolate these areas but it is still necessary to 'dig-in' in order to close the design loop.

  'Movies' are important for presentations and communication to others on the design team, therefore they should be made after the interrogation. 'Movies' should never be the sole visualization method.

- No esoteric requirements

  pV3 can only be a useful tool if it is readily available and is easily used! Like **Visual3**, **pV3** is distributed free of charge and does not require any special or proprietary software. It is targeted for the compute environments that are currently performing large-scale unsteady CFD applications.

  The hardware requirements for compute also target what is currently being used. The graphics hardware is a little more restrictive, but equipment like expensive frame buffers on super computers or MPPs is not required. Head mounted displays and data gloves are also not needed. The same techniques used in **Visual3** are implemented to allow pointing in 3-space.

- Co-processing

  An important part of **pV3** is the ability to visualize the data as the solver or model progresses in time. It is also designed to allow the solver to run as independently as possible. If the solution procedure takes hours to days, **pV3** can 'plug-into' the calculation, allow viewing of the data as it changes, then can 'unplug' with the worst side-effect being the temporary allocation of memory and a possible load imbalance.

  This also has the advantage that all the data need not be stored and then played back to get a continuous and smooth viewing of the data. The data required can be 10s to 100s of gigabytes [3] putting a huge storage equipment (and financial) burden on the compute facility. If the solver is fast enough (on the order of an iteration a second or less), then only a coarse sampling of data in time need be placed on disk. When one wishes to view the solution at a finer time interval, the solution scheme and visualization are restarted from the appropriate saved snap-shot and re-run.

  In many cases, the time required to read the complete volume of a saved time frame from disk is now longer than the compute time for a set of iterations. And, the data bandwidth from disks have not kept up with the advances in compute speeds so this spread will probably widen. Therefore, depending on data stored in a traditional disk farm or data server can have a negative impact on interactivity. Co-processing with a cluster of workstations can help even with post-processing applications. If the data is also distributed across the cluster, each machine reads its portion of the solution giving an aggregate disk I/O rate equal to the sum of the clusters' rate.

- **Visual3** functionality

  pV3 provides the same kind of functionality as **Visual3** with the same suite of tools and probes[4]. The data represented to the investigator (the 3D, 2D and 1D windows with cursor mapping) is the same. Also the same Graphical User Interface (GUI) is used.

  This means that, from the user's perspective, yet another scientific visualization system need not be learned.

- **Visual3**-like programming

  Another goal for **pV3** is that the programming be very **Visual3**-like. For the desired flexibility and the merging of the visualization with the solver, some programming is required. The coding is simple; like **Visual3**, all that is required of the programmer is the knowledge of the solver's data. Learning the details of the underlying graphics, data extraction, and movement (for the visualization) is not needed. If the data is distributed in a cluster of machines, **pV3** deals with this, resulting in few complications to the user.

  In some cases, the calls or routines provided are identical to the **Visual3** programming interface. For someone familiar with **Visual3**, programming of **pV3** requires little new knowledge.

3 Classification of Unsteady Cases

For some of the discussions in this paper is it useful to classify the types of unsteadiness. Listed below are the three types based on what is changing from iteration to iteration:

- Data Unsteady

  In this type of application the grid structure and position are fixed in time. The data defined at the nodes (both scalar and vector) changes with each time step. An example of a 'Data Unsteady' case is where a varying boundary condition causes changes in the domain. This can reflect some disturbance entering the volume.

- Grid Unsteady

  'Grid Unsteady' cases are 'Data Unsteady' plus the grid coordinates associated with each node are also allowed to move with each snapshot. An example of this is stator/rotor interaction in turbomachinery. The stator and rotor grids are separate, with the rotor grid sliding past the stator grid. In this case the stator portion is actually 'Data Unsteady' and the rotor grid moves by it in a radial manner.

- Structure Unsteady

  If the number of nodes, number of cells or cell connectivity changes from iteration to iteration the case is 'Structure Unsteady'. An example of this mode is store separation. At first there is little or no grid between the aircraft and the store. As the store peals away from the aircraft grid is added or changed until the store has left the domain. This class is not supported by **Visual3**.

4 Computing Environment

The current compute environment that most researchers are using for the calculation of 3D unsteady CFD results is a super computer class machine (including some MPPs) and clusters of workstations acting as a single MPP by concurrently working on the same task. The cluster of Reduced Instruction-Set Computers (RISC) is a recent advent based on the low-cost and high-performance that the workstation vendors are providing. The cluster, with the proper software, can act as a MIMD (Multiple Instruction/Multiple Data) machine. For the same price/performance reasons, the MPP vendors are also moving away from SIMD (Single Instruction/Multiple Data) architectures to collections of RISC chips connected with high bandwidth interconnects. It appears that SIMD machines may not have a future in general scientific computing. There are parts of many models that require synchronization but not calculation over the entire domain, such as boundary conditions, which stall these machines. For these reasons, SIMD machines are not be targeted by **pV3**.

Currently, **Visual3** can perform unsteady visualization. The limitation is that the entire volume of data needs to be transferred to the workstation for each time frame. This can easily overwhelm any network. A small 3D viscous CFD case may require 500,000 nodes. Assuming the case is 'Data Unsteady' and only the one scalar field and one vector field variable are transferred, 8 Megabytes of data is required. Assuming maximum Ethernet speeds (and nothing else on the network) that results in one frame every 7 seconds for the data transfer alone. **Visual3** clearly cannot meet all the goals.

Once the decision is made to move parts of the visualization task off the workstation, the question arises: What gets distributed?

- Low-level graphics

  By far the simplest method is to distribute the graphics. PEX (the PHIGS Extension to X) is designed to run across a network and there is a distributed form of GL (SGI's Graphics Language). This mode of operation has two problems in meeting the design goals. The first is that these network based graphics systems assume a one client/one server model. This restriction makes the visualization from a cluster of machines impossible.

  Secondly, performance can be degraded in comparison to the client and server running on the same workstation. The distributed mode works best if the entire set of data required to generate the scene is sent to the server at initialization. If the data is static (or only moves based on local transformations) little additional network traffic is required for interaction with the scene. This works best for CAD systems and is what PHIGS was designed for. But it is not the best model for scientific volumetric visualization where surfaces are being generated and then destroyed as the session progresses. The problem becomes acute for the visualization of unsteady systems where no surface data may be retained between scenes presented to the investigator.

  Running **Visual3** across a network (using PEXlib or dGL) is always much slower than single workstation visualization. The network, using this method of distribution, is the barrier to interactivity.

- Scene generation

  Another possible method to distribute the visualization task is to generate parts of the resultant scene on each compute node (or in the super computer). This can be done in one of two ways (only the 3D window will be discussed):

  - Window partitioning

    This method breaks up the 3D window into a set of sub-windows, each processed by a single compute node. This requires all graphical objects to be broadcast to all compute nodes doing the rendering (or some intelligent preprocessing is required). Then when all sub-windows are complete, the resultant pixmaps must be transmitted to the workstation with the screen for final assembly and display.

  - Scene merging

    To avoid the broadcast of all graphical objects, each compute node processes its objects. When this is complete, the entire window (pixmap and Z-buffer) must be transmitted to one workstation where the scenes are merged pixel by pixel. The pixel value is selected based on the Z value closest to the viewer. This produces a fixed amount of network traffic, but the amount is large. For a 512x512 window each processing node needs to move 1.5 megabytes using 24 bit color and a 24 bit Z-buffer.

Using the super computer model, both of the above methods are the same, but the compute resources are not well utilized. It is well known that specialized graphics hardware (found in workstations) does a more efficient job of 3D drawing than general purpose systems. That is why Cray Research has developed MPGS[5], that renders on Silicon Graphics Inc (SGI) workstations.

For the cluster of workstations, a set of benchmarks indicates that it would take between 10 and 20 high-end RISC workstations dedicated to rendering alone to equal the power of the VGX graphics systems from SGI (and assuming no network data needs to be transmitted). These machines would be better used on a compute task.

- Lower dimensional objects

  Most visualization algorithms are embarrassing parallel. This is because at some level most operations are performed on an element and these cells are (or can be easily) distributed throughout the compute arena. Little work need be done to take the current tools and distribute the compute portion to where the data resides. An example is iso-surface generation on a SIMD machine by Hanson and Hinker[6]. The ***REAL*** problem is how to minimally describe the result of the tool and move it to where it is needed. A fine discussion of this concept, 'extracts', can be found in a treatise by Globus[7]. 'Extracts' are ***NOT*** graphics but are objects that contain enough information to generate the graphic. A graphic object is inflexible. It contains vertex information, such as coordinate position and color, and polygonal data (and optionally normals) so the graphic can be included in the final scene. 'Extracts' contain the coordinate data but color is not assigned. Scalar and even vector data can be assigned to each vertex. The associated cell information can be used to generate the graphic and also used for further probing.

  'Extracts' allow the type of freedom required for a **Visual3**-like system. For example, first an iso-surface is generated. Now, to render it with the value of another scalar field, only the color mapping need be changed. This does not require re-computing the iso-surface, but while the graphic is generated (during rendering), the color associated with each vertex is changed. The only new data required to change the color is the scalar field values at each vertex of the object.

  Clearly, producing 'extracts' where the volume of data exists and only passing the results at a reduced dimensionality does not add any additional message passing burden. Only the data required to draw the result of the tool and/or probe is moved over the network.

The obvious choice is 'extracts'. In fact the data passed at the packet level of **pV3** is 'sub-extracts'. Collections of triangles, quadrilaterals, 3D coordinates, mesh, scalar and vector data are each passed as a separate item. In this way, only the objects that are currently needed for scene generation are transmitted. And if the vector or scalar vertex values change from one iteration to the next, only they need to be retransmitted. It becomes the responsibility of the workstation dealing with the graphics to put the scene together. This also frees up the compute cluster from all graphics. Rotations, panning, scrolling and zooming (any changing of the viewing transformation matrix) is handled only by that workstation.

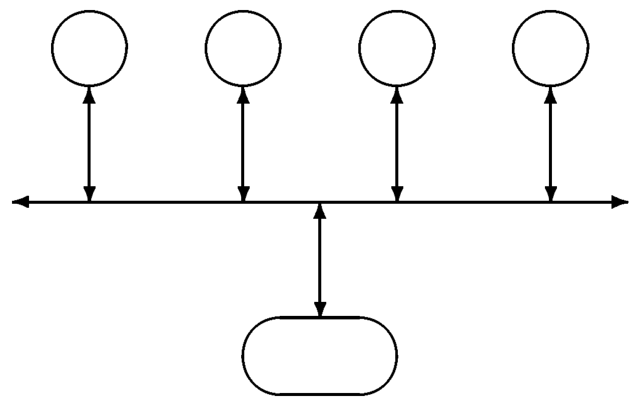

Figure 1: Network based Architecture of pV3

5 The Architecture of pV3

pV3 requires splitting up the unsteady visualization task to allow execution across a network of workstation(s) and compute engines. In this computing model (see Figure 1), the network will probably always be the bottleneck so much of the effort involves techniques to reduce the size of the data transferred between machines.

The software used for the movement of data across the network is **PVM** [8] from Oak Ridge National Laboratory. **PVM** (parallel virtual machine) is public domain software that provides the mechanisms required to transform a (heterogeneous) network of machines to one parallel computer. **PVM** provides all the required 'hooks' including efficient data transfers, broadcasting, message passing, synchronization and the ability to target certain machines to specific tasks. **PVM** is available directly from Convex and IBM for their cluster offerings and will also be available from the traditional MPP vendors.

It should be noted that a network based distributed system is not a perfect solution. There will always be latency issues and synchronization problems, even on a lightly loaded network. Error recovery from a node that prematurely terminates is a situation that is difficult to properly handle.

The architecture of **pV3** can be best understood by examining the two distribution extremes:

- Two Machine Visualization

  This is a subset of the multi-machine model.

  The computational portion of the visualization task runs on a single computer. Co-processing unsteady applications require a super computer class machine (or an MPP such as a CM-5 or KSR-1 that can be viewed from the network as a single processor). In this example the distribution model is similar to the commercial product MPGS[9]. Also NASA Ames implemented a distributed model in producing PLOT3D[10] with RIP[11] for steady-state problems. This combination was used to compute and visualize stream-lines when the workstation technology was not powerful enough to give good interactivity.

  Visual3, and, **pV3** are always animating in a large loop. Figure 2 shows that the **Visual3** loop is composed of three basic parts, see Haimes and Giles[2] for a complete description. The first part is event collection. The user input is pulled off the X-event queue and internal flags, transformation matrices and other internal states are changed appropriately. The second portion is data collection and (re)calculation of any surfaces or other objects that have changed. The final section is the drawing portion. When the data in any window has been modified the window is redrawn to reflect the new data.

  For **pV3**, the first and last portions of the animation loop remain on the graphics workstation (GWS) but the middle part is handled in the computer that contains the volume of data. The only data transferred to the workstation is the information required to draw the objects on the scene, the 'extracts'. The actual rendering is performed by the GWS to allow 3D rotations and lower dimensional mappings and probes without having an impact on the machine performing the compute.

  This scheme greatly reduces the data bandwidth requirements over **Visual3**. Only selected '2D' (and lower dimensional) data is sent to the GWS for each time frame. Each 'extract' is only a small percentage of the entire volume of data (between 0.3% and 6.7% for surfaces[7]), therefore the break even point is on the order of 30 or more '2D' objects. This is an amount that would result in screen clutter and not usually requested. Also, in general, the larger the 3D data set, the proportionally smaller the 'extract'. '2D' objects tend to be order $N^{2/3}$, where N is the number of nodes in the volume. '1D' objects such as stream-lines and '0D' objects (particles) require the movement of such small amounts of data that they have little or no impact.

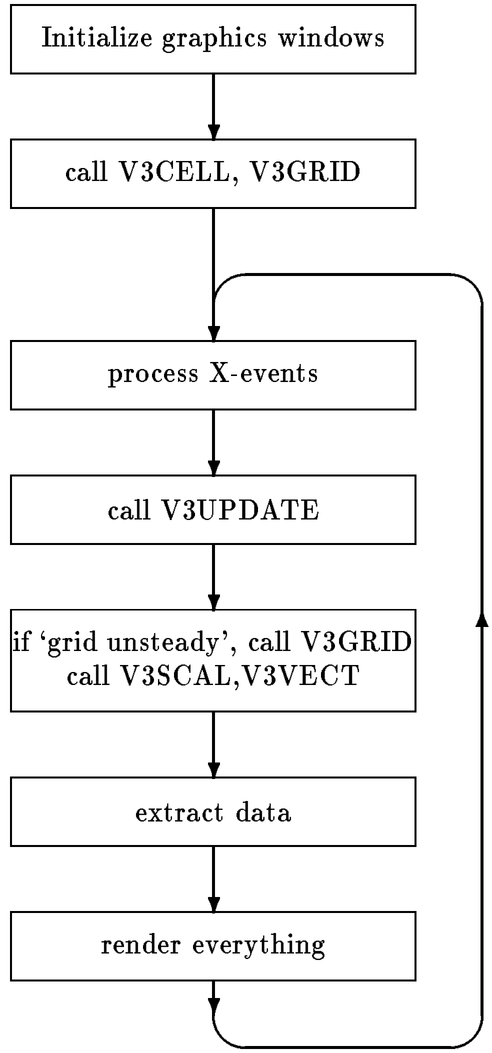

Figure 2: Flow diagram of **Visual3** for an unsteady application

- Multi-machine Visualization

  This visualization model assumes that the calculation is being performed in a multi-workstation environment (or on a massively parallel machine where each compute node is exposed to the network) and the problem already has been split in some spatial way. The 'load balancing' required to get efficient compute should have been performed. Again, the results can be viewed and steered during the compute cycle.

  Two different modes are implemented. In the first, asynchronous viewing, the data coming from the compute processors is displayed when it reaches the visualization workstation with no regard to the simulation time. This is useful for watching steady-state calculations converge. In the second, lock-step viewing, the data is presented in a time accurate manner. This is the normal mode for viewing unsteady calculations.

The scheme follows:

- Graphics workstation

  This workstation controls the visualization and broadcasts current visualization state information, such as the current vector and scalar fields and the active tools and probes. Then it sets up to receive the data for the next incoming time. This goes on concurrently with the actual visualization controlled by the investigator. If the user causes any changes that effect the visualization portion running on the compute processors, that information gets transmitted with the next broadcast.

  This workstation collects the pieces of the visualization extracts from the compute processors and assembles and renders the scene presented to the user.

- Compute processors

  Each compute node or processor performs its appropriate calculations for the problem, then as its portion of the solution is updated, routines supplied to perform the visualization task are called. These routines look at the current visualization state and then compute and produce the appropriate extracts for this part of the volume at this time. The data is sent to the visualization workstation and then control is returned to the compute task.

  The compute task need only be modified in a simple manner (the addition of a single call at the appropriate time plus some additional routines to define the scalar and vector field variables of interest) in order for this scheme to work.

A programming model for a typical iterative CFD application can be seen in Figure 3. In general, at the top of each time step boundary conditions are calculated and set, the solver computes the change to the solution for the entire volume. Then the change to the solution is applied.

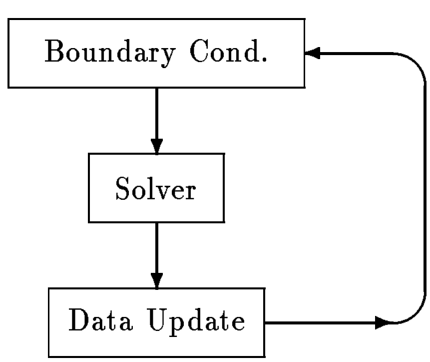

Figure 3: Archetypical Iterative CFD Solver

The solution scheme is easily modified to allow co-processing visualization with **pV3**. All that is added is a call

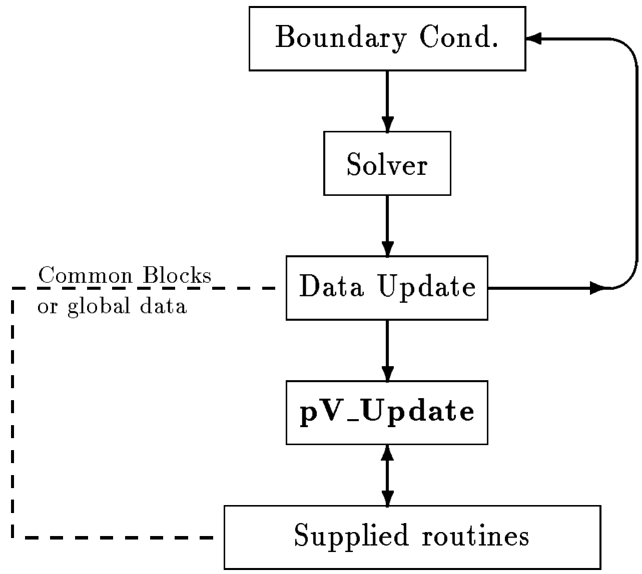

Figure 4: Flow diagram of the **pV3** client program

to the package after the solution space has been updated, as can be seen in Figure 4. The programmer supplied routines feed **pV3** with the solver's data when needed. The major routines are discussed in the Client Programming section.

There are two basic methods for starting the visualization (**PVM** terminology in *italics*):

- Connect to solver

  The assumption is that the solver is already executing. Without the **pV3** server running, every time the solution is updated and **pV_Update** is called, a check is made for any members in the *group* 'ServerpV3'. If none is found, this routine returns. When the **pV3** server *enrolls*, it waits for initialization messages from the clients. The next time the client calls **pV_Update**, an initialization message is sent to the server. The **pV3** server processes all initialization messages, figures out the number of clients and the session begins. Each subsequent time the client calls **pV_Update**, visualization state messages and 'extract' requests are gathered. Then the appropriate data is calculated, collected and sent to the GWS. Like in **Visual3**, when the code calls **pV_Update**, additional routines provided by the programmer are called to supply **pV3** with data about coordinates, scalar and vector fields.

  When the user is finished with the visualization, the GWS sends a termination message, leaves the server *group*, and exits. The client *instances* receive the message and clean up any memory allocations used for the visualization. **pV_Update** reverts to looking for *enrollment* in the 'ServerpV3' *group*.

- GWS initiated

  In this model, the **pV3** server starts on the GWS. It *initiates* the client *instances* that will calculate (or otherwise gather) the solution data. From this point on, the distributed system acts like the above technique where the control of the data is handled through the routine **pV_Update**.

  When the visualization session is terminated, the client *instances* producing the data also exit.

6 Discussion

For this architecture to meet the design goals, attention must be paid to the selection of the two crucial hardware components, the network and the GWS. **pV3** will run on an Ethernet sub-net (10 Megabit/sec), but much better overall performance can be realized on a FDDI (100 Megabit/sec) or faster network. **PVM** is based on standard TCP/IP protocol and that doesn't change with the type of wire. Therefore no re-coding is required when upgrading the local sub-net.

The other important part, the GWS, must not be encumbered with other compute tasks. It requires high-end 3D graphics hardware, a reasonable amount of physical memory (you don't want to be paging!), and good compute power. The GWS is also doing a large amount of data handling as well as dealing with the users interactions. In fact the tasks can be segregated into three categories; (1) dealing with the user through the GUI, (2) rendering the scenes and drawing to the windows, and (3) dealing with the compute node(s), broadcasts and collecting the resultant 'sub-extracts'. The first is not time consuming and therefore has little impact. The second and third are both crucial for interactivity and can possibly interfere with each other. While rendering an object, enough data may be coming to the GWS from the client(s) that buffers may fill causing a locking situation, or worse the loss of data, or a wedged cluster.

Early versions of **pV3** require a GWS from SGI. Only SGI and SUN provide workstations that support multi-threading (two or more processes executing concurrently and sharing the same address space) but SGI also supports the type of high-performance 3D rendering required. The Power Series or the new Onyx machines also allow the threads to run on seperate processors. Multi-threading with atleast two threads relieves the potential problems of the graphics interfering with the data collection, gives better overall interactivity, and results in cleaner software development. One processor handles the user interaction, scene construction, and window updates through graphics double buffering. The second thread broadcasts out the state information and then just waits to collect the incoming data. An added benefit is that data double-buffering (between the two threads) is as simple as changing a shared index, thus minimizing the amount of data copying. See Figure 5.

If the GWS has more than two processors, it can also be part of the compute cluster in the distributed workstation model. In this situation the processors unused by the visualization task can be doing compute, but the shared memory model is not used. Communication between the visualization and compute tasks is still done by **PVM**. A special case is where the entire compute task is on a multi-processor machine. **Visual3** can be used in the co-processing mode taking full advantage of the shared memory model and avoiding latency due to network bandwidth and distributed asynchronous operation.

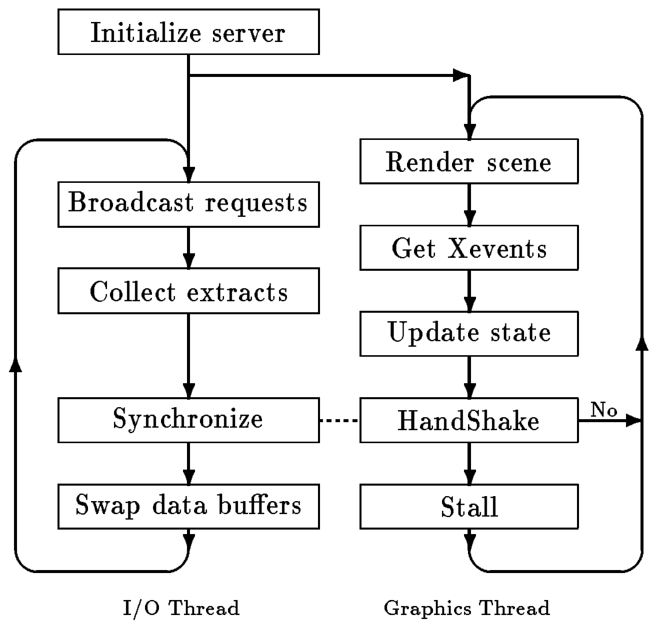

Figure 5: Flow diagram of the **pV3** server

The GWS requirement clearly doesn't fulfill the design goal about unusual equipment. But the benefits far outweigh this restriction in meeting the interactivity and high performance goals. It is hoped that in the future other workstation vendors will provide similar equipment.

7 Client Programming

In a multi-client simulation each client can have a different class of unsteadiness. Each mode causes a different internal calling sequence. In general, the application must first call **pV_Init** to initialize the **pV3** client subsystem and then call **pV_Update** after every time the solution space has been updated, see Figure 4.

Programmer-supplied routines:

- pVStruc: supplies the sizes of the current state of the problem (required for 'Structure Unsteady' only)

- pVCell: returns the cell data structures (required for hybrid and non-structured block cases)
- pVSurface: supplies the bounding surface data structures
- pVEquiv: returns the node equivalency data (optional)
- pVGrid: returns the grid coordinates (XYZ triads) for all nodes
- pVBlank: supplies the blanking (optional and only for structured blocks)
- pVScal: returns the requested scalar function values for all nodes
- pVVect: returns the requested vector function values (optional)

Obviously, all cases of unsteadiness are a subset of 'Structure Unsteady'. The classification of applications in this manner facilitates reducing the visualization computation load and allows intelligent passing of 'sub-extracts'.

The **pV3** client calling sequences based on unsteadiness is:

- Data Unsteady

Call	Calls in Sequence
pV_Init	pVCell
	pVSurface
	pVEquiv
	pVGrid
	pVBlank
pV_Update	pVScal
	pVVect
	all others as needed

For geometric cuts and domain surface mappings the 'sub-extracts' that define the surface (the collections of tris, quads, coordinates and mesh data) are only sent once. The scalar and vector field 'sub-extracts' only get passed when the surface or cut is plotted with an attribute that requires the data.

- Grid Unsteady

Call	Calls in Sequence
pV_Init	pVCell (optional)
	pVSurface
	pVEquiv
pV_Update	pVGrid
	pVBlank
	pVScal
	pVVect
	all others as needed

For domain surface mappings the 'sub-extracts' that define the surface structure (the collections of tris, quads and mesh data) are only sent once. This is because the underlying structure of the surface is not changing. The node coordinate, scalar and vector field 'sub-extracts' only get passed for the surface when it is plotted with an attribute that requires the data.

- Structure Unsteady

Call	Calls in Sequence
pV_Init	NONE
pV_Update	pVStruc
	pVCell
	pVSurface
	pVEquiv
	pVGrid
	pVBlank
	pVScal
	pVVect
	all others as needed

Because there is the chance that everything has changed from last iteration, no data is retained (unless the user has paused the visualization).

pV_Update is where all interaction with the graphics workstation is performed. Therefore the overall response and the interactive latency depends on how often this routine is called. About one call per second is optimal. If the solution is updated significantly faster then most of the compute cycles will be used for the visualization, moving the solution slowly forward in time. In this case it is advisable to call **pV_Update** only every N times the solver updates the solution.

The more difficult case is when the solution update rate is much slower than optimal. In this situation, there are two choices; (1) live with large lags between user requests and screen response or (2) setup another task between the solver and the **pV3** server. This software's responsibility is to communicate with the solver. It should be the task to make all **pV3** calls.

This secondary task can communicate with the solver using **PVM** (and therefore must be on the same machine to avoid large network transfers). Or, if the machine supports multi-threading, the task can be a second thread and perform double-buffering of the solution space, so no data need be transferred. These methods are a trade-off of memory usage for interactivity.

8 Conclusions

Many performance issues had to be solved during the design and development of **pV3** to achieve the desired interactivity and allow interrogation of large scale 3D volumetric unsteady data-sets. Handling the types of transient data produced by CFD codes required much re-thinking of the standard approaches to scientific visualization.

A distributed visualization system has been developed specifically for large scale (order million nodes or more) volumetric transient scientific problems. An important and unique aspect is the ability to perform the visualization as the solution is generated. This involves coupling to the solution scheme and having parts of the visualization system resident in the compute arena used for the solver. The architecture developed minimizes the amount of data transferred and does not sacrifice functionality or interactivity.

Acknowledgments

The author would like to thank Mike Giles of Oxford University for early design discussions. Dave Lane (Computer Sciences Corp.) at NASA Ames Research Center helped to maintain a perspective on the size of the problems, and Al Globus (Computer Sciences Corp.) also at Ames for general articulation on various visualization subjects.

Matthew Scott and Philip Raymond of the Computational Analysis Group at SGI were instrumental in this effort by supplying the correct equipment. In addition, the author wishes to thank the NASA Ames Research Center, NASA Lewis Research Center and Rolls Royce *plc* for their financial support of this project.

References

[1] M. B. Giles and R. Haimes. Advanced Interactive Visualization for CFD. *Computing Systems in Engineering*, 1(1):51–62, 1990.

[2] R. Haimes and M. Giles. VISUAL3: Interactive Unsteady Unstructured 3D Visualization. AIAA Paper 91-0794, 1991.

[3] D. Lane. Visualization of Time-dependent Flow Fields. submitted to IEEE Computer Society, Visualization '93, October 1993.

[4] R. Haimes and D. Darmofal. Visualization in Computational Fluid Dynamics: A Case Study. IEEE Computer Society, Visualization '91, October 1991.

[5] A. Grimsrud and G. Lorig. Implementing a Distributed Process between Workstation and Supercomputer. Proceedings of the first International Conference on Applications of Supercomputers in Engineering, September 1989.

[6] C. D. Hanson and P. Hinker. Massively Parallel Isosurface Extraction. IEEE Computer Society, Visualization '92, October 1992.

[7] A. Globus. A Software Model for Visualization of Time Dependent 3-D Computational Fluid Dynamics Results. NASA Report RNR-92-031, November 1992.

[8] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and Sunderam V. A Users' Guide to PVM: Parallel Virtual Machine. ORNL Report TM-11826, July 1991.

[9] Cray Research Inc. MultiPurpose Graphic System Users Manual. Revision 4.1, APG-5527.

[10] P. Buning and J. L. Steger. Graphics and Flow Visualization in Computational Fluid Dynamics. AIAA Paper 85-1507, 1985.

[11] S. Rogers. Distributed Interactive Graphics Applications in Computational Fluid Dynamics. *Int. J. of Supercomputing Applications*, 1(4), 1987.