FAST[1]: A MULTI-PROCESSED ENVIRONMENT FOR VISUALIZATION OF COMPUTATIONAL FLUID DYNAMICS

Gordon V. Bancroft
Fergus J. Merritt
Todd C. Plessel
Paul G. Kelaita
R. Kevin McCabe

Sterling Federal Systems Inc.
1121 San Antonio Road
Palo Alto, California 94303

ABSTRACT

Three-dimensional, unsteady, multi-zoned fluid dynamics simulations over full scale aircraft is typical of problems being computed at NASA Ames' Numerical Aerodynamic Simulation (NAS) facility on CRAY2 and CRAY-YMP supercomputers. With multiple processor workstations available in the 10-30 Mflop range, we feel that these new developments in scientific computing warrant a new approach to the design and implementation of analysis tools. These larger, more complex problems create a need for new visualization techniques not possible with the existing software or systems available as of this writing and these visualization techniques will change as the supercomputing environment, and hence the scientific methods employed,[1] evolve even further.

Visualization of computational aerodynamics requires *flexible*, *extensible*, and *adaptable* software tools for performing analysis tasks. *Flexible* means the ability to handle a diverse range of problems. *Extensible* means the ability to interact at all levels of the software hierarchy, either through existing built-in functionality or through the implementation of custom "plug-in" modules. *Adaptable* means the ability to adapt to new software and hardware configurations through the use of modular structured programming methods, a graphics library standard, and the use of common network communication protocols (like UNIX sockets) for the distribution of processing.

This paper discusses FAST (Flow Analysis Software Toolkit), an implementation of a software system for fluid mechanics analysis that is based on this approach.

BACKGROUND

Computational Fluid Dynamics (CFD), involves the use of high speed computers to simulate the characteristics of flow physics. Computational aerodynamicists use CFD methods and solvers to study subsonic, supersonic, transonic and hypersonic (compressible) regimes of flight, in addition to studying incompressible problems within particular systems. Examples of ongoing studies on full-scale aircraft configurations at NASA Ames include the Space Shuttle, F16, and the Aerospace Plane. Specialized areas of research include jet-engine turbine flow, VSTOL and ground effect research, and even flow through an artificial heart. Basic CFD research involves unsteady flow phenomena like vortex shedding and turbulence modelling.

A flow solver running on a supercomputer must handle input files (finite difference grids, ref. 7,15,16) that are typically very large. For example, the number of xyz triplets (each represented by three eight-byte floating point numbers) in a 100 x 100 x 100 grid yields a 24 Mbyte file. If complexity is added, or the grid resolution (density of points) must be raised for flow solving to yield acceptable results, the files grow proportionally in size. Once the solver has been run, there are from five to eight variables for each grid node, again, each represented by an eight-byte floating point number. For the 24 Mbyte example, five variables for each grid point yields a 40 Mbyte raw data file. This is a total of 64 Mbytes (grid plus the solution) for this example. The F16 mentioned previously, which consists of 29 grid zones, is over 108 Mbytes worth of data! *(Note: On the workstations these become four-byte IEEE format floating point numbers making the files about half this size)*

---

[1] FAST (Flow Analysis Software Toolkit) Developed by Sterling Federal Systems Inc. under contract to NASA Ames Research Center NASA Contract #NAS2-13210.

Three examples of grid generation programs are:

3DGRAPE 3-dimensional grids about anything by Poissons Equation (Sorensen)
EAGLE
GRIDGEN3D

A list of commonly used flow solvers are:

ARC2D Ames Research Center 2-dimensional solver (Pulliam)
ARC3D Ames Research Center 3-dimensional solver (Pulliam)
TNS Transonic Navier Stokes solver (Flores)
CNS Compressible Navier Stokes solver (Flores)
PNS Parabolized Navier Stokes solver (Chausee)
INS3D Incompressible Navier Stokes solver (Kwak)
TWING Transonic Wing solver (Thomas)

Programs available for visualization of CFD data sets are:

PLOT3D A command line driven Fortran program that computes CFD quantities (Buning [7])
SURF Allows for the rendering of smooth, wireframe, and function mapped surfaces with a more interactive interface (Plessel[8])
GAS Combines graphics generated from PLOT3D and SURF and allows animations to be created and recorded (Merritt[9])
RIP A program for interactive particle tracing (Rogers[19])

FAST OVERVIEW

The software cycle for the creation and and analysis of computational fluids results could be reduced to the following conceptual model:

- Data generation *(Flow solving)*
- Data manipulation *(The original data may need to be filtered or transfered)*
- Data abstraction *(A graphical object is defined using the data)*
- Data rendering *(Viewing on a workstation)*
- Data interpretation *(analysis)*
- Feedback *(Perhaps go back to previous phases)*

A problem with the existing CFD software is that it takes a non integrated approach to dealing with the different steps of the CFD process. The grid generation and flow solver programs are involved in the data generation phase. The visualization software is part of the abstraction, rendering and analysis phases. The various programs present the user with different interfaces, and there is little attention paid to the data manipulation and feedback steps. In the current system, large data sets flow from one step to another from disk to ram and back to disk (perhaps from one computer to another), taking on different file formats along the way.

The design criteria for FAST were:

- Minimize the data path in the CFD process
- Provide a consistent user interface
- Allow for quick user feedback
- Provide an extensible software architecture
- Provide a quick path through the CFD process
- Provide libraries and tools so that application modules could be added easily
- To isolate 3D viewing tasks from the application programmer

In order to achieve these design goals FAST has evolved into collection of programs that communicate via Unix sockets with a central hub process that manages a pool a shared memory. A fundamental data type is loaded or generated and stored into shared memory (data generation and manipulation), a collection of programs (modules) operate on data and produce additional data (objects) that are also placed into shared memory (data abstraction). The objects are rendered using the fast viewing system (data rendering). Data is analyzed by additional modules or visual inspection (data analysis). Depending on the results of the analysis the user changes input to any of the previous modules (feedback). In addition there is a collection of libraries and utilities that are used to build the application modules.

The use of shared memory reduces the flow of data in the system. The use of a viewing process relieves the burden of three dimensional interactive viewing from the application programmer. The fact that the fundamental data type(s) reside in shared memory makes it easy to make changes based on the feedback obtained from the analysis phase. Finally the use of FAST libraries and utilities makes it easy to add new modules.

We are aware of other scientific visualization packages and visualization capabilities in existence and/or under development. These include visual programming examples like CONMAN (Silicon Graphics[3]) and AVS (Application Visualization System, Stardent Computer[4]), and other scientific visualization environments like MPGS (Multi-Purpose Graphics System, Cray Research), and the Personal Visualizer (Wavefront), as well as 'scripting' languages like PVWAVE (Precision Visuals), IVIEW (Intelligent Light), and VISAGE (Visual Edge) to name a few. While FAST is built specifically around the research tasks involved in CFD analysis, these other environments and packages typically take a much more generalized approach towards visualization, for the obvious reason that CFD research is a relatively small part of their intended audience. These systems and environments often require a certain level (a 'power' user, visual programmer, or animation /rendering expert) of skill with computer graphics above and beyond the level of the typical CFD scientist. In researching these other more general approaches, we have discovered that the results (data) get handed off at some point to the 'power' user (or perhaps even computer graphics group or expert) and this person (or group) creates the animations, films or videos. FAST is built around a model where the scientist is the first and last person in the data chain and FAST is a toolset for his environment. This is not meant as a criticism of these other approaches, as the need for generalization dictates the need for this other level of user. It is our belief, though, that the techniques used in FAST presented in this paper would also apply and be very useful in the more general environments.

Graphics, CPU, and memory handling performance were key considerations in the FAST design and development process. For graphics, a base-line level of what is commonly termed (but undefined) as "real-time" had to be established and agreed upon as acceptable. This was determined to be a minimum of 3 frames/sec for a typical 10-20 Mbyte problem (techniques used for rendering would determine the problem size in this range). This base line frame rate was determined to be essential in visualization of fluid mechanics for understanding the dynamics of the simulations. For the development platform, the Silicon Graphics 4D220/GTX (16 Mbytes memory) this goal was reached, and we are pleased with the current performance level. The Silicon Graphics 4D320/VGX, has even higher levels of cpu and graphics performance[18], although specific test results do not yet exist.

We have implemented in FAST new techniques and capabilities non-existent in the previous tools and expanded on others. For example, the colormap editing capabilities were enhanced to include banded, spectrum, dynamic, contour, striped, and two-tone function mapping. Surface rendering includes the ability to 'sweep' planes through the data either grid oriented, arbitrarily oriented, or a contour surface (isosurface). Enhanced titling and labelling features include the use of postscript type fonts and symbols, where typeface, font point size, and style can be specified. The animation capability is substantially enhanced beyond what was available in GAS (Graphics Animation System[9]). These enhancements include greater control by allowing the ability to edit scenes, views, and objects. Another capability allows for separate scenes to be rendered in separate windows giving the scientist/user even more flexibility and animation control.

At the time of this printing, the software is in Beta testing at NASA Ames Research Center. The typical workstation environment is a Silicon Graphics 4D/VGX Power Series class machine. The Beta release users currently include approximately a dozen CFD research scientists and application programmers at approximately 250 sites across the country.

FAST ARCHITECTURE

Each separate process communicates through the FAST Hub while managing shared memory and communicating using standard Berkeley UNIX Interprocess Communication (IPC[11]).

Hub

The central process of the FAST environment is the Hub module (Hub, figure 2). The Hub module invokes and shuts down the FAST modules yet its main function is to process requests sent by the modules. These requests might be to allocate a segment of shared memory and return the shared memory id, or to delete a shared memory segment. Since the Hub process is always running as long as FAST is active, the data allocated through the Hub remains accessible even when the original process which requested it is terminated. The Hub module is essentially transparent to the user, in that it has no panels.

Viewer

This is the central module for processing, from the users perspective (Viewer, figure 3). This is where the graphical data pool generated by other modules is managed and interactively viewed. FAST Central, unlike other FAST modules, runs continuously while FAST is up and running. Other modules can be spawned or shut down as they are needed from the Viewer module. In addition Viewer allows object attributes to be set (e.g. transparency, mirroring, line width), scene attributes to be set (e.g. lighting, color map editing, background color), viewing preferences to be set(e.g. toggle axis, mouse axis modes) as well containing the animation control panels. Animator is used to create and record smooth (spline interpolated) keyframe animation sequences.

File I/O

The file i/o module (file i/o, figure 4) loads pre-computed *PLOT3D* type grid, solution, and function files as well as ARCGraph[20] files into FAST's shared memory. It consists of three control panels. The file input panel is used to list file names and information and to load data into shared memory. The data sub-panel displays pertinent information about the previously loaded grids and solutions. The ARCGraph panel is used for handling this type of file input.

CFD Calculator

The CFD Calculator (figure 5) module allows the scientist to attach to the grid and solution data that has been loaded and to calculate a variety of scalar and vector functions for analyzing the computed solution. The Calculator has the appearance and functionality of a real programmable calculator but instead of operating on numbers it operates on *fields* of numbers (scalars) and *fields* of vectors.

Its basic operations (e.g., +, -, MAG, CURL), are applied to entire fields - either component-wise or vector-wise. For example, + applied to two scalar fields will produce a new scalar field of values that are the sums of the corresponding values of the two operand scalar fields. And LOG applied to a vector field will generate a new vector field by taking the logarithm of each component of the corresponding operand vector. In addition to component, scalar and vector binary operators there are also special operations such as GRADIENT, DIVERGENCE, DOT, and CROSS that apply to entire fields and produce new scalar or vector fields.

The scientist can select a range of active solution zones on which to operate and use the CFD Calculator to compute about 100 different built-in CFD scalar and vector functions such as Pressure, Enthalpy, Normalized Helicity, Velocity, and Vorticity [16]. These fields are stored in one of the Calculator's scalar or vector *registers*. The Calculator can then be *programmed* with formulas that operate on these fields and produce new ones using the basic operations already mentioned. The CFD Data Panel is used to copy, move, delete, and display information fields (such as min-max) stored in the Calculator's registers. These features, and others, help make the CFD Calculator an interactive, powerful tool that the CFD scientist can use to compute important quantities for analyzing computed solutions.

SURFER

The SURFace Extractor and Renderer module (figure 6) attaches to grids (loaded by the file i/o Module) and scalar and vector fields (generated by the CFD Calculator) and renders grid surfaces as points, lines, vectors, or polygons. These *grid surface objects* are also stored in shared memory so they can be rendered in the FAST environment. The grid surfaces can show the grid geometry, for example, a lighted, Gouraud [2] shaded polygon surface of the Space Shuttle, or they can display the scalar data as function colored lines or polygons, or vector data as line vectors, vector heads, or polygon vector deformation surfaces (vector heads connected in a surface). Grid surface objects can represent grid geometries, scalar fields, and vector fields.

In addition to changing data types, surface rendering and other attributes, SURFER can sweep through all surfaces in a given grid direction.This creates a dynamic image showing even more features of the flow field.

Titler

The Titler module (figure 7) is used to create high quality Postscript text suitable as titles for images in videos, slides, and movies. Title attributes include font, point size, position, color, drop shadows, and a snap-to-grid feature to make alignment easier. Like other graphical objects, *title objects* are stored in shared memory so they can be added to other scenes. Postscript fonts from other sources may be imported and created titles may be saved for later use.

Isolev

Isolev (figure 8) performs three functions using a single algorithm. One, it draws surfaces of constant value in 3D scalar fields, i.e. isosurfaces. Two, it draws cutting planes function mapped by the scalar field of interest. Cutting planes may be at any angle, and are consistently oriented throughout a multi-zoned grid. Three, it draws vector field deformation surfaces originating at cutting planes or isosurfaces. Iso and deformation surfaces are lighted and smooth shaded. Both isosurfaces and cutting planes may be rendered as dots for improved performance. Interactive grid coarsening is available to improve interactivity. The user may also set up sweeps, where isolev automatically sweeps the isovalue (or cutting plane location) through all possible values, or within a user specified range. This can be used to get a feel for the entire volume. The marching cubes algorithm [Kerlick,13] is used to generate polygons. Level scalar fields are created to generate cutting planes function mapped by the scalar field of interest. Edge crossings, a faster algorithm, is used to generate points. A user selected vector field may be used to draw vectors originating at the crossing points.

Tracer

The tracer module (figure 9) is used to compute particle traces and render them as vectors through the flow field. Tracer attaches to a grid and solution and allows the user to interactively select the point of release or rake[7] from which the traces are computed. The traces can either be computed forward or backward in time as well as allowing the user to selectively save traces. Once traces are saved, a delta time factor may be interactively adjusted through the panel to allow particle trace "cycling".

Topology

The topology module identifies and classifies critical points in a flow field. Critical points are marked with icons which visually identify the class of the point. Traces can be computed at or about these critical points. Topology can find and display vortex cores by examining eigan vectors.

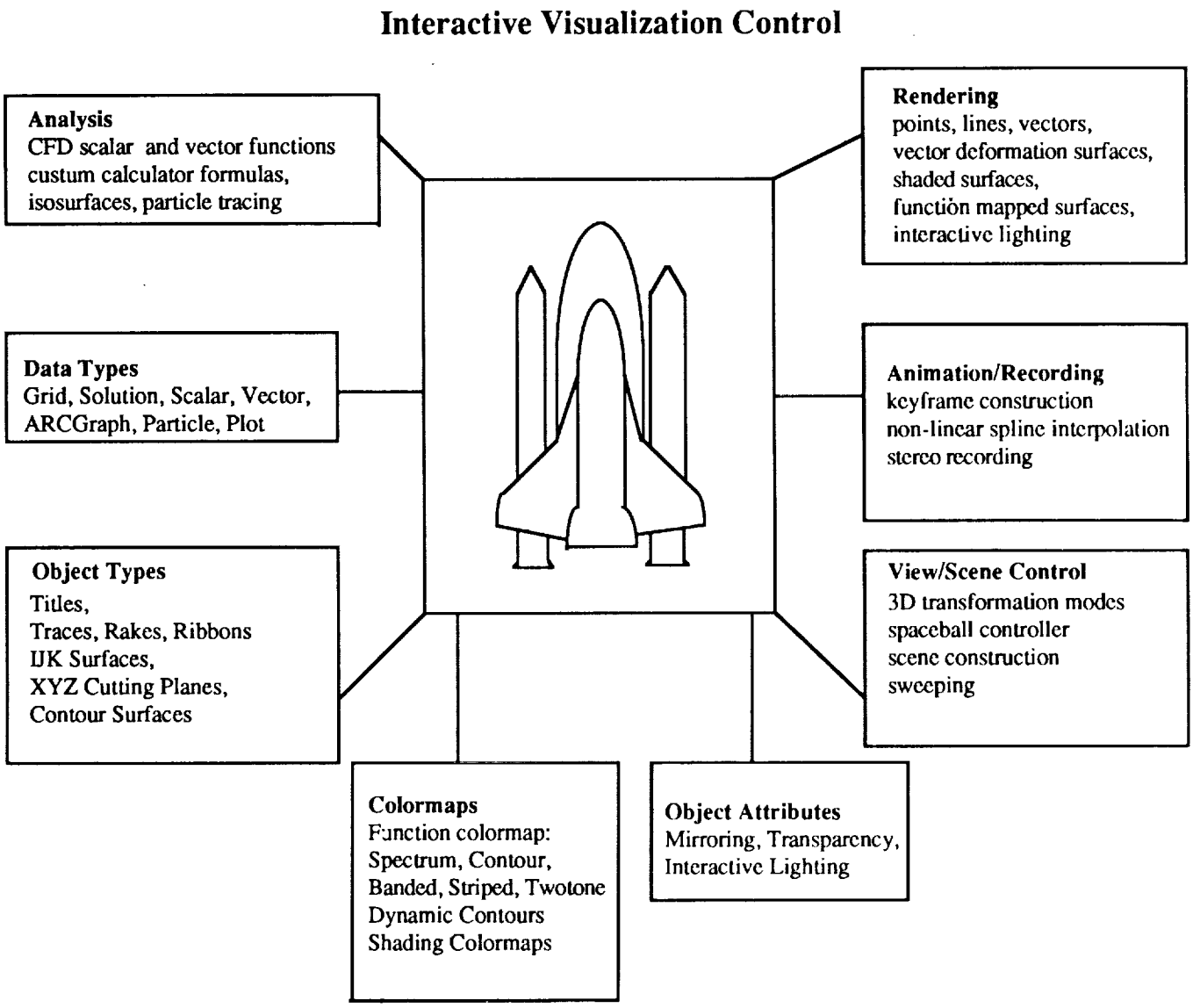

Figure 1, "FAST Interactive Visualization Control"

Interactive Visualization Control

Multi-processed: In figure 3, several modules worked together to generate the scene: Surfer generated the grid surface objects, CFD Calculator computed the scalar and vector fields, Titler was used to generate the text, and Viewer was used for image handling and color map editing. When modules are not needed they can be iconified so they occupy less screen space and CPU resources. Because of this, the FAST environment can be running while other applications are also being used. Alternatively, FAST modules can be terminated without exiting the FAST environment - and this has no effect on their data since it is already in shared memory. Unlike standard dynamic memory, shared memory remains available even after the allocating process is killed. All shared memory segments are removed when FAST is exited via the Quit selection of the Viewer module.

Powerful. The FAST environment contains sophisticated tools such as the CFD Calculator that enable the scientist to analyze computed solutions by examining many relevant "CFD quantities", such as normalized helicity, shock, perturbation velocity, and vorticity . And if these "built-in" functions are not adequate the scientist can program the Calculator to compute customized functions using the rich set of component, scalar, and vector operators. In figure

3, the CFD Calculator was used to compute entropy and pressure scalar fields and a velocity vector field (see "FAST Architecture, CFD Calculator, Page 7).

Flexible. Storing data and graphical objects in shared memory has enabled the complex scenes in figure 3 to be constructed by mixing and matching shared data from any module that is currently plugged into the FAST environment. The figure shows how grid, scalar, and vector data has been combined to generate grid surfaces rendered as grid lines, scalar colored smooth polygon surfaces, and vectors.

Interactive. Surfer provides the ability to interactively alter scene attributes such as coloring the data by a different scalar field, displaying a different vector field, adjusting the legend, normalization, and clipping ranges, or changing rendering and data types. For example, the vector field can be rendered as a Gouraud shaded, lighted, vector deformation surface. With the looping option turned on Surfer will sweep through all data in the current grid direction - providing a dynamic visualization ability. And while this is happening the scientist (from Viewer) can transform (e.g., rotate, or zoom) all or part of the scene or use the color map editor to adjust the function color mapping by inserting, deleting, and changing colors, or selecting a different colormap types such as *Spectrum, Contour, Striped, Twotone*, or *Banded.*

IPC and Shared Memory Implementation

It was decided that an interprocess communication (IPC) package must be implemented to allow FAST to operate as a modular environment where resources could be shared among different machines as well as a single host. Specifically, Unix System V shared memory facilities are used to allow each process (module) to access the environment's data, while the Berkeley IPC package's implementation of Internet domain stream sockets allows for the coordination of this data.

As each module is executed by the FAST hub, it must immediately establish a two-way communication channel between it and the hub. Because an Internet domain address consists of a machine network address and a port number, these two values are used in establishing this connection. The following command is therefore executed at the beginning of a module's main routine:

```
socket_establish_and_accept (hub_host, hub_port, &rsock, &wsock);
```

This does the following:

1) create a socket from which to read
2) determine a local port and listen on it
3) create a writeable socket and establish a connection to the hub (using the hub's hostname and port number which came in as arguments)
4) now send the port number to the hub and
5) accept a connection from the hub

At the same time, the hub process executes this statement:

```
socket_accept_and_establish( sock, module_host, &wsock );
```

which does the following:

1) create one socket from which to read from all modules
2) accept a connection from the next module
3) read in the module's port number
4) create a writable socket and connect this socket to the module

After a two-way connection has been established, both the hub and the module are left with two socket descriptors each. These are used exactly as a file descriptor is used, one for writing (wsock) and the other for reading (rsock). The hub actually stores these descriptors along with other pertinent information, such as module status, in an array of structures - one structure for each module.

The modules specified for inclusion in the FAST environment are specifically listed in a "run command" file called $HOME/.fastrc. Also included within this file is information about initial placement of a module's main panels, the name of the host where the module resides, and the complete path name of the particular module.

Once a module has been executed by the FAST hub using the Unix system(3) call and the communication channels have been established, the hub enters a loop where it waits on a request from any of the active modules to perform some sort of action. The hub process uses the Unix select(2) call to examine all available read socket file descriptors to determine if they are ready for reading. This appears as follows:

```
while (continue_looping) {
        for each module
                load read socket id into fds, file descriptor structure
        select (fds, 1,000,000 seconds)  i.e. pause here until a request is detected
        communication is detected ... determine from which module
        read up the request from that module
        process request
} end while
```

Information sent between a module and the hub (and vice-versa) is always preceded by a standard sized structure which contains, the command and four information fields. The necessary information, if any, is then written back to the module, and the flow control takes the hub back up to the point where it can again wait for a request.

One example of a request that a module might make would be the allocation of memory which may eventually be used by another module. It must first send a request to the hub to do this. The hub then allocates the memory as a shared memory segment and retrieves the shared memory identifier associated with this segment. This identifier is then stored by the hub in a data structure possibly to be accessed by another module at a later time. Finally this identifier is sent back to the module so that it may attach the shared data to it's virtual memory address space.

At any time that a different module would like to access this data, a request is similarly sent to the hub to retrieve the shared memory identifiers so that it too may attach to the data.

A consequence of using shared memory instead of standard dynamic memory is that dynamic data structures such as linked list nodes no longer have a *pointer* to the next node but rather the *shared memory id* of the next (and current) node. And this shared memory id must be explicitly *attached* to and *detached* from whenever the structure is traversed.

The FAST environment contains several lists of this form: a list of grids, a list of solutions, a set of scalar and vector lists (one for each register of the CFD Calculator), and a list of graphical objects. A typical request that a module would make of the Hub is to gain access to a particular list node, for example, a node from one of the CFD Calculator's vector register lists. This would involve setting up the fast_infobuf with the appropriate information about the request, writing it to the Hub, reading the node's shared memory id from the Hub, and attaching to generate a virtual address for the requesting module process. The Hub process detects the socket write in its main event loop and executes a socket read and calls the function process_request() to handle the module's request:

Modules that generate data to be shared must: 1) change low-level usage of pointers to shared memory ids, 2) alter management routines to explicitly attach and detach in addition to allocate and deallocate, 3) provide a library of routines that modules can link with that provide access to the actual data stored in these structures, 4) provide a library of routines that the Hub can use to create and destroy these structures (recall that the Hub is the single process that does all shared memory allocation and deallocation).

Graphical objects are also shared which means the structures that define them must reside in shared memory. Note that part of this structure references the shared memory ids of the grid, scalar and other data needed to draw a grid surface object. The routine draw_grid_surface() accepts this structure and draws it. This routine is part of the viewing library which is linked to every FAST graphical module so they can all include grid surface objects in their scenes.

Using shared memory and sockets, FAST is able to quickly and easily share all the data used within the environment. Even though shared memory can not yet be shared over different machines as it is on a single host, FAST has been designed with that feature in mind. When indeed we can accomplish this, the ultimate power of FAST can be realized.

DISCUSSION

For an existing SGI visualization application to be converted into a FAST module:

- Command line arguments must be used to establish window location and Hub communication - and nothing else.
- Periodically, each module must check for exit command IPC from the Hub. This is usually done once each time through the main event loop.
- Standard input should not be used.
- Standard output should be used sparingly for status and error messages.
- The colormap must be used according to FAST conventions. FAST library functions must be used to get color indexes for drawing. A few indexes are reserved for modules to create their own colors, but most of the colormap is only modified via the FAST COLORMAP module.
- Grid, vector and scalar field data must be accessed via FAST shared memory.
- The panel library should be used for menus, buttons, sliders, etc.
- The panel library's nap time or blocking should be turned on when waiting for user input to avoid excessive context switching.
- The application's drawing code must be integrated into the viewing library so that it's visualizations can appear in all modules.
- The data needed to draw must be placed in shared memory and made available to the viewing mode

There are several advantages to integrating applications into FAST as modules. These advantages include:

- Shared memory speeds which allow users to interactively view their data from several modules without long disk IO delays.
- Access to CFD Calculator generated vector and scalar fields.
- Precalculated min and max for grids, vector and scalar fields. This reduces the time needed to access data in many cases.
- Sophisticated colormap manipulation using the FAST COLORMAP module.
- Integration of visualizations created by several modules into a single scene.
- Trivial integration of visualizations into animations.
- Interactive access to most of the generic capabilities of the SGI graphics hardware, e.g. rot-tran-scale, using the viewing library panels.
- Other synergistic effects of multiple modules accessing the same data.
- New applications can be built quickly since many functions are be made available by existing FAST modules and libraries.

There are also some disadvantages, of course. These include:

- Time to learn to use the FAST libraries and intermodule communications as well as to keep up with future changes.
- Performance overhead due to multiple processes busy waiting.

Future plans for FAST include the capability for use across high speed LANs for 'smart' distribution of processing. Compute intensive modules could be distributed or broken up into components that communicate over these networks, or perhaps memory could be shared across systems.

As flow solvers become fully integrated, and interactive 3-d grid generation becomes a reality, FAST will continue to offer more effective visualizations of computational aerodynamics in all aspects of fluid flow simulations.