A Library for Visualization and Steering of Distributed Simulations using PVM and AVS

James Arthur Kohl Philip M. Papadopoulos *

Abstract

Large simulations often produce an immense amount of data that needs to be analyzed. Usually, a simulation is conducted on a parallel machine, multi-gigabyte volumes of data are written out to a parallel storage device and the resulting output is visualized long after the simulation has completed. Although this scenario is suitable for final production results where every output data point can be crucial, it is wasteful in the debugging and tuning stages. It also precludes the capability to steer interactively the computation in directions of interest. A better solution is to transfer a subset of data to a visualization front-end and view the output as it is being produced. Then mechanisms can be provided to monitor and control the computation on-the-fly.

The library described in this paper allows a programmer to set up an interactive steering and visualization interface between a commercial visualization package, such as AVS, and a data-distributed parallel application. The library manages the task of assembling a complete data "frame" from the distributed set so that one, two, or three dimensional fields can be animated. Data movement is kept to a minimum by performing dataset down sizing on the distributed processors before sending to the visualization front-end. Both the region of interest and the level of detail of the region can be specified interactively during a run. Several steering parameters can be sent back to the program to modify the simulation while it is running. The library uses PVM as the message passing interface and is therefore portable to a wide variety of platforms. The visualization front end may also be dynamically attached (and detached) to the parallel program allowing for a great deal of flexibility. A simple visualization of an expanding gas cloud is given as an example to illustrate the interface.

1 Introduction

Many large applications can strangle massively parallel computers with their data I/O requirements. Often, gigabytes of information are written out to a parallel storage array during the run. After a simulation has completed, the data is read off of a parallel storage device and analyzed off-line. This approach is inadequate in terms of debugging an application or steering a computation. For these cases, interactive control must be exerted during the program's execution. It is beneficial to integrate the power of visualization packages at run time, rather than merely as post-mortem analysis tools. By doing so, interactive capabilities can be added to a large simulation and the data can be analyzed during the execution. The latter aspect is very important in the debugging and tuning stages where generating a complete data result is unnecessary and wasteful, especially if an error could be discovered early on in the program. Visualizing the data is a simple way to discover non-physical phenomenon and identify other computational errors.

Because linking visualization packages with user applications can be especially complicated in the distributed computing environment, a library to simplify this linkage is useful. The library should free the programmer from worrying about the arcane aspects of communicating with the visualization front end, reconstructing a coherent view of a distributed

*Authors are with the Mathematical Sciences Section, Oak Ridge National Laboratory, P.O. Box 2008, Oak Ridge, TN 37831-6267.

dataset, and various other transport mechanisms. The library should be portable so that a researcher can view the data that is being generated on a parallel machine with a visualization workstation on the desktop. Since data is often generated at resolutions far greater that what can be rendered effectively on a computer screen, the minimum amount of data should sent from the computational back-end to the visualization workstation.

The interface described here provides a simple mechanism to transfer data from a parallel application to the visualization engine. The interface uses PVM (Parallel Virtual Machine) [2] to maintain portability and seamless connectivity among machines of differing architectures. AVS is used as the visualization front-end because of its flexibility and large number of supported platforms.

This paper is divided into six sections. Section 2 overviews the relevant features of PVM and AVS. Section 3 explains the high-level design of the interface and animation frame construction. Section 4 details some the protocols employed to achieve the high-level design. Section 5 shows some screen output for a small example program and Section 6 presents conclusions.

2 An Overview of PVM and AVS

2.1 An Overview of PVM

PVM is a portable software system that allows users to define a set of computers and program them as a single computational resource using message passing primitives. It has become a *de facto* message passing standard and reliably runs on a gamut of computers from desktop workstations to massively parallel processors. PVM allows for an integrated heterogeneous collection of computers and handles all aspects of translating data among machines with differing data representations. The network version of PVM runs on top of UDP, TCP/IP, and Unix domain sockets and inter-operates with optimized versions that use native message passing primitives on parallel machines.

PVM consists of two distinct pieces, the user library and a message router called the *pvmd*. A single pvmd runs on each host and the collection of pvmd's running on different hosts make up the user's "virtual machine." The user library provides both C and Fortran bindings for packing, sending and receiving messages; spawning processes onto the virtual machine; querying the state of processes or hosts; adding and deleting hosts from the virtual machine to dynamically reconfigure the host pool; and posting of various event notifications. In general, messages sent from one process to a remote process are routed to the local pvmd, transferred to the remote pvmd and finally delivered to the remote process.

PVM provides a name lookup mechanism so processes that are spawned independently on the virtual machine may "look up" a programmer-defined name and discover information about each other. This mechanism is critical to the library presented here where the visualization front-end is allowed to dynamically attach and re-attach to the computational back-end. PVM also provides event notification mechanisms to allow the programmer to provide fault tolerance. Suppose, for example, the computational back-end is waiting for a message from the front-end and the user shuts down the visualization system before the message is sent. Instead of waiting indefinitely, the user's code receives notification that the visualization process has exited and can then properly handle this condition.

PVM is an ongoing research project at Oak Ridge National Laboratory, the University of Tennessee, Emory University and Carnegie Mellon University and continues to evolve. More information about PVM can be retrieved using the worldwide web with the address http://www.netlib.org/pvm3/index.html.

2.2 An Overview of AVS

Scientific visualization is the application of computer graphics to represent multi-dimensional numerical data in a meaningful visual form using color and shape. Visualization in this setting provides a high-level interface to large amounts of raw data and organizes the data so scientists can see general trends and properties. In addition, animated visual representations can be used to mimic the physical phenomenon being modeled. The interactive nature of scientific visualization provides for a variety of tools to dissect and traverse large data sets. In terms of scientific programming, these visualizations also prove essential to debugging a large application and can provide a means for interactively steering a computation. This application of computer graphics is becoming increasingly valuable to a wide range of sciences and disciplines [4].

AVS is a commercial software product developed by Advanced Visual Systems Inc. for interactive scientific visualization. It is flexible, modular, extensible and executes on a wide variety of platforms. Views in AVS are constructed using a graphical network editor that allows the user to visually "program" a view by connecting individual components or *modules*. The underlying programming model is data flow where raw data is injected at the top of the network, flows through filter and mapper modules, and is finally displayed in a 2D or 3D rendering module. AVS includes hundreds of predefined modules such as orthogonal slicers, isosurface generators, and color mappers. Users may define their own modules which, once compiled, look and act just like any of the AVS-supplied modules. The library interface to AVS described here consists primarily of a user-defined module.

AVS supports the usual data types of integer and floating point with scalars and arrays. However, the fundamental input data type is an AVS field. A field is an abstraction of array data with geometric specification. The library described here provides automatic conversion between an array of data in the parallel program and a field of data in AVS.

2.3 Why PVM and AVS?

The goal of this library is simple and straightforward and it is therefore reasonable to ask if other combinations are possible? In particular, is MPI suitable for such a library? Can other visualization systems be used? Is there an existing integrated system?

The emergence of MPI [5] as a proposed standard elicits questions about the suitability of other message passing systems. MPI and PVM both provide message passing paradigms, but there are significant differences that make PVM more suitable for this library. First, MPI uses a static process model. This would mean that the visualization front-end would always be present and instantiated when the parallel program runs. Consider running a large parallel application on an Intel Paragon with AVS on a local SGI workstation. To enable visualization, a process must begin on the workstation at the same time that the parallel process is started on the Paragon. This is cumbersome for long-running applications where a user may not want the visualization process attached to the parallel computation for the entire run. In PVM, a long running parallel program may be started on the Paragon and when visualization is needed the user's workstation can be attached to the virtual machine and the connection made. Another significant restriction with MPI is that no existing implementation of MPI allows for heterogeneous communication between, for example, a Paragon and an SGI or any other workstation. The dynamics of PVM and its process control are necessary. Practically, no MPI implementation exists in the configuration that we need.

AVS was chosen as a convenience. It is widely used and not prohibitively expensive. However, any visualization tool that provides user extensibility and an interactive interface could be used. The authors know of no other integrated solution that presently suits these needs. Combining PVM and AVS for this library was a natural choice, taking advantage of the best features of both systems. Certainly, AVS by itself doesn't supply low-level message passing, even though it supports execution of different modules in a view network on different workstations. In addition, literally thousands of existing programs are written in PVM, so this library allows a dynamic connection to a powerful GUI with minimal changes to existing parallel codes.

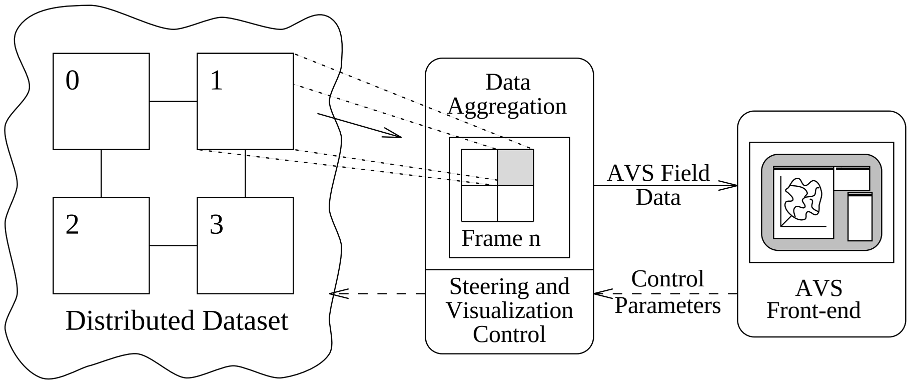

Figure 1: Overview of the library functionality

3 Library Design and Frame Construction

This section describes the high-level programming interface and the construction of animation frames by AVS. The library can handle one, two, or three-dimensional arrays of floating point data. The data might represent, for example, pressure, temperature or density. Figure 1 shows an overview of the logical organization. At each time step, every node sends data to AVS to construct a visualization frame, and a sequence of frames make up an animation. The library handles the task of aggregating the data into an AVS field and then injecting the field into the AVS data flow network. Notice that this interaction is two-way, and steering parameters can be sent back to the parallel application at each time step.

There are two distinct pieces to the library: the visualization stub and the AVS module. The visualization stub is a set of functions compiled into all of the nodes running on the computational back-end. This stub handles all the down-sizing and packing of data to be sent to the AVS module. The AVS module gathers the data from all the nodes and assembles the array into an AVS field. If any parameters have been modified from the AVS interface, that information is broadcast back to all of the processors. Data is sent at the end of each iteration from the parallel program through the stub to the AVS module to construct a frame.

The library is as generic as possible and is independent of the specific data decomposition boundaries. Figure 2 shows one possible decomposition and illustrates the computational region, visualization region, and cell sizes. In this figure, the overall 3×3 processor grid shows the full computational region. Notice that the visualization region is a subset of the computational domain and can be a coarser grid relative to this domain. The maximum extents of the computational grid must be known by all processors and are sent to the module through the stub. No units are used on the computational region, instead the array size in each dimension ($0 - (N-1)$) is used to specify all coordinates. The visualization region's extents are specified in terms of the computational grid. The term "cell size" is used to specify the resolution of the visualization region. The idea behind this is that a visualization cell represents data from one or more computational grid points. A cell size of n indicates that every Nth data point is sent to AVS for visualization. A different cell size can be specified in each coordinate so that differing spatial resolutions are possible.

The current restriction on data decomposition is that each processor must hold contiguous blocks of data in relation to the computational grid. Cyclical and other non-contiguous mappings could be handled, but would require extra information and would slightly complicate the code that assembles the distributed data set into a single visualization frame. Since cyclical distributions are not currently supported, the AVS front-end requires only the number of nodes holding the distributed data, not the specific decomposition used.

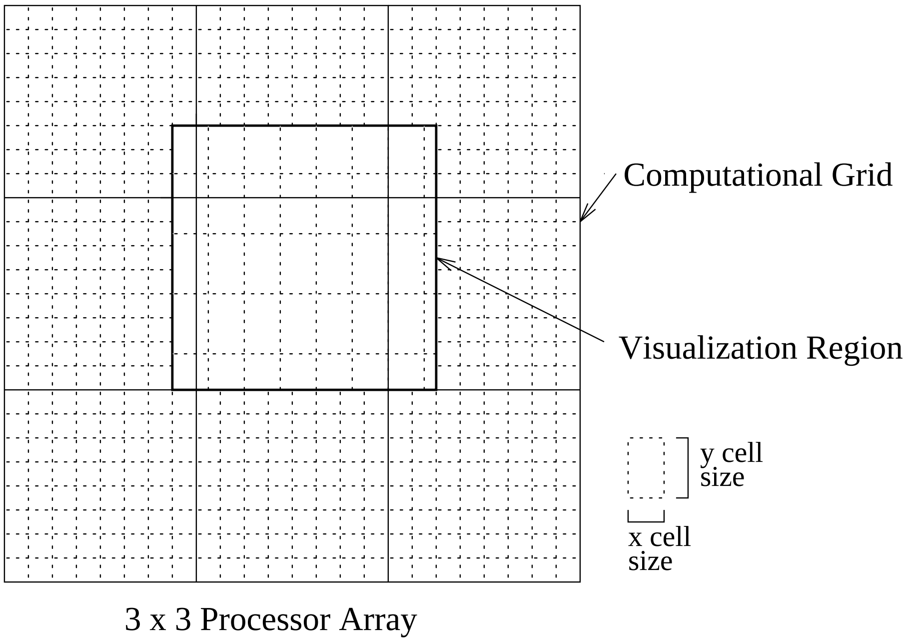

Figure 2: Data Decomposition and the Visualization Region

3.1 Independent Startup

The visualization front-end may start before or after the parallel computation has begun. The logic in the visualization stub is such that if the AVS module is not present then the parallel program is allowed to continue. The presence of the AVS module will be queried each time the parallel program enters the stub at the end of each complete iteration. If the AVS module has exited, the parallel program will not be blocked. In this way, communication with the AVS module can be started and stopped interactively. Thus, the parallel program may run with or without the visualization front-end actually being present.

3.2 Synchronization

The parallel processes and the AVS module need to be loosely synchronized because the steering and visualization parameters sent from AVS to the nodes can change at any time. Both the module and the stub maintain a global *sequence number* that is initialized to zero at start-up and increments each time a process sends its subset of data to AVS. It is designed so that each process keeps a locally-updated copy of this global counter instead of sending a message to retrieve the current value. Flow control is employed to keep the front-end from being swamped with data, insuring that no two nodes have copies of the sequence number that differ by more than one. The sequence number is used as a synchronization device so that the module can construct a valid frame at a particular sequence number without a barrier-type synchronization of the parallel code. To maximize efficiency, the rendering of frame N occurs in parallel with the computation of frame $N + 1$.

3.3 Dataset Down-sizing

Dataset downsizing is processed in parallel on each node. For example, if the x-cell size is four, then only every fourth point in the X direction is sent to AVS. If a processor's portion of the computational region does not overlap the visualization region, then no data is sent. However, in this case, a message must still be sent to AVS to indicate that the process has completed the iteration.

Sequence Number
Parameter Sequence Number
Number of Nodes
Dimension of Computational Region
Extents of Computational Region
Extents of the Visualization Region
Extents of the Visualization Subregion
Cell Size Vector
Number of Parameters
Parameter 1
$\vdots$
Parameter j
Number of arrays
Dimension of array 1
Extents of array 1
array data - packed by column
-1
$\vdots$
Dimension of array k
Extents of array k
array data - packed by column
-1

Figure 3: Data Message header sent by each processor to the AVS module

4 Low-Level Protocol Details

This section explains some of the low-level protocols that implement the features described in the previous section. There are four basic messages that must be interpreted: a data message from a node to AVS, a "transmit on" (XON) message from AVS to each node, a steering parameter message from AVS to the processing nodes, and notify messages from PVM to AVS or the parallel processing nodes to indicate that a process has exited the virtual machine.

Every data message contains header information (Figure 3) that describes its contents so that the AVS module can decode and assemble the data into a frame. The first entry is the global sequence number described in subsection 3.2. The parameter sequence number is keyed from the sequence number and is updated whenever a steering parameter has been changed in AVS. The remaining information tabulates the computational region, visualization region, cell size vector and the visualization subregion being sent by the individual processor.

The sequence number is used as the primary synchronization device. However, due to message latencies, it is possible for two nodes in the parallel code to receive parameter messages from the AVS front-end at different sequence numbers. For example, suppose node 1 has sent data for frame N, but node 0 has not. If the user decides to change the visualization region, then node 0 will have information about the new visualization region before sending data for Frame N. Node 0 would encode its data according to the new parameters (see Figure 4). Frame N has now been encoded differently by the two nodes and it is up to the AVS module to resolve this problem. The module simply discards any frames where there is a conflict in header information being sent from the nodes. For a frame to be valid, all nodes must send the same visualization region, computational region, parameters, cell size vector, and number of nodes. From a performance viewpoint, if AVS decides to drop frame N, then the animation stalls because a frame is not injected into the network. Frame $N+1$ will be constructed correctly since node 1 will retrieve the new parameters when it attempts to send this frame. The animation will then continue from this point. A design choice was made to simply drop frames with mismatched headers instead of trying employ synchronization each time new parameters are sent. The former is less intrusive to the parallel computation but means that effects of new parameters may not be seen immediately in AVS.

4.1 Program States

Each time the stub routine is called by a node process, all messages for controlling the visualization are read from the message queue. The node process can be in one of three states:

- Not Connected to AVS
- Connected to AVS – Cleared to Send
- Connected to AVS – Sending Blocked

If a node is not connected to AVS, it will query the virtual machine using PVM's name look-up capability to determine if the AVS module is running. If AVS is not found, the stub finishes executing and control is returned to the calling program. If the node is connected to AVS and is cleared to send data, then all pending AVS messages are interpreted to determine if parameters have been changed. Data is then packed according to the new parameters (if any) and sent to AVS. A local variable is set to indicate that an XON message is required before the next frame can be sent. Control is then returned to the calling program. In the last state, the sender is blocked until either an XON message from AVS is sent, or a notification message that AVS has exited is sent. Either message frees the program to proceed. Control is then returned to the calling program.

4.2 Parameter Sequences

One of the attractions of the library is that parameters can be updated from the AVS interface and sent to the parallel code. When a parameter is changed on the AVS interface, a message is immediately broadcast to all the nodes. Each stub will receive this message the next time its process sends a subset of data. The local steering parameters can then be updated on the node.

Figure 4 illustrates a race condition that is possible when non-explicit synchronization is used. In this figure, the parameter change message is detected by node 0 at the start of sequence 1. However, node 1 does not detect the new parameters until sequence 2. This could be catastrophic for the parallel code, especially if the code resets on new parameter information. This race condition motivates the choice for a separate parameter sequence number. The parameter sequence number is keyed from the AVS copy of the sequence number. To eliminate the race, the node processes do not interpret new parameters until their local copies of the sequence number equal the parameter sequence plus two. Notice in Figure 4 that the nodes and AVS will have consistent views of the parameters when the sequence number is two.

5 Examples

The application *fcloud* used for this example models an expanding spherical gas cloud emanating from a point source, The computational domain for this simulation covers a three-dimensional cube with 99 uniformly spaced points in each direction for a total of about one million data points. The application is written in Fortran using PVM and is distributed

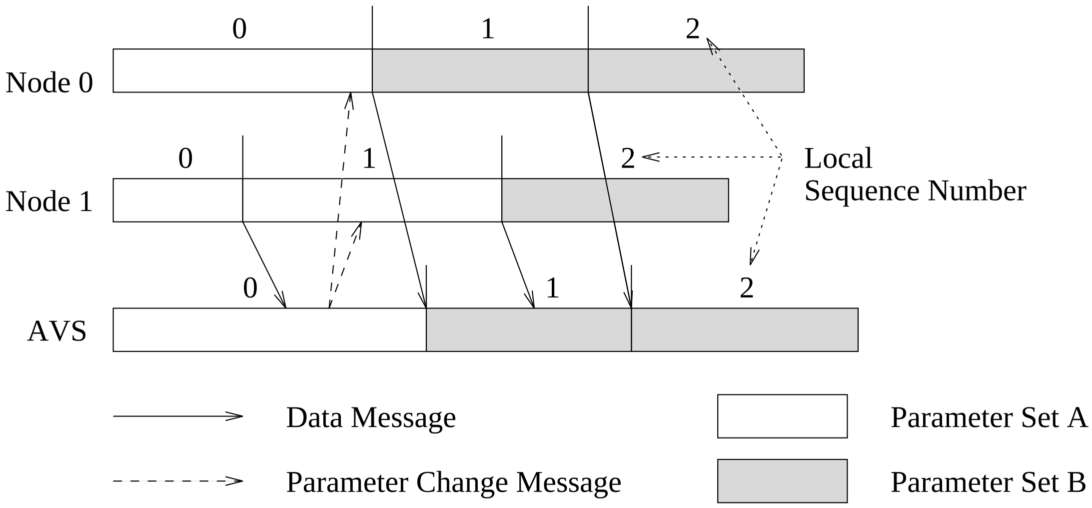

Figure 4: A possible race condition on parameter changes. Node 1 does not discover new parameters until its sequence number is 2.

over a network of workstations connected by ethernet. Several different workstation architectures are used, including SGI Indy and Indigo2, IBM RS6000 and SUN SparcClassic workstations. The data is distributed in a simple three-dimensional grid, with each of 9 processes computing on a $33 \times 33 \times 99$ cube.

The AVS visualization front-end is run on an SGI Indigo2. The AVS view constructed for this application uses the *pvm_read* module from the library to assemble the array data as an AVS field. The field created by the pvm_read module is injected into the network where system-defined modules are used to produce a three-dimensional geometry of data points, each represented by a small sphere. This network is shown in Figure 5. In addition, the field is also routed through an *isosurface* module to generate a constant density surface among the data point spheres. An *orthogonal slicer* module produces a two-dimensional plane that slices through the cloud to show internal data values. Such a slicer can be used to examine the density of the cloud impacting on a flat surface. This kind of information would be of interest in, for example, the depositing of a material onto a substrate.

The design of the AVS network is completely separate from the design of the parallel code. On initial synchronization, the AVS pvm_read module receives the computational domain from the fcloud processes and then proceeds to sleep, awaiting user input to start the simulation. This gives the user the opportunity to set the size of the visualization region and to reduce the amount of data to be transferred to a more manageable figure. To this end, the cell sizes in the X, Y, and Z directions are initially set to 4, as shown in Figure 6, to retrieve every fourth data point in each dimension. Therefore, only 15 thousand of the one million data points are sent for the entire region. For this initial setting the isosurface is set to display the zero-density surface around the gas cloud, and the slicer is set to the bisect the cloud.

After running the simulation for several iterations, the gas cloud reaches a significant size, and it becomes desirable to examine the detail at the expanding front of the cloud. The visualization region is adjusted to focus in on just a small portion of the sphere, an octant, with the cell sizes set to 1 for full detail. The result of this adjustment is shown in Figure 7. Note that this detailed perspective requires 125,000 data points be transferred.

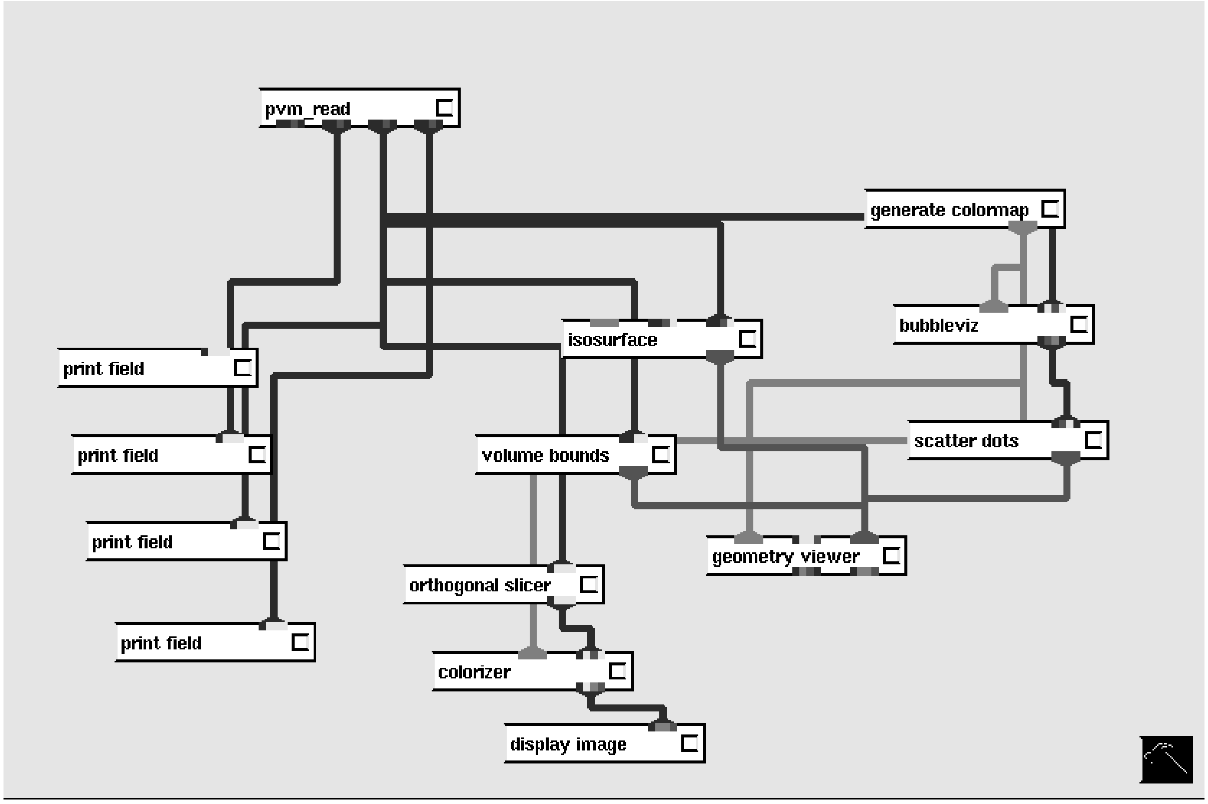

Figure 5: AVS View Network

6 Conclusions

In this paper, we have described a library that can be used to visualize and animate data that has been distributed over a set of processors. The library provides a simple link between a parallel code and a visualization package so that data can viewed as it is being produced. The library handles the details of communication and the assembly of the data set into an AVS field suitable for injection into a view network. Both the visualization region and its resolution can be specified interactively during run time so that a user may focus attention on particular areas of the data set. User-defined steering parameters are made available to support interaction with the parallel code using AVS as the front-end. The library is portable and supports the natural heterogeneity found when coupling massively parallel processors with desktop visualization hardware.

References

[1] Advanced Visual Systems, Inc. *AVS User's Guide*, 1992, Waltham, MA

[2] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1994.

[3] K.M. Chandy and J. Misra, *Parallel Program Design, A Foundation*, Addison-Wesley, 1989.

[4] J.A. Kohl, "Visual Techniques for Parallel Processing," University of Iowa, Department of Electrical and Computer Engineering, ECETR-910726, July 1991.

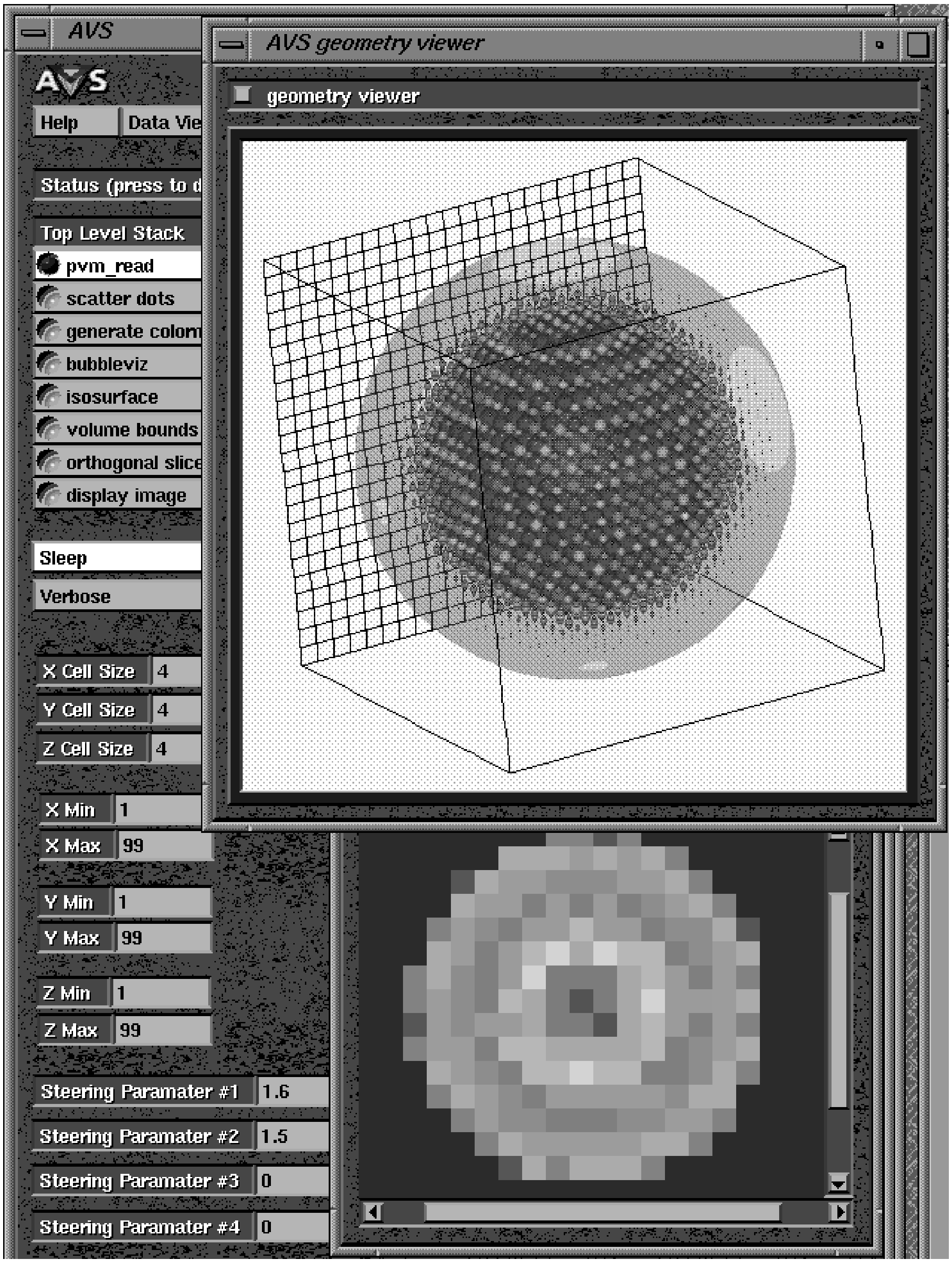

Figure 6: Initial View Setting

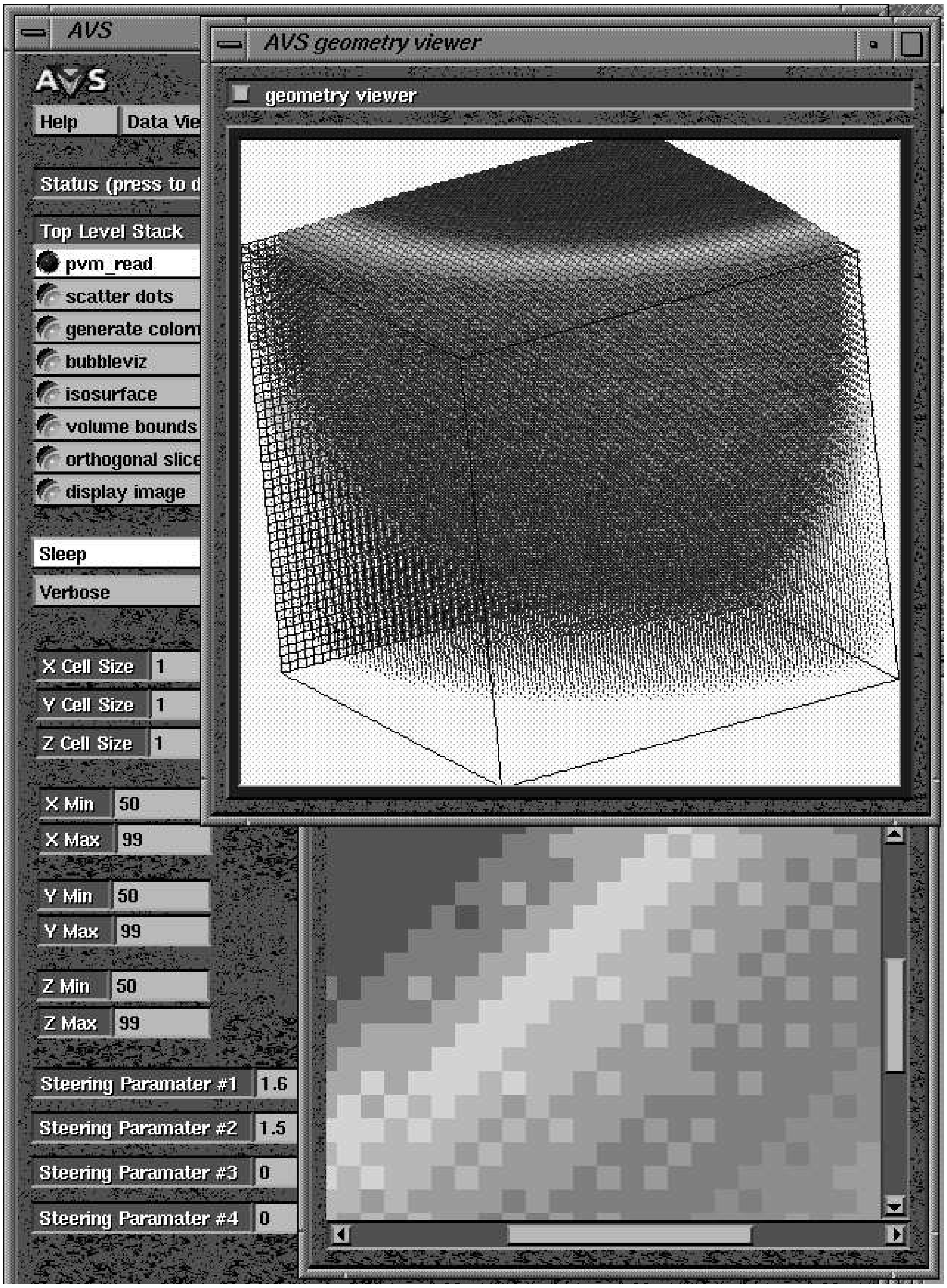

Figure 7: Detailed View Setting

[5] Message Passing Interface Forum. MPI: A message-passing interface standard. Computer Science Dept. Technical Report CS-94-230, University of Tennessee, Knoxville, TN, April 1994. and *International Journal of Supercomputer Applications, Volume 8, Number 3/4, 1994*.