Adapting AVS to Support Scientific Applications as Heterogeneous Distributed Programs

Patrick T. Homer
Richard D. Schlichting

TR 91-23

ABSTRACT

Most scientific applications are currently structured as a series of computational steps, each of which is implemented by a separate program with files being used to transmit data between the steps. For example, a vectorized computation with graphical output may involve executing the computation on a remote supercomputer, transferring the output file over the Internet, and then viewing the results on a local workstation. Here, an alternative model is described in which the application is constructed as a heterogeneous, distributed program in order to improve facilities for user interaction. In this model, the application is structured as a collection of interacting processes (tasks) in which distribution and heterogeneity of machine architecture and programming language are handled transparently by a remote procedure call (RPC) mechanism. The specific focus here is on describing how AVS (Application Visualization System) from Stardent Computer, Inc. has been adapted to support this type of application using the Schooner RPC system. An example involving self-organizing neural nets is used to illustrate the details of this approach.

September 26, 1991

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

This research supported in part by the National Science Foundation under grant CCR-8811923.

Adapting AVS to Support Scientific Applications as Heterogeneous Distributed Programs

1. Introduction

Current techniques for constructing, executing, and analyzing the results of scientific computations are usually inflexible and constraining. Typically, the overall problem is divided into a series of independent computational steps that are performed in sequence, with the output of one step being used as input to the next. In this paradigm, the steps are realized by separate programs that are manually initiated, and the one-way data flow is implemented by intermediate ASCII files that store the output from one program to be used as input to the next. For example, a modeling application may involve running a program on a remote vectorizing or parallel machine with the numerical output placed in a file, transferring the file over the Internet to a local workstation, and then invoking a graphics tool to view the results. If, after looking at the results, a researcher wants to use a different input parameter, the entire process must be repeated from the beginning. In essence, scientific applications today are written using a ''file-centered'' programming model that has been inherited largely unchanged from the batch models supported by earlier operating systems.

In this paper, we describe an alternative model in which a scientific application is written as a *heterogeneous, distributed program.* In this scenario, the application is constructed not as a series of independent programs that pass information using files, but rather as a collection of processes (tasks) that interact using interprocess communication. The distributed aspect of the program comes from the ability to execute portions of the program on multiple machines connected by a network, potentially across large geographic distances; the heterogeneity aspect comes from the ability to have the portions execute on architecturally-diverse machines and to be written in different programming languages. This functionality is implemented by a software subsystem that realizes distribution and heterogeneity in a manner that is virtually transparent to the user.

This alternative model gives the user a degree of flexibility not easily realized in the old model, especially with regard to improving facilities for user interaction. For example, it enables the user to modify input parameters and see the results of those changes directly in the output without having to reexecute a series of programs. It also becomes possible to monitor the progress of a long run by viewing intermediate results; this allows for modification of parameters during execution or for aborting the run in a situation where an intermediate picture makes it clear there is no point in waiting for the final results. Intermediate images may also lead to insights into the nature of the problem that would not occur from only looking at the final image, such as the rate of convergence or the existence of transient fluctuations.

Our specific focus in this paper is on outlining how we have adapted AVS[1] (Application

Visualization System) to support the construction of scientific applications as heterogeneous, distributed programs. The adaptation is based on using AVS's existing facilities for user-defined extensions together with a remote procedure call (RPC) system called Schooner. AVS provides a state-of-the-art graphics system for viewing the results of scientific computation, while Schooner implements transparent distribution and heterogeneity. Schooner is, in turn, based on a previous system called MLP [Haye87, Haye88, Haye90].

This paper is organized as follows. In Section 2, we give brief overviews of both AVS and Schooner. Section 3 then describes how AVS has been adapted to support this new model, with an emphasis on constructing applications using this technique and how transparent communication is realized using Schooner. An example application involving self-organizing neural nets is outlined in Section 4. Finally, Section 5 offers some conclusions and directions for future work.

2. The Individual Systems

2.1. An Overview of AVS

AVS is a graphics system for displaying images generated by scientific computations [Star91]. The data model is oriented strongly toward these types of applications, with an underlying assumption that the data represents a two- or three-dimensional grid with values located at each point in the grid. The grid need not be uniform; AVS also supports rectilinear and irregular grids. The values at the grid points can be scalar or vector values, where the vector can be of arbitrary length.

AVS reads data files and provides a large palette of tools for manipulating the data and the resulting images. These tools allow the user complete freedom in determining the appearance of the data. For example, the user can display all of the data or only a portion. If the data is symmetrical, AVS provides mirror tools for reflecting the image through an axis of symmetry. For volumetric data, the user can display slices through the volume. Color can be used to display variations in values across a surface.

AVS also provides for direct manipulation of the image itself. For example, the objects in the image can be moved and rotated, and the viewing camera can be moved and rotated about the object. Various lighting models are also provided and can be combined in a number of ways. The positions and types of the lights are under the control of the user as well, which allows adjustment of the model to highlight the important features of the image.

Computations are performed in AVS by pieces of code known as *modules.* Each module typically acts as a filter, receiving data from other modules, transforming it in some way, and passing it on to other modules. These connections are specified graphically using a visual interface called the *Network Editor.* This editor allows a given module to be chosen from a menu, dragged into the correct position, and connected to other modules to indicate the data flow. The resulting network of modules realizes the entire process needed to render the image.

[1]AVS is a trademark of Stardent Computer Inc.

The data flow and scheduling of module execution is implemented by the AVS kernel. A module is considered to be ready for execution if new data is waiting at one of its input ports as the result of execution of some other module, or if one of its input parameters has changed. These input parameters allow the user to interact with the module, usually by manipulating the visual interface in some way. For example, a module that shows a slice through a volume would have controls to specify the position of the slice within the volume, a mirror module would allow specification of the axis of symmetry, and a module that reads a data file would have a file browser control in which the user clicks on the desired file.

Limited facilities for concurrency are provided in the form of what AVS refers to as co-routines. These are modules that can run independently of the other modules; specifically, such a module can execute without waiting to be explicitly scheduled by the AVS kernel. While providing some concurrency, it is also possible for data to be lost should the co-routine run too fast. Also, co-routines provide only limited ability to execute an actual distributed or concurrent computation and provide the results directly to AVS.

AVS also provides support for executing standard modules (i.e., not co-routines) on other machines in a local-area network. However, unlike our approach, each such remote module must be executed on a machine that supports AVS and must be written to conform with the standard data-flow AVS programming model rather than a general procedural model. Moreover, AVS requires a separate module be present for each remote machine that an application may use in a computation, with no provision for modules that can dynamically select which machine to use at runtime.

2.2. An Overview of Schooner

Schooner is an RPC system designed to facilitate the construction of heterogeneous distributed programs. To accomplish this, Schooner provides three services: an external data representation, a procedure specification language and associated stub compilers, and a runtime system to implement control flow. The data representation and specification language are combined into a type language called UTS (Universal Type System) [Haye89].

The external data representation aspect of UTS allows data to be represented in a machine- and language-independent manner. It includes most common data types found in languages, plus full support for array and record types. Library calls are provided that convert data from the host machine’s native format to and from the UTS format. In addition, UTS supports a *represented type* whose representation is under the control of the user. This allows, for example, the passing of arguments that are not standard within a given language or whose size or type are not completely known until runtime.

The specification language portion of UTS is used to specify the interface—essentially, the number and type of arguments—for each procedure that can be called remotely in the application. There is both an import specification and an export specification; the import specification is associated with each file that invokes the procedure and the export specification is associated with the file containing the code for the procedure. Typechecking is done at program initialization time to ensure that the specifications are compatible.

The specification file is also used as the basis for automatic generation of stub procedures for each imported and exported procedure to handle conversion of argument values between the language- and machine-specific format and the UTS data representations. The stub handles the conversions for all values automatically except those values whose type is represented. There is one stub compiler for each supported programming language. Currently, Schooner has stub compilers for C and FORTRAN; various versions of the predecessor MLP system also supported Pascal, Icon [Gris83], and Emerald [Blac86, Blac87].

A program using Schooner is formed out of multiple *components*, each containing one or more procedures. A component is implemented by a process, and so is also the unit of distribution in the system. Control flow is based on a sequential, procedural programming model in which there is logically one thread of control that transfers between components when it executes an RPC. Such a call proceeds as follows. The stub first encodes the arguments for the procedure into a message. The message is then sent to the component containing the desired procedure; should this be the first time the component is contacted, it is first located using facilities in the Schooner runtime. The receiving stub procedure is then passed the message, which it reads to retrieve the arguments and call the invoked procedure. When the procedure completes, the stub encodes the return values and sends the reply message to the component containing the calling procedure.

In addition to the components, there are two other types of processes involved in Schooner: a manager process and one server process per machine. The manager performs three basic functions: component initiation, procedure name lookup, and program termination. Initiation is done by sending a request to the server process on the appropriate machine. The startup protocol for a component also registers the names of its exported procedures with the manager, which stores the mapping from procedure name to machine and component information for use with subsequent procedure lookup requests. The manager handles program termination by sending termination messages to each component; this will occur either when the main routine exits, when an error occurs in a component forcing that component to quit, or when a component times out waiting for a message. In addition to handling initiation requests from the manager, a server process also stores the location of the manager to pass on to components that start independently. This last service was provided explicitly for use with AVS, although it should prove useful in other contexts as well.

There are many other RPC schemes with features such as external data representations, specification languages, and stub compilers that are similar to Schooner [Alme85, Birr84, Sun88, Xero81]. Several of these systems also emphasize heterogeneity, including Matchmaker [Jone85], Horus [Gibb87], and HRPC (Heterogeneous RPC) [Bers87]. The primary distinction between this work and Schooner is one of orientation: the main aim of the other systems is to support interprocess communication for client/server style operating system services, whereas Schooner is intended for building user-level applications. This emphasis is reflected in such features as our support for arbitrary calling structures (e.g., recursive procedures), the flexibility of the type scheme, and the relatively straightforward way in which it can be used. Another related system is Polylith, in which heterogeneous software modules can be plugged into a ‘‘software bus’’ that allows communication [Purt91a,Purt91b]. We note in passing that most of these systems could also have served as the basis for the adaptation outlined in this paper, albeit at the cost of requiring somewhat more effort on the part of the programmer.

3. Adapting AVS

As outlined in the Introduction, the goal of this work has been to adapt AVS to support the writing of scientific applications as heterogeneous distributed programs. To do this, we provide facilities for invoking external computations from within AVS; these computations may be performed on machines other than the one being used to run AVS, may be written in a different programming language, and may, in fact, use a drastically different programming model than AVS (e.g., massively parallel) to arrive at an answer. In other words, AVS figuratively sits at the end of a chain potentially consisting of multiple computations executing on multiple diverse machines, where Schooner provides the software infrastructure to connect these other computations and machines to AVS, taking care of any variations in machine architecture, data representations, etc. This new model also supports the connection of multiple computations to AVS at the same time; this provides multiple streams of data into an AVS network, thus allowing AVS to combine data from more than one source into a single image. The difference between the traditional file-centered use of AVS and our use of the system to support applications as heterogeneous programs is illustrated in Figures 1 and 2.

3.1. Constructing Applications

While providing for a new application model, we have not altered substantially either the way in which the user exercises control over AVS and the application during execution, or the way in which the application is constructed. For the former, the general AVS interaction model remains unchanged. Essentially, it appears to the user as though the external computation is

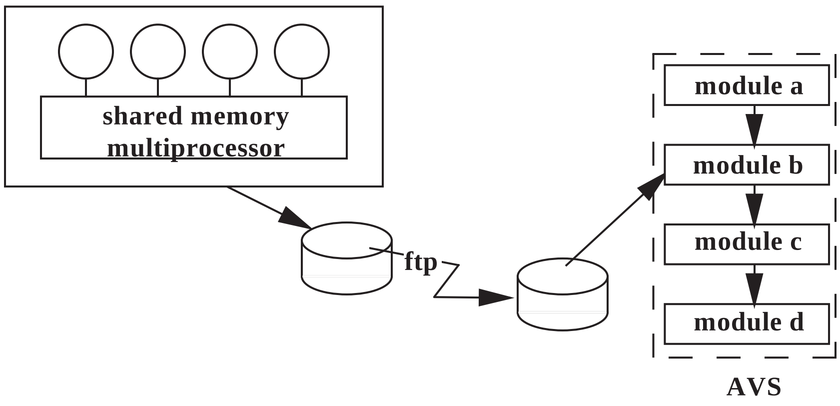


Figure 1 — Traditional AVS Model

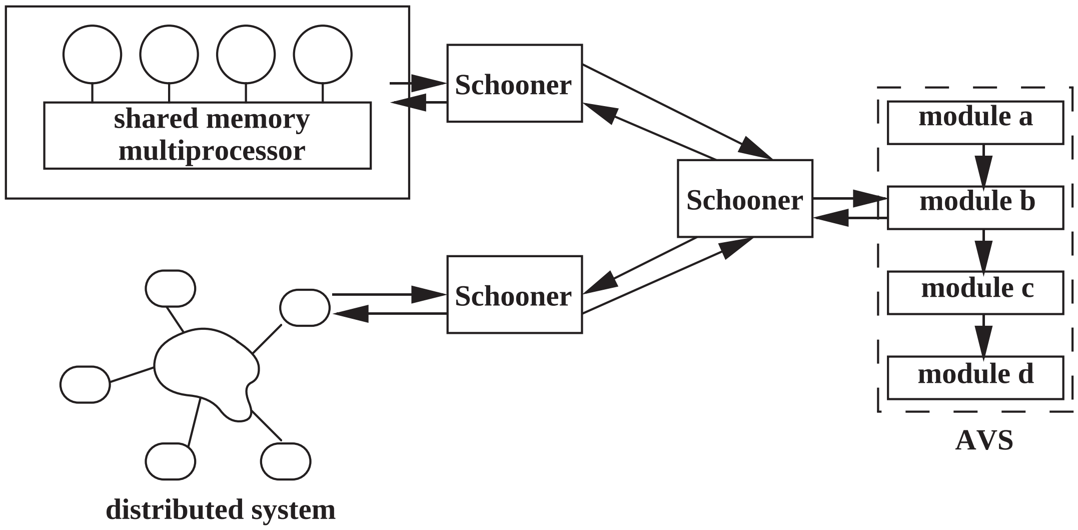


Figure 2 — New Application Model

running within AVS, with control being exercised using the normal AVS interface. Of course, extra features are typically added to implement the added user interaction that is the motivation for much of this work. For example, controls would need to be added to execute the external computation successively with new parameters without leaving AVS to create additional data files. However, the net result is that the fundamental ‘‘feel’’ of AVS has been retained even while extending its capabilities to support a fundamentally different application model.

The process of building an application for the new system also shares many similarities with the techniques used for standard AVS applications. For example, the way in which AVS’s graphics processing facilities are used is not affected. AVS’s extensive collection of tools for building application interfaces also remains available. These include ‘‘widgets,’’ which automate to a large extent the process of taking input from the user and passing it to a module by, for example, checking the input against upper and lower ranges. Widgets are displayed to the user as dials, sliders, type-in windows, file browsers, etc.

In order to make use of our technique, a few additional steps must be taken to integrate what had been a standalone program into AVS. The first is to modify the computation to change it from a program into a procedure. For the most part, this simply involves changing the main routine into a procedure and accepting as arguments to this new procedure appropriate control values. The procedure will also need to return those values the user wishes to display rather than outputting them to a file. Generally, these returned values will be one or more arrays containing values calculated at the grid points in the problem. For problems involving irregular grids, both the coordinates and the values have to be returned.

The second step involves writing a new module to generate the data values to be displayed by invoking the computation procedure directly. The structure of this new module can be broken down into three basic tasks: collecting input to use as argument values for the procedure, invoking the procedure to perform the computation, and outputting the results. All are usually straightforward. The input is typically done using the widget mechanisms described above, while the procedure is invoked using the language's standard procedure call mechanism; as described more fully below, Schooner takes care of actually locating and invoking the (possibly remote) procedure, as well as handling the details of performing any required data conversions. Output is accomplished by passing along the results from the procedure invocation to subsequent modules that will process it for display purposes. The one possibly complicated part of this process occurs if the data being produced does not fit into the standard AVS data model; in this case, it would be necessary for the user-written module to transform the data, for example, by producing a geometry that can be rendered by AVS. Note, however, that such a transformation would have been necessary even if the data had been read from a file.

The final step for the user is to write two UTS specification files, one for the procedure performing the computation and the other for the AVS module. These two sets of specifications, which are virtually identical, describe the structure and type of the arguments needed by the procedure. As explained in Section 2.2, these specification files are used by the Schooner stub compilers.

Although the above are all that are *required* to execute an application using our scheme, it may be beneficial to consider restructuring the computation. In particular, if the execution time is long, it might better be modify the computation to be a procedure that returns intermediate values. In this model, the procedure retains values between calls and will, on each call, proceed for some number of iterations before returning results to be displayed. Given this structure, the user must also decide if the module containing the call should be a standard AVS module or a co-routine. If it is a standard module, it is scheduled in turn with other modules and so will be idle when other parts of AVS are running; if designed as a co-routine, the module is scheduled independently and so can execute concurrently with the rest of the system. The co-routine choice is particularly appropriate when the execution time of each set of iterations is long compared with the time required for AVS to render each image.

3.2. Implementation

The role of Schooner in this adaptation of AVS is to connect together the new AVS module and the procedure realizing the external computation. The stub compilers read the specification files, producing the necessary stub procedures. These stubs and the user written code are then combined and linked with the Schooner libraries to produce two components, one becoming the AVS module and the other becoming the computation component.

Initialization of the application now occurs in two stages. First, the Schooner manager process is started with the name of the computation component as an argument; this value is used to either start the component directly or as an argument in a request to the server process on the appropriate machine to start the component. The manager process then acquires the procedure export information from the computation component, and registers itself with the Schooner server on the machine where AVS will be executed.

The second stage occurs when AVS is started and the Network Editor is used to create the network containing the new module. When the module is instantiated by placing it in the network, it contacts the Schooner server on its machine and obtains from it the information about the location of the manager process. The module will then contact the manager process, sending information about any exported procedures it has.

When the module is scheduled by the AVS kernel for execution, it will attempt to invoke the external computation using a standard procedure call. At this point, Schooner takes over and transparently transforms the call into an RPC. As described in Section 2.2, this proceeds by the stub encoding the procedure arguments, which are marshalled into a message that is sent to the component containing the requested procedure. After being received, the message is passed to the procedure stub where the arguments are decoded and the actual computation procedure called. When the procedure returns, the stub encodes the return values and sends them back to the calling component as a reply message. These values are then decoded by the stub on that end, and passed back to the AVS module.

4. An Example: Self-Organizing Neural Nets

To illustrate these ideas, we took a scientific application that required substantial execution time to reach convergence and integrated it into our adapted AVS system. The goal here was to improve user interaction with the application. In particular, the user wanted to be able to monitor the progress of the application by viewing intermediate results and change parameters while the application is running, as well as to view the final results.

4.1. Application Description

The application is a neural net code that implements a short-cut version of the Kohonen self-organizing neural network using Gaussian-type interconnection strengths and a conjugate-gradient learning algorithm. The Gaussian function uses a periodic norm to measure distance. Points representing processing elements fall within a two-dimensional unit square and are reflected back inside the square if the calculation tries to place them outside. The inputs to the program include the size of the problem, the amount of randomness in the initial configuration, the half-width of the Gaussian function (which affects the learning curve of the network), and the desired number of learning iterations. There are actually several versions of the code, each of which is tuned to the particular architecture on which it executes. For example, the version that runs on a Sequent Symmetry is a parallel solution that exploits the multiprocessor architecture of the machine. All of the versions are written in C.

Once the program has reached convergence, a plot with the points connected to each other will show uniformly spaced vertical lines. A similar plot done at the start of a computation will show random lines zigzagging across the space. Plots made during the run will give a feel for how the problem is approaching convergence. A substantial number of iterations are typically required to come close to the final uniform-spacing solution; for example, in one run involving 32 x 32 points, 100,000 iterations were required to get a reasonably good solution.

Given the long-running nature of this application, the primary motivation for using the new programming model was to be able to monitor its progress during a run. To do this, options were provided for steering the program, including provisions for halting the run and for changing the parameter representing the half-width of the Gaussian function in mid-stream.

4.2. Integrating the Application into AVS

To integrate this program into AVS, only a few modifications along the lines of those detailed in the previous sections were required. First, several changes were made to the application. The most obvious was changing the main program into a procedure that took the various parameters as inputs. At the same time, we also modified the program to dynamically allocate array space rather than relying on a compile-time constant; this allowed the size of the problem to become a controllable input parameter. Next, the computation was restructured to return intermediate results in the arrays at the end of a certain number of iterations; this number is specified as a parameter at runtime. One implication of returning intermediate results in this way is that the state of the computation—primarily the current array values—had to be stored between calls to allow the calculation to proceed from that point in response to the next call. Since the arrays are now dynamically allocated, it was sufficient to make the array pointers global within the component. A check is done on each procedure call to determine if the call is a continuation or a restarting of the problem. If a restart, the dynamically allocated memory from the previous calls is first released, then new memory is allocated; if a continuation, the calculation proceeds using the retained values. All of these techniques were applied to both a sequential version of the neural net code and a parallel, shared-memory version.

The second step was to write the new AVS module to replace the one that previously read the data values from a file. This module is designed in the normal way that AVS provides. Typically, AVS user-written modules consist of initialization and destruction functions, a compute function, and a spec function that describes the interface widgets and identifies the names of the other functions. This module is a co-routine, so it does not contain a compute function, but rather has a main program instead; as described above, this allows the module to run as a completely independent process without being scheduled by the AVS kernel.

The AVS spec procedure is involved primarily in setting up the various widgets needed for the application. In this case, there are widgets for each of the input parameters to the application, *num_points*, *side*, *alpha*, *start*, and *increment.* There is also a widget which allows for stopping/resuming/restarting the application. A final widget is currently in the process of being implemented. This widget will be a simple toggle that allows the user to have the intermediate results saved in files as well as displaying them. This will allow the user to examine any stage of the results at some later point without the need to rerun the code. Since the neural net code generally (depending on the requested size of the problem) takes considerably longer to iterate to the next intermediate result than AVS requires to render the image, our tentative decision is to have the module in AVS be responsible for performing the actual file writes in order to avoid slowing down the computation. From an implementation point of view, this could just as easily have been done at the other end by the procedure performing the computation.

The module initialization routine involves calls to two library routines that transfer information to Schooner. The first specifies the location of the Schooner manager process. In many cases, this will be running on the same machine as AVS, however, it is also possible to specify a different location should that be more appropriate. The second library routine is provided for those applications that can run on a variety of platforms. This call allows the user to specify a list of machines that can run the user's code, along with the pathnames on those machines for locating the code itself. The manager, in response to this library call, will query each of the listed machines to determine if the Schooner server is running, and to inquire of the server if the specified executable code is present. A widget consisting of radio buttons will appear for the module, with a button for each machine from which a positive response was received. If only one machine is specified or only one machine is found, this machine is the default machine and no machine widget appears for the module.

The destroy procedure in an AVS module is invoked when the module is removed from the network. In this application, this routine consists of a call that notifies the Schooner manager process about the shutdown of the module. This allows the manager to gracefully shut down any other processes started in the course of executing the application, and then to terminate itself.

For this particular application, it was also necessary to add a second part to the new AVS module. The data produced by this application consists only of coordinates of points in an irregular grid with no actual data values at those points. AVS does not have modules in its standard module library to deal with showing points with no values, so code was written that took the coordinates generated by the co-routine described above and created a geometry showing the points as spheres and connecting them with lines. A widget is provided to control the radius of the displayed spheres. We are in the process of moving this functionality into a second new module that will receive output from the first one using standard AVS data-flow facilities.

The final step in performing the integration was to write the UTS specification files. The export specification for the computation procedure, which is associated with the component containing that procedure, is as follows (the import specification is analogous):

```
export net prog(
    "num_points" val integer, "side" val float, "alpha" val float,
    "start" val integer, "increment" val integer,
    "rep_x" res rep array[-] of float, "rep_y" res rep array[-] of float)
```

Here, *net* is the name of the procedure. The parameters *num_points*, *side*, and *alpha* control the neural net; *num_points* is the size of the problem space, *side* is the initial randomization factor, and *alpha* is the half-width of the Gaussian function. *start* and *increment* specify the current iteration number and the number of iterations to proceed before returning values, respectively. *rep_x* and *rep_y* contain the x and y coordinates of the points and are the result of calling the procedure.

The keyword **val** before the type specifies that the associated argument is passed by value, while **res** indicates a result parameter. In this example, arrays are returned as result parameters; this reduces communications load relative to specifying them as variable parameters since it means that the data only need be transmitted in one direction. Note that this is a useful by-product of retaining the state of the neural net in between calls. The specification of *rep_x* and *rep_y* as **rep array** indicates that these parameters are represented types as described in Section 2.2; this is done so that the problem size can be an input to the procedure. This choice requires a small amount of additional work on the part of the programmer since UTS library routines must now be used to encode explicitly the array values into UTS form in the user AVS module and subsequently decode them in the procedure. To illustrate this, the following code was used to encode the values for the *rep_x* array:

```
long rep_x;
size_squared = num_points * num_points;
/* space for array plus header */
user_rep_new(rep_x, size_squared * (sizeof(double) + 1) + 50);
sch_start_array(rep_x, 1 , size_squared);
/* put each value into the represented array */
for (i = 0; i < size_squared; i++)
       sch_encode_float(rep_x, (double)x[i]);
sch_end_array(rep_x);
rep_fixup(rep_x);
```

The code to perform the other encodes and decodes is analogous. If the programmer prefers not to do this explicit conversion, fixed-size arrays could have been used for *x* and *y*. It would still be possible to have the problem size be a parameter, but the chosen array size would now be an upper-bound on the problem size.

The one additional aspect of the integration to be discussed is the choice between making the new module a co-routine or a standard module. As alluded to above, we decided to make it a co-routine. This allows the module to execute continuously without having to wait for new data or a change in widget state as would a standard module. This choice provides some concurrency to the computation, since the module can initiate calculation of the next series of iterations at the same time that AVS is rendering the image from the previous series. However, it also means that the computation cannot be halted precisely on demand, but only at intermediate stopping points. The user requests this action by toggling a stop widget that has been added to the interface, which will halt the computation following the completion of the currently executing series of iterations. The alternative would be to require the user to explicitly toggle a widget to initiate each series of iterations.

Actually executing the application is straightforward. First, the Schooner manager process is started. Then, AVS is started and the desired network is created using the standard Network Editor. When the user-module is dragged from the menu into the edit space, the initialization routine is run and the various widgets are displayed. This is also the point at which the module establishes contact with the manager and the queries to the various requested machines are made to determine which ones are available. Once the network is complete and starts the application running, the co-routine nature of the module becomes apparent as it begins executing and passing results on through the network. The Appendix gives a snapshot of the screen during the execution of this application.

4.3. Experiments

The neural net application using our adapted version of AVS has been tested on several different hardware configurations and with two different versions of the computational code. The most extensive tests were carried out at the Advanced Computing Laboratory at Los Alamos National Laboratory (LANL). In this case, the sequential version of the neural net code executed on a Sun SPARC 2, with AVS running on an FPS-350. The machines are located on different local area networks connected over a space of a few miles by a T1 line.

A second configuration that was tested had a beta-test Sun version of AVS executing on a SPARCStation at LANL, but with the actual computation being performed on a Sequent Symmetry S81 at The University of Arizona. In this case, a parallel version of the algorithm was used to take advantage of the multiprocessor architecture of the Sequent. This experiment demonstrates both the feasibility of performing RPCs across relatively large geographical distances to execute applications, and the way in which our adaptation of AVS is suitable for computations using a different programming model. Performance studies designed to measure RPC and data conversion overhead in both application configurations are just beginning.

5. Conclusions

In this paper, we have described how we adapted AVS to support the construction of scientific applications as heterogeneous, distributed programs. There are many advantages to this approach, with perhaps the most important being that it makes it feasible to construct applications with significantly better user interaction facilities. This adaptation was realized by replacing the module that reads display data from a file in a standard AVS network with a new module that uses RPC to invoke a procedure to perform the actual computation. The interprocess communication implicit in the call plus the data conversions required by heterogeneity are handled transparently by the Schooner RPC system. The feasibility of this approach has been demonstrated with two versions of a neural net application, one of which is parallelized to improve performance.

It is worth emphasizing that adapting AVS for this use was straightforward, largely because we were able to exploit the extensibility of AVS—albeit in a way perhaps not envisioned by its designers—rather than having to modify the system itself. This extensibility coupled with the facilities provided by Schooner also means that actually using the system to construct applications is straightforward as well. Whether these properties plus the benefits of improved interaction capabilities are sufficient to entice scientific researchers to adopt this application model will only be determined by further experimental evaluations.

Our future efforts in this area will be directed along several lines of investigation. The most immediate is continued testing of the neural net application with, as noted in Section 4.3, the goal of determining the performance of our approach. In the process, we also intend to experiment further with interaction to determine what additional facilities might be desirable.

A second, related line of investigation will focus on the combination AVS/Schooner system independent of a particular application. One possibility here is to develop a ‘‘module compiler’’ that would automatically generate the new AVS module required in our scheme based on a specification written in an extended version of the UTS specification language. The specification would provide the arguments to the remote procedure call, along with information pertaining to the type of widgets to be used for each input parameter, the output of the module, etc.

Finally, we intend to look at more long-term issues concerning the development of scientific applications as heterogeneous, distributed programs. This will encompass investigations of both application-level programming issues and the question of the best form of system support for this activity. For the former, we will examine such items as the usefulness of having interprocess communication primitives other than RPC available (e.g., primitives for asynchronous message passing) and ways to utilize existing codes in this model without modifications. For the latter, we will look at utilizing other graphics visualization tools such as Khoros [Khor91] or apE [Vand90] in addition to AVS, as well as exploring further refinements and enhancements to Schooner. One specific category of enhancements to be investigated are techniques for optimizing the transfer of the large amounts of data that are typical in scientific applications.

Acknowledgements

The authors wish to acknowledge the Advanced Computing Laboratory of Los Alamos National Laboratory, Los Alamos, NM 87545. This work was performed in part on computing resources located at this facility. Charles Hansen of the Advanced Computing Laboratory was particularly helpful in explaining some of the inner-workings of AVS. The authors also wish to thank Christy Bergman, a Ph.D. student in the Department of Mathematics at Stanford University, who wrote the original version of the neural net code and explained its workings sufficiently to allow its adaptation for this project.

References

[Alme85] Almes, G.T., Black, A.P., Lazowska, E.D., and Noe, J.D. The Eden system: A technical review. *IEEE Trans. on Softw. Eng. SE-11*, 1 (Jan. 1985), 43-59.

[Bers87] Bershad, B.N., Ching, D.T., *et al.* A remote procedure call facility for interconnecting heterogeneous computer systems. *IEEE Trans. on Softw. Eng. SE-13*, 8 (Aug. 1987), 880-894.

[Blac86] Black, A., Hutchinson, N., Jul, E., and Levy, H. Object structure in the Emerald System. *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, Portland, OR (Oct. 1986), 78-86.

[Blac87] Black, A., Hutchinson, N., Jul, E., Levy, H., and Carter, L. Distribution and abstract types in Emerald. *IEEE Trans. on Softw. Eng. SE-13*, 1 (Jan. 1987), 65-76.

[Birr84] Birrell, A. D. and Nelson, B. J. Implementing remote procedure calls. *ACM Trans. on Computer Systems 2*, 1 (Feb. 1984), 39-59.

[Gris83] Griswold, R. and Griswold, M. *The Icon Programming Language*. Prentice-Hall, Englewood Cliffs, N.J. 1983.

[Gibb87] Gibbons, P.B. A stub generator for multi-language RPC in heterogeneous environments. *IEEE Trans. on Softw. Eng. SE-13*, 1 (Jan. 1987), 77-87.

[Haye87] Hayes, R. and Schlichting, R.D. Facilitating mixed language programming in distributed systems. *IEEE Trans. on Softw. Eng. SE-13*,12 (December 1987), 1254-1264.

[Haye88] Hayes, R., Manweiler, S., and Schlichting, R.D. A simple system for constructing distributed, mixed-language programs. *Software—Practice and Experience 18*,7 (July

1988), 641-660.

[Haye89] Hayes, R. UTS: A Type System for Facilitating Data Communication, Ph.D. Dissertation, Dept. of Computer Science, Univ. of Arizona, August 1989.

[Haye90] Hayes, R., Hutchinson, N.C., and Schlichting, R.D. Integrating Emerald into a system for mixed-language programming. *Computer Languages 15*,2 (1990), 95-108.

[Jone85] Jones, M.B., Rashid, R.F., Thompson, M.R. Matchmaker: An interface specification language for distributed processing. *Proc. 12th Symp. on Prin. of Prog. Lang*, New Orleans, (Jan. 1985), 225-235.

[Khor91] The Khoros Group, *Khoros Manual* (Vol. 1: User's Manual). Dept. of Elec. and Comp. Eng., Univ. of New Mexico, Albuquerque, NM, 1991.

[Purt91a] Purtilo, J., and Jalote, P. An environment for prototyping distributed applications. *Computer Languages 16*,3/4, (1991), 197-207.

[Purt91b] Purtilo, J. The Polylith software bus. *ACM Trans. on Prog. Lang. and Sys.* (1991), to appear.

[Sun88] Sun Microsystems, Inc. *Network Programming* (Revision A), Part number 800-1779-10, Sun Microsystems, Inc., Mountain View, Calif., May 1988.

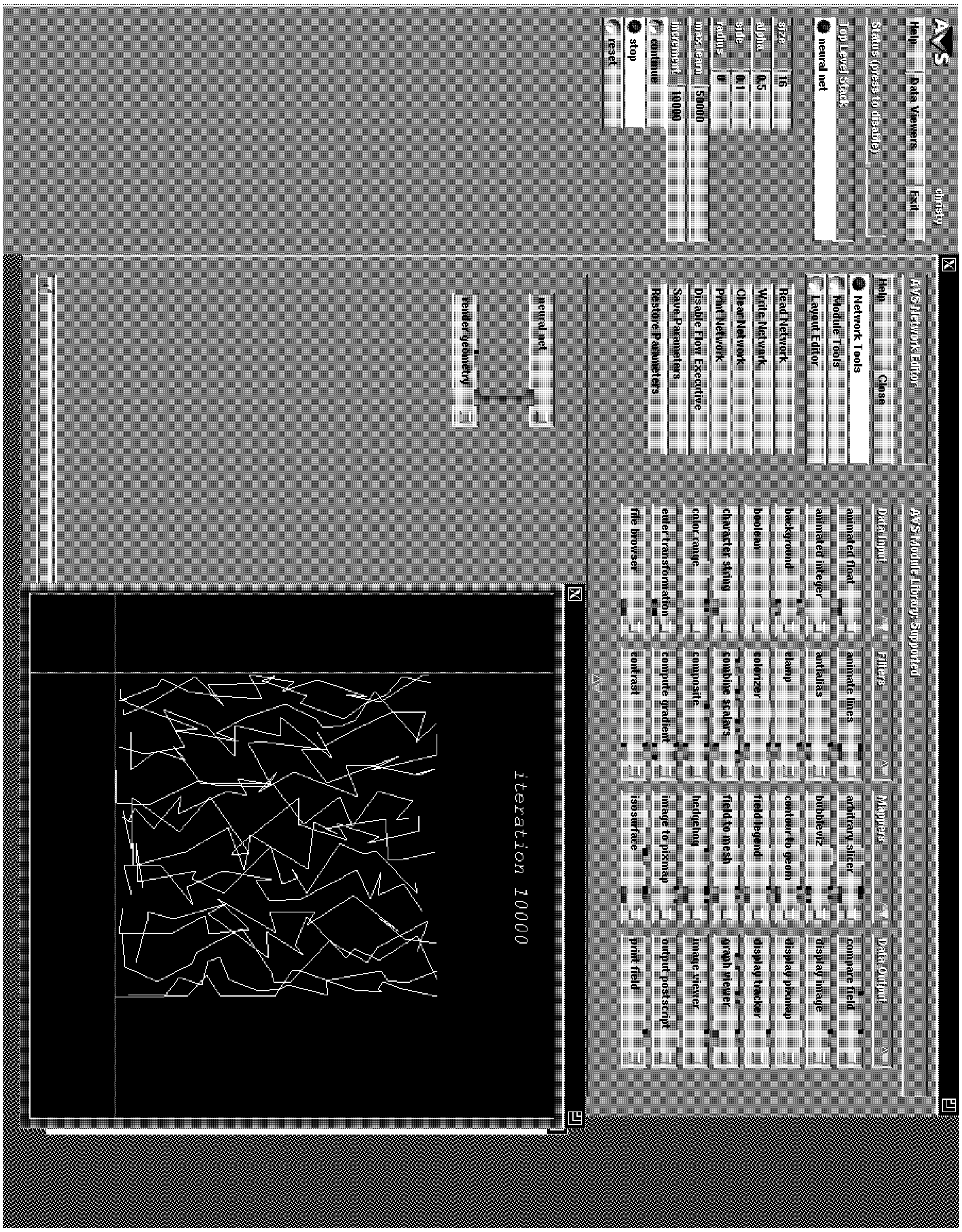
[Star91] Stardent Computer, Inc. *AVS User's Guide* (Release 3.0), Part number 340-0132-02, Stardent Computer, Inc., Concord, Mass., April 1991.

[Vand90] VandeWettering, M. apE 2.0, *Pixel 1*,4 (Nov./Dec. 1990), 30-35.

[Xero81] Xerox Corp. *Courier: The Remote Procedure Call Protocol*. Xerox System Integration Standard XSIS 038112, Xerox Corp., Stamford, Conn., Dec. 1981.

Appendix: Snapshot of Screen During Execution

The snapshot was taken after 10,000 iterations on a 16 x 16 instance of the problem. The AVS module palette and Network Editor are in the background, with an intermediate graphical result being displayed in the right foreground. To the left are the widgets used to control the application.



AVS
christy
Help
Data Viewers
Exit
Status (press to disable)
Top Level Stack
neural net
size 16
alpha 0.5
side 0.1
radius 0
max learn 50000
increment 10000
continue
stop
reset
AVS Network Editor
Help
Close
Network Tools
Module Tools
Layout Editor
Read Network
Write Network
Clear Network
Print Network
Disable Flow Executive
Save Parameters
Restore Parameters
neural net
render geometry
AVS Module Library: Supported
Data Input
animated float
animated integer
background
boolean
character string
color range
euler transformation
file browser
Filters
animate lines
antialias
clamp
colorizer
combine scalars
composite
compute gradient
contrast
Mappers
arbitrary slicer
bubbleviz
contour to geom
field legend
field to mesh
hedgehog
image to pixmap
isosurface
Data Output
compare field
display image
display pixmap
display tracker
graph viewer
image viewer
output postscript
print field
iteration 10000