A Software Platform for Constructing Scientific Applications from Heterogeneous Resources

Patrick T. Homer
Richard D. Schlichting

TR 92-30[1]

ABSTRACT

Support for heterogeneous processing is useful for increasing the functionality available to designers of scientific applications. For example, rather than implement an application requiring remote vector processing and local visualization as two separate programs, such support allows an alternative structure in which the application is a single logical program with transparent transfer of control and data between phases. In addition to being simpler and more intuitive, such structuring makes it feasible to enhance the way in which users interact with the application to do, for instance, model steering. Here, a software platform that facilitates the construction of this type of scientific application is described. Its key component is Schooner, an interconnection system that includes an intermediate data representation, a simple specification language, and a heterogeneous remote procedure call (RPC) facility; to provide sophisticated visualization capabilities and an execution framework, AVS is included as well. Two applications built using this platform, one from molecular dynamics and the other involving neural nets, are also described. One important conclusion is that enhanced monitoring and interaction facilities impose very little overhead for applications such as these.

November 17, 1992

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

---

[1]Replaces TR 91-23 entitled ‘‘Adapting AVS to Support Scientific Applications as Heterogeneous Distributed Programs’’

This work supported in part by the National Science Foundation under grant ASC-9204021. Homer is supported by the National Aeronautics and Space Administration under GSRP grant NGT-50966.

A Software Platform for Constructing Scientific Applications from Heterogeneous Resources

1. Introduction

The ability to exploit heterogeneous resources in scientific applications is useful for a variety of reasons. One is that in some applications, collections of heterogeneous machines can be used to execute a parallel algorithm, thereby allowing the application to produce results faster. Facilitating the use of heterogeneity for such purposes is the target of systems such as PVM [Begu91, Sund90], p4 [Butl92], and APPL [Quea92], which provide sophisticated software infrastructures that, in essence, turn a network of (possibly) heterogeneous workstations into a cost-effective parallel machine. Experience has shown the usefulness of exploiting heterogeneity in this way for many different applications [Nede92, Sund92a, Sund92b].

In this paper, however, we focus on a second motivation for supporting heterogeneous processing, that of increasing the functionality available to the application writer within the framework of a single logical program. With this view, heterogeneity is not just *accommodated* to support inexpensive parallelism, but rather *exploited* to provide access to hardware or software resources that would otherwise not be available due to architectural, operating system, or programming language differences. For example, consider a scientific application in which data is first processed by a vector processor at a remote supercomputer center, then by an existing code that only runs on a particular parallel machine, and finally by a local workstation with visualization software. A typical structure for such an application when no support for heterogeneity is available would be as a series of separate programs with files being used for data transfer. With appropriate support for heterogeneity, however, this application can be written as a single integrated program in which control and data is transparently transferred from vector processor to parallel machine to local workstation.

Structuring scientific applications in this way has a myriad of potential benefits. For one, it is simply more convenient and intuitive for the application designer. Even more important, however, is that it makes it feasible to dramatically enhance the way the user interacts with the application. For example, *model steering*, in which the user views intermediate results and modifies parameters during execution, is usually difficult when computational phases are implemented as separate programs, but easy when they are all combined into one. Thus, the overall vision is of a highly-interactive scientific application structured as a single program that seamlessly accesses any of the vast array of heterogeneous hardware and software resources found in the Internet.

Our specific goal here is to describe a software platform that takes a step towards realizing this vision. The key component of this platform is Schooner, an interconnection system that provides an application-level heterogeneous remote procedure call (RPC) facility based on a machine- and language-independent type system called UTS [Haye89]. UTS, which also served as the basis for Schooner's predecessor system MLP [Haye87, Haye88, Haye90], provides a simple specification language and a canonical intermediate data representation that simplifies heterogeneous processing. To provide sophisticated visualization capabilities and an overall execution framework, the AVS scientific visualization system [AVS92] is included as the second component of the platform. The net result is an infrastructure that allows new and existing software modules executing on a wide variety of machines to be composed into a single application.

This paper is structured as follows. In Section 2, we give an overview of Schooner and describe the three specific services that make up the system: UTS, a collection of stub compilers, and a runtime system that implements RPC-type control transfer. Performance measurements are also given. Section 3 then outlines how AVS and Schooner have been combined to provide a software platform supporting heterogeneous scientific applications. The specifics of two applications are described in Section 4; one is a molecular dynamics program, while the other involves neural nets. Finally, Section 5 describes related work in more detail, while Section 6 contains conclusions.

2. The Schooner Interconnection System

2.1. Overview

Schooner is an interconnection system designed to facilitate the construction of programs that require access to heterogeneous software and hardware resources in the Internet. The goal is to provide the programmer with an easy-to-use system that allows an application to be built across a variety of diverse architectures and languages, including those that support substantially different programming models. Schooner can be viewed, then, as a go-between system that joins disparate pieces of a single program, as illustrated in Figure 1.

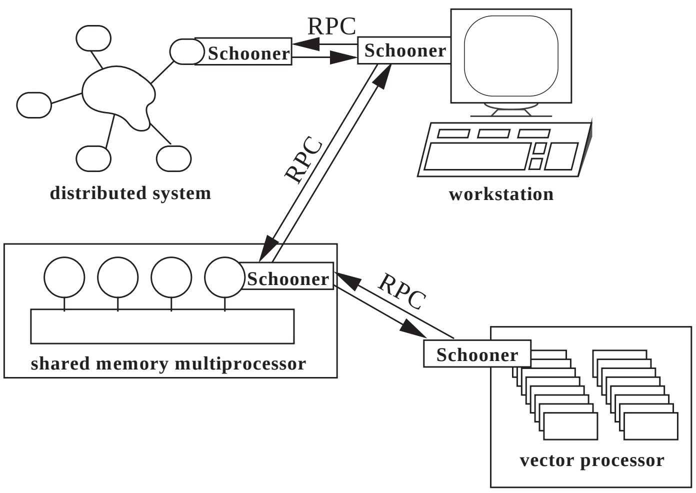


Figure 1 — A Schooner application

The design of a program in Schooner is based on *components* that are composed of one or more procedures. In developing the program, the user is free to write procedures using whatever combination of machine architecture, programming language, and computational model that is most appropriate for solving each part of the problem. For example, one procedure may implement an algorithm that works well on a vector processor, while another might employ a parallel algorithm. Yet a third procedure might be an existing library routine written in a language different from the first two.

A Schooner program is formed by collecting similar procedures together to form components. Procedures are deemed similar when they are intended for the same architecture and are written in the same language. Schooner will support any number of components, including multiple components running on the same machine. Existing programs can be integrated into a Schooner application by identifying at least one procedure in the application to use as the entry point; often this is done by modifying the main program to turn it into a procedure.

Typically, a component is designed with a particular target machine in mind; however, components that are more generally written can easily be moved within Schooner to different machines for various runs. Thus, a programmer might make use of a locally available machine for development of a component and initial testing. When longer production runs are involved, the component can be executed on a different, possibly distant, machine. If the user's computational code is portable between the machines, Schooner will transparently handle the communications regardless of which machine is chosen for a particular run. It is also simple to sub-divide a component into two or more components if desired. This might occur, for example, when another algorithm becomes available to solve a part of the problem and this new algorithm requires a different machine architecture or programming language.

Once a component is written, the only additional work involves writing a specification file. As explained in more detail below, this file contains a specification written in the UTS specification language for each procedure either imported or exported by the component. The specification lists the number and types of the parameters for each such procedure.

Schooner makes use of a sequential execution model where control passes from procedure to procedure. Within this model, however, parallel algorithms can be used by encapsulating the parallel algorithm inside a procedure, as illustrated in Figure 2. These parallel algorithms can execute on specialized hardware such as hypercubes, or could even be realized using a system such as PVM on a collection of workstations. In either case, the role of Schooner is limited to connecting the procedure to the other parts of the application. Thus, Schooner is able to provide connections to a variety of architectural and programming models to further the goal of increased interaction and connectivity, while maintaining the simple programming model implied by RPC and an easy-to-use interface.

Schooner accomplishes its interconnection goal by providing three services: UTS, a collection of stub compilers, and a runtime system to implement control flow and handle communications. To a large extent, these services are distinct and orthogonal, so that changing the implementation of one does not affect the others. Each service can also be used for other

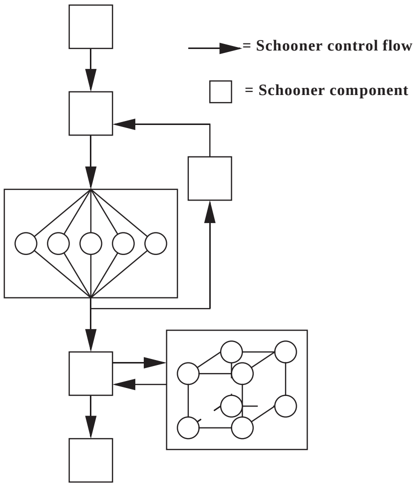


Figure 2 — Parallel algorithms within a Schooner application

purposes as well; for example, the type specification language has also been used as the basis for a command language interpreter [Andr87].

2.2. The Universal Type System

The Universal Type System (UTS) consists of an intermediate data representation and a specification language. The intermediate data representation allows data to be represented in a machine- and language-independent manner. It includes most simple data types, plus full support for array and record types. Library routines are provided to convert data from the host machine's native representation to and from UTS representation; this process is called *encoding* and *decoding*, respectively. In most cases, these routines are only invoked within the automatically generated stub procedures. The representation used by UTS includes type tags on each data element. The tags allow the component on the receiving end of a communication to validate the type of data being received, and also facilitates the streaming of data between components.

The actual representation currently used by UTS is based on the IEEE standards for integers and floating point numbers. However, the specific representation is relatively independent of UTS in the sense that the system is designed to allow a different representation to easily be used for any or all of the supported data types. Similarly, adding data types to the system only requires adding the appropriate encode and decode routines to the UTS library and determining a tag for the type. Adding a machine to the Schooner system requires changes to the UTS library for those data types the new machine supports that do not match an existing UTS representation.

The specification language portion of UTS is used to specify the interface—essentially, the number and type of the 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 component that invokes the procedure and the export specification is associated with the component containing the code for the procedure. For example, the specification for an exported procedure that takes an array of floating point values and returns its sum would be

```
export sum prog("param" val array [100] of float)
      returns ("answer" float)
```

The corresponding import specification for a procedure that wants to call `sum` is usually identical, except that the keyword `import` is used instead of `export`.[2] The keyword `val` in the example above indicates that the parameter is to be passed by value. UTS also supports value-result (`var`) and result (`res`) parameters. The strings in quotes are treated by UTS as comments, allowing the programmer to identify parameters by their intended use as well as their type. Since Schooner components are separately compiled, often on different machines, typechecking is deferred until the program is executed. The first time that a component makes a call to another component's exported procedure, the import and export specifications are compared to ensure type compatibility.

In addition to standard types, UTS supports a *represented type* for parameters that either do not map into any data type in the programming language of the receiving component, or that map into multiple types. This allows, for example, the passing of an array whose size is not known until runtime, or the passing of C unions or Pascal variant records where the types of the fields are not completely known at compile time. When a type is declared as ‘‘represented’’ in this way, a representative or ‘‘ticket’’ for the UTS value is supplied as the argument instead of the actual value. Routines in the UTS library can then be invoked with the representative as an argument to allow the programmer to inquire as to the UTS type of the corresponding value, and to perform explicit conversion of values between UTS representation and the representation of the host language.

2.3. Stub compilers

In addition to being used for type checking, the specification file is also used as the basis for automatic generation of stub procedures. There is one such procedure generated for each imported and exported procedure in a component, and it is here that argument values are converted between native and UTS representations. This is done by having the stub invoke the correct encode and decode routines from the UTS library. 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 [Gris90], and Emerald [Blac86, Blac87].

After stubs for a component are generated from the specification, they are compiled using the appropriate language processor. The resulting object module is then linked with the user's

[2]The specification language actually allows parameters to be described by sets of types, which means that import and export specifications need not match exactly. See [Haye88] for details.

code, the UTS libraries, and the Schooner runtime support libraries to produce an executable. This sequence of steps is illustrated in Figure 3.

2.4. Runtime system

A component in Schooner is implemented by a process at runtime, which means that a component is also the unit of distribution in the system. As already noted, control flow is based on a sequential procedural programming model in which there is logically one thread of control that transfers between components when an RPC is executed. This control flow is implemented by the communication library portion of Schooner's runtime system, which is linked in with every component. The remainder of the runtime consists of two types of system processes. The first is a Schooner Manager process; there is one such process per program, which is used to manage component initiation and termination, perform typechecking, and implement the runtime mapping between procedure names and component location. The second is the Schooner Server process; there is one per machine, and its task is to perform the actual component initiation when requested to do so by a Schooner Manager. The Server also responds to requests by independently started components on its machine for the location of the Manager.

The Schooner communication subsystem currently supports message passing using either TCP virtual circuits or UDP datagrams. The choice is made by the user when the Schooner program is run. The system can easily be adapted to use other communication protocols by simply adding the appropriate send, receive, open, and close functions to the communications library.

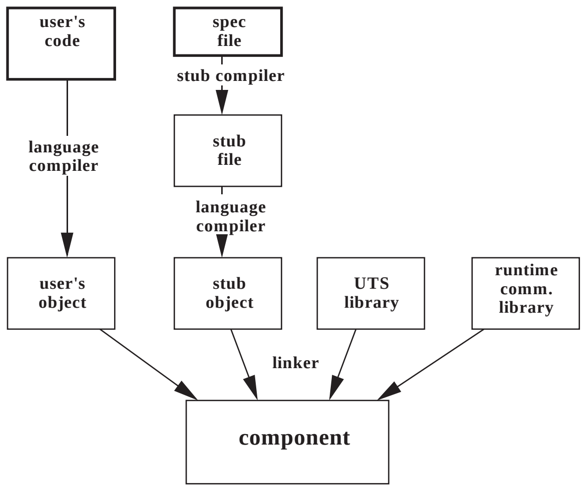


Figure 3 — Producing a Schooner executable

2.5. Performance

To quantify the overhead involved in using Schooner, two series of tests have been conducted. In the first, two Sun Sparc 2s located on the same physical Ethernet were used. The test program consisted of two components, with the component containing the main program making a call to a null procedure in the second component. Except for the first test, an array of double-precision floats was passed as a value-result (`var`) parameter. To measure the impact of using UTS' intermediate representation, measurements were made to determine the time needed to encode and decode the array, as well as the elapsed time for the round-trip procedure call.

The results of these experiments are shown in Table 1. The method involved having the application calculate the elapsed time after 100 calls to the procedure; in the case where no arguments were passed, the test involved 500 calls to get a more accurate measurement. Each case was run five times. The table shows the results after averaging the five runs and dividing by the number of calls. The encode time is the time the application spent in the stub utilizing the UTS library routines to encode the array of values; the decode time is analogous. This was determined using the results reported by the `getrusage` system routine and then adding together the user and system times. Finally, the percentage column indicates the proportion of the total time that was spent in the two encode and two decode sections of the RPC. The Sun Sparc architecture uses the IEEE floating point representation and also stores the bytes in network order, meaning that the encode and decode times are the time needed to copy the bytes for each value and its tag into and out of the message buffer. Given that 8-byte double-precision values are being passed, each encode or decode time represents the processing of 8x(array size) data bytes.

The second series of tests were designed to determine the costs involved when data values must be converted across machines, and the costs involved in doing RPC calls outside a local network connection. In this set, the component containing the main program was run on a Sun

Array size	Main Program encode	Main Program decode	Remote Procedure decode	Remote Procedure encode	Time	Percentage
0	*	*	*	*	10.8	*
1,000	5.4	5.0	4.8	5.4	53.6	39
2,000	10.7	10.0	9.4	10.5	99.7	42
3,000	15.5	15.3	14.0	15.4	142.5	43
4,000	21.2	19.6	18.9	20.0	185.7	44
5,000	27.4	24.1	23.3	25.6	231.5	44
6,000	34.7	29.3	29.0	31.3	279.8	46
7,000	37.4	33.8	33.9	37.1	320.1	44
8,000	42.3	38.6	38.9	42.6	367.9	44
9,000	48.4	43.3	43.5	48.5	416.0	44
10,000	53.5	48.1	47.7	54.2	457.5	44

Table 1 — Sun-Sun test (times in milliseconds)

Sparc 2 and the other component on a Convex C220; the two machines are located in the same building, but with several network gateways between them. The Convex does not use the IEEE format for its native floating point mode; thus, bit manipulations were required to correctly convert both its mantissa and its exponent. This also involved tests to determine if underflow or overflow occurred. During the tests, the user was the only login active on the Sun, but the Convex was handling a varying number of other users. As in the first set, each test consisted of 100 calls and was run five times.

The results are summarized in Table 2. Not surprisingly, the percentage of the total time devoted to encoding and decoding is smaller here, mainly due to the increased amount of time needed to traverse the gateways between the two machines. The encode and decode times for the Convex are also larger than those needed for the Sun due to the time needed to perform the conversions.

While the data conversion aspect of an intermediate data representation certainly imposes some cost, our feeling is that the advantages outweigh the disadvantages for this type of application. For example, it makes Schooner easier to use since the user need not specify in advance the type of machine to which the data will be sent or the programming language that is being used. It also simplifies the implementation and makes it easier to port to other machines. Finally, note that, since null procedures are being used here, the percentages in the tables for encoding and decoding are essentially being measured relative to the RPC itself and not an entire application. In other words, as the amount of time spent doing actual computation in the

Array size	Main Program encode	Main Program decode	Remote Procedure decode	Remote Procedure encode	Time	Percentage
0	*	*	*	*	56.4	*
1,000	5.5	5.3	8.2	10.8	415.6	7
2,000	10.9	10.0	15.6	19.4	446.7	13
3,000	16.9	15.1	23.2	31.3	629.7	14
4,000	21.6	19.8	30.5	38.4	868.7	13
5,000	27.3	24.2	38.1	48.1	967.6	14
6,000	32.8	28.9	45.5	57.7	1,183.3	14
7,000	38.1	33.8	53.0	67.3	1,141.4	17
8,000	43.2	38.9	60.6	77.0	1,479.9	15
9,000	49.0	43.3	69.2	94.3	1,499.8	17
10,000	54.3	48.1	75.1	102.8	1,568.4	18

Table 2 — Sun-Convex test (times in milliseconds)

application increases, the overhead involved in encoding and decoding data values will diminish greatly in significance.

3. Schooner and AVS

As alluded to in the Introduction, Schooner has been combined with the AVS system to provide a platform for constructing scientific applications that can access heterogeneous hardware and software resources. In this platform, AVS provides visualization capabilities and the basic structure of the computation, while Schooner is used to connect with codes that are executed as external computations independent of AVS. Here, we first provide a short overview of AVS, and then describe how applications are written using the platform. Our primary emphasis has been on taking existing scientific codes and incorporating them into this platform to improve the ability of the user to monitor and manipulate the computation.

3.1. Overview of AVS

AVS is a graphics system for displaying images generated by scientific computations. 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. To manipulate both the data and the resulting images(s), AVS provides a large palette of tools. Examples of data manipulations include filters to reduce the amount of data and perform computations such as determining vector magnitudes, and modules to read a variety of data formats and convert these to AVS data fields. Images can be manipulated through rotations of the objects, reflections, positioning of lights and choice of lighting models, applying color to objects according to data values, etc.

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. 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 desired 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 as a data flow network. In addition to receiving values through the network, a module can accept input from the user through parameters. A module that uses parameters will have a ‘‘widget’’ such as a dial, slider, or type-in window appear on the screen for each parameter.

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. As each module completes execution, its output data, if any, is passed along to ‘‘downstream’’ modules. Modules can have side-effects as well, with the most common being the display of images or the creation of output files.

In addition to a large selection of standard modules, AVS includes provisions for user-written modules as well. Two types of such modules are supported: *procedure modules* and *co-routine modules*. A procedure module is scheduled by the AVS kernel and executes once each time it is scheduled using the criteria given above. A co-routine module, on the other hand, is scheduled independently of the AVS kernel, so these modules are typically structured as a continuous loop that outputs data to downstream modules at regular intervals. While this mechanism can be used to achieve some concurrency when multiple processors are available, it has a number of deficiencies, not the least of which is that it subtly alters the programming model as perceived by the application writer.

3.2. Using Schooner with AVS

The wide variety of visualization tools provided by AVS and its support for user-written modules make it an attractive tool for use as the visualization end of a heterogeneous application. In building our Schooner/AVS platform, we have not had to alter substantially either AVS or the way in which the application is written. The user still exercises control over AVS in the normal manner; essentially, it appears to the user as though the external computation is running within AVS. 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, to halt the computation, or to modify parameters during execution. However, the net result is that the fundamental ‘‘feel’’ of AVS has been retained, minimal changes are needed for the application, and the user is able to take advantage of heterogeneous resources in a transparent manner.

The process of writing a scientific application using the Schooner/AVS platform is straightforward, especially in the common case where the computational phases already exist as stand-alone programs that send their output to one or more files. In these situations, the goal is to make as few modifications as possible, while adding the ability to view the intermediate and final results, and to modify one or more parameters during execution. The first step in constructing such an application is to change the computational phases into procedures. 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 (or perhaps, in addition to) writing 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. This feature of accepting input and sending output as parameters to the computation procedure often simplifies the application’s code, as there is no need to devise routines to create a particular file format or to read a particular input format. While some work may be needed on the AVS end to convert the arrays of numbers into, for example, an AVS data field, this is often necessary when reading non-AVS formatted data files in any event.

The next step involves writing a new AVS 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 making the results available on one or more output ports. All of these tasks are usually straightforward and very similar to those required by any new AVS module. Input is done using the widget mechanisms described above if from the user, or from data ports if from other AVS modules. The invocation of the remote procedure is written using the language’s standard procedure call mechanism, which is, of course, later transformed by the appropriate Schooner stub compiler. Output is performed 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 one of the standard AVS data models; 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 were coming, for instance, from a file rather than a remote computation.

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 Sections 2.2 and 2.3, these specification files are used by the Schooner stub compilers and for typechecking.

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 be better to modify the computation to be a procedure that returns intermediate values. This will allow the user to monitor the progress of the computation and modify parameters during the computation. For example, there might be a need to establish new boundary values as the computation approaches a steady-state condition. In this model, the procedure will typically retain 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 AVS module containing the call should be a procedure module or a co-routine. The procedure choice is particularly appropriate when the user anticipates the need for frequent interaction with the computation. The co-routine choice is best when interaction is needed only occasionally and the main intent is to monitor the computation. When each computation is long and the visualization being performed by AVS is complex, the co-routine choice will also gain some concurrency since the next call to the remote procedure will be started while AVS is still working on rendering the results of the previous call.

Finally, it should be noted that AVS does provide limited support for executing procedure modules (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, in the situation where two or more versions of a component are available, AVS requires separate modules in the network for each. Thus, the selection of a different machine requires changing the network by deleting the previous module and adding the new one. Schooner’s dynamic startup capability, in contrast, allows the user to select among remote components using a widget; no change is needed to the network and only one AVS module is required.

4. Example Applications

A number of example scientific applications have been constructed using the Schooner/AVS software platform, two of which are described here: one from molecular dynamics and the other involving neural nets. In both cases, the primary goal was to increase the interaction capabilities of existing programs, although in the neural net example, the use of the platform also made it easy to use a parallel algorithm on a geographically remote machine to perform the actual computation. In this section, we describe how these applications were adapted and also attempt to illustrate that the increase in interaction capabilities incurs only a minimal performance cost.

4.1. Molecular Dynamics

This program computes constant energy-volume-number particle dynamics for Lennard-Jones 12-6 interparticle forces using an algorithm based on [Alle82]. On each time step, it advances the positions of the particles, and then computes the forces among the particles, and the kinetic energy and thermodynamics of the system. The particles reside in a unit cube and their initial positions are uniformly distributed within the cube. The initial velocities are randomly determined about a specified standard deviation. The original application produced data files that recorded the velocities and positions of the particles, and each iteration of the main loop would proceed for a specified number of time steps, then update the files. The original intent of the program was to produce an animation showing the positions of the particles in the system over time, with arrows indicating the direction and magnitude of the velocity of each particle.

The difficulty with the original application design was that the production of data files and their viewing were separate steps done by separate programs. Since part of the point of the program was to determine the preferred values for such parameters as the number and size of the time steps between frames in the animation, having to generate a series of frames and view them separately was an awkward and time-consuming process. Directly connecting the viewing of the frames with the computation solved this problem.

To implement this application using the Schooner/AVS platform required only a few modifications along the lines of those detailed above. The most significant change was to the main program. The original program took input from the keyboard, generated a set of particles and ran the computation for a specified number of iterations before writing the output file. The revised program makes use of two procedures: `generate`, which determines the initial positions of the particles, and `dynamic`, which advances the computation a specified number of iterations, using existing procedures from the original program to compute positions, forces, etc. Both procedures return the positions and velocities of the particles in the form of two arrays. They also retain the ability to produce output files as well, since such files have potential value both for restarting interrupted computations and for creating an animation data file. The two procedures share global data structures to retain the particles' positions and velocities across calls. This allowed the results of `generate` to be used by `dynamic`, and eliminated the need to send the data in both directions on each call.

The second step was to write the new AVS module. This module is designed in the normal way that AVS provides. AVS modules consist of two required functions, `spec` and `compute`, and two optional functions, `init` and `destroy.` `init`, when present, is called when the module is instantiated in an AVS network; `destroy` is called when the module is removed from the network. The `spec` function describes the widgets, sets up the input and output data ports, and identifies the names of the other functions. The `spec`, `init`, and `destroy` functions are affected only a small degree by the use of Schooner. Specifically, `spec` sets up one additional widget for Schooner to allow the user to indicate on which machine to run the remote computation, while `destroy` contains one Schooner library call to notify the Manager when the AVS module is terminated.

The only real difference between this module as used in Schooner and other user-written AVS modules lies in the `compute` function. In our version, `compute` still handles the input from AVS widgets and input ports, and forms the output to pass to downstream modules, as is done in normal AVS modules. But the computation itself appears only as a procedure call to one or the other of the two procedures exported by the separate Schooner component that actually performs the computation. The choice of which procedure to call is controlled by the user. The module also has a restart widget that indicates when a computation is to start over by calling `generate.`

A final change in `compute` was the addition of an invocation to a library function to handle the Schooner initialization protocol properly. Specifically, this function contacts the Schooner Server on the AVS machine to obtain the address of the Manager process. With this information, the component is then able to contact the Manager and proceed with the exchange of mapping information as outlined in Section 2.4. The library routine also takes care of the request to the Manager to start the remote component on the machine the user selected. The library routine is called only the first time the computation function is invoked.

The final step in performing the integration was to write the UTS specification files. The export specification for the component containing `dynamic` is as follows; the export specification for `generate`, and the import specifications for both procedures, are analogous.

```
export dynamic prog(
   "num_particles" val integer, "cutoff" val float,
   "density" val float,
   "time_step" val float, "num_time_steps" val integer,
   "positions" res array [300] of
           record{"x" float, "y" float, "z" float},
   "velocities" res array [300] of
           record{"x" float, "y" float, "z" float})
```

Here, `num_particles`, `cutoff`, and `density` control the dynamics of the simulation. `time_step` indicates the size, in seconds, of each time increment for the main compute loop, while `num_time_steps` indicates the number of these steps the procedure is to proceed before returning. `positions` and `velocities` are the result of calling the procedure; the application makes use of an AVS irregular grid, which requires that `generate` and `dynamic` return both of these sets of values.

The keyword `val` before the type specifies that the associated argument is passed by value, while `res` indicates a result parameter. In this application, the arrays are returned as result parameters, since the data need only be transmitted in one direction. The size of the arrays is set at 300, a choice based on the number of particles that might reasonably be viewed within AVS. The AVS widget for `num_particles` uses this value as a maximum and does not allow the user to select a larger value, even though it is possible using Schooner to have variable size arrays that depend on choices made by the user at runtime. An example of this option is in the neural net application described below.

The one additional aspect of the integration to be discussed is the choice between making the new module a co-routine module or a procedure module. In this case, it was determined that selecting the procedural option was the better choice. The decision was based on the desire to have complete control over the execution of the module, allowing interaction after each call to the procedure, and on the need to save views during the execution. It was not known at the start how many or how often views might want to be saved, and it was anticipated that a trial-and-error approach would be necessary to determine the right size and number of time steps.

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, or a previously saved network is read in. When the user-module is dragged from the menu into the edit space, the initialization routine is run and the various widgets are displayed. The user can then specify the machine where the computation is to be run, along with providing values for the other parameter widgets. When the module is first executed by AVS, the Manager is contacted and mapping information is exchanged. During this first run, the Manager also establishes contact with the indicated machine. From this point on, the module works like any other AVS module: when a change is made to a parameter, the module is called and Schooner transparently handles the communication with the remote procedure. A screen snapshot taken during the execution of the application is shown in the Appendix; the widgets are on the left side of the screen, the AVS network in the middle, and the graphical output on the right.

The molecular dynamics application has been tested on several combinations of machines. One set of tests used machines at Los Alamos National Laboratories. Here, the AVS portion of the application was executed on either a Stardent or SGI machine, with the dynamics portion running on a Sun or Convex. The dynamics portion was also run remotely on Sun and Convex machines at The University of Arizona. Additional tests were run at NASA Lewis Research Center. Here, AVS was executed on both Sun and SGI machines, with the dynamics portion running on Sun or Convex machines at Lewis or The University of Arizona.

To test the performance, a series of tests were conducted with AVS running on a Sun Sparc 2 and the dynamics portion of the application running on a Convex C220. This is the same configuration used for the second set of performance measurements reported in Section 2.5, with both machines being in the same building, but on different networks. The results are shown in Table 3. The problem size is the number of particles. The times shown are in seconds and reflect the elapsed time on each machine for one call. The call consisted of a request to execute 100 iterations with a time step of 0.001 seconds per iteration. The difference column is the proportion of the total elapsed time not spent actually executing the computation, and hence attributable (mostly) to communication overhead. Note that, given the use of fixed size arrays, this

Problem size	Convex	Sun	% difference
8	0.090	0.267	66.3
24	0.614	0.789	22.2
40	1.650	1.835	10.1
56	3.190	3.365	5.2
72	5.188	5.367	3.3
88	9.125	9.301	1.9
104	13.711	13.887	1.3
120	18.784	18.975	1.0

Table 3 — Molecular dynamics application (times in seconds)

communication cost is the sum of a constant encoding/decoding time plus a (slightly) variable network delay between the machines. The computation time, on the other hand, is dependent on an $O(n^2)$ algorithm and increases accordingly. Thus, the communication overhead as a percentage of the total execution cost rapidly decreases as the problem size grows, becoming quite small for moderate to large problem sizes. For small problems, this implies that it would be more productive to execute the dynamics portion of the application on a local machine, perhaps even the same machine running AVS, to minimize the communications cost. As larger problem sizes are run, however, it becomes advantageous to use the faster processor, as the communication costs become a very small fraction of the execution time.

4.2. Neural Net

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 number of elements in the net, 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.

The processing elements in the neural net use information about their neighbors in a learning process to determine their position within a grid. Once the program has reached convergence, a plot with the points connected to each other should show uniformly spaced vertical and horizontal 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 nets with a high initial degree of randomness; for example, in one run involving 32 x 32 points, 210,000 iterations were required to get a reasonably good solution.

Given the long-running nature of this application, the primary motivation for using the Schooner/AVS platform was to be able to monitor its progress during a run. To do this, options were provided for viewing intermediate results, as well as for steering the program. The latter included provisions for halting the run and for changing the parameter representing the half-width of the Gaussian function in mid-stream.

The process of implementing the neural net program using the Schooner/AVS platform was very similar to that described above for the molecular dynamics application. The neural net program was changed into two procedures: a `configure_net` procedure to handle the initial semi-random placement of the processing elements and a `compute_net` procedure to handle a specified number of iterations of the neural net computation. There were two major differences compared to the molecular dynamics application. The first was the decision to write the AVS module for the neural net as a co-routine. Such a module can execute on its own independently of other AVS modules, which means that when the remote procedure is executing, the AVS kernel can schedule other modules in the network to run without waiting for the remote procedure to return. For large neural nets, this provides some concurrency since the neural net module can initiate the next remote call while the other modules in the network are still rendering the image from the previous call.

The other important difference concerned the use of variable size arrays. Unlike the dynamics application, in this case the problem requirements mandated that no predefined limit be set for the size of the neural net. To illustrate how this is handled in Schooner, we start with the UTS specification for `compute_net`:

```
export compute_net prog(
   "num_points" val integer, "alpha" val float,
   "cur_count" var integer, "increment" val integer,
   "rep_x" res rep array [-] of float,
   "rep_y" res rep array [-] of float)
```

Here, `num_points` specifies the size of the neural net, which will be `num_points` x `num_points`, while `alpha` is the half-width of the Gaussian function. `cur_count` is the current iteration count and `increment` is the number of iterations to proceed on the current call. `rep_x` and `rep_y` contain the x and y coordinates of the points and are the result of calling the procedure.

The key to allowing variable-sized arrays is the specification of `rep_x` and `rep_y` as `rep` arrays, indicating that these parameters are represented types as described in Section 2.2. While allowing the size to be established at run-time, this choice does require some additional work on the part of the programmer since UTS library routines must now be used to encode explicitly the array values in `compute_net` and subsequently decode them in the AVS module. For example, the following is the actual code used to encode the values for the `rep_x` array:

```
long rep_x;
size_squared = num_points * num_points;
/* get space for array plus header */
user_rep_new(rep_x, size_squared * (sizeof(double) + 1) + 50);
uts_start_array(rep_x, 1, size_squared);
/* put each value into the represented array */
for (i = 0; i < size_squared; i++)
   uts_encode_float(rep_x, x[i]);
uts_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 as was done in the molecular dynamics example. Even in this case, it is still possible to have the problem size be a parameter, but the chosen array size would now be an upper-bound on the problem size. This option also incurs some performance penalty since the entire array is transmitted during the RPC even if just a portion is actually being used.

Several versions of the code have been developed, 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 shared-memory, multiprocessor architecture of the machine. This version uses a parallel algorithm that divides the elements of the neural net into horizontal bands and allocates each band to a processor. The code is implemented using the parallel library provided by Sequent to synchronize the necessary exchange of boundary values on each iteration and to allow the use of shared arrays to hold the results. Connecting this version of the neural net application to AVS was no different than connecting the sequential algorithms except for the addition of one more parameter to allow the user to specify the number of processors to be used in the calculation. This parameter is used in `configure_net` and can be changed whenever a restart of the computation is requested. Thus, the flow of control is single threaded from the AVS co-routine until the remote procedure call arrives at the Sequent. Then, the flow divides among the processors for the requested number of iterations. Once those are complete, the flow becomes single-threaded again for the reply back to the AVS co-routine.

This application has been run successfully on a number of machine combinations. These tests have included combinations with AVS running on machines at Los Alamos, including both a Stardent and an SGI, and machines at NASA Lewis, including a Stardent, an SGI, and a Sun Sparc 2. The computational component was run either on a local machine, including a Convex C220, Sun Sparc 2, and SGI, or on remote machines at The University of Arizona, including a

Sequent Symmetry, Sun Sparc 2 and Convex C240.

To measure the performance of this application, a combination was chosen consisting of AVS on a Sun Sparc 2 at NASA Lewis and the parallel version of the neural net application on the Sequent at The University of Arizona. For the tests, the number of processes used for parallelism was set at 8, and tests were run for several problem sizes, each of which is a multiple of 8 to balance the work required by each process. It should also be noted that while the Sequent architecture supports the IEEE floating point format, the byte ordering used is reversed from that used by the Sun architecture. Each test was run for a total of 1,000 iterations. The Appendix gives a snapshot of the screen taken during execution of the application.

To measure the impact of monitoring the application, the tests were run with two versions. In one, a single call was made for the 1,000 iterations; in the second, a series of 10 calls for 100 iterations each were made. The results are shown in Table 4, where the times were measured using `gettimeofday`. The values in the Sequent column represent the time actually spent executing `compute_net`, while the two Sun times correspond to the elapsed time for the one call and ten call versions, respectively. The last column is the percentage difference for the Sun times between the one call and the ten call version; this indicates the performance decline imposed by monitoring the computation more often. As can be seen, this penalty is small even when the monitoring frequency is fairly high. In the example mentioned earlier where 210,000 iterations were used, monitoring every 100 iterations would equate to 2,100 calls for an estimated 5.9% slower execution.

5. Related Work

The PVM (Parallel Virtual Machine) [Sund90, Begu91] project is representative of a class of systems (including p4 [Butl92], APPL [Quea92], and others) that provides general support for constructing parallel and distributed programs using a collection of heterogeneous machines. To

Problem size	Sequent	Sun (1 call)	Sun (10 calls)	% difference
16 x 16	63.2	64.1	74.5	16.2
32 x 32	250.0	251.7	266.6	5.9
48 x 48	560.4	563.4	587.7	4.3
64 x 64	995.4	1,002.1	1,044.7	4.3

Table 4 — Neural net application (times in seconds)

do this, PVM provides a set of library routines that implement such things as asynchronous message passing between processes and broadcast communication, as well as other facilities for synchronization and consensus.

A PVM program statically consists of one or more *components.* During execution, these components are instantiated as one or more processes under program control, so that a running PVM application consists of a varying number of interacting processes. Program execution is managed by a PVM daemon process that runs on each machine used by the application. The application is started by executing the main program on one machine, with subsequent requests for process initiation being routed to the appropriate daemon for action. In addition to startup, the daemons handle synchronization tasks, and typically act as a relay point for local messages destined for processes on other machines or for incoming messages destined for local processes.

Differences in data representation between machines is handled in PVM by using Sun XDR [Sun90] as an intermediate representation, with a collection of library routines being provided to convert standard data types. These calls are made prior to a send or after a receive. Also, components can be compiled for multiple architectures. During execution, the PVM daemon will select the correct compiled version of a component to start on the indicated architecture. The choice of architecture is determined explicitly when a request is made to start a process, or implicitly through a configuration file if the architecture is not specified.

The p4 system is similar to PVM in intent, but with some additional features. For example, execution of p4 programs on shared memory architectures is optimized, whereas PVM programs on similar architectures retain their basic message passing paradigm. The APPL project has also implemented some optimizations for specific shared-memory and distributed memory architectures. All of these systems have a primary goal of portability of codes across different architectures and configurations of machines.

Despite a common theme of heterogeneous processing, there are a number of differences between PVM (and similar systems), and the Schooner component of the platform described in this paper. Perhaps the most significant is that the two systems have different goals: PVM seeks to facilitate the implementation of parallel algorithms for purposes of speedup, whereas our platform is oriented towards making it easy to connect existing or new codes into a single application to improve interaction capabilities. As a result, PVM must be more general and complete, while Schooner need only support a single logical thread of control, with the resulting simplification of (especially) runtime aspects. We point out again, however, that this single thread of control does not preclude the use of a parallel component within a Schooner application, as illustrated by the neural net application. Indeed, Schooner could be used to connect a PVM program to other computations in much the same way as the parallel algorithm was included in the neural net example; in this case, the PVM program would be viewed as one particular component of a larger application.

Another difference is the use of a machine- and language-independent type specification language in Schooner. This portion of UTS simplifies the generation of the library calls necessary to convert values across machines. In particular, while PVM and the other systems use a series of library calls to handle conversions much like Schooner does, the user must explicitly insert the calls around the send and receive primitive. In contrast, with Schooner, the user need only include the standard procedure call and write a one-time specification that is used by the stub compilers to automatically generate the appropriate library calls.

Also closely related to the Schooner component of the platform are other RPC schemes with features such as external data representations, specification languages, and stub compilers [Alme85, Birr84, Sun90, 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 [Purt91, Purt92].

6. Conclusions

The software platform we have constructed out of the combination of Schooner and AVS provides a simple and intuitive way for programmers to construct scientific applications that require access to heterogeneous resources. These resources can either be hardware, such as the case where a parallel machine is appropriate for one part of the application and a graphics engine for another, or software, such as the case where software components written in different programming languages or that execute on different machines are needed for different computational phases. By writing such applications as a single program rather than a series of separate programs that use files for data transfer, it becomes feasible to improve substantially the way in which the user interacts with the application. Moreover, as demonstrated by our molecular dynamics and neural net examples, such an improvement can often be gained with little additional execution overhead or programming effort.

The Schooner/AVS platform is also fundamentally a simple system, both in the way it is seen by the applications programmer and end user, and the way in which it is designed and implemented. One reason for this simplicity is that, in contrast with more general systems for heterogeneous processing, Schooner/AVS only supports a model in which a single logical thread of control sequentially executes distinct computational phases. This preserves a straightforward programming model at the high level, while not precluding the use of parallelism within a given computational phase where specialized languages or systems can be more judiciously employed. Another reason for this simplicity is the use of a type system—UTS in our case—that provides a specification language and data representation that are independent of a particular machine architecture or programming language. The availability of represented types in UTS adds greatly to the flexibility afforded the programmer, and is one of the features that distinguishes UTS from similar systems such as Sun XDR and ASN.1 [ISO87a, ISO87b].

Our future efforts in this area will be directed along several lines of investigation. One is using the Schooner/AVS platform for more and larger applications; in this vein, we are currently involved with using the platform to connect existing codes for jet engine simulation as part of the NPSS project at NASA Lewis Research Center [Clau91]. Another is upgrading the platform's functionality. For example, Schooner currently allows only one instance of a given remote procedure to exist within an application. This implies an inability to have two versions of the same procedure available to the user within an AVS network, even if the versions are running on different machines.

Finally, we intend to look at more long-term issues concerning the development of heterogeneous scientific applications. 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 ways to divide scientific applications given a particular set of heterogeneous resources and better techniques for utilizing existing codes in this model without modifications. For the latter, we will look at utilizing other graphics visualization tools such as Khoros [Rasu91, Merc92] 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

S. Lustig wrote the original version of the molecular dynamics program and provided assistance and feedback, while C. Bergman did the same for the neural net example. This work was performed in part on computing resources at the Advanced Computing Laboratory at Los Alamos National Laboratory and NASA Lewis Research Center; thanks to C. Hansen and G. Follen for providing support and advice at Los Alamos and NASA Lewis, respectively.

References

[Alle82] Allen, M.P. and Tildesley, D.J. Computer simulation of liquids. *CCP5 Quarterly 6*, 4 (1982).

[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.

[Andr87] Andrews, G.R., Schlichting, R.D., Hayes, R., and Purdin, T.D.M. The design of the Saguaro distributed operating system. *IEEE Trans. Softw. Eng., SE-13*, 1 (Jan. 1987), 104-118.

[AVS92] Advanced Visual Systems Inc. *AVS Developer's Guide* (Release 4.0), Part number: 320-0013-02, Rev B, Advanced Visual Systems Inc., Waltham, Mass., May 1992.

[Begu91] Beguelin, A., Dongarra, J. J., Geist, G. A., Manchek, R., and Sunderam, V. S. Graphical development tools for network-based concurrent supercomputing. *Proc. Supercomputing '91* (Nov. 1991), 435-444.

[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.

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

[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*, 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. 87), 65-76.

[Butl92] Butler, R. and Lusk, E. User's Guide to the p4 Parallel Programming System, Argonne National Laboratory, Argonne, IL, August 1992.

[Clau91] Claus, R.W., Evans, A.L., Lylte, J.K., and Nichols, L.D. Numerical Propulsion System Simulation. *Computing Systems in Engineering 2*, 4 (Apr. 1991), 357-364.

[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.

[Gris90] Griswold, R. and Griswold, M. *The Icon Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[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.

[ISO87a] *Information Processing Systems—Open Systems Interconnection—Specification of Abstract Syntax Notation One (ASN.1)*. International Organization for Standardization, International Standard 8824, December 1987.

[ISO87b] *Information Processing Systems—Open Systems Interconnection—Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*. International Organization for Standardization, International Standard 8825, December 1987.

[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.

[Merc92] Mercurio, P.J. Khoros. *Pixel 3*, 2 (Mar./Apr. 1992), 28-33.

[Nede92] Nedeljkovic, N., and Quinn, M. Data-parallel programming on a network of heterogeneous workstations. *Proc. 1st Int. Symp. on High-Performance Distributed Computing*, Syracuse, NY (Sept. 1992), 28-36.

[Purt91] Purtilo, J., and Jalote, P. An environment for prototyping distributed applications. *Computer Languages 16*, 3/4 (1991), 197-207.

[Purt92] Purtilo, J. The Polylith software bus. *ACM Trans. on Prog. Lang. and Sys.* (1992), to appear.

[Quea92] Quealy, A., Cole, J., and Blech, R. Portable Programming on Parallel/Networked Computers using the Application Portable Parallel Library (APPL), NASA Technical Manual, Fall 1992, to appear.

[Rasu91] Rasure, J. and Williams, C. An integrated visual language and software development environment. *Jour. of Visual Languages and Computing 2* (1991), 217-246.

[Sun90] Sun Microsystems, Inc. *Network Programming Guide* (Revision A), Part number 800-3850-10, Sun Microsystems, Inc., Mountain View, CA, March 1990.

[Sund90] Sunderam, V. S. PVM: A framework for parallel distributed computing. *Concurrency—Practice and Experience 2* (Dec. 1990), 315-339.

[Sund92a] Sunderam, V.S. Design issues in heterogeneous network computing. *Proc. Workshop on Heterogeneous Processing*, Beverly Hills, CA (March 1992), 101-112.

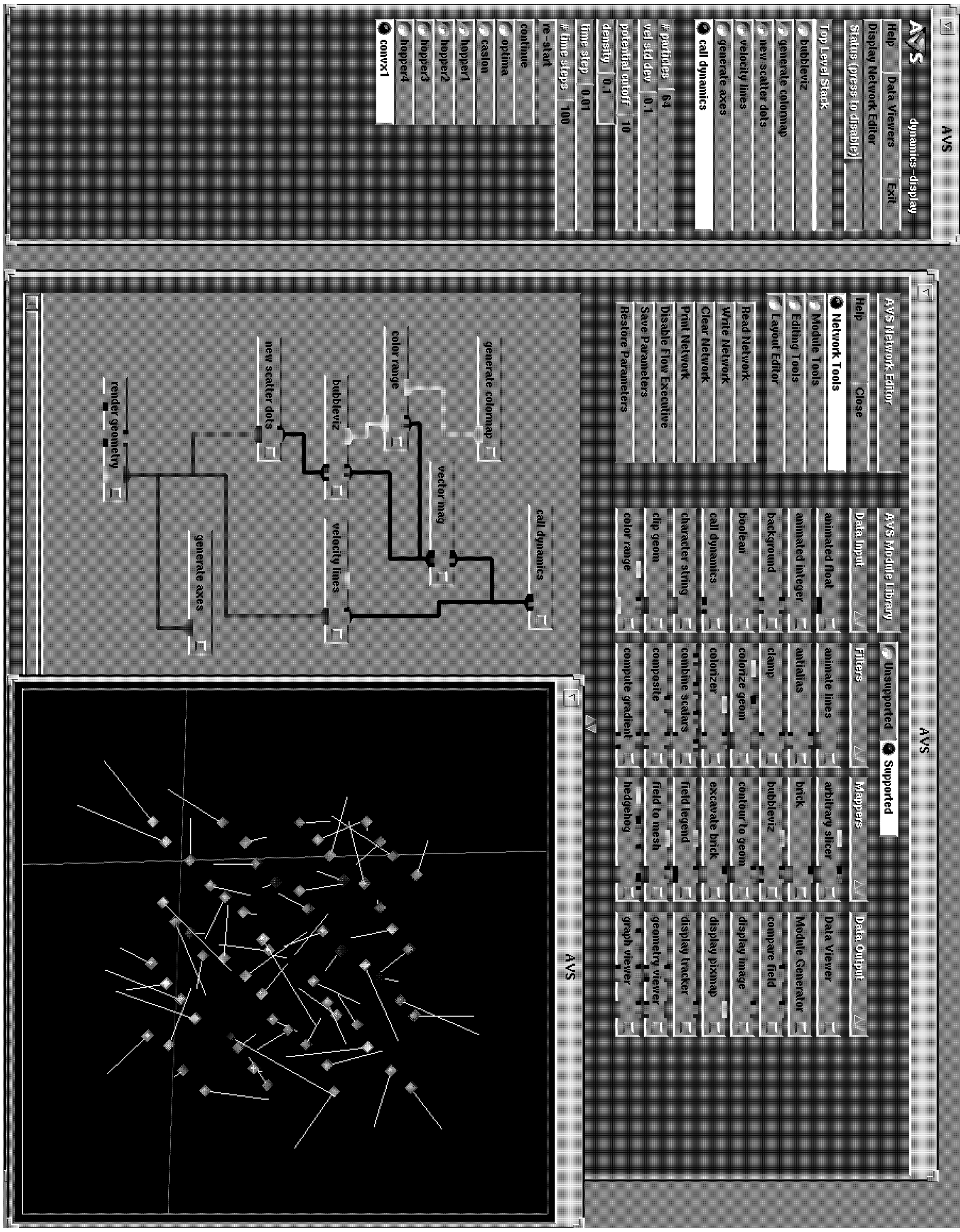
[Sund92b] Sunderam, V.S. and Geist G.A. Supercomputer level concurrent computation on a network of IBM RS/6000 powerstations. *Scientific Excellence in Supercomputing*, K. Billingsley, H. Brown, E. Derohanes (eds). Baldwin Press, pp. 779-804, March 1992.

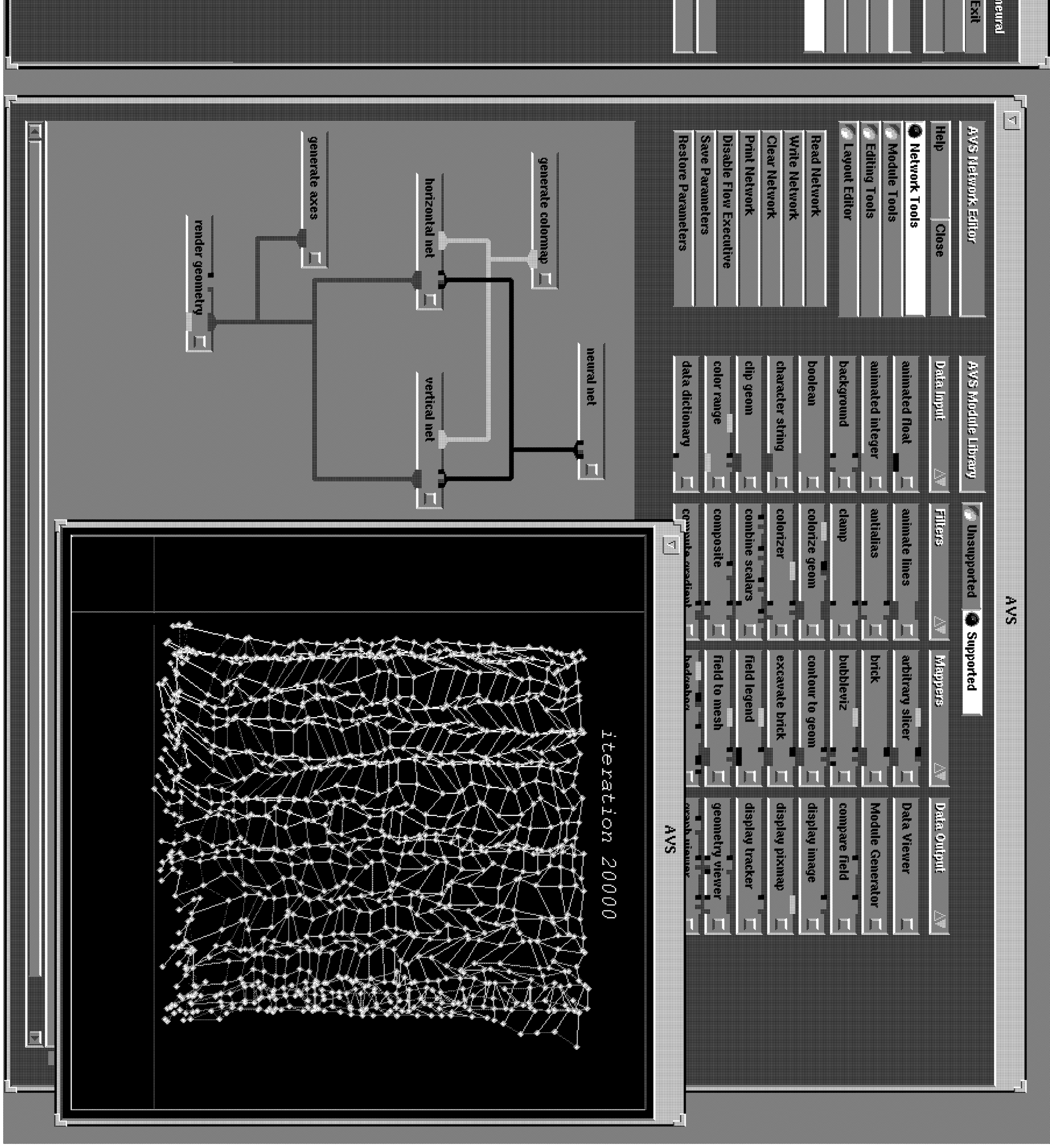
[Vand90] VandeWettering, M. apE 2.0, *Pixel 1*, 4 (Nov./Dec. 1990), 30-35.

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

Appendix: Screen Snapshots

The following two pages are snapshots taken during the execution of the molecular dynamics and neural net applications, respectively. 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
dynamics-display
Help
Data Viewers
Exit
Display Network Editor
Status (press to disable)
Top Level Stack
bubbleviz
generate colormap
new scatter dots
velocity lines
generate axes
call dynamics
particles
64
vel std dev
0.1
potential cutoff
10
density
0.1
time step
0.01
time steps
100
re-start
continue
optima
caslon
hopper1
hopper2
hopper3
hopper4
convx1
AVS Network Editor
Close
Network Tools
Module Tools
Editing Tools
Layout Editor
Read Network
Write Network
Clear Network
Print Network
Disable Flow Executive
Save Parameters
Restore Parameters
generate colormap
color range
bubbleviz
new scatter dots
render geometry
vector mag
call dynamics
velocity lines
generate axes
AVS Module Library
Unsupported
Supported
Data Input
animated float
animated integer
background
boolean
call dynamics
character string
clip geom
color range
Filters
animate lines
antialias
clamp
colorize geom
colorizer
combine scalars
composite
compute gradient
Mappers
arbitrary slicer
brick
bubbleviz
contour to geom
excavate brick
field legend
field to mesh
hedgehog
Data Output
Data Viewer
Module Generator
compare field
display image
display pixmap
display tracker
geometry viewer
graph viewer


AVS
AVS Network Editor
Help
Close
Network Tools
Module Tools
Editing Tools
Layout Editor
Read Network
Write Network
Clear Network
Print Network
Disable Flow Executive
Save Parameters
Restore Parameters
AVS Module Library
Unsupported
Supported
Data Input
animated float
animated integer
background
boolean
character string
clip geom
color range
data dictionary
Filters
animate lines
antialias
clamp
colorize geom
colorizer
combine scalars
composite
Mappers
arbitrary slicer
brick
bubbleviz
contour to geom
excavate brick
field legend
field to mesh
Data Output
Data Viewer
Module Generator
compare field
display image
display pixmap
display tracker
geometry viewer
neural net
generate colormap
horizontal net
vertical net
generate axes
render geometry
iteration 20000
Exit