Monitoring and Controlling Remote Parallel Computations Using Schooner*

Zhanliang Chen and Richard D. Schlichting

TR 94-32

Abstract

Scientific visualization systems such as AVS have the potential to help users of parallel systems monitor and control their computations. Unfortunately, the machines most suitable for visualization systems are not the parallel systems on which the computation executes, often leading to the use of two distinct machines and the viewing of results only after the computation has completed. Here, an approach to solving this problem is presented in which AVS and a remote parallel computation are incorporated into a single metacomputation using the Schooner software interconnection system. This scheme gives the user enhanced control, including the ability to dynamically select the parallel platform to be used, monitor the progress of the computation, and modify parameters. An example application is described in which AVS and a parallel neural net code executing on either an Intel Paragon, Sequent Symmetry, or PVM-based Sun Sparcstation cluster are interconnected. These experiments demonstrate not just the feasibility of structuring parallel computations as part of a larger metacomputation using Schooner, but also that these benefits can be achieved with only modest cost.

December 11, 1994

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

*This work supported in part by the National Science Foundation under grant ASC-9204021

1 Introduction

The process followed in solving a scientific problem using computer simulation is typically quite iterative. Initial values are determined for parameters, the code is executed, the results are analyzed, and then the process repeated again with updated parameter values. These steps are iterated until the final results are deemed satisfactory. Given that the person running the experiment is usually involved at every step, the result can be a significant time investment. Hence, any success in reducing either the execution time of the computation or the time needed for analysis and reexecution has the potential to offer significant benefits.

In this paper, we describe an approach that addresses both aspects of the problem by combining parallel execution of the computation with improved facilities for monitoring, configuring, and controlling the resulting execution. Parallelism is, of course, widely recognized as a technique for reducing the execution time of scientific computations, either by using parallel hardware such as an Intel Paragon or virtual parallel machines with software systems such as PVM [Sun90]. Enhanced facilities for monitoring and steering the execution, which can be based on visualization systems such as AVS [AVS92], allow the user to view results and make appropriate parameter changes in a convenient way. The problem—and the focus of this paper—is that the machines most suitable for visualization systems are not the parallel systems on which the computation executes. Rather, users must normally run the computation to completion, and then manually transfer the results to a second machine for visualization and analysis. This decouples the computation from the visualization system, thereby limiting the ability of the user to monitor and control the computation.

Our approach involves incorporating the visualization system and parallel computation into a single *heterogeneous distributed program* or *metacomputation* [KPSW93] using the Schooner software interconnection system [HS94b]. With this scheme, Schooner is used to transfer data and control transparently with remote procedure call (RPC) semantics between AVS executing on a workstation and parallel computations running on a variety of platforms, including an Intel Paragon, a Sequent Symmetry, and a collection of Sun Sparcstations using PVM (Figure 1). Coupling the visualization and computation components together in this way gives the user enhanced control, such as the ability to dynamically select the parallel platform to be used, monitor the progress of the computation, and modify parameters. Here, we report on experiments conducted using a parallel code implementing a short-cut version of the Kohonen self-organizing neural network. These experiments demonstrate not just the feasibility of structuring parallel computations as part of a larger metacomputation using Schooner, but also the relatively small performance cost associated with the extra data and control transfers. An earlier paper demonstrates similar conclusions for sequential and vector computations [HS94a].

This paper is structured as follows. Section 2 gives some background on Schooner, AVS, and the neural net application. Section 3 then describes the details of our experiments, including how the neural net computation was parallelized, how Schooner was used to connect the computation with AVS, and the type of timing measurements taken. The timing results, plus description of changes made to Schooner to optimize the type of data transfer common in scientific applications, are then given in section 4. Finally, section 5 presents more details

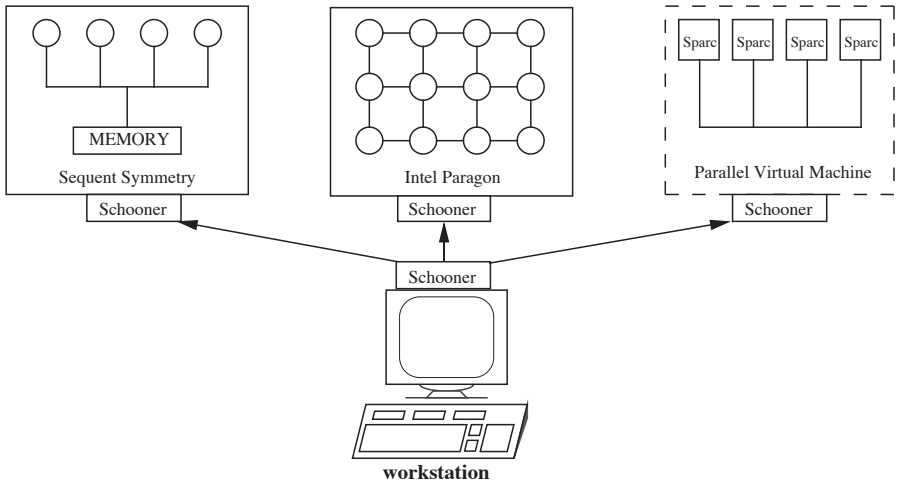

Figure 1: Connecting a parallel computation with AVS

about related work, including PVM and other message-passing systems, while Section 6 offers some conclusions.

2 Background

2.1 Schooner Interconnection System

Schooner is a software interconnection system designed to facilitate the construction of scientific applications that require access to heterogeneous software and hardware resources in the Internet. Under Schooner, a program is constructed from multiple *components*, each of which is composed of one or more procedures written in the same programming language. At runtime, each component is instantiated as a process on a specified host machine, where the architecture and physical location of the machine can vary from component to component. Procedures within one component can make calls to procedures in other components, with the transfer of control to the remote machine and differences in machine architecture being handled transparently by the Schooner runtime. Thus, Schooner realizes an application-level *heterogeneous RPC* facility.

To realize this functionality, Schooner provides four services:

1. An *interface specification language*, which is used to specify the number and type of parameters for procedures in Schooner components. An *export specification* is given for each procedure that a component makes available for remote invocation, while an *import specification* is given for each external procedure potentially invoked by a component.

2. An *intermediate data representation*, which specifies how data is represented as it is transmitted from one component to another. The interface specification language and the

intermediate data representation used in Schooner are collectively called the *Universal Type System* (UTS) [Hay89].

3. *Stub compilers*, which convert the import and export specifications into C or Fortran stub routines. These stub routines automatically handle the encoding and decoding of data between the UTS intermediate representation and the representation of the host machine.

4. A *runtime system*, which implements the data transfer, configuration, and name mapping aspects of the systems. Specifically, its job includes starting and stopping components, mapping procedure names to components, marshalling RPC arguments into messages, and implementing the message passing needed to transfer control from one component to another. As part of the runtime system, each application has a *Schooner manager* process that acts as the configuration manager for the application, handling tasks such as mapping procedure names to the host IP address and port number of the component that exports the procedure. A *Schooner server* process runs on each host involved in the application to start components on that host, and to allow the components to connect to the Schooner manager.

Figure 2 illustrates the metacomputation programming model supported by Schooner. Note that parallel computations can be encapsulated within a Schooner component, a key feature exploited in this work.

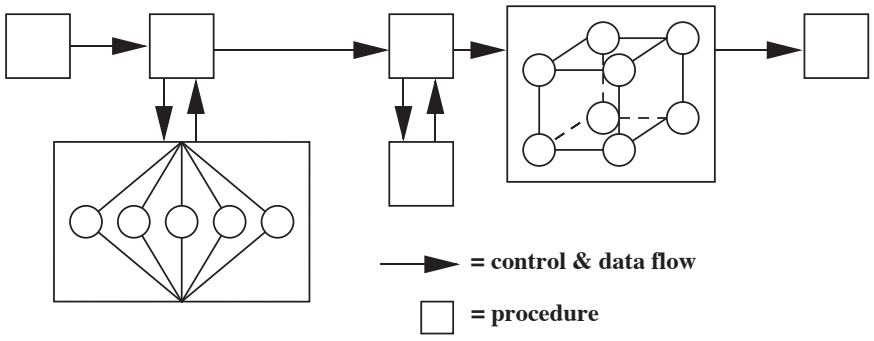

Figure 2: Programming model supported by Schooner

2.2 AVS Scientific Visualization System

AVS is a graphics system for visualizing results from scientific computations. The data to be visualized is generated and/or manipulated by software modules that are connected into a dataflow graph or *network* using the AVS network editor. Figure 3 shows the AVS network from the neural net application presented below, where the rectangles are modules and the lines represent the dataflow connections. Values that are generated by a module flow over the network as dictated by the connections to become the input of other modules. Typically, an

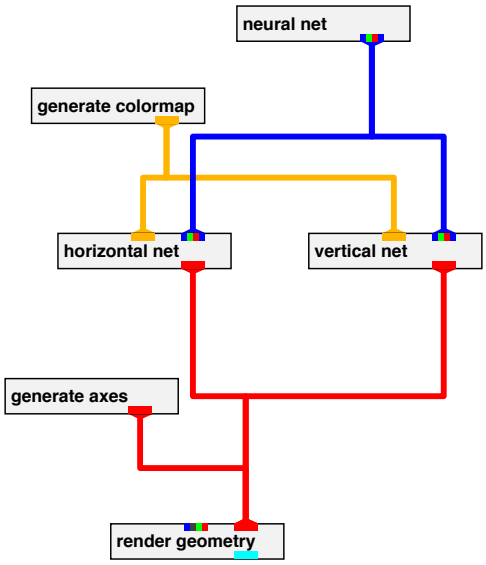

Figure 3: AVS network from neural net application

AVS network has a module as the source of the data. This module may generate the data by doing the computation internally, by reading from a file, or in our case, by making a Schooner RPC to a remote component to perform the computation. This data then flows through some number of filter modules that transform it into a form that can be displayed. The task of actually generating the display on the screen is done by various output modules. AVS also allows modules to create *widgets* such as dials, sliders, or type-in windows that can be used to set parameter values for the module. A complete AVS screen dump for the parallel neural net application is found in the Appendix.

AVS modules can either be user-written or selected from a large collection of standard modules. Two types of modules are supported: *procedure modules* and *co-routine modules*. A procedure module is scheduled by the AVS kernel, and executes once whenever one or more of its inputs have changed. A co-routine module, on the other hand, is scheduled independently of the availability of input values.

2.3 Neural Net Application

As an example application, we used 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 neural net consists of a two-dimensional array of points representing processing elements. These processing elements are allowed to move inside a unit square, with the goal being to form a uniform grid. At the end of the computation, a plot with the points connected to each other should show uniformly spaced vertical and horizontal lines. Initially, all points are placed at some random distance from their ideal positions. They then use the position of their neighbors in a learning process to alter their

position within the grid.

The neural net computation is an iterative process, where each processing element moves once during each iteration. The movement is organized so that a processing element is more likely to move toward its ideal position than away from it. The time needed to compute the movement of one element does not depend on the size of the neural net, so the overall execution time of each iteration is proportional to the number of processing elements.

3 Experiments

To demonstrate the feasibility of constructing metacomputations incorporating both parallel codes and AVS, we ported the neural net application to a variety of parallel platforms and used Schooner to connect it with AVS running on a workstation. Several steps were needed to accomplish this. First, the neural net code had to be parallelized. Second, several AVS modules were written to control the computation and display the results. Finally, a metacomputation was configured using Schooner; this involved writing UTS specifications for both the AVS module and neural net code, running the stub compilers to generate interface code, and then initiating the computation. These steps and the issues that had to be addressed in each are discussed below, followed by a short description of the overhead involved in using Schooner for such metacomputations.

3.1 Parallelizing the Neural Net Code

Transforming the neural net code from a sequential into a parallel application was relatively easy using standard domain decomposition. The two-dimensional grid of processing elements is divided along the x direction into several groups of equal sizes, each group for one physical processor. After each iteration, processors exchange boundary conditions with their neighboring processors.

One issue concerned how to communicate the results from the multiple threads in the parallel execution to the single active thread supported within Schooner's execution model. The approach chosen was to use one *master process* to act as an interface between Schooner and the multiple *slave processes* performing the actual work. To realize this model, the master process is configured as a Schooner component, while the slave processes are not. Certain procedures in the master process are then exported using Schooner and invoked by AVS to perform the actual computation. When these procedures are called, the master process sends commands to the slave processes using the communication facility provided by the parallel architecture. The slave processes perform the computation and send their results to the master process, which returns them to the invoking AVS module.

3.2 Writing AVS Modules

Three AVS modules in Figure 3—`neural net`, `horizontal net`, and `vertical net`—were written to collect the results of the computation and transform the data so it could be displayed by the standard library module `render geometry`. The first we term

the *control module*. This module is configured as a Schooner component and executes as an AVS co-routine. Its task is to invoke the remote computation component, gather the results, and send them along the dataflow network. It also creates a number of widgets on the screen to allow the user to set the parameters of the neural net and control the computation. We call the second and third modules *visualization modules*. They are are responsible for connecting the neural net points horizontally and vertically, respectively; the second also generates the spheres that represent the points. These modules are both procedure modules since there is no need to modify the graphics display unless the data from the first module changes. The output from both modules are sent to the `render geometry` display module mentioned above.

The overall structure of the neural net metacomputation is illustrated in Figure 4.

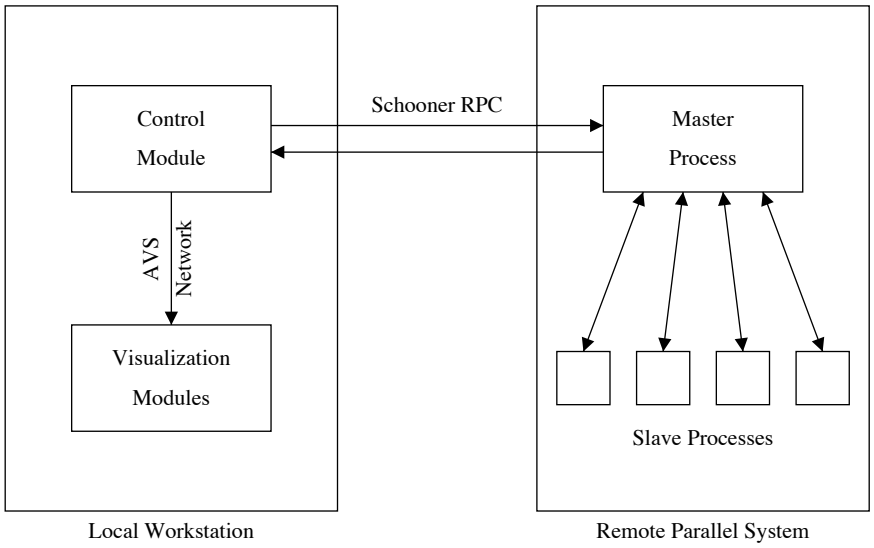

Figure 4: Metacomputation structure

3.3 Interconnecting Components using Schooner

To allow access to the computation component using Schooner, the code for the master process exports two procedures: `configure_net`, which handles the initial semi-random placement of the processing elements, and `compute_net`, which executes the neural net computation. The UTS specifications for these two procedures are:

```
export configure_net prog(
    "side"      val float,
    "size"      val integer,
    "num_procs" val integer,
    "rep_x" res rep array[-] of float,
    "rep_y" res rep array[-] of float)
```

```
export compute_net prog(
    "alpha"     val float,
    "maxlearn"  val integer,
    "size"      val integer,
    "current"   var integer,
    "increment" val integer,
    "num_procs" val integer,
    "rep_x" res rep array[-] of float,
    "rep_y" res rep array[-] of float)
```

Here, `size` specifies the number of rows and columns in the neural net; that is, the net has `size` $\times$ `size` processing elements. `num_procs` specifies the number of physical processors to be used for the parallel computation. In the parameters to `compute_net`, `increment` is used to control the number of iterations to execute during the current call. `rep_x` and `rep_y` are used to return the x and y coordinates of the points—i.e., the results of the computation—back to the caller. These specifications are used as input to the C stub compiler to generate the C stub routines.

Note that the sizes of arrays `rep_x` and `rep_y` are given as a minus sign, meaning that the number of elements is unspecified at configuration time. This allows the size of an array to vary dynamically from call to call, a feature that adds flexibility and eliminates the need to specify (and transmit) arrays of some maximum size. Here, the size of the arrays depends on the size of the neural net; more specifically, each array has `size` $\times$ `size` elements. Schooner requires that variable size arrays such as this be declared as *represented arguments*, which is indicated by the keyword `rep` in the above specifications. Encoding and decoding of such arguments are not done automatically in the stub, but rather must be handled explicitly by the user with Schooner library routines. For example, the following code is used to encode the values for the `rep_x` array:

```
long rep_x;
int size_squared;

size_squared = size * size;
/* get space for array plus header */
user_rep_new(rep_x,size_squared * (UTS_FLOAT_SIZE + UTS_TAG_SIZE) + 50);
/* encode the array */
uts_encode_float_array(rep_x,x,1,&size_squared);
```

It is worth noting that this code is much simpler than the analogous code in [HS94a], where each individual element is encoded individually using a loop. In doing this neural net example, we discovered that optimized encoding and decoding routines for homogeneous arrays of floats resulted in a significant time savings for large arrays, as well as simplifying the task of the programmer. Section 4.2 elaborates on this issue.

3.4 Application Startup

An interesting aspect of a metacomputation of this type is how the different components are started, especially those related to parallel computations. In this case, as with any other

Schooner application, the user must first start Schooner servers on both the local workstation on which AVS is executed and the remote system that hosts the parallel computation. The user then starts the Schooner manager and AVS on the local workstation. Once AVS is running, the user starts the control module and visualization modules by constructing the AVS network shown in Figure 3; alternatively, the network can be read from a file. Depending on the particular parallel architecture, some additional initialization may also be necessary. For example, if PVM is being used, PVM daemons must be started on all hosts that make up the virtual machine.

The control module contains widgets that allow the user to specify the size of the neural net, enter initial parameters, choose the name of the host on which to execute the parallel computation and how to achieve parallelism (e.g., native library versus PVM), and select the number of processors to be used for the parallel computation. There are also two control buttons, labeled "awake" and "sleep" respectively, for starting and pausing the computation. These widgets are shown in Figure 5.

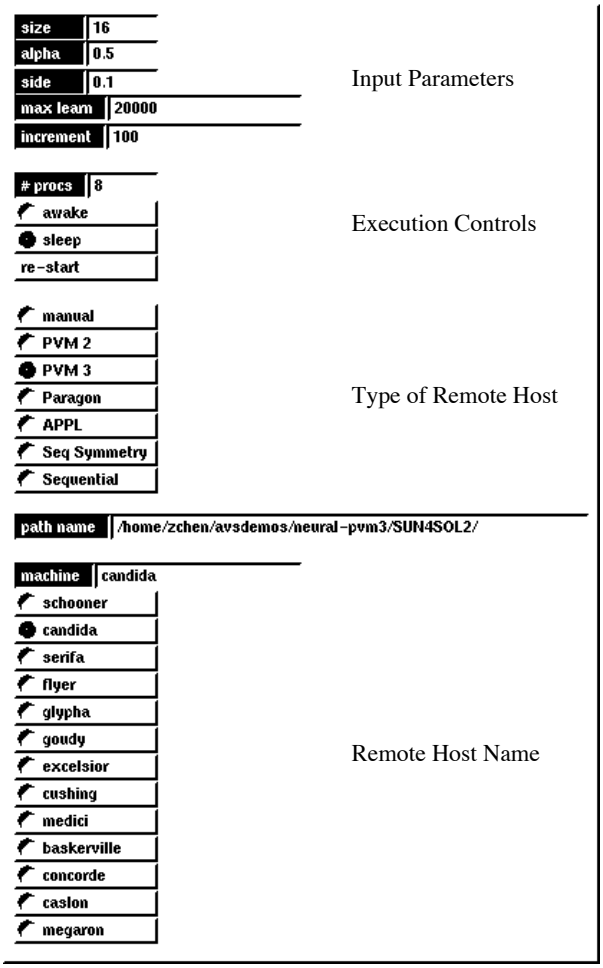

Figure 5: AVS widgets for controlling computation

After setting the parameters, the user clicks the "awake" button on the screen to start the computation. The control module notices that the computation component has not yet been started, and calls the Schooner library routine `sch_start_component` to start the component on the remote host specified by the user. This new component becomes the master process of the parallel computation.

The control module then invokes the `configure_net` procedure exported by the master process to initialize the neural net. Slave processes for doing the parallel computation are also started at this time. The master process for different parallel architectures uses different ways to start the slave processes. For example, if a PVM virtual machine is used, `pvm_spawn` is called, while on an Intel Paragon, `nx_load` is used. After the slave processes are started, the master process uses the communication mechanism provided by the parallel architecture to send a "configure" command to all slave processes. Each slave process initializes its own portion of the neural net, and sends the initial configuration back to the master process. The master process then returns from the `configure_net` procedure, returning the configuration results to the control module.

Thereafter, the control module calls `compute_net` repeatedly to carry out the computation, until it is halted by the user or a pre-determined number of total iterations have been reached. It is important to note that each call causes only some portion of the total number of iterations to be executed, not the entire computation. This allows intermediate results to be returned for display and monitoring by the user. Such a facility is feasible only when the visualization and computation components are configured together in a metacomputation, a significant benefit of this approach. Finally, using the widgets provided by the control module, the user can change the size of the neural net or other computation parameters, restart the computation from the beginning, or even kill the current computation component and start a new one on another, possibly different, remote host.

3.5 Overhead

In addition to the time spent by the parallel processes performing the actual computation, there are several types of overhead involved in the neural net application.

- *Schooner overhead*. Time spent to start the remote computation component, encode and decode arguments, and perform RPCs.
- *Parallel process communication overhead*. Time spent by the slave processes exchanging boundary conditions after each iteration in the parallel computation. Also, we include the time spent on communication between the master and slave processes in this category, since this varies depending on the parallel architecture.
- *Visualization overhead*. Time spent by various AVS modules converting the results into the graphics seen by the user.

Note that some of the above actions may take place concurrently with each other or the parallel computation, so that the total computation overhead is not a simple sum of the values

for each. For example, when the AVS modules are working on graphics visualization, the computation component may have already started the next round of computation.

Of the three types of overheads, we are most interested in the Schooner overhead since this, in some sense, quantifies the cost associated with enhanced monitoring and control of the parallel computation. The parallel process communication overhead is inherent in this type of parallel computation, while the visualization overhead cannot be avoided if we want to see the graphical display of the result. Also, as noted above, the visualization process can proceed concurrently with the computation.

4 Timing Results

4.1 Measurement

To determine the overhead added by our system, we measured the execution time of the neural net application for several different parallel architectures: an Intel Paragon XP/S Model A4, a Sequent Symmetry S81, and a virtual machine formed by a cluster of 8 Sun Sparc IPCs using PVM version 3.3.4. The local workstation used for visualization is a Sun Sparc 10/41. All hosts are on the same 10 MB/sec Ethernet, but several are separated by bridges. The times reported below are for each Schooner invocation from the AVS control module to the parallel computation, where 100 iterations are executed on each call. For comparison, approximately 200,000 iterations are needed in the 32 by 32 case to reach a reasonable final alignment.

		16×16	24×24	32×32	64×64
Sparc 10 sequential time		1.823	4.098	7.276	29.264
Paragon sequential time		1.651	3.711	6.598	26.457
1 proc	parallel comp	1.783	3.855	6.748	26.604
	Schooner overhead	0.059	0.073	0.105	0.193
	percentage	3.23	1.87	1.54	0.72
2 proc	parallel comp	0.947	1.991	3.442	13.429
	Schooner overhead	0.058	0.073	0.091	0.196
	percentage	5.78	3.54	2.57	1.44
4 proc	parallel comp	0.531	1.060	1.788	6.792
	Schooner overhead	0.063	0.071	0.090	0.195
	percentage	10.60	6.31	4.80	2.79
8 proc	parallel comp	0.324	0.591	0.953	3.458
	Schooner overhead	0.062	0.073	0.089	0.200
	percentage	16.15	10.98	8.51	5.48

Table 1: Execution time for 100 iterations of neural net application (in seconds)

Table 1 gives a comparison between running the computation on the workstation where AVS is running and on the Intel Paragon using different numbers of processors. The first row

of the table shows the different neural net sizes used in the computation, where 16×16 means the neural net processing elements are arranged in a 16 by 16 grid, for a total of 256 points. The second row gives the time it takes to run 100 iterations on the local Sparc 10 workstation. Only the actual computation time is included here, i.e., the code was run without either AVS or Schooner. This time was measured as wall-clock time to be consistent with the numbers reported for the parallel computations below, although the actual CPU time was within a few percentage points. The third row shows similar information for the Paragon; a single node is allocated in the compute partition and a sequential calculation is run on that node. All times are measured in seconds.

The rest of the table gives the execution times for a parallel version of the computation running in the compute partition of the Paragon using 1, 2, 4, and 8 processors. Note that the 1 processor case does not give a real parallel computation, but does provide a baseline for the parallel code. For any given number of processors and neural net size, three numbers are given. The first gives the time spent executing the parallel component. This value is the sum of the actual computation time plus the time spent on communication among the slave processes, and between the master process and the slave processes. The second is the Schooner overhead, including the time to encode and decode arguments and perform the RPC. The third reports the same overhead, but as a percentage of total execution time rather than absolute values. The measurements are done as follows. After the computation component is started, `configure_net` is called to initialize the neural net. Then `compute_net` is called 50 times to do the computation. During each call to `compute_net`, 100 iterations of the computation are performed. The total time spent on the 50 calls to `compute_net` is measured by using the `gettimeofday` function. The total time spent by the slave and master processes is also measured in the same way. The difference between the two gives the Schooner overhead. Finally, the results are divided by 50 to get the average time spent during one call.

Note that `compute_net` is invoked 50 times, with each call performing 100 iterations, instead of doing the full 5000 iterations in one call. As noted above, this allows the user to monitor the progress of the computation, instead of just seeing the final result. The number of iterations per call (100) is selected so that the graphics is updated approximately once every few seconds. Of course, the frequency of monitoring can easily be changed by the user using the control widgets provided by the control module.

From Table 1, we can see that the sequential version of the neural net application runs slightly faster on a single node of the Paragon than on the Sparc 10/41. When more processors are deployed and a parallel version is executed, the Paragon far out-performs the Sparc. For example, with 8 processors on a 64 by 64 neural net, the Paragon only takes 12.5 % as long as the Sparc to perform the computation, even with the communication cost between processes and Schooner overhead taken into account.

As expected, the Schooner overhead does not depend on the number of processors used for computation. This is because Schooner is only used to connect the AVS control module with a single master process on the parallel machine, no matter how many slave processes are used for the computation. However, the percentage of time spent by Schooner naturally increases as the number of processors increases since the overall computation time decreases. In a realistic situation, the user may compensate for this by running more iterations during

each call to `compute_net`, thus keeping the duration of each call unchanged.

The Schooner overhead also increases as the size of the neural net increases, but at a slower rate than the computation time. As noted above, this overhead is mainly composed of two parts: the time to encode and decode the arguments, and the time to send the arguments and other control information over the network to realize RPC semantics. Among the arguments, the arrays used to return the results to the AVS module take the most space, and hence, the most time to encode and decode. Moreover, the size of the arrays is proportional to the number of processing elements in the neural net, and thus, proportional to the overall computation time. As a result, the time for argument encoding and decoding should be roughly proportional to the computation time. This hypothesis has been verified by our experimental results (see Table 3 below). The time for Schooner communication also increases as the size of the arguments increases, but the relationship is less than linear.

Although our goal in this work was not to investigate parallel neural net algorithms, we were nonetheless pleased by the speedup achieved. With a big neural net size of 64 by 64, 8 processors runs 7.65 times as fast as a single processor running the sequential version. Even with a small 16 by 16 neural net, we still achieved 5.10 time speedup. This shows that the communication overhead between processors on the Paragon is small due to the high-speed internal network, and that the algorithm employed has an acceptable computation/communication ratio. As the size of the neural net increases, the actual computation time increases at an $O(n^2)$ rate, where n is the size of the neural net in one dimension. This is verified by the timing of the sequential algorithm. However, the amount of data that needs to be exchanged between each pair of slave processes only increases at an $O(n)$ rate. The actual increase in communication time would be less than linear though, since there is a fixed per-message cost as well.

	Total	Computation	Communication	Schooner	% Schooner
Paragon	1.041	0.825	0.128	0.089	8.51
PVM	5.479	3.939	1.261	0.279	5.09
Symmetry	24.738	24.304	0.221	0.213	0.86

Table 2: Comparison of different parallel architectures (in seconds)

Table 2 compares the three parallel architectures used in our experiments: the Paragon, the 8-node PVM virtual machine, and the Sequent Symmetry. The data given are for a 32 by 32 neural net, with 8 processors allocated for the parallel computation. For each architecture, the table shows the total time spent in one call to `compute_net`, and then breaks it down into three parts: the time for the actual computation, the time for communication between different parallel processes, and the time spent by Schooner. Finally, the percentage of the time spent by Schooner is computed relative to the total time. The measurement conditions are the same as the last table.

Note that the Schooner overhead is less than 10 % of the total time in all three cases. The percentage is the highest for the Paragon since the computation time itself is very small. Note that this is somewhat pessimistic, however, since it is generally not necessary to monitor the computation every second as done here.

4.2 Improvements to Schooner

In the course of performing these experiments, several changes and improvements were made to the Schooner system. The most important was the collection of new library routines for encoding and decoding homogeneous arrays mentioned in section 3.3 above. These routines encode and decode homogeneous arrays of integers, floating-point numbers, or double-precision floats. Arrays of strings, records (C struct), or unions are still processed in the standard fashion.

Below is the code that would be used within the AVS modules to decode an array using the original routines provided by Schooner. Note that such explicit decoding by the user is needed only because the array is of variable size; for fixed size arrays, the automatically generated stub routine handles the decoding transparently to the user.

```
long rep;
float dest[];
int size_squared, num_dims, dims, i;
long index;

size_squared = size * size;

num_dims = 1;
if ( 1 != uts_dims(rep,&dims,&num_dims) )
        /* signal error */
if (dims != size_squared)
        /* signal error */

for (i = 0; i < size_squared; i++) {
        index = uts_index(rep,i);
        if ( 1 != uts_decode_float(index,&dest[i]) )
                /* signal error */
}
```

Here, `rep` is the representative of the array returned by the parallel component, while `dest` is the destination into which the array is to be decoded. The array is expected to be one-dimensional, with `size_squared` elements. The code first gets the actual dimension and size of the array from the representative by calling `uts_dims`, and signals error if either value is incorrect. Next, the array is decoded one element at a time. `uts_index(rep, i)` returns a handle to the `i`th element of the array, which is then passed to `uts_decode_float` to retrieve the actual element.

This approach has two problems. First, it requires an inordinate amount of code to be written by the user. Second, and more importantly, the decoding operation is very slow. The inefficiency lies in `uts_index`, which is a general index function that handles all types of arrays, including arrays with non-uniform element sizes such as arrays of strings or unions. Since `uts_index` cannot determine if all array elements are the same size, it cannot simply calculate the address of the `i`th element by multiplying `i` by the size of an element. Instead, it has to start from the beginning of the array and skip `i` elements, implying that the time spent by `uts_index` is proportional to `i`. In all, the time complexity for decoding the array is $O(size\_squared^2)$ or $O(size^4)$, even though there are only $size^2$ elements.

One way to speed up this array decoding process is to modify `uts_index` so that it can recognize arrays with uniform element sizes. To make this possible, however, the UTS intermediate data representation would have to be modified to add a flag to indicate whether or not the array is homogeneous. While possible, we chose the alternative approach of adding three new functions in the UTS library— `uts_decode_float_array`, `uts_decode_double_array`, and `uts_decode_int_array`—so that homogeneous arrays can be decoded with just one call. This not only makes the decoding faster for the types of arrays most commonly found in scientific applications, but also simplifies the programming task. Analogous routines for array encoding have also been added.

Using the new routines, the code to decode an array becomes:

```
long rep;
float dest[];
int size_squared, num_dims, dims;

size_squared = size * size;

num_dims = 1;
if (uts_decode_float_array(rep,dest,&num_dims,&dims,size_squared) != OK)
        /* signal error */

if (num_dims != 1 || dims != size_squared)
        /* signal error */
```

Table 3 shows the effects of these changes. The first line shows the size of the neural net, and thus the size of the arrays. The second line shows the time spent by a Sun Sparc 10/41 to decode one array using the old method (calling `uts_index`), while the third line shows the time using the new method (calling `uts_decode_float_array`). All times are measured in milliseconds.

	16×16	24×24	32×32	64×64
old decoding routine	26.01	116.45	352.91	5426.13
new decoding routine	0.36	0.82	1.42	5.53

Table 3: Comparison of different array decoding techniques

As can be seen, the savings is enormous, especially for larger arrays.

Finally, note that our approach of adding additional library routines does lose one potential advantage of modifying `uts_index`: the ability to efficiently decode only a specified portion of an array. Our feeling is that this is of secondary concern since decoding an entire array is undoubtedly the more common operation. However, these two options are actually orthogonal, so that, if necessary, `uts_index` could still be changed to make its operation linear in time for homogeneous arrays.

5 Related Work

A number of other software systems are available for interconnecting processes running on heterogeneous hosts, including connecting a visualization tool running on the local workstation with a computation running on a remote machine. PVM [Sun90], p4 [BL92], and APPL [QCB93] realize such connections using message passing, while Sun RPC [SM90] and other RPC schemes implement message-passing semantics similar to Schooner.

The PVM system consists of two parts: a PVM daemon program, and a library of PVM interface routines. A PVM application executes on a virtual machine that is composed of a number of (possibly heterogeneous) physical machines. All processes inside the application are linked with the PVM library, which handles the message passing between processes. Library services are also available to start new processes within the virtual machine, or to add or delete hosts to/from the virtual machine. The PVM daemon runs on all member hosts of the virtual machine and is responsible for relaying messages from one physical machine to another. PVM uses Sun XDR for an intermediate data representation. Before sending a message, the user must explicitly call PVM library routines to encode the arguments. Then, on the receiving side, the data must be explicitly decoded.

The goal of PVM is to allow execution of parallel algorithms on workstation clusters, with multiple threads of control executing simultaneously on different processors. The goal of Schooner, on the other hand, is to facilitate connecting multiple components into a single logical metacomputation spanning different architectures, where only a single logical thread of control is required. Thus, Schooner presents a higher-level model that complements PVM by allowing parallel computations written using PVM to be configured into larger metacomputations.

APPL and p4 are similar to PVM, in that they both provide a way to pass messages between processes running on possibly different hosts. They both optimize the communication between two processes running on the same host by using shared memory whenever possible. p4 provides data conversions between different architectures, while APPL does not. APPL also requires that all processes be started by an *initiator process*, so that an independently started process cannot easily join other processes to become part of an APPL application. This limitation makes it hard to use APPL to connect visualization tools like AVS with computation processes, since AVS modules must be started by AVS itself.

Sun RPC and other RPC systems are similar to Schooner in that they provide remote procedure call semantics, with stub compilers, argument marshalling, and underlying message transport. Some include heterogeneity aspects as well, such as automatic data conversion to deal with diverse data representations. However, as a class, such RPC systems are oriented primarily towards use in client/server style operating system services. In contrast, Schooner is designed for user-level applications, and so, supports a more general procedure call model closer to that found in programming languages. For example, unlike many other RPC systems, recursive calls are allowed.

6 Conclusions

The experiments reported in this paper demonstrate the feasibility of combining a remote parallel computation and a local visualization system into a single metacomputation using the Schooner interconnection system. In particular, we were able to take a neural net application, parallelize it on three different parallel platforms, and then connect each remote computation to AVS with relative ease. With the application structured in this way, the user has more flexibility than with traditional approaches. For example, the user can select which parallel platform to use by simply clicking the appropriate control button on the screen, or modify computation parameters using widgets. It is also straightforward to monitor the application by having the parallel component return intermediate results for display. Moreover, these experiments illustrate that such benefits can be achieved with only modest overhead.

References

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

[BL92] R. Butler and E. Lusk. User's guide to the p4 parallel programming system. Technical Report ANL-92/17, Argonne National Laboratory, Oct 1992.

[Hay89] R. Hayes. *UTS: A Type System for Facilitating Data Communication*. PhD thesis, Dept of Computer Science, University of Arizona, Tucson, AZ, 1989.

[HS94a] P. Homer and R.D. Schlichting. A software platform for constructing scientific applications from heterogeneous resources. *Journal of Parallel and Distributed Computing*, 21(3):301–315, June 1994. (Special Issue on Heterogeneous Processing).

[HS94b] P. Homer and R.D. Schlichting. Using Schooner to support distribution and heterogeneity in the Numerical Propulsion System Simulation project. *Concurrency–Practice and Experience*, 6(4):271–287, June 1994.

[KPSW93] A.A. Khokhar, V.K. Prasanna, M.E. Shaaban, and C. Wang. Heterogeneous computing: Challenges and opportunities. *IEEE Computer*, 26(6):18–27, June 1993.

[QCB93] A. Quealy, G. L. Cole, and R. A. Blech. Portable programming on parallel/networked computers using the Application Portable Parallel Library (APPL). Technical Report Technical Memorandum 106238, NASA, Jul 1993.

[SM90] Inc. Sun Microsystems. *Network Programming Guide (Revision A)*. Sun Microsystems, Inc., Mountain View, CA, March 1990. Part number: 800-3850-10.

[Sun90] V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency–Practice and Experience*, 2(4):315–339, Dec 1990.

Appendix: Screen Snapshot of Neural Net Application