Supporting Heterogeneity and Distribution in the Numerical Propulsion System Simulation Project

Patrick T. Homer
Richard D. Schlichting

TR 92-38a

ABSTRACT

The Numerical Propulsion System Simulation (NPSS) project has been initiated by NASA to explore the use of computer simulation in the development of new aircraft propulsion technology. With this approach, each engine component is modeled by a separate computational code, with a simulation executive connecting the codes and modeling component interactions. Since each code potentially executes on a different machine in a network, a simulation run is a heterogeneous distributed program in which diverse software and hardware elements are incorporated into a single computation. In this paper, a prototype simulation executive that supports this type of programming is described. The two components of this executive are the AVS scientific visualization system and the Schooner heterogeneous remote procedure call (RPC) facility. In addition, the match between Schooner's capabilities and the needs of NPSS is evaluated based on our experience with a collection of test codes. The basic conclusion is that, while Schooner fared well in general, it exhibited certain deficiencies that dictated changes in its design and implementation. This discussion not only documents the evolution of Schooner, but also serves to highlight the practical problems that can be encountered when dealing with heterogeneity and distribution in such applications.

December 31, 1992
revised: December 1, 1993

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

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.

Supporting Heterogeneity and Distribution in the
Numerical Propulsion System Simulation Project

1. Introduction

The Numerical Propulsion System Simulation (NPSS) project, sponsored by NASA Lewis Research Center, aims to reduce the high cost of designing and implementing new propulsion technologies by using computer simulation [Claus92, Claus91]. Specifically, the project, which is part of the High Performance Computing and Communications (HPCC) initiative, involves developing both computational codes to model various engine components, and a simulation executive to control the simulation and model component interactions. Codes have already been written for a number of engine components, with others currently under development. The hardware used by these codes ranges from vector processors to parallel machines to clusters of workstations. Work on the simulation executive is also underway. The focus in this effort is on providing sophisticated capabilities to interact with codes, as well as the ability to substitute different codes at varying degrees of fidelity. A scientific visualization system such as AVS [AVS92] or Khoros [Rasure91, Mercurio92] will likely form a major component of the finished product.

The nature of the NPSS project makes it imperative that the simulation executive support *heterogeneous distributed processing*, in which diverse software and hardware elements on local- or wide-area networks are incorporated into a single simulation. Providing such support poses many non-trivial problems. For example, a given code may be suitable for execution only on a specific machine architecture, thereby requiring the ability to pass data and control transparently to and from codes executing elsewhere. The role of the executive, then, is to act as the ‘‘glue’’ that connects new and existing codes to produce a program that encompasses all aspects of a given simulation. The user of such a program should not see individual codes that execute in isolation, but rather a single integrated program. In addition to being simpler and more intuitive, such a model allows interaction capabilities not possible when codes are run separately. For example, intermediate results can be viewed and parameters modified to affect subsequent parts of the computation; long running computations can easily be monitored and controlled.

This paper describes how heterogeneous distributed processing is supported in a prototype version of the NPSS simulation executive. The key to this capability is the Schooner interconnection system [Homer92a, Homer92b], an application-level remote procedure call (RPC) facility that includes a simple specification language, an intermediate data representation, a collection of stub compilers, and a run-time system implementing cross-machine control transfer. These pieces, together with an execution framework provided by AVS, form a simple simulation executive that supports heterogeneous distributed processing with capabilities approaching those required by NPSS.

In addition to describing the way in which this executive works, we also analyze how well Schooner's capabilities matched the needs of NPSS. Where deficiencies in Schooner were discovered, remedies are outlined; most have already been incorporated, while others are in the implementation stage. Discussing how Schooner has evolved in this way not only documents the evolution of the system, but also serves to highlight the practical problems that can be encountered when dealing with heterogeneity and distribution in such applications.

This paper is organized as follows. In Section 2, we describe the NPSS project in more detail, including its hardware and software aspects. The way in which heterogeneity is supported in NPSS using Schooner is the topic of Section 3, while Section 4 outlines how Schooner has and will change based on this experience. Finally, Section 5 offers some conclusions.

2. The NPSS Project

2.1. Overview

The NASA Numerical Propulsion System Simulation (NPSS) is designed to reduce the cost of designing and implementing aircraft propulsion systems through the use of numerical computation and simulation. Typically, one major difficulty in the design process lies in understanding the interaction among engine components. In the past, these interactions could not be studied until the engine components were built and tested, too often resulting in major re-design at a fairly late stage in the development cycle. The goal of NPSS is to provide improved component codes and a numerical testbed where these engine components can be tested together and interactions between components identified at a much earlier stage.

The project has two aspects. One is the improvement of existing codes and the development of new codes to model engine components. Existing codes are being improved to take advantage of improvements in current hardware. New codes are being developed to make use of new advances in massively parallel machines and clustered workstations. However, even with these improved machines and algorithms, it will still prove too costly to simulate all the components of an engine in complete detail. Thus, five levels of fidelity are being used; these range from level 1, a steady-state thermodynamic model, to level 5, a three-dimensional time accurate model.

The second aspect is development of a simulation executive that enables a user to select from the available codes to construct a complete engine model. This is the computational equivalent of an engine test cell, with a primary motivation being to simulate the interactions between engine components by exchanging data values and boundary conditions between the codes modeling different parts of the engine. The user will be able to model the entire engine or a subset of the engine through the specific component codes selected. The model will typically have most of the engine components at the same level of fidelity, although one or two components of interest will often be modeled at a higher level. This strategy allows a reasonable compromise between the need to model the entire engine to capture properly the interactions among the components, and the need to keep the computational costs and time factors at affordable levels.

2.2. Hardware

NPSS will use a variety of hardware architectures. Historically, vector machines have been used in simulating engine components. NPSS intends to use these existing codes and the machines they run on. Improvements in vector machines, particularly those improvements that result in faster execution, will allow these codes to execute in less time and/or include additional physics in the simulation.

A major goal of NPSS, however, is to bring parallel algorithms into more common use in engine simulation. This effort seeks to achieve higher performance at lower cost than can be achieved with sequential algorithms and platforms. Again, the performance improvements gained can be used to speed the simulation, or traded to allow more detail. The effort includes current machines such as the Intel i860 and the CM-5, as well as planning for future generations of massively-parallel machines and clusters of workstations. In some cases, parallel machines can also provide a more natural solution to a problem, i.e., being able to assign one processor to each blade row when modeling an engine fan.

It is quite likely that all the component codes for an engine will not run optimally on the same type of hardware. Thus, mechanisms must be incorporated into NPSS that will allow codes to communicate across machine boundaries. In many cases, this will result in communication across long distances, since not every site will have local access to all the types of machines needed to run a complete engine simulation. Part of NPSS, then, is to explore methods for efficiently running simulations in such a widely-dispersed, heterogeneous environment. The intent here is to take advantage of advances in network hardware to improve the bandwidth between nodes, and improvements in network software to reduce latency.

2.3. Software

The NPSS software effort is concentrating in three areas:

- Connecting codes that execute on different architectures and/or use different programming models,
- Integrating component simulations at different levels of fidelity and incorporating various software packages, such as graphics tools, into the system, and
- Improving user interaction with the simulation.

In the first area, the key problem is dealing with data exchange between a variety of different machines. This facet involves not only solving the problems associated with differing data formats, but also potentially the need to communicate between different computational models. A parallel algorithm, for example, needs to be able to collect scattered values to pass on to a sequential algorithm executing on a vector machine or workstation. As another example, differences in the ability of machines to handle communications will need to be accommodated in the design of the application and algorithms. Bottlenecks, such as occur when fast machines are talking to slow machines, need to be addressed. In some cases, simple buffering to allow the slow machine to catch up will be sufficient. In others, the slower machine may need to filter the data selectively rather than attempt to use all of it.

In the second area, a major goal is *zooming*, that is, integrating codes that model at different levels of fidelity into the same simulation. Achieving such a capability involves developing techniques to extract, for example, the essential data from a higher-level computation for passing to a lower-level analysis. Another goal is to take advantage of existing software when available. This includes the incorporation of existing codes to model engine components and the use of such tools as graphics packages for displaying results. Having the ability to handle multiple graphics packages, for example, will allow a particular code to be incorporated without the need to convert its output.

In the third area, the rationale is that the user is ultimately responsible for deciding the right tradeoffs in applications such as NPSS. For example, it is the user who must decide on such issues as whether a non-optimum local machine is better than an optimum remote machine, or on what the best level of fidelity for each engine component should be. Thus, the system has to provide reasonable default actions, while still allowing a high degree of user interaction. This interaction extends not only to the selection of which engine components to model, but also to the setting of parameters, both for the individual codes and for the simulation as a whole. The user will also need the ability to monitor the simulation through selectively viewing graphical results or monitoring particular values from selected component codes.

2.4. The Simulation Executive

The NPSS simulation executive is intended to bring together all the individual codes and to coordinate them to simulate the entire engine. The overall goal is a system that allows the user to

- Bring up one of a choice of complete or partial engine simulations,
- Choose a set of operating conditions, i.e., high or low altitude, moist or dry air,
- Modify the engine model by substituting different codes for one or more engine components,
- Set starting parameters for the engine, and modify them during a simulation run, and
- Build an engine from scratch by selecting engine components and linking them together.

Such capabilities allow the user to model a wide range of engines and to do so under a variety of conditions. These would include being able to ‘‘start’’ the engine and ‘‘fly’’ it through a flight profile, or to test operation of the engine in the presence of failures.

A prototype of such a simulation executive has been built using a combination of the AVS scientific visualization system and Schooner. Along one dimension, AVS provides a state-of-the art environment for viewing scientific data. In particular, it provides a large number of tools for processing and displaying data. Another important feature of AVS—and the one that is actually most important for the purposes of NPSS—is the ability to create, modify, and save programs using its Network Editor. This editor allows the user to create programs by visually dragging modules into a workspace and connecting them into a dataflow graph. In addition to the modules supplied with AVS, the user is free to write additional AVS modules that can be integrated into a network in the same way. In the context of NPSS, the Network Editor allows the user to incorporate the specific codes needed for a simulation. The dataflow in this case models the flow of air through the engine.

Interaction with the modules is possible through the setting of parameters associated with each component. In AVS, this is realized using ‘‘widgets’’ that appear in control panels as dials, sliders, type-in boxes, etc. Using the widgets, the user is able both to set initial values for each module and also to modify values during execution, giving a great degree of control over each engine component during a simulation run.

3. Supporting Heterogeneity using Schooner

As alluded to in the Introduction, the need to incorporate heterogeneous hardware and software components follows naturally from the goals of NPSS. Heterogeneity is reflected in the nature of jet engines and the different algorithms needed to simulate their various components. It is also reflected in the variety of machine architectures and programming models being used in and developed for the NPSS project. The need to deal with the distribution of these resources is also inherent in NPSS, since different parts of the simulation may execute on different machines. Indeed, given the special-purpose nature of these machines, it is not unreasonable to expect that they may be separated by significant geographic distances.

These issues of heterogeneity and distribution are dealt with in the prototype NPSS simulation executive by using Schooner, a heterogeneous RPC facility. This section presents an overview of Schooner and describes the prototype executive formed by the combination of AVS and Schooner.

3.1. Schooner Overview

Schooner is designed to be an application-level RPC facility that can be used by programmers to invoke procedures on other machines in a straightforward manner despite the complications of heterogeneity and distribution. As such, a Schooner program is designed in the same manner as a normal procedural program, but with the significant advantage that the programmer is not constrained to a single machine, architecture, programming model, or programming language. Instead, procedures can be written that are tailored to the hardware and software combination most suited to a given application. At runtime, the procedures are instantiated as processes, with calls implemented using a message passing library. The Schooner system handles all data conversions and message passing between processes, thereby preserving a familiar and easy method for constructing programs from a collection of procedures.

Being procedure-oriented, the execution of a Schooner program is essentially sequential, with control proceeding from one procedure to the next as shown in Figure 1. Note, however, that this does not preclude the use of parallel algorithms where appropriate; to use such an algorithm, it is only necessary to encapsulate it within a procedure as illustrated in the figure. This allows the use of, for example, a particular hardware platform's native parallel library, or the incorporation of a computation in which a system such as PVM [Sunderam90] is used to achieve parallel execution on a cluster of workstations.

To carry out its task of connecting components and masking heterogeneity, Schooner provides three largely orthogonal services: the Universal Type System (UTS) [Hayes89], which includes a type specification language and intermediate data format, a collection of stub compilers, and a runtime system. The UTS type specification language uses a Pascal-like syntax to describe the parameters that are expected for each procedure. It provides for the common

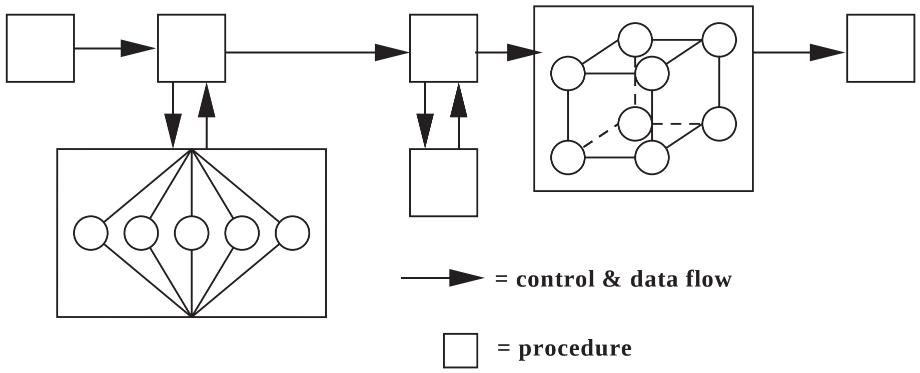

Figure 1 — A Schooner program

simple types such as float, integer, byte, and string, as well as structured types such as arrays and records. An *export specification* is written for each procedure that is to be publically available, while a nearly identical *import specification* is written and associated with the invoking code. UTS also provides a common data interchange format. This is implemented by library functions that handle conversions between a machine's native format and the common interchange format.

A stub compiler is provided for each supported language to read the specification files and produce a stub for each imported and exported procedure. This stub acts as the interface between the user's code and the Schooner runtime. Specifically, it handles the marshaling and unmarshaling of arguments through calls to the UTS library, and utilizes the Schooner library to locate and communicate with the remote procedures. Schooner currently supports C and Fortran, while the predecessor MLP system [Hayes87, Hayes89] also incorporated languages as diverse as Pascal, Icon [Griswold90], and Emerald [Black87].

The Schooner runtime system consists of a communication library and two types of system processes. The communication library is linked with every procedure to handle the sending and receiving of messages implicit in RPC. The system processes are the Schooner Manager and Schooner Servers, respectively. The Manager is responsible for startup and shutdown of processes, maintaining a table of exported procedures and their locations, and performing runtime type-checking of procedure calls based on the UTS specifications. There is one such process per executing program. The Servers are used by Manager processes to start processes on remote machines. There is one Server per machine involved in a given computation.

Systems such as PVM, p4 [Butler92], and APPL [Quealy92] also support heterogeneous distributed processing. However, these systems are oriented primarily towards exploiting clusters of workstations and/or parallel machines to achieve affordable parallel speedup, and as such, support a general message passing paradigm. Schooner, on the other hand, is oriented primarily towards connecting heterogeneous resources to increase the functionality available to the programmer, a task for which RPC is sufficient. Given that RPC is closer to the standard procedural paradigm familiar to most users and simpler to implement, it is a logical choice for the type of problems addressed by Schooner. Also, as compared with these other systems, the availability of UTS simplifies the task of generating the library calls needed to convert and transmit data between machines.

Heterogeneity in scientific applications is also being addressed in several other research projects [Chen93, Freund93, Khokhar93, Wang92]. These projects seek to exploit inherent heterogeneity in the application by automatically partitioning the algorithm to run on a collection of heterogeneous processors. This work includes research in the area of hardware design, as well as low-level software support for communicating among the processors and compiler design to detect heterogeneity. Schooner differs from this work primarily in the granularity of the computation used as the unit of distribution and heterogeneity. Specifically, Schooner deals with connecting together coarse-grained computations and exploiting long-distance networks, whereas other efforts are focusing on detecting and exploiting fine-grain heterogeneity, and improving communication among closely-coupled processors.

Schooner’s use of an external data representation, a specification language, and stub compilers is similar to other RPC systems [Almes85, Birrell84, Sun90, Xerox81]. Several of these systems also emphasize heterogeneity, including Matchmaker [Jones85], Horus [Gibbons87], and HRPC (Heterogeneous RPC) [Bershad87]. Schooner differs from these systems mainly in its orientation toward designing applications in a distributed environment, rather than as a client-server operating system mechanism.

3.2. The Prototype Executive

A prototype NPSS executive has been constructed by combining the capabilities of the AVS scientific visualization system and Schooner. AVS, executing on a workstation, provides visualization capabilities and an execution framework through its dataflow graph of modules. Schooner, in turn, provides the ability to perform the actual computation associated with a module—that is, the simulation code itself—on a remote, potentially heterogeneous, machine.

The executive is being tested using the Turbofan Engine System Simulator (TESS) [Reed93], a complete, one-dimensional engine simulation. TESS represents each of the principal components of an engine as an AVS module. An engine is constructed in the AVS Network Editor by connecting the modules to represent the airflow through the engine. Figure 2 shows an AVS network for modeling an F100 engine using TESS. The control panel for the low speed shaft, one of two instances of the shaft module in the F100 engine, is displayed at the left. Inputs to each module can come from three sources: data passed through the dataflow network from upstream modules, values passed through the widget mechanism, and data files read by the module. The low speed shaft module receives data from the upstream low pressure compressor. Its control panel has widgets that accept user inputs for the *moment inertia*, *spool speed*, and *spool speed-op* parameters. On each execution of the shaft module, the data flow network and the widgets pass values to the module. For modules that also read data files, AVS provides a browser

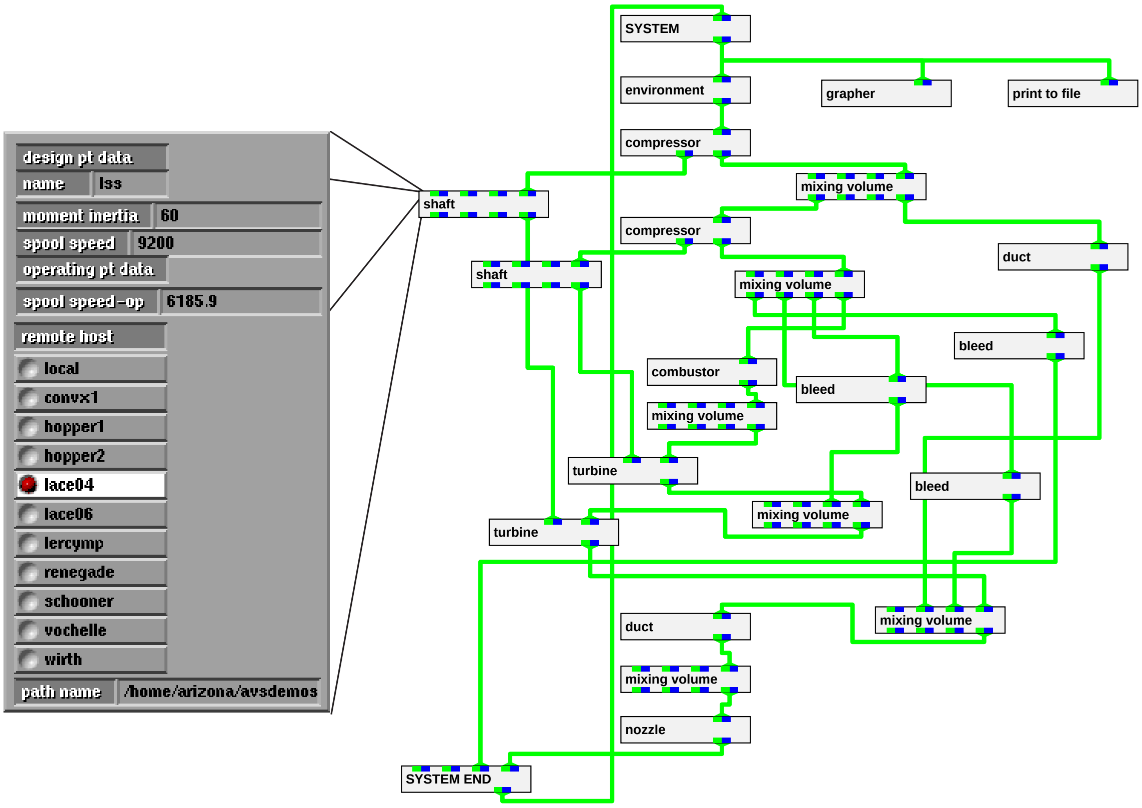

Figure 2 — Prototype simulation executive

widget that allows the user to select the file name. In TESS, this method is used for the compressor and turbine modules to select performance maps. For three of the engine components—compressor, combustor, and nozzle—transient control schedules are provided to allow the setting of the stator angles during the transient. These provide widgets that allow the user the option of varying the stator angle by specifying angles at certain times during the transient with TESS interpolating the angle at other times.

The system module provides widgets for selecting the solution methods for both the steady-state and transient thermodynamic simulations supported by TESS, and provides overall control of the simulation run. For steady state solutions, the user can choose from Newton-Raphson and Fourth-order Runge-Kutta. For transient solutions, the user can choose from Modified Euler, Fourth-order Runge-Kutta, Adams, and Gear. When execution is started, TESS first attempts to balance the engine at the initial operating point through a steady-state calculation. The engine transient begins once the engine is balanced and proceeds up to the number of seconds specified by the user.

3.3. TESS and Schooner

Four of the engine modules have been modified so that their computations are executed remotely using Schooner: the shaft, duct, combustor, and nozzle modules. The changes made in adapting the shaft module are described below; the changes needed in the other three cases were similar.

In adapting an existing code to use Schooner, four tasks must be performed. The first is to decide which procedure or procedures should be executed remotely. This requires evaluating available architectures and programming languages to decide which would be most appropriate for the given computation. In some cases, the decision may also take into account factors such as the most convenient place to locate data files containing, for example, the computational grid. Once a decision has been reached, the procedure, along with any supporting code, is separated and moved to the remote machine. In the case of the shaft code, for example, the AVS module makes calls to two procedures: `setshaft` and `shaft`. The `setshaft` procedure is called once at the start of a steady-state computation, while the `shaft` procedure is called repeatedly during both steady-state and transient computations. Both of these were selected to be a part of the remote computation. The files `npss-setshaft.f` and `npss-shaft.f` contain the `setshaft` and `shaft` procedures, respectively. Both files were moved to the destination machine.

In the next step, two UTS specification files are written, one associated with the remote procedure, and one with the AVS module. The two are nearly identical, differing only as to whether they designate the procedures as imports or exports.[1] For `shaft`, the relevant export specification is as follows:

```
export setshaft prog(
   "ecom"   val array[4] of float,  "incom"  val integer,
   "etur"   val array[4] of float,  "intur"  val integer,
   "ecorr"  res float)

export shaft prog(
   "ecom"   val array[4] of float,  "incom"  val integer,
   "etur"   val array[4] of float,  "intur"  val integer,
   "ecorr"  val float,              "xspool" val float,
   "xmyi"   val float,              "dxspl"  res float)
```

Note that in this example, all parameters are specified as either value or result parameters; UTS supports `var` (value/result) parameters as well. This export specification is co-located with the `npss-setshaft.f` and `npss-shaft.f` files on the remote machine, while the matching import specification is co-located with the code for the invoking AVS module.

The final step involves modifying the AVS module slightly to add whatever interaction facilities are desired and to coordinate startup with the Schooner Manager. In general, three short pieces of code are needed to accomplish these tasks. The first is placed in the `spec` function of

[1]UTS actually allows the import to be, in essence, a subset of the export, but this facility is not currently exploited in NPSS.

the module, which specifies the AVS input and output data streams, and the widgets the module will use. For the shaft module, two widgets were added to allow the user to specify the machine on which to execute the remote procedure and its pathname, as follows:

```
iparm35=AVSADD_PARAMETER('Remote Machine', 'choice', 'lace04',
                         'convx1:hopper1:hopper2:lace04:lace06:
                         lercymp:renegade:schooner:vochelle:wirth', ':')
call AVSCONNECT_WIDGET(iparm35, 'radio_buttons')

iparm36 = AVSADD_PARAMETER('path name', 'string',
                           '/usr/patrick/avsdemos/tess/remote/', ' ', ' ')
call AVSCONNECT_WIDGET(iparm36, 'typein')
```

The first two statements create the radio buttons that allow the selection of the remote machine. The strings between colons represent machines at Lewis Research Center and The University of Arizona that can be chosen interactively as the location to run the computation. The second two statements create a type-in widget that allows the user to specify the pathname to the executable that will correspond to the machine selected.

The second section of code is added at the beginning of the `compute` function in each AVS module that is invoking a remote computation. This function is a standard routine that is executed each time the module is scheduled for execution by AVS. The added code invokes a Schooner library function that registers the AVS module with the Schooner Manager and asks the Manager to start the remote process. This is done as follows:

```
character*(*) machine
character*(*) path
save          first_time_shaft
logical       first_time_shaft/.true./

if (first_time_shaft) then
   CALL sch_start_component(machine, 'shaft', path)
   first_time_shaft = .false.
endif
```

The value of `machine` and `path`, which are passed as arguments to `sch_contact_schx`, are set when the user makes the selection of a remote machine and types in the pathname using the two widgets described above.

The final piece of additional code is placed in the AVS `destroy` function, which is invoked when the module is removed from a network or the entire network is cleared. This code is simply a call to the Schooner library function `sch_i_quit`, which notifies the Manager that the AVS module is being destroyed. When this occurs, the Manager sends shutdown messages to the remote procedures, instructing them to terminate.

As already mentioned, the steps followed for the other three adapted modules were essentially the same. As with the shaft module, two remote procedures were involved: one that was called once to initialize values, and one that was called repeatedly during steady-state and transient calculations. It would be possible with Schooner to separate the two remote procedures, executing each on a different remote machine, or executing one on the (local) AVS machine and one on a remote machine. For TESS, however, the setup procedure for each of the selected modules is called only once per simulation, so there is no real performance gain from separating the two procedures.

3.4. Experiments

Each of the adapted AVS modules were tested separately on a variety of machine combinations over both local and wide-area networks. These tests took place primarily at NASA Lewis Research Center, with wide-area tests involving machines at The University of Arizona. Some of the more interesting combinations are summarized in Table 1. Since TESS provides a complete engine model, each adapted module could be tested to ensure that the steady-state and transient calculations converged correctly.

Additional tests were performed combining two, three, and all four of the adapted modules. An example of a four-module test is shown in Table 2. With repeated instances of two of the adapted modules present in the simulation, there were a total of six modules with remote computations. TESS was run through a steady-state computation using the Newton-Raphson method to balance the engine and a one second transient simulation using the Improved Euler method. To verify that the adapted modules were working correctly, the results were compared with the same computation using the original local-compute-only versions of the four modules.

4. The Evolution of Schooner

A primary reason for constructing and testing the prototype executive was to determine how well Schooner's capabilities matched the needs of scientific applications like NPSS. Although Schooner fared well in general, deficiencies were found that led to the evolution of the system. In this section, we first document the incremental changes that were made to Schooner as minor deficiencies were inevitably uncovered during our experimentation with an early version of the

AVS Machine	Remote Machine	Connecting Network
Sun Sparc 10	SGI 4D/480	local Ethernet
Sun Sparc 10	Convex C220	same building, multiple gateways
SGI 4D/480	Cray YMP	same building, multiple gateways
SGI 4D/480 Lewis Research Center	Sun Sparc 10 The University of Arizona	via Internet
Sun Sparc 10 The University of Arizona	IBM RS6000 Lewis Research Center	via Internet

Table 1—TESS and Schooner individual module tests

AVS Machine	Module	# of Instances	Remote Machine	
TESS Simulation executed on Sun Sparc 10 at U. of Arizona	combustor	1	SGI 4D/340	U. of Arizona
	duct	2	Cray YMP	Lewis Research Center
	nozzle	1	SGI 4D/420	Lewis Research Center
	shaft	2	IBM RS6000	Lewis Research Center

Table 2—TESS and Schooner combined test

NPSS prototype. These modifications were designed to overcome specific and relatively small problems, or to extend the functionality of the system. We then turn to describing the one more significant extension that we found desirable based on our experience with NPSS. This extension involved modifying Schooner's programming model to support more dynamic configuration of remote procedures into the overall computation. The incremental changes were made as work with the early version of the prototype executive progressed. The modifications needed to realize the programming model extension were recently completed and used in the TESS experiments described in the previous section.

4.1. Incremental Changes

Incremental changes to Schooner included the addition of the Cray YMP and IBM RS6000 to the list of supported machines, the addition of a new floating point type to UTS, and a change in the Schooner startup protocol. We describe here the work involved with incorporating the Cray YMP; the work for the IBM RS6000 was similar. Adding the Cray was straightforward and involved work in two areas. The first was writing UTS conversion routines for the Cray data types, especially the ones for integer and floating point values, which are used heavily in NPSS. Such routines are easily written since they simply involve converting the Cray's internal data representations to and from the UTS intermediate representation. The only problem was that the Cray's integer and float representations support larger magnitudes than the IEEE standard used by UTS. Two remedies were considered: treating such out-of-range Cray values as an error, or converting them to the IEEE ‘‘infinity’’ value. After consultation with researchers involved in developing NPSS code, the first option was chosen.

The other area where work was needed to incorporate the Cray was modifying the Schooner runtime system to support communications with the machine. In general, this was no more difficult than for other machines, requiring only a few changes to include files and type declarations. The one area where an unexpected problem did arise was in the naming of Fortran procedures. On most machines, procedure names are converted to lower case by their respective Fortran compilers, while the compiler on the Cray uses upper case. This inconsistency caused a surprising number of naming problems, both for the writer of Schooner programs and for the Schooner implementation itself. For example, if this inconsistency had been retained, a user writing a program that calls a remote Fortran procedure would need to know beforehand whether it would be run on a Cray or some other machine. Moreover, having Schooner standardize all procedure names to, for example, lower case is not satisfactory because that would interfere with common naming conventions in other languages such as C. In the end, the choice was made to accept both upper and lower case names for Fortran procedures, and then treat them as synonyms within Schooner. This was done primarily by changing the Manager so that it stored both the upper and lower case alternatives in its mapping tables.

The second incremental change was expanding the UTS type system to include both single- and double-precision floating point types instead of just double-precision. The original decision to include only double-precision was in keeping with the Kernighan and Ritchie C specification [Kernighan88], which requires that values of both float and double types passed as arguments be coerced to double for the call. With the addition of Fortran to Schooner and the development of the ANSI C specification, this practice is no longer adequate. Additionally, having both types is an advantage since it allows the user to specify more precisely the size of the argument value to be passed.

To support both sizes of float values, two changes were required. The first was to add both *float* and *double* to the UTS specification language, with the corresponding changes to the parsers for the stub compilers. The second change involved adding the the appropriate encode and decode functions to the UTS library for each of the supported architectures.

The third area where changes to Schooner were needed involved the startup protocol. Previously, Schooner programs were started by executing the Manager as a command and specifying the various files containing Schooner procedures and the appropriate machines as its arguments. Once started, the Manager would create processes to execute all the remote procedures on the appropriate machines, and then invoke the program's main routine. Since the Manager controlled everything, it could easily collect the mapping information needed to resolve subsequent remote invocations. When AVS is involved, however, the Manager is no longer in control. Specifically, it now has no way of inferring when or where a process should be instantiated for a remote procedure, since this now occurs when a module is configured into an AVS network rather than when the Manager is started. To solve this problem, a new protocol was devised that allows a newly-configured module to establish initial contact the Manager and to send requests for a remote procedure to be started on a specific machine.

4.2. Extended Schooner Model

As mentioned above, the original Schooner model of how a program would be executed was oriented around the traditional command line paradigm where everything is specified *a priori*. Unfortunately, this model has proved too restrictive for NPSS, where the pieces of the overall application are configured dynamically at runtime using the Network Editor. In fact, this would be the case for any application written using AVS or similar systems.

One easy change that extended this model to a degree was the new startup protocol described above. This allows remote procedures to be initiated only when needed, thereby facilitating such features as interactive user placement of a remote computation using a widget. However, this change did not solve all the problems, largely because certain simplifying assumptions had been made in the implementation; while these assumptions were valid given the original orientation, they became less so as the orientation changed.

As a prime example, an original assumption in Schooner was that only one procedure of a given name would be present in a program. This assumption was implicit in the procedure call paradigm. A traditional sequential program only has one instance of each named procedure and adding the ability to call remote procedures did not change this assumption. While reasonable, such an assumption is too restrictive for NPSS. In this environment, there may well be multiple instances of the same module in the network, and therefore, multiple remote procedures with the same name. This scenario, in fact, appears in the TESS F100 simulation (see Figure 2) where the network contains multiple instances each of the bleed, compressor, duct, mixing volume, shaft, and turbine modules.

A number of additional problems were discovered, most of which were also related to the sequential execution model. A (non-exhaustive) list of these included:

- The Schooner runtime was written assuming no concurrent execution of remote procedures; with the dataflow model of AVS, this may not be true.
- The original Schooner shutdown procedure terminated the entire program when any part executes `sch_i_quit` or an error occurs; deleting an individual module in AVS should, in fact, result only in the termination of those remote computations associated with the module.
- Remote procedures cannot be moved once instantiated as processes; given the potentially long-running nature of codes in NPSS, moving the computation should be an option so that, for example, scheduled downtimes can be avoided.

All these problems forced a rethinking of the Schooner model and corresponding changes to the implementation. Our goals in doing so were first, to retain an intuitive, easy to use, and general programming model for the user, and second, to minimize implementation effort.

One option for an extended model that was considered and quickly discarded was to treat each module in an AVS network and its associated remote computations as a separate Schooner program. In this model, each module would have its own Manager process to handle remote startup, name mapping, etc. While workable, this strategy would put an undue burden on the user, who becomes responsible for managing which AVS module is associated with which Manager. It would also preclude sharing remote procedures between modules, something that is desirable in certain situations.

The option that was, in the end, chosen involves extending the model of a Schooner program to include multiple threads of control, which we call *lines.*[2] Each line is equivalent to the previous notion of a Schooner program; that is, it is a sequential execution of procedures, some of which may be located on remote machines and/or written in various programming languages. Any procedure in a line can request the initiation of other remote procedures by using the appropriate Schooner library function; such newly started procedures are considered part of the requesting procedure's line and are callable only from other procedures in that line. As before, there is a single Manager, but it now handles the initiation and name mapping chores for multiple control threads. Given the more dynamic nature of the computation, the Manager also becomes a persistent process that must be explicitly initiated and terminated by the user.

The extension of the model to include multiple lines solves many of the problems associated with the original model without unduly complicating either the user's task or the implementation. For example, concurrency is possible, but controlled; each line is sequential and executes independently of the others with no synchronization. Similarly, no duplicate procedure names are permitted within a line, but multiple lines can contain remote procedures with the same name; in this case, each line will have its own instance. Reasonable shutdown semantics are also supported easily in this extended model. The shutdown protocol now involves only the procedures in a single line. Thus, when an AVS module is removed from the network or an error occurs, the Manager terminates only the remote procedures within the affected line.

Implementing lines within Schooner primarily involved changes to the Manager. Where the Manager had previously maintained a single procedure name database, it now maintains a separate database for each line. When a mapping request is received by the Manager, only the line from which the request is originated is searched. In a similar fashion, when a procedure stops, either through an error or a call to `sch_i_quit`, the Manager sends shutdown messages only to those procedures in the same line. The persistent nature of the Manager process in this new model also allows multiple runs of a simulation to be handled, including re-loading the same or a different engine model into AVS.

Another feature supported by the new model is the ability to move a remote procedure from one machine to another during execution. This is useful when a machine is approaching a scheduled down time, or when the load on the current machine grows too large and a more lightly loaded machine is available. To carry out a move, a Schooner library function is invoked in which the procedure to be moved and the target machine are specified as arguments. This results in the Manager first sending a shutdown message to the original procedure, and then starting a new copy on the specified machine. The Manager then updates the procedure name mapping information for the line, so that future calls go to the new location. Procedure name caches within each procedure in the line are updated when the next call to the procedure is attempted. The call to the old location fails, resulting in an automatic call to the Manager for the new information. It should be noted that this kind of procedure migration is currently feasible only if the procedure is stateless. This condition is satisfied by many of the codes in the TESS prototype and by many other scientific codes. A planned addition to Schooner will utilize an extension to

[2]Line is the nautical term for rope.

the UTS interface description language to describe a list of state variables whose values are to be transferred when the procedure is moved.

Shared procedures are also planned for Schooner as a result of the addition of lines. A procedure will be designated as shared at startup time, indicating that it is not part of the line from which the startup request originated, but is available for use by any line. The Manager will maintain a separate database for shared procedures. Mapping requests to the Manager will be checked first against procedures in the line from which the request is received, and then against a list of shared procedures. When a shared procedure is terminated or moved, the mapping database is updated for all lines.

This addition of lines to the Schooner model represents, we believe, a good compromise between the overly restrictive model used previously and the complexity that results from a complete generalization. With this scheme, the user can manipulate an NPSS network from within AVS in such a way that the remote computations behave reasonably, while still retaining simplicity of use.

5. Conclusions

Developing a suitable simulation executive is undoubtedly one of the most important and challenging aspects of the NPSS project due to the myriad of responsibilities assigned to the software. Not only must it provide features that can be used to control and interact with the computation, but it must also mask from the user the effects of heterogeneity and distribution. Here, we have described a prototype executive that takes a step towards meeting these needs by combining the strengths of AVS and Schooner. Specifically, AVS provides the ability to compose and control a simulation using a high-level network editor and associated widgets, while Schooner provides transparent access to heterogeneous and distributed resources. In evaluating the capabilities of Schooner in this regard, we concluded that, although the system was a reasonable match when the project started, some modifications were needed to provide stronger support for the requirements of NPSS and similar scientific projects.

Acknowledgements

The NPSS project is managed by the Interdisciplinary Technology Office (ITO) at NASA Lewis Research Center (LeRC). Thanks to A. Afjeh and J. Reed of the U. of Toledo and C. Putt, B. Perry, and B. Armstead of LeRC, who provided assistance with understanding the software and hardware requirements of NPSS. Special thanks also to G. Follen for his support and advice. This work was performed in part on computing resources at the Advanced Computing Concepts Laboratory and the Computer Services Division at LeRC.

References

[Almes85] 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.

[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.

[Bershad87] Bershad, B.N., Ching, D.T., Lazowska, E.D., Sanislo, J., and Schwartz, M. A remote procedure call facility for interconnecting heterogeneous computer systems. *IEEE Trans. on Softw. Eng. SE-13*, 8 (Aug. 1987), 880-894.

[Birrell84] Birrell, A. D. and Nelson, B. J. Implementing remote procedure calls. *ACM Trans. on Computer Systems 2*, 1 (Feb. 1984), 39-59.

[Black87] 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.

[Butler92] Butler, R. and Lusk, E. User's guide to the p4 parallel programming system, Argonne National Laboratory, Argonne, IL, August 1992.

[Chen93] Chen, S., Eshaghian, M., Khokhar, A., and Shaaban, M. A selection theory and methodology for heterogeneous supercomputing. *Proc. Workshop on Heterogeneous Processing*, Newport Beach, CA (Apr. 1993), 15-22.

[Claus92] Claus, R.W., Evans, A.L., and Follen, G.J. Multidisciplinary propulsion simulation using NPSS. *4th AIAA/USAF/NASA/OAI Symposium on Multi-disciplinary Analysis and Optimization*, Cleveland, Ohio, September 21-23, 1992.

[Claus91] 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.

[Freund93] Freund, R. F., and Siegel, H. J. Heterogeneous processing. *Computer 26*, 6 (June 1993), 13-17.

[Gibbons87] 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.

[Griswold90] Griswold, R. and Griswold, M. *The Icon Programming Language*, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Hayes87] 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.

[Hayes88] 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.

[Hayes89] Hayes, R. *UTS: A Type System for Facilitating Data Communication*, Ph.D. Dissertation, Dept. of Computer Science, Univ. of Arizona, August 1989.

[Hayes90] 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.

[Homer92a] Homer, P.T., and Schlichting, R.D. Adapting AVS to support scientific applications as heterogeneous, distributed programs (extended abstract). *Proc. Workshop on Heterogeneous Processing*, Beverly Hills, CA (Mar. 1992), 50-53.

[Homer92b] Homer, P.T., and Schlichting, R.D. A software platform for constructing scientific applications from heterogeneous resources. Tech. Report 92-30, Dept. of Computer Science, Univ. of Arizona, Nov. 1992.

[Jones85] 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, LA (Jan. 1985), 225-235.

[Kernighan88] Kernighan, B.W., and Ritchie, D.M. *The C Programming Language*, second edition, Prentice Hall, Englewood Cliffs, NJ, 1988.

[Khokhar93] Khokhar, A. A., Prasanna, V. K., Shaaban, M. E., and Wang, C. Heterogeneous computing: Challenges and opportunities. *Computer 26*, 6 (Jun. 1993), 18-27.

[Mercurio92] Mercurio, P.J. Khoros. *Pixel 3*, 2 (Mar./Apr. 1992), 28-33.

[Quealy92] Quealy, A., Cole, J., and Blech, R. Portable programming on parallel/networked computers using the Application Portable Parallel Library (APPL), NASA Technical Manual, 1992.

[Rasure91] Rasure, J. and Williams, C. An integrated visual language and software development environment. *Jour. of Visual Languages and Computing 2* (1991), 217-246.

[Reed93] Reed, John A. Development of an interactive graphical aircraft propulsion system simulator. Master of Science Thesis, University of Toledo, August 1993.

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

[Sunderam90] Sunderam, V. S. PVM: A framework for parallel distributed computing. *Concurrency—Practice and Experience 2* (Dec. 1990), 315-339.

[Wang92] Wang, M., Kim, S., Nichols, M., Freund, R., Seigel, H., and Nation, W. Augmenting the optimal selection theory for superconcurrency. *Proc. Workshop on Heterogeneous Processing*, Beverly Hills, CA (Mar. 1992), 13-22.

[Xerox81] Xerox Corp. *Courier: The Remote Procedure Call Protocol*. Xerox System Integration Standard XSIS 038112, Xerox Corp., Stamford, Conn., Dec. 1981.