CONSTRUCTING SCIENTIFIC APPLICATIONS FROM HETEROGENEOUS RESOURCES

(Ph.D. Dissertation)

Patrick T. Homer

TR 94-33

December 1, 1994

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 and by the National Aeronautics and Space Administration under GSRP grant NGT-50966.

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirement for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED:_____

ACKNOWLEDGMENTS

I am very grateful to my advisor, Rick Schlichting, for his patience with my sometimes slow acquisition of research skills and for his questions and ideas that kept me constantly striving to keep up with him in our many discussions. His willingness to discuss all aspects of research — writing, locating grants, asking the right questions — has helped prepare me for life after graduate school. I may never find the right abstraction, but he has convinced me it is out there somewhere.

I thank my other major advisors, Mary Bailey and Larry Peterson, for their advice, comments, and helpful questions about my research, and for their careful reading of my dissertation.

The research in this grant is dependent on collaboration with many scientists and engineers. I am especially indebted to Abdollah Afjee and John Reed of the University of Toledo, Henry Lewandowski of Cleveland State University, and Greg Follen and Dave Clark of NASA Lewis Research Center for their patient explanations of their own work and their willingness to try the ideas and software described in this dissertation. Many others have assisted by providing applications or useful advice, including Charles Hansen of Los Alamos National Labs, Steve Lustig of Dupont Central Research, and Christy Bergman of Stanford University.

A lot of folks made my stay in Tucson enjoyable, the Computer Science Department a great place to work, and the mountains wonderful places to hike and camp. I am especially grateful to Nina Bhatti, John Cropper, Curtis Dyreson, Cliff Hathaway, Tyson Henry, Clint Jeffery, Phil Kaslo, Nick Kline, Jim Knight, Ed Menze, Shamim Mohamed, Gary Newell, Bob Simms, Mike Soo, Wendy Swartz, Vic Thomas, Dean Vinson, Susie Wagner, Beth Weiss, Ken Walker, and Andrey Yeatts.

I thank my parents and my sisters whose faith, prayers, and support saw me through the course work, tests, and research. My Father had trouble understanding why I kept going back to school, but he never lost faith that I would finish.

This research was supported in part by the National Science Foundation under grant ASC-9204021 and by the National Aeronautics and Space Administration under grant NGT-50966.

DEDICATION

To Ann Danowitz, for your faith in me especially during the “dry spells” in the research. Your spirit lighting the way made the final steep climbs possible.

To my Mother, who always knew I would finish if I “just put my mind to it.” As always, she was right.

TABLE OF CONTENTS

TABLE OF CONTENTS — *continued*

TABLE OF CONTENTS — *continued*

LIST OF FIGURES

LIST OF FIGURES — *continued*

LIST OF TABLES

ABSTRACT

The computer simulation of scientific processes is playing an increasingly important role in scientific research. For example, the development of adequate flight simulation environments, numeric wind tunnels, and numeric propulsion systems is reducing the danger and expense involved in prototyping new aircraft and engine designs. One serious problem that hinders the full realization of the potential of scientific simulation is the lack of tools and techniques for dealing with the heterogeneity inherent in today's computational resources and applications. Typically, either ad hoc connection techniques, such as manual file transfer between machines, or approximation techniques, such as boundary value equations, are employed.

This dissertation develops a programming model in which scientific applications are designed as heterogeneous distributed programs, or *meta-computations*. The central feature of the model is an *interconnection system* that handles the transfer of control and data among the heterogeneous components of the meta-computation, and provides configuration tools to assist the user in starting and controlling the distributed computation. Key benefits of this programming model include the ability to simulate the interactions among the physical processes being modeled through the free exchange of data between computational components. Another benefit is the possibility of improved user interaction with the meta-computation, allowing the monitoring of intermediate results during long simulations and the ability to steer the simulation, either directly by the user or through the incorporation of an expert system into the meta-computation.

This dissertation describes a specific realization of this model in the Schooner interconnection system, and its use in the construction of a number of scientific meta-computations. Schooner uses a type specification language and an application-level remote procedure call mechanism to ease the task of the scientific programmer in building meta-computations. It also provides static and dynamic configuration management features that support the creation of meta-computations from components at runtime, and their modification during execution. Meta-computations constructed using Schooner include examples involving molecular dynamics and neural nets. Schooner is also in use in several major projects as part of a NASA effort to develop improved jet engine simulations.

Chapter 1
INTRODUCTION

The computer simulation of scientific processes is playing an increasingly important role in scientific research because of its potential for performing controlled experiments where cost or danger limit real-world experiments. For example, the development of adequate flight simulation environments and numeric wind tunnels can reduce the danger and expense involved in prototyping new aircraft designs. Climate modelling provides another example where the possibility of predicting long-term climatic effects can help in determining appropriate measures to avoid future problems.

One serious problem that hinders the full realization of the potential of scientific computing is the lack of tools and techniques for dealing with the *heterogeneity* inherent in today's computational resources and applications. Computational heterogeneity occurs in several dimensions, including programming languages, programming models, and machine architectures. For example, a complex simulation might require a parallel algorithm that maps to a MIMD parallel machine like the Intel Paragon, another algorithm that requires a vector machine such as a Cray C-90, and a third that needs a high-end graphics workstation such as an SGI Crimson. Unfortunately, due to a lack of adequate system support, a researcher is often forced to structure such an application so that each algorithm is encapsulated as a separate program, with manual file transfer used to move data between the programs. Clearly, such an approach is awkward at best.

This dissertation proposes that scientific applications requiring heterogeneous resources be constructed as *heterogeneous distributed programs,* or *metacomputations* [Khokhar93], composed of multiple software modules executing on different types of machines. In addition to presenting this model, a software infrastructure called Schooner that supports the construction of programs according to this model is described. The system also gives the user complete control over the configuration and execution of the resulting application, thereby making it easy to bring a variety of computational resources together to solve scientific problems.

1.1 Current Model

The current state-of-the-art in scientific computing presents a dilemma to software developers. On one hand, the complexity of a real-world system requires a corresponding complexity in any attempt to model it computationally. For a given application, this complexity often mandates not only a substantial number of cycles, but the use of different programming models and hardware and software resources to implement the various parts of the solution. Unfortunately, the other side of the dilemma is that little software support is available for incorporating such heterogeneous resources into a single logical program. The resulting model is one in which an application consists of multiple individual programs or *components*, where each is executed separately and

files are used to transfer data from one component to the next.

In examining the current model more closely, two specific problems can be identified. The first is the *interconnection problem*, i.e., how to transfer control and communicate data among the heterogeneous components that comprise the application. Without adequate software support for communicating data among components, the developer is forced into one of two inadequate methods: data files or approximation. The data file approach has each component write its results to one or more data files, with these becoming the inputs to other components. Beyond the burden this places on the user to manage the files and move them among machines and file systems, this method does not provide the frequency of data exchange needed in many situations, and implies an essentially disconnected execution of the components. The second method avoids data exchange altogether and uses approximation instead. For example, in computational fluid dynamics (CFD), a common practice is to represent the upstream fluid data with a boundary that approximates the data with either a constant set of values, or time-varying functions to approximate patterns present in the real system. Yet, neither of these boundary approaches can provide the realistic data needed in many simulations, or provide the bi-directional data that is critical in understanding how the physical processes on either side of the boundary interact.

Solving the connection problem results in a heterogeneous distributed program, but leaves the user with the second problem, which we call the *configuration problem*. Where the user had been executing each component separately, the situation now requires all components to execute at the same time. As a result, the user is presented with the difficulties of manually starting components on a varied collection of machines, establishing the communication links needed among the components, and controlling the thread of execution as it passes from component to component. Currently, little support is available in the domain of scientific computing for configuring such a collection of components into a single application, or for controlling the subsequent execution flow.

Thus, the current model is based on the premise that connecting heterogeneous applications is a hard problem. This results in the use of techniques that fail to accurately simulate real-world conditions. It is time to re-think this approach. Heterogeneity can and should be an advantage in designing a meta-computation, not a barrier. A new model of scientific programming is needed that embraces heterogeneity, and provides the necessary software infrastructure to assist the developer and user in creating and controlling meta-computations.

1.2 Example Applications

To illustrate the potential benefits of a new paradigm for writing scientific programs that supports access to heterogeneous resources, two canonical scientific applications are presented. In both cases, the complex systems being modelled require a variety of algorithms, with different hardware and software requirements, to simulate the parts of the real-world system. Development efforts have previously focused on improving the performance of the component applications.

Now, efforts are being directed toward integrating the components into a single logical application to provide a more accurate simulation of the real-world system.

1.2.1 Climate Modelling

Recent developments in numeric global climate models are promising for the first time controlled climate experiments that can support detailed research into the effect of factors such as industrial pollutants or volcanic ash. Ideally, a program implementing such a model should exhibit two characteristics. First, it should complete quickly enough and with fine enough resolution that decades-long simulations can be performed in a reasonable amount of time. Second, it should include the ability to monitor and steer the simulation through, for example, a graphical interface. Both of these characteristics are approached but not fully realized in existing systems.

A specific example of this class of application is the global climate modelling project [Mechoso92, Mechoso91] being done as part of the CASA gigabit network testbed [Bergman91]. This project has two goals. The first is to improve two global models, one modelling atmospheric circulation and the other oceanic circulation. The desired improvements fall into two categories common in scientific computing: improving the resolution of the simulation through the use of a finer grid, and improving the accuracy of the simulation through the inclusion of more of the physical factors that affect the climate. For example, the ocean circulation model is extending its southern edge to include the ice flows that surround the Antarctic continent. This requires the addition of equations governing the influence of the ice on the ocean currents.

The second goal is to integrate the two models into a single meta-computation. This will provide a more accurate simulation as both models are influenced by forces that occur at the ocean/atmosphere boundary. Here, the surface temperature of the water affects the equations governing the air currents, and the wind stress and heat fluxes of the atmosphere affect the ocean currents. When independently executed, as has been the practice until now, each model uses monthly averages to approximate conditions along this boundary, limiting the accuracy of the simulation as the variations that occur during a month are missing.

The integration of the two models must overcome a number of heterogeneity challenges. Some are imposed by the lack of an adequate programming model for interconnecting scientific applications, while others are a result of the models themselves. These challenges include:

- The heterogeneous programming models and machine requirements of the two applications: the ocean model is implemented as a SIMD code on a CM-2; the atmospheric model is implemented as a vector code on a Cray YMP.

- Coordinating the two models given their different execution times. In general, the ocean model completes one time step several times faster than the atmospheric model; however, the specific machines available for an experiment and their load can have a large impact on the relative speeds.

- The resolutions of the grids in the two models is different, with the ocean model using a finer grid. This requires the use of averaging functions to match values along the common boundary.

The meta-computation also needs support for configuration and control; it is not enough to merely execute the two models. One reason is the times at which the two models exchange data. These can be selected to allow overlap of computation and communication by taking advantage of two sub-tasks within the atmospheric model: physics and dynamics. The physics task computes the effect of subgrid-scale processes on large-scale motions, and supplies data to the dynamics task as forcing terms in the hydrodynamic equations. It turns out that the boundary values required by the ocean model are computed by the physics task; thus, the ocean model and the dynamics part of the atmospheric model can be run concurrently.

A second reason for configuration and control support are plans to include a graphical workstation to allow on-line monitoring of the climate model results. Eventually, the graphical interface will be extended beyond monitoring to give the user greater control over the application, including the ability to halt or extend the simulation, and re-play portions of a simulation with varied parameters.

1.2.2 Jet Engine Design

Jet engine simulation involves modelling the flow of air through the various components—duct, fan, compressor, combustor, turbine—that make up the engine. Numeric codes have been developed for each of these components at different levels of fidelity, from one-dimensional to full three-dimensional, time-varying simulations. The computational requirements of simulating a component at a high degree of fidelity has precluded simulating the interaction between components. The pressure, temperature, and velocity of the air flow at the boundaries between components are typically approximated as steady-state values, or as time-varying functions. Unfortunately, this can result in unexpected interactions between components that may not be discovered until a prototype engine is constructed.

The NASA Numerical Propulsion System Simulation (NPSS) project is an effort to improve the jet engine design process through the use of advanced computer hardware and software [Claus91, Claus92]. NPSS is moving in two directions. One is the application of parallel computing technology to the design of improved numeric models for the various components of a jet engine, seeking both shorter execution time and improved model accuracy. The second direction is the construction of a simulation executive, an example of the kind of system support for connecting and controlling applications referred to above. This executive is essentially a numerical test cell for modelling an entire jet engine with a major goal being the modelling of the interactions among the engine components. This will be accomplished by executing each component and allowing the components to exchange boundary values during the simulation. To address the

issue of long compute times, the executive will support *zooming*, a technique that will allow engine components at different levels of fidelity to be combined in the same simulation.

The ultimate goal of NPSS is an environment and a set of engine component codes that allows the engine designer to "build" a simulated engine from available components, interconnecting them to create an engine model. Configuration and control support will allow specific characteristics of the engine to be determined at runtime through specifying the exact parameters for each component, and then testing the engine with a set of overall operating conditions. The user will be able to zoom in on a component of interest during the simulation by re-configuring the engine to include a high-fidelity simulation of the component. A necessary piece will be the inclusion of visualization tools that allow the user to watch the simulation in progress, and the ability to modify parameters both at the start and during a test run. The simulation environment will need a seamless integration of a variety of machines, both to handle the variety of programming models used in developing the component codes, and to provide the necessary computational resources to model all the engine components.

1.3 A New Model for Scientific Applications

This dissertation proposes and develops a new model for scientific applications that allows the interconnection of heterogeneous components, and facilitates control over the configuration and execution of the application.

Specifically, a model for connecting scientific applications is presented that goes beyond simple accommodation of heterogeneity to exploitation of heterogeneity in the solution of scientific problems. Components can be selected based on their value to the computation, regardless of their programming model, programming language, or machine architecture requirements. Such a scientific application is a heterogeneous distributed program, or *meta-computation*, consisting of a distributed collection of component codes, executing on a variety of machines, spanning both short- and long-haul networks. No longer is the user forced to approximate data when more accurate data can be supplied by another code. Files are not needed as a data exchange mechanism since the data can now be sent directly to its destination, and can be sent as many times as needed. The user creates the application by selecting the components needed; the support system transparently handles the task of connecting the components together.

The central feature needed to implement this model is a software *interconnection system* that provides connections among components and implements a configuration and control system. Each component contains one computation code or data manipulation tool that accomplishes a specific set of tasks. A component can be developed using the combination of programming language, model, or architecture that is most suitable; thus, the component becomes the unit of heterogeneity in the system. At runtime, components export services for use by other components in the application. This is accomplished by attaching an automatically generated interface to each component. The interface advertises those services the component makes available to other

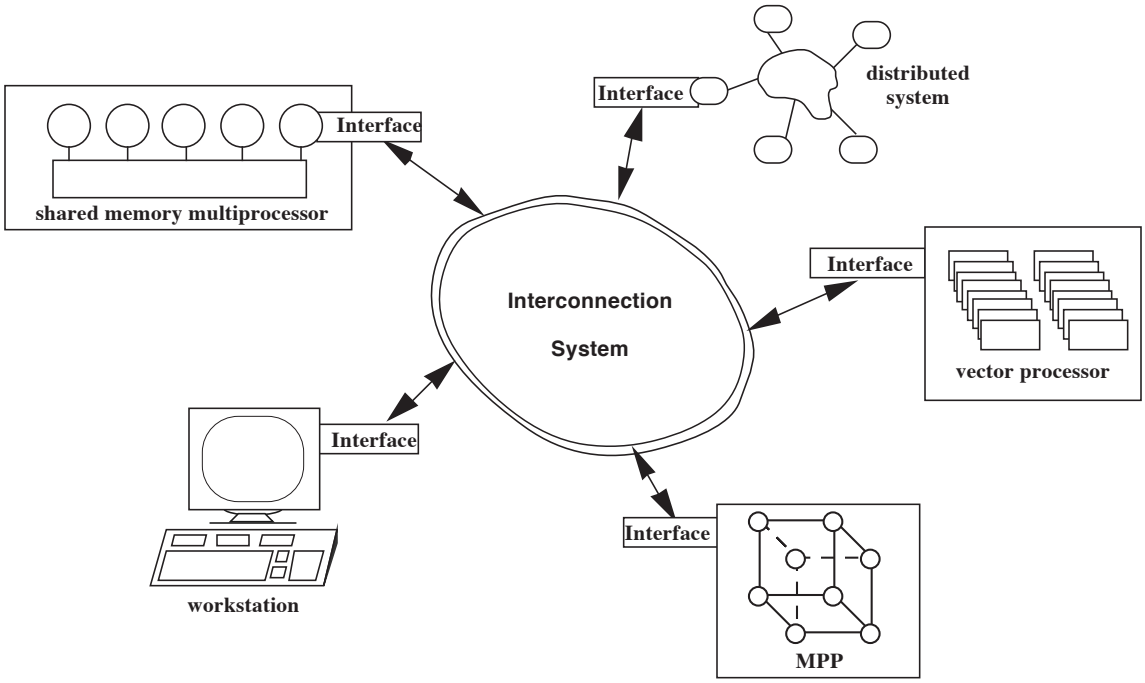


Figure 1-1: Interconnection System Model

components, and handles the job of locating and accessing the external services needed by the component. Figure 1-1 illustrates this concept.

An interconnection system results in a meta-computation distributed over a variety of machines, requiring configuration tools to assist the user in starting and controlling the component applications. The configuration management features of the model give the user both static and dynamic configuration control. Static control allows the user to select the components that will be needed for the execution, and to start and execute the meta-computation. Once execution has begun, dynamic control allows components to be added or removed as needed by the user or through commands issued by the components themselves.

A very useful benefit of this approach is the potential for improved user interaction with a simulation. The meta-computation can be configured to include a visualization tool that allows the user to monitor the results of the computation in real-time, for example. Through controlling the parameters for some, or all, of the component applications, the user can steer the simulation based on intermediate results.

The Schooner interconnection system realizes this model of scientific computing, providing for the creation of meta-computations, and supplying a configuration and control mechanism for executing heterogeneous distributed computations. There are four, mostly orthogonal, parts to Schooner: a specification language, an intermediate data representation and accompanying data exchange library, a set of stub compilers, and a runtime support system. The Universal Type

System (UTS) provides both the specification language and the intermediate data representation [Hayes89]. The specification language is machine- and language-independent and is used to describe the interface for each component application. The UTS intermediate data representation provides a medium for exchanging data across machine architectures and handling data structure differences among languages. The stub compilers, one for each supported language, read the UTS specifications and create the interface. The runtime system implements application-level remote procedure call (RPC) control transfer between components, as well as configuration and control features. It provides the user with a means of integrating the various components into the meta-computation, and provides the underlying communication and management support.

The Schooner system grew out of the MLP system [Hayes87, Hayes88]. MLP implemented similar RPC functionality that was capable of spanning a collection of heterogeneous machines connected by a local area network. In particular, the UTS system that formed the heart of MLP comprises a significant piece of Schooner. We have found the use of the UTS specification language in particular to be a necessary and important piece in the design of the Schooner interconnection system. The philosophy used in the design of MLP—to produce a system that meets most of the needs of the user while retaining a clean, easy to modify implementation—has been retained in Schooner.

The research in this dissertation has developed and refined the interconnection model, with a particular emphasis on support for heterogeneity, and applied it to the domain of scientific computing. Other contributions of this work include

- Development of the Schooner system to support the new paradigm, building upon the core MLP software,
- Experimentation with the model and Schooner system in a variety of scientific applications, including a prototype simulation executive for NASA's NPSS project,
- Recognition of the need for dynamic configuration tools, and their implementation,
- Support for the inclusion of graphical user interface toolkits and scientific visualization systems, and
- Extending the system to support a new language and additional architectures needed for scientific applications.

The Schooner system currently supports the following:

- Architectures: Cray and Convex vector machines, Sequent Symmetry, Intel Paragon, Digital VAX, IBM SP-2, and IBM RS6000, Sun, SGI, and Stardent workstations
- Languages: C and FORTRAN
- Visualizers/GUIs: AVS, TAE+ GUI toolkit
- Message-passing systems: PVM 2 and 3, Paragon and Sequent native libraries

1.4 Dissertation Outline

This dissertation further develops this model of scientific programming. It also describes the Schooner interconnection system with a special focus on a number of small and large scientific meta-computations that have been constructed. In the next chapter, a brief survey of other systems currently used in constructing distributed scientific applications is given, with particular attention paid to their potential usefulness in constructing meta-computations.

Chapter 3 further develops the meta-computation model and outlines the major components of the basic Schooner system. Chapter 4 then describes the use of the basic system in the context of two example applications, a molecular dynamics simulation and a neural net.

Chapter 5 describes the extended model of Schooner, including its support for concurrency and dynamic configuration. Chapter 6 explains how Schooner is being used in the context of the NPSS project, including its role in the development of a prototype simulation executive.

Finally, chapter 7 contains some concluding remarks and outlines some directions for future uses of Schooner.

Chapter 2
SYSTEMS FOR DESIGNING META-COMPUTATIONS

An interconnection system for constructing scientific meta-computations needs to provide support for heterogeneity, and control over the configuration of the meta-computation. The heterogeneity support should extend in a number of dimensions, including support for a variety of machine architectures, programming models, programming languages, and network protocols. Configuration control should allow easy creation of new components and support for binding them into the meta-computation. Finally, the system should be easy to use since computational scientists are programmers, but are not computer scientists. They need support tools that are useful without requiring them to learn new computer science techniques.

There are a number of systems that address issues of interprocess communication and heterogeneity as they apply to scientific computations and others that address configuration and control issues. Each system addresses some aspects of the heterogeneity, communication, or configuration needs of an interconnection system, but stops short of providing a complete solution. These systems can be divided into three categories based on the specific goals of each:

- Message passing systems that are designed to facilitate the implementation of parallel algorithms,
- Systems that seek to automatically recognize heterogeneity in an application, and
- Systems that provide configuration management capabilities for distributed computations.

2.1 Message Passing Systems

Systems such as PVM [Sunderam90, Beguelin91], p4 [Butler92], APPL [Quealy93], Zipcode [Skjellum93] and others, are representative of a class of systems that provides general support for constructing parallel programs using a collection of (in some cases, heterogeneous) machines.

2.1.1 PVM

PVM provides a set of library routines, callable from C or FORTRAN programs, that implements:

- asynchronous message passing between processes,
- broadcast/multicast communication,
- process creation/deletion,
- synchronization and consensus.

Communication between tasks is supported in PVM by an asynchronous send and two receive

operations. The asynchronous send delays the sender only until the message buffer on the sending machine is freed. A blocking receive delays until a message arrives while a non-blocking receive returns either a message or a flag indicating no message is available when the primitive is executed. Broadcast is supported by using a distinguished destination address in the send operation to indicate that it is sent to all processes. There are no inherent limits on the size of messages exchanged between machines. Differences in data representation between machines are handled in PVM using Sun XDR [Sun90] as an intermediate representation, with a collection of library routines provided to convert standard data types. These calls are inserted by the user prior to a send or after a receive.

A PVM program executes on a "virtual machine" that is created from a collection of (possibly heterogeneous) machines. Configuration is controlled initially through a host file that lists the machines to be used in the program. A PVM daemon process is started on each machine in the list. The program is then executed by starting the main task, which "joins" the PVM program, and spawns additional tasks as needed. As new tasks are spawned, they can be implicitly mapped to hosts by PVM in a round-robin manner or explicitly mapped by arguments given to the spawn function. PVM provides dynamic configuration control through the ability to add and delete hosts, and through the ability of tasks to join and exit the program and create new tasks. The PVM daemon processes collectively implement the configuration and control aspects of the system. They handle process creation and synchronization, and typically act as a relay point for local messages destined for processes on other machines or for incoming messages destined for local processes.

With version 3 of PVM a simple process group abstraction was added that allows a task to join or leave groups at will. The group operations include a barrier, which delays processes that invoke it until a specified number of processes within the group have reached the barrier, and a multicast, which sends a message to all members of the group.

PVM supports a variety of parallel programming models. The system is general enough to execute an MIMD computation, with each task potentially working on a separate and independent part of the problem. The two most commonly used models, however, are a master/slave configuration where the master process parcels out tasks to the slaves, and an SPMD model where the tasks are identical but operate on different parts of the data.

2.1.2 APPL

APPL supports a message-passing programming model on collections of homogeneous machines using C or FORTRAN. The goals of the project are to provide a system that is portable across a number of platforms without changes to the source code, and to provide good performance on all the platforms.

To realize these goals, APPL uses a library of communication primitives that are called from within each task. Both blocking and non-blocking send operations are provided. The blocking

send delays until the message has been received and acknowledged. The non-blocking send delays only until the message buffer on the sending machine has been freed. A blocking receive is provided that delays the process until a message of a specified type arrives. A non-blocking probe operation is provided to determine if a message of a specified type is awaiting receipt. APPL provides a broadcast send operation, and the global data reduction operations sum, product, max, and min.

To realize the goal of providing good performance across a variety of platforms, the mapping of APPL messaging routines to underlying hardware varies based on the type of host. Two basic categories of machines are recognized by APPL. For distributed memory MPP machines, such as the Touchstone Delta, the APPL calls are mapped directly to the machine's native messaging library. On shared memory machines, such as the Silicon Graphics IRIS4D series, APPL supports a *process cluster*. Messages between processes in the same cluster are passed using shared memory. If two or more shared memory machines are available, messages within clusters take advantage of shared memory, while messages that cross cluster boundaries are sent via TCP sockets. When a collection of workstations is available, it can be treated as a distributed memory machine with one process per workstation, or it can be treated as multiple shared memory machines, with a process cluster on each workstation.

The configuration of a parallel program for the different cases above is specified statically by the user in a *procdef* file. This file lists the tasks that are to be placed on each available processor. The format of the procdef file varies depending on the platform used and encapsulates the user's directions to APPL on how to distribute the workload across the available hosts, and the size of each process cluster. Thus, the source code for an APPL program is not changed when moving the program to a different type of platform, but the procdef file for the new platform will be different.

At execution, an initiator process is run that reads the procdef file and creates the needed processes on each processor. When process clusters are used, one process within each cluster is designated as a master process. This process creates the other processes that make up the cluster, and creates and manages the shared memory region the processes use for exchanging messages.

APPL supports a wide variety of computer architectures, but does not provide cross-architecture calls. With performance as a primary goal, it was felt the need to convert data in cross-architecture calls imposed unacceptable overhead.

2.1.3 p4

The p4 system is similar to PVM and APPL in its goal of providing a message-passing library, callable from C or FORTRAN, for constructing portable parallel programs. It is closest to APPL in its implementation, in that both make use of shared memory and TCP sockets for communication. Like APPL, p4 provides two forms of asynchronous sends, one that blocks until the message is received, and one that blocks only until the message is sent. A blocking receive is provided, as is a probe operation for checking whether a message is available for receipt. A broadcast send is

provided, but multicast is not supported. Like PVM, p4 supports heterogeneous machines through the use of XDR when encoding data elements into messages. Unlike PVM, the p4 user controls when XDR is used. This is accomplished through the use of different send and receive primitives. For example, `p4_send` transmits the message buffer as is, while `p4_sendx` will first use XDR encoding on the message buffer. An additional parameter is included in `p4_sendx` to indicate the type of the data in the buffer. Even when XDR is specified, p4 optimizes by not invoking the XDR encoding unless the sending and receiving machines use different data formats. A set of global data reduction operations is supported to allow the determination of max, min, sums and products.

As was the case with APPL, p4 optimizes the transmission of a message through the use of process clusters on shared-memory architectures. Each cluster has a master process and a collection of zero or more slave processes. p4 extends the shared memory abstraction by allowing the programmer direct access through a set of functions that allocate and free shared memory regions. In addition, p4 provides monitors, a synchronization method for controlling access to the shared memory. The monitors and shared memory regions are available to processes within the same process cluster, but cannot be shared across cluster boundaries.

The configuration of a program is controlled by a *procgroup specification*. This indicates the machines to be used and the number of slave processes to be created on each machine. The specification can be in the form of a file read by p4, similar to the procdef file used by APPL. However, p4 extends this idea by allowing a program to create a *procgroup data structure*, then create processes according to this data structure. This second option allows the application to make some choices after the start of execution before deciding on the needed configuration. In either situation, the actual creation of the processes is static once the procgroup specification is created. Typically, one process cluster is created on each machine; however, it is possible to create multiple process clusters on a machine by listing it multiple times in the procgroup specification. Processes that are created through the procgroup specification can communicate with processes outside their own process cluster. There is also a limited ability to create additional processes dynamically once execution has begun using a library routine. Such processes reside in the same cluster as the creating process and are limited to communicating only within the cluster.

p4 supports the master/slave programming model quite well, and the use of shared memory both improves communication performance and simplifies the sharing of data structures using monitors. However, the FORTRAN support within p4 is not as complete as the support provided for C. In particular, the shared memory and monitor commands are not available to FORTRAN programs, even though message passing within a FORTRAN process cluster is still implemented using shared memory.

2.1.4 MPI

The success of message passing systems for implementing parallel algorithms has lead to creation of a standard Message-Passing Interface (MPI) [MPI94]. This effort has recently resulted

in a standard that specifies both C and FORTRAN interfaces to a system that supports process clusters, asynchronous message passing including broadcast and multicast, and a collection of global data reduction operations. It is anticipated that existing message-passing systems will soon support at least a subset of the MPI interface. Indeed, one goal of MPI is to provide a consistent interface for programmers to use when implementing parallel algorithms. This will allow execution of the resulting parallel program on top of any system that supports MPI.

2.1.5 Evaluation

Message-passing libraries have some potential as an interconnection system. In particular, they support process creation across a variety of machines and provide a mechanism for exchanging data among processes. Support is generally provided for a variety of parallel programming models, and is available for both FORTRAN and C, the most commonly used languages in scientific programming.

However, message-passing libraries lack adequate support for interconnection in two key areas: heterogeneity and configuration. The heterogeneity shortfall is primarily due to the orientation of such libraries. They have performance improvement as a primary goal and seek to achieve this through implementing parallel algorithms, i.e., spreading the work among a collection of processors. No direct support is provided for connecting another application to the parallel algorithm, or for configuring such a heterogeneous collection. For example, no provision is available for connecting a PVM program with a p4 program. The MPI standard can be seen as a partial solution since it seeks to replace the current collection of message-passing libraries with a single interface. This only facilitates the connection of MPI programs, but leaves open the issue of how to configure, for example, an already developed PVM program into an MPI application, or, as another example, how to include a vector program or visualization tool. Finally, message-passing requires the programmer to deal with the problem of encoding and decoding the data directly every time data is exchanged. For a programmer familiar with the parallel constructs used in the message-passing libraries, this is not too difficult, but it can be a serious drawback for someone wanting to connect a collection of vector codes.

2.2 Systems That Recognize Heterogeneity

Exploiting heterogeneity in scientific applications is the goal of several research projects [Chen93, Freund93, Khokhar93, Wang92]. These projects strive to recognize the inherent heterogeneity within the application and (in some cases, automatically) partition the algorithm to run on a collection of heterogeneous processors. This work includes research in the area of hardware design, including the design of multi-processor, heterogeneous machines that combine processors in unique ways to exploit both the parallelism and heterogeneity present in the target applications. Work in this area also involves low-level software support for communications among processors,

and higher-level compiler and language design work to detect heterogeneity. This general line of research attempts to improve the performance of specific algorithms through parallel exploitation of heterogeneity and through the ability to execute multiple algorithms, or multiple instances of the same algorithm, on different platforms.

2.2.1 Image Understanding

Image understanding is the task of taking an image and determining the objects that make up the scene and perhaps taking actions based on the scene. One obvious application of this technology is to control robots in situations where human control is infeasible or impossible, such as a robot exploring Mars. The large computational requirements have fueled a drive toward parallel processing; however, the variety of algorithms employed has made any one type of parallel architecture unsuitable for solving the overall problem. Thus, image understanding has become a motivating application for work in recognizing heterogeneity [Weems93].

There are three commonly accepted categories for describing the processing necessary in image understanding. At the low level, the work is image oriented, concentrating primarily on the pixels. The work may involve operations applied to all the pixels in the image, or to certain subsets. The work on subsets of pixels will typically extract features such as lines, curves, etc. Often the processing at this level is “embarrassingly parallel” and maps well onto data-parallel architectures. Depending on the types of features expected in the image, it may prove faster to employ a MIMD multiprocessor, with each processor looking across all the pixels for a single feature. Even better, of course, would be a set of SIMD arrays operating in parallel to exploit both data and control parallelism.

A typical intermediate level activity involves identifying objects composed of features extracted during low level processing. These features, called *tokens*, can be grouped in a number of ways, depending on the application. For example, edge tokens might be grouped into long straight lines, and straight lines might be grouped into parallel and orthogonal sets or into specific geometric arrangements. Edge tokens might help to define a region, or a region might be used to identify groups of tokens that make up a feature. As tokens are combined into recognized features, these new features become tokens. These larger tokens may be subject to further grouping.

The grouping process is driven by a set of rules. The technique is to search for sets of tokens that satisfy one or more of the rules. Initially, this process has a vast amount of potential data parallelism, since the search to find matches to a given rule initially must look through the entire set of tokens. The parallelism quickly diminishes as the selection process eliminates most of the tokens from consideration. The selected tokens are further compared with each other as additional rules are brought into play to attempt to identify larger objects in the scene. This process may involve only a single thread of control, but with many (possibly parallel) branches as the many-to-many comparisons are made among the selected tokens. A machine for this activity needs data-parallel processors for the initial phase of token elimination and control-parallel processors for the task of

combining the selected tokens into larger features. Such a combination would also need to address problems of data movement, since the selected tokens still represent a potentially large quantity of data to be shipped between the data-parallel and control-parallel processors.

The highest level of processing is knowledge based. Once objects have been identified at the intermediate level, they can be collected together to create larger objects and build an understanding of a scene. Consider, for example, a collection of objects making up the front view of an automobile: headlights, front grill, air intake, bumper features, etc. The task at the highest level might be to identify the particular make of car, perhaps even the specific year of manufacture. One mechanism used is a *blackboard*, a shared data structure with hypotheses about elements of the scene. Parallel processes are invoked to gather, process, and evaluate information that can add elements to the blackboard, or that confirms or refutes elements on the blackboard. For example, one process may positively identify a headlight on one side of the image. This might become a supporting argument for another process recognizing a second, less distinct, headlight on the other side of the image. A database would be available containing relationships among objects. Processes searching this database may follow branches of a tree looking for relationships to fit the features found in the image. Various nodes in the tree may contain graphs that provide rules for connecting features together. For example, a tree may contain rules for determining the make of a car from the features that make up the front of the car. Each node in the tree would contain a graph showing the relationships that exist among the headlights, front grill, etc. This highest level is a good example of a MIMD application where the component processes are not all replicas of the same process, but different processes working on different parts of the image.

At the hardware level, work is focusing on methods for combining the different types of parallel processors, such as data-parallel and control-parallel. The key to this work is not so much the internal architectures, but how to connect and control the processors. This also requires advances in associated resources, such as sufficient bus bandwidth and adequate memory ports. Designs have included a master processor with a number of directly connected slave processors of various types, and a collection of peer processors of different types. Both cases make use of direct, fixed mappings between the different processors. A better approach will be the design of a basic parallel processor chassis that lets the buyer configure it with additional parallelism to suit the specific application.

Software efforts are concentrating on some of the same issues raised by message-passing systems, including low latency protocols and providing data format heterogeneity. Additionally, image processing systems need to address language and compiler design to recognize the need for heterogeneity in the algorithm and map these needs onto the available processors. Systems have been hand-designed to achieve reasonable performance, but are difficult to modify. Languages and compilers that can recognize the type of processor needed by an algorithm and automatically assign the best available processor will greatly facilitate the design of new imaging systems. Such work requires the compiler and operating system to handle situations where the right type of

processor may be available, but not in sufficient quantity. The system will then map a particular algorithm, or portion of an algorithm, onto a less suitable processor in situations where the most appropriate processor is better employed on another piece of the problem.

2.2.2 Optimal Selection Theory

The Optimal Selection Theory (OST) [Freund89] is a tool for selecting the optimal configuration of machines when executing an application consisting of a heterogeneous collection of algorithms. The approach assumes an application is subdivided into *code segments*, each of which can have a different programming model, and thus a different preferred machine architecture. The code segments are executed sequentially and are thus ordered in time. Each code segment is made up of one or more *code blocks*. When more than one code block is present, they represent the parallelism inherent within a segment.

OST makes two assumptions: the code blocks within a segment are homogeneous in their machine needs, and there are sufficient machines of each type to handle all the code blocks within a segment simultaneously. OST recognizes the principal machine types, including MIMD, SIMD, vector, etc. Code profiling and analytical benchmark results are used to assign machine types automatically to each code segment with the goal of achieving the best performance. It is also possible to impose a constraint, such as cost, on the performance goal.

The basic OST theory has been extended in several ways. AOST (Augmented OST) [Wang92] extends the theory to consider the problem of assigning code blocks within segments when there are not enough machines of the optimal type, but additional machines of a non-optimal type are available. Again, code profiles and analytic benchmarks are used to find the best mapping of code blocks to available processors. AOST also extends the original theory by considering the situation when non-uniform decompositions of code segments are possible. This can happen in two ways: when the code blocks themselves do not have equal amounts of work to perform, or when multiple machines of a type are available, but are not identical. This last case occurs, for example, with two CM-2 machines containing differing numbers of processors. AOST attempts to identify these situations and generate a mapping that makes best use of available resources.

HOST (Heterogeneous OST) [Chen93] extends the OST and AOST to situations in which the code blocks contain heterogeneous subtasks. It attempts to find fine-grain mappings onto the heterogeneous processor suite that will minimize the execution time. HOST uses a hierarchical arrangement of available processors. Specifically, processors are grouped into clusters where a cluster can contain a single processor, a collection of similar processors (e.g., the processors in a Paragon), or a collection of dissimilar processors, possibly spanning machine boundaries. Cluster boundaries are established to allow a cluster to be fully connected to other clusters at the same level. Thus, the grouping of processors into clusters is dependent on the communication available: processors connected in a hypercube would be grouped differently from processors connected in a mesh. As with other variants of the theory, the goal is to improve performance by mapping tasks onto the

available processors. HOST differs in that it maps fine grain tasks in addition to medium and coarse grain tasks.

2.2.3 Evaluation

Work on recognizing heterogeneity goes beyond the support for heterogeneity offered by message-passing libraries, and thus comes closer to the goal of an interconnection system. In particular, this work accepts a wider range of programming models, including non-parallel models such as vector codes. The work on recognizing heterogeneity and automatically allocating tasks to available processors can improve the situation by reducing the work needed to connect applications.

The two main drawbacks of these approaches are the current state-of-the-art, and the need to annotate the source. At present, the work in this area is preliminary in nature, or targeted to specific applications such as image understanding. The support for hardware heterogeneity is under development, but current systems are custom made for a specific application. The network support between dissimilar processors is still in a state of flux given the current work on high-bandwidth network protocols and interconnect switches. As a result, the software effort remains primarily in the design stage. Perhaps the most promising part of this research is the work on using compilers coupled with suitable system support to assign tasks to available hardware. However, it appears likely that either new languages or annotations in existing languages will be necessary to implement this approach, which may limit the ability to incorporate existing applications. Finally, much of this work is concentrating on fine-grain parallelism, while the task of connecting applications into meta-computations is essentially a coarse-grain problem.

2.3 Configuration Management Systems

A number of projects are investigating issues of configuration management, i.e., how to create and control collections of processes. Most include the abilities desired in an interconnection system: creating meta-computations at runtime from collections of components and changing the configuration of the meta-computation during execution. Unfortunately, the support for heterogeneity is sparse; typically, the systems only provide support for a variety of machine architectures, with no support for multiple languages or programming models.

2.3.1 Distributed Programming Systems

One difficulty with evaluating these systems is the different definitions used for configuration management. In addition to the configuration features needed for an interconnection system, many systems also support the ability to automatically update software for different target machines [Callahan91], and fault-tolerance features such as restarting components when failures occur and

providing atomic actions in the communications layer [Becker94]. While interesting in their own context, these features are not necessary in an interconnection system.

For example, [Zimmerman94] describes a system that supports the initial configuration and subsequent re-configuration of long-running applications. The system uses C++ and a specification technique to support the integration of application and management functionality into each component. This creates an environment where configuration, fault, and performance management form an integral part of the runtime environment for a distributed application. The system can detect faults and re-configure the application by transferring the saved state of a component to a new host and re-starting it there.

The Regis programming environment is similar in its support for C++ and the use of a specification language, called Darwin [Magee94]. Regis, however, concentrates on the communication needs of the components. The user is responsible for developing the functionality for the component. Darwin is then used to describe the communication needs of each component. The programmer can also describe composite components by specifying the components each contains, and the bindings between those components. Regis configures the application by satisfying the communication needs of each component. The resulting distributed programs are statically defined at runtime. However, this has been extended to allow a type of dynamic configuration based on lazy instantiation — avoiding the creation of component instances until actually invoked — and by allowing recursive descriptions.

As another example, the Mentat Programming Language (MPL) [Grimshaw93] has configuration aspects. MPL uses annotations to C++ to achieve high performance on parallel architectures. MPL uses compiler techniques to automatically recognize parallel constructs within an object and map the object onto multiple processors. Configuration is also supported through annotations the programmer can add to the MPL source. The annotations tell the compiler and runtime system that certain objects can be executed concurrently. The hierarchical object structure can support an arbitrarily deep collection of objects comprising those automatically implemented in parallel and those with annotations directing concurrent execution. MPL provides a degree of heterogeneity in its support for different types of architectures that can be combined in the same problem.

The next section describes one project that comes closer to meeting the goals outlined in Chapter 1 for an interconnection system by accommodating heterogeneity in both programming languages and machine architectures.

2.3.2 Polylith and Polygen

The Polylith system supports configuration through the use of a module interconnection language (MIL) [Purtilo94]. Each module to be included in a computation has a specification in MIL, called a *primitive specification*, that describes resources defined within the module and resources the module will need from other sources. There is also a *composite specification* for the meta-computation that lists the constituent modules and specifies the bindings between the

modules. It is possible to specify only a subset of the bindings, allowing the underlying system to infer the rest. In this case, the system will choose from among those modules that provide the requested service, often by selecting a library that provides the service and binding it into the application.

Polylith uses a *software bus*, an abstract communications mechanism into which software modules are plugged. The software bus creates the connections among the applications. The form the software bus assumes in each case is dependent on factors such as the location of the modules that make up the application and preferences stated by the user. For example, two modules executed on the same machine may be placed in the same process by the software bus if they are written using the same language. If developed using different languages, they may still be placed in the same executable, but along with a coercion stub that handles inter-language conversions. If placed on separate hosts, each module will have a stub to handle inter-machine (and, possibly, inter-language) conversions. In this case, each module and its stub will be a process, which communicates with other modules through a software bus that contains the necessary communication functions. The bus can take on a more active nature with additional processes involved. For example, routing daemons may be used to improve throughput by collecting messages bound for a common destination, gateway processes may provide security, or a process may serve as a converter between two network protocols. The ability to place extra processes in the software bus also allows for profiling and network monitoring.

The Polygen system [Callahan91] uses Polylith to provide a higher-level packaging service that builds the binaries for each module of the application. A module is created for the intended architecture by linking its object form with the appropriate stubs, conversion routines, and communication libraries. To create an application from the modules, Polygen uses a database of information about the computing environment. The database is created for each site and each language by an administrator using Prolog rules. These describe the features present at the site and the constraints imposed by the environment, such as which network protocols are supported by which hosts. An inference system is then used to examine the application's primitive and composite specifications, and compare them with the site and language constraints to determine if one (or more) possible applications can be created. If one is found, Polygen goes on to create the correct binaries for each module usually by creating and executing the appropriate makefiles.

By using Polygen, the user can create the binaries for an application without having to make changes in the source code or in the primitive specification files. To create a different application, for example by binding modules to different hosts, the user can invoke Polygen again after making the appropriate change in the composite specification file. Polygen may create an application that allows modules to be bound into a single process for execution on the same machine, or create the appropriate stubs and bind them along with communication libraries into each module to create an application that spans multiple machines. The only change required by the user in either scenario is to the composite specification file.

2.3.3 Evaluation

Polygen and Polylith offer a number of advantages. The most important of these are the ability to create in an automatic way the correct binaries for an application with little change needed to the source code. The main change needed to port an application to a different collection of hosts, for example, is in the composite specification file. Polygen also offers the ability to infer from the source code much of the information needed for primitive specification files, thereby simplifying this task for the user. In addition, Polylith offers a mechanism for easily incorporating changes in an environment, such as the addition of a new network connection. A one time change to the environment specification is made by an administrator, which makes the change available to all.

The main drawbacks of Polygen/Polylith as an interconnection system lie in the static configuration method. To create an application in Polygen, the user specifies all the modules that will be used. While the system is capable of inferring the module interconnection structure needed to satisfy references, there is no corresponding ability to change the composition of an application once execution has begun. Thus, the user must completely specify all resources that might be needed, rather than being able to add modules and hosts dynamically. A smaller drawback lies in the current lack of support for Fortran, making a large number of scientific codes unavailable for incorporation into applications. However, given the languages currently supported (C, Ada, Pascal, and Lisp), it would no doubt be easy to rectify this omission.

2.4 Summary

This chapter has presented a number of current systems that satisfy some of the properties required for an interconnection system designed for scientific computation. The fact that none provide a complete solution has been a motivating factor in the development of the research described in this dissertation. Scientists should be able to concentrate on their research without having to spend time working around the limitations of existing systems. An interconnection system must provide support for heterogeneity that allows multiple machine architectures, various programming languages, and different programming models to be included in a meta-computation. It also needs to provide configuration management that supports easy composition of meta-computations from components and dynamic re-configuration during execution. The Schooner interconnection system, which provides these features, is described in the following chapters.

Chapter 3
THE SCHOONER INTERCONNECTION SYSTEM

A model of scientific computing is proposed that recognizes the diverse requirements of the various algorithms needed for a scientific computation. In this model, each algorithm of the computation is independently developed into a component. These components are then combined into a single meta-computation executing on a collection of machines connected by local-area and long-haul networks.

The central requirement in this model is an interconnection system that allows the components to freely exchange data. The system provides a means of communication among the components that accommodates the heterogeneity inherent in the computation. The distributed nature of a meta-computation adds the requirement that the system provide configuration capabilities as well.

The focus on scientific applications imposes two additional requirements. First, the system must be easy to use, so that the computational scientist who actually constructs the application will be willing to make the transition from more traditional methods. Second, the impact on the source code for each component in the system must be minimized. These codes are often independently developed and the source may be unfamiliar to the programmer, maintained by a different group, or simply unavailable.

The Schooner interconnection system implements the scientific meta-computation model described above. It supports heterogeneity among components, providing a transparent means of connecting a wide variety of algorithms into a meta-computation. Schooner provides two forms of configuration: static and dynamic. With static configuration, the user describes the configuration of the application — that is, the components and machines to be used — prior to execution. The runtime then starts the components and executes the meta-computation. With the dynamic option, the user has the additional ability to control the meta-computation during execution by adding, deleting, and moving components. This facility allows the user to exercise more interactive control, which can be especially useful for adapting the application to changes in requirements or execution environments.

This chapter describes the basic Schooner system, which provides full interconnection capabilities together with static configuration support [Homer94a]. The static configuration support is modeled after the normal method of executing an application from the command line. In particular, the user lists the {component, host} pairs as arguments when starting the meta-computation.

3.1 Building Block Paradigm

Schooner is a component interconnection system that supports the construction of scientific applications according to a building block paradigm. A meta-computation is configured by the user from a collection of independently developed components, with various options provided for

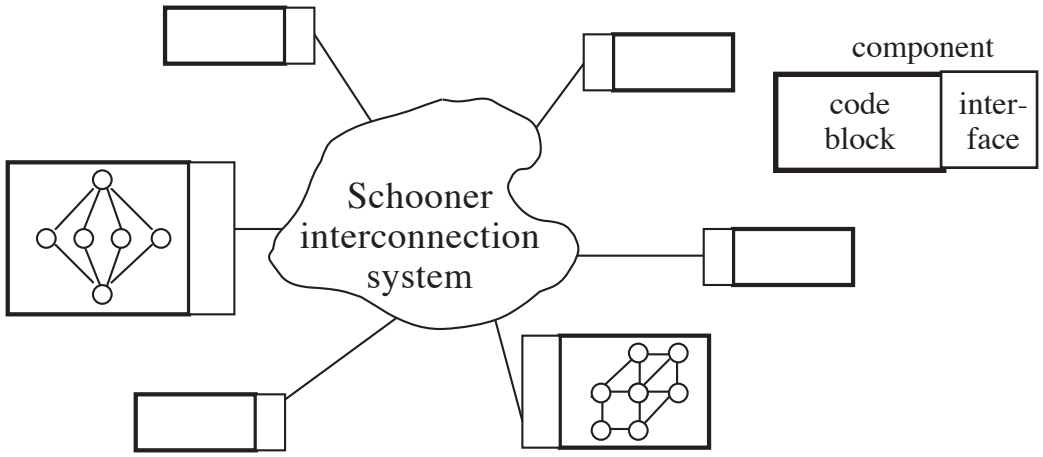


Figure 3-1: Meta-computation in Schooner

binding components into the meta-computation and mapping them onto the available hardware. Schooner handles the connections between components, including providing a name service to locate the component, data conversion, and selection of the appropriate network protocol.

In the building block paradigm, each component is composed of two parts: a *code block* and an *interface*. Each block is a code or tool the scientific programmer wants to use in the application. For example, in a fluid dynamics meta-computation, a grid generator, a fluid dynamics computation, and a visualization tool might be combined to allow the user to generate sample grids, perform a computation on the grid, and view the results of the computation on top of the grid. The user might then make modifications to the generated grid and/or the computation and iterate again. Each of the three tools would be one code block.

The component interface is a stub and collection of library routines that are attached to a code block to carry out data conversions and implement the message passing required to transfer data between the processes that execute the components. Each interface is described by an interface specification that defines the services made available to other components and the services the component requires that are elsewhere in the application. At runtime, the Schooner system uses these specifications to match requirements and form the necessary connections between the component implementing a given service and its users (Figure 3-1). In the fluid dynamics example described above, the grid generation component, for example might produce grids in multiple formats to satisfy the needs of different computation and visualization tools. The specification for such a tool would have a description for each grid type. Similarly, the specification for a computation tool would have a description of its input grid. When the two tools are combined, the Schooner runtime would perform type checking to ensure the compatibility of the output and input grids. Doing such a check without an interface specification would be difficult at best.

Schooner realizes control flow in the meta-computation using the remote procedure call (RPC) paradigm. Thus, the interface specification is simply a description of the procedures made avail-

able by a component for external invocation, as well as those external procedures that the component invokes. The runtime system configures applications by starting components on their target machines, collecting information from the interface specification of each component to build a location database, and checking the appropriate procedure specifications for type compatibility.

3.2 Constructing Meta-computations in Schooner

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

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

Once a component is written, the only additional work involves writing a specification file that will be used by the Schooner system to create the interface for the component. As explained in more detail below, this file contains a specification written in the UTS specification language for each procedure either imported or exported by the component. The specification lists the number and type of the parameters for each such procedure.

Schooner makes use of a sequential execution model where control passes from procedure to procedure. Within this model, however, parallel algorithms can be used by encapsulating the parallel algorithm inside a component, as illustrated in Figure 3-2. These parallel algorithms can execute on specialized hardware such as hypercubes, or could even be realized using a system such as PVM on a collection of workstations [Sunderam90, Beguelin91]. In either case, the role of Schooner is to connect the procedure to the other parts of the application. Thus, Schooner is able to provide connections to a variety of architectural and programming models to further the goal of increased interaction and connectivity, while maintaining the simple programming model implied by RPC and an easy-to-use interface.

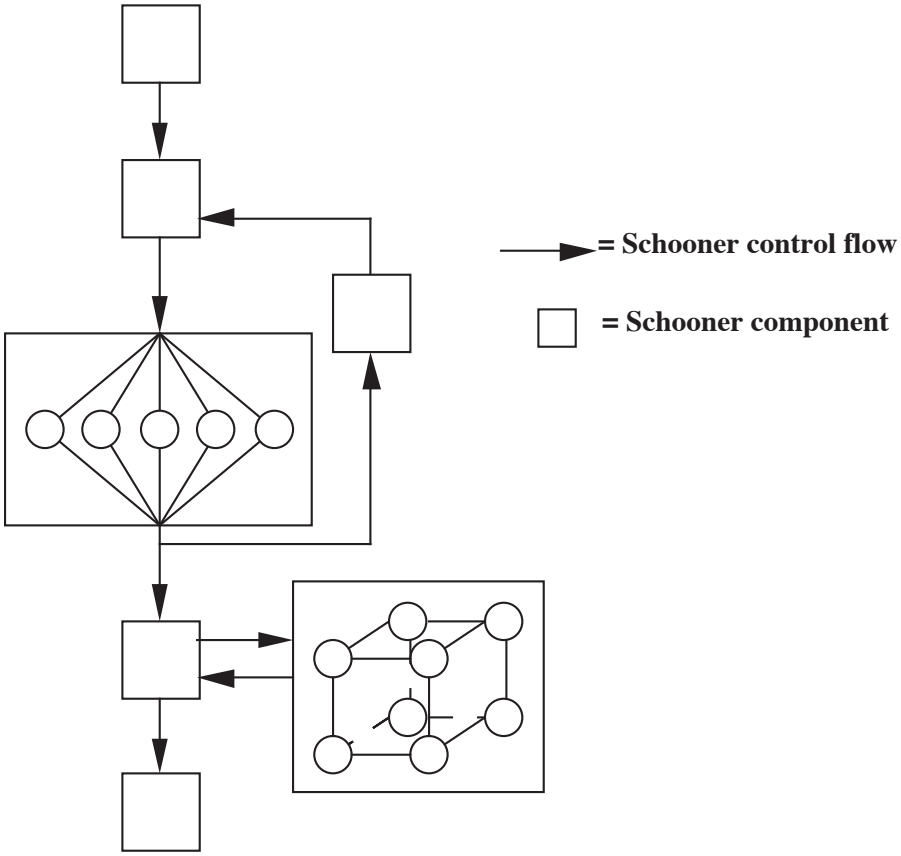


Figure 3-2: Parallel Algorithms within a Schooner Application

3.3 Schooner Services

Schooner accomplishes its interconnection goal by providing four services: a specification language, a data exchange library, a collection of stub compilers, and a runtime system to implement control flow and handle communications. To a large extent, these services are distinct and orthogonal, so that changing the implementation of one does not affect the others. Each service can also be used for other purposes as well; for example, the UTS type specification language has also been used as the basis for a command language interpreter [Andrews87]. Table 3-1 outlines the steps required to adapt an application to utilize Schooner's facilities, illustrating the role played by each service. The function of each service is elaborated below.

3.3.1 The Universal Type System

The Universal Type System (UTS) combines two of the four Schooner services: an intermediate data representation and a specification language. The intermediate data representation allows

1.	Form components from procedures that make up program.
2.	Write UTS specification file for each component.
3.	Run the appropriate language's stub compiler on each specification file.
4.	Compile source and stub files for each component. Link with UTS and Schooner communication libraries.
5.	Start Schooner Server on each target machine.
6.	Execute program by starting the Schooner Manager, specifying host for each component.

Table 3-1: Steps in Converting and Executing an Application

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

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

The specification language portion of UTS is used to specify the interface, essentially the number and type of the arguments for each procedure that can be called remotely in the application. There is both an *import* specification and an *export* specification; the import specification is associated with each component that invokes the procedure and the export specification is associated with the component containing the code for the procedure. For example, the interface specification for a component that exports a fan code in the NPSS jet engine simulation project described in Chapter 1 is shown in Figure 3-3.

The keywords `val`, `var`, and `res` indicate the direction of data transfer for each parameter, while `export` indicates that the component will provide this procedure for other components to call. Components wishing to call the procedure `fann` would then have a matching specification that uses the key word `import` rather than `export`. The words in quotes are treated as comments

```
export fann prog (
   "pin"    val float,   "pout"   val float,
   "tin"    val float,   "tout"   val float,
   "xspool" val float,   "cshift" res float,
   "hout"   res float,   "eout"   res float,
   "pmap1"  var array [14] of float,
   "pmap2"  var array [14, 12, 3] of float,
   "pmap3"  var array [3] of float)
```

Figure 3-3: NPSS Fan Code Export Specification

by UTS, and will be used as parameter names during the creation of the interface.

The UTS language allows specification of all the typical simple data types (integer, float, double, character, boolean), as well as the structured types array, record, and string. Procedure values are also supported. Schooner treats each such value as a capability to the named procedure, thereby allowing other components to perform callback operations by invoking the capability. Data types that are not present explicitly in UTS can often be described in terms of available types; for example, the FORTRAN complex type has no direct equivalent in UTS, but can be described as

```
record {float, float}
```

and the double complex type as

```
record {double, double}
```

In most cases, the only difference between the import and export specifications is the use of the keyword `import` or `export`. UTS does, however, allow the import to be, in essence, a subset of the export. This can increase the usefulness of an exported procedure for configurable applications by allowing it to handle a variety of inputs, rather than the alternative of writing a collection of procedures all performing nearly the same operation on slightly different inputs. As a small example, a procedure with the specification

```
export sample prog (val (integer or float))
```

can accept either an integer or a float as its argument. A component wishing to call this procedure can use either of the following specifications:

```
import sample prog (val integer)
import sample prog (val float)
```

Since components in Schooner are separately compiled, often on different machines, typechecking is deferred until the program is executed. The first time a component makes a call to another component's exported procedure, the import and export specifications are compared to ensure type compatibility. The specification language actually allows parameters to be described by sets of types, which means that import and export specifications need not match exactly. In the example above, since the export specification describes a set that is a union containing both integer and float types, either version of the import specification for `sample` will be compatible.

UTS supports a *represented type* for parameters that either do not map into any data type in the

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

To illustrate the use of representatives, consider the problem of passing command-line arguments to a user-supplied main function. In Schooner, such a function is named `user_main` and upon startup, the system will pass user-supplied command line arguments to it. The export specification for this case would be:

```
export user_main prog (val rep array [-] of string[-])
```

The keyword `rep` indicates the presence of a parameter that the user is taking responsibility for decoding (in this case) or encoding. The code needed to extract the arguments must first extract the size of the array, `argc`, then each string in the array can be decoded. The code for doing this is shown in Figure 3-4.

3.3.2 Stub Compilers

In addition to being used for type checking, the specification file is also used as the basis for creating the interface that is a part of each component. The interface is automatically generated by a stub compiler. The interface consists of stub procedures, one for each imported and exported procedure in the component. The stub procedure primarily contains calls to the UTS library for converting the procedure's parameters between native and UTS representations. There is one stub compiler for each supported programming language. Currently, Schooner has stub compilers for C and FORTRAN; various versions of the predecessor MLP system also supported Pascal, Icon [Griswold90], and Emerald [Black86, Black87, Hayes90].

After stubs for a component are generated from the specification, they are compiled using the appropriate language processor. The resulting object module is then linked with the user's code, the UTS libraries, and the Schooner runtime support libraries to produce an executable. This sequence of steps is illustrated in Figure 3-5.

3.3.3 Runtime System

A component in Schooner is implemented by a process at runtime, which means that a component is also the unit of distribution in the system. As already noted, control flow in the basic model is based on a sequential procedural programming model in which there is logically one thread of control that transfers between components when an RPC is executed. This control flow is imple-

```
user_main(rep_ptr)
   long rep_ptr; /* storage is allocated by the stub */
{
int i, str_len;
long argc, num_dims = 1, str_rep;
char **argv;

/* extract the size of the argv[] array */
uts_dims (rep_ptr, &argc, &num_dims);
if (argc != 0) {
   argv = (char **) malloc (argc * sizeof (char *));
   for (i = 0; i < argc; i+) {
      /* extract one string from the representative
       * note it is returned as a representative */
      str_rep = uts_index (rep_ptr, i);
      /* check the string's type and find its length */
      str_len = uts_length (str_rep)
      if (str_len == -1) {
         …error handling/reporting, bad string tag/size
         return; /* so system will shutdown components */
      }
      /* now get the string from its representative */
      argv[i] = (char *) malloc (str_len + 1);
      uts_decode_string(str_rep, argv[i], str_len);
}
else
   printf ("user_main(): no cmd-line arguments");
dispose_rep (&rep_ptr); /* to free allocated storage */
… continue with user_main() …
```

Figure 3-4: Decoding UTS Represented Data

mented by the communication library portion of Schooner's runtime system, which is linked with every component. The communications library implements the interface to Schooner's dynamic name resolution and interprocess communication facilities. The interface to the name resolution mechanism consists of two routines, one that registers a component with the Manager process (described below) and the other that queries the Manager when an imported procedure is invoked for the first time. The interface to the interprocess communication facilities consists of a send operation for outgoing calls and a receive operation for incoming calls. The send operation takes the marshalled arguments provided by the stub, places them in the message along with return address information, sends the message to the remote component, and handles the return message, passing the marshalled return values to the stub as needed. The receive operation is structured analogously. The Schooner communication subsystem currently supports message passing using either TCP virtual circuits [Postel81] or UDP datagrams [Postel80]. The choice is made by the user when the Schooner program is started. The system can easily be adapted to use other commu-

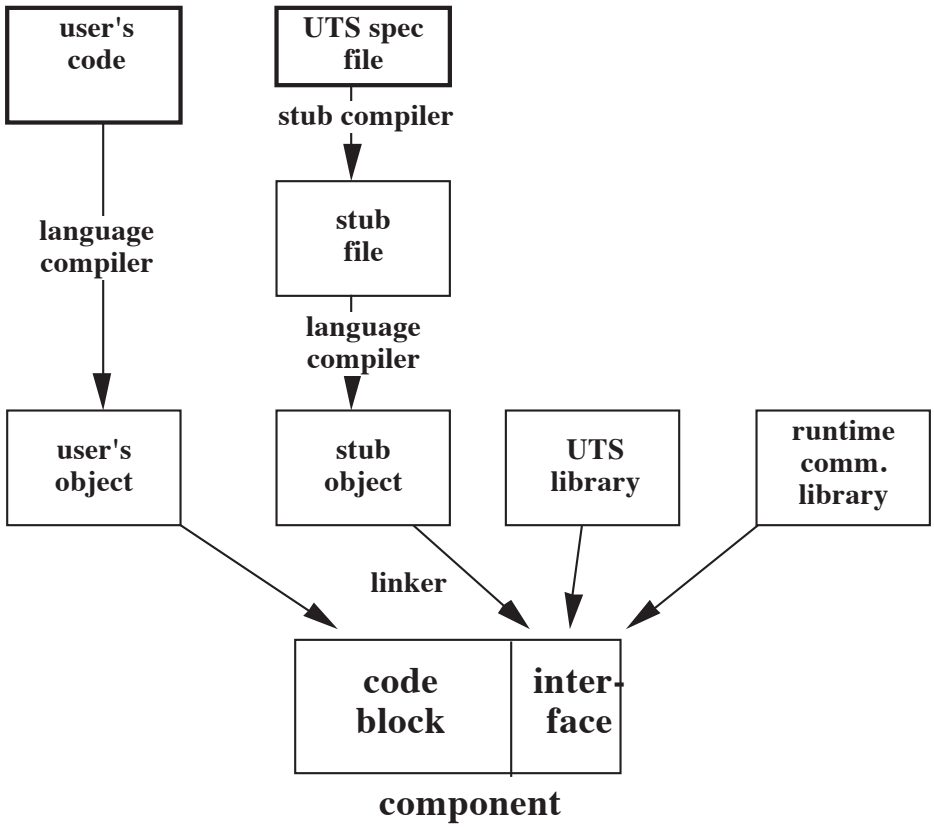


Figure 3-5: Producing a Schooner Executable

nication protocols, however, by simply adding the appropriate send, receive, open, and close functions to the communications library.

The remainder of the runtime consists of two types of system processes. The first is a Schooner Manager process; there is one such process per program and it typically executes on the user's home machine. The second is a Server process; there is one such process per host machine used in the meta-computation.

The Manager is the central coordinator of the application, handling the twin configuration tasks of mapping components onto hosts and binding components into the meta-computation. The mapping task is carried out with the cooperation of the Server. Specifically, for each component, the Manager sends a request to the Server on the target host, passing the name and path of the executable. The Server then starts the component with the location of the Manager as an argument. Any startup errors are reported back to the Manager.

3.3.4 Static Configuration

The basic model uses a static configuration method in which the user specifies the {component, host} pairs at the beginning of execution. This is done by listing the pairs on the Manager's

command line. Schooner also provides for passing command line arguments to the main procedure of the application. The Manager begins the creation of the components comprising the meta-computation by sending start requests to the Servers on the appropriate machines.

Each component becomes bound into the meta-computation through the registration call referred to earlier. As part of the startup protocol, a component receives the Manager's location from the Server. This allows the component to call the Manager's registration procedure. The call contains as an argument the UTS specifications for all the procedures exported by the component. The Manager enters this information into its procedure name database and replies to the component, completing the process of binding the component into the application.

Once all the components have completed the registration process, the configuration phase is complete and the Manager begins execution of the meta-computation by an invocation to the component exporting the main procedure. Control is then passed from component to component through the chain of remote calls as execution proceeds. As described above, name resolution requests are forwarded to the Manager whenever an imported procedure is invoked for the first time. In addition to the name of the procedure, this request includes the appropriate UTS import specification. Upon receiving this request, the Manager looks up the procedure name and performs type checking by comparing the import specification in the request with the export specification in the database. If the match is legitimate, the location of the target procedure is returned to the calling component, which proceeds with the remote invocation. This location information is cached at the calling component so that future calls proceed directly.

When execution terminates, normally through the main procedure returning, the Manager terminates each of the components in the application. Shutdown will also occur in the event of an error. In addition to the usual errors that occur in an application, e.g., floating point exceptions, Schooner will also report startup and type checking errors.

3.4 Incremental Changes

Schooner is an evolving system that has undergone a number of minor changes from the original MLP system. These have included the addition of a number of machines to the list of supported machines, and additions to the UTS specification language and the corresponding changes to the stub compilers and UTS library.

3.4.1 Adding Machines to Schooner

The predecessor MLP system supported Sun workstations and the Digital VAX architecture [Hayes89]. In the course of constructing Schooner, a number of machines have been added, including the Cray YMP, Convex C2x0 series, SGI and Stardent graphics workstations, IBM RS/6000 class architectures (including both workstations and the SP-1 and SP-2 parallel processor), the Sequent Symmetry and the Intel Paragon. We describe here the work needed to incorporate the

Cray YMP. The work involved for the other machines 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 scientific applications. 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 scientific researchers, 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 other machines encountered up to this point, procedure names were converted to lower case by their respective Fortran compilers. On the Cray, the Fortran compiler 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 Fortran stub compiler generates internal names for exported procedures that allow the Manager to identify the names as coming from a Fortran component.

3.4.2 Adding Simple Types to UTS

The UTS type system has been expanded from that used in MLP 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 not 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

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

Table 3-2: Sun-Sun Test (times in milliseconds)

and code generators for the stub compilers. The second change involved adding the appropriate encode and decode functions to the UTS library for each of the supported architectures.

3.5 Performance

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

The results of these experiments are shown in Table 3-2. The method involved having the application calculate the elapsed time after 100 calls to the procedure; in the case where no arguments were passed, the test involved 500 calls to get a more accurate measurement. Each case was run five times. The table shows the results after averaging the five runs and dividing by the number of calls. The encode time is the time the application spent in the stub utilizing the UTS library routines to encode the array of values; the decode time is analogous. This was determined using the results reported by the `getrusage` system routine and then adding together the user and system times. Finally, the percentage column indicates the proportion of the total time that was spent in the two encode and two decode sections of the RPC. The Sun Sparc architecture uses the

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

Table 3-3: Sun-Convex Test (times in milliseconds)

IEEE floating point representation and also stores the bytes in network order, meaning that the encode and decode times are the time needed to copy the bytes for each value and its tag into and out of the message buffer. Given that 8-byte double-precision values are being passed, each encode or decode time represents the processing of 8x(array size) data bytes.

The second series of tests were designed to determine the costs involved when data values must be converted across machines, and the costs involved in doing RPC calls outside a local network connection. In this set, the component containing the main program was run on a Sun Sparc 2 and the other component on a Convex C220; the two machines are located in the same building, but with several network gateways between them. The Convex does not use the IEEE format for its native floating point mode; thus, bit manipulations were required to correctly convert both its mantissa and its exponent. This also involved tests to determine if underflow or overflow occurred. During the tests, the user was the only login active on the Sun, but the Convex was handling a varying number of other users. As in the first set, each test consisted of 100 calls and was run five times.

The results are summarized in Table 3-3. Not surprisingly, the percentage of the total time devoted to encoding and decoding is smaller here, mainly due to the increased amount of time needed to traverse the gateways between the two machines. The encode and decode times for the Convex are also larger than those needed for the Sun due to the time needed to perform the conversions.

While the data conversion aspect of an intermediate data representation certainly imposes

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

Optimizations are possible within Schooner to enhance performance on specific machines, and for specific applications. For example, the Convex float conversion includes a test to determine if the Convex exponent will overflow the IEEE exponent. While necessary in a general solution, such a test could be left out in an application where the user was sure overflow would not occur. Additional optimizations to improve performance are discussed in Chapter 7.

3.6 Other RPC Systems

Many RPC systems have been developed since the original work described in [Nelson81]. Most offer features similar to Schooner's RPC, including external data representations, specification languages, and stub compilers [Almes85, Birrell84, Sun90, Xerox81]. Several of these systems also emphasize heterogeneity, including Matchmaker [Jones85], Horus [Gibbons87], HRPC (Heterogeneous RPC) [Bershad87] and Cicero/Nestor [Huang94].

3.6.1 Basic RPC Paradigm

Most RPC systems use a *client-server* paradigm, where the client is the process requesting a service by initiating the RPC and the server is the process that will carry out the service. This works well in the context of providing operating systems services across a distributed collection of machines, an area where RPC has been extensively used. To provide a service to many clients concurrently, multiple instances of a server may exist, each bound to one client. The canonical example is a distributed file system, such as Sun's Network File System [Sun90] and the Andrew File System [Morris86]. In the basic design, each host in the system provides a server to manage the files resident on the host. A client opens a file through an RPC to the appropriate server. An instance of the server is then created, opens the file and returns the file handle to the client. Subsequent operations on the file are invocations to the same server instance. Accesses by multiple clients to files on the same host are each handled by a different instance of the host's file server.

There are two drawbacks to the client-server approach in the context of scientific computation. First, it requires a different programming model. Components bound into a meta-computation have to be designated as client or server during the design process. Second, and more significant, only the server can export procedures for use by any process. Although servers can make calls to

other servers, the model supports only a limited ability for a server to call a client. A client, when invoking a procedure in the server, can pass a capability to make a callback to a procedure within the client. Thus, only a limited form of recursion involving a single client can be supported. These two factors can have a major impact on the design of the meta-computation, forcing the user to consider calling patterns among the components. Schooner is more appropriate for the scientific programmer as it provides a model that is closer to standard procedural semantics. This application orientation also shows in the configuration aspects. Schooner allows the user to select the components at runtime that make up the application, while the client/server model presumes the server is already executing, or is started in an manner independent of the client. As described in Chapter 5, Schooner also allows for dynamic configuration after execution has started through mechanisms that allow components to join and leave the meta-computation.

Several RPC systems, however, do offer features that are useful in an interconnection system. In particular, systems that support heterogeneity and those supporting a variation on RPC known as asynchronous RPC.

3.6.2 RPC Systems that Support Heterogeneity

HRPC (Heterogeneous RPC) provided the basis for the Heterogeneous Computer Systems (HCS) project at the University of Washington [Bershad87]. The overall goal of HCS is to provide a way to simplify the interconnection of heterogeneous computer systems. HRPC is the cornerstone of the effort, providing a client/server interface for a variety of machines and operating systems. It is possible in the system to write a client program for one architecture that is able to communicate transparently with server programs written to take advantage of the native RPC mechanisms on a variety of platforms, including Sun RPC and Xerox Courier RPC. HRPC does not mask heterogeneity, but exploits it to allow clients to connect to a great variety of servers without having to provide different versions of the client. HRPC can incorporate both services that are built using HRPC and services that are built for various native RPC systems. Clients are able to access either type of service transparently through the provided naming and binding mechanisms. HRPC uses a type interface specification language and associated stub generation to create clients and servers, and delays binding decisions until runtime.

The Cicero/Nestor system focuses on cross-RPC communication in a large heterogeneous distributed environment in which many different RPC protocols may be in use [Huang94].The system implements an *agent synthesis* scheme, which uses a synthesizer to generate implementations of RPC agents from high-level descriptions. Cicero provides the high-level description language, and Nestor the run-time environment to synthesize and activate RPC agents automatically. The two goals are to provide developers with cross-RPC service and customized RPC service. The first provides agents to allow clients to connect to servers implemented in a variety of RPC systems. The customized RPC service facilitates fast prototyping of new RPC systems. The system goes beyond the support provided by HRPC by importing a *remote protocol construction*, a

description of the implementation of the remote server's RPC system, to assist in synthesizing a local agent. This precludes the need to have local libraries on every machine for every RPC protocol available in the system, thereby simplifying the task of adding a new RPC protocol.

Both HRPC and Cicero/Nestor provide transparent access to heterogeneous machines, and assist the developer in writing applications to take advantage of services available. HRPC attempts to present a common interface across a wide variety of machines and RPC protocols. Cicero/Nestor provides a different solution by automatically creating agents that handle the communications between the two different RPC protocols. They do not, however, provide support for connecting to parallel programs, such as a parallel code implemented with PVM. The disadvantages listed in the previous section for system RPC's also apply to both of these systems.

3.6.3 Asynchronous RPC

A number of RPC systems have been implemented that change the semantics of the remote procedure call to increase the concurrency of the client-server pair [Ananda92]. The goal is to allow parallel execution of the client and server, specifically by allowing the client to continue executing after invoking the service rather than blocking until receiving the reply from the server.

An example of such a system is ASTRA [Ananda91]. This system uses a reliable delivery system to guarantee delivery of the invocation message, and to retain the ordering of invocations at the server. The procedure invocation returns a unique identifier to the client that can later be used to claim the reply from the server. A client can claim its replies from a server in any order. The client can also choose not to claim some or all of the replies. Clients have several options for receiving replies. A client can claim a specific reply by blocking until the reply is received, or through a non-blocking operation that will return an error code if the reply is not available. Analogous routines are available that allow the client to claim the first available reply. The system also supports a real-time constraint on receiving replies where the client can specify a time interval. The client will block until the indicated reply is received or the time limit has passed. ASTRA provides a specification language for describing the interfaces of the client and the server, and handles the automatic encoding and decoding of arguments.

ASTRA changes the synchronous nature of a procedure call after the invocation is made. Prior to the first invocation, the server is blocked awaiting a call. The server begins executing when the invocation message is received. The client now continues executing and the two processes are executing concurrently. This is very close to the semantics offered by message passing systems. There are two basic differences. The first is in the implicit synchronization prior to the first invocation from client to server. The server is not executing prior to this point. The second difference lies in the automatic encoding and decoding of arguments.

The concurrent execution in ASTRA has advantages for scientific computing, but at the cost of changing the programming model to one very close to a general concurrency model. Thus, it requires mastery of a different programming style, and increases the burden on the developer and

user to control the application. The extended Schooner model, described in Chapter 5, does offer a limited concurrency model that retains the normal procedure semantics, in particular, the implicit synchronization of procedure calls. There are situations where breaking the implicit synchronization is necessary, and one is described in Chapter 5. However, the use should be narrowly focused to avoid placing too large a burden on the developer and user.

3.7 Summary

This chapter has presented the basic Schooner interconnection system, describing the programming model and Schooner's realization of the model. The model supports construction of components from the user's code block and an automatically generated interface. The Schooner system uses the UTS specification language to describe the interface, and the UTS intermediate data representation to handle data exchanges between machines. The stub compilers generate the interface from the UTS specifications. The Schooner runtime handles instantiation of the components and provides a static method of configuring the components into the meta-computation. Incremental changes to the system have been made including additions to UTS and the list of supported machines. Finally, a review of related RPC systems was provided.

Chapter 4
SCHOONER/AVS META-COMPUTATIONS

The basic Schooner system has been combined with the AVS scientific visualization system [AVS92] to create a software platform for executing meta-computations that allows the user to monitor and steer scientific simulations. In this platform, AVS provides visualization capabilities and an execution framework that allows user interaction with the computation, while Schooner is used to interconnect and manage components that are executed as external computations independent of AVS. The primary focus in this work has been on taking existing scientific codes and incorporating them into this platform to improve the ability of the user to monitor and manipulate the computation.

In this chapter, a short overview of AVS is provided, followed by a description of how meta-computations are written using the combined platform. A number of example scientific meta-computations have been constructed using the Schooner/AVS software platform. Two of these applications are described in this chapter: one involving a molecular dynamics program and the other a neural net code. In both cases, the primary goal was to increase the interaction capabilities of existing programs. In the neural net example, the use of the platform also made it easy to choose one of several different implementations of a parallel algorithm to perform the actual computation.

4.1 Overview of AVS

AVS is a graphics system for displaying images generated by scientific computations. The data model is oriented strongly toward these types of applications, with an underlying assumption that the data represents a two- or three-dimensional grid with values located at each point in the grid. To manipulate both the data and the resulting image(s), AVS provides a large palette of tools. Examples of data manipulations include filters to reduce the amount of data and perform computations such as determining vector magnitudes, and modules to read a variety of data formats and convert these to AVS data fields. Images can be manipulated through rotations of the objects, reflections, positioning of lights and choice of lighting models, applying color to objects according to data values, etc.

A major feature of AVS is the Network Editor where the user can construct dataflow programs. Computations in these programs are performed by pieces of code known as *modules*. Each module typically acts as a filter, receiving data from other modules, transforming it in some way, and passing it on. Connections among the modules are specified graphically as illustrated in Figure 4-1. The editor allows a given module to be chosen from a menu, dragged into the desired position, and connected to other modules to indicate the data flow. The resulting network of modules realizes the entire process needed to render the image as a dataflow network. Data in the network is typed, with colored lines representing the different types. AVS supports such types as

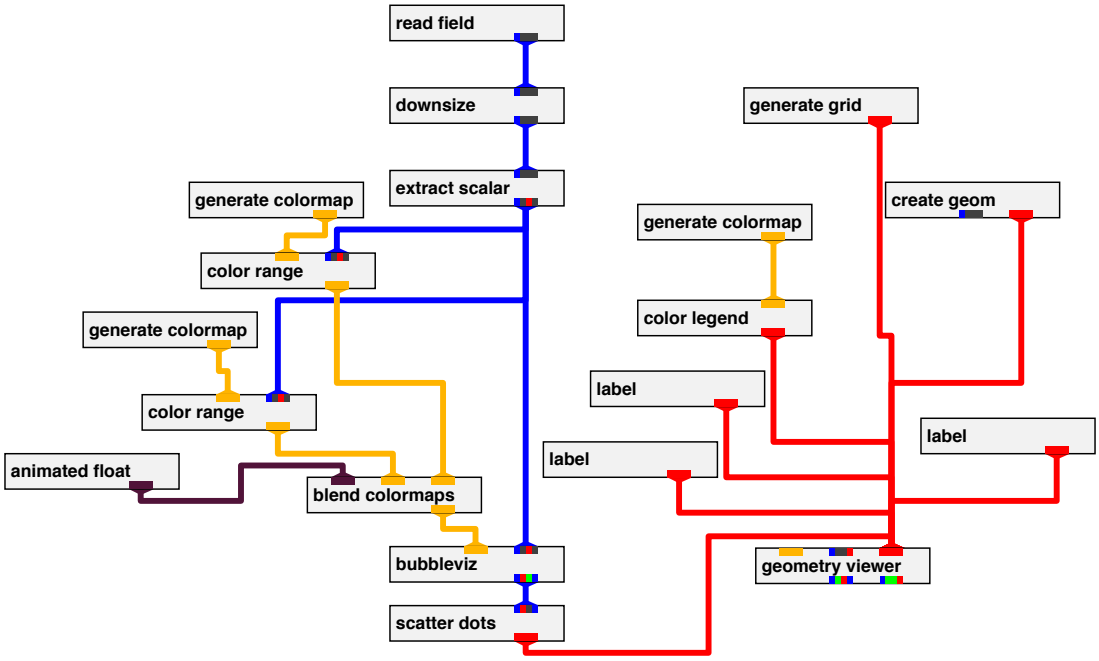


Figure 4-1: Example AVS Network

color maps, geometries, and fields. The field type is the most common type the user will handle directly, since it contains the data to be displayed. For example, the read field module in Figure 4-1 is used to access a data file that specifies a vector field. The dataflow then takes the vector field to the downsize module, used in this network to reduce the amount of data by removing every nth element from the vector field in each dimension.

In addition to receiving values through the network, a module can accept input from the user through parameters. A module that uses parameters will have a *widget* such as a dial, slider, or type-in window appear on the screen for each parameter. Figure 4-2 shows the widgets for the generate grid module from the network in Figure 4-1. This module allows the user to specify a flat grid that will cover the xy, xz, or yz planes. The *width*, *height*, and *depth* widgets determine the size of the grid, and the *nx*, *ny*, and *nz* widgets specify the number of grid lines in each dimension. The *XY*, *XZ*, and *YZ* widgets are toggles that specify which planes are included in the generated grid. Finally, the three dial widgets determine the origin of the grid.

The flow of data and scheduling of module execution is implemented by the AVS kernel. A module is considered to be ready for execution if new data is waiting at one of its input ports as the result of execution of another module, or if one of its input parameters has changed. As each module completes execution, its output data, if any, is passed along to “downstream” modules. Modules can have side-effects as well, with the most common being the display of images or the creation of output files.

In addition to a large selection of standard modules, AVS includes provisions for user-written

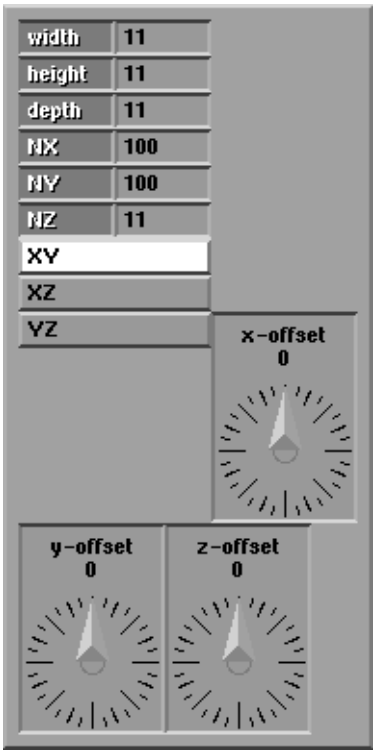


Figure 4-2: Generate Grid Widgets

modules as well. These modules have access to the widget mechanism and can declare widgets of any type to allow the user to input values to the module. Also, the user-written modules can accept as input and can output any of the typed data flows available in AVS. Additionally, the user can describe a different typed data flow, using a C-style structure, and pass data of this type among the user-written modules.

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

4.2 Using Schooner with AVS

The wide variety of visualization tools provided by AVS and its support for user-written modules make it an attractive tool for use as the visualization component of a heterogeneous appli-

cation. In building the Schooner/AVS platform, it was not necessary to alter substantially either AVS or the way in which the application is written. The user still exercises control over AVS in the normal manner; essentially, it appears to the user as though the external computation connected using Schooner is running within AVS. Of course, extra features are typically added to implement the added user interaction that is the motivation for much of this work. For example, controls would need to be added to execute the external computation successively with new parameters, to halt the computation, or to modify parameters during execution. However, the net result is that the fundamental "feel" of AVS has been retained, minimal changes are needed for the application, and the user is able to take advantage of heterogeneous resources in a transparent manner.

The process of writing a scientific meta-computation using the Schooner/AVS platform is straightforward, especially in the common case where the computational phases already exist as stand-alone programs that send their output to one or more files. In these situations, the goal is to make as few modifications as possible, while adding the ability to view the intermediate and final results, and to modify one or more parameters during execution. The first step in constructing such an application is to change the computational phases into procedures. For the most part, this simply involves changing the main routine into a procedure and accepting as arguments appropriate control values. The procedure will also need to return those values the user wishes to display rather than (or perhaps, in addition to) writing them to a file. Generally, these returned values will be one or more arrays containing values calculated at the grid points in the problem. For problems involving irregular grids, both the coordinates and the values have to be returned. This feature of accepting input and sending output as parameters to the computation procedure often simplifies the application's code, as there is no need to devise routines to create a particular file format or to read a particular input format. While some work may be needed on the AVS end to convert the arrays of numbers into, for example, an AVS data field, this is often necessary when reading non-AVS formatted data files in any event.

The next step involves writing a new AVS module to generate the data values to be displayed by invoking the computation procedure directly. The structure of this new module can be broken down into three basic tasks: collecting input to use as argument values for the procedure, invoking the procedure to perform the computation, and making the results available on one or more output ports. All these tasks are usually straightforward and very similar to those required by any new AVS module. Input is done using the widget mechanisms described above if from the user, or from data ports if from other AVS modules. The invocation of the remote procedure is written using the language's standard procedure call mechanism, which is, of course, later transformed by the stub and Schooner libraries. Output is performed by passing along the results from the procedure invocation to subsequent modules that will process it for display purposes. The one possibly complicated part of this process occurs if the data being produced does not fit into one of the standard AVS data models; in this case, it would be necessary for the user-written module to transform the data, for example, by producing a geometry that can be rendered by AVS. Note, however, that such

1.	Modify computations to export one or more procedures.
2.	Design AVS module: a. Add widgets to allow user input of parameters, b. Call remote computation(s), c. Output results of computation to AVS dataflow network.
3.	Write UTS specification files for AVS module and each remote component.

Table 4-1: Steps in Creating Schooner/AVS Meta-computations

a transformation would have been necessary even if the data were coming, for instance, from a file rather than a remote computation.

The final step for the user is to write two UTS specification files, one for the procedure performing the computation and the other for the AVS module. These two sets of specifications, which are virtually identical, describe the structure and type of the arguments needed by the procedure. As explained in Chapter 3, these specification files are used by the Schooner stub compilers and for typechecking. Table 4-1 summarizes the steps in creating meta-computations using the Schooner/AVS platform.

Although the above are all that are required to execute an application using this scheme, it may be beneficial to consider restructuring the computation. In particular, if the execution time is long, it might be better to modify the computation to be a procedure that returns intermediate values. This will allow the user to monitor the progress of the computation and modify parameters during the computation. For example, there might be a need to establish new boundary values as the computation approaches a steady-state condition. In this model, the procedure will typically retain values between calls and will, on each call, proceed for some number of iterations before returning results to be displayed. Given this structure, the user must also decide if the AVS module containing the call should be a procedure module or a co-routine. The procedure choice is particularly appropriate when the user anticipates the need for frequent interaction with the computation. The co-routine choice is best when interaction is needed only occasionally and the main intent is to monitor the computation. When each computation is long and the visualization being performed by AVS is complex, the co-routine choice will also gain some concurrency since the next call to the remote procedure will be started while AVS is still working on rendering the results of the previous call.

Finally, it should be noted that AVS does provide limited support for executing procedure modules (not co-routines) on other machines in a local-area network. However, unlike our approach, each such remote module must be executed on a machine that supports AVS and must be written to conform with the standard data-flow AVS programming model rather than a general procedural model. Moreover, in the situation where two or more versions of a component are available, AVS requires separate modules in the network for each. Thus, the selection of a different

machine requires changing the network by deleting the previous module and adding the new one. Schooner's dynamic startup capability, in contrast, allows the user to select among remote components using a widget, as discussed in more detail below. This requires no change to the network and only one AVS module.

4.3 Molecular Dynamics

This program computes constant energy-volume-number particle dynamics for Lennard-Jones 12-6 interparticle forces using an algorithm based on [Allen82]. On each time step, it advances the positions of the particles, and then computes the forces among the particles, and the kinetic energy and thermodynamics of the system. The particles reside in a unit cube and their initial positions are uniformly distributed within the cube. The initial velocities are randomly determined about a mean of zero using a Gaussian distribution and a user-specified standard deviation. The original application produced data files that recorded the velocities and positions of the particles, and each iteration of the main loop would proceed for a specified number of time steps, then update the files. The original intent of the program was to produce an animation showing the positions of the particles in the system over time, with arrows indicating the direction and magnitude of the velocity of each particle.

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

4.3.1 Adapting the Application

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

each call.

The second step was to write the new AVS module. This module is designed in the normal way that AVS provides. AVS modules consist of two required functions, `spec` and `compute`, and two optional functions, `init` and `destroy`. `init`, when present, is called when the module is instantiated in an AVS network; `destroy` is called when the module is removed from the network. The `spec` function describes the widgets, sets up the input and output data ports, and identifies the names of the other functions. The `spec` and `destroy` functions are affected only a small degree by the use of Schooner. Specifically, `spec` sets up two additional widgets for Schooner: one to allow the user to choose the machine to run the remote computation, and one for specifying the path to the executable. The `destroy` function contains one Schooner library call to notify the Manager when the AVS module is terminated. The `init` function is not affected by the use of Schooner. Appendix A contains the complete source for the molecular dynamics AVS module. In the `dynamics_spec` function, the assignments to the variables `param9` and `param10` create the two Schooner widgets.

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

A final change in `compute` was the addition of an invocation to a library function to handle the Schooner initialization protocol properly. This is a change from the command-line procedure for starting meta-computations described in Chapter 3. The command-line approach assumes all the components and hosts to be used are known at startup time. When AVS is involved, this is not possible, since the user determines where the remote component is to be located after AVS has started and the AVS network has been created. The library function is a solution that allows the user to first select, using the AVS widget, the remote machine to be used. Specifically, this function performs two functions. First, it contacts the Schooner Server on the AVS machine to obtain the address of the Manager process. With this information, the component is then able to contact the Manager and proceed with the exchange of mapping information as outlined in Chapter 3. Second, the library routine takes care of the request to the Manager to start the remote component on the machine the user selected. The library routine is called only the first time the computation function is invoked. Appendix A shows this section at the beginning of the `dynamics_body` function. The code is placed inside a loop that employs the AVS alert mechanism to notify the user when an error has occurred during startup.

The final step in performing the integration was to write the UTS specification files. The export specification for the component containing `dynamic` and `generate` is shown in Figure 4-

```
export GenerateParticles prog(
   "num_particles" val integer,
   "vel_std_dev" val float,
   "positions" res array [300] of record
            record {"x" float, "y" float, "z" float},
   “velocities” res array [300] of
            record {"x" float, "y" float, "z" float})

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

Figure 4-3: Molecular Dynamics UTS Specification File

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

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

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

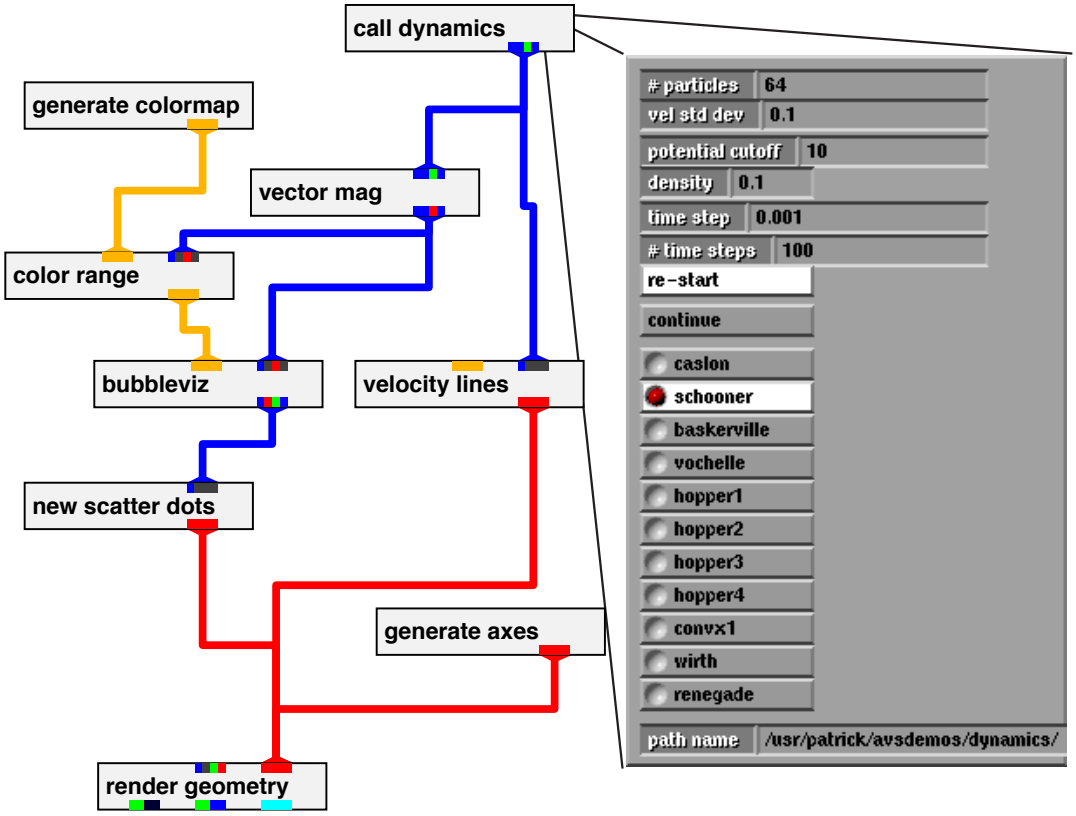


Figure 4-4: Molecular Dynamics Application: Network and Widgets

Actually executing the application is straightforward. First, the Schooner Manager process is started. Then, AVS is started and the desired network is created using the standard Network Editor, or a previously saved network is read in. Figure 4-4 shows the network used in testing the dynamics application and the widgets for the AVS call dynamics module. When the call dynamics module is instantiated, the initialization routine is run and the various widgets are displayed. The user can then specify the machine where the computation is to be run, the pathname leading to the executable on that machine, and provide values for the other parameter widgets. When the module is first executed by AVS, after the user clicks on the *continue* widget, the Manager is contacted and mapping information is exchanged. During this first run, the Manager also starts the remote component on the requested machine. From this point on, the module works like any other AVS module: when a change is made to a parameter, the module is called and Schooner transparently handles the communication with the remote procedure. A screen snapshot taken during the execution of the application is shown in Appendix A; the widgets are on the left side of the screen, the AVS network in the middle, and the graphical output on the right.

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

Table 4-2: Molecular Dynamics Application Timings

4.3.2 Experiments

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

To test the performance, a series of tests were conducted with AVS running on a Sun Sparc 2 and the dynamics portion of the application running on a Convex C220. This is the same configuration used for the second set of performance measurements reported in Chapter 3, with both machines being in the same building, but on different networks. The results are shown in Table 4-2. The problem size is the number of particles. The times shown are in seconds and reflect the elapsed time on each machine for one call. The call consisted of a request to execute 100 iterations with a time step of 0.001 seconds per iteration. The difference column is the proportion of the total elapsed time not spent actually executing the computation, and hence attributable (mostly) to communication overhead. Note that, given the use of fixed size arrays, this communication cost is the sum of a constant encoding/decoding time plus a (slightly) variable network delay between the machines. The computation time, on the other hand, is dependent on an $O(n^2)$ algorithm and increases accordingly. Thus, the communication overhead as a percentage of the total execution cost rapidly decreases as the problem size grows, becoming quite small for moderate to large problem sizes. For small problems, this implies that it would be more productive to execute the dynamics portion of the application on a local machine, perhaps even the same machine running AVS, to minimize the communications cost. As larger problem sizes are run, however, it becomes

advantageous to use the faster processor, as the communication costs become a very small fraction of the execution time.

4.4 Neural Net

The application is a neural net code that implements a short-cut version of the Kohonen self-organizing neural network using Gaussian type interconnection strengths and a conjugate-gradient learning algorithm. The Gaussian function uses a periodic norm to measure distance. Points representing processing elements fall within a two-dimensional unit square and are reflected back inside the square if the calculation tries to place them outside. The inputs to the program include the number of elements in the net, the amount of randomness in the initial configuration, the half-width of the Gaussian function (which affects the learning curve of the network), and the desired number of learning iterations.

The processing elements in the neural net use information about their neighbors in a learning process to determine their position within a grid. Once the program has reached convergence, a plot with the points connected to each other should show uniformly spaced vertical and horizontal lines; a similar plot done at the start of a computation will show random lines zigzagging across the space. Plots made during the run will give a feel for how the problem is approaching convergence. A substantial number of iterations are typically required to come close to the final uniform-spacing solution for nets with a high initial degree of randomness; for example, in one run involving 32 x 32 points, 210,000 iterations were required to get a reasonably good solution.

4.4.1 Adapting the Application

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

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

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

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

Figure 4-5: Neural Net UTS Specification File

initiate the next remote call while the other modules in the network are still rendering the image from the previous call. A co-routine is also self-executing, a more suitable solution in a situation where the user primarily wants to monitor the computation rather than changing parameters often.

The other important difference concerned the use of variable size arrays. Unlike the dynamics application, in this case the problem requirements mandated that no predefined limit be set for the size of the neural net. Figure 4-5 illustrates how this is handled in Schooner in the UTS specification file. Here, `num_points` specifies the size of the neural net, which will be (`num_points` x `num_points`), `side` is a random factor for the initial distortion of the points, `alpha` is the half-width of the Gaussian function, `cur_count` is the current iteration count, and `increment` is the number of iterations to proceed on the current call. `rep_x` and `rep_y` contain the x and y coordinates of the points and are the result of calling the procedure. Figure 4-6 shows the AVS network for the neural net and, on the right, the widgets for the neural module.

The key to allowing variable-sized arrays is the specification of `rep_x` and `rep_y` as `rep` arrays, indicating that these parameters are represented types as described in Chapter 3. While allowing the size to be established at run-time, this choice does require some additional work on the part of the programmer since UTS library routines must now be used to encode explicitly the array values in `compute_net` and subsequently decode them in the AVS module. The actual code used to encode the values for the `rep_x` array is shown in Figure 4-7. The code to perform the other encodes and decodes is analogous. If the programmer prefers not to do this explicit conversion, fixed-size arrays could have been used for x and y as was done in the molecular dynamics example. Even in this case, it is still possible to have the problem size be a parameter, but the chosen array size would now be an upper-bound on the problem size. This option also incurs some performance penalty since the entire array is transmitted during the RPC even if just a portion is actually being used.

Several versions of the code have been developed, each of which is tuned to the particular

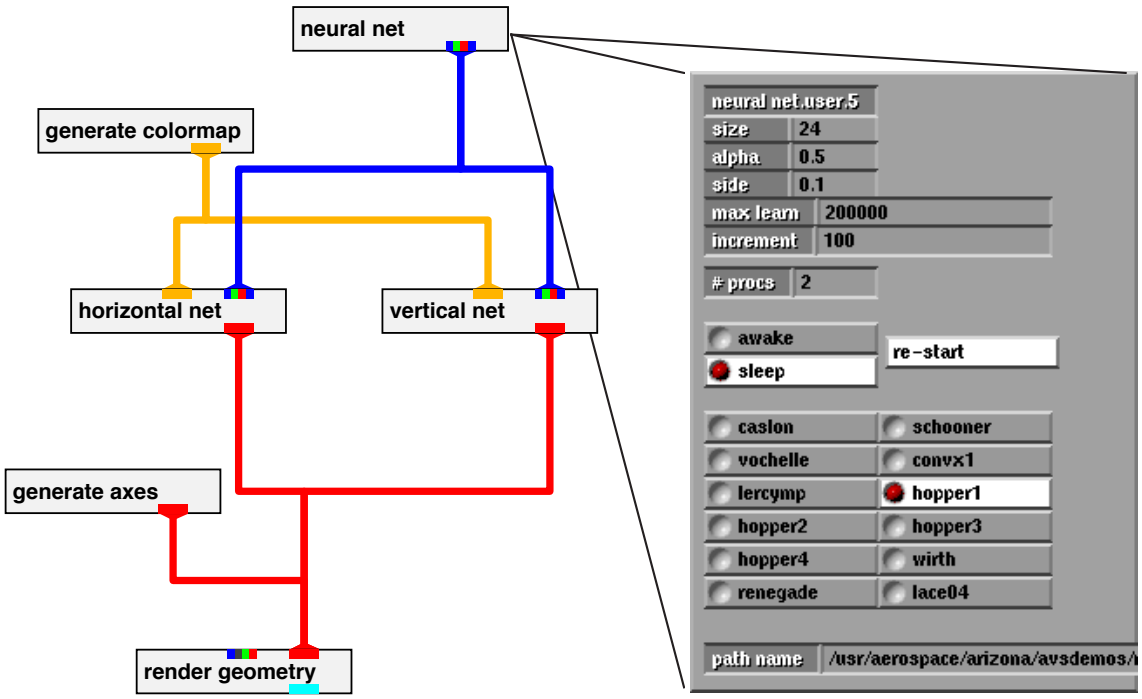


Figure 4-6: Neural Net Application: Network and Widgets

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

Figure 4-7: Encoding a Variable-Sized Array

architecture on which it executes. For example, the version that runs on a Sequent Symmetry is a parallel solution that exploits the shared-memory, multiprocessor architecture of the machine. This version uses a parallel algorithm that divides the elements of the neural net into horizontal bands and allocates each band to a processor. The code is implemented using the parallel library provided by Sequent to synchronize the processes for the necessary exchange of boundary values on each iteration and to allow the use of shared arrays to hold the results. Connecting this version of the neural net application to AVS was no different than connecting the sequential algorithms except for the addition of one more parameter to allow the user to specify the number of proces-

sors to be used in the calculation. This is the `num_procs` parameter from `configure_net` and can be changed whenever a restart of the computation is requested. Thus, the flow of control is single threaded from the AVS co-routine until the remote procedure call arrives at the Sequent. Then, the flow divides among the processors for the requested number of iterations. Once those are complete, the flow becomes single-threaded again for the reply back to the AVS co-routine.

Recently, the neural net code has been ported to run on an Intel Paragon using Intel's message library, and on a cluster of Sun workstations using the PVM message-passing system [Sunderam90, Beguelin91]. Both of these versions employ the same parallel algorithm as the Sequent version, but make use of messages to exchange boundary values between the processes. Multiple implementations provide greater flexibility when running experiments. The user can select a platform based on factors such as machine load, and the dynamic startup feature allows this choice to be deferred until runtime.

4.4.2 Experiments

This application has been run successfully on a number of machine combinations. These tests have included combinations with AVS running on Stardent and SGI machines at Los Alamos, and machines at NASA Lewis, including Stardent, SGI, and Sun Sparc 2. The computational component was run either on a local machine, including a Convex C220, Sun Sparc 2, and SGI, or on remote machines at The University of Arizona, including a Sequent Symmetry, Sun Sparc 2 and Convex C240. The Paragon and PVM versions have been tested over local networks connecting to a Sun Sparc 10 running AVS.

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

To measure the impact of monitoring the application, the tests were run with two versions. In one, a single call was made for the 1,000 iterations; in the second, a series of 10 calls for 100 iterations each were made. The results are shown in Table 4-3, where the times were measured using `gettimeofday`. The values in the Sequent column represent the time actually spent executing `compute_net`, while the two Sun times correspond to the elapsed time for the one call and ten call versions, respectively. The last column is the percentage difference for the Sun times between the one call and the ten call version; this indicates the performance penalty imposed by monitoring the computation more often. As can be seen, this penalty is small even when the monitoring frequency is fairly high. In the example mentioned earlier where 210,000 iterations were used,

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

Table 4-3: Neural Net Application Timings

monitoring every 100 iterations would equate to 2,100 calls for an estimated 5.9% slower execution.

4.5 Summary

This chapter has described the techniques used in building meta-computations using a Schooner/AVS platform. In this combination, Schooner provides the capability to exploit remote and varied machine resources, including parallel platforms. AVS provides an interaction and visualization tool that can be used to monitor and control the computation. Adapting applications to take advantage of this environment is a straightforward process, typically involving only a few changes to the AVS module and the remote computation.

Two applications built to exploit this platform were presented. The molecular dynamics example shows the ability of the user to interact with the computation in determining the optimum settings for a set of parameters. The neural net example illustrates both the ability to monitor a long-running computation and the ease of connecting to a variety of parallel machines.

Chapter 5
EXTENDING THE MODEL

In addition to the static model described in the previous two chapters, Schooner supports a degree of concurrency in metacomputations, as well as dynamic configuration capabilities. Both were added as a direct result of the needs of scientific computing. One example of the need for concurrency comes from the long-running nature of some scientific calculations. This creates situations where a scientist will want multiple computations executing simultaneously, in much the same way a biologist will have overlapping experiments, awaiting the completion of generational cycles in each. Another example comes from AVS, where the dataflow execution model implies a need for both concurrency and dynamic configuration capabilities. In particular, the network may have many execution paths active, which leads to concurrent execution. The user may also need to add and delete modules during execution, which requires configuration support.

This chapter describes the extended Schooner model by focusing on its support for these two features. Each section discusses the motivation and possible solutions, then describes the solution adopted by Schooner. The final section presents implementation details.

5.1 Concurrency

5.1.1 Issues

Concurrency occurs in scientific computations at four levels: fine-grain, medium-grain, and coarse-grain parallelism, and the concurrent execution of multiple experiments. Both fine- and medium-grain parallelism use multiple processors to solve a problem with the specific goal of shortening execution time. Examples are the use of PVM on a cluster of workstations, and the use of a native library on an Intel Paragon. From the perspective of an interconnection system, computations using each of these techniques is encapsulated within a component as described in Chapter 3 and does not require direct support.

Coarse-grain parallelism and concurrent execution of multiple experiments are areas requiring explicit support by an interconnection system. Coarse-grain parallelism within a meta-computation can take two forms: pipelining the execution of different components, and using parallel threads of control for different parts of a simulation. One simple example of pipelining is a visualization and calculation component, where the rendering of a geometry in the visualization component can be pipelined with the calculation of the next round of data. An example of parallel threads of control arises in jet engine simulation, where each logical unit of the engine may be simulated by a single computation component. With such an organization, the thread of control among the components typically follows the air flow through the engine. As a result, in a two stream engine, the air flow separates past the fan, implying that subsequent components in each stream can

execute in parallel.

Executing multiple simultaneous meta-computations becomes advantageous when each requires long runs to complete. For example, a scientist may run multiple experiments, varying the parameters in each. The neural net application in Chapter 4 provides an example of this possibility. With versions of the program available for a variety of heterogeneous platforms, experiments involving different degrees of randomness in the starting arrangement of nodes could be run simultaneously. In this scenario, each experiment could take advantage of the special characteristics of a given platform to test different aspects of the compute component.

5.1.2 Possible Approaches

The designer of an interconnection system can choose from a range of possible concurrency solutions to assist the scientific user. Two options mark the opposite ends of the spectrum. One approach is to provide no explicit support for concurrency, essentially the option available to the user under the basic Schooner model. In this model, the user can explicitly start one instance of Schooner for each concurrent meta-computation. However, this leaves the user responsible for keeping the various meta-computations untangled. It is also awkward and wasteful since multiple Managers and Servers must be running on each participating machine. A more serious problem arises when concurrency is needed for coarse-grained parallelism within a meta-computation. Since this approach treats the meta-computations as distinct and single-threaded, it becomes difficult to add a thread to a meta-computation to solve the problem in parallel.

At the other end of the spectrum is a general concurrency model. This is the type of model presented by the message-passing systems described in Chapter 2, which provides the ability to create and destroy threads within a meta-computation. A general concurrency model has two drawbacks. First, the user must change programming models from a simple procedural model to the more complicated message-passing model. This shift is complicated by the loss of the implicit synchronization implied in the procedural model, and the number of changes needed to adapt the source code. The other drawback is that a general concurrency model requires a more complex implementation. For example, the implementation must provide the user with explicit synchronization primitives and a greater variety of messaging primitives, as well as a more complex runtime to support arbitrary message routing.

An approach between these two extremes offers several advantages for an interconnection system. It avoids the complexity of the fully general solution for both the user and the implementation, yet offers more flexibility than the single thread model. It also allows retention of the procedure interface and its implicit synchronization, which simplifies the job of creating components for a meta-computation.

5.1.3 Lines

The concurrency model supported in Schooner is an example of an intermediate approach. The model uses *lines*, each equivalent to the notion of a Schooner program described in Chapter 3. That is, a line is a sequential execution of procedures, some of which may be located on remote machines and/or written in various programming languages. Within each line, there is a single sequential thread of control and the implicit synchronization of procedures calls is still present. Concurrency comes from the ability to have many lines executing simultaneously. Each line executes independently of the others with no implicit cross-line synchronization. This allows the user to run multiple meta-computations simultaneously, or to have multiple lines within a meta-computation under the control of a single Manager process. New lines can be started and previous lines terminated, without affecting other lines.

The addition of lines opens a number of opportunities without unduly complicating either the user's task or Schooner's implementation. When multiple lines are active, each can employ any number of components executing on intersecting sets of machines. The retention of procedure call semantics within a line allows the user to continue to take advantage of a familiar mechanism and the implicit synchronization it supports. Multiple instances of the same line can be executed to solve distinct parts of the same problem in parallel, or to repeat the same experiment with different parameters. A component can also be replicated in multiple lines, allowing the use of its procedures in different calculations. Explicit cross-line synchronization can be made available through *shared components*, which provide exported procedures that are available to all lines.

The user composes lines from components using a number of methods. One is analogous to the approach used in the basic Schooner model, where a meta-computation is specified on the Manager's command line. Components included in the list are all placed in the same line. The other two methods are implicit and explicit assignment of components to lines. In the implicit method, the Manager assigns components to lines with no input required on the part of the programmer or the user. This happens in two situations. The first is when a component is started at the request of another component using the `sch_start_component` routine. In this case, the new component is implicitly bound into the same line as the requesting component. The second is when an independently started component contacts the Manager, such as described in connection with AVS in Section 4.3.1. In this case, the Manager starts a new line, with the requesting component being its initial member.

The explicit method allows the user or the programmer to control the binding of components into lines. There are two ways this is made available. One involves library routines that allow the programmer to access the Manager's dynamic configuration capabilities. These routines are discussed in more detail in Section 5.2.3. The other technique involves the Controller, an X-

---

Line is the nautical term for rope.

windows interface discussed in Section 5.2.5. With the Controller, the user can explicitly start components as part of a designated line, or specify a line for an independently started component.

Errors are handled differently with the addition of lines to Schooner. In the basic model, an error would result in termination of all the components and the Manager. Now, an error results in termination of only the line containing the component that generated the error. The Manager reports the error, shuts down the components in the affected line, then continues with operations associated with other lines. The Manager now terminates only when specifically requested to do so by the user. Stopping the Manager results in termination of all active lines and their components.

5.2 Dynamic Configuration

5.2.1 Issues

Dynamic configuration is the ability to modify the binding of components to lines and the mapping of components to hosts after a meta-computation begins execution. Thus, the meta-computation does not have to be fully defined at the beginning of execution, but rather can be modified during execution to adapt to the needs of the calculations or the availability of hosts. To be fully useful, such changes should be possible from within the meta-computation itself, as well as by direct intervention of the user during a run. This allows maximum flexibility in adapting the meta-computation to conditions that vary between different execution runs [Homer94c].

5.2.2 Possible Approaches

As with concurrency, one option is to not provide support for dynamic configuration. This has several drawbacks, but chief among them is inflexibility. The user is constrained to identify all the resources that *might* be needed during a computation and start all the components and lines at the beginning of the run. Factors that arise after execution has begun, such as machine availability, can prematurely halt the experiment.

The opposite extreme is different for the two aspects of configuration. For binding components into a meta-computation, the opposite extreme occurs in a general concurrency model, such as that provided by the message-passing systems described earlier. These systems can support a component joining an execution in progress and participating in the execution, for example, by sending and receiving messages. The drawbacks of a general concurrency model as an interconnection system have been described in Section 5.1.2. Of particular note in this context is the need for the programmer to explicitly manage the synchronization in a general concurrency model.

For mapping components to hosts, the opposite extreme is *process migration*, the ability to move a process from one machine to another at any time. This requires saving the complete state of the process, transferring the state to the new host, instantiating the process, and restoring its state. Several projects have addressed process migration. One is described in [Douglis91], which

summarizes the issues involved in process migration and describes an implementation of this feature in the Sprite distributed operating system [Ousterhout88]. This implementation migrates the process state and associated execution environment, including, for example, open file pointers and the contents of virtual memory pages. The system works well, but requires considerable support from the Sprite kernel and does not support cross-architecture migration.

Another project, Surgeon, is studying the problem of process migration in the context of heterogeneous distributed applications [Hofmeister93]. Surgeon is layered on top of the Polylith/Polygen system that was described in Chapter 2. The project has demonstrated a limited ability to migrate processes under certain constraints, such as the use of atomic actions to send and receive messages. The current work includes defining three classes of modules that characterize when migration is possible. The first class contains modules that require no change in the user's source code. These are processes that do not have state variables that need to be restored when moved to a new location, and are not in a state that requires synchronization with another process, such as awaiting a reply to a remote procedure call. The second class contains processes that can be migrated using automatically generated modifications to the user's source. The third class contains those processes that require the programmer to make explicit changes. In conjunction with the work on defining the three classes, Surgeon is also studying techniques for automatically determining the class of a module.

An approach between these extremes is available to Schooner given its procedure call orientation. When dynamically binding components into a line, a library routine to request the start of a new component can take advantage of the implicit synchronization of a procedure call to block execution of the line until the new component is ready for use. Similarly, a component that does not retain values across calls can be moved when between calls.

The specific mechanisms included in Schooner support dynamic configuration in three ways:

- Dynamic integration of components into lines through improved support for startup and shutdown during execution,
- Limited support for component movement during a computation, and
- An X-windows Controller to provide the user with interactive control over meta-computations.

The first two methods are implemented by library routines added to the Schooner runtime system, and are invoked from within components. The Controller provides the user with direct access to the Schooner configuration library, and with testing and tracing facilities.

5.2.3 Dynamic Integration

Component requests to start a new component or to join a line are supported in the extended Schooner model through additions to the Manager's configuration abilities. Components can start at any point during a meta-computation, and computations that make use of more than one line

`sch_contact_schx()`
Register independently started component with the Manager.

`sch_start_component(remote_host, remote_name, remote_path)`
Start a new component, where `remote_host` will be the site of the new component, and `remote_name` and `remote_path` specify the executable and its location on `remote_host`.

`int request_line_id()`
Allocate a new line and return its id.

```
sch_start_component_in_line(line_id, remote_host, remote_name,
                            remote_path)
```
Start component and bind into line `line_id`.

`sch_i_quit()`
Terminate component.

Figure 5-1: Library Routines Supporting Dynamic Integration

have the ability to start and stop a line without affecting other lines. The interface is provided to the programmer as a set of library routines (Figure 5-1). Each is described in more detail in what follows.

The registration call, `sch_contact_schx`, allows a component to be started independently and then contact the Manager to join a meta-computation. When working with tools such as AVS that have their own startup protocols, this allows a component to be a part of both the tool and Schooner. Essentially, this library routine is an interface to the registration call described in Section 3.3.4 that each component normally makes to the Manager upon startup, but with two differences. One difference is the need to contact the local Server first to learn the location of the Manager. When the Manager begins execution, it sends its location to the Server, which then caches this information to pass on to independent components. The second difference is the lack of a line number. When the Manager initiates a component, whether through the command line list of components or the dynamic mechanisms described below, it is passed a line number by the Server. For the independent startup case, however, the Manager assigns a line number to the component in the reply to the registration call. As described above, the default is to assign a new line identifier for independently started components. If the Controller is active, however, the Manager can query the user about the line assignment for the new component, and, based on the response, either bind it into an existing line or create a new line for it.

The `sch_start_component` library routine allows a component to request the initiation of other remote components. Components started in this manner are considered part of the requesting component's line and can be called only by other components in that line. The Manager receives the start request, forwarding it to the Server on the indicated machine for startup. Once the newly

```
sch_move_component(old_comp_name, old_comp_machine,
                   new_comp_name, new_comp_path, new_comp_machine)
```

Move `old_comp_name` from `old_comp_machine` to `new_comp_machine`. `new_comp_path` and `new_comp_name` specify the executable to use on the new machine.

Figure 5-2: Component Movement Library Routine

started component has completed its registration call and been bound into the line of the requestor, the Manager replies to the start request, which allows execution of the line to proceed. The `sch_start_component_in_line` library routine is similar, but specifies a different line for the component. The associated `request_line_id` routine requests allocation of a new line identifier by the Manager.

Finally, the semantics of shutdown have changed in the extended model, both for normal shutdown and when errors occur. The situation for errors was discussed above in Section 5.1.3. The case for normal shutdown is similar. When a line reaches normal termination, such as occurs at the end of the main routine, or when a component executes the `sch_i_quit` library call, the Manager will terminate all the components in the line. Thus, when a meta-computation involves multiple lines, the termination of one line does not affect the other lines. For example, within AVS, the user is able to remove a module from a dataflow network without affecting other lines being managed by Schooner.

5.2.4 Component Movement

Re-configuration of lines is also supported through a limited ability to move components from one machine to another during execution. The long compute times that characterize many scientific computations make the ability to move components among machines useful. In particular, components can be moved to avoid scheduled downtimes, thereby allowing the computation to continue without interruption. It is also useful to perform load balancing. The ability to move components in Schooner is less ambitious than process migration in keeping with the philosophy of minimizing the changes needed in the user's source code. The additions needed to the Schooner runtime are also considerably less complex than would be required for process migration.

A move is carried out by the `sch_move_component` library routine, shown in Figure 5-2. The Manager first sends a shutdown message to the original component, deletes its procedures from the procedure name mapping tables, and starts a new copy on the specified machine. After the registration call from the new component is received, the Manager updates the procedure name mapping information for the line and replies to the move call. Execution of the line can now proceed. Procedure name caches within each component in the line are updated when the next call to a moved procedure is attempted. A procedure call to the old location fails, resulting in an automatic call to the Manager for the new location.

This scheme takes advantage of Schooner's use of RPC, allowing a component to move when it is between calls without requiring changes to the user's source code. To maintain the correct semantics, however, the procedures in the component must be stateless; that is, the component cannot contain variables that retain values across calls or be in the middle of a recursive chain of calls. One planned addition to Schooner will remove the restriction on state variables by using an extension to the UTS specification language. The user will then be able to describe a list of global state variables whose values are to be transferred when the component is moved. As part of the move process, then, the original component will send the current values of these variables in a message to the Manager before shutting down. The Manager will then start the new component, which will make a call-back to the Manager to obtain the current values.

5.2.5 Controller

The Schooner X-windows Controller is a graphical user interface (GUI) for monitoring and dynamically configuring meta-computations. The Controller satisfies this role by providing the ability to:

- Create components on specified hosts,
- Assign components to lines,
- Execute a line or individual procedures, and
- Trace remote procedure calls and replies during execution.

The Controller implements this functionality by providing, in essence, interactive access to Schooner's dynamic integration and component movement libraries.

The first two abilities allow the user to create meta-computations using the Controller. The user starts a component by specifying the executable and the host, either in a type-in window or by selecting the appropriate name from a menu. To implement this functionality, the Controller invokes the manager's start component function and reports any error messages from the Manager, such as a bad host name or file not found, allowing the user to try again.

As already noted, the Controller also allows the user to control line creation and component binding. In this case, the Controller receives the relevant information from the Manager when the component registers, and then queries the user for the line assignment before binding the component.

The Controller supports two modes of execution. In the first, the main procedure of the line is invoked, with command line arguments being passed from the user via the Controller and Manager. Components can subsequently be added to the line by the user as execution proceeds. For example, a name resolution request made to the Manager looking for a procedure not in the database will prompt a warning message to the Controller. At this point, the user can start a component containing the needed procedure, bind it into the line, and then allow execution to

proceed.

In the second execution mode, the Controller is used to call any exported procedure, which allows the testing of the individual components of a meta-computation. The UTS export specifications for the procedures allow the Controller to prompt the user for values to assign to each of the parameters. The Controller then calls the procedure and receives the reply, again using the UTS specifications to display the results for the user.

Finally, the Controller is able to trace the remote procedure calls. To do this, each component forwards copies of its calls and replies to the Controller, which displays them to the user. The UTS specification and tagged presentation are again useful in this context to decode and display the arguments and return values found in these messages.

The development of the Controller is an on-going project, and not all the features described here have been implemented. At this time, the Controller is able to monitor lines that are implicitly started, create new lines, initiate components, and execute a line's main routine. The Manager can trace calls and replies, but the ability to pass the them to the Controller is still being completed. The Controller's abilities to move components and to execute individual procedures are likewise still under development.

Figure 5-3 shows the Controller main window during execution of multiple neural net and molecular dynamics applications described in Chapter 4. At the top of this window, the File and Control menus support global commands such as creation of new lines and quitting. The Command box allows the user to enter command line arguments, pathnames, etc. Below the command box, each line is represented by a horizontal box. Within each line's box, the left portion contains the line id, trace toggle, and Operate button. The remainder of the box contains a button for each component in the line. The trace toggle turns procedure call tracing on and off for the line. The Operate button pops up a menu that contains commands specific to the line, such as starting a component, terminating the line, and moving a component. Selecting a component button brings up a Component Info window. Figure 5-3 shows the Component Info window for the net component, the remote part of the neural net application, from line 5.

5.3 Implementation Notes

Adding concurrency and dynamic configuration to Schooner involved changes to the Schooner runtime system and the addition of the Controller. Most of the changes to the runtime system involved the Manager; the components themselves did not change significantly, largely because there remains only a single thread of control within each line. These changes are described in Section 5.3.1.

The Controller was designed to be a component to simplify its implementation. However, the Controller does differ from normal component in one significant way — the need to respond to user-generated events. This necessitated two additions to Schooner, both of which are described in Section 5.3.2

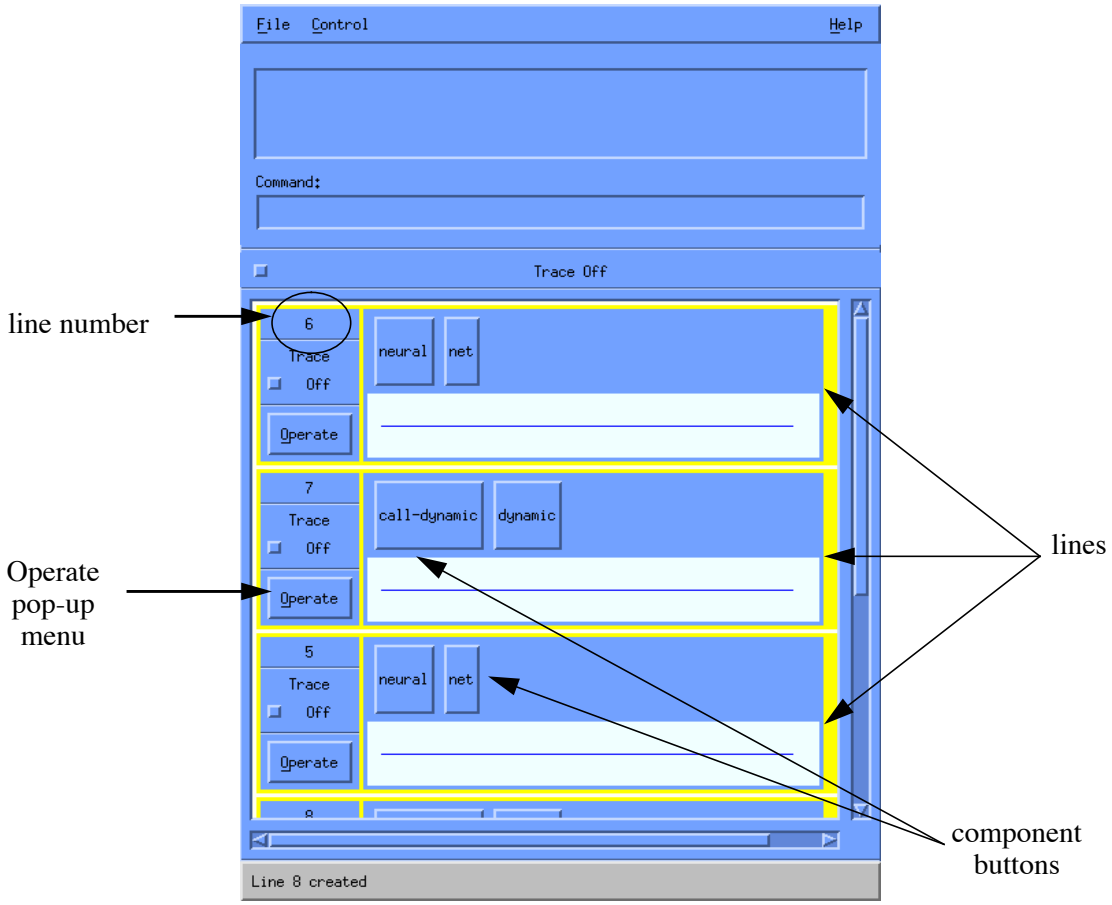


Figure 5-3: Controller: Main Window

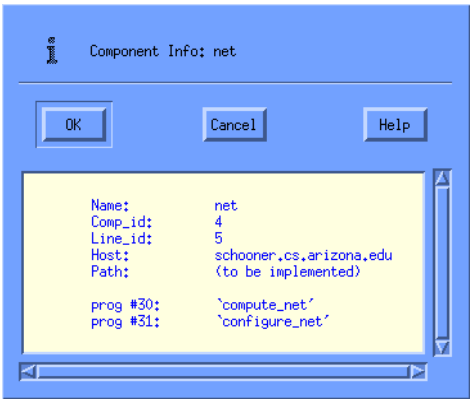


Figure 5-4: Controller: Component Info Window

5.3.1 Changes to the Manager

The changes to the Manager were primarily to support the concurrency enhancements in the extended model. These services are provided to the components in the form of exported procedures that the components can invoke. From the component's perspective, then, each such invocation is a simple procedure call that will blocks until the service is performed. At the Manager, however, some calls result in delays before the reply is generated. Specifically, it may be necessary for the Manager to request service from the Controller, a Server, or another component before replying to the call. This makes it impossible for the implementation of the Manager to rely on the internal call stack as was done previously to match calls and replies; that is, when a reply is ready, it may be from a line other than the line associated with the most recent call. Hence, the structure of the Manager must be changed significantly over that used to support the basic model.

As an example of the type of scenario that cannot be supported by the original Manager, consider a call from a component requesting the startup of another component in the same line. To do this, the requesting component makes use of the `start_component` procedure exported by the Manager, with the newly started component invoking the `register_component` procedure. The UTS specifications for these two procedures are shown in Figure 5-5. `start_component` provides the information the Manager needs to compose the start request, which is then sent to the Server on the specified host. Similarly, the arguments for `register_component` contain the specifications for the procedures exported by the component and the component's port information.

The process of starting a component begins when `start_component` is invoked. At this point, the Manager sends a start order to the Server on the designated machine, but does not respond to the requesting component until after the registration call from the new component has been processed. This delay ensures that the component requesting the start will not attempt to call a procedure in the new component until after the Manager has added the procedure to the mapping tables. Under the basic model, such a concurrent call was not a problem since the Manager could be considered part of the single thread of control in the meta-computation. Under the extended model, however, this is no longer the case, since calls from other lines or the Controller may arrive.

One possible solution to implementing this functionality is to make the Manager multi-threaded. In this scheme, the Manager would create one thread per line, which would allow replies associated with a given line to be implicitly matched with calls. This idea was not used for three reasons. One is the need to keep the Schooner code portable across a variety of platforms. While thread packages are available on many machines, the specific primitives and semantics can vary significantly. The second reason is to avoid complicating the structure of the Controller. With a multi-threaded Manager, the Controller would potentially have to deal with concurrent calls from Manager threads, thereby requiring that it also be multi-threaded. Finally, there is no compelling

```
export start_component prog(
      "host_name"        val string[-],
      "comp_name"        val string[-],
      "server_flag"      val integer,
      "sch_timeout"      val integer,
      "line_id"          val integer,
      "started_by_line"  val integer)
      returns ("result" integer)

export register_component prog(
      "component_rep" val rep record {
            "c_name" string[-],
            "c_port" record "Stream" {
                  "stream_type" string[-],
                  "socket_addr" byte[-],
                  "socket_port" integer},
            "shut_down"       integer,
            "line_id"         integer,
            "comp_id"         integer,
            "started_by_line" integer,
            "started_by_comp" integer,
            "c_progs"    array[-] of record "prog" {
                  "p_name"  string[-],
                  "prog_id" integer,
                  "p_type"  signature}
            "remote_host" string[-],
      "component_id" res integer)
      returns ("line_id" integer)
```

Figure 5-5: Manager's Exported Start and Registration Procedures

performance reason for creating a multi-threaded Manager in this application domain. The bulk of the interaction between a component and the Manager occurs at component startup, which is a relatively short period compared to the lifetime of a typical scientific application. As a result, there is minimal contention for access to the Manager, and hence, no need for separate threads to deal with each incoming call.

Instead of multi-threading, the solution adopted is to have the Manager maintain an explicit call stack for each line. In the basic model, each call was decoded in the stub and the exported procedure called with the appropriate arguments. When the Manager finished processing the call, control returned to the stub where the result parameters were encoded and the reply message sent out. This is illustrated in Figure 5-6.

The extended Schooner model makes two changes in this picture. First, the stub for each of the Manager's exported procedures is divided into separate decode and encode stubs. The decode stub decodes the arguments and calls the Manager's procedure, while the encode stub encodes the result parameters and sends the reply. Splitting the stub in this way is necessary so that information

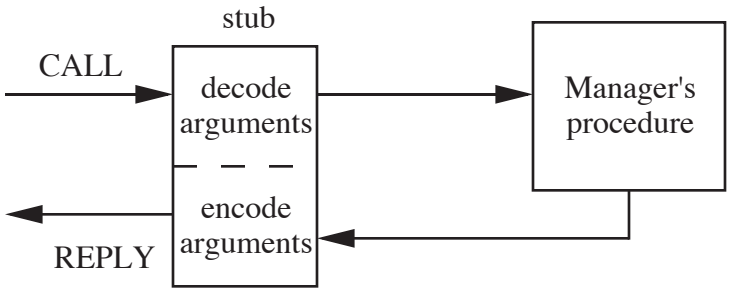


Figure 5-6: Basic Model: Manager Control Flow

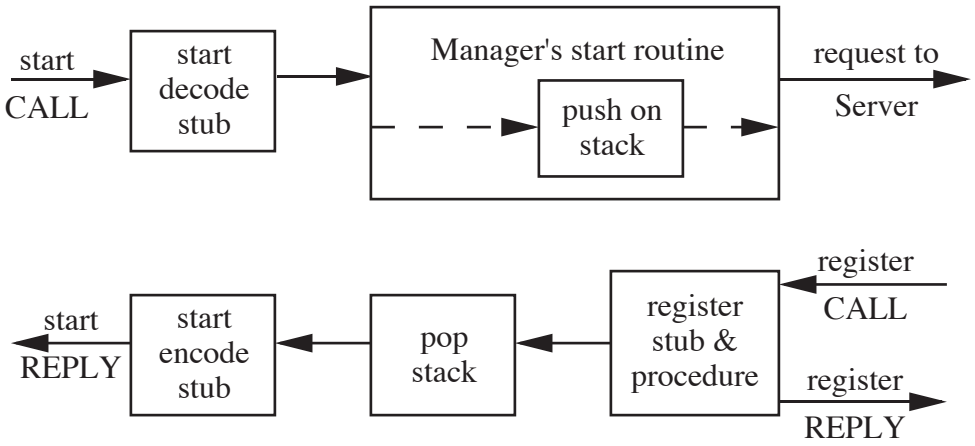


Figure 5-7: Handling Start Component Requests

can be placed on an explicit call stack after the decode stub, and removed before the encode stub.

The second change occurs in those procedures where the Manager must make a remote procedure call to request an outside service before replying to the original call. For example, when procedure call tracing is enabled, the component calls an exported Manager procedure with the tracing information, which is then passed along to the Controller. To handle this situation, the Manager implements, in essence, a procedure call stack for each active line. To make an outside call, the Manager first pushes information onto the call stack for the line, which allows it to make the reply at a later time. This information includes the type for the original call (e.g., a register component call) and the calling component's port information. After the outside call has completed, the Manager pops the call stack for the line, and uses the identifier to determine the encode stub to invoke. The port information is then passed to the encode stub, thereby ensuring that the reply message is sent to the correct component.

To illustrate how this works in a specific case, consider the receipt of a request to start a component from line X. Figure 5-7 illustrates the Manager's control flow for such a request. The

decode stub first receives the call message and then decodes the parameters, passing them to the Manager's start procedure. Here the Manager readies the start request to be sent to the Server and pushes the call type (a start in this case) and the component's port information onto a call stack associated with line X. Once the request has been transmitted, further processing on this call waits until the new component begins execution and registers with the Manager. Accordingly, control returns from the start decode stub and the Manager blocks waiting for either the registration call from the new component or a new call from another line.

At some later time, the registration call from the new component is received. At this point the Manager accepts the invocation and performs the work necessary to register the component. The Manager is typically able to complete the registration with no remote calls, so no use is made of the call stack in most cases. Hence, the Manager can reply immediately to the new component. Once this has been done, the Manager checks the call stack for the line and pops the information needed to reply to the original start component invocation. The encode stub is invoked to encode the reply arguments and send the reply to the requesting component. The requesting component (and line) can now continue execution.

When the Controller is active, the procedure just described is complicated by the need to inform the Controller of the new component. To keep the Controller's information current, it is notified after the registration call from the new component has been received, but before the execution of the line resumes. To accomplish this, the register procedure also has separate encode and decode stubs. The Manager makes a procedure call to the Controller, then pushes the new component's information onto the line's call stack to await the Controller's reply. At this point, there are two items on the line's stack: the register call on top, and the start call underneath. Once the Controller has replied, the Manager pops the stack and invokes the register encode stub, then pops the stack a second time and invokes the start decode stub.

A final change to the Manager involves dealing with name resolution in multiple lines. Names are considered local to a line, so separate procedure name mapping tables are used to perform name lookups for each line. The Manager itself is considered part of a special system line also occupied by the Controller. When a call is received, the table corresponding to the Manager is first checked to determine if the requested procedure is a system routine. If this lookup fails, the mapping table for the line from which the request came is checked. Should this also fail, a line maintained for shared components is checked. Subsequent actions, such as type-checking and possible error reporting, are identical to that described in Chapter 3.

5.3.2 Controller Implementation

As noted above, the Controller is implemented as a component and executes as part of the special system line. When the user uses the visual interface to request, for example, the start of a component, the Controller invokes a procedure that passes this request to the Manager, with the host, executable, pathname, and line identifier for the new component as arguments. The Manager

replies after the new component has registered.

The Controller differs from the typical component, however, in one important way. Normally, a line has only one thread of control; thus, a component is either executing, or passively waiting for a call or reply. The Controller, however, needs its own thread of control to allow it to respond to user-generated events; it cannot be idle when awaiting calls or replies from the Manager. This is a problem not limited to the Controller, since any event-driven interface that is part of a meta-computation would face the same dilemma.

The solution adopted is in two parts:

- A *dispatch* primitive to start the Controller, and
- A method that allows the Controller to treat procedure calls as events.

5.3.2.1 Dispatch

The dispatch primitive is used in Schooner to create a new thread of control within a line. To start the Controller, then, the Manager dispatches the Controller's main routine. The implementation of a dispatch is essentially the first half of a procedure call. Arguments can be passed to the dispatched routine and are encoded normally by the stub. However, once the dispatch has been sent and received by the component containing the dispatched routine, the caller continues execution. This is similar to the asynchronous RPC systems described in Chapter 3, but without the provisions for deferred delivery and receipt of the reply. Dispatch is also similar to a send in a message-passing system, with the added features of automatic encoding of the arguments and the creation of a new thread of control in the receiving component.

To add this feature to Schooner, two changes were needed. First, the UTS type definition language was extended by the addition of `dispatch` as an alternative to `prog`; thus, the UTS specification for the Controller's main routine became:

```
export controller_main dispatch()
```

In this example, there are no parameters to pass. When parameters are present, they are limited to being `val` parameters, due to the one-way nature of the communication. For the same reason, a dispatch cannot have a return value.

Second, additions were made to the stub compilers and Schooner runtime libraries. In a normal procedure call, the stub compiler produces the code for encoding the arguments, an invocation of the Schooner library routine `sch_call`, and the code for decoding the arguments after the reply is received. For a dispatch, the stub compiler produces only the code for encoding the arguments plus an invocation of a new Schooner library routine `sch_dispatch`, that returns immediately, rather than waiting for the reply message. Figure 5-8 shows the control flow through the dispatch stub and the Schooner runtime library.

Dispatch has slightly different semantics depending on whether the user has chosen TCP or UDP as the communications mechanism. With TCP, the user can be certain that the call has

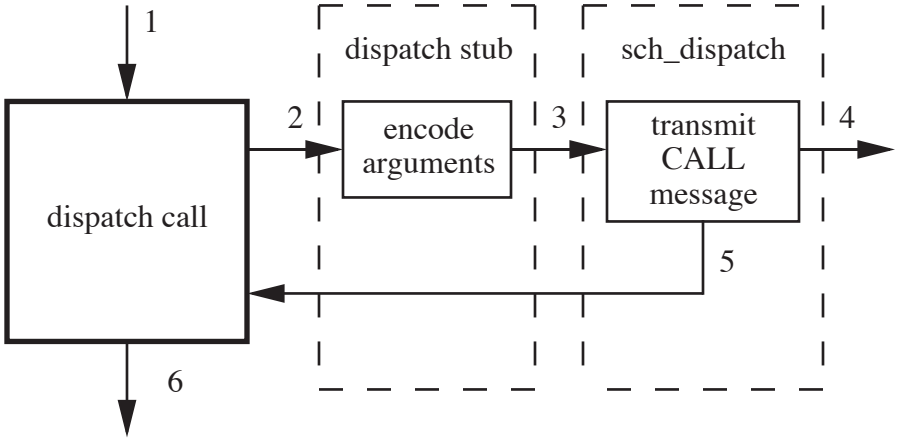


Figure 5-8: Control Flow in a Dispatch

reached the destination component when control is returned to the user code. With UDP, the user only knows the message has been sent, since UDP does not provide any indication of successful receipt.

5.3.2.2 Procedure Calls as Events

Components such as the Controller must respond to events generated by the user, as well as to procedure calls from other components in the line. In essence, a line containing such a component has two threads of control. One thread is the normal thread, which is passed among the components by procedure calls, while the event-driven component has a separate thread solely to handle external events. The Controller, for example, shares the Manager's thread of control when processing calls from the Manager that report on the various activities of the meta-computation and uses its own thread to handle X-events generated by the user.

The Controller switches from its own to the Manager's thread in two cases. The first occurs when the Controller invokes one of the Manager's exported procedures. In this case, the Controller behaves as a normal component, and blocks to await the reply. The second occurs when the Manager invokes a procedure exported by the Controller. In this case, the Controller needs the ability to switch to the Manager's thread long enough to process the call and send the reply. To provide this ability, a routine was added to the Schooner communications library that allows a component to poll its communication port periodically to determine if an incoming call has been received. Each time the function `sch_listen_poll` is invoked, it checks the component's port for a call. If no call is waiting, the function returns. If one or more calls are waiting, the function processes the first call, and then returns. By placing a call to this function inside the main event

For the Controller, it is possible to have multiple calls queued from the Manager, each relating to a different line.

loop, the Controller is able to treat such calls at the same priority as X-events.

While developed for the Controller, this technique has proven useful in general when event-driven applications are included in Schooner meta-computations. Chapter 6 describes an example in the context of the NPSS project.

5.4 Summary

This Chapter has presented the extended Schooner model, describing its concurrency and dynamic configuration features. The extended model supports multiple threads of control in scientific meta-computations, allows re-configuration of computations through improved dynamic integration and component movement, and provides the user with greater control and flexibility through the Controller. The Chapter also highlighted the principal changes in Schooner's implementation to support these features.

Chapter 6
NPSS CASE STUDY

The Numerical Propulsion System Simulation (NPSS) project aims to reduce the high cost of designing and implementing new propulsion technologies by using computer simulation [Claus91, Claus92]. Specifically, the project, which is part of the High Performance Computing and Communications (HPCC) initiative [HPCC94], involves developing both computational codes to model various engine components, and a simulation executive to control the simulation and provide a platform for modelling interactions among components. 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 simulation executive is a meta-computation that must support diverse software and hardware elements on local- and wide-area networks. The user should see a single integrated tool, not individual codes that execute in isolation. 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 be easily monitored and controlled; and expert system technology can be integrated to assist in controlling long simulations. The simulation executive, thus, requires the services of an interconnection system. Dynamic re-configuration allows the easy incorporation of different engine modules into simulations and adaptation to machine outages that can occur during long simulations. Concurrent execution can be used to simulate the parallel air flows that can occur in jet engines. Finally, monitoring and trace facilities can assist the programmer in debugging simulations.

This chapter describes how heterogeneous distributed processing is supported in a prototype version of the NPSS simulation executive, with the Schooner interconnection system providing the key connection capability. Schooner is being used in conjunction with an execution framework provided by AVS to form a simple simulation executive that approaches the capabilities required by NPSS [Homer94b]. In a separate project, Schooner is also being used to connect a monitoring interface with a remote high fidelity fan simulation.

6.1 The NPSS Project

6.1.1 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 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.

6.1.2 Hardware

NPSS uses 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 Hypercube and the IBM SP-1, as well as planning for future generations of massively-parallel machines and workstation clusters. In some cases, parallel machines can also provide a more natural solution to a problem, e.g., 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.

6.1.3 Software

The NPSS software effort is concentrating in three areas:

- Connecting codes that execute on different architectures and/or 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 codes written using 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 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 and allowing the user to switch to a higher or lower level code during a 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 trade-offs 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.

6.2 The Prototype 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, described in Chapter 4, 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. For NPSS, the dataflow can be used to model the flow of air through the engine. The AVS widget mechanism can be used to allow the user to interact with the individual engine components, providing data at the beginning of the simulation to configure the components, and during the simulation to allow the user to "fly" the engine. Schooner, in turn, provides the ability to perform the actual computation

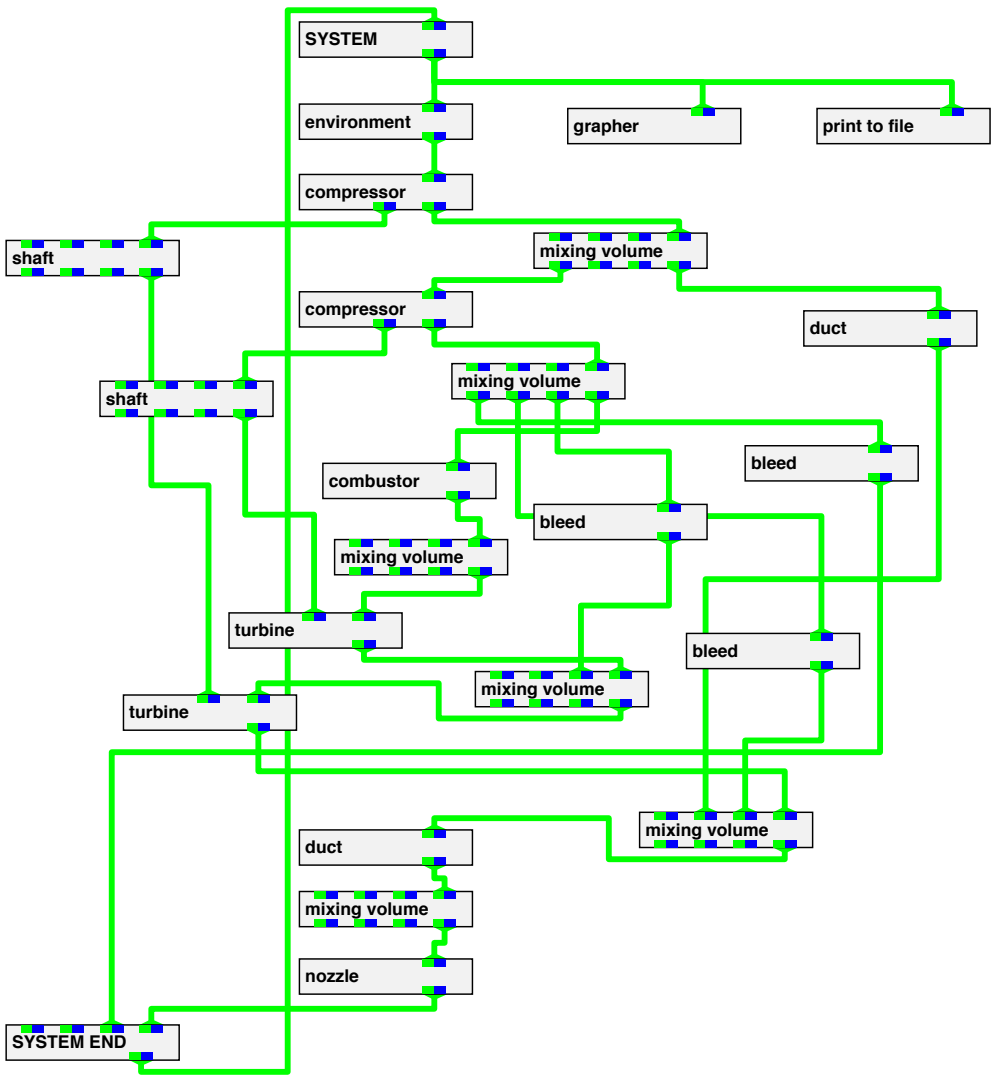
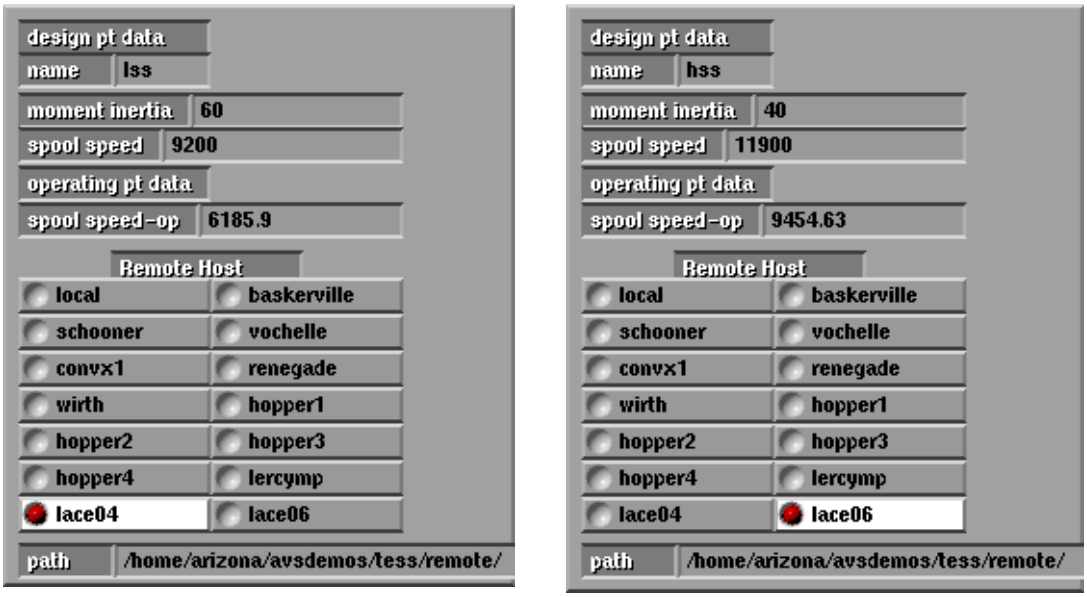


Figure 6-1: TESS F100 Engine Network

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), a complete one-dimensional transient thermodynamic aircraft engine simulation [Reed93]. 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 6-1 shows an AVS network for modeling an F100 engine using TESS. The particular engine to be modeled is determined by the modules chosen to represent the engine, and the inputs to each module. The inputs can come from three sources: data passed through the data-flow network from upstream modules, values passed through the widget mechanism, and data files read by the module.



low speed shaft high speed shaft

Figure 6-2: TESS F100 Shaft Controls

A module in TESS can be used to represent more than one component in the simulated engine. The characteristics of each instance of a given module are determined by the dataflow connections to other engine components and by values the user supplies through the widget mechanism. For example, in the F100 engine there are two instances of the shaft module: one models the high-speed shaft and the other the low-speed shaft. The control panels for these two shaft modules are shown in Figure 6-2. The control panel has widgets that accept user inputs for the *moment inertia*, *spool speed*, and *spool speed-op* parameters. Each shaft module is connected on the upstream side to the low-pressure or high-pressure compressor and on the downstream side to the low-pressure or high-pressure turbine, as appropriate. 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 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 variable factors, such as stator angle and fuel flow during the transient. These provide widgets that give the user the ability to specify new values the parameter will have at specified times during the transient. TESS uses linear interpolation between the specified values at other times.

The system module in TESS 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 Modi-

fied 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. Once the engine is balanced, the engine transient begins and proceeds up to the number of seconds specified by the user. The AVS graph viewer is used to provide a means of plotting some parameters of interest in the simulation. For example, the user can plot the number of equations that have converged during the steady state computation and can then plot the temperature, pressure, or mass flow rate at selected points in the engine during the transient calculation.

6.2.1 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 the shaft module to use Schooner, the procedure was very similar to that described in Chapter 4 for the molecular dynamics and neural net applications. In the case of the shaft code, 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.

The two UTS specification files, one associated with the remote procedure and one with the AVS module, were written. For `shaft`, the two were identical except for the use of the keywords `import` and `export`; the relevant export specification is shown in the top half of Figure 6-3.

The final step involved modifications to the AVS module to add the desired interaction facilities and to access Schooner's dynamic configuration library. Three short pieces of code are needed to accomplish these tasks. The first is placed in the `spec` function of the module to define two widgets that allow the user to specify the machine on which to execute the remote procedure and its pathname. These lines are shown in the bottom half of Figure 6-3. 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. These two widgets appear in the lower half of the shaft control panels in Figure 6-2.

The second section of code is added at the beginning of the `compute` function in each AVS module that is invoking a remote computation. The added code accesses the dynamic configuration library of Schooner to register the AVS module with the Schooner Manager and ask the Manager to start the remote process. On subsequent passes through the AVS module, the code may

```
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)

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')
```

Figure 6-3: Shaft Specification File and AVS Initialization Code

invoke the move routine asking the Manager to place the computation on a different machine. The code is shown in Figure 6-4. The value of machine and path, which are passed as arguments to `sch_start_component` or `sch_move_component`, are set when the user makes the selection of a remote machine and enters the pathname using the two widgets described above. The widgets serve to specify the initial placement of the remote component. Once execution of the simulation has begun, the same widgets are used to specify the machine to which the remote component is to be moved.

```
character*(*) machine
character*(*) path
character*(100) current_machine
save first_time_shaft, current_machine
logical first_time_shaft/.true./

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

if (current_machine .ne. machine) then
   call sch_move_component('shaft', current_machine,
         'shaft', path_name, machine)
   current_machine = machine
endif
```

Figure 6-4: Code to Configure Remote Shaft Component

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 6-1: TESS and Schooner Individual Module Tests

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. Since TESS calls the setup procedure for each of the selected modules only once per simulation run, there is no performance gain from separating the two procedures. Finally, in TESS, all the remote computations are stateless. As a result, the move routine was added to each to allow the remote component to be shifted to another machine during a simulation.

6.2.2 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 6-1. 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 6-2. 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 both one and ten second transient simulations 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. The experiments also included simulations where each of the remote components were moved to different machines during the run and the results again verified by comparison with the local-compute-only version.

6.2.3 Evaluation

The TESS prototype represents a first step toward the goal of a simulation executive as outlined at the beginning of this chapter. The use of AVS provides an environment for developing a prototype modular engine simulation without the need to create a custom user interface or data flow system. Schooner provides transparent access to remote heterogeneous resources and dynamic configuration. Work is continuing on further development of the prototype to include additional features, one example of which is described in the next section.

6.3 Monitoring and Steering a Remote Simulation

As noted above, two goals of NPSS are to allow the user to steer the engine simulation and to use zooming to include higher-fidelity components. Schooner is currently part of an experiment seeking to understand how to accomplish these goals. Researchers from the University of Toledo are extending the TESS engine simulation to include a high-fidelity simulation of the fan component. Another researcher from Cleveland State University is building a graphical interface to allow the user to monitor the execution progress of the engine components, and to experiment with the use of expert system techniques to control the engine simulation. The eventual goal is to connect the two projects to allow the user to monitor the remote execution of the fan, and to develop rules for the expert system to assist in controlling the simulation. Schooner is being employed to link the codes in the two projects.

AVS Machine	Module	# of Instances	Remote Machine
TESS Simulation executed on Sun Sparc 10, 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 6-2: TESS and Schooner Combined Test, Initial Placements

The initial phase of this project has consisted of two sub-projects: integrating a high-fidelity fan simulation into TESS, and the creation of a graphical interface to monitor the execution of the fan. Schooner is primarily involved in the monitoring portion of the initial phase, connecting it with the machine executing the fan simulation. This section outlines this initial experiment, then reports on the changes now being implemented as a result of the lessons learned from the two sub-projects.

The fan component chosen for integration into TESS is the Advanced Ducted Propfan Analysis Code (ADPAC) [Hall93]. Originally developed for the study of high-speed ducted propfan aircraft propulsion systems, ADPAC has become a general solver for turbomachinery components. It predicts the flow field in and around the fan through an Euler/Navier-Stokes numerical analysis that uses a three-dimensional, time-marching procedure along with a flexible coupled 2-D/3-D multi-block geometric grid representation. The code is written in FORTRAN and has been ported to a variety of machines including Cray YMP, Convex C240, and IBM RS/6000 and SGI workstations.

6.3.1 Monitoring ADPAC

The monitoring tool of the meta-computation consists of an interface constructed using the TAE+ Graphical User Interface toolkit [TAE]. TAE+ facilitates the construction of X-window GUIs by providing a workbench that the programmer can use to create windows and position controls. TAE+ then generates the code for creating the windows and builds an event loop to handle events to/from the controls within the windows. The user adds callback functions where necessary to perform application specific tasks associated with the controls. CLIPS, an expert system toolkit, is being integrated into the monitoring tool through the callback functions [CLIPS].

The initial goals of the monitoring project are to report the residual — a measure of how well ADPAC is converging — detect warnings, and report final results of the ADPAC run. Figure 6-5 shows the fan window of the monitoring tool designed with TAE+ for this project as a snapshot taken at the end of a run. The chart on the lower-right portion plots the residual on a log scale over the most recent 100 iterations. As the computation approaches convergence, the residual will have dropped four orders of magnitude. The calculation will terminate when the residual remains below 10^{-4}. When a computation is finished, the plot on the upper-right shows the pressure plot at 52 points along a slice through the fan. The scales on the upper left report the flow rate of air into and out of the fan. The scale on the lower left shows the final pressure ratio (outlet/inlet) for the fan.

A complicating aspect of this project is that the ADPAC source code is maintained by a separate group at Lewis and is not available for public release. However, ADPAC does produce a number of data files, one of which is continuously updated during a run to report a number of quantities, including the residual, and several types of warnings. While not optimal, this is a realistic situation. It is adequate for testing the feasibility of integrating the monitoring tool with ADPAC and determining the specific changes needed for ADPAC. There is another positive

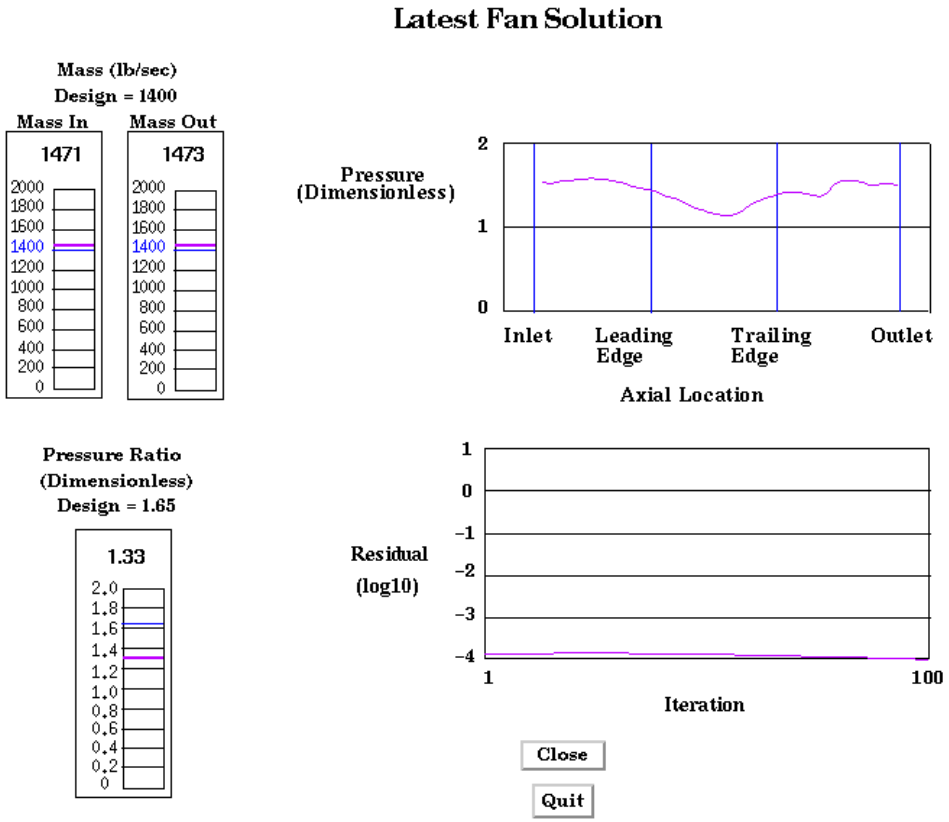


Figure 6-5: ADPAC Monitoring Interface

feature to the approach of using the output files: It would be relatively straight-forward to substitute a different high-fidelity fan simulation and provide a similar level of monitoring through watching its output file.

To simulate the type of monitoring desired given the constraints, a watch-dog process was created on the machine executing ADPAC. This process uses an infinite loop to continuously check ADPAC's output file for new data. Whenever the file changes, the watch-dog examines the file for values of interest, specifically the residual values from each iteration, any warnings generated, and the completion report. When one of these is found, it is reported to the monitoring tool. Eventually, if the project is successful, the functions performed by the watch-dog process may be built into ADPAC providing a direct connection between the monitoring tool and ADPAC.

The UTS specification for the watch-dog process is shown in Figure 6-6. The specification for the monitoring tool is analogous. The watch-dog uses a dispatch to begin the infinite loop that monitors the output files from ADPAC. The other procedures are all exported by the monitoring tool and receive the reports sent by the watch-dog process. For this initial experiment, the watch-dog terminates after sending the final reports.

```
# starting watch-dog
export monitor_files dispatch( "filename" val string[40])

# residual report
import residual_report prog(
        "cycle"    val integer,
        "residual" val float)

# warning reports
import warning_report1 prog( "message"  val string[-])
import warning_report2 prog( "message"  val string[-])

# final reports
import mass_in_report  prog( "mass_in"  val float)
import mass_out_report prog( "mass_out" val float)
import pressure_report prog( "pressure_ratio" val float)
import pressure_plot_report prog(
        "pressure" val array[52] of float)
import last_report prog()
```

Figure 6-6: ADPAC Watch-dog UTS Specification

The monitoring tool makes use of two features that were introduced for the Controller: the dispatch primitive and the ability to poll the communications port. During its initialization phase, the monitoring tool creates the watch-dog by a call to `sch_start_component`. It then calls the `monitor_files` dispatch routine. When the initialization phase is complete, the monitoring tool enters its main event loop. A call to `sch_listen_poll` is included within the main event loop to allow the receipt of reports from the watch-dog.

6.3.2 Evaluation

The first phase of this project was a partial success. The monitoring tool and watch-dog process performed well, with the expected performance impediment from the need to monitor ADPAC's output file rather than using a built-in ADPAC connection. The problem was exacerbated by the fact that the file buffering used by ADPAC would produce results from 80 to 100 iterations at one time rather than continuously. As a result bursts of calls to the monitoring tool are generated at intervals of roughly 90 seconds rather than the expected rate of about one call per second. This will be solved by having the watch-dog process report an array of results for all new iterations each time the file has changed.

A more significant result came from the work on integrating ADPAC into TESS. The key zooming problem addressed in this project is how to deal with boundary values at the 1-D/3-D boundary. The fan component was specifically chosen for the project over other engine components because the fan's upstream boundary conditions can be more easily modeled with 1-D/3-D

coupling. In the case of TESS/ADPAC, this process begins with the single inlet boundary values for stagnation pressure, stagnation temperature, and Mach number, and the exit boundary value of static pressure from the one-dimensional TESS code's fan component. These are then extrapolated to appropriate three-dimensional field distributions and applied as boundary conditions to the ADPAC simulation. The results of ADPAC are then integrated to determine the mass-flow rate, and the mass-averaged values of outlet/inlet ratios for the stagnation pressure and stagnation temperature. The mass-flow rate and the averaged stagnation pressure ratio are then compared with the same values computed across the one-dimensional TESS model. If the two pairs of values are compatible, then the extrapolated field distributions are proved to be suitable representations and the averaged values may be used in the one-dimensional simulation.

Typically, however, either the mass-flow rate or the averaged stagnation pressure ratio will not initially match the low-fidelity simulator results, and the three-dimensional boundary condition representations must be redefined and the above process repeated until the necessary match is found. It was originally assumed that an iterative process would suffice and produce a balanced engine after only a few iterations. This turned out not to work, however, requiring many iterations to achieve a balance. Worse, in many instances, the combination of the iterative approach and the numerical methods used to solve the system of equations governing the engine operation resulted in an oscillatory mode where convergence could not be achieved.

The solution found for this dilemma is the use of multiple runs of ADPAC and the construction of a performance map. Each ADPAC run has slightly varying parameters. From the multiple runs, a single-curve performance map is constructed and the appropriate value can then be chosen from the map for TESS, interpolating as needed. Thus far, tests have been conducted using as many as eight runs of ADPAC to create each constant-speed performance maps, and the solution appears to be a satisfactory one [Reed94]. To shorten the overall time for the simulation, the multiple ADPAC runs can be performed in parallel when the necessary computational resources are available.

Work is now underway to modify the monitoring tool to monitor (and eventually provide control over) multiple simultaneous ADPAC runs, and to update and display the performance map as each ADPAC run completes. Indications are that only a subset of runs started at any given time may actually be necessary. As some runs finish and the performance map is constructed, it may be possible to ignore (and halt) the remaining runs. Among the problems to be addressed in the monitoring tool are:

- With multiple simultaneous ADPAC executions, the fan window can become confusing. One choice is to create a display for each ADPAC instance and show all the results at once. Another choice is to show one at a time and allow the user to switch among them.

- Frequency of updates to the ADPAC display. It is not necessary to report results of each iteration, so a better scheme may be to show results at variable intervals. For example, one possibility is to report more frequently at the beginning of a set of ADPAC runs to determine

if the solution is converging. A control in the monitoring tool that allows the frequency to be changed under user direction is one possibility.

- Determining when enough ADPAC results have been received to create the performance map. This is especially useful when a heterogeneous collection of machines is available for the ADPAC runs, since it may not be necessary to wait for the slower machines to complete each time.

The ability to support shared components in Schooner, described in Chapter 5, is expected to be useful in the first two cases listed above. The monitoring tool would register with the Manager as a shared component, with each instance of ADPAC (or the watch-dog process) being created in a separate line, using Schooner's dynamic configuration library. On the first residual report, each ADPAC would send its location and line identifier to the monitoring tool. The other calls shown in Figure 6-6 would be changed to include the line identifier as a means of distinguishing the caller. Work is underway on the creation of an improved interface to display results for the multiple processes and to provide enhanced user control over ADPAC during execution. Work is also proceeding on the expert system layer to develop rules for determining when the performance map is complete. This would allow the excess instances of ADPAC to be terminated. Various extensions to CLIPS that support fuzzy logic are being evaluated in this context.

6.4 Summary

This chapter has outlined NASA's NPSS project and Schooner's participation in it. NPSS is providing realistic and on-going tests of Schooner's abilities. Both the concurrency and dynamic configuration features of the extended model have proven particularly useful in the TESS simulation executive. The work on the Controller proved fortuitous in providing techniques needed in the construction of the ADPAC monitoring tool. Lines and shared components will certainly prove useful in the extensions planned for the ADPAC system.

Chapter 7
EVALUATION AND FUTURE DIRECTIONS

7.1 Summary

This dissertation has developed a new model for scientific applications that exploits heterogeneity in the construction of meta-computations. Each meta-computation is constructed from a collection of component codes, regardless of the programming language, programming model, or machine architecture requirements of each. The resulting application is a heterogeneous, distributed program that executes on a variety of machines, spanning both short- and long-haul networks. Static and dynamic configuration give the user a great deal of flexibility in creating and executing the meta-computation.

The Schooner interconnection system realizes this programming model through a toolkit for the construction and execution of scientific meta-computations. Schooner employs an application-level remote procedure call mechanism to simplify the task of the scientific programmer. The UTS specification language is used to describe the interface for each component in a meta-computation. The runtime system supports the creation of the meta-computation by providing startup services, binding the components into the meta-computation, and resolving procedure names located in remote components. Schooner also provides two extended features. One is a limited form of concurrency that allows the execution of multiple simultaneous meta-computations and the use of coarse-grain parallelism within a meta-computation. The other is configuration management that supports dynamically adding, deleting, and moving components during execution, allowing the meta-computation to adapt to the demands of the computation without requiring the pre-allocation of resources.

The programming model and the Schooner interconnection system have been tested in a number of scientific applications. The molecular dynamics and neural net applications established the basic validity of the programming model and demonstrated the usefulness of the improved user interaction possible in this model. The on-going collaborative work with NASA on the NPSS project is providing applications that test the extended model and demonstrate its usefulness in a complex simulation.

7.2 Evaluation

The meta-computation model has worked well across a variety of applications. In the neural net, molecular dynamics, and ADPAC monitoring examples, the meta-computation was constructed by connecting existing codes and tools. In the TESS example, the meta-computation was created by splitting existing modules to separate each into remote and local components. In all cases, the resulting meta-computation provided improved interaction capabilities to the user, and

took advantage of the model's static and dynamic configuration abilities.

Schooner has performed well in its role as an interconnection system. The meta-computation abstraction and the use of procedure call semantics have both been very useful in developing scientific applications. The remainder of this section examines both high and low points of Schooner. These include features that succeeded, and features which have not yet proven useful. Finally, there are areas of Schooner's implementation where improvements are needed to correct shortcomings or provide more robust service.

7.2.1 Successes

The use of a procedure call paradigm allows for easy inclusion of components where the source code is available and provides a clean, familiar synchronization model for the scientific programmer. This factor was a primary motivation in retaining a procedure call interface in the extended Schooner model. A significant factor in the success of the procedure call paradigm is the UTS specification language, which makes it relatively painless to incorporate components into meta-computations. The runtime type checking, based on the UTS specifications and performed by the Manager, has proven helpful in getting remote and local specifications in agreement with each other. This has assisted both the scientific programmer in connecting components, and in our own system development when debugging the Manager, Server, Controller, and the Schooner runtime libraries.

The basic idea of a component consisting of a code block and an interface worked very well. In implementing the extended model's features in Schooner, the component did not change beyond some internal data structures within the interface. Thus, for example, it was not necessary to change UTS specifications for applications in migrating from the basic to the extended Schooner model; only re-compilation was needed. The line concurrency model worked well in the TESS simulation, which provided one of the original motivations for this feature. Lines are also providing the right level of concurrency for the ADPAC monitoring meta-computation.

Both static and dynamic configuration have contributed greatly to the construction of successful meta-computations. Creating the meta-computation at runtime gives the user considerable flexibility in selecting the specific components and machines without the need to re-compile. Dynamic configuration also eases the difficulties of long computations by providing the ability to incorporate new components when needed. For example, in the ADPAC monitoring tool, each successive set of ADPAC runs can take advantage of a different collection of machines as the need arises.

7.2.2 Promises Not (Yet) Realized

The underspecification aspect of UTS, which allows the user to incompletely specify array dimensions and records, has not yet been fully utilized in a meta-computation. There are a number

of reasons for this, but perhaps the most important is that scientific applications rarely use the type of complex data structures that could benefit from this feature. The one exception is the ability to underspecify arrays, which has found some usefulness. For instance, the neural net example from Chapter 4 allows variable size nets by not specifying the size of the arrays used to transfer the current node positions.

The cross-language call ability of Schooner is another feature that has been underutilized. In particular, the existing meta-computations have been written entirely in a single language, with the exception of the TESS/ADPAC monitoring project described in Chapter 6. In this meta-computation, one cross language call is made from TESS, which is written in FORTRAN, to the monitoring tool, which is written in C. The first major use of this feature is expected to be when the project reaches the point of modifying the FORTRAN source code for ADPAC. The watch-dog process will be replaced by incorporating its functions directly into ADPAC, allowing data to be exchanged with the monitoring tool directly. The cross-language capability has, however, been very useful in the implementation of Schooner since the runtime system is written in C and must communicate with components written in any supported language.

7.2.3 Changes Needed

As with most projects, there are things about Schooner that did not turn out quite as expected and would be done differently given the benefit of hindsight. One problem is that almost one-quarter of the Schooner Manager and runtime libraries consists of files that could be automatically generated by the C stub compiler, but are not at this time. The problem is that these files differ from the standard stubs in two principal ways. First, the internal data structures used in the system stubs differ, both in the conventions used for naming the stubs, and in the procedure identifier assigned to each stub. Second, the extended model required the Manager's stubs be split into two stubs, one to handle the decoding of the arguments and the second to handle the encoding. During development, it was easier to use the C stub compiler and make the few required changes to the generated stub by hand. The differences between the generated stub and the stub needed for system components have now stabilized, but remain a significant source of error whenever changes or additions are made to the Manager's services or the runtime libraries.

Another problem relates to error handling. The current Schooner system handles errors by intercepting signals within each component and passing an error message to the Manager. There are two limitations to this system. First, the `SIGKILL` and `SIGSTOP` signals in Unix cannot be intercepted. When these occur, the component is terminated without a message to the Manager, leaving the other components in the line still active but with execution suspended waiting for the now terminated component. The second problem is an inability to distinguish between a very long procedure call and a network or machine failure. Components in Schooner do make use of a time out mechanism that will raise an error when too much time has elapsed between a call and a reply. However, this is dependent on the user making a good guess of the time needed for a procedure to

complete a call. The solution to both problems is, first, to have the Manager keep a list of machines on which components have been started and periodically poll the Servers on those machines to determine if the machine and network connections are still active. Second, each Server can keep a list of components it has started and notify the Manager should one of them terminate.

7.3 Improvements to Schooner

The current implementation of Schooner can be improved in a number of ways, several of which are identified below. Generally, the implementation effort on Schooner up to now has been driven by the specific needs of particular meta-computations. The items in this list are possibilities that are currently under consideration, but for which a specific need has not yet arisen. They fall into two broad categories:

- Improvements that affect the performance of the system, and
- Extensions to the configuration and heterogeneity support.

7.3.1 Performance Enhancements

The UTS libraries and generated stubs offer several opportunities for improving the rate of data transfer. One option is to take advantage of specific machine features to improve the encoding and decoding of values, for example, by employing vector or parallel techniques for an array of simple types. Given the predominance of large arrays of floating point numbers in scientific applications, this option could yield a significant improvement in communication time.

Another option is to omit the UTS type tags to reduce the size of messages used for calls and replies. This option should be applied carefully, however, since the tags have been found useful when debugging meta-computations. One possibility is to modify the stub compilers to allow the choice of tagged or untagged data at compile time. This would allow the tags to be used during development, with only a re-compilation necessary to remove the tags. Since scientific applications make extensive use of float and double values, a variation of this idea is to omit tags on individual elements of any array containing only simple types. A single tag would be included in the header of the array in this case to indicate the type of all the elements.

A third option is to not convert data to an intermediate format unless required. The component could include the machine type in the exported procedure information sent to the Manager. Then, when a call or reply is made, the sender can determine if a conversion is needed. This would allow, for example, a Cray to call another Cray or a call between two machine types that both support the IEEE floating point standard to proceed without data conversions. When the two machines use different types, the UTS encode and decode library would be used.

Finally, one area that has not seen significant improvement since the original MLP work is the communication libraries. Support can be added to take better advantage of high-bandwidth networks. For example, developing a version of Schooner that uses the *x*-kernel [Hutchinson91]

on Mach would provide both a faster interface to existing networks and facilitate easier experimentation with emerging high-performance network protocols. Another extension would allow components to choose the best protocol to use for each procedure call. The information about the available communication protocols for a component's exported procedures would be included in the registration information sent to the Manager. When making a call, a component would choose the best protocol from the intersection of its own protocols and the remote procedure's protocols.

7.3.2 Configuration and Heterogeneity Enhancements

As described in Chapter 5, a limited ability to transfer global state information when moving a component would make components that retain state across procedure calls eligible for movement. One way to realize such functionality is to exploit the availability of UTS. Specifically, a UTS type specification can be used to describe globally defined variables whose values are to be transferred to the Manager in reply to the termination call associated with a component move.The Manager would then send these values to the new instance of the component after it has registered.

Another enhancement concerns dealing with unresolved procedure names. When the Manager does not find the procedure name in the mapping tables, it currently declares an error and terminates the line. Two additional solutions are possible. One is the approach taken with the design of the Controller, where the Manager passes the procedure name to the Controller and the user is then queried about a component to bind into the line that contains the missing procedure. A second approach would be to add a default search path, i.e., a pre-defined list of machines and directories that are searched for the missing component. The Manager would examine UTS specification files in these locations with the assistance of the Server on each machine. If the missing procedure were exported by one of the files, the corresponding component would be started and automatically bound into the line.

7.4 Continuing Projects and Future Research

Schooner is a part of several on-going projects. One is the ADPAC monitoring tool described in Chapter 6. Another is a project at NASA Lewis Research Center to monitor the results of a jet engine test in real-time, producing plots of the temperature and pressure at and between telemetry points inside the engine. The goal is to update the display at one second intervals during the test and take advantage of computational fluid dynamics (CFD) techniques to compute values between the telemetry points. The system uses AVS as the visualization end of the meta-computation. It has been tested with Schooner providing the connection to a remote interpolation routine that executes on a Convex or Cray computer. Parallel routines to perform CFD computations rather than interpolations are under development for use on either an IBM SP-2 or a Cray T3D. A third project is an NSF-funded HPCC Grand Challenge project at The University of Arizona that will couple an ecosystem modelling code with a geological database. Schooner will provide connections between

the simulation engine, the geological database, and a graphics workstation, and will support user interaction with and control over the simulation.

A number of research issues remain unanswered. One is fault tolerance as it applies to the construction of meta-computations, such as an investigation of which existing techniques are most appropriate. Facilities for checkpointing and recovery of scientific applications are important given their long-running nature, the instability of new machines, and the possibility of service interruptions on long-haul networks. An investigation is needed into the functionality required and how to incorporate it so that it is transparent to the application.

A second issue is possible alternatives to UTS. Although the UTS specification language and intermediate representation have worked well, they were developed in the mid-80's and more recognized alternatives (e.g., ASN.1) have been developed in the interim. An evaluation of alternatives based on the needs of scientific applications can also include exploring the automatic generation of specification files from source code and the usefulness of underspecification in scientific applications.

A final issue is the role of the computer scientist in creating meta-computations. The experience to date indicates that each meta-computation is generally a collaborative effort among a team of scientists and engineers representing various disciplines. The computer scientist in such a team has often filled a role similar to that of the interconnection system, acting as the enabler for the project and resident resource on computing-related matters. For example, scientific programmers are often not aware of the interconnection possibilities inherent in today's computing systems. In such situations, the computer scientist is forced into the dual roles of advocating the meta-computation concept and educating the other team members in its use and benefits. We feel this is partly due to the lack of appropriate software tools and the need to assist new users in exploring tools such as Schooner.

Perhaps the real question is the extent to which computer scientists can or should remain long-term members of such collaborative teams. We feel that continuing to require the presence of a computer scientist on every collaborative team will act as a limiting factor in the number of successful collaborations. To achieve the goal of independence, however, two requirements must be satisfied. The first is to expand on the work begun with Schooner to create better software tools that hide the technical details of connecting applications, while allowing the scientific programmer to access the full power of interconnected heterogeneous machines without specialized assistance. The second requirement is to develop quality supporting documentation that allows the scientist and engineer to employ the tool without assistance, a factor at least as important to the goal of independence as the software interconnection system itself.

Appendix A

MOLECULAR DYNAMICS AVS MODULE

```
#include <stdio.h>
#include <gettime.h>
#include <avs/avs_math.h>
#include <avs/avs.h>
#include <avs/field.h>
#include <avs/port.h>

#define MAX_PARTICLES 300


typedef struct vp_struct {
   double x, y, z;
};


dynamics_destroy()
{
    sch_i_quit();
} /* dynamics_destroy */


/*********************************************************/
dynamics_spec()
{
    int param1, param2, param3, param4, param5, param6, param7,
        param8, param9, param10;

    AVSset_module_name("call dynamics", MODULE_DATA);

    AVScreate_output_port("Output Field",
                          "field 3D irregular 3-vector float");

    param1 = AVSadd_parameter("# particles", "integer", 8, 1,
                              MAX_PARTICLES);
    AVSconnect_widget(param1, "typein_integer");

    param2 = AVSadd_float_parameter("vel std dev", 0.1, 0.0, 1.0);
    AVSconnect_widget(param2, "typein_real");

    param3 = AVSadd_float_parameter("potential cutoff", 10.0, 0.0,
                                    100.0);
    AVSconnect_widget(param3, "typein_real");

    param4 = AVSadd_float_parameter("density", 0.1, 0.0, 1.0);
    AVSconnect_widget(param4, "typein_real");
```

```
    param5 = AVSadd_float_parameter("time step", 0.001, 0.0, 1.0);
    AVSconnect_widget(param5, "typein_real");

    param6 = AVSadd_parameter("# time steps", "integer", 100, 1,
                              INT_UNBOUND);
    AVSconnect_widget(param6, "typein_integer");

    param7 = AVSadd_parameter("re-start", "boolean", "true", NULL,
                              NULL);
    AVSconnect_widget(param7, "toggle");

    param8 = AVSadd_parameter("continue", "boolean", "true", NULL,
                               NULL);
    AVSconnect_widget(param8, "toggle");

    /* Create the two Schooner widgets */
    /* First, allow the user to select the remote machine */
    param9 = AVSadd_parameter(
        "machine", "choice",
        "hopper3.lerc.nasa.gov",
        "caslon:schooner:hopper1:hopper2:hopper3:convx1:wirth:
         renegade:lace04", ":");
    AVSconnect_widget(param9, "radio_buttons");

    /* Second, allow the user to type-in the pathname */
    param10 = AVSadd_parameter("path name", "string",
                               "/usr/patrick/avsdemos/dynamics",
                               NULL, NULL);
    AVSconnect_widget(param10, "typein");
    AVSadd_parameter_prop(param10, "width", "integer", 6);

    AVSset_compute_proc(dynamics_body);
    AVSset_destroy_proc(dynamics_destroy);
} /* dynamics_spec */


/*********************************************************/
dynamics_body(output, no_particles, vel_std_dev, cutoff,
              density, time_step, num_time_steps,
              re_start, go_on, machine, path_name)
    AVSfield_float **output;
    int no_particles;
    float *vel_std_dev, *cutoff, *density, *time_step;
    int num_time_steps, re_start, go_on;
    char *machine, *path_name;
{
    int i, j, size, side, dims[3];
    struct vp_struct positions[MAX_PARTICLES],
                     velocities[MAX_PARTICLES];
```

```
static long cur_time_count = 0;

static int machine_called = 0;   /* = 1 once manager called */
static char my_name[] = "call-dynamic";
char *user_response; /* user's reply to AVS error message */
int  res; /* result of attempt to start remote component */

/* Start the remote Schooner component */
if (machine_called == 0) {
    do {
        res = sch_start_component(my_name, machine,
                                  "/dynamic", path_name);
        if (res == -1)
            user_response = AVSmessage("call-dynamic",
                                       AVS_Error,
                                       NULL,
                                       "dynamics_body",
                                       "Try again!Exit",
      "Failed to start remote component dynamic on machine %s",
                                       machine);
    } while ( (res == -1) &&
              (strcmp(user_response, "Try again") == 0) );

    if (res == -1)
        return 0; /* to indicate failure */

    machine_called = 1;
}

/* create AVS field to hold data */
if (*output)
    free(*output); /* free memory from previous call */

size = array_size(no_particles);
side = size * size * size;
dims[0] = dims[1] = dims[2] = size;
*output = (AVSfield_float *)
          AVSdata_alloc("field 3D irregular 3-vector float",
                        dims);

if (re_start)
    AVSmodify_parameter("re-start", AVS_VALUE, 0);

AVSmodify_parameter("continue", AVS_VALUE, 0);
```

```
    /* call appropriate remote procedure */
    if (re_start) {
        GenerateParticles(no_particles, *vel_std_dev, positions,
                          velocities);
        cur_time_count = 0;
    }
    else
        dynamic(no_particles, *vel_std_dev, *cutoff, *density,
               *time_step, num_time_steps, &cur_time_count,
               positions, velocities);

    /* put data into AVS field */
    for (i = 0, j = 0; i < no_particles; i++, j++) {
        (*output)->points[j] = (float)positions[i].x;
        (*output)->points[j + side] = (float)positions[i].y;
        (*output)->points[j + 2 * side] = (float)positions[i].z;
    }

    for (i = 0, j = 0; i < no_particles; i++, j += 3) {
        (*output)->data[j] = (float)velocities[i].x;
        (*output)->data[j + 1] = (float)velocities[i].y;
        (*output)->data[j + 2] = (float)velocities[i].z;
    }

    return(1);
} /* dynamics_body */


/*********************************************************/
int ((*module_list[])()) = {
    dynamics_spec,
};



/*********************************************************/
AVSinit_modules()
{
    AVSinit_from_module_list(module_list,
        sizeof(module_list)/sizeof(module_list[0]));
} /* AVSinit_modules */
```

```
/*************************************************************/
int
array_size( x )
    int x;
{
    int size;
    float fsize;

    fsize = pow ((double)x, (1. / 3.));
    size = (int) fsize;
    if (fsize == (float) size)
        return size;
    else
        return size + 1;
} /* array_size */
```

Appendix B
SCHOONER-AVS SCREEN SNAPSHOTS

The next two pages display screen dumps for the molecular dynamics and neural net meta-computations, respectively, that are described in Chapter 4. For each screen dump, the left panel contains the widgets for the application, including the widgets that allow the user to select the remote machine and specify the path to the executable. The AVS Network Editor window is in the background with the module palette across the top and the network for the application on the left. An intermediate graphical result is shown in the lower right foreground.

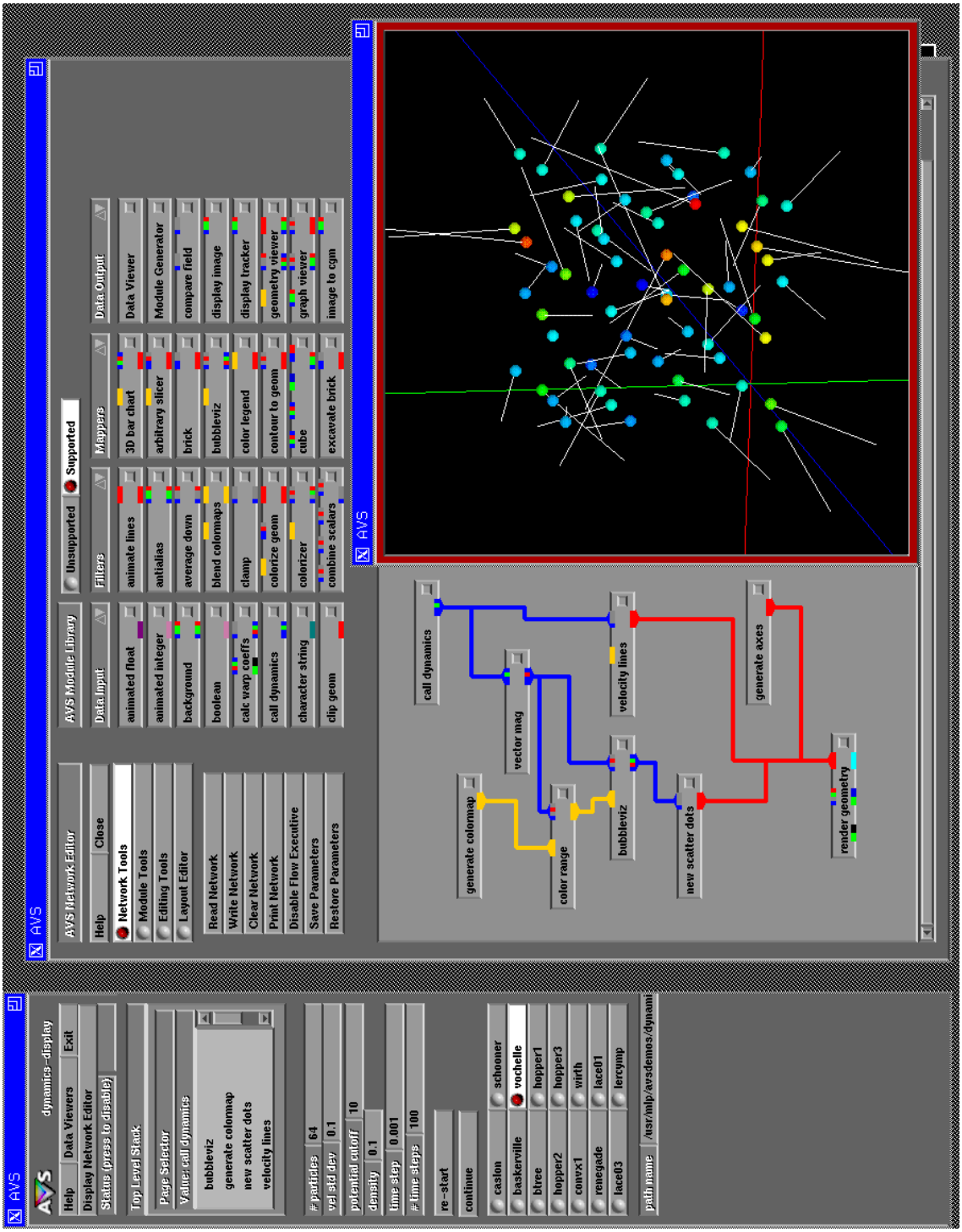


Figure B-1: Molecular Dynamics Screen

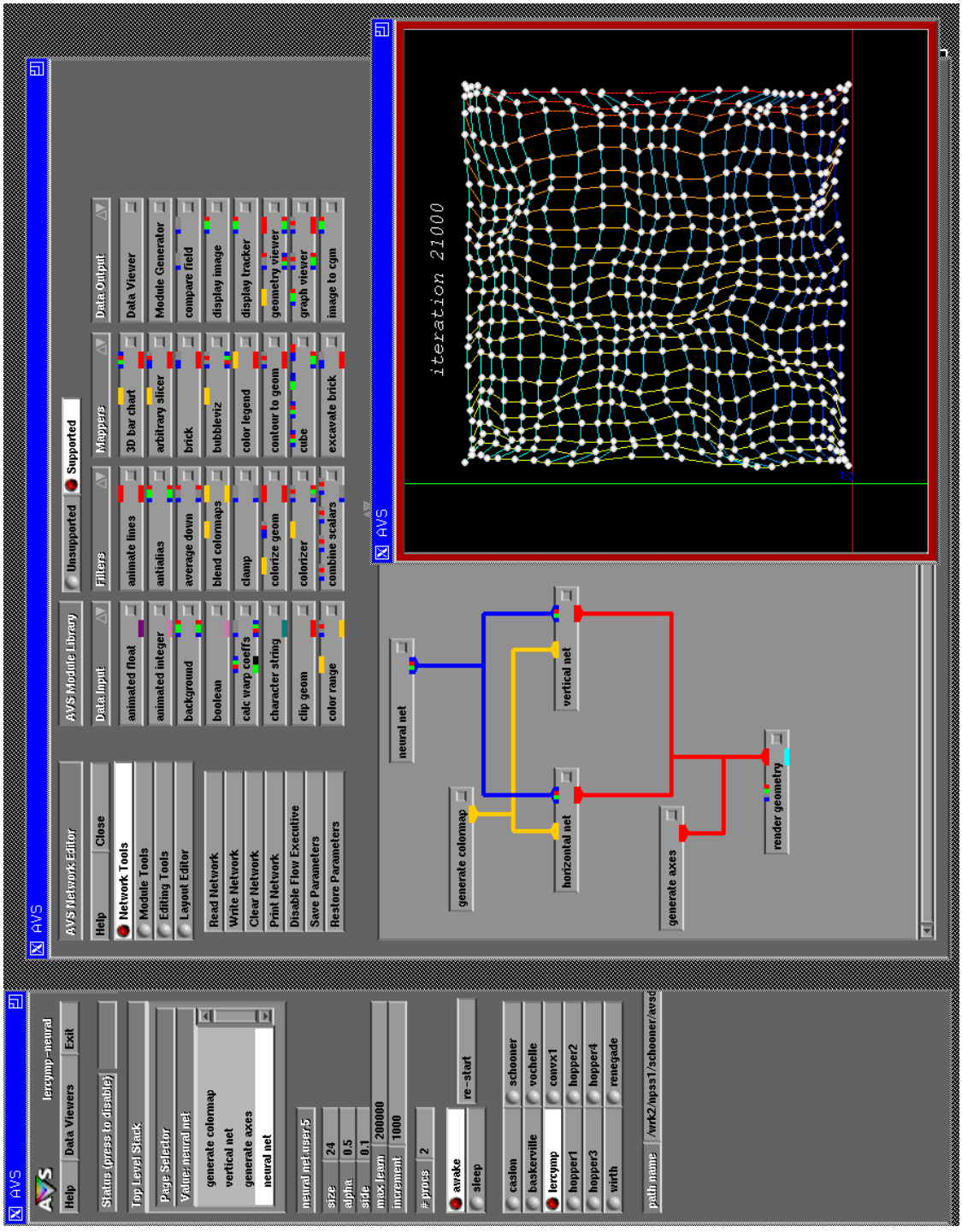


Figure B-2: Neural Net Screen

REFERENCES

[Allen82] M.P. Allen and D.J. Tildesley. Computer simulation of liquids. *CCP5 Quarterly 6*, 4 (1982).

[Almes85] G.T. Almes, A.P. Black, E.D. Lazowska, and J.D. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering SE-11*, 1 (January 1985), 43-59.

[Ananda92] A.L. Ananda, B.H. Tay, and E.K. Koh. A survey of asynchronous remote procedure calls. *Operating Systems Review* 26, 2 (April 1992), 92-109.

[Ananda91] A.L. Ananda, B.H. Tay and E.K. Koh. ASTRA–An asynchronous remote procedure call facility. *Proceedings of the 11th International Conference on Distributed Computing Systems*, Arlington TX (May 1991), 172-179.

[Andrews87] G.R. Andrews, R.D. Schlichting, R. Hayes, and T.D.M. Purdin. The design of the Saguaro distributed operating system. *IEEE Transactions on Software Engineering SE-13*, 1 (January 1987), 104-118.

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

[Becker94] T. Becker. Application-transparent fault tolerance in distributed systems. *Proceedings of the 2nd International Workshop on Configurable Distributed Systems*, Pittsburgh, PA (March 1994), 36-45.

[Beguelin91] A. Beguelin, J.J. Dongarra, G.A. Geist, R. Manchek, and V.S. Sunderam. Graphical development tools for network-based concurrent supercomputing. *Supercomputing '91*, Albuquerque, NM (November 1991), 435-444.

[Bergman91] L. Bergman, H. Braun, A. Kolawa, A. Kuppermann, R. Mechoso, P. Messina and J. Morrison. CASA Gigabit Network Testbed: 1991 Annual Report. CCSF-3-91, Caltech Concurrent Supercomputing Facilities, California Institute of Technology, July 1991.

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

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

[Black87] A. Black, N.C. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering SE-13*, 1 (January 1987), 65-76.

[Black86] A. Black, N.C. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, (October 1986), 78-86.

[Butler92] R. Butler and E. Lusk. User's Guide to the p4 Parallel Programming System. Technical Report ANL-92/17. Mathematics and Computer Science Division, Argonne National Laboratory. October 1992.

[Callahan91] J.R. Callahan and J.M. Purtilo. A packaging system for heterogeneous execution environments. *IEEE Transactions on Software Engineering 17*, 6 (June 1991), 626-635.

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

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

[Claus91] R.W. Claus, A.L. Evans, J.K. Lylte, and L.D. Nichols. Numerical propulsion system simulation. *Computing Systems in Engineering 2*, 4 (April 1991), 357-364.

[CLIPS] CLIPS Reference Manual, Basic Programming Guide. Software Technology Branch,

Lyndon B. Johnson Space Center. CLIPS Version 5.1, September 10, 1991.

[Douglis91] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software—Practice and Experience 21*, 8 (August 1991), 757-785.

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

[Freund89] R.F. Freund. Optimal selection theory for superconcurrency. *Supercomputing '89*, (November 1989), 699-703.

[Gibbons87] P.B. Gibbons. A stub generator for multilanguage RPC in heterogeneous environments. *IEEE Transactions on Software Engineering SE-13*, 1 (January 1987), 77-87.

[Grimshaw93] A.S. Grimshaw. Easy-to-use object-oriented parallel processing with Mentat. *Computer 26*, 5 (May 1993), 39-51.

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

[Hall93] E. J. Hall, R. A. Delaney, and J. L. Bettner. Investigation of Advanced Counterrotation Blade Configuration Concepts for High Speed Turboprop Systems, Task 5 — Unsteady Counterrotation Ducted Propfan Analysis Computer Program User's Manual, NASA CR-187125, January 1993.

[Hayes90] R. Hayes, N.C. Hutchinson, and R.D. Schlichting. Integrating Emerald into a system for mixed-language programming. *Computer Languages 15*, 2 (1990), 95-108.

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

[Hayes88] R. Hayes, S. Manweiler, and R.D. Schlichting. A simple system for constructing distributed, mixed-language programs. *Software—Practice and Experience 18*, 7 (July 1988), 641-660.

[Hayes87] R. Hayes and R.D. Schlichting. Facilitating mixed language programming in distributed systems. *IEEE Transactions on Software Engineering SE-13*, 12 (December 1987), 1254-1264.

[Hofmeister93] C. Hofmeister, E. White, and J.M. Purtilo. Surgeon: A packager for dynamically reconfigurable distributed applications. *IEE Software Engineering Journal 8*, 2, 95-101.

[Homer94a] P.T. Homer and R.D. Schlichting. A software platform for constructing scientific applications from heterogeneous resources. *Journal of Parallel and Distributed Computing 21*, (June 1994), 301-315.

[Homer94b] P.T. 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 (June 1994) 271-287.

[Homer94c] P.T. Homer and R.D. Schlichting. Configuring scientific applications in a heterogeneous distributed system. *Proceedings of the 2nd International Workshop on Configurable Distributed Systems*, Pittsburgh, PA (March 1994), 159-168.

[HPCC94] High Performance Computing and Communications: Technology for the National Information Infrastructure. Supplement to the President's Fiscal Year 1995 Budget. Committee on Information and Communication (CIC) of the National Science and Technology Council (NSTC).

[Huang94] Y. Huang and C.V. Ravishankar. Designing an agent synthesis system for cross-RPC communication. *IEEE Transactions on Software Engineering 20*, 3 (March 1994), 188-198.

[Hutchinson91] N.C. Hutchinson and L.L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering 17*, 1 (January 1991), 64-76.

[Jones85] M.B. Jones, R.F. Rashid and M.R. Thompson. Matchmaker: An interface specification language for distributed processing. *Proceedings of the 12th Symposium on Principles*

of Programming Languages, New Orleans, LA (January 1985), 225-235.

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

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

[Magee94] J. Magee, N. Dulay, and J. Kramer. A consturctive development environment for parallel and distributed programs. *Proceedings of the 2nd International Workshop on Configurable Distributed Systems*, Pittsburgh, PA (March 1994), 4-14.

[Mechoso92] C.R. Mechoso, C.-C. Ma, J.D. Farrara, J.A. Spahr, and R.W. Moore. Distributing a climate model across gigabit networks. *Proceedings of the 1st International Symposium on High-Performance Distributed Computing*, Syracuse, NY (September 1992), 16-25.

[Mechoso91] C.R. Mechoso, C.-C. Ma, J.D. Farrara, J.A. Spahr and R.W. Moore. Distribution of a climate model across high-speed networks. *Proceedings Supercomputing '91*, Albuquerque, NM (November 1991), 253-260.

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

[Morris86] J.H. Morris, M. Satyanarayanan, M.H. Conner, J.H. Howard, D.S. Rosenthal, and F.D. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM 29*, 3 (Mar. 1986), 184-201.

[MPI94] Message Passing Interface Forum. Document for a Standard Message-Passing Interface. March 22, 1994.

[Nelson81] B.J. Nelson. Remote Procedure Call. Ph.D. Dissertation, Carnegie Mellon University, May 1981. Xerox PARC Technical Report CSL-81-9.

[Ousterhout88] J.K. Ousterhout, A.R. Cherenson, F. Douglis, M.N. Nelson, and B.B. Welch. The Sprite network operating system. *Computer 21*, 2 (February 1988), 23-36.

[Postel81] Jon Postel. *Transmission Control Protocol—DARPA Internet Program Ptotocol Specification*, RFC 793, Network Information Center, SRI International, Menlo Park, CA, September 1981.

[Postel80] Jon Postel. *User Datagram Protocol*, RFC 768, Network Information Center, SRI International, Menlo Park, CA, August 1980.

[Purtilo94] J.M. Purtilo. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems 16,* 1 (January 1994), 151-174.

[Quealy93] A. Quealy, G.L. Cole, and R.A. Blech. Portable Programming on Parallel/Networked Computers Using the Application Portable Parallel Library (APPL). NASA Technical Memorandum 106238, July 1993.

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

[Reed94] J.A. Reed and A.A. Afjeh. Distributed and parallel programming in support of zooming in numerical propulsion system simulation, *OAI/OSC/NASA Symposium on Application of Parallel and Distributed Computing*, Columbus, OH, (April 1994).

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

[Skjellum93] A. Skjellum. Scalable libraries in a heterogeneous environment. *Proceedings of the 2nd International Symposium on High-Performance Distributed Computing*, Spokane, WA (July 1993), 13-20.

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

[Sunderam90] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency—Practice and Experience 2*, 4 (December 1990), 315-339.

[TAE] Transportable Applications Environment Plus. Programmer's Manual, Version 5.2. Goddard Space Flight Center, National Aeronautics and Space Administration.

December 1992.

[Wang92] M. Wang, S. Kim, M.A. Nichols, R.F. Freund, H.J. Siegel, and W.G. Nation. Augmenting the optimal selection theory for superconcurrency. *Proceedings of the 1st Workshop on Heterogeneous Processing*, Beverly Hills, CA (March 1992), 13-21.

[Weems93] C.C. Weems, Jr. Image understanding: A driving application for research in heterogeneous parallel processing. *Proceedings of the 2nd Workshop on Heterogeneous Processing*, Newport Beach, CA (April 1993), 119-126.

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

[Zimmerman94] M. Zimmermann and O. Drobnik. Specification and implementation of reconfigurable distributed applications. *Proceedings of the 2nd International Workshop on Configurable Distributed Systems*, Pittsburgh, PA (March 1994), 23-34.