Distributed Problem Solving Environments for Scientific Computing

by
Colin Joseph deSa

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
Computer Science and Applications.

APPROVED:

Calvin J. Ribbens, Chairman

Layne T. Watson

Dennis G. Kafura

June, 1991
Blacksburg, Virginia

LD
5655
V855
1991
D483
C.2

A DISTRIBUTED PROBLEM SOLVING ENVIRONMENT FOR SCIENTIFIC COMPUTING

by

Colin Joseph deSa

Calvin J. Ribbens, Ph.D., Dept. of Computer Science

(ABSTRACT)

The aim of this thesis is to research the issues involved in creating distributed problem solving environments for scientific computing. As part of our evaluation, we have developed a distributed problem solving environment called DPSolve which combines a very high level language, an interactive X Windows interface and a set of powerful problem solving methods into a single environment. The interface is designed to work on any system running X Windows, whilst the computations are done on a more powerful parallel computer. We implemented the interface on a DEC3100 workstation running ULTRIX, which communicates with procedures running on a Sequent S81 with 10 processors, running DYNIX via RPC.

The design decisions and implementation details of our system are discussed at length along with a detailed example of the system at work. We critically evaluate the approach we have taken and show why it can scale to a very large class of scientific problems. We conclude that this distributed environment should be a representative of future scientific problem solving environments.

ACKNOWLEDGEMENTS

I would firstly like to thank my advisor Dr. Calvin J. Ribbens for his support and excellent guidance over the last two years. Dr. Ribbens always seemed to come up with insightful suggestions whenever I came up with a problem that at the time seemed insurmountable.

Similar thanks go to the rest of my committee: Dr. Layne T. Watson and Dr. Dennis G. Kafura. I especially would like to thank Dr. Watson for very carefully reviewing this work and making many suggestions for improvement.

My parents have always been very supportive and have worked hard to provide me with the best they could. Without their support this work would not have been possible. I also express my gratitude to my fiance Zita for always being there for me. Her never failing optimism and encouragement helped me get through times when I felt this work would never reach completion.

I'd also like to thank my friends in Blacksburg for making my stay here an enjoyable one.

Contents

List of Figures

1 Introduction

Over the last few years advances in hardware technology have resulted in a manifold increase in computing power available to researchers in scientific computing. The advent of powerful vector and parallel computers has enabled the solution of problems which were considered to be unmanageably large only a few years ago.

However the advances in scientific software development have been significantly slower. The research scientist in most cases, still has to be concerned with the low level programming details necessary to solve his problems. This can be a tedious and time consuming task.

Rapid advances in workstation technology coupled with the development of powerful windowing software have tremendous implications for the scientific computing community. Using networking software, a user is able to distribute work between his workstation and a powerful host computer, which could be a vector or parallel machine. This model of computing can be viewed as a client/server model which has gained considerable popularity in distributed computing.

It has been predicted that the 90's will see an enormous increase in the use of distributed computer facilities organized hierarchically in terms of computing power, connected with appropriate networking hardware and software. Closing the gap between hardware and software technology is a very important issue, especially in the field of scientific computing.

It is important that the scientific researcher is provided with an integrated problem solving environment to aid him in his work. Such an environment should include a set of powerful problem solving modules, tools for interactive program specification, interactive module execution, and output analysis and visualization tools. Such an environment should exploit the gains made in hardware technology, and at the same time hide all unnecessary details, such as data communications, detailed knowledge of local and host machine architecture, and so on, from the end user.

The development of such an environment is not any easy problem. In the following subsection, we discuss some of the issues the designer of a distributed problem solving environment must address.

1.1 Basic issues and approach

There are several important issues and design goals which must be addressed in developing a useful distributed problem solving environment. First, such a system must be accessible to a novice user, but powerful enough for an expert. While it should require minimal previous computer experience, it should also provide an experienced user with an advanced user interface. This would allow him to quickly define his problem to the system without getting slowed down by an over-friendly (to the expert) novice interface.

The system should be user-customizable. This means that the user should be able to choose only those resources he needs for solving his specific problem, and in a sense build his own "mini system".

The system should utilize the distributed computing power of the network, and hence provide a distributed environment for problem solving. The performance of most numeric algorithms depends on the underlying machine architecture. By providing a distributed environment, an entire application or a sub-component can be executed on the machine for which the best performance can be attained. Since most scientific users have at their disposal a variety of powerful machines, i.e., vector and parallel machines, systolic array processors, etc., providing a distributed problem solving environment enables optimal utilisation of network resources. However, the user should not have to take responsibility for the low level details that make remote execution possible. This requires, for example, that the communications protocol take care of issues like machine representation of data.

Such a system should allow a user to interactively execute portions of his program, and reset various parameters at runtime if necessary. This is particularly

important in the arena of scientific computing as by changing a few parameters a user can redefine his problem, without having to recompile and rebuild the entire system.

The system should support visualization of solutions. It should provide tools to the user to analyze his results graphically. As Hamming said several years ago, "The purpose of computing is insight, not numbers". In most cases, a single picture can convey more meaning than pages of figures.

The reliability of such a system is a very important issue. Since such a system contains many different components–user interfaces, graphics tools, communications software, problem solving modules–building a reliable system is not an easy task.

1.1.1 Approach taken

The aim of this thesis is to study the various issues in developing distributed problem solving environments for scientific computing. To aid in our research we developed a prototype system, *DPSolve*, which provides a computational environment for problem solving and research in adaptive methods for partial differential equations. We combine a very high level problem description language, an interactive X Window based user interface, and a set of problem solving methods into a single problem solving environment. The system is distributed in that the user interface runs on a workstation, while numerical computations are carried out on a more powerful parallel computer. The workstation we use is a DEC3100, running Ultrix 4.2. The host computer is a Sequent Symmetry S81 with 10 processors, running DYNIX. All communications are done via remote procedure calls between the client on the work station and the server on the host parallel computer. We feel that such a system is representative of problem solving environments that should be available for other problem domains, and that building such a system is a useful step toward a better understanding of distributed problem solving environments

in general.

1.1.2 Outline

The remainder of Chapter 1 discusses background materials essential to understanding the contents of the later chapters. In particular, the three major software systems on which DPSolve depends are reviewed, namely RPC, X, and ELLPACK. Chapter 2 surveys related work in the area. Chapter 3 discusses DPSolve from a user's point of view, describing session building and execution tools. Chapter 4 discusses the design of the system and various implementation issues. In Chapter 5 we evaluate the system, draw some conclusions for distributed scientific problem solving environments in general, and discuss future work.

1.2 Remote procedure calls

1.2.1 Distributed computing

The first computer systems developed were single user systems. Batch systems were next developed in the 1960's to improve CPU utilization. Batch systems did not allow the user any interaction with an executing job, which was fine for large jobs which required no user interaction, but this posed a problem for jobs dependent on user interaction. Timesharing systems were developed with a view to providing multiple users with interactive use of a computing system. A large number of users were allowed to interact with a single CPU, which switched from one user program to the next whenever the program completed execution, requested I/O, or its allocated time slice expired, giving each user the impression that he had his own computer. It was not until the early 1970's that timesharing became prevalent. The first designs for networks emerged in the late 1960's. One of the first networks to be designed and developed was the Arpanet (completed in 1977), see [McQu 77]. The emergence of Arpanet enabled communication between various geographically distant sites, allowing hardware and software to be shared economically by a wide

community of users. Each site maintained its own file system. The only way a file could be accessed by a user at another site was via the File Transfer Protocol (FTP) mechanism provided by Arpanet. In the mid 1980's the concept of *file sharing* evolved, whereby a workstation could make its filesystem visible across a network, enabling various users to share information.

The next logical step is distributed computing. In a distributed computing system a number of independent processors are connected together, and communicate using high-speed networks. A *distributed computing environment* is one which makes a collection of loosely connected computers appear to be a single entity.

Developers of distributed scientific problem solving software derive many benefits from working in a distributed computing environment. The most important benefit is that such an environment makes it easy to develop applications that use any of the resources the network has to offer, thus exploiting the computational power of the network. The tasks that involve vector operations can be run on a vector computer, whereas tasks which can be parallelized can be run on a parallel computer.

A distributed computing environment allows user applications to run over a heterogeneous network. Such environments are rapidly gaining popularity amongst distributed application developers. The OSF Distributed Computing Environment (DCE), see [OSF 90a], allows system applications like file systems, backup servers, database applications, and a variety of user-defined applications to run over a heterogeneous network. Various other vendors, like SUN Microsystems, also have developed distributed computing environments. Both SUN and OSF use the remote procedure call (RPC) model for interprocess communications.

1.2.2 The remote procedure call model

The following discussion on RPC is based on [SUN 88]. For a more complete explanation refer to [SUN 88] and [OSF 90a]. In the local procedure call model the

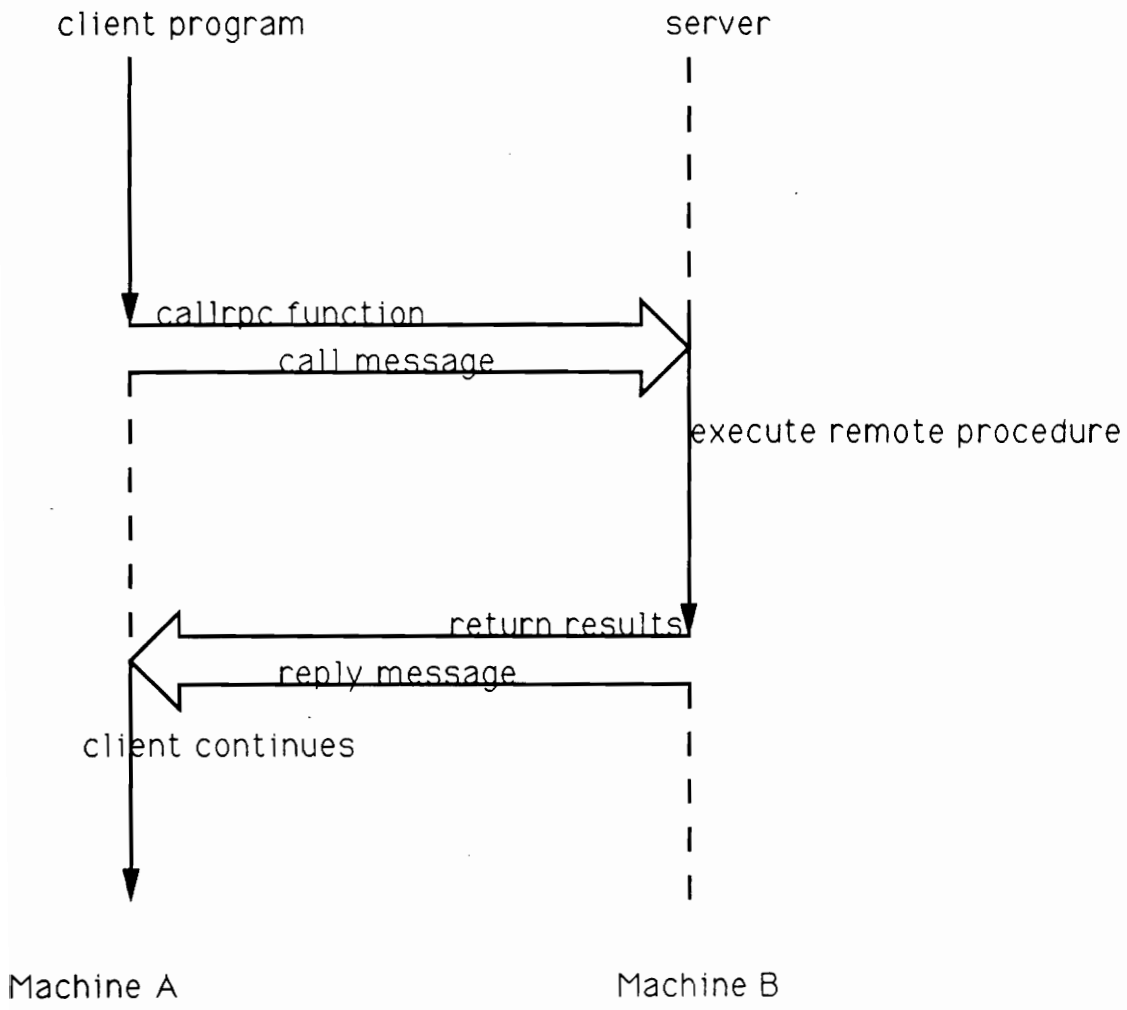

Figure 1: The RPC model of network communication.

caller places arguments in an activation record for the callee, and transfers control to the callee. The callee computes the results, and places them on the stack. When the callee has completed execution the caller regains control, retrieves the results and continues its execution.

The remote procedure call is the distributed analog of the local procedure call except now the callee and caller are distinct processes. The caller process sends a call message to the server and waits for a reply. The call message contains the caller's parameters, and the reply message contains the procedure's results. Once the results have been extracted the caller continues.

On the server machine, a server process is dormant awaiting a call message. When such a message arrives, the server process extracts the procedure's parameters, computes the results and sends a reply message and waits for the next call message. Figure 1 gives a detailed diagram of the RPC model.

RPCs allow distributed applications to run over multiple, heterogeneous systems, making the system architectures, and the underlying network protocols connecting the systems, transparent to the application procedures.

The different ways in which components of a heterogeneous system represent data initially posed a major problem in the communication of data using the RPC model. RPC systems use basically two different approaches to dealing with this problem.

The first approach is the *canonical* approach. That is, a single data representation is defined, i.e., a single byte order, a single floating point representation, and so on. Unless the client's data representation is the same as that of the predefined canonical format, it is converted to this form. The same procedure is repeated for the server. This works fine, except when both machines have the same data representation format which is not the same as the canonical form. Then, data would unnecessarily be converted and reconverted into its original format. Nevertheless, this approach has gained wide acceptance. This approach has been taken by the

developers of SUN RPC.

The second approach, used in the OSF DCE RPC, is to tag all calls with a description of the calling machine's data representation, which can be captured in less than four bytes. When the receiver gets the call, it converts the data from the data representation of the caller to its own data representation. But this only occurs if the caller's descriptor tags do not match those of the receiver.

In our implementation, we had to use the first approach, since we use SUN's RPC mechanism. The data from our workstation was converted to standard External Data Representation format (XDR), and then mapped to the data representation format of the Sequent Symmetry S81, which was our host machine.

1.2.3 Advantages of using RPCs

We briefly list some of the advantages of using RPCs for developing distributed applications.

- The RPC protocol is network transparent and network independent, thus applications based on RPCs can run on any network, and need not be rewritten for different networks.

- RPCs free the distributed applications programmer from having to be concerned with the low level interprocess communication details.

- RPCs are particularly suited to the scientific computing scenario. Procedure-level granularity is usually appropriate for scientific computing, hence tasks one would want to execute remotely can be represented by a remote procedure call, not by something at a lower or higher level. Thus total time-to-solution can be reduced by running computationally intensive jobs on high speed servers, via RPC.

1.3 The X Window System

We briefly describe some of the salient features of the X Window System. We have based this discussion on the excellent discussion in [Nye 88a, Nye 88b, Nye 90a, Nye 90b] and [Young 89]. The interested reader is also referred to papers by Simpson [Simp 90a, Simp 90b]. For a comprehensive description of programming in X as well as the underlying implementation decisions made by the original designers, refer to [Sche 86].

The X Window System is an industry standard for developing graphical user interfaces. The system provides network-based, bit mapped graphics and window management mechanisms. The most unique feature of X is its device independent architecture. This allows applications to display windows on any system supporting the X Protocol, without having to be modified.

X was first conceived at MIT's Laboratory for Computer Science, as part of Project Athena. It was designed to meet Project Athena's needs for a hardware-independent distributed user interface platform. Its initial development was supported by Digital Equipment Corporation (DEC). X was strongly influenced by another windowing system called W [Asen 84], which was developed by Brian Reed and Paul Asente at Stanford University. Robert Scheifler of MIT, and Jim Gettys of DEC [Sche 86], were the original designers and implementors of version 10 of X. Since then many other individuals have contributed to the current version 11. X was originally used only at DEC and MIT, but soon caught the interest of hardware and software vendors looking for a standard base for user interface development on their platforms. A vast number of companies have joined the X Consortium, and use X as their windowing management system. One of the reasons for X's popularity is that its specifications are controlled by the X Consortium, and any system allowing programming of X window applications must provide a standard set of libraries, namely Xlib and Xt. X is based on the client/server model of computing (refer to Figure 2). The X server controls the user's workstation or

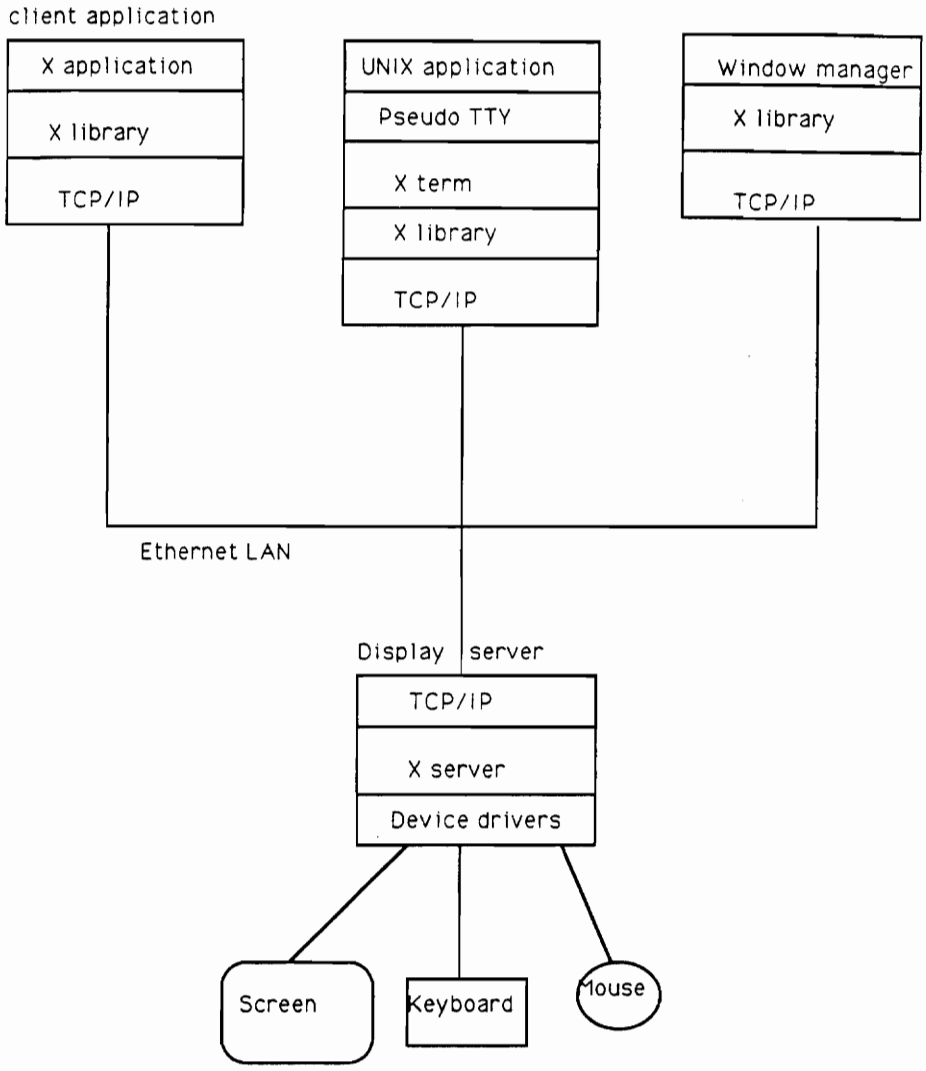

Figure 2: The client/server model of X. The client and server are separated via the X Network Protocol interface. (This figure is adapted from [Simp 90a]).

terminal display. For this reason it is sometimes known as a display server. It controls all I/O devices, and provides a layer between the application and the display hardware. It contains all the device specific code and thus frees applications from having to deal with differences in display hardware. The server provides the following services:

- It allows or denies access to the display to multiple clients.
- It interprets client messages and performs the appropriate actions.
- It passes user input, such as mouse button clicks or key presses, to clients by generating network messages called *events*. The server is responsible for passing the appropriate events to the client that requested it. Since multiple clients may be controlled by a single server it must "remember" what each client requested.
- It maintains the complex window and font data structures. Clients refer to these data structures simply by their ID's, which reduces the flow of information to be passed through the network.

Clients communicate with the server via an asynchronous byte stream protocol named the X Protocol. X Protocol requests are generated by making calls to Xlib routines. Any client can communicate with any server provided that both client and server understand the X Protocol.

The software composing the X Window System is organized hierarchically (see Figure 3). The lowest programmable interface to the system is Xlib, a library of C based graphics subroutines. Programmers make use of the Xlib interface if their applications need to use low level graphics primitives. This library is difficult to use, and programming at this level is tedious. Most vendors supply an X toolkit, which consists of a layer known as the Xt Intrinsics, and a collection of user interface components, called *widgets*. Xt is a library built above Xlib,

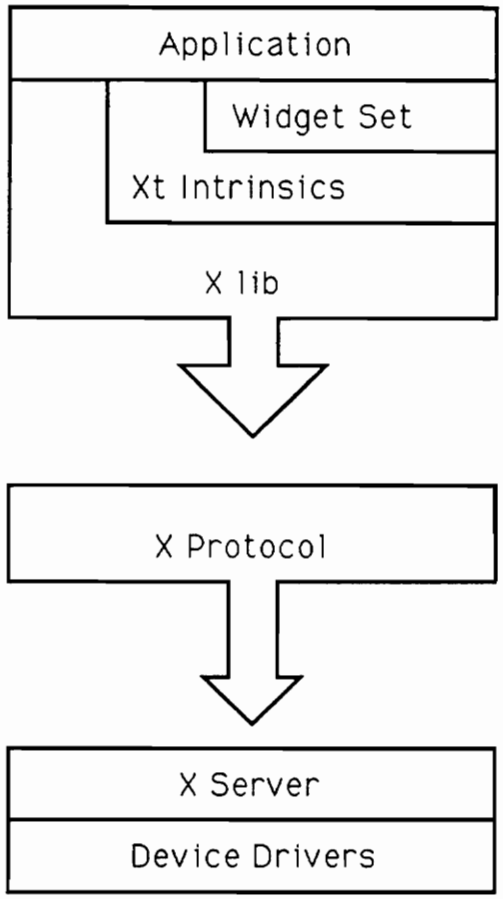

Figure 3: The software hierarchy of the X Window System.

designed to supply the user with subroutines to create and use widgets. A widget is a reusable, application configurable piece of code that has a predefined set of ways in which it can interact with applications. Most vendors support a widget set, which is a collection of widgets which provide commonly used user interface components such as scroll bars, dialog boxes, menus, and so on.

The DPSolve system described in Chapters 3 and 4 is based on X 11 Release 4, and uses the Athena widget set.

1.4 The ELLPACK system

ELLPACK is a mathematical software system for solving elliptic partial differential equations (PDEs) (see Rice and Boisvert [Rice 85]). It consists of a very high level language, and a set of methods, or problem solving modules.

The user specifies the problem he wishes to solve, and the sequence of methods to be used, in the ELLPACK language. A preprocessor translates the ELLPACK program into a Fortran control program. This program is then compiled and linked with the ELLPACK module library.

The ELLPACK language is an extension of Fortran. It provides a user with high level constructs to specify elliptic partial differential equations and boundary conditions on two and three-dimensional domains. The building block, or basic statement of the ELLPACK language is the *segment.* There are two classes of segments in the ELLPACK language. Group 1 segments define the elliptic problem, and must appear before any Group 2 segments. There are four Group 1 segments: *equation*, *boundary*, *hole*, *arc.* These segments can appear at most once in an ELLPACK program.

Group 2 segments specify ELLPACK modules and can appear more than once. The following are Group 2 segments: *grid*, *discretization*, *indexing*, *solution*, *output*, and *triple.*

ELLPACK statements invoke routines in the ELLPACK module library. There

are six basic types of modules:

GRID PROCESSING. A grid is defined, and the position of each grid point relative to the boundary is located.

DISCRETIZATION. The PDE and boundary conditions are approximated by a finite system of linear algebraic equations.

INDEXING. The equations and unknowns of the discretized system are reordered to facilitate solving the system.

SOLUTION. The system of algebraic equations is solved.

TRIPLE. Discretization, indexing, and solution are performed in a single step.

OUTPUT. A specified function (e.g., solution, residual, etc.) is plotted or tabulated.

Writing an ELLPACK program is a sequential process. One must list the appropriate ELLPACK statements in the correct order, beginning with the specification of the PDE, and its boundary conditions.

The ELLPACK system is based on a "software parts" technology [Batz 82], wherein each software part is an independent subprogram with predefined sets of inputs and outputs. In this way many different parts can be joined with each other. The ELLPACK control program invokes each part in turn.

Communication between parts is done through a set of interfaces. These include:

- PROBLEM DEFINITION INTERFACE
- DISCRETE DOMAIN INTERFACE
- DISCRETE OPERATOR INTERFACE

- EQUATION/UNKNOWN INDEXING INTERFACE
- LINEAR ALGEBRAIC SOLUTION INTERFACE

The domain processor module produces output in the discrete domain interface, a discretization module produces output in the discrete operator interface, and so on.

This design helps to reduce the communication overhead involved in making remote procedure calls to routines in the ELLPACK module library. We were able to export the ELLPACK interface definitions generated by the preprocessor, before invoking the RPCs on the host machine. In this way whenever an RPC is made to perform an independent ELLPACK module, its inputs are the output stored by its predecessor in their common interface. Only in a few cases must more than a few variables be passed.

ELLPACK's design is excellent when we have a single server process, but could cause problems if the PDE solving process is split up into many subcomponents (each component representing an ELLPACK module) each resident on a different remote machine. In this scenario it would perhaps be best to rewrite the ELLPACK modules so that inter-module communication takes place via parameter passing instead of common interface blocks. However, it could be argued that the most numerically intensive component of PDE solving is the solution of the discretized system. Hence only the different solution modules should be distributed to run on architectures which best exploit the structure of the solution algorithm. In this case only the solution modules need rewriting, and only variables in the solution interface need to be passed via parameter passing.

1.4.1 Using ELLPACK as a test bed for distributed problem solving

The aim of this thesis is to investigate issues in developing a distributed problem solving environment for a scientific computing application. We have created a prototype system to study the issues we described earlier in this chapter. We

decided that our prototype system should not only serve as a tool to help us study the pros and cons of a distributed problem solving environment, but also be of use to researchers in scientific computing. As discussed in Chapters 3 and 4, our system provides the user with many powerful tools to aid in problem solving. The specific application domain for our system is the solution of elliptic partial differential equations, but we claim that the approach we take is applicable to a much larger class of scientific problems. Since ELLPACK is a well-known tool for solving elliptic partial differential equations we have created a distributed problem solving environment which makes use of its problem solving modules and grammar, but adds a powerful dimension, namely an interactive environment to aid in problem specification and solution, and a distributed environment to solve the problems specified by the user.

ELLPACK was not originally designed as a distributed system. Still, the "software parts" design technology makes it an ideal candidate for distributed problem solving. As noted above, the various parts communicate via predefined interface state variables. Hence the number of parameters to be passed to the module driver routines on the remote machine typically is not very large. We had a maximum of 15 parameters, consisting of floating point numbers and integers, which needed to be passed.

2 Related Work

This chapter surveys related work in the field of distributed scientific problem solving environments. We begin by looking at various extensions to the ELLPACK language, namely Interactive ELLPACK, Multidomain ELLPACK and XELLPACK. We then look at other recently developed scientific problem solving environments.

With the advent of modern workstations and systems such as the X Window System, a small amount of progress has been made in the domain of scientific problem solving environments. However there still remains a lot of work to be done in this area, and many unanswered questions to be addressed. Also, because most efforts are ongoing, not much has been published yet. We do try however to briefly describe some of the problem solving environments which influenced our approach.

Even though we do not specifically deal with the performance evaluation of parallel numeric algorithms in this work, we feel that it is an extremely important tool in scientific parallel programming, and hence try to give a brief discussion of the recent work done in this area. It may be appropriate to incorporate a performance monitoring tool into our system.

2.1 Extensions to ELLPACK

Since the evolution of the ELLPACK system, several extensions to the original system have been developed. Four such extensions are discussed in this section.

2.1.1 Interactive ELLPACK

Interactive ELLPACK is an extension to the ELLPACK system developed by Wayne Dyksen and Calvin J. Ribbens of Purdue [Dyks 87]. Interactive ELLPACK adds a few extra language features to ELLPACK, along with modules to implement the added features. The extended language features allow the user to interactively

build grids, choose solution methods, and analyze results. Interactive ELLPACK is a reasonably good example of an interactive problem solving environment. The system allows a user to interact with his program at runtime, by allowing him to change grids, and even discretization and solution methods, as long as he had included those methods at problem definition time.

Interactive ELLPACK provides graphics tools for viewing and manipulating solutions. It supports 2-D and 3-D plotting routines. The system was ported to only a few platforms: Tektronix graphic displays, SUN, and Ridge workstations. Porting Interactive ELLPACK is a non-trivial problem as the low level device dependent graphic routines in the system have to be rewritten for each new hardware platform.

One of the drawbacks of the system is that it does not provide the user with interactive tools to help him define his problem, and how he wants it solved. The user is assumed to have a knowledge of the extended ELLPACK language. The user is expected to define the problem, as well as the discretization, solution and output methods he needs, in a program written in the ELLPACK language. Menus are specified in a menu segment, a language feature added to the ELLPACK language. Program development is not interactive, and from this point of view, Interactive ELLPACK still suffers from the drawbacks of batch oriented ELLPACK. Another drawback is that the system only runs on a single machine, which imposes limitations on the classes of problems one can solve. No attempt was made to create a distributed problem solving environment, which we believe to be ideally suited to the kind of problems ELLPACK is designed to solve.

2.1.2 Multidomain ELLPACK

Multidomain ELLPACK was developed by Calvin J. Ribbens at Purdue University, see [Ribb 86]. It is an extension to Interactive ELLPACK. It is based on the idea of *domain mappings*, wherein a partial differential equation problem is replaced by an

equivalent problem which is easier to solve, with a new unknown and a transformed equation and domain, using a change of variables transformation. After computing a solution to the transformed problem, the transformation is undone to get results in the original domain. Multidomain ELLPACK provides various adaptive grid schemes to implement domain mappings, as well as so-called regularization mappings to transform problems on certain irregular regions to problems on rectangles. DPSolve uses some of the language extensions of Multidomain ELLPACK, as well as its preprocessor.

2.1.3 XELLPACK and //ELLPACK

XELLPACK [Bono 88] is a reimplementation of Interactive ELLPACK using the X Window System. XELLPACK provides the same features provided by Interactive ELLPACK. Since the graphical interface is based on the X Window System, XELLPACK can be ported to any machine which runs X, which makes it more accessible than Interactive ELLPACK.

The //ELLPACK project, currently underway at Purdue University [Hous 90] is attempting to develop parallel versions of ELLPACK to run on various architectures. While it provides good post processing tools for output visualization, it does not provide extensive interactive tools for program development.

2.2 Other related systems

In this section we describe other related systems recently developed, or under development. Most of these systems use the X Window System for user interface and device independent graphics development. They are designed to solve numerical problems, or analyze the performance of parallel numeric algorithms.

2.2.1 CLAM and CLAMSHELL

CLAM (Computational Linear Algebra Machine) and the CLAMSHELL interactive user interface are currently being developed by David E. Foulser of Yale University and William D. Gropp of Scientific Computing Associates. See [Foul 90] for a detailed description.

CLAM is an environment which provides user access to numeric computational and graphics tools. CLAMSHELL is an X Window System based interface to CLAM, designed to make CLAM easier to use by research scientists, allowing them to concentrate on problem solving rather than implementation.

CLAM is a programming language with many interesting features. It provides three data storage formats for real and complex arrays. Arrays can be stored in dense, banded, or sparse format. Selection of format is under user control, or can be selected automatically by CLAM. Matrix operations in CLAM are written independent of the manner in which the data is stored. For example, A*B is used to specify the multiplication of matrix A by matrix B, regardless of the way in which A and B are stored. One can thus benefit from sparse matrix algorithms and data structures in CLAM, without having to concern oneself with explicit sparse matrix programming.

Another interesting feature in CLAM is the ability to call "foreign subroutines" written in C or Fortran. In this way a user can link in his favorite library of optimized or parallelized code very easily. This approach makes CLAM easily extendible.

CLAM makes use of LINPACK, EISPACK, and various public domain FFT routines for numeric computation. It also provides a portable X Window based graphics environment. This environment includes a number of plotting routines, and includes the ability to display 3-D solids.

CLAMSHELL is the user interface to CLAM, and assists the user in developing and viewing the results of CLAM programs.

A drawback of CLAM is that after a user has specified his problem he has no means of interacting with the problem solving process, by having the ability to execute parts of his program, look at intermediate results, and then based on these results decide on executing other segments of code. The designers of CLAM have not addressed the broader problem of creating a distributed problem solving environment for computational numerical linear algebra applications. Both CLAM and its interface CLAMSHELL run on a single machine running the X Window System.

Other related uniprocessor based problem solving environments include MATLAB [Matl 89], an interactive software package for scientific and engineering numeric computing, symbolic computing environments like MATHEMATICA [Wolf 88], and MAPLE [Char 90]. Various visualization environments for scientific computing have been developed recently of which apE [Vand 90] is good example.

2.2.2 The SHMAP tool for visualization of parallel matrix algorithms

The Shared Memory Access Pattern (SHMAP) program was designed by Jack J. Dongarra et al. [Dong 89]. It is an aid in analyzing the performance of parallel matrix algorithms, designed for shared memory machines. It consists of two distinct components: the SHMAP1 tool and the SHMAPA tool.

The SHMAP1 preprocessor takes as input a Fortran program and inserts code calling SHMAP1 library routines whenever it sees a reference to a matrix element. When the preprocessor output is compiled and linked to the SHMAP1 library, it executes the original code and generates an ASCII file which contains information detailing how arrays in the program have been referenced.

Once the trace file has been generated, it is analyzed with the SHMAPA tool, which generates a visual display of the information in the trace file. The SHMAPA tool operates under SunView or X11, and thus can easily be ported to many different platforms. The SHMAPA tool allows a user to study memory access

patterns, different cache strategies, and the effects of multiprocessors on matrix algorithms.

2.2.3 The HYPERMON system

Malony et al. [Malo 90] are currently working on a tool to study the performance of message passing parallel architectures. It consists of two components, a data collection component and a visualization environment.

The GNU C compiler is modified so that it emits instrumented code for the Intel iPSC/2, by inserting calls to monitoring functions before and after each procedure call. At execution time these monitoring functions pass the procedure's name, and entry and exit events to NX/2, the operating system on each node. The application events are then merged with operating system events: message, process, and system call. The NX/2 operating system source code was modified so that the operating system events could be captured. The NX/2 operating system then records both application and operating system events in each node's memory. After the application completes, the individual node traces are sent to the host for processing. The authors are currently trying to implement a hardware monitor, HYPERMON, that will allow each node to directly send each event to the monitor by writing to an I/O port in the Intel 80386. The monitor will then capture and time stamp each write.

The visualization environment consists of an X Window based system, with a number of tools to visualize trace event data. This includes widgets for dials, bar charts, matrix views, and a general graph display, used for both procedure call graphs, and hypercube topological views.

Other related environments which provide tools for the development of of parallel programs, i.e., integrated program restructuring editors, compilers, and parallel debuggers and performance evaluation tools are SCHEDULE [Dong 86] and BUILD an X Window System based interface to SCHEDULE, [Brew 89] which

aids in development of parallel Fortran programs for a variety of parallel machines, Faust [Guar 89] an environment for development of parallel programs in Fortran and C for the Alliant FX/8, POLYLITH [Purt 88] an environment for prototyping message passing parallel programs, PARASCOPE [Call 88] an environment for developing parallel Fortran programs for the IBM 3090 and Sequent Symmetry.

3 User's Guide to DPSolve

This chapter describes DPSolve from a user's point of view. Section 3.1 describes the problem definition interface, and Section 3.2 describes the module selection interface. These two interfaces are used in session building. Section 3.3 describes tools available to the user at session execution time. Section 3.4 gives an example of the system at work in solving a PDE.

DPSolve uses parts of the Multidomain ELLPACK software, i.e., its preprocessor and some problem solving modules. Though it is X based, it does not use any of the XELLPACK or Interactive ELLPACK software. DPSolve cannot be viewed as just as an extension to ELLPACK since it represents a completely new approach to scientific computing, i.e., the creation of an integrated distributed problem solving environment, not seen in any of the current ELLPACK extensions. DPSolve can be viewed as a prototype system for scientific computing in general since its design will scale to a much larger class of scientific problems than the solution of PDEs.

DPSolve allows the user to create his own problem solving environment, and then using the tools available in this environment, investigate many possible approaches to solving his problem. We refer to these two phases as session building, and session execution.

The session building phase consists of a powerful tool which enables a user to interactively define his problem to the system, and select possible tools for solving it. He is aided by a number of dialog boxes which appear in response to his selections, as well as menus listing all the possible selections available. This phase consists of two sub-components: the problem definition interface, and the module selection interface.

3.1 Problem definition interface

The problem definition interface (see Figure 4) provides the user with an interactive way to specify the partial differential equation and its domain and boundary conditions. We also allow the user to define his own menu items in this phase. We now describe in detail the features provided by each button in this interface.

Option

Option allows the user to find out the memory storage requirements for the solution of the problem. This information is generated by the ELLPACK preprocessor. Other options allow the user to set the output level, which determines the output to be printed, to set the minimum and maximum dimensions of the workspace array, and to determine the time taken by each module.

Equation

Equation allows a user to define a PDE to DPSolve. There are 3 options available to the user.

Retrieve from file: The *retrieve from file* option prompts the user for the name of a file where the equation is defined.

New: The *new* option asks the user to enter the equation in a dialog box.

Example: The user can use the example we have provided. This option can be used when learning to use the system.

Boundary

Boundary allows the user to specify the domain of the PDE and its boundary conditions. The boundary can be specified in the same three ways as the equation, namely in a file, by interactively typing it in at the keyboard, or by using our

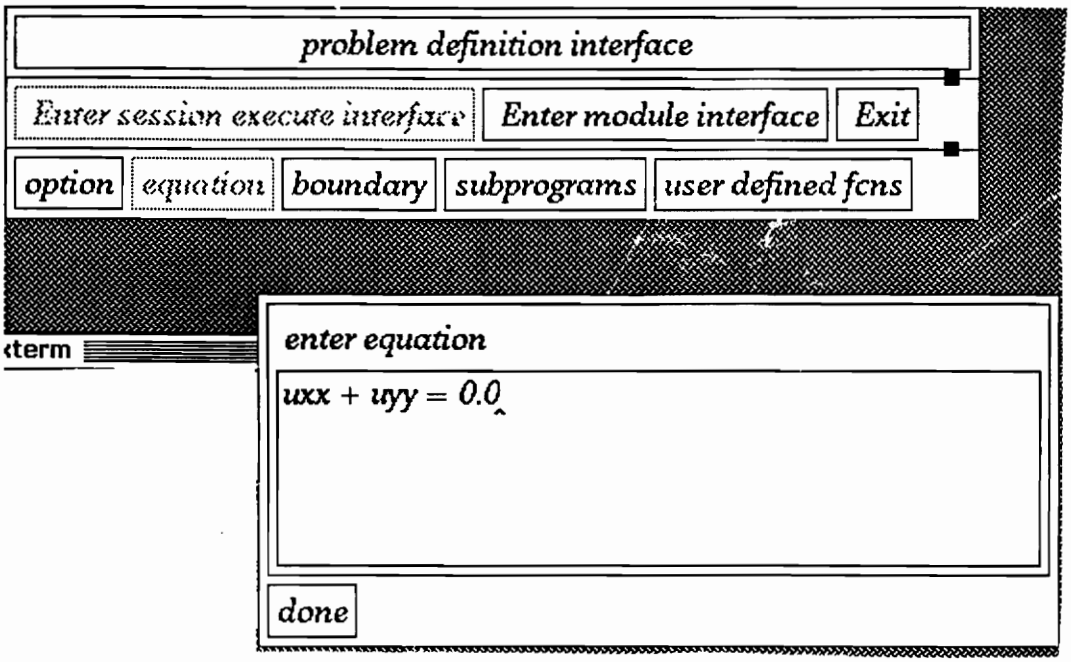

Figure 4: A view of the problem definition interface.

example.

Subprograms

Subprograms allows arbitrary Fortran subprograms to be included. Good examples are subprograms which define complicated PDE coefficients or the true solution.

User defined fcns

User defined fcns provides the user with a tool to create his own menu items. This provides an advanced user a way of crafting an interface to suit his needs. The user is allowed to define his own menu names, and interactively is prompted for the name of the file containing the action to be performed, and the name to be given to this action. These actions are executed remotely when the user selects the appropriate menu item at session execution time. The current system restricts the user to using Fortran subroutines to specify the actions, but it is easily extensible to C as well.

The user is expected to use correct ELLPACK syntax to specify the equation and boundary (see Rice[85] for a description of the equation and boundary syntax). The syntax is very similar to standard mathematical notation. The user is expected to have entered an equation and boundary condition before attempting to enter the module selection interface. If not, a warning message asking for this information appears on the screen, and the user is not allowed to enter the module selection interface until this information is provided.

3.2 The module selection interface

This interface is designed to allow a novice user to select ELLPACK modules. Please refer to Figure 5. We provide the user with a user friendly menu driven interface to a large number of ELLPACK modules which can be categorized as

follows :

GRID modules. Specifies a set of vertical and horizontal grid lines.

DISCRETIZATION modules. Specifies a module to create a discrete approximation to the elliptic problem.

ADAPT modules. These modules are used to perform adaptive grid domain mappings.

INDEXING modules. Specifies a module to reorder the linear equations and unknowns.

SOLUTION modules. Specifies a module to solve the linear equations.

OUTPUT modules. Specifies ELLPACK-generated output.

At present, we have provided a representative set of ELLPACK modules from each of the basic categories. Extending DPSolve to include new modules is an easy task however, and is described in the next chapter. The various in the module selection interface are now defined.

Grid

The *grid* button pops up a dialog box which prompts a user to specify the number of x and y grid lines. The grid dimensions should be at least 3 x 3. The PDE is discretized on the user selected grid. At present, ELLPACK allows only rectangular grids.

Discretization

The *discretization* button pops up a drop-down menu containing all the possible modules the user may use to discretize his problem. To select a module all the user has to do is select the corresponding menu entry using the mouse. The discretization menu provides the user with the following methods: *five point star*, *hermite*

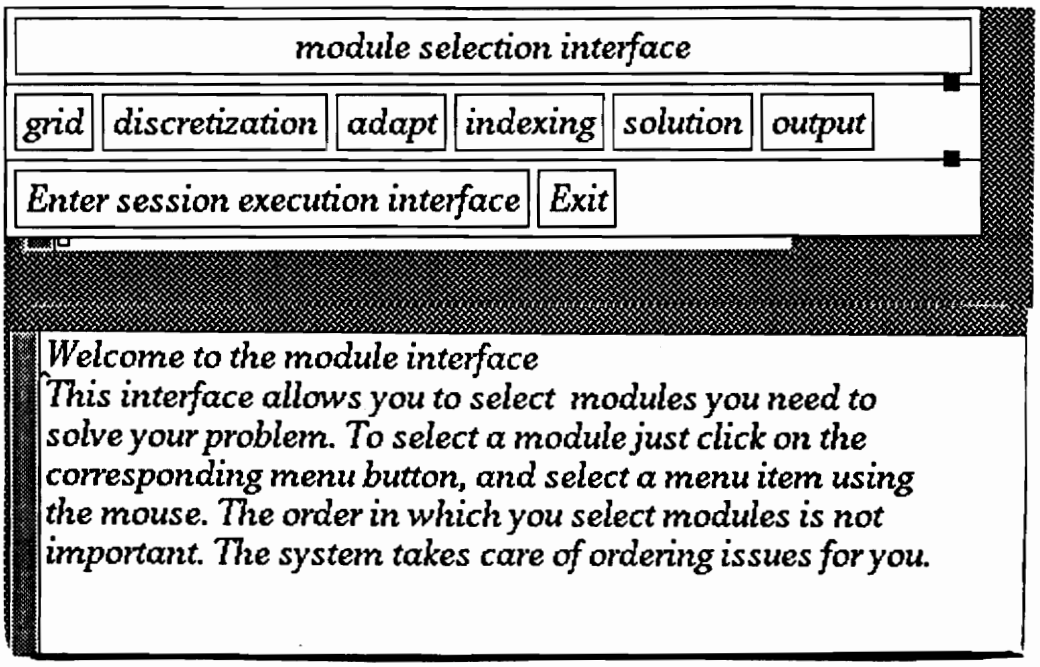

Figure 5: A view of the module selection interface.

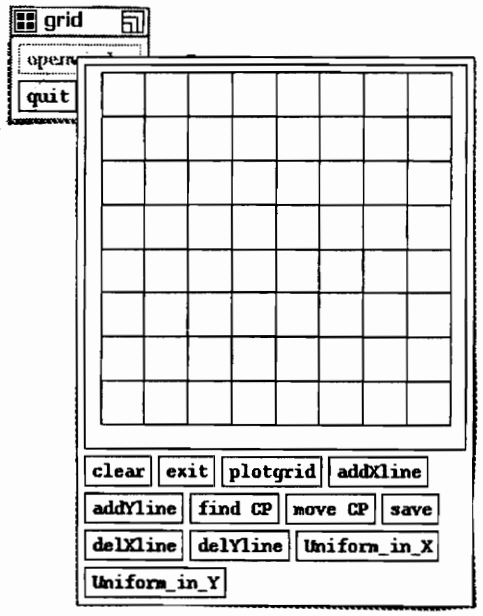

Figure 6: The *interactive grid* tool allows the user to modify the grid defining an adaptive transformation.

collocation, *spline galerkin*, and *interior collocation.*

Indexing

The *indexing* button pops up a drop-down indexing menu. The indexing menu provides the user with tools to reorder the system of linear equations. We allow the user to leave the system *as is*, or reorder it using the *red-black* reordering scheme.

Adapt

The *adapt* button pops up a drop-down adapt menu. The adapt menu provides routines which construct adaptive domain mappings. See [Ribb 86] for a more detailed description of grid adaptation methods. Options include *global* and *interactive grid.* The *interactive grid* tool allows the user to view and modify the grid which defines an adaptive domain mapping. Figure 6 shows this tool. It can be used to create a new mapping grid, or modify an existing grid. We provide

a number of options to help the user modify the grid. Grid lines are added by selecting the *addXline* or *addYline* buttons, and then clicking the mouse button in the grid window at the position where the line is to be added. Deleting lines works in a similar fashion. The *move CP* button allows the user to move grid points, by clicking the mouse button at the (x,y) position where the nearest grid point is to be moved. The *Uniform_in_X* and *Uniform_in_Y* buttons allow the user to make the grid uniform in the x or y directions, respectively.

Solution

The *solution* button pops up a drop-down solution menu, which provides the user with various direct solvers, namely *linpack band*, *band gauss elimination*, *linpack spd band*,*band ge (no pivoting)*, as well as iterative solvers from ITPACK, like *sor*, and *jacobi cg*.

Output

The *output* button pops up a drop-down output menu (see Figure 7) which provides the user with several different ways of analyzing results. The user is provided with tools to tabulate the values of the computed solution and its derivatives, as well as the values of the error and true solution (if available). He can also find the maximum of the above on a specified grid. The plotting routines provide the user with color 2-D wireframe plots of the solution and its derivatives, as well as of the error.

3.3 The session execution interface

Once the user has chosen the modules he needs to build his own problem solving environment, he can enter the session execution phase. Figure 8 shows the session execution interface. This interface basically consists of the tools selected by the user in the session building phase. In this way a user can build his own environ-

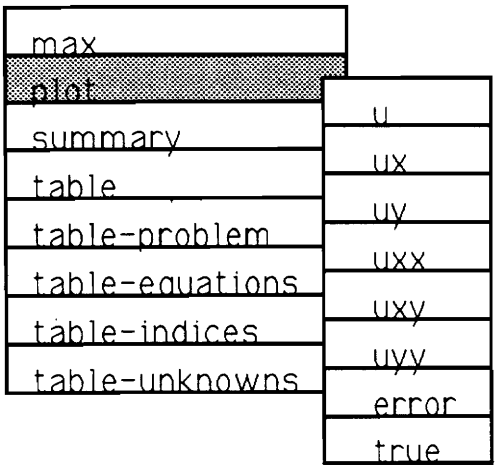

Figure 7: A view of the output menu.

ment in which various approaches to solving a problem can be studied. Notice that only the menus the user has selected earlier will appear in the session execution interface. Also, if only a few of the possible menu items have been selected to perform a particular task, the user is only provided with those items at session execution time. For example, if only *band ge*, *linpack band* and *jacobi cg* modules were chosen to solve the linear system generated by discretization of the PDE, at session execution time the solution menu will only contain the afore-mentioned entries. Figure 9 depicts this example. Below, each of the buttons in the session execution interface is defined.

Display prog

Display prog displays the ELLPACK program DPSolve has created based on the user's input. The user need not be concerned about this program. We have provided this button only to enable the user to browse through the program if he so wishes, and check if there are some more options he wishes to add to it. The role of

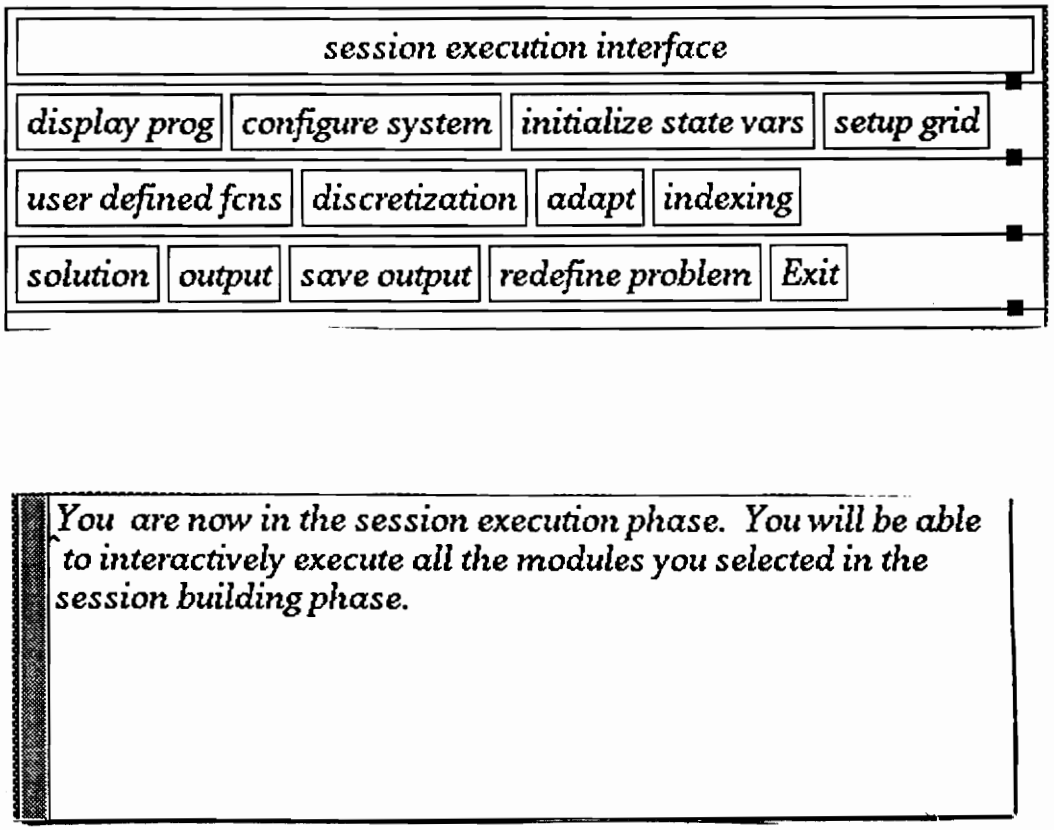

Figure 8: A view of the session execution interface.

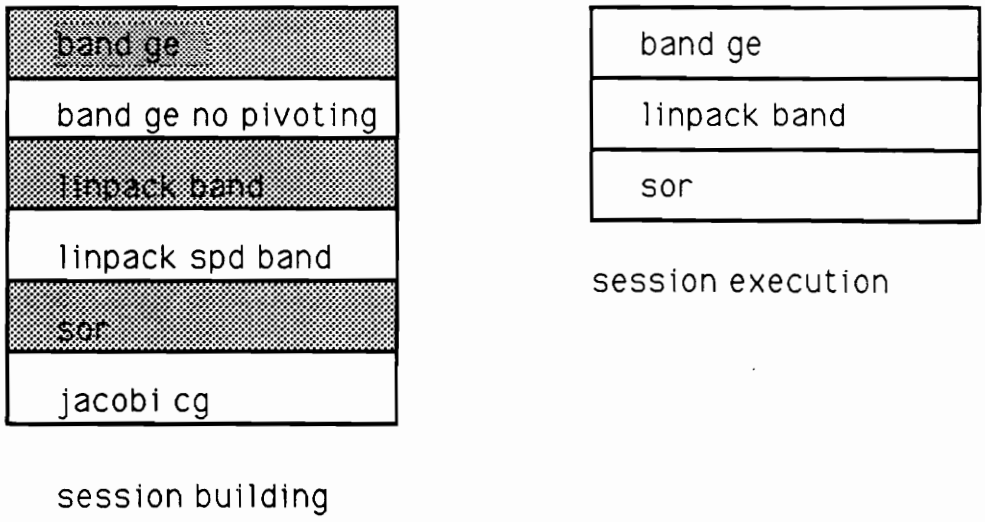

Figure 9: A view of the Solution menu at session building time and at session execution time, assuming the user chooses *band ge*, *linpack band* and *sor* at session building time.

this program is described in Chapter 4. The program is displayed in a text widget with horizontal and vertical scrollbars to enable a user to easily scroll through it. The user is not allowed to edit the program. If he wishes to make changes he has to go back to the problem definition or module selection interface and make the necessary changes.

Configure system

The *configure system* button builds the system, i.e., compiles user defined action procedures, calls the ELLPACK preprocessor to generate code to define and initialize ELLPACK state variables, and links in appropriate rpc libraries. All this is transparent to the user. The user need not even that the preprocessor is running, or that his program is being distributed so that the numerically intensive portions will be executed on a parallel computer. For a more detailed description

of what happens at system configuration time the interested reader is referred to Section 4.2.

After configuring the system the user can begin to interactively use the tools in the environment he has built. If the user forgets to configure the system, a warning message is displayed when he tries to execute any of the other tools in the session execution interface. He is not allowed access to the other tools until the system is configured.

Initialize

The *initialize* button initializes the ELLPACK state variables on the remote machine. The user is required to initialize these variables before proceeding, and is prevented from advancing until he does so.

Setup grid

The *setup grid* button defines a grid on which the problem will be discretized.

User defined fcns

The *user defined fcns* button pops up a drop-down menu containing the list of user defined tools. By selecting an item in this menu, user defined functions are executed remotely on the remote machine.

Using other DPSolve modules created in session building phase

The *discretize*, *adapt*, *indexing*, *solution* and *output* menus contain selections made by the user at session building time. Selecting a menu entry results in the execution of the remote procedure associated with the menu entry.

Redefine problem

The *redefine problem* button allows the user to re-enter the problem definition in-

terface and change some or all of the specifications of the problem. For example, he can define a completely new problem by changing the boundary conditions, and grid size. Or he can change the PDE itself. He can then re-enter the session execution interface, and reconfigure the system.

Save output

The *save output* button allows the user to save the results of the ELLPACK run in a file. These results are normally printed in an output window by the *xview* tool described below.

Exit

The *Exit* button ends the session.

We have also provided tools to enable a user to view the intermediate results of his selections. The *xview* tool monitors the ELLPACK output file on the remote machine and whenever this file is updated prints the newly written text in a window on the local workstation. Figure 10 shows a snapshot view of the *xview* tool in action.

The *xview* tool is designed to function as an independent X program, and can be run in the background by the user at any time. To run the xview program at the command line type: xview &.

3.4 Example

We now present an example wherein we solve an elliptic partial differential equation using DPSolve. The example is Example 3.1 from [Ribb86]. We solve a Poisson problem $u_{xx} + u_{yy} = f$ on the unit square, with Dirichlet boundary conditions, with f chosen so that the known true solution is

$u(x, y) = e^{100((x-0.5)^2+(y-0.117)^2)}(x^2 - x)(y^2 - y)$. We use the following methods

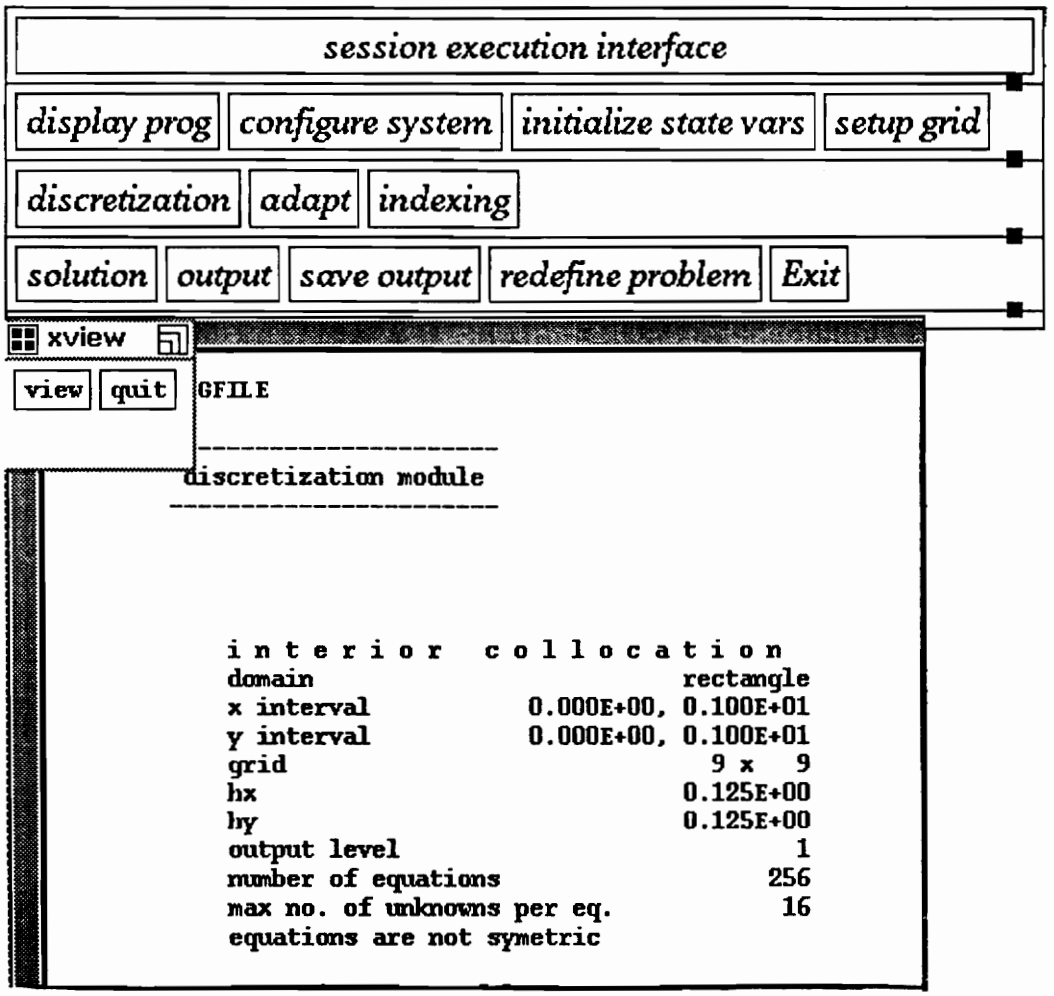

Figure 10: The *xview* tool shows the latest update to the ELLPACK output file.

to solve the problem.

1. Fourth-order accurate collocation with Hermite bicubic basis functions, (ELLPACK module *interior collocation*), followed by a band gauss elimination linear solver.

2. Adaptive mapping defined by the *global* method developed by [Ribb 86]. A 9×9 mapping grid is generated by the *global* method. Figure 11 shows the mapping grid generated by *global*. The problem is then solved in the solution domain using *interior collocation*, and *band ge*.

Figure 12 shows the plot of the error function when no adaptation is used, and Figure 13 shows the plot of the error function when the *global* method was used for grid adaptation. The markings at the adjacent rectangular edges represent points on the x and y axes respectively. These figures are monochrome versions of actual color plots. Height along the z axis is coded by color.

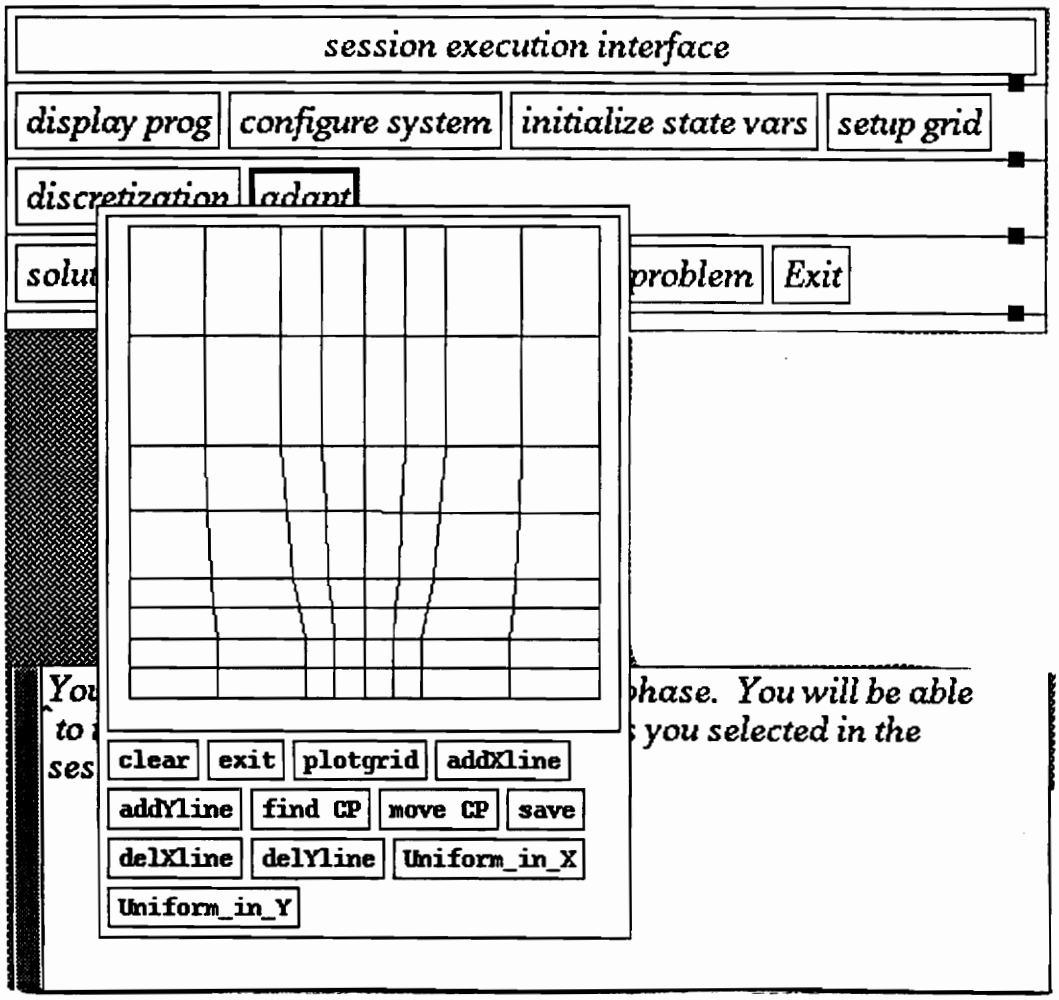

Figure 11: The 9×9 mapping grid generated by the *global method.*

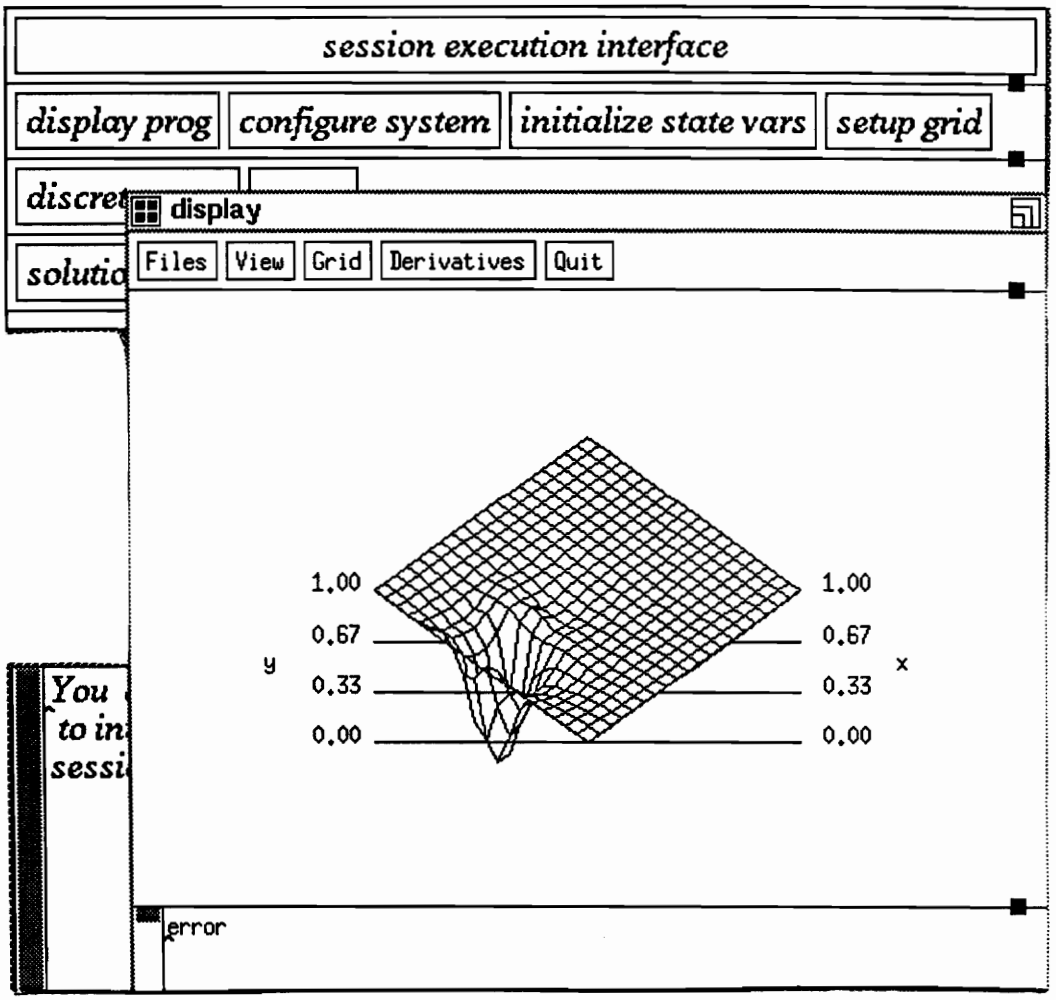

Figure 12: The error function after non adaptive solution. The maximum absolute error is 2.98E-03.

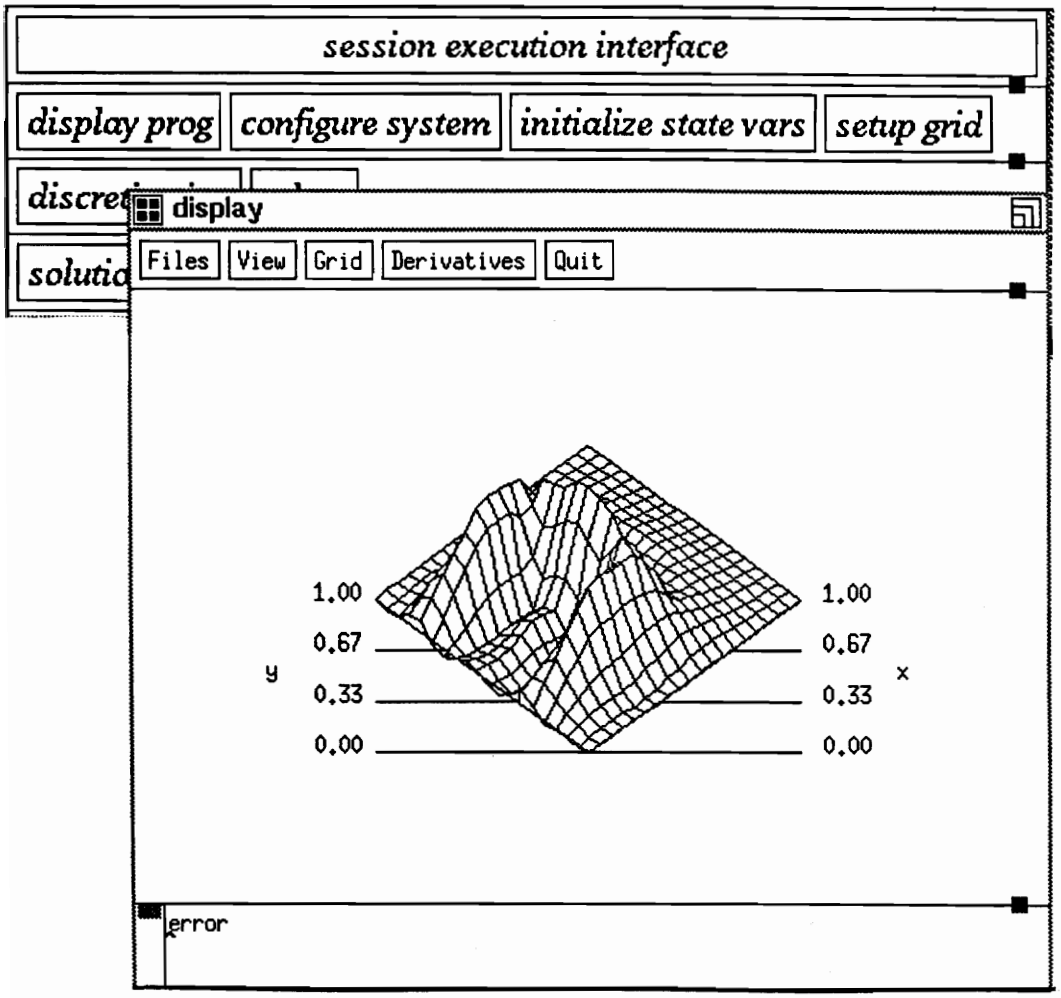

Figure 13: The error function after adaptive solution. The maximum absolute error is 2.00E-04.

4 Design and implementation of DPSolve

This chapter discuses DPSolve from the point of view of the designer and implementor. DPSolve consists of two parts: a front-end and a back-end. Section 4.1 describes the front-end and Section 4.2 describes the back-end. Figure 14 depicts the overall design of the system.

4.1 The front-end

The front-end is based on the X Window System. It contains all the code necessary to define user interface objects like dialog boxes, menus, command buttons, etc., as well as code to configure these objects when mapped to the display. It also contains interactive tools to view results. The duties of the front-end can be summarized as follows :

- helping the user in problem definition.
- providing a menu driven interface to discretization, adapt, indexing, solution and output modules.
- passing parameters to the back-end.
- providing an interactive run time environment for PDE solution and solution visualization.

Section 4.1.1 describes how we configured the user interface objects. Section 4.1.2 describes the approach we used to communicate user events to the interface. The remainder of this section briefly mentions the tools used to develop the front-end.

The front-end is implemented for X11 Release 4 systems. We have used the X Toolkit, as well as Xlib routines to implement the interface.

The X Toolkit consists of the Xt Intrinsics and the Athena R4 widget set. Xt is a set of intrinsic functions built on top of Xlib, the lowest level programmable

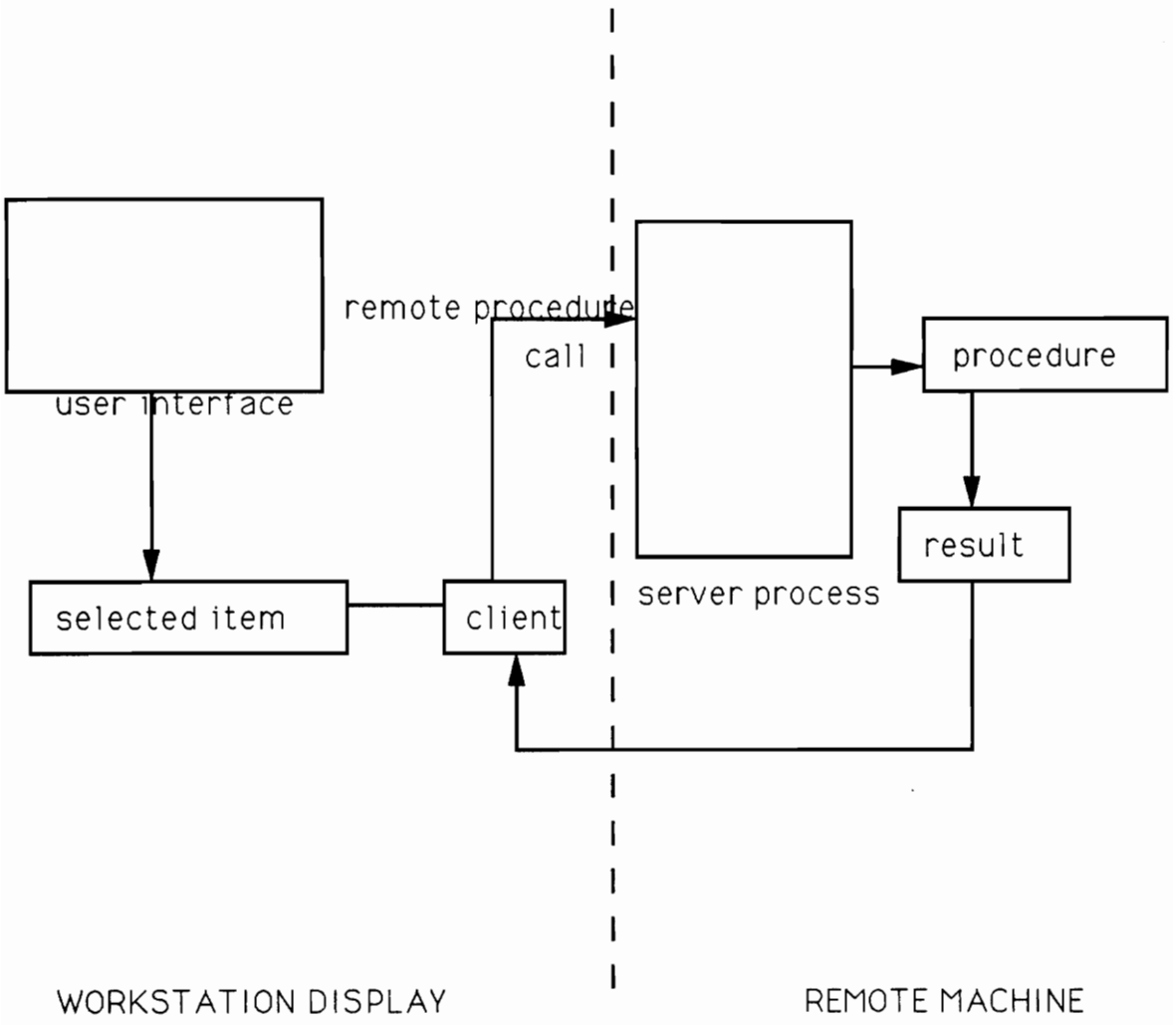

Figure 14: The design of DPSolve.

interface to X. It provides tools for the development of reusable objects called widgets.

The Athena R4 widget library consists of several different *classes* arranged in a hierarchical manner. The members of a class share the same properties, which distinguish them from members of another class. Each time a user creates a widget he creates an *instance* of one of the Athena widget classes. A widget inherits all the properties of its superclass, i.e., the set of all ancestors in the hierarchical class tree, in addition to the properties of its class.

4.1.1 Configuring widget resources

Each widget class contains a predefined set of resources which are user configurable. A resource is a widget variable which is application configurable. As an example, consider the way menus are created in our application. Each menu contains a number of entries called menu panes. We use widgets of the bSBMenuEntryObject class to create a menu pane. Each pane must have a different title. The bSBMenuEntryObject class has a resource called XtNstring which defines the menu pane title, and is application configurable. At each instantiation of a member of the bsBMenuEntryObject class we set the XtNstring resource to the appropriate value.

There are two ways in which widget resources can be set using Xt. We have used both for reasons that are explained below. The first way is by hardcoding resources in the application program using Xt Intrinsics calls.

The second way to configure a widget's resources is by placing resource values in a resource file called an application-defaults file, which is read in by the *resource manager*[1] which sets the widget's resources to the values defined in the application-defaults file.

[1]The *resource manager* is an Xlib program which merges a database consisting of several ASCII files which might define the applications resources and values hardcoded by the application, and then determines a unique value for each resource of a widget.

The advantage of using the second approach is that a user with no knowledge of X can alter the look of the interface. Also from an application programmer's point of view, it saves time since he does not have to recompile his entire application each time he wants to experiment with different widget configurations.

There are certain resources however which we do not want an end user to have control over as he could endanger the system's performance by naively resetting them. These resources have been hardcoded in the application code using the Xt Intrinsics calls described above.

4.1.2 Widget-application interaction

Widgets can function as independent entities, but would not be of much use in building an interface if used in that way. Xt provides ways for widgets to interact with application code. We have used *callback functions* and *action procedures* to interface widgets with application code. A *callback list* is a hook that allows application defined procedures called *callback functions* or *callbacks* to be invoked when some widget-specific condition occurs. An application can associate a callback function with the callback list provided by a widget class using the XtAddCallback or XtAddCallbacks call. Under Xt, callback lists are widget resources, which means that the application can easily set or change the callback function to be invoked.

The other method used for widget-application interaction is the *action* routine. This mechanism allows designated routines to be invoked in response to certain *events*. There are 33 different events that the X server recognizes. An X event is a packet of data sent by the server to the client window that requested it. These events are generated in response to mouse and keyboard activity, or changes occurring due to changes in the windowing configuration, e.g., by resizing, moving or raising windows. In order to receive a particular event from the event queue, a client window must select the event in its translation table. Each widget can define

its own translation table, wherein it can map events to actions. Xt's translation manager selects the specified events and invokes the associated action procedure. If an event is generated in some window that does not select it, it is passed upwards to each of the widget's ancestors until it finds a window that recognizes it.

The major application of actions is adding features to widgets other than those provided by the widget designer. We used action procedures to modify the behavior of a few widget classes, the most notable being the dialogWidget class.

4.1.3 Other functions and tools

To aid in development, we developed a library of functions above the X Toolkit. These functions perform often used tasks such as creating various interface objects (e.g., menus, ASCII text and disk widgets, dialog boxes), along with the appropriate callback functions. These routines also contain code necessary to do scaling, mapping and correct placement of these objects relative to the window of the calling routine. If an application programmer wishes to use these routines, all he needs to do is modify the callback functions in our library.

For example, the CreateMenu function in our library takes as parameters the ID of the menu button widget (the widget which when selected, causes a drop-down menu to pop just below itself), a list of entries, and the number of entries. It creates the drop-down menu containing these items, and performs the necessary tasks to display the menu when the menu button is pressed by the user. Most importantly it decides which callback function to associate with each menu item, depending on whether the menu is going to be used at session building or at session execution time. If the menu item is selected at session building time, the menu item name is stored in a list containing all the menu items selected by the user. When the session building phase is over, at the time of creation of the session execution interface, a menu button with the name of the menu is created, and CreateMenu is invoked again to create new menu entry widgets corresponding to the items

kept in the list. In this way we prevent the user from accessing modules he has not selected at session building time. If no menu items from a menu appearing at session building time are selected, then this menu does not appear at session execution time. For example, if the user does not want to use grid adaptation, then when a scan of the list associated with this menu shows that it is empty, no menu button corresponding to adaptation is created.

Similar routines were written to create other user interface objects. This simplified the implementation process, whilst avoiding unnecessary code duplication.

We also implemented a tool called *xview* which is used to view on the workstation monitor the computational results returned by ELLPACK modules running remotely on the host machine. This tool makes use of the features provided by X to obtain a file as well as monitor it for future activity. Incidentally, these X functions can also be used for pipe and socket input, since under UNIX, these are variations of files. Thus, with minor modifications the *xview* tool can incorporate the use of socket as well as pipe input.

The *interactive grid* tool makes use of low level Xlib function calls to plot the mapping grid. The grid is scaled to the size of the display window. It is then plotted in this window. If the user wishes to modify the grid using the mouse, the position of the cursor when clicked can be obtained from the *event* structure returned by the server to the client in response to a mouse button click event. The smallest polygon enclosing the mouse button position is determined, following which the vertex nearest to the position of the cursor can be obtained. If the user desires to move a grid point to the cursor position, we update the coordinates of the nearest vertex to those of the cursor position, and redraw the grid.

4.2 The back-end

The back-end of our system consists of a library of client routines on the user's workstation and a suite of remote procedures on the remote machine. We have

used the SUNRPC software to implement remote procedure calls. The rpcgen compiler provided by SUN was used as a tool in writing client and server stubs, which were then linked with the client and server routines respectively.

Once the user has defined his problem to the system, an ELLPACK program is generated using the information supplied by the user. This ELLPACK program contains the problem definition and a list of the modules selected by the user at session building time. The ELLPACK preprocessor is then invoked to generate declarations of the ELLPACK state variables, based on this ELLPACK program. The ELLPACK preprocessor also generates code to initialize these variables with values supplied by the user at session building time. The preprocessor also generates functions which define the problem, e.g., functions which represent the PDE coefficients, the right hand side, boundary conditions, etc. This code is transferred, along with the declarations and initializations of the ELLPACK state variables, to the remote machine via NFS. This step is necessary since the dimensions of various work arrays depends on the grid size specified by the user and modules selected. The routines which depend on the definition of the state variables are compiled remotely using remote shell (rsh), and linked to our library of remote procedures, the ELLPACK library, and SUNRPC library to form an executable file, which is then executed (using rsh) on the remote machine and run in the background. This is the server process.

If the system is reconfigured, a shellscript on the remote machine is executed which kills the server process. The steps detailed above are then carried out resulting in the execution of a new server process.

The selection of an item in the session execution phase results in a client procedure being invoked. The data passed to the client from the session execution interface is converted to External Data Representation (XDR) format using the XDR library on the workstation. This data is then passed to the remote machine, where it is deserialized (converted from XDR format to the host machine's data

representation format), and passed to the remote procedure which on completion of execution passes a result back to the client, and control is transferred back to the session execution interface.

The result passed back to the client reports the success or failure of the rpc call. In the case of failure the client, based on the value of the result variable, does some error checking to see why the call failed and reports the results of its findings back to the user. If the remote procedure on the host machine dumps core then the server process is terminated. The RPC connection is broken, and a system error ("RPC: remote system error - Connection refused"), causes the clnt_call routine in the RPC library to terminate with an error condition. This terminates the process running on the workstation. We have tested the remote procedures we wrote to try to ensure robust behavior. It is the user's responsibility to write reliable code implementing user defined functions. If the user supplied code for user defined functions causes a core dump, our system cannot be responsible for error recovery.

The computational results generated by ELLPACK modules are written to a log file. The *xview* tool on the client machine monitors this file and each time it sees something new being written to it, displays the newly written portion of the file in a text window on the workstation display. This allows the user to immediately observe the results of his actions.

In the case of plotting routines, a remote procedure is invoked on the remote machine to generate a data file to be used for display. A tool developed at Purdue by the //ELLPACK group, see [Hous 90] is executed on the workstation display which reads this file using NFS and generates wireframe images on the workstation monitor.

Our system also allows a user to extend it by defining his own set of menu items which perform tasks he wants his environment to provide. At session building time we prompt the user for the name of the file containing code to perform a task he needs, as well as the name of the menu item he wants to associate with

this task. The user is restricted to only using the variables in the ELLPACK state. At present, we expect the code to be written as a Fortran subroutine. No parameter passing is supported. The reason for this is that the procedure is to be executed remotely using an RPC made by a pre-compiled client procedure in the client library. In order for parameter passing to be supported, client routines would have to be automatically generated for each user defined procedure, and then compiled. A client and server stub would also need to be generated. One major problem with this approach would be that it would change the declaration file where the definitions of parameters to be passed to remote procedures is kept at every session. Client and server routines depend on this file and would have to be recompiled at every session. This would make the system unbearably slow. The restriction we have made still provides the user with the ability to manipulate any variable contained in the ELLPACK state on the host machine, whilst configuring the system reasonably fast.

The file containing the task definition is scanned and the subroutine name is replaced by a system specific name, by which the system will know the subprogram. The system builds a map which associates an integer key with each user defined menu item entry. All user defined functions are then stored in a single ASCII file which is passed to the remote machine via NFS along with the definition of the ELLPACK state and compiled there. When the user selects an item from the user defined menu at session execution time, the key corresponding to the item is retrieved from the mapping table and passed to a client procedure, which passes this value to a remote procedure on the host machine, which then knows which user defined function to start up. These extensions are local to a given session, i.e., they are lost when the user builds a new session.

In order to add permanent new features to the system, the systems programmer seeking to extend the PSE has at his disposal a variety of high level interface tools written for creating interface objects like dialog boxes, menus, text entry and

display areas, etc. We have written a special set of callback and action procedures for each of these objects, which with the following exceptions should not be altered. Only the following functions need to be altered: MenuSelect in menumgr.c, SessionExecute in callbacks.c and Callrpc in menumgr.c.

Assume a new menu **newmenu** needs to be added. Let se_num_menu_items denote the number of menu selections by the user during session building (the prefix se_ indicates that the variable will be used during session execution); se_newmenu_items denotes a list where these selections are kept; data_1, ..., data_n denote data objects to be passed to the client procedure newmenu_cl. We outline the additions to be made to the above functions.

```
MenuSelect(...)
{
...
    if newmenu is selected {
        Save name of the menu item in se_newmenu_items;
        Generate ELLPACK language statement;
        se_ num_newmenu_items++;
    }
}

SessionExecute(...)
{
...,
    CreateMenu(se_newmenu_items, se_num_newmenu_items);
}
```

```
Callrpc(...)
{
...
  if se_newmenu is selected {
       newmenu_cl(item_num, data_1, ..., data_n);
  }
}
```

Thus, adding a new interface object is not difficult. A software module which conforms to the ELLPACK guidelines, which are clearly stated in [Rice 85] must be supplied. Client and server routines must also be written. The programmer does not have to go through a long learning process to get the communications to work. All he needs to do is copy the client and server procedure which most closely suits his needs and alter it to fit his needs.

5 Discussion and Conclusions

In this chapter we analyse some of the decisions taken in designing the DPSolve system, evaluate them and describe some possible directions for future work. Section 5.1 discusses and critically evaluates these decisions. Section 5.2 draws some conclusions about the feasibility of implementation and usefulness of distributed problem solving environments for scientific computing. Section 5.3 describes directions for future work.

5.1 Design decisions and criticisms

There are a plethora of issues an implementor of a distributed problem solving environment needs to consider. In this section several issues are listed which have the the most bearing on problems of a scientific nature. While this list is not claimed to be exhaustive, it is clear that a distributed problem solving environment designer would benefit by using these guidelines to design his own problem solving environment (PSE).

We summarize the design issues as follows:

Problem domain and need for PSE: Machura [Mach 85] states that a designer of a PSE should first determine that there is a real need for a PSE for a particular problem domain, then decide whether it is worth building. This might be done by identifying users who are interested in the kinds of problems the PSE could potentially solve.

Target community: The community that will be using the PSE should be targeted. Is the system going to be used by experts, novices, or both? It should be understood here that the term "novice" is used to refer to a person who is not computer skilled or is a beginning user of the PSE. Since the application area is the field of scientific computing the end users are likely to be highly

qualified persons possessing at least a good working knowledge of the theory behind the problem solving process.

Utilization of distributed computing resources: The PSE should be able to use various network resources in a manner transparent to the end user. It must also be determined if most users will have an appropriate range of resources easily available.

Unified appearance: Though the PSE will contain may different tools, the end user must not need to know which subprograms are being invoked, or how data is being transferred in response to his requests.

User customization: The user should be able to customize the PSE to suit his needs. This implies that the user be allowed to use the PSE tools to build a "mini system" to solve problems of special interest.

Extensibility: The system should be *open*, i.e., easily extensible. The implementation code should be easily available, and the designer should design the system with an emphasis on allowing other users to easily add to it.

Portability: The system should be easily portable to a wide range of architectures.

Output visualization: The user should be able to visualize the solutions to his problems, rather than have to scan pages of numerical data.

Reliability: This is an important issue when dealing with numeric applications, and becomes all the more difficult to handle in a distributed environment. No user wants an unreliable piece of software, and every effort should be made to ensure that the PSE is reliable.

The application domain for our system is the solution of elliptic partial differential equations. Elliptic partial differential equations are used to model a wide

variety of different physical systems. Advances in a very diverse spectrum of application areas ranging from structural mechanics, atmospheric modeling, electrostatics, to nuclear reactor design have depended on the ability to find quick and accurate solutions to elliptic PDEs. A great deal of research has been done designing algorithms to solve PDEs in recent years. A number of these algorithms have been specially designed with a target architecture in mind, and hence their performance is heavily dependent on the underlying machine. The provision of a distributed PSE for elliptic PDE problem solving would be of great benefit to scientists in the fields mentioned above, as well as to researchers in mathematics and computer science who are provided with a framework to easily test the performance of existing algorithms and find ways to design new methods.

Regarding target community, we felt DPSolve should be of use to a wide variety of users. It should not be aimed only at people who are experts in the ELLPACK language and system, since even though this language is relatively easy to learn, the number of such users is not very large. We decided to design a system accessible to a novice ELLPACK user, or a scientist with the basic idea of the steps involved in solving a PDE. The only prerequisite is the ability to state the PDE and boundary conditions to the system. We have removed most of the restrictions imposed by standard ELLPACK. For example, the ELLPACK language imposes a strict ordering on the definition of ELLPACK segments. If segments are not selected in the right order the preprocessor generates errors. We felt this to be too restrictive in an interactive environment. Our system allows a user to pick modules in any order. If the user has not supplied input crucial to the solution process, the system prompts him via messages sent to the interface. It is the system's responsibility to then assemble all the user-defined selections in a manner understandable to the ELLPACK preprocessor. The removal of this restriction frees the user from having to concern himself with needless language issues in order to specify his problem.

However, we still wanted this system to be of use to expert ELLPACK users.

With this in mind the user is provided with the tools to describe menu items and associate them with procedures he feels are useful in solving the PDE. The expert user is thus provided with a way to access any variable contained in the ELLPACK state definition of the PDE, and change its value.

As stated throughout this thesis, we feel that the days of stand-alone platforms for scientific computing are numbered. Most scientific researchers have available a multitude of powerful computers interconnected by a high bandwidth network. A problem solving environment which is designed for a single computer does not make efficient use of the resources of the network. Our system is designed so that the user interface runs on a workstation running X, whereas most of the computation is done on a more powerful parallel computer. However, this is completely hidden from the user. At installation time the system administrator can set the name of the host machine on which the remote procedures can run. The end-user need not even know that the menu items selected actually invoke remote procedures. To the end-user, the entire system seems to run on a single machine. Thus the system provides a unified appearance to the end user. We feel that this is a powerful feature of our system, which should become a standard for measuring the performance of other distributed problem solving environments.

We felt it important that a user be allowed to define his own problem solving environment using the tools provided to him. Our system allows the user to select only those tools he feels are applicable to his problem from a large selection of choices we offer to him. This selection is done at session building time. Our system keeps a record of the tools the user selects and at session execution time builds an environment consisting of these tools only. This allows the scientific researcher to choose only those PDE methods he is interested in, and perform experiments to study the performance of these problem solving methods on his problem. Thus he is allowed to use DPSolve to build his own "mini environment" suited to solving his particular problem. It should however be pointed out that

the user customizations are lost when the user ends the session.

There are two ways in which a user can extend this system. The first is by defining his own menu items at session building time. Thus he extends the systems capabilities for the course of a session, after which they are lost. The second way is to build new features into the system on a permanent basis, refer to Section 4.2. As seen in Chapter 4, this task is simplified by the interface tools we have built, and the way the system is designed. The programmer can make use of the lessons we learned without having to go through the entire learning process we went through. Thus, needless repetition of work is eliminated.

Another design objective we had was that our system be easily portable to other hardware platforms. For this reason we implemented our interface using the X Window System. The X Window System has become an industry standard and most workstation vendors supply X11 Release 4 with their machines. As a result our interface can run on a wide variety of different platforms. Incidentally, all the software we used is in the public domain. We used the SUN RPC facility for client/server communications. Hence our system does require that the SUN RPC libraries are installed on both client and server machines. This software is available free of charge from many sites on uunet. We found it to be easily portable to our machine.

We also felt that it was important that the user be able to visualize the results of his selections immediately. This would give him a chance to interact with the solution process, and learn from his experiences. We feel that we have provided him with adequate tools to view his results. The *xview* tool enables him to immediately view the results of the remote procedures performing discretization, adaptation, indexing and solution on the host machine. As stated in Chapter 3, we allow him to access a large base of information which can tell him a great deal about his problem. The graphics tools allow him to visualize the solution of his problem.

We feel however that though we have provided the user with powerful visu-

alization tools there is still room for improvement in this area. For example the system could be extended to incorporate contour plots, and 3-D graphics.

Reliability is an important issue in a distributed system. In an interactive environment such as ours this is not an easy issue to solve. The sequence of actions performed by the user during a session is not predictable. We have tried to insure that at session building time the user selects ELLPACK segments which are absolutely essential to defining a problem to the system. One drawback of our system is that it requires a user to enter the equation and boundary definitions in a form understandable by ELLPACK. The ELLPACK format allows a user to specify these conditions in almost the same way as is used in standard mathematical notation, which we assume our users to be familiar with. Hence we feel this was not a big disadvantage. We have tried to insure that the user correctly defines his problem to the system, and only allow him to advance if he has selected the essential methods. There is still a chance errors could occur, however. For example, if a user selects the "linpack spd band" solution module, and the problem is not symmetric positive definite (spd) the linpack band routine will fail. Since there is no easy way to check whether the input matrix is spd ELLPACK assumes it is, and allows linpack spd band to start solving the system. Since ELLPACK is very robust, the moment it realizes that some dangerous computation (e.g., a divide by zero) is about to be done it calls an error reporting routine, and **stops**. Unfortunately the rpc calling linpack spd band cannot return (because the calling process has just been stopped), which causes the system to crash. We avoided this problem by modifying the ELLPACK error routines to return an error condition, and allow the caller to continue.

5.2 Summary and conclusions

In this work we studied the issues involved in creating a distributed problem solving environment for scientific computing. Scientific software development has not

kept up with the latest developments in workstation, windowing and networking technologies. The scientific researcher suffers from a lack of high level tools which would enable him to solve his problems without having to occupy himself with low level programming details. The scientific researcher has at his disposal the ability to solve his problem on a variety of different machines, because of the huge progress made in networking over the last few years. Unfortunately most scientific software runs from start to finish on a single machine, thus making inefficient use of the available computational power. Keeping these shortcomings in mind we investigated the issues and problems in creating a distributed problem solving environment.

In Chapter 2 we surveyed various problem solving environments for scientific computing. We discussed extensions to the ELLPACK system like Interactive ELLPACK, Multidomain ELLPACK, XELLPACK and //ELLPACK. We also looked at CLAM, a software environment for computational linear algebra, and environments like SHMAP, which provide tools for visualizing the performance of parallel numeric algorithms for shared and distributed memory machines.

None of the systems discussed in Chapter 2 provides a distributed problem solving environment, which we feel is crucial to scientific computing, so to aid our work we developed our own distributed problem solving environment. This system combines a very high level language, an interactive X Window interface, and a set of powerful solution methods into a single problem solving environment. This system is designed to solve elliptic partial differential equations, but we claim that the approach we took could apply to creating a problem solving environment for any other application domain. The system is distributed between a workstation running X and a more powerful parallel machine. Communication is done via RPC. The system provides many interesting tools for problem specification and solution, as well as analysis of results. The user can choose only those that he needs, and build his own mini environment tailored to suit his specific needs. This enables

the user to investigate the performance of various solution methods on different problems. Chapter 3 described the user's guide for our system. As described, the system is easy to use and does not require the user to have any programming knowledge. This is due to the fact that it is menu driven and, with the exception of equation and boundary specification, the user can build his entire environment using the mouse to select menu items he needs. The system also provides tools to enable the advanced user to add his own functions to those provided by the system. Thus it is useful to both novice and expert.

We next discussed the design and implementation of our system in Chapter 4. We used Xlib for low level graphics routines, and the Xt Intrinsics and the Release 4 Athena widget set to craft the user interface. Communication between client and server processes on the workstation and host machine is done via RPC. The low level implementation details were also discussed in Chapter 4. In Chapter 5 we critically evaluated the system, explaining the various design decisions we took, such as whom we felt the system should be targeted for, solution visualization, and other important issues.

We now address the question of whether our work will transfer to other areas in scientific computing. One of the fascinating features of scientific computing is that it encompasses an extremely diverse problem domain, each problem having its own requirements in terms of numerical accuracy of solutions, computational power required to solve the problem, output visualisation, and many other related issues.

The work done in this thesis focused on the use of the client/server model of distributed computing, and the design of user interfaces to ease problem description and solution. We feel that the days of monolithic standalone computers for scientific computing are over. With the availability of high bandwidth network links, it is possible for a user to have access to a very diverse set of powerful computers; e.g., to a supercomputer like the Cray 2, a distributed memory ma-

chine like the Intel iPSC hypercube, and a shared memory machine like a Sequent. We claim that scientific software designed to solve a variety of different problems, which can be mapped to many different machine architectures, will benefit the most from the approach to scientific problem solving that we have taken. Please refer to Section 5.3 for the design we envision for such software packages. Software which is focussed on a very narrow application domain may work best on a single machine, but can still be modified so that the user is provided with a user interface which runs on his workstation, and a computational backend which could run on another machine. In this way the user benefits from the ease of problem description provided by the user interface, and the perhaps more powerful computing power of the machine running the backend. The question as to whether a user needs a graphical user interface cannot easily be answered. It depends on too many different parameters, such as level of expertise of the user, ease of use of the software, amount of user interaction, etc. It is our experience, that in general, scientific software is not user friendly, and learning to use a new software package requires a fair amount of start up time. By providing a user interface like ours, the software can be made available to a much broader audience.

Since we have made use of tools which are freely available in the public domain to implement our system, installing it on new systems does not introduce any new software purchase costs. The X Window system which we used to implement our interface is available at no cost from MIT, whereas the RPC software is also available at no cost from SUN. These tools are easily portable to almost any platform running UNIX, which enhances the portability of our system.

We conclude this section by reaffirming the fact that DPSolve is a prototype distributed problem solving environment. It differs from conventional mathematical software packages like Mathematica or MATLAB in that it provides a distributed environment for research in problem solving, whereas the afore-mentioned packages are uniprocessor based. DPSolve is focussed on a much narrower application

domain than either of the two packages. However it provides ready access to a precompiled partial differential equation solution library which does not need to be modified in any way in order to be incorporated into the system. In order for Mathematica or MATLAB to borrow from ELLPACK the PDE algorithms in ELLPACK would have to be reimplemented to enable them to mesh in with the rest of the system. Nevertheless it is still possible to use the same approach we have taken in designing DPSolve to create a distributed Mathematica or MATLAB. Such a system would of course be much more general purpose than DPSolve and probably suited to a wider audience.

5.3 Future Work

DPSolve's interface was designed and implemented using the Athena widget set for the simple reason that this widget set is freely available in the public domain. Hence, it inherits the somewhat "clumsy" look of this software. OSF Motif [OSF 90b] has become an industry standard in graphical user interface development, and provides all the tools necessary to create a very professional looking user interface. We envision the user interface code being modified to use OSF Motif instead of the Athena R4 widgets.

Our system provides 2D wireframe images of functions selected by the user. We think that it is important to provide a 3-D data visualization tool. At present, the NCSA X Data Slice facility developed at the University of Illinois [NCSA 89] provides a way to visualize 3-D output on a 2-D terminal by viewing slices of the 3-D volume. This could be incorporated into our system, and greatly enhance its visualization capabilities. Another enhancement would be the provision of tools to specify non-rectangular domains interactively, using a cursor driven device.

We provide the user with ways to run his own set of procedures remotely. However we have not yet implemented a way of dealing with remote procedures which require terminal input. One way to handle this scenario is to have a tool

on the remote machine which parses the user defined source code and whenever it sees an input statement, replaces it with code to call routines which would make a callback to a client process on the local machine which would query the user for the required data and pass it back to the remote procedure.

We forsee this system being used as a problem solving environment for research in parallel scientific computational algorithms. There is a lot of work going on in the area of performance evaluation of parallel algorithms. In Chapter 2 we discussed the excellent work currently being carried out in developing tools to visualize the parallel performance of numerical algorithms on both shared and distributed memory machines. Our system could be extended to incorporate similar performance evaluation tools.

We believe our system has the potential to become a widely used tool in the solution of partial differential equations. We now discuss a possible design for an extension to DPSolve, see Figure 15. It is possible to modify the server process so that it in effect becomes a resource manager process, and distributes work to different nodes on the network. Clients on different machines would be able to communicate with a server process on one node of the network via rpc calls. The server process would then choose which machine to execute the process. For example, the ITPACK modules in ELLPACK run best on a vector computer, whilst other routines could run on a parallel machine. The different machines would compute the results of their computations and send it back to the server, which would then route it back to the client machine. This would require some revision of the ELLPACK problem solving modules so that communication is achieved via parameter passing instead of via global state variables.

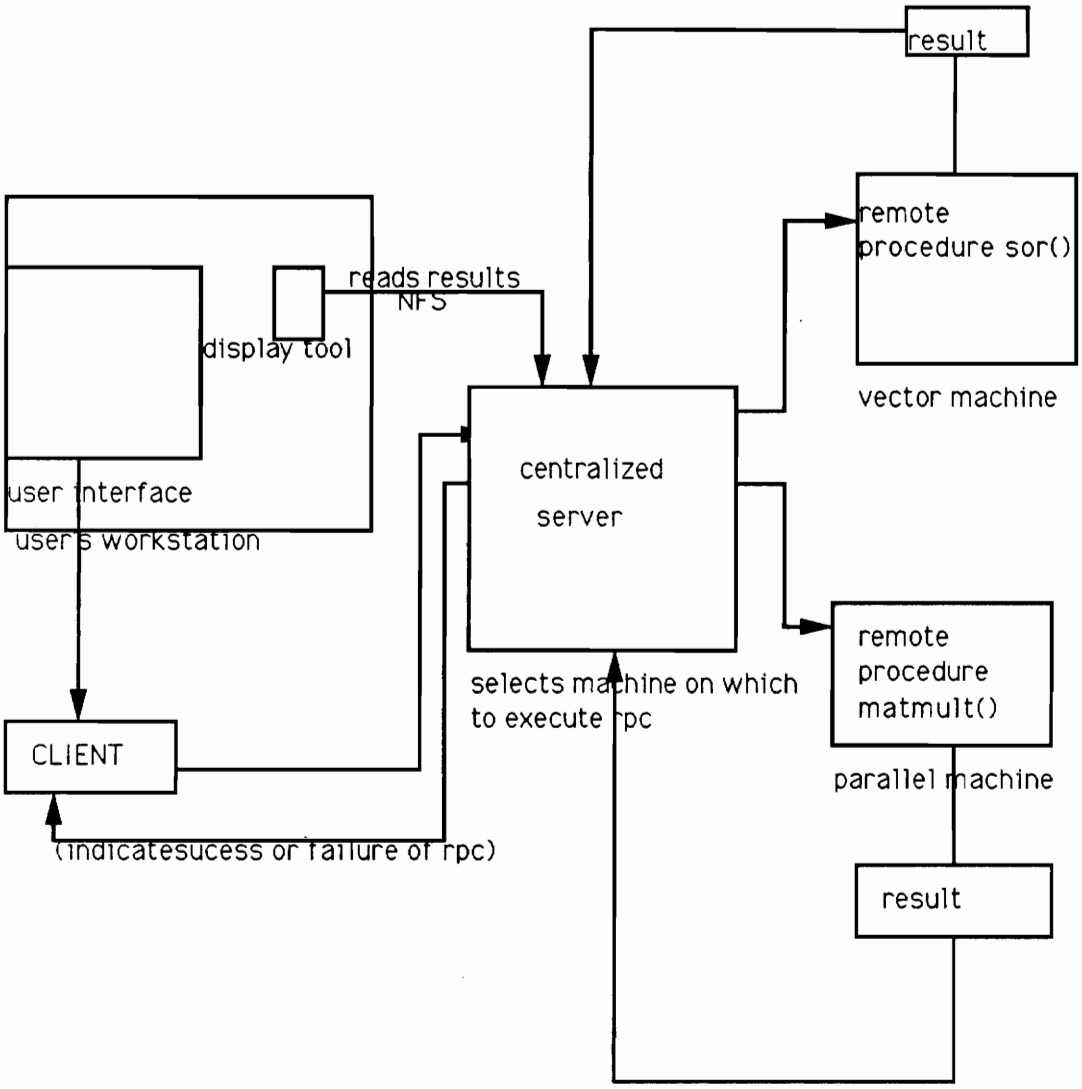

Figure 15: A design for a future version of DPSolve.

References

[Asen 84] P. Asente, *W reference manual*, Internal document, Dept. Computer Science, Stanford University, California (1984).

[Batz 82] J.C. Batz, P.M. Cohen, S.T. Redwine, and J.R. Rice. *The application specific task area*, IEEE Computer 16 (1983), pp. 78–85.

[Bono 88] J.P. Bonomo and W.R. Dyksen. *XELLPACK: An interactive problem-solving environment for elliptic partial differential equations*, Purdue University CSD-TR-839 (1988).

[Brew 89] O. Brewer, J.J. Dongarra, and D.C. Sorensen. *A graphics tool to aid in the generation of parallel Fortran programs,* Proceedings of the Thirteenth Annual International Computer Software and Application Conference (1989).

[Call 88] D.C. Callahan, K.D. Cooper, R.T. Hood, K. Kennedy, and L. Torczon. *ParaScope: A parallel programming environment,* International Journal of Supercomputer Applications 2 (1988), pp. 84-89.

[Char 90] B.W. Char, K.O. Geddes, G.H. Gannet, M.B. Monagan and S.M. Watt. *MAPLE, Tutorial and Reference Manual*, Watcom Publications, Waterloo, Canada (1990).

[Dong 86] J.J. Dongarra, and D.C. Sorensen. *SCHEDULE user's guide*, Report ANL-MCS-TM-76, Argonne National Laboratory (1986).

[Dong 89] J.J. Dongarra, O. Brewer, S. Fineberg, and J.A. Kohl. *A tool to aid in the design, implementation, and understanding of matrix algorithms for parallel processors*, Technical Report CS-89-91, Department of Computer Science, University of Tennessee, Knoxville, Tennessee (1989).

[Dyks 87] W.R. Dyksen and C.J. Ribbens. *Interactive ELLPACK: An interactive problem-solving environment for elliptic partial differential equations*, ACM Trans. Math. Softw, 13 (1987), pp. 113-132.

[Foul 90] D.E. Foulser and W.D. Gropp. *CLAM and CLAMSHELL: An interactive front-end for parallel computing and visualization* , Proceedings of the 1990 International Conference on Parallel Processing, P.C Yew, ed. (1990), pp. 35–42.

[Guar 89] V.A. Guarna, D. Gannon, D. Jablonowski, A.D. Malony, and Y. Gaur. *Faust: An integrated environment for paralllel programming*, IEEE Software (1989), pp. 21–26.

[Gett 86] J. Gettys. *Flexibility Is key to meet Requirements for X Window System Design*, Proceedings of the Winter, 1989 USENIX Conference (1989), pp. 125–138.

[Hous 90] E.N. Houstis, J.R. Rice, N.P. Chriscochoides, H.C. Karathanasis, P.N. Papachiou, M.K. Samartzis, E.A. Vavalis, Ko Yang Wang, and S.Weerawarana. *//ELLPACK: A numerical simulation programming environment for parallel MIMD machines*, Proceedings of the International Conference on Supercomputing (1991), to appear.

[Malo 90] A. D. Malony, D. A.Reed, J.W. Arendt, R.A. Aydt, D. Grabas, and B.K. Totty. *An integrated performance data collection, analysis, and visualization system*, Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications (1991), to appear.

[Mach 85] M. Machura. *Issues in the design of problem solving environments,* Proceedings of the IFIP TC 2/W.G.2.5 Working Conference on Problem Soving Environments for Scientific Computing (1985), pp. 263–278.

[Matl 89] *PRO-MATLAB User's Guide*, The Math Works, Inc., South Natick, MA (1989).

[McQu 77] J.M. McQuillan, and D.C. Walden, *The ARPA Network Design Decisions*, Computer Networks, 1 (1966), pp. 243–289.

[Nye 88a] A. Nye, and T.O'Reilly. *The Xlib Programming Manual*, O'Reilly and Associates, Inc., Sebastopol, CA (1988).

[Nye 88b] A.Nye, and T.O'Reilly. *The Xlib Reference Manual*, O'Reilly and Associates, Inc., Sebastopol, CA (1988).

[Nye 90a] A. Nye, and T.O'Reilly. *The X Toolkit Intrinsics Programming Manual*, O'Reilly and Associates, Inc., Sebastopol, CA (1990).

[Nye 90b] A. Nye, and T. O'Reilly. *The X Toolkit Reference Manual*, O'Reilly and Associates, Inc., Sebastopol, CA (1990).

[NCSA 89] NCSA group. *NCSA X data slice for the X Window System*, Technical report, Nat. Ctr. Supercomputer Appl., University of Illinois at Urbana-Champaign (1989).

[OSF 90a] Open Software Foundation White Paper. *Remote procedure call in a distributed computing environment*, (1990), pp. 1–12.

[OSF 90b] Open Software Foundation. *OSF/Motif Programmer's Reference* Open Software Foundation, Prentice Hall, Princeton, N.J. (1990).

[Purt 88] J.M. Purtilo, D.A. Reed, and D.C. Grunwald. *Research notes on environments for prototyping parallel programs*, Journal of Parallel and Distributed Computing 5 (1988), pp. 421–437.

[Ribb 86] C.J. Ribbens. *Domain Mappings: A tool for the development of vector algorithms for numerical solutions of partial differential equations*, Ph.D. Thesis, Purdue University (1986).

[Rice 85] J.R. Rice, and R.F. Boisvert. *Solving Elliptic Problems Using ELLPACK*, Springer-Verlag, New York (1985).

[Sche 86] R.W. Scheiffler, and J. Gettys. *The X Window System*, ACM Transactions on Graphics, 5 (1986), pp. 79–109.

[Simp 90a] D. Simpson. *Y's and Z's of the X Window System*, Systems Integration, Jan (1990), pp. 37–42.

[Simp 90b] D. Simpson. *Windows into networks*, Systems Integration, March (1990), pp. 37–43.

[SUN 88] SUN Microsystems. *Network Programming*, SUN Microsystems, Mountain View, CA (1988).

[Young 89] D.A. Young. *X Windows Systems Programming and Applications with Xt*, Englewood Cliffs, New Jersey (1989).

[Vand 90] M. VandeWettering. *apE 2.0* Pixel, Nov (1990), pp 30-35.

[Wolf 88] Stephen Wolfram. *Mathematica, a System for Doing Mathematics by Computer*, Addison-Wesley, Reading, MA (1988).

VITA

Colin Joseph deSa was born on April 26, 1966 in Bombay. He is the son of Michael and Ninette deSa. He graduated from the University of Bombay with a Bachelor of Science degree in Mathematics in August 1987. From 1987 to 1989 he attended Colorado State University where he earned a Master's degree in Mathematics. While at Colorado he served as a teaching assistant in the Department of Mathematics. He received a Master's degree in Computer Science and Applications from Virginia Polytechnic Institute and State University in June 1991. He served as a research assistant in the Department of Computer science from August 1989 till June 1991.