

Distributed Seismic Unix

by

Alejandro E. Murillo

ProQuest Number: 10794192

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10794192

Published by ProQuest LLC (2018). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

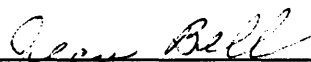
ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Master of Science (Mathematical and Computer Sciences).

Golden, Colorado


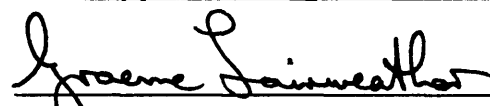
Date 4/1/96

Signed: 
Alejandro E. Murillo

Approved: 
Dr. Jean Bell
Professor of Mathematical and
Computer Sciences
Thesis Advisor

Golden, Colorado

Date 4/1/96



Dr. Graeme Fairweather
Professor and Head,
Department of Mathematical and
Computer Sciences

ABSTRACT

This paper describes TCL/TK based software called DSU (Distributed Seismic Unix) designed to assist geophysicists in developing and executing sequences of Seismic Unix (SU) applications in clusters of workstations as well as on tightly coupled multiprocessor machines. SU is a collection of subroutine libraries, graphics tools, and fundamental seismic data processing applications that is freely available via the Internet from the Center for the Wave Phenomena (CWP) of Colorado School of Mines (see appendix A). SU is currently used at more than 500 different sites in 32 countries around the world. DSU is built on top of three publicly available software packages: SU itself; TCL/TK, which provides the necessary tools to build the graphical user interface (GUI); and PVM (Parallel Virtual Machine), which supports process management and communication. DSU is intended to handle tree-like graphs representing sequences of SU applications. Nodes of the graphs represent SU applications, while the arcs represent the way the data flow from the root node to the leaf nodes of the tree. In general the root node corresponds to an application that reads or creates synthetic seismic data and the leaf nodes are associated with applications that write or display the processed seismic data; intermediate nodes are usually associated with typical seismic processing applications like filters, convolutions, signal processing, etc. Pipelining parallelism is obtained when executing single branch tree sequences, while a higher degree of parallelism is obtained when executing sequences with several branches. DSU provides tools for creating, editing, setting parameters, saving in plain ASCII files and executing SU applications sequences over several types of multiprocessor environments.

TABLE OF CONTENTS

ABSTRACT	iii
LIST OF FIGURES	vi
LIST OF TABLES	viii
ACKNOWLEDGEMENT	ix
1 INTRODUCTION	1
1.1 Motivations and design goals	7
1.2 Overview of this thesis	8
2 WHAT IS SU	10
2.1 SU application structure	13
3 DSU SYSTEM ANALYSIS	17
3.1 Problem statement	17
3.2 Static aspects	18
3.3 Object model	18
3.4 Identifying associations and attributes	19
3.5 Dynamic aspects	21
4 DSU SYSTEM DESIGN	25
4.1 The graphical user interface	25
4.2 The loading and saving tasks	26
4.3 Interface with the message-passing software	29
4.4 Distributing and monitoring subsystem	30
4.5 Data structures	32
4.5.1 Format to represent a graph in a plain ASCII file	32
4.5.2 Data structure to represent a sequence of DSU applications	35
5 DSU IMPLEMENTATION DETAILS	36
5.1 The graphical user interface (GUI)	37
5.2 Loading and saving procedures	39

5.3	Distributed execution	39
5.4	Modifications in the applications	41
5.5	Monitoring task	44
6	DSU GRAPHICAL USER INTERFACE	47
6.1	Mouse buttons	50
6.2	A typical session with DSU	52
7	ESTIMATING THE PERFORMANCE OF DSU	54
7.1	Why we expect speedup	54
7.2	A simple performance model	56
7.3	Computational experiments	58
8	SUMMARY AND FUTURE WORK	65
8.1	How DSU helps other research	67
8.2	Future research	68
	REFERENCES	70
A	ABOUT SU	72
A.1	How to Get a Copy of SU	73
A.2	Requirements for installing the package	74
A.3	Some core programs	75
A.3.1	Examining the trace headers	75
A.3.2	Some common processing programs	75
A.3.3	Some common plotting programs	76
A.4	A brief tour of the source directories	76
B	DSU SOFTWARE ORGANIZATION	78
B.1	Requirements for installing the package	78
B.2	Files in the DSU root directory	79
B.3	A brief tour of the source directories	79
B.4	Brief summary of the steps to install DSU	80
B.5	How to run dsu	81
C	TCL/TK OVERVIEW	83

LIST OF FIGURES

1.1	Example of a seismic experiment to obtain seismic data	2
1.2	Applying the seismic experiment	3
1.3	Example of a shell script representing a sequence of SU applications .	4
1.4	Example of a multi-branch SU application sequence as built using the DSU graphical tool	5
2.1	Organization of the main directories of SU software	11
2.2	Initialization step as coded for the SU application <code>susynlv</code>	14
2.3	Pipelining Seismic traces through a sequence of SU applications . . .	16
3.1	DSU main objects	19
3.2	State diagram for the Graphical-tool object	21
3.3	Execution pattern followed for each SU application when spawned by DSU	22
4.1	The DSU subsystem architecture	26
4.2	Recursive algorithm followed by DSU to save a graph-tree representing a sequence of SU applications in a plain ASCII file.	29
4.3	The distributing task	31
4.4	Recursive algorithm followed by DSU to spawn the applications asso- ciated with the nodes of a graph-tree representing a sequence of SU applications.	32
4.5	Plain ASCII file representation of the multi-branch sequence of applic- ations shown in Figure 1.4	33
4.6	Tree data structure used to store in memory a graph representing a sequence of SU applications	34
6.1	DSU Graphical User interface and some of its menus.	48
6.2	Scrollable help window for the application <code>susynlv</code>	50

6.3	Window for entering parameters values for the application <code>susynlv</code> .	51
6.4	Typical user session with the graphical tool of DSU.	53
7.1	Sequence 1 used to benchmark DSU	59
7.2	Sequence 2 used to benchmark DSU	59

LIST OF TABLES

7.1	Gantt chart representing the load of each processor in time, when executing a sequence of SU the applications with DSU	57
7.2	Processing 300 traces of 300 samples each with sequence 1	61
7.3	Processing 500 traces of 500 samples each with sequence 1	61
7.4	Processing 300 traces of 300 samples each with sequence 2	62
7.5	Processing 500 traces of 500 samples each with sequence 2	62
7.6	Estimating the overhead time with DSU	63

ACKNOWLEDGEMENT

I am extremely grateful to my advisor, Dr. Jean Bell, for her enthusiastic help, and guidance that made this work possible. I also indebted to Dr. Steven Pruess who has continually advised me since I first came to Mines. I really appreciate his willingness to sit down with me to discuss about anything. I would like to thank Dr. Manavendra Misra for his great job teaching advanced computer architectures last spring. That course gave me a lot of fundamental insights that facilitated the work presented here. I am tremendously grateful to John Scales for his great interest in my work and his continual encouragement. Last, but not least, I want to express my full gratitude to Dr. Jack Cohen and John Stockwell for their terrific work making SU what it is today. Without their continual hard work improving SU, this work would not have been possible.

Chapter 1

INTRODUCTION

The objective of a seismic study is to obtain a description or approximate picture of the subsurface. To achieve that, geophysicists produce vibrations (through controlled explosions or vibroseis trucks) at the surface of the earth to obtain data for determining the substructure of the subsurface.

Figure 1.1 is a cartoon of that process. First, a group of geophones are laid out on the surface of the area where the earth interior is under study. Then, vibrations are produced using dynamite or a vibroseis truck. Right after the vibrations are produced, in a period of a few seconds, each geophone registers the response of the possible geological structures in the subsurface every δt units of time (usually in milliseconds).

When the experiment is completed each geophone has collected a seismogram or seismic trace. Packing together these seismograms, the geophysicists obtain a picture of the subsurface in the *space-time* domain. Since the desired picture must be in the *space-depth* domain, several geophysical applications, that include signal processing and other transformations, have to be applied to those seismograms to obtain the final picture.

For example, if we conduct the seismic experiment in an area whose geological structure is like the one shown in the top part of Figure 1.2, then the picture corresponding to the seismograms obtained by the geophones (*space-time*) is the one shown in the lower left corner. The corresponding picture obtained after applying the required geophysical applications to those seismograms is the one shown in the lower

The Seismic Experiment

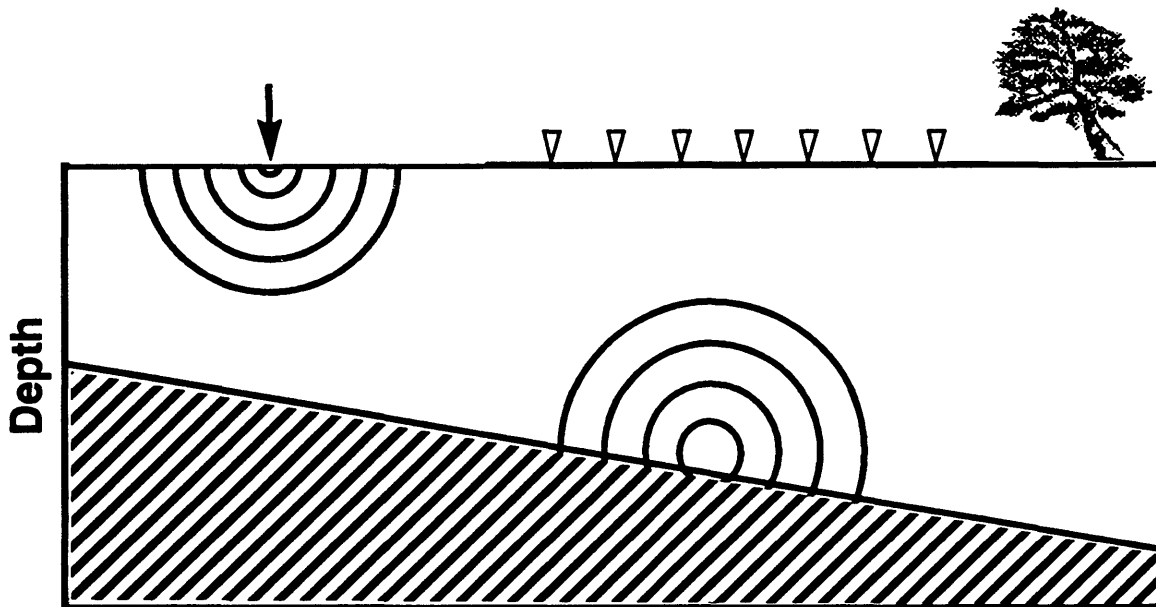


FIG. 1.1. Example of a seismic experiment to obtain seismic data. Vibrations are produced on the surface and the response of the geological structures in the subsurface are recorded by the geophones placed in the surface. Each geophone collects a seismogram or seismic trace.

right side of that figure. Clearly, this last picture is a much better approximation of the real image.

Seismic Unix (SU) is a collection of computer programs and subroutine libraries to assist the geophysicist in the task of converting the image from the *space-time* domain to the required one in the *space-depth* domain. It also facilitates the creation and testing of new applications that improve the accuracy of the obtained image (or picture) of the subsurface.

SU applications are expected to read their input traces from the standard input file and write their output to the standard output file. In this way, the user can

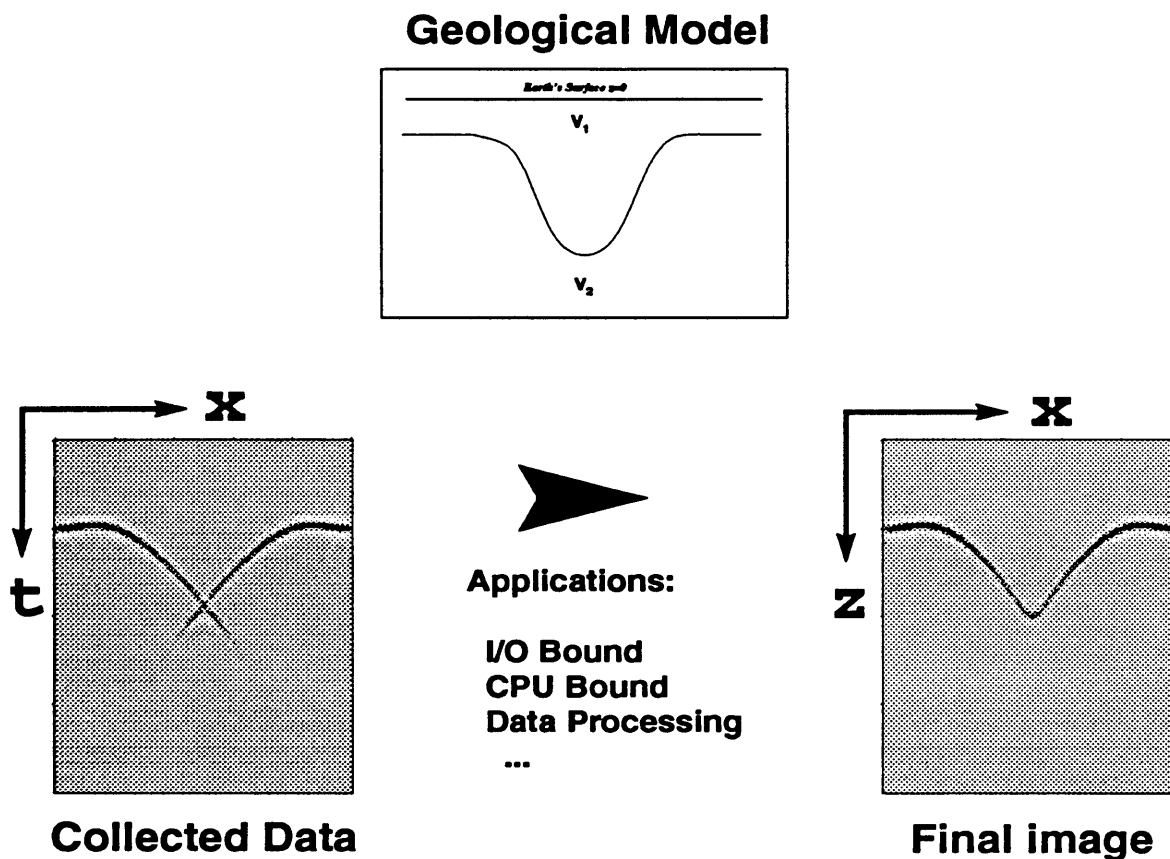


FIG. 1.2. The top part of this figure shows a possible geological structure in the subsurface. The picture in the lower left corner shows the seismograms in the *space-time* domain. The one in the lower right corner is the final image obtained after seismic applications have been applied to the data.

create sequences of SU (Seismic Unix) applications by means of shell script files in which several applications (along with their corresponding parameters) are specified, each one separated from the next by the pipeline operator (`|`) provided by Unix shells. When the shell script is executed, the output from one application is communicated to the next one through a pipeline file. Figure 1.3 shows a listing of a Bourne shell script representing a sequence of four SU applications. In it, synthetic data are generated with the modeling application `suspik`; fields of the trace headers representing that

```

#!/bin/sh
#
#
# This shell script computes impulse response:
#     It uses SUSPIKE to make test data
#     Migrates with SUGAZMIG
#     Displays output with SUXMOVIE

suspik nspk=3 |
sushw key=dt,d2 a=50000,.05 |
sugazmig tmig=0 vmig=1 |
suxmovie perc=99 title="Suspik data"

```

FIG. 1.3. Example of a shell script representing a sequence of SU applications. This sequence is composed of four applications (separated by |): **suspik**, **sushw**, **sugazmig** and **suxmovie**.

data are modified using **sushw**. Next, that seismic data are migrated using **sugazmig** and finally, the migrated data are displayed for analysis using **suxmovie**.

In this thesis I describe DSU (Distributed Seismic Unix), a tool and methodology to assist the geophysical community in executing sequences of SU applications in a multiprocessor environment. Additionally, DSU extends the capabilities of the regular SU software system by providing the user with capabilities to create multi-branch sequences of SU applications such as the one shown in Figure 1.4. Handling multi-branch sequences of SU applications is especially useful, since it allows the user to simultaneously execute several copies of a single branch sequence, but using different parameters on each branch; or to be able to quickly compare the answers of several algorithms that have the same objective. For instance, Figure 1.4 shows a sequence intended to compare the results of applying two different migration algorithms, in this case **supsmig** and **sugazmig**, to a specific input data created with the **susynlv** application.

DSU provides the necessary tools to set up a sequence of SU applications and ex-

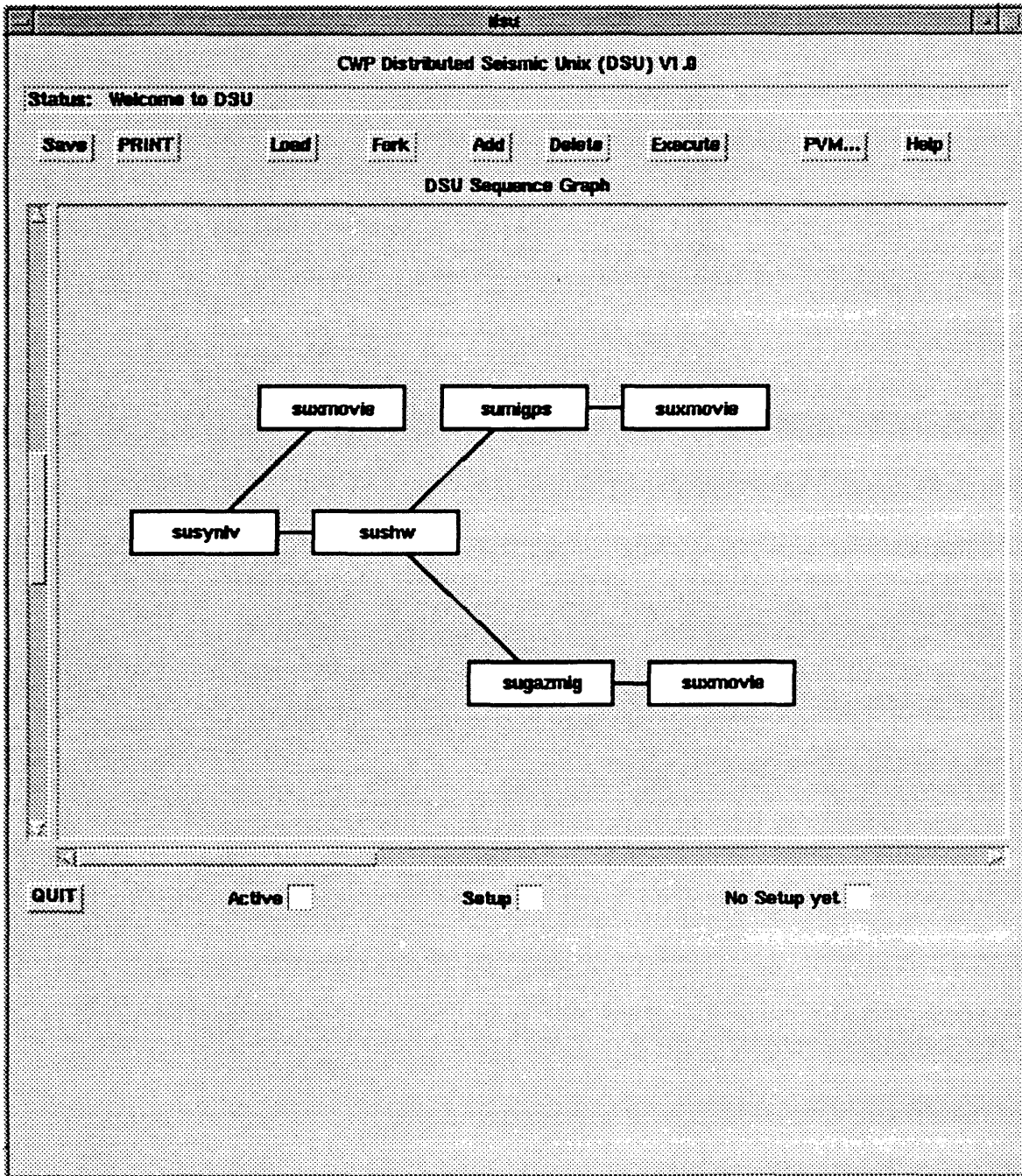


FIG. 1.4. Example of a multi-branch SU application sequence as built using the DSU graphical tool. The objective of this particular one is to compare the response of two different migration algorithms: `sumigps` and `sugazmig`.

ecute it over a multiprocessor environment. DSU attempts to execute each application of the sequence in the machine best suited to its needs. The general idea is to have the user explicitly specify, and setup, a sequence of SU applications and then allow the user to edit and execute it over a wide range of multiprocessor environments. A graphical user interface (GUI) is provided to facilitate the use of those tools.

DSU users make use of the GUI to specify SU application sequences in the form of tree-like graphs, where the nodes represent SU applications, and the arcs represent the way the data flow from the root node to the leaf nodes of the tree. The constructed sequence can then be edited, executed, and saved for posteriori execution.

A prototype version of this system has been implemented using TCL/TK for the GUI aspects and the message passing system PVM V3.9 for the process management and communication issues. The system has been tested on a network of LINUX and Silicon Graphics workstations. For the sake of portability, PVM functions are not directly invoked from the application; instead, functions from an intermediate subroutine library, created along with DSU, are explicitly invoked. In this way, we avoid strict dependency on a specific message-passing software (in this case PVM). Thus, the intermediate library facilitates the implementation of DSU with several message-passing systems by just implementing the intermediate library functions with the desired message-passing system. This is especially useful when going to a multiprocessor machine, where optimized message-passing libraries are usually provided.

The intermediate library contains the most common message-passing functions: send, receive and broadcast floats, integer, character arrays, and their combinations. Additionally, it contains functions to create and set up a specific communication pattern among a group of processes. For example, it provides a function to create and set up a pipeline among a group of similar processes.

1.1 Motivations and design goals

Using SU to create an accurate picture of the subsurface requires executing many sequences of applications several times before finding a convincing result. Sequences of SU applications contain programs with diverse computer requirements. Some of them are CPU bound, others I/O bound, some require windowing capacities, etc. In many cases, the processing sequences are so resource demanding that a single computer is not enough to get them executed. The execution of these sequences in a single machine is limited by the capacity of the machine being used. Furthermore, the use of pipe files to communicate the data among the applications restricts SU to single branch sequences of applications.

With DSU, these shortcomings are overcome by providing the necessary tools to execute sequences of SU applications on a heterogeneous network of computers. Heterogeneous computer networks represent an important resource for many companies and universities. DSU provides a tool that will allow the user an effective use of heterogeneous networks. The features in the final DSU product include:

1. providing an interface that hides the implementation details such as process synchronization and data handling;
2. providing easy to extend software, which can be implemented on local area networks and on distributed memory parallel computers as well;
3. performing competitively both on clusters of workstations and on expensive supercomputers.

Since performance is usually the major reason for considering parallel or distributed computing, much of the work done in parallel processing in geophysics has been

dedicated to the parallelization of specific CPU-bounded applications like seismic data modeling and migration ([Almasi *et al.*, 1993],[Black and Su, 1992]). Furthermore, the vast majority of programming environments that make distributed computing available to the application programmer usually are intended to provide tools for single application development and testing [Geist *et al.*, 1991], [BabaOglu *et al.*, 1991], [Carreiro and Gelernter, 1989], [Flower *et al.*, 1991]. However, not too much has been done in providing environments for assisting the user when executing several types of applications (parallelized or not) at the same time, under a distributed environment or in a tightly coupled multiprocessor environment. The work in this paper, as described below, starts out in this last direction.

1.2 Overview of this thesis

The organization of this thesis is as follows. In chapter 2, we include a brief introduction to the SU system software. Chapters 3 through 5 describe the DSU system requirements, design and implementation. The last three chapters of this thesis contain a description of the graphical user interface tool and the facilities it provides; a brief analysis of the expected performance of DSU; and, a summary of the goals achieved and the future research directions in this work.

To describe the software model underlying DSU, we will use object-oriented concepts. Specifically, we will employ the object modeling technique (OMT). This technique provides a graphical notation that facilitates the specification of the dynamic, functional and static aspects of the software model being described [Rumbaugh *et al.*, 1991]. The development of DSU is divided into three main phases, as described below.

System analysis. The goal of this section is to develop a model of what DSU will do. The model will be expressed in terms of objects, relationships, and dynamic

control flow. Here we present information that is meaningful from a real world perspective and should present the external view of the system. These are the main components of our analysis:

1. Problem statement
2. Object Model
3. Dynamic Model

System Design. In this phase the main components of the final system are derived. The system is described in terms of subsystems. The data structures to be used are determined and the most appropriate algorithms are selected for each one of the tasks to be accomplished. Emphasis is placed on:

1. The architecture of the system
2. The subsystems composing the package

Implementation details. In this phase we explain in detail how each one of the subsystems, data structures and algorithms derived in the design phase were really implemented.

Chapter 2

WHAT IS SU

The Seismic Unix (SU) system is a self-contained software environment for seismic research and data processing [Cohen and Stockwell Jr., 1991]. This system was created under two simple and basic ideas: simple user interface and full Unix compatibility. The former idea aims for extendibility and the latter one has allowed its use in several platforms due to the wide use of the Unix system. Here we will concentrate in describing the functional and programming aspects of SU; the use of the system is described in [Cohen and Stockwell Jr., 1991].

SU is a group of seismic processing applications complemented with a variety of applications for manipulating and displaying seismic data. It also provides a robust set of subroutine libraries that greatly facilitate the development of new SU applications. The applications provided with SU are classified according to their purpose.

Figure 2.1 shows the organization of the directories of the SU software. The main directory of the distribution package contains four subdirectories. The **bin** subdirectory contains the executable for each one of the applications provided with SU. The directory **include** contains all the include files (i.e. **.h** files) necessary for compiling programs that utilize functions of the SU subroutine libraries. The **lib** subdirectory stores each one of the libraries distributed with the software. The source code for each one of the main applications provided with SU, along with the corresponding Makefile used to generate its executable, are stored under the directory **src**. The applications are organized in this subdirectory according to their purpose and also according to their software requirements (some applications require windowing capabilities, for ex-

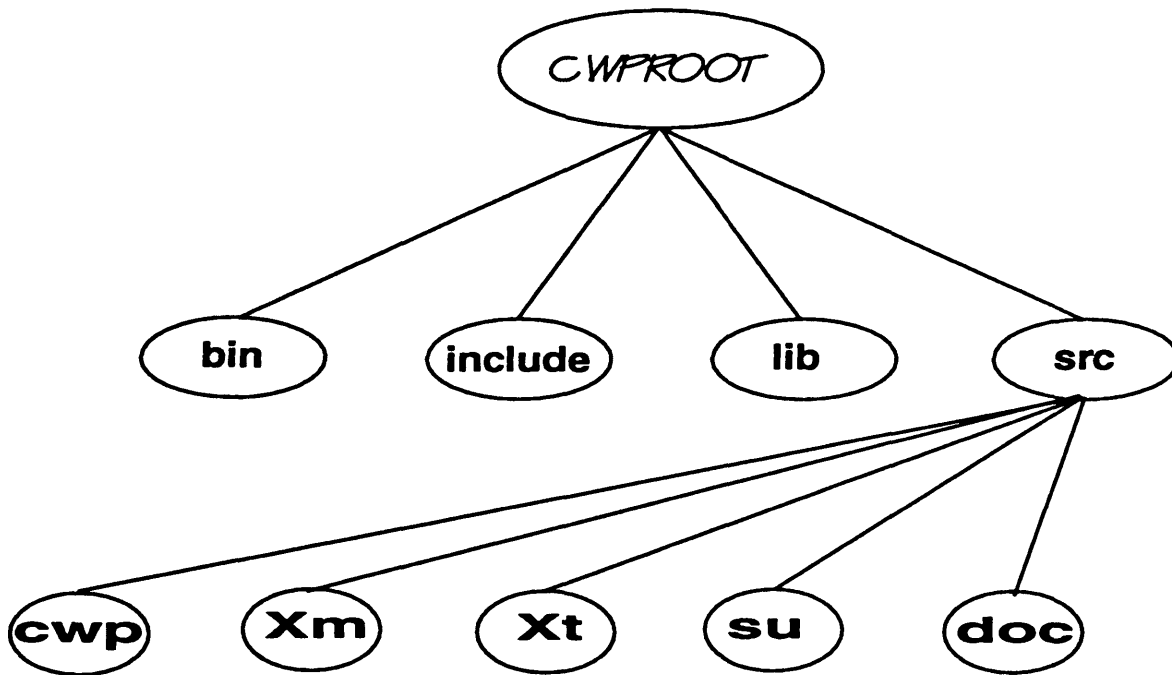


FIG. 2.1. Organization of the main directories of SU software

ample). The directory **cwp** contains utility programs developed using functions of the different libraries provided with SU. It also contains programs that illustrate how to use some of those functions. The subdirectory **Xm** contains the source code of applications requiring the motif window manager software. The subdirectory **Xt** contains those applications requiring X-window software libraries. The **SU** subdirectory contains the source code of all the strictly geophysical applications, which constitute the core of the system itself.

The basic concept behind the main applications composing SU is that each one of them is intended to be used as any other Unix command, which makes them suitable to be included in any conventional Unix shell script. In this way, the user can make effective use of them by simply exploiting the programming facilities provided by most of the Unix-shell programs currently available (i.e. *csh*, *tcsh*, *ksh*, etc).

In summary, the main components of SU are the following:

- **SU main programs:** these programs constitute the core of the system. There are more than one hundred applications available, ranging from applications to display seismic trace datasets to advanced seismic migration and modeling codes;
- **SU support subroutine libraries:** the most important subroutine libraries of SU are `libcwp.a`, `libsu.a`, and `libpar.a`. These libraries implement a wide range of functions, ranging from basic functions to allocate space for multidimensional arrays to more complex numeric calculations like mixed-radix fast Fourier transformation. Supporting functions for performing tasks such as parameter reading and modification of trace headers are provided as well;
- **SU documentation programs:** SU is delivered with a complete set of document files that provides the user with help about almost anything in the package. Virtually all SU programs give information about themselves when you type the name of the program without any arguments (self-documentation). SU provides an utility called `sudoc` that lists the self-documentation of all programs in the package. Subroutines and the few mains and shell scripts that do not show a `selfdoc` will have a `sudoc` entry.
- **SU Shell tools and utilities:** the development and wide use of SU has produced a lot of utility programs useful in the developing of new applications as well as in the manipulation and visualization of seismic data.

2.1 SU application structure

Any SU application is meant to be handled as any native Unix command, so that it can be included in pipeline sequences of commands just like any native Unix command. Each SU application expects its input from the standard input file and writes its output to the standard output file. Every main application in SU is structured in the following way:

1. Get/check arguments
2. Process data (two alternatives):

Loop (while not End Of File)	(a) Loop to read all traces
(a) Read a trace: <code>gettr()</code>	(b) Process all traces
(b) Process a trace (application code)	(c) Loop to write all traces
(c) Write a trace: <code>puttr()</code> .	
EndLoop	
3. Exit procedure

A set of subroutine libraries is provided to accomplish most of these steps. The step one above is composed of several calls to functions belonging to the *libpar.a* library; this library allows the user to read the command line parameters provided to the application in the format *par_name=par_value* in a very easy way. In this portion of the code there is a call to the function *initargs()* and several calls to the family of functions *get_pars* (*getparint()*, *getparfloat()*, and others).

Figure 2.2 shows step one as coded for the application *susynlv*. The first sub-step is an invocation of the subroutine *initargs()* to establish the source of the parameters

```

/* hook up getpar to handle the parameters */
initargs(argc,argv);
requestdoc(0);
/* get required parameters */
if (!getparint("nx",&nx)) err("must specify nx!\n");
if (!getparint("nz",&nz)) err("must specify nz!\n");
if (!getparfloat("tmax",&tmax)) err("must specify tmax!\n");
nxs = countparval("xs");
nzs = countparval("zs");
if (nxs!=nzs)
    err("number of xs = %d must equal number of zs = %d\n",
        nxs,nzs);
ns = nxs;
if (ns==0) err("must specify xs and zs!\n");
getparfloat("xs",xs);
getparfloat("zs",zs);
/* get optional parameters */
if (!getparint("nt",&nt)) nt = 0;
if (!getparint("mt",&mt)) mt = 1;
if (!getparfloat("dx",&dx)) dx = 1.0;
if (!getparfloat("fx",&fx)) fx = 0.0;
if (!getparfloat("dz",&dz)) dz = 1.0;
if (!getparfloat("fz",&fz)) fz = 0.0;

```

FIG. 2.2. Initialization step as coded for the SU application `susynlv`. The `initargs()` function determines the place where the parameters are; the family of `getpar...()` functions are utilized to retrieve the provided parameter values.

for this application; in this case the parameters come from the command line (`argv`). The following sub-steps are coded with successive calls to the family of functions `getpar()`. These functions retrieve the value, if provided, for a specific parameter; the parameter name is provided and the function returns its value. There is a `getpar()` function for each predefined type.

Step two is considered the core part of each application. It corresponds to obtaining, processing and delivering of the seismic data to be processed. Each SU application expects its input in a format called SU seismic trace. Roughly speaking, a SU seismic trace is a contiguous array of floating point numbers preceded by a header

of 240 bytes. The floating point numbers represent samples of seismic data and the header contains additional information about those samples. To read a trace from the standard input, SU applications make use of the function *fgettr()*. To write a trace to the standard output the function *fputtr()* is used.

The second step of the scheme presented above has two alternatives. Some SU applications can process the incoming data trace by trace, but some of them require all the traces in memory before beginning to process the data. The latter type of applications make repeated calls to the function *fgettr()* to get all the traces, apply their code, and then make repeated calls to the *fputtr()* function to deliver all the traces. In contrast, the former type will process a trace at a time, delivering it immediately to the process(es) following in the sequence.

Consequently, trace by trace applications will allow more parallelism when executing the sequence in a multiprocessor environment, because all the applications can be working on different traces at the same time as is shown in Figure 2.3. To overcome the possible bottlenecks created by applications that require all the traces before applying their codes, load balancing techniques should be incorporated so that they get executed on the fastest available machines.

The final or exit step of the SU application structure performs cleanup tasks. Here the program informs the user about the success of the execution performed. Each SU program indicates a successful completion by executing the instruction *return(EXIT_SUCCESS)*. An unsuccessful completion is indicated by stopping the execution with a call to *return(EXIT_FAILURE)*.

Each step of the standard structure of an SU application is implemented by invoking a few specific subroutines, provided in the SU libraries. As we will see later, this standard structure greatly facilitated the implementation of Distributed Seismic

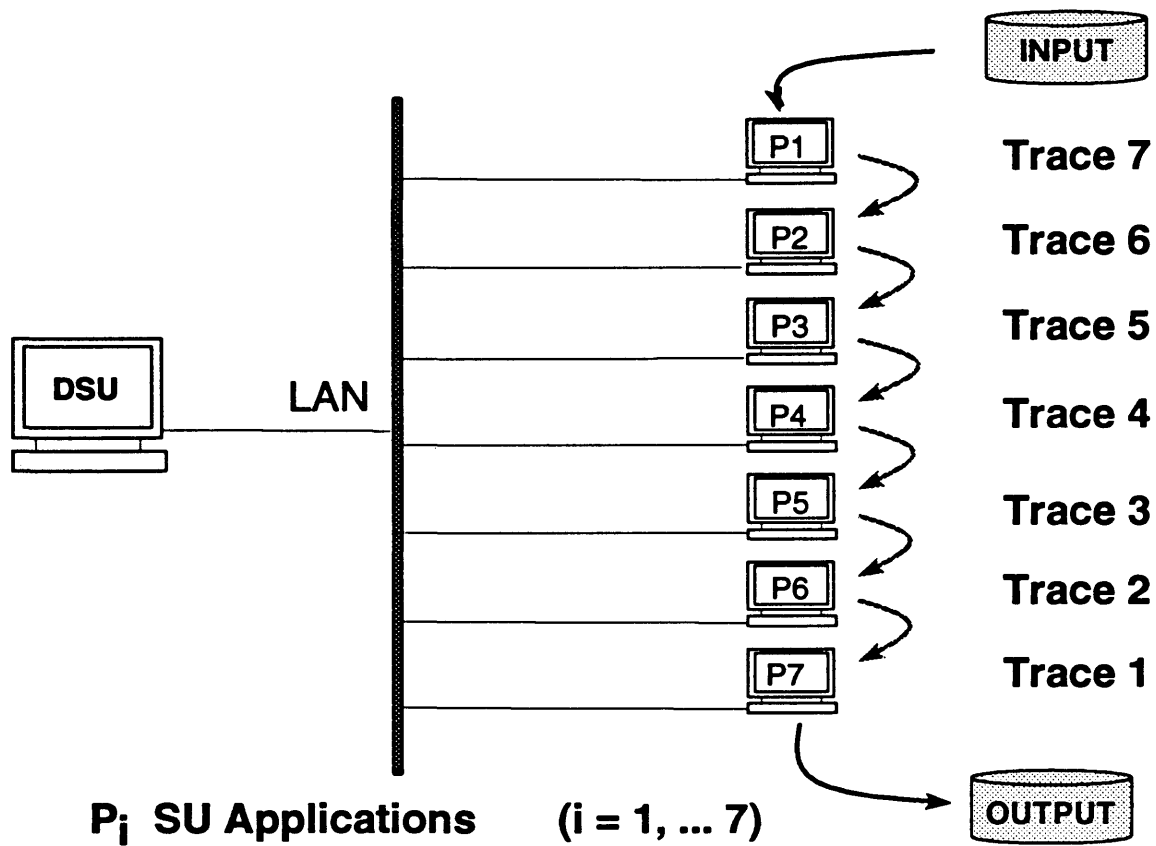


FIG. 2.3. Pipelining Seismic traces through a sequence of SU applications. Note how the applications work simultaneously, but on different traces.

Unix, since most of the work involved only modifications to those subroutines and not the applications themselves. This is extremely convenient, given the large number of SU applications currently available.

Chapter 3

DSU SYSTEM ANALYSIS

This chapter presents a model of Distributed Seismic Unix. The model is expressed in terms of objects, relationships and dynamic control flow. The model represents information that is meaningful from a real world perspective and presents the external view of the system.

3.1 Problem statement

The goal of the DSU software system is to provide the tools for the efficient creation, control and management of sequences of SU applications. The system should provide facilities to execute the created sequences on a network of workstations and on a multiprocessor machine as well.

The system should be able to handle conventional sequences of SU applications as well as extended (or multi-branch) sequences such as the one shown in Figure 1.4. It must provide capabilities to set individual application parameters, host preference for execution and input/output filenames when required. The user should be able to save created sequences in plain ASCII files for later use.

The software is intended to be portable to several types of platforms. Given the wide range of application types available with SU, the implementation of this system on a networked group of heterogeneous machines must allow a most effective use of each one of the resources available by executing each application in the machine best suited to its needs.

Finally, the system must provide a facility to monitor the execution of the se-

quences on a distributed environment, allowing the user to keep track of what is going on during execution and reporting possible execution errors.

3.2 Static aspects

This section describes the most important objects involved in the DSU environment, and how they are interrelated. The abstract data types involved in the DSU software are described in regard to what information they represent and how they provide that information to the other objects.

3.3 Object model

DSU is composed of five general abstract objects. Figure 3.1 is a graphical representation of these five objects and how they are associated with each other. In this figure, every box represents an object, the top label of the box indicates the name of the object. DSU is comprised of the following objects:

1. The **Graphical-tool** object
2. The **Task-Graph** or just **Graph** object representing the SU application sequence
3. The **Task** or node of the graph object
4. The **Network-info** object representing the information about individual processing units
5. The **Application-info** object representing individual application information.

Each one of these objects is composed of other objects that for the sake of simplicity are not explained here. In general, when implemented each one of those objects

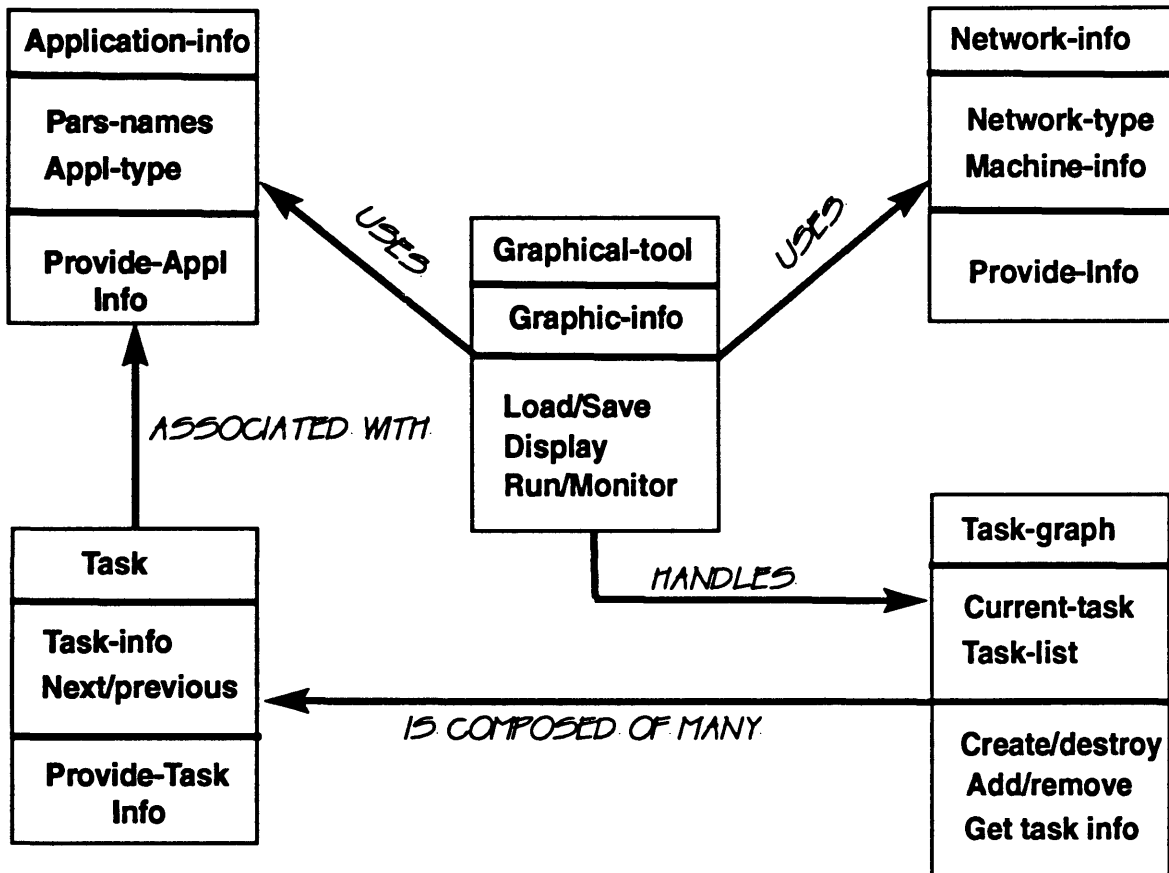


FIG. 3.1. DSU main objects. Each box represents an object; its top label is the object name, the middle labels are the object attributes and the bottom labels are the functions or methods associated with the object.

becomes a data structure with a group of subroutines intended to manipulate their contents.

3.4 Identifying associations and attributes

This section identifies associations between the above classes. An association is any dependency between two classes or a reference from a class to another; they usually correspond to stative verbs or verb phrases (i.e. *works for*). We also specify

what information each object should provide and how it is provided. In Figure 3.1, attributes correspond to the middle labels of the boxes. The labels in the bottom of each box represent the operations or methods associated with the object. The methods of an object, most of the time, are the vehicles used to request information about that object.

1. The DSU **Graphical-tool** is the object in charge of handling (creating, displaying, etc) the graph object. It gets implemented as the graphical user interface. This object combines the information provided by the **Network-info** and **Application-info** objects to efficiently perform a distributed execution of the sequence represented by the graph.
2. The **Task-graph** object is an abstract representation of the sequence of SU applications. As shown in Figure 3.1, this object is composed of a list of **Task** objects that together describe a sequence of SU applications.
3. The **Task** object is used to represent a single SU application in a graph. It contains information about the application it represents, a pointer to the graphical objects representing it in the graphical user interface, the hostname, I/O filenames and other specific information associated with the application it is currently representing.
4. The **Application-info** object provides specific information about the characteristics of each one of the available SU applications. The **Graphical-tool** object can request information from this object regarding the CPU, I/O boundedness or any other feature of a particular SU application. Each task of the graph will be associated with a specific instance of this object. When a single application occurs many times in the graph representing a SU sequence, a single instance

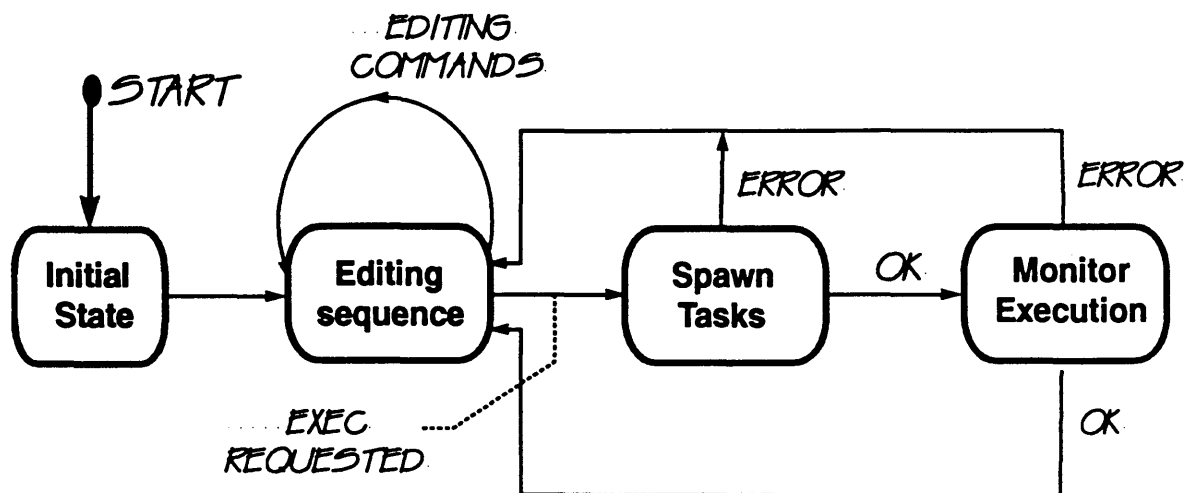


FIG. 3.2. State diagram for the **Graphical-tool** object. The main states are: initialization, editing, spawning and monitoring.

of this **application-info** object is referenced by as many **Task-nodes** objects of the graph.

5. The **Network-info** object provides specific information about the processing units available. This object is especially useful under a heterogeneous network of computers; it supplies the distributing task with information such as CPU and I/O capacity of each machine available, which is fundamental information to achieve a good load balancing.

3.5 Dynamic aspects

This section identifies and explains the possible states that objects can achieve during a typical session with DSU. Only the **Graphical-tool** object has extensive dynamic behavior; therefore, we will concentrate on explaining the dynamic aspects of just that object.

To describe how the **Graphical-tool** object behaves in time, during a session

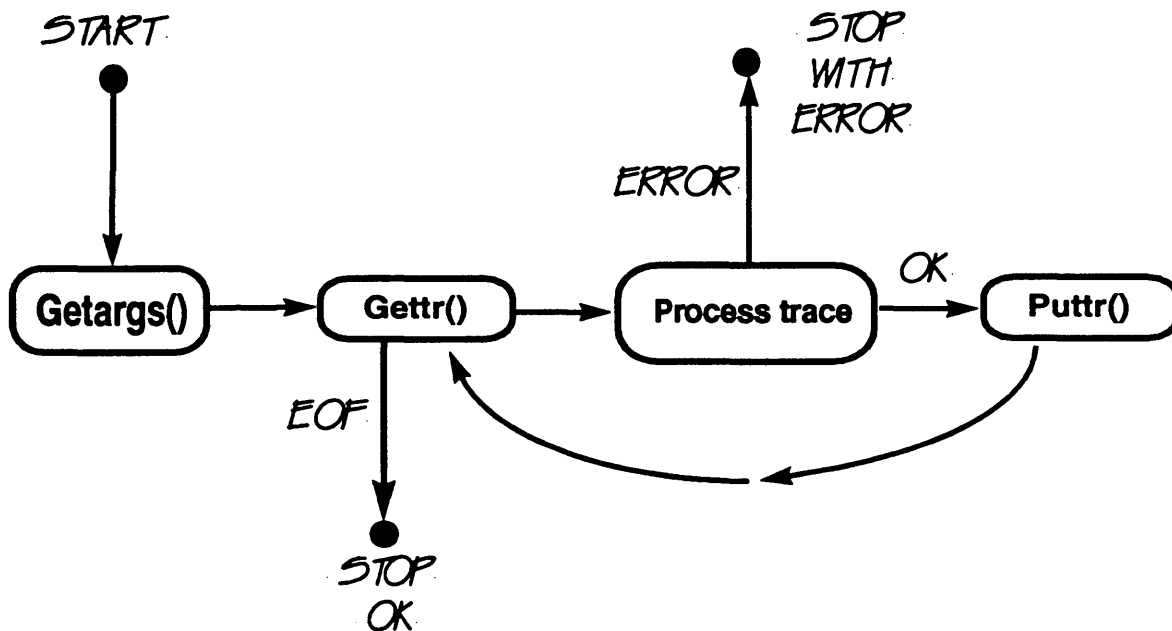


FIG. 3.3. Execution pattern followed for each SU application when spawned by DSU. First, it gets and checks the parameters provided. If no error is found, the application enters a loop to receive, process and deliver the data traces, until an end of data signal is received.

with DSU, we will use the concept of state diagram. A state diagram is a graphical representation of a sequence of operations that an object accomplishes, without regard for what the operations do, what they operate on, or how they are implemented [Rumbaugh *et al.*, 1991].

Figure 3.2 represents the state diagram for the **Graphical-tool** object. At the beginning of a session with DSU, this object executes some initialization tasks. Immediately after, it enters an editing state in which commands to create and edit sequences of applications are processed. When the execution of the sequence is requested, some specific steps are accomplished to execute the sequence on the multiprocessor environment available. Right after the execution is initiated, the **Graphical-tool** object switches to another state in which the progress of the execution is monitored.

The **Graphical-tool** object is expected to be in the editing state most of the time ; this is, processing commands for building a sequence of SU applications. This state corresponds to the graphical user interface aspects of this application. The spawning state corresponds to the task of executing a sequence of applications on the multiprocessor environment available. Essentially, what this object does here is to spawn each application of the sequence as an independent process on a specific processing unit and then provide each of those processes with its respective parameters and the information of its parent and children applications in the sequence.

When the spawning process has been completed, the **Graphical-tool** object switches to the monitoring state. Here two events can happen; first, one of the applications has a run time error, in which case, the DSU **Graphical-tool** stops the execution by killing all the processes. The second possible event is that the execution was successfully completed and no special procedure has to be performed. In both cases the user is notified about the success or failure of the execution and the **Graphical-tool** returns to the editing state.

When a process is spawned by the **Graphical-tool**, it follows the execution pattern shown in Figure 3.3. First, it reads and checks the parameters provided by the **Graphical-tool**. In case of detecting an error, it notifies the **Graphical-tool** of it and stops. Otherwise the process proceeds to obtain input traces from its parent application, processes it and delivers it to its child applications, until an End of Data message is received. Similarly, if an error occurs during that time, then the **Graphical-tool** is notified and the execution stops. This execution pattern is derived from the structure of an SU application that was described in chapter 2.

A key point about these two diagrams is that they contain all the information needed to comprehend how DSU executes a SU sequence of applications. The state

diagram for the **Graphical-tool** object shown in Figure 3.2 turns out to be the main program of the implemented system, and the flow chart followed by each application when spawned by DSU (shown in Figure 3.3) turns out to be the pattern followed to accomplish the modifications on the SU applications to introduce the new synchronization scheme.

Chapter 4

DSU SYSTEM DESIGN

In DSU the user builds graphs that represent sequences of SU applications. Each node of the graph represents a specific SU application, while the arcs represent how the input seismic data flow through the applications. A shell script representing a sequence of SU applications is considered a single branch graph-tree (each application has only one child application), while extended sequences of SU applications, such as the one shown in Figure 1.4, are considered multi-branch graph-trees.

In DSU there are four independent subsystems that are available to the user through the Graphical User Interface (GUI). Figure 4.1 shows how these subsystems are inter-related. The implementation of these subsystems will require the utilization of external subroutine libraries for accomplishing the graphical user interface, process handling and communication aspects.

4.1 The graphical user interface

DSU must provide the user with a graphic tool to create and manipulate graphs representing SU application sequences. This subsystem is the user interface itself and accomplishes the functions of displaying and handling the graph representing the sequence of SU applications. It also provides all the facilities to enter parameters for the applications and the context sensitive help features. It must provide buttons and menus to allow the user the invocation of the subsystems explained below.

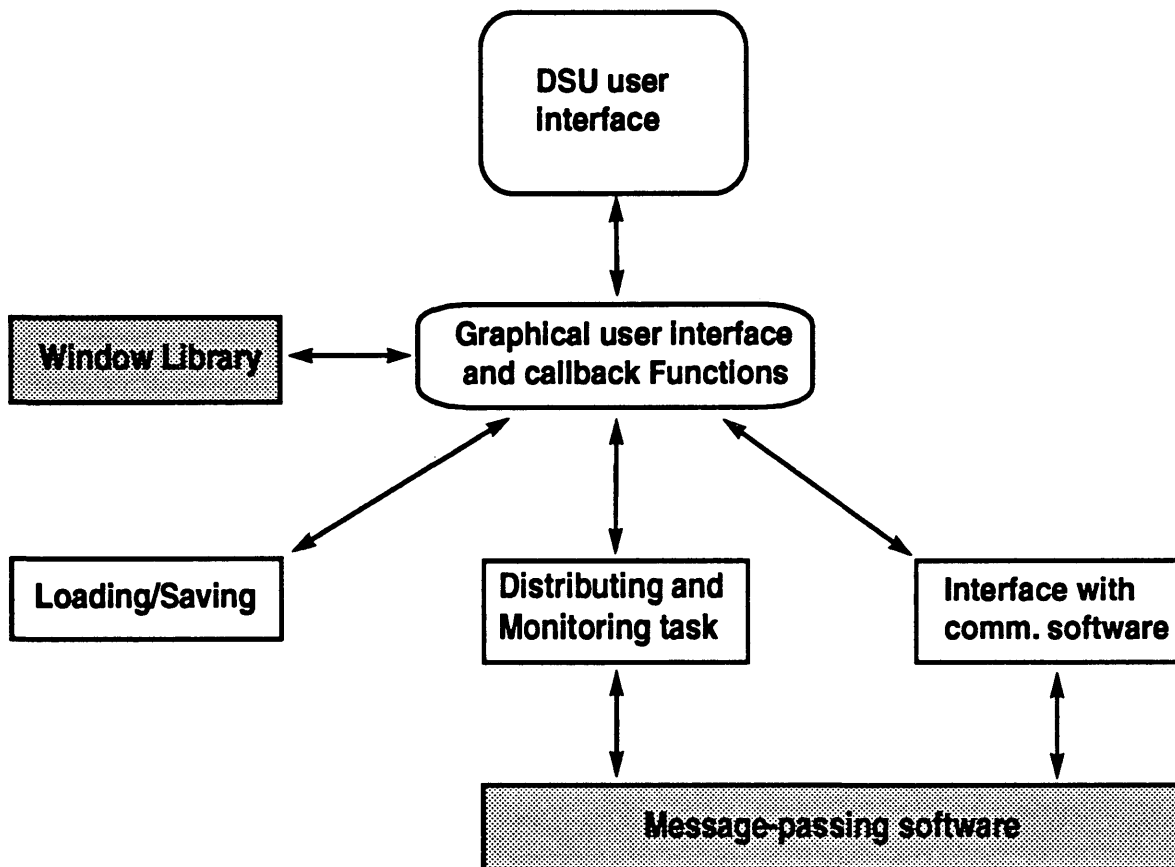


FIG. 4.1. The DSU subsystem architecture. Shaded boxes represent external subsystems.

4.2 The loading and saving tasks

This subsystem deals with the tasks of loading/saving sequences of applications from/to plain ASCII files. The format for representing a sequence in a plain file was adapted from the one used to represent regular SU application sequences: a shell script. This format is explained in detail below. The loading procedure is a kind of lexical analyzer program that extracts a graph representing a sequence of SU applications from a plain ASCII file. On the other end, the saving procedure will store a given graph on a plain ASCII file using the format explained below.

The loading procedure is in charge of reading a sequence represented in a plain ASCII file and storing it in memory. This procedure is based on a lexical analyzer program whose job is to read the input file and assemble the characters into tokens, and in a subprogram in charge of handling those tokens to assemble the sequence represented in the file. The lexical analyzer procedure developed here is based in the one described by [Rochkind, 1985].

A plain ASCII file representing a sequence of SU applications consists of a sequence of one or more SU applications (commands) separated with bars (|). Each application name is specified as explained below. Additionally, before starting the specification of the sequence of applications, these files might contain assignment commands, in which macros or variables are assigned with values. The loading procedure will process those assignments and store the values of the variables in a symbol table. The lexical analyzer subprogram will then be able to look up the value of those variables when a token of type \$MACRO is specified.

Each time the lexical analyzer is called, a token is returned. It works as a finite state machine: as characters are read they are either recognized immediately as tokens or they are accumulated (characters of a word, for example). It bypasses irrelevant characters of the input, such as blanks separating arguments. With each character, the lexical analyzer subprocess can switch into a new state which serves to remember what it is doing and how characters are to be interpreted. Our lexical analyzer has five basic states:

1. **NEUTRAL**: The starting state. Blanks and tabs are skipped. The characters |, &, <, >, # are recognized immediately as tokens. A quote causes a switch to state **INQUOTE**. A \$ character causes a switch to state **INMACRO**. A # character causes a switch to the **COMMENT** state. Anything else is considered

the beginning of an unquoted word, saved in a buffer and the state is switched to **INWORD**;

2. **INQUOTE**: In this state characters are accumulated until the closing quote is read. Then a token type word (and the word) is returned;
3. **INMACRO**: This state means that a \$ character was read. Characters are accumulated until a word is formed. This word is considered a variable and its value is looked up in a symbol table. Then a token type word is returned along with the value of the variable.
4. **INWORD**: This state means that the first character of a word was read. Following characters are accumulated until a nonword (delimiter) character is read. Then a token type word is returned along with the word.
5. **COMMENT**: Whenever a non-escaped # character is found, the lexical analyzer skips the following characters until an end of line character is read. Then it turns to the **NEUTRAL** state.

Using this lexical analyzer program, the task of building the graph out of the file is reduced to the following loop:

```

While ( (Token = GetToken()) != EOF ) {
    If (Token == '|')
        Add node to the graph
    Else
        Add information to current node
}

```

The loop calls on the **GetToken()** function, which is the lexical analyzer program. Some error considerations were omitted, but the loop above expresses the essential idea.

```

RECURSIVE_VISIT_TO_SAVE(Node T, parent_node, counter)
{
  if (T == NIL) return;
  me = counter;          /* Save the actual counter value */
  counter++;            /* Increment the counter value */
  Print_this_node_info(T, me, parent_node);

  /*
   Recursive visit to the children nodes.
   Note that the counter gets modified as
   the recursion goes down.
  */
  for (i = 0; i < max_number_of_children; i++)
    RECURSIVE_VISIT_TO_SAVE(T.child[i], me, &counter)
}

```

FIG. 4.2. Recursive algorithm followed by DSU to save a graph-tree representing a sequence of SU applications in a plain ASCII file.

The task of saving the graph in a plain ASCII file consists of recursively visiting and enumerating each node of the graph, starting from the root node. The algorithm is shown in Figure 4.2. This algorithm is invoked the very first time with the root node and the recursion will visit all the nodes. The function `print_this_node_info()` will print the information about that node in the format explained below.

4.3 Interface with the message-passing software

This subsystem involves the communication and process management software to be used. This subsystem is in charge of handling all interaction of the user with that specific software. For example, software systems like PVM allow the user to dynamically add or remove a machine to/from the so-called virtual machine. The user must be able to perform this kind of task through the DSU graphical tool, without interacting directly with PVM.

4.4 Distributing and monitoring subsystem

This subsystem handles the task of executing the sequence represented by the graph and a posteriori monitoring of that execution. It is in charge of initiating the execution of each node (application) represented in the graph. It is also in charge of communicating the respective parameters to each application. Similarly, this subsystem provides each spawned application with the process identification of its parent application (since we are handling tree-like graphs, each node (application) in the graph only has one parent) and the process identification of its children applications, so that the input data read (or generated) in the root node of the graph flow in the way indicated by the arcs of the graph.

Figure 4.3 shows how this subsystem works. When it is invoked, it combines the information about the hardware platform available and the information about each one of the applications included in the graph to attempt an efficient execution of the sequence on that platform. For example, over a heterogeneous network of computers, CPU intensive applications would be assigned to computers with high processing capacity, while I/O intensive applications would be executed on disk or tape servers.

The task of initiating each application is achieved by visiting each node of the graph using a recursive pre-order visit of the nodes of the graph. Starting from the root node, each application is spawned and quickly provided with the parameter and parent/children nodes information, as explained in more detail below.

The selected algorithm works as follows. Starting from the root node, a pre-order recursive scheme is used to visit all the nodes of the tree. On the way down, the applications are spawned and the parameters are communicated. On the way up, each spawned application is informed of the process identification of its parent and children applications. In this way, while DSU is still spawning applications, the

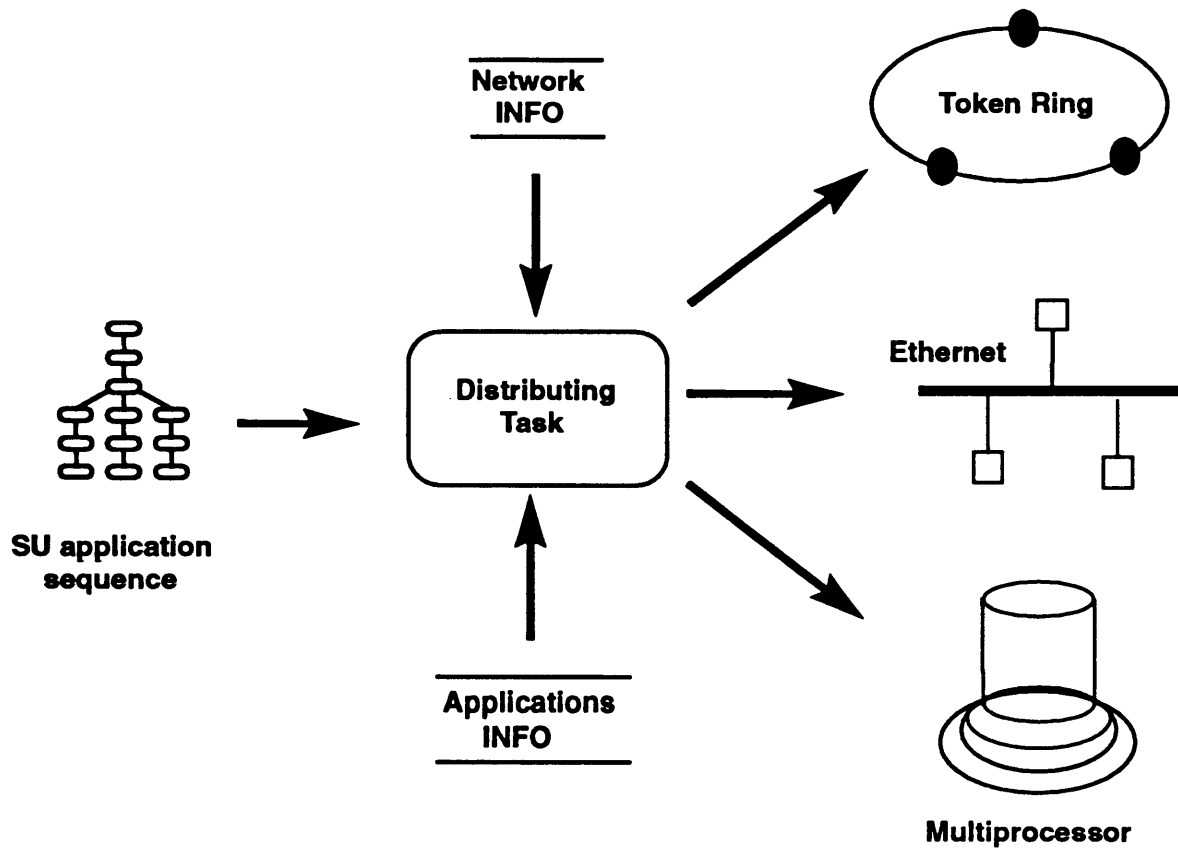


FIG. 4.3. The distributing task. This task should efficiently execute a sequence of SU applications on any platform (Ethernet, token ring and others), by combining the information about the applications and the components of the available network.

ones already spawned can simultaneously start reading/checking their parameters and getting ready to receive the information about their parent and children applications. The process identification of an application can only be obtained after the application has been spawned. Therefore, it is more convenient to provide each application with the process id of its parent and children application when the recursive algorithm is on its way up. Figure 4.4 shows the listing of this recursive algorithm.

```

RECURSIVE_VISIT_TO_SPAWN(Node T, parent_pid)
{
  if (T == NIL) return;
  mypid = spawn(T.application);
  Send_parameters(mypid, T.parameters)
  for (i = 0; i < max_number_of_children; i++)
    child_pid[i] = RECURSIVE_VISIT_TO_SPAWN(T.child[i], mypid)
  Send_parent_children_pid(mypid, T.child_pid, parent_pid)
  return(mypid)
}

```

FIG. 4.4. Recursive algorithm followed by DSU to spawn the applications associated with the nodes of a graph-tree representing a sequence of SU applications.

4.5 Data structures

As mentioned before, we abstractly represent a sequence of SU applications by a tree-like graph, whose nodes represent the applications and whose arcs indicate how the data must flow from the root node to the leaf nodes of the graph. This section describes the format used to represent those graphs (sequences) in a plain ASCII file and the data structure to be used to store the graph in memory once it has been loaded from a plain file or interactively created by the user through the interface tool.

4.5.1 Format to represent a graph in a plain ASCII file

To provide the user with an option to save a graph representing a sequence of SU applications for a posteriori use, we have defined a format to represent those graphs in a plain ASCII file. Since a lot of sequences of SU applications are already described by means of shell scripts, it was decided to keep that structure in this new format. However, since this system is able to handle sequences with several branches, some

changes to the format had to be made to uniquely identify the parent and children applications of each application. The format is described as follows.

```
# This sequence was written by dsu: Fri Dec 8 15:11:49 1995
susynlv:1,-1 nt="101" dt="0.04" ft="0.0" |
sushv:2,1 key="d2" a=".05" |
sumigps@boltzmann:3,2 tmig="0,4" vmig="1,2.0" |
suxmovie:4,3 perc="99" title="SUPSMIG" label1="Time" |
sugazmig@volterra:5,2 tmig="0,4" vmig="1,2.0" |
suxmovie:6,5 perc="99" title="SUGAZMIG" label1="Time" |
suxmovie:7,1 perc="99" title="Synthetic data" label1="Time"
exit
```

FIG. 4.5. Plain ASCII file representation of the multi-branch sequence of applications shown in Figure 1.4. It looks very similar to a shell script representing a sequence of applications with the exception that the applications names are specified along with the node number, the parent node number and optionally the hostname.

A plain ASCII file representing a graph of SU applications intended to be handled with DSU consists of a sequence of applications separated with bars |. Each application is accompanied by its respective parameters. The format to represent each application is as follows:

```
Application_name@hostname:node_number,parent_node_number  
parameter1=parameter1_value  
parameter2=parameter2_value ... parameter_n=parameter_n_value
```

The **application_name** field corresponds to one of the SU main application programs (e.g. **sufilter**). The **hostname** field is optional and indicates that this particular application must be executed in the machine with that hostname. The **node_number** integer field uniquely identifies the application in the file. The **parent_node_number**

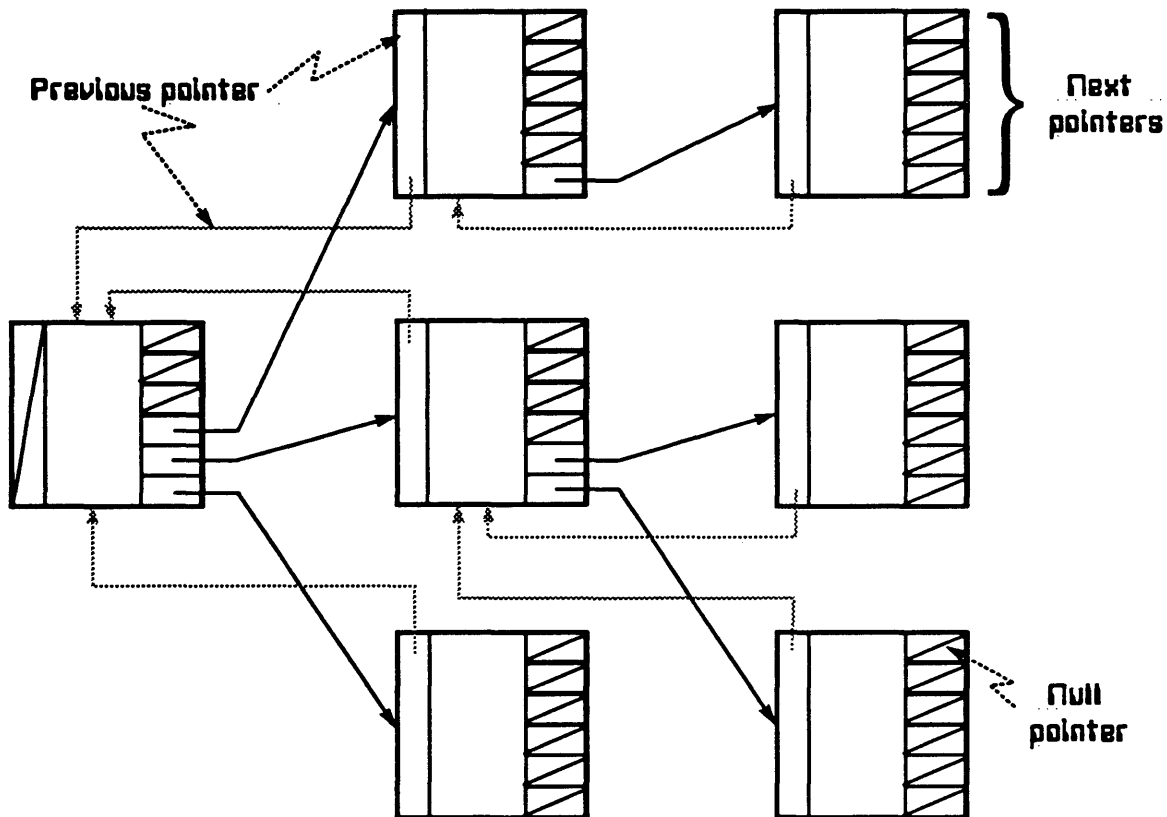


FIG. 4.6. Tree data structure used to store in memory a graph representing a sequence of SU applications

is an integer field that indicates the **node_number** of the parent node of that node. Since the root node does not have a parent node, by convention we selected -1 as its **parent_node_number**. Along with this specification, a list of the parameters provided for the application is specified using the following format:

parameter_name=parameter_value.

Figure 4.5 shows how the sequence represented by the graph shown in Figure 1.4 is stored in a plain ASCII file. Note that the **hostname** field has been set up for the **sumigps** and **sugazmig** applications, but since it is an optional field, it is not

explicitly specified in every application. Also, by inspecting the `parent_node` fields, it can be easily determined that the application `sushw` (node 2) has `sugazmig` (node 5) and `supsmig` (node 3) as children applications.

4.5.2 Data structure to represent a sequence of DSU applications

A tree data structure has been chosen to represent the graph or sequence once it has been loaded from a plain ASCII file or interactively created by the user. Each element of the tree will contain information describing the application it represents. It also contains pointers to its parent and children elements in the graph. Additionally, it maintains information about the graphical objects necessary for displaying the whole tree (sequence) through the GUI. Figure 4.6 is a graphical representation of the tree data structure to be used.

Chapter 5

DSU IMPLEMENTATION DETAILS

DSU is an event-driven graphic user interface whose main program switches among the states of the **graphical-tool** object shown in Figure 3.2. In summary, the DSU main program accomplishes some initialization tasks, and then enters into a loop to accept commands for building sequences of SU applications. When requested by the user, it executes and monitors the execution of the created sequence. To accomplish these tasks several subprocesses were implemented; their details are explained below.

DSU is implemented using TCL/TK for the GUI aspects and PVM for the process management and communication aspects. TCL/TK is a free, extensible scripting language with some extensions that incorporate an X11 toolkit (called TK) that make it easy to build graphical user interfaces [Ousterhout, 1994]. PVM is a software package that allows the utilization of a heterogeneous network of parallel and serial computers as a single computational resource [Geist *et al.*, 1994]. PVM provides facilities for spawning, communication and synchronization of processes over a network of heterogeneous machines. There are several message-passing systems available [Snair *et al.*, 1996], [Gropp *et al.*, 1994], [Carreiro and Gelernter, 1989], [Flower *et al.*, 1991]. PVM was selected mainly because it provides spawning capabilities and also has been successfully ported to several shared and distributed memory supercomputers. However, for the sake of easy portability, an intermediate library is used to avoid embedding direct PVM subroutine calls in the code.

The GUI has a similar look and feel to the X-window interface of PVM (XPVM) [Geist *et al.*, 1994]; actually, some portions of the code were extracted from the XPVM

code. The steps followed to obtain a preliminary implementation of DSU are the following:

1. A Graphical User Interface (GUI) was implemented using TCL/TK. Panels, Menus and buttons were designed to provide the user with options to create, edit and handle sequences of SU applications.
2. The procedures associated with the tasks of loading and saving sequences of SU applications in plain ASCII files were implemented.
3. The procedure for running the sequence of applications in the network was developed. The work here involves spawning a process for each application in the sequence and the distribution of the respective parameters. It also involved the introduction of changes inside the code of the SU applications to achieve the necessary synchronization and data communication, once the applications have been spawned.
4. Finally, some functions were implemented to provide the GUI with a mechanism to report about the status of the execution of a sequence.

5.1 The graphical user interface (GUI)

As we mentioned above, the GUI was implemented using the scripting language TCL and the X-window toolkit TK. The GUI is composed of a panel of buttons, message widgets and a scrollable canvas where the graph representing a sequence of SU applications is displayed. As in a typical X-window based GUI interface, each button, menu item and mouse button has a callback procedure associated with it. The

callback procedure is fired by X-window whenever the associated widget is clicked by the user (e.g. a button is clicked on). Chapter 6 is a full description of the GUI.

The GUI provides several options for handling the tree-graph representing the sequence of SU applications. Several drawing functions were developed to display that graph in the canvas area of the GUI. The graph is stored in memory as a tree. Each element of the tree represents an application and contains information about the graphical objects associated with that task, pointers to the application it is associated with, to the parent element and to the children elements in the tree. Information regarding process status is also stored in it.

In the initialization phase of the GUI, several tasks are performed. DSU creates a virtual machine in this phase, since a virtual machine must be created before spawning any PVM process. The GUI needs specific information about each SU application (e.g. names of its parameters) to properly create the panels to interact with the user. For example, when the user wants to specify parameters for a specific application, the GUI must prompt using the corresponding parameter name. To do that, during the initialization phase a linked list is created to store the important information about each application. Each element of this list represents a SU application and contains among other information:

- Application name;
- File name containing source code
- File name containing help information
- List of parameter names

A similar list is created to store information regarding each processing unit available in the network. The most important information is the machine name and its

speed.

The information in these two lists can be appropriately combined by the task in charge of executing the sequence to effectively spawn each application.

5.2 Loading and saving procedures

The loading procedure is in charge of reading a sequence represented in a plain ASCII file and storing it in memory. This procedure is based in a lexical analyzer program whose job is to read the input file and assemble the characters into tokens, and in a subprogram in charge of handling those tokens to assemble the sequence represented in the file. The lexical analyzer procedure implemented here is based on the one described by [Rochkind, 1985].

5.3 Distributed execution

In the chapter 4 we explained the algorithm to be performed when the execution of a sequence is requested. To implement this algorithm we have used the spawning facilities provided by PVM. When using PVM as the parallel machine (virtual), each spawned process is identified with a unique integer number, called **tid**. These identifiers are used by the applications to specify the recipient of a message to be sent or the source (sender) of a received message. Therefore, an important task of the distributing procedure is to inform each process associated with an application of the sequence the **tid** of its parent and children applications.

Starting from the root node, the distributing process performs a recursive pre-order visit of each node of the tree representing the sequence of applications. When a node is visited, the application associated with it is pvm-spawned and its **tid** (returned by the PVM spawning function) is saved to be returned later as the function return

value. Then, each subsequence starting at this node is recursively spawned. Each recursive call will return a **tid** that corresponds to a child subprocess of this task. When all the children **tids** are collected they are sent to the application along with the parent **tid**. To finish the visit to this node, the **tid** of this node is returned.

The PVM process spawning function allows us to provide the application parameters by means of the standard **argv** and **argc** arguments of the main program of the application being spawned. Before spawning an application, the distributing task builds the **argv** array and sets the **argc** value for that application and provides it to the PVM spawn function. In some cases the application associated with the root node requires a file to read input from. The name of that file is provided to the task after it has been spawned. Similarly, in some cases, the application associated with a leaf node of the graph requires a file to deposit its output. The name of that file is also provided just after the task has been spawned.

As we mentioned before, the user expects DSU to spawn each task on the machine best suited to its needs. To do this, an algorithm must be implemented. The input for this algorithm is the application name and the list of machines available. It will return the machine most appropriate for that application, most probably based on a cost matrix. A cost matrix is a matrix whose rows are the applications and the columns are the machines available. The entry (i, j) of the matrix indicates how expensive it is to execute application i on machine j . Many tasks can have a single machine as the one best suited for them, if those tasks happen to be in the same sequence, the algorithm must be able to avoid starting all of them on that particular machine.

Since the PVM spawning task performs its own algorithm to assign applications to machines, in the current DSU implementation we rely on PVM to accomplish that task. No optimization is performed by DSU itself. Beyond the capabilities provided

by PVM, DSU needs to make sure that applications requiring windows capabilities are forced to be initiated on the same machine where the GUI interface is running. To identify that an application requires graphical capabilities, at the initialization phase DSU loads information about each one of the SU applications. At that time a flag is set ON for applications requiring those capabilities. Additionally, DSU allows the user to explicitly designate the name of a host where a specific application must be executed.

The next section describes the modifications introduced in the applications. These modifications are needed so that, after being PVM-spawned, applications can receive the information about the **tids** of their parent application and their children applications. The **tid** of the parent application is used to receive traces (data) from the parent application. The **tids** of the children applications are used to send the traces to those applications.

5.4 Modifications in the applications

As mentioned in chapter 2, the structure of an SU application is divided into three phases: parameter reading and initialization; trace handling and processing; and the cleanup stage. Fortunately, each SU application performs these phases by invoking specific subroutine library functions. Therefore, to adapt SU applications to the new communication scheme, we only had to rewrite a few functions. Next, we explain the changes made to the selected functions and why they had to be modified.

As we explained above, right after DSU spawns an application, it also sends to that application the information about the **tids** of its parent and children applications. Fortunately, each SU application initiates its execution by invoking the subroutine *initargs()*. This function is contained in the subroutine library *libpar.a*. The corres-

ponding *receive()* function calls were added to this function so that each spawned application receives those *tids*. Additionally, since the application may require file names for reading or writing its input or output, appropriate receiving functions were added too.

Also, as part of the monitoring facilities provided with DSU, every application is requested to report its progress in a specific file. To achieve that, the output destined for the *stderr* file is redirected to a specific file by calling the subroutine *InitLog()* in the *initargs()* function.

When an SU application turns to the second phase, several calls to the functions *fgettr()* and *fputtr()* are performed. These functions belong to the subroutine library *libsua*. The function *fgettr()* reads a seismic trace from the standard input file and *fputtr()* writes a seismic trace to the standard output file on Unix systems.

At the beginning of each SU application the following macros are defined:

```
/* DEFINES */
#define gettr(x)      fgettr(stdin, (x))
#define puttr(x)     fputtr(stdout, (x))
```

In the second phase of an SU application these macros are used to specify the function to get data from the standard input (*gettr()* in this case) and to write data to the standard output (*fputtr()* in this case). In this way, by changing these macros, which are indeed defined in a file included by each SU application, we can change the way applications read and write without modifying their source code directly.

In regular SU, applications always read the data from the standard input file and write the output to the standard output file, so a static definition like the above works for any application. In DSU, however, only the application associated with the root node of the graph representing the sequence reads from the standard input file and only the ones associated with leaf nodes of that graph write to the standard output file.

Each of the other applications must get the data from its parent application and deliver it to its children applications by means of PVM messages. To handle this situation without introducing any changes in the application source codes, we have redefined the macros *gettr()* and *puttr()* so that they get expanded as function pointers, as we show here:

```

/*      DSU modifications to segy.h
*/

#ifdef gettr
#undef gettr
#endif
#define gettr(x)      (*DsuGet) (stdin, (x))

#ifdef puttr
#undef puttr
#endif
#define puttr(x)      (*DsuPut) (stdout, (x))

```

In this way, depending on its location in the graph, each application, after being spawned receives some information from the spawning task that allows it to dynamically decide which functions are to be used to read and deliver the data. In the *initargs()* subroutine, these function pointers are assigned in the following way. If the application is associated with the root node of the graph representing the sequence, it does not have a parent application, so if it performs *gettr()*, then the regular *fgettr()* function must be used. If the application is associated with an internal node of the sequence, then the function pointers must be assigned with *dsufgettr()* and *dsufputtr()* respectively. If the application is associated with a leaf node of the graph, and it happens to perform *puttr()*, then the pointer must be assigned with the function *fputtr()*.

The functions *dsufgettr()* and *dsufputtr()* allow the applications to obtain their data, through messages, from the parent application and send the processed data to each one of the children applications. They are implemented using PVM and belong to a new library added to SU called *libdsu.a*. This library includes some other functions that can be used to communicate the data between the applications.

In the third phase of the execution of a SU application, cleanup tasks are performed. If the application reaches a successful execution, it returns the value of the predefined ANSI macro `EXIT_SUCCESS`. In the event of unsuccessful termination, it returns the value of the macro `EXIT_FAILURE`. These two macros were redefined so that a function is called instead:

```
#ifdef EXIT_FAILURE
#undef EXIT_FAILURE
#endif

#define EXIT_FAILURE (DsuExit(&ThisDsuTask, MSGERR))

#ifdef EXIT_SUCCESS
#undef EXIT_SUCCESS
#endif

#define EXIT_SUCCESS (DsuExit(&ThisDsuTask, MSGEOF))
```

In this way, when an SU application finishes, the function *DsuExit()* is performed. In successful cases, this function will notify the children applications (if there are any) of the end of processing. In unsuccessful cases, the children applications are notified that no more data will be forwarded. In both cases, the file associated with the *stderr* output is closed and a message indicating the completion status is sent to the GUI. When the GUI receives a message indicating an abnormal end, it stops the execution by killing all the spawned tasks and resetting PVM by performing the *pvm_reset()* function.

5.5 Monitoring task

Since DSU is an X-window graphical user interface, the interaction with it has to be done through events. Since it is not desirable to block the GUI with the monitoring process, several alternatives are possible. The first one is to initiate an independent monitoring process. The second alternative is have the user request report of the progress once in a while, in which case the GUI can fire a subprocess that provides

snapshot information. For simplicity, in the current implementation we followed a combination of those alternatives. It works in the following way. At interval times, after spawning all the applications of a sequence, DSU will fire a process in charge of checking for messages from the applications. Before finishing their execution, the applications have to send a message to the GUI indicating their completion status. In case of messages indicating successful completion the user is informed, but in cases of abnormal completion, the whole execution is aborted and the user is advised to check the log file for the application abnormally ending. DSU keeps track of the number of applications spawned and when all of them have completed their execution, a message indicating the elapsed time taken by the execution is shown to the user.

The mechanism to create the log file is accomplished as follows. When DSU is requested to execute a sequence, it immediately performs the subroutine *CleanLog()*. This subroutine will read the value of the environment variable **DSULOG** to determine the name of the directory where the user wants to accumulate the standard error output of each application.

Similarly, just after being spawned, an SU application immediately invokes the subroutine *InitLog()*. This subroutine uses the directory name stored in the environment variable **DSULOG** to open a file in it and redirects the output destined to the *stderr* file of that application. The name of this file is then sent to the GUI, so that it can be shown to the user when requested. Usually, the directory name pointed by the variable **DSULOG** can be accessed by all the machines available; otherwise the directory */tmp* is used. PVM provides functions to redirect the *stderr* output to a specific file. However, due to portability reasons, we decided to employ our own mechanism to avoid direct dependencies with PVM.

The monitoring task would be facilitated if PVM provided active messages. With

this kind of message the applications would be able to request the GUI to inform the user about their progress without having the user explicitly fire an extra process in the GUI. Since near future versions of PVM are expected to provide this kind of message, the implementation of a more effective monitoring system has been postponed until those features are available.

Chapter 6

DSU GRAPHICAL USER INTERFACE

DSU provides a graphical user interface (GUI) tool for creating, editing, setting parameters and executing a graph representing a sequence of SU applications on a multiprocessor environment. It allows the user to set up application parameters, input/output filenames, and other similar tasks related to the applications. It provides a context sensitive help about almost every component of SU. It also allows the user to interactively check the source code of selected main programs and subroutine library functions.

Figure 6.1 shows the DSU main interface. The upper part of the interface window is a message widget, where the current status is displayed. Beneath it, there is a panel of buttons that allows the user to build and handle the sequence of applications. The button allows the user to save the current sequence of applications. It will allow the user to surf over the available file system to select the appropriate place to store the sequence. The sequence is saved in a plain ASCII file using the format that was described in section 4.5.1. The button will display the information contained in each element (application), belonging to the subsequence starting at the current or active node on the screen. This information includes application name, parameter names and values. The button allows the user to load a sequence of applications represented in a plain ASCII file.

The second group of buttons are intended to handle single application nodes (single nodes of the graph). The and buttons prompt the user for an

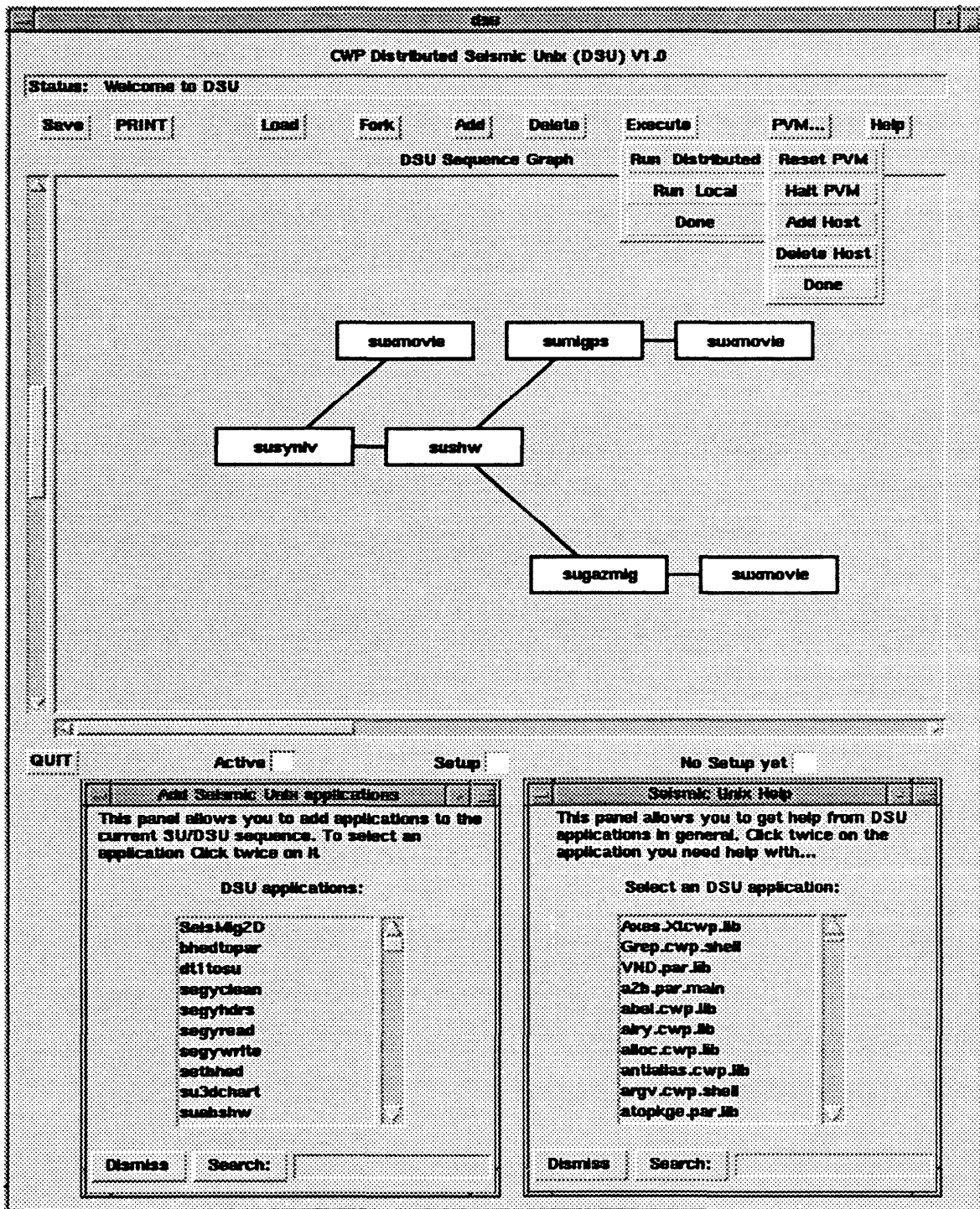


FIG. 6.1. DSU Graphical User interface and some of its menus.

application name to be added to the graph just after the current one. The current application is highlighted with a green color. A node of the graph becomes the current one if it was the last one inserted, if the user clicked the left mouse button on it or if one of its children nodes was removed. The `fork` button allows the user to create branches in the sequence. The `delete` button removes the current node from the graph. DSU ensures that nodes with multiple branches deriving from them can not be removed.

The third group of buttons has to do with executing the represented sequence. The `execute` button is actually a menu with two options. The first option allows the execution of the sequence with regular SU (under pipeline communication); since it is not possible to execute multi-branch sequences via pipe-file communication, only the main branch of a multi-branch sequence will be executed. This is useful when we want to test only the main branch of a sequence. The second option allows a distributed execution of the sequence. By selecting it, the user will fire the DSU distributing task. This task will initiate each application of the sequence on the machine best suited to its needs and will setup the communication among them accordingly to the arcs of the graph representing the sequence.

The other button associated with the execution of the graph (sequence) handles the interface with the message-passing system, PVM. As shown in Figure 6.1, the button labeled `PVM` is a menu with options to start, stop and reset PVM. Additionally, it allows the insertion and removing of machines to the virtual machine.

The farthest right button of the interface is the `help` button. This button will prompt the user for the entity he/she needs help with. There, one can request help about almost anything related with SU: library functions, main programs and others. Once the user has selected the entity, a scrollable window containing the help for that

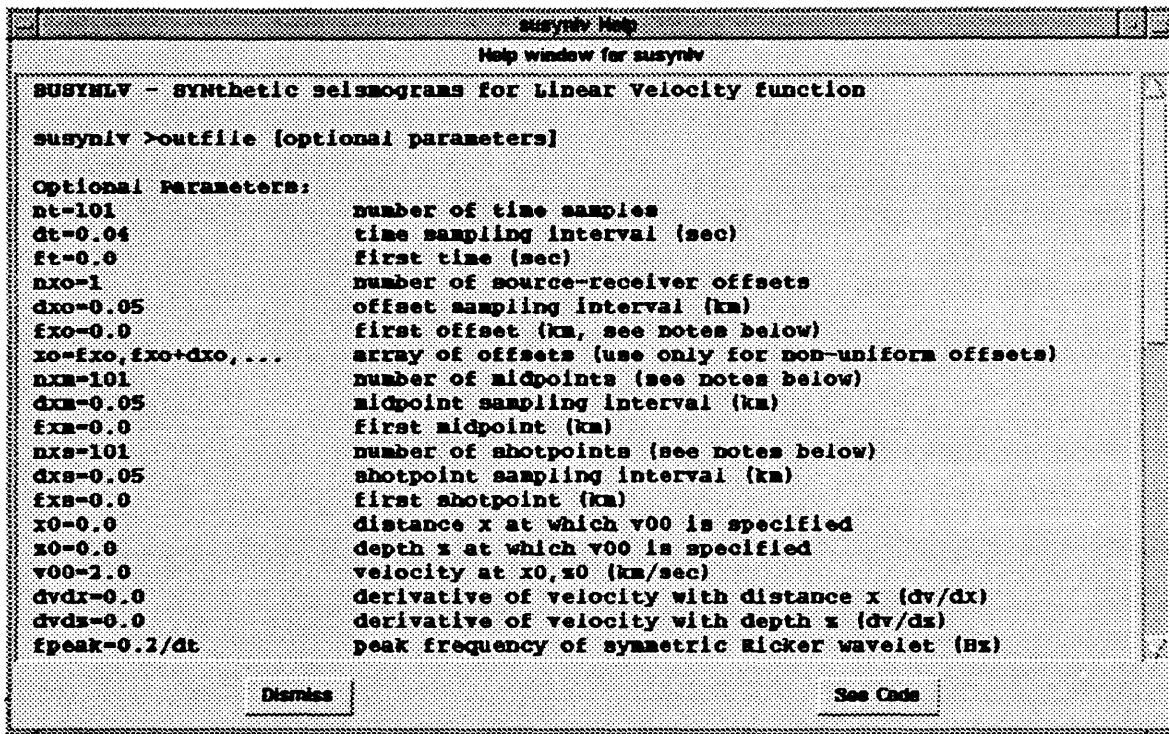


FIG. 6.2. Scrollable help window for the application `susynlv`. It is obtained by clicking the right button of the mouse on the icon representing that application.

entity will be shown. When obtaining help for SU main applications, an `see code` button is provided to be able to see the source code of the application.

6.1 Mouse buttons

In addition to the buttons on the screen, the buttons on the mouse are an essential part of the user interface. Clicking once on the icon representing an application with the right button of the mouse will bring up a scrollable window containing the help information for that application. Figure 6.2 shows the help window for the application `susynlv`. This window provides the `see code` button which allows the user to take a look at the actual source code of that application.

Get parameters for susynlv (node 1)			
Enter parameters for susynlv. You can type in the various entries and use tabs to move circularly between the entries.			
nt:	101	dt:	0.04
ft:	0.0	x0:	void
nxc:	1	dx:	void
fx:	void	nxs:	void
dx:	void	fxs:	void
num:	101	dxm:	0.05
fxm:	0.0	nxs:	void
dxs:	void	fxs:	void
x0:	void	z0:	void
v00:	1.00	dvdx:	0
dvdz:	0.2	fpeak:	void
is:	void	er:	0
ob:	1	tmn:	void
ndpfz:	void	smooth:	1
verbose:	void	ref:	0,5;1.0,5;2,1.0;2.5,1.5;3.0,1.0;4.0,5;
Dismiss		Save	
stdin		stdout	
Hostname		Options	

FIG. 6.3. Window for entering parameters values for the application `susynlv`. It is obtained by clicking twice on the left button of the mouse on the application icon.

Clicking twice on the left button of the mouse on an application icon will bring up the window that allows the insertion of values for the parameters of the application represented by that icon. Figure 6.3 is an example of a window to get parameters for the `susynlv` SU application. This window has a `hostname` button that allows the specification of a hostname for that application; also, by using the buttons `stdin` and `stdout`, filenames can be entered to setup the file names for the standard input and output, respectively, of that application. Only the root node is allowed to have

the standard input redirected; each other application in the sequence must expect its input from its parent node. Similarly, only leaf nodes of the graph (sequence) are allowed to have the standard output redirected. The button allows the insertion of some parameters under the format: *-option option_value*; this syntax for parameters is understood for some X-Windows related applications that are part of SU, like *suzmovie* for example. Clicking the button will save the parameter values entered; alternatively, hitting the *Enter* key on any parameter field will automatically save all the parameter values entered too. Using *void* as a parameter value, indicates that the default value must be used.

Finally, clicking and holding the middle button of the mouse on an application icon will allow the user to move that icon to any position inside the drawing canvas.

6.2 A typical session with DSU

Figure 6.4 shows a typical session with the DSU graphical tool. The user usually loads or creates a sequence of applications and then spends most of the time providing parameters to the applications, adding and removing single applications to the sequence and executing it. The user will occasionally save the current sequence with some specific parameters and then modify them and save again. Next, we discuss some typical actions when using DSU.

There are two ways to start the creation of a SU application graph, the first one is to use the load button to load a previously created sequence and work from there. The second one consists of clicking the or buttons to bring up the panel containing all the available SU applications. Then by clicking twice on the desired application name, that application gets added to the sequence just after the current one.

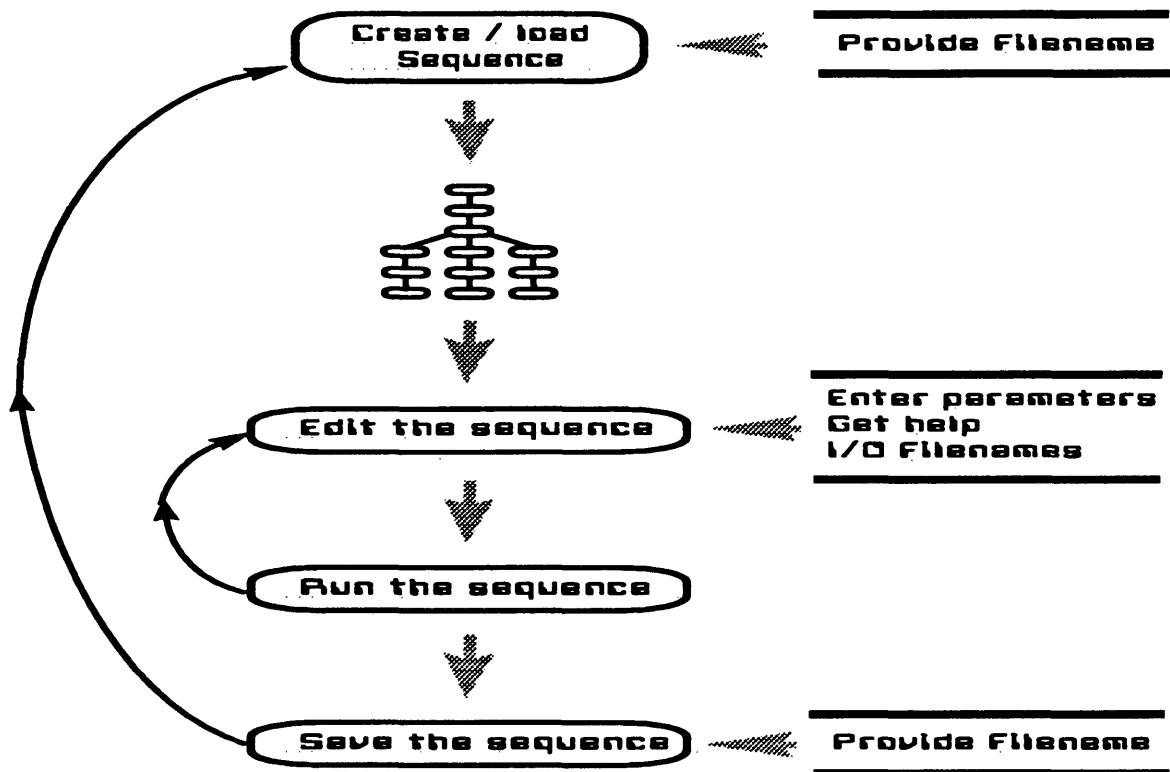


FIG. 6.4. Typical user session with the graphical tool of DSU.

The GUI provides a button to select the type of execution that the user wants DSU to perform. Clicking this button will raise a menu with two options for execution: one is called **Local** and the other **Distributed**. The **local** option corresponds to executing the sequence as in a regular Unix-shell environment. The **Distributed** option will execute the sequence over the multiprocessor environment available.

Chapter 7

ESTIMATING THE PERFORMANCE OF DSU

In the introduction we mentioned that performance is not the only important issue to consider when going to parallel processing. However, it is always important to estimate performance. This chapter explains some basic facts that lead us to expect better performance out of DSU. Also, a simple performance model predicts how much speedup could be obtained by executing a sequence of SU applications on several machines instead of just using one.

The most frequently used performance metric in parallel processing is **speedup**. It is defined as sequential execution time over parallel time. There are several interpretations for **speedup**[Ortega and Voight, 1991] that are important to explain here. When the speedup is measured against a commonly used sequential algorithm it is called absolute speedup [Hembold and McDowell, 1990]. Another widely used interpretation is the relative speedup, which uses the uniprocessor execution time of the parallel algorithm as the sequential time. The performance of many parallel algorithms varies with the number of processors available (scalability) [Gustafson, 1988]; this is one of the reasons for which relative speedup is the speedup commonly used in performance studies.

7.1 Why we expect speedup

In [Hembold and McDowell, 1990] several factors for obtaining superlinear speedup are explained in detail. Here, those arguments are used to claim that DSU will in fact lead to better performance.

The first factor is the performance gained by reduced overhead. The amount of time spent in operations related to resource management depends highly on the number of processes residing (concurrently running) in a processor. Therefore, executing large sequences of SU applications in a single processor translates to a large amount of time dedicated to overhead. Roughly speaking, we can say that by distributing a sequence of n applications on p processors we are also distributing the overhead involved in their execution. Of course, by doing, so we are introducing a communication overhead not present in a single processor. However, this debt is paid off by the opportunity of handling bigger sequences of applications and input sizes that are otherwise intractable in a single machine.

The second important factor is concerned with memory latency. There are various factors related to memory latency that can highly impact the execution of a sequence of SU applications in a single machine. Since there are several processes running, each one will get assigned fewer pages of the main memory, and therefore more page faults are expected. Similarly, if the available cache memory is shared among the processes (not flushed with context switch) the cache miss ratio (probability of not finding a referenced memory word in the cache) and the need for context switch also increase as the number of processes increases.

Finally, we must point out the fact that when executing sequences of SU applications in a single machine, the communication among the applications is done via pipeline files. Pipeline files are usually implemented by means of memory buffers, which translate into an extra memory overhead that in some cases (large amount of data) might be even worse than the one introduced by communicating the data to other machines. This also can highly limit the number of applications in a sequence.

7.2 A simple performance model

As in any performance model, we have to make some assumptions that in most cases lead us to consider the worst case (when we might expect the worst speedup). Since only single branch sequences can be handled with regular SU, the analysis is devoted to those kind of sequences. We will handle the following parameters:

1. Number of traces to process: n
2. Size of a single trace in bytes: b
3. Number of applications in the sequence: m
4. Number of processors available: p
5. Network bandwidth.

We assume that we have as many processing units as applications in the sequence (one application per processor or $m = p$). Also, we always assume $n > p$. Regarding the communication network among the processors, we do not consider the topology, we just take into account its bandwidth; that is, how much data can be interchanged among all the processors per second.

A sequence of SU applications is usually composed of many different types of applications. Some of them are CPU bound, others I/O bound, etc. In this model, we represent the load of each application of a sequence with a simple trace scaling application (multiply each input trace by a scalar number). The reason for making this assumption is that this application represents one of the least computationally expensive applications in DSU and therefore the worst case for the communication/computation ratio.

<i>Processor #</i>	τ_1	τ_2	τ_3	\dots	τ_p	\dots	τ_n	\dots	τ_{n+p-1}
<i>App₁</i>	t_1	t_2	t_3	\dots	t_p	\dots	t_n		
<i>App₂</i>		t_1	t_2	\dots	t_{p-1}	\dots	t_{n-1}	\dots	
<i>App₃</i>			t_1	\dots	t_{p-2}	\dots	t_{n-2}	\dots	
\dots									
<i>App_p</i>					t_1	\dots	t_{n-p}	\dots	t_n

Table 7.1. Pipelining traces t_j through the applications App_i . Each application multiplies each trace by a scalar. Total time is $(n + p - 1)\tau$ (τ is the time consumed by multiplying a trace by a scalar).

The test is very simple; suppose we have a sequence of p applications (therefore p processors) and we desire to process n input traces. If we denote the time of scaling one seismic trace by τ , then the total time expected to complete the whole processing in one machine would be:

$$T(1) = p * n * \tau$$

Table 7.1 shows a Gantt chart [Lewis and El-Rewini, 1992] representing the load of each processor in time during a distributed execution of p applications (using p processors), to process n traces (t_1 through t_n). Each column of that table represents the period of time spent scaling a single trace. This time is denoted by τ . Each row is associated with an application or processor and indicates when, during the execution, that application (processor) is active.

From that table we can deduce that the total time to complete the processing of the n traces is:

$$T(p) = (n + p - 1) * \tau \approx (n + p) * \tau$$

Therefore, the absolute speedup obtained, neglecting the communication cost, is given by:

$$speedup = T(1)/T(p) = n * p / (n + p)$$

This speedup is very good if n is large. For example for an input size of 500 traces ($n = 500$) and a sequence of 10 applications ($m = 10$), with 10 machines ($p = 10$) available, we might get a maximum absolute speedup of almost 9.8. This is a very good improvement if we also consider the efficiency, which is defined as speedup divided by number of processing units used. In this example we obtain almost 1, which is the ideal case.

The number of bytes simultaneously communicated reaches its maximum when the last application of the sequence receives the first trace. It stays there until the first application of the sequence processes the last trace. This maximum number is $p * m$, where m is the size in bytes of a trace. As long as this number stays far below the bandwidth of the communication network available, the impact of the communication in the performance will be negligible. However, if it gets closer to the bandwidth, the impact can be considerable. Let us check the impact under an Ethernet network. Let us suppose that the traces considered in the previous example are of 4K bytes each. The pipelining of the 500 traces through the 10 processes will require sending $10 * 4K = 40K$ bytes 490 times ($n - p$). Ethernet has an estimated bandwidth of 1000KB/sec in regular conditions, so the cost of transmitting 40K simultaneously would cost 1/25th of a second. Thus, we can estimate the communication overhead of pipelining those 500 traces through the 10 applications as 20 seconds ($490 * 1/25$).

7.3 Computational experiments

This section presents some timing results obtained by executing sequences of SU applications using DSU. The theoretical performance analysis presented above was based in a sequence composed of just trace scaling applications, since that kind of sequences represents the worst case regarding the communication/computation ratio.

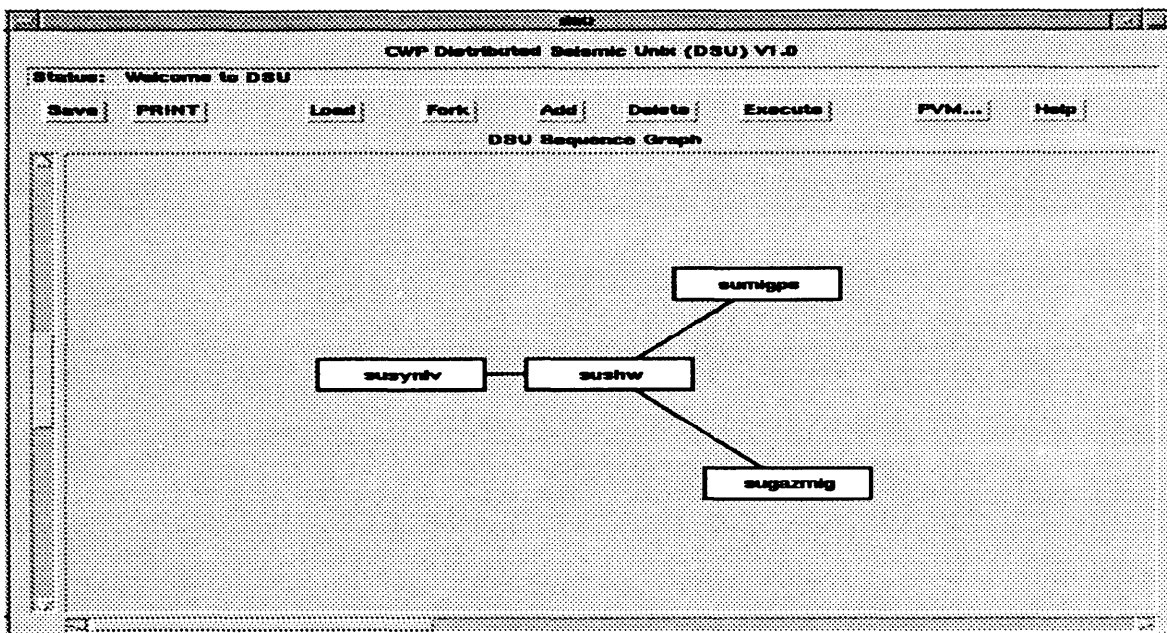


FIG. 7.1. Sequence 1 used to benchmark DSU using four Indigo II workstations connected via Ethernet.

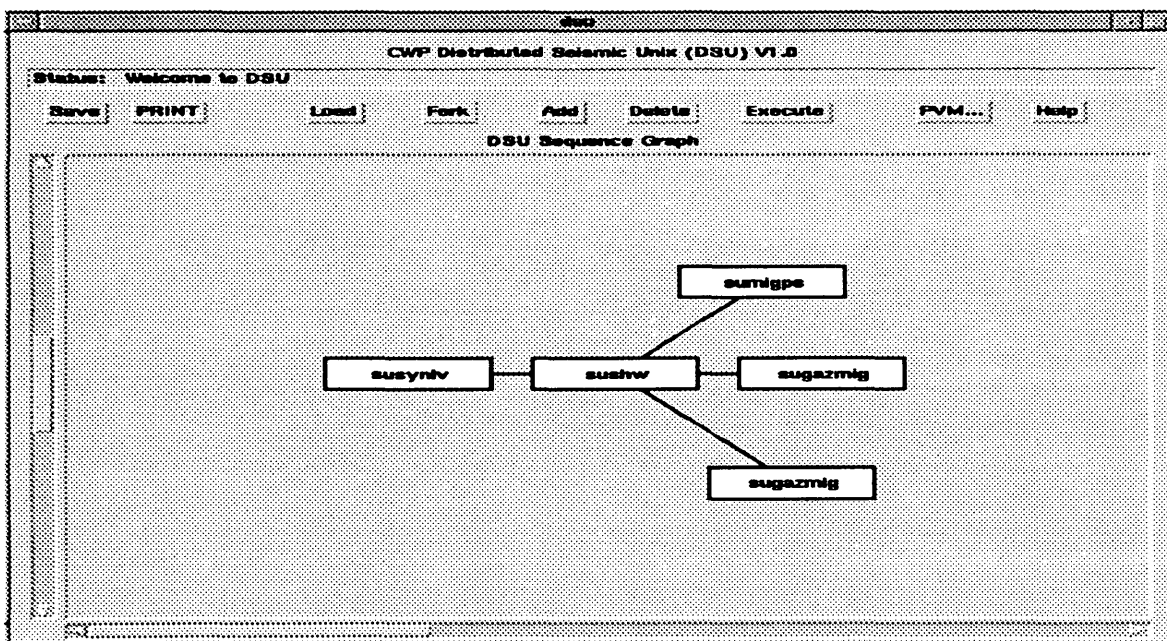


FIG. 7.2. Sequence 2 used to benchmark DSU using four Indigo II workstations connected via Ethernet. Obtained by adding a sugazmig application to sequence 1.

In contrast, the timing results shown here were obtained by executing sequences composed of frequently used numerical intensive SU applications. We consider it to be more important to provide an idea about the performance of DSU with commonly used sequences, instead of showing the worst cases involving a not very common sequence.

The sequences used in the experiments are relatively small, therefore the speedup due to pipelining parallelism is expected to be small. What we intend to explore with these experiments is three fold:

1. How the speedup changes when varying the input size;
2. How the speedup changes when adding a new application to a sequence;
3. The overhead time involved when executing a sequence with DSU.

Figures 7.1 and 7.2 show the sequences selected for the experiments. To measure the variation of speedup with regard to the input size (number of traces to process), we executed sequence 1 with two different medium size inputs. The first input set contains 300 traces of 300 samples each, while the second is composed of 500 traces with 500 samples each.

To measure the variation of speedup when adding a new application to a sequence, we added a new application to sequence 1, to obtain sequence 2, which is shown in Figure 7.2.

To measure the overhead time with DSU we compare the time obtained executing sequence 2 with DSU, with the best time we can expect to obtain if the communication time is ignored. For example, when DSU executes sequence 2, the `susynlv` and the `sushw` applications are executed sequentially and then, the other 3 migration programs are executed simultaneously. Ignoring the overhead, the ideal time expected with DSU

Machines	Traces/size	Elapsed time (sec)	Speedup (row 1)	speedup (row 2)
1	300/300	112	-	-
1	300/300	61	-	-
4	300/300	50	2.24	1.22

Table 7.2. Processing 300 traces of 300 samples each with sequence 1. Rows 1 and 2 show the sequential time on 100 MHZ and 200 MHZ machines respectively. Row 3 shows the time obtained with DSU using three 200 MHZ and one 100 MHZ Indigo II machines. Columns 4 and 5 show absolute speedup with respect to the 100 MHZ machine and to the 200 MHZ machine respectively.

Machines	Traces/size	Elapsed time (sec)	Speedup (row 1)	speedup (row 2)
1	500/500	464	-	-
1	500/500	265	-	-
4	500/500	200	2.5	1.30

Table 7.3. Processing 500 traces of 500 samples each with sequence 1. Rows 1 and 2 show the sequential time on 100 MHZ and 200 MHZ machines respectively. Row 3 shows the time obtained with DSU using three 200 MHZ and one 100 MHZ Indigo II machines. Columns 4 and 5 show absolute speedup with respect to the 100 MHZ machine and to the 200MHZ machine respectively.

should be: $T_{susyntv} + T_{sushw} + MAX(T_{sugazmig}, T_{sumigps})$ (By experiment, we know that $MAX(T_{sugazmig}, T_{sumigps}) = T_{sugazmig}$, for any number of traces).

Since the selected sequences have more than one branch, to run them sequentially we had to break them in several pieces or shell scripts. The total sequential time is then given by adding the execution time of each piece. The tests were executed using four Silicon Graphics Indigo II machines; one of them running at 100 MHZ and the other 3 running at 200MHZ. The results are compared, in terms of absolute speedup, to the ones obtained by executing the same sequences sequentially (by means of shell scripts).

Table 7.2 and Table 7.3 show the results obtained by processing sequence 1 with the two different input sizes (300 and 500). Rows 4 and 5 of those tables show the absolute speedup obtained by executing sequence 1 under DSU with respect to the sequential time in two different machines. In any case, we can see that as the number of traces went from 300 to 500, the speedup improved. This means that even though we are increasing the communication load (more and bigger seismic traces have to be communicated), its effect on the final elapsed time slightly decreased as the number of traces increased.

Machines	Traces/size	Elapsed time (sec)	Speedup (row 1)	speedup (row 2)
1	300/300	201	-	-
1	300/300	109	-	-
4	300/300	58	3.4	1.87

Table 7.4. Processing 300 traces of 300 samples each with sequence 2. Rows 1 and 2 show the sequential time on 100 MHZ and 200 MHZ machines respectively. Row 3 shows the time obtained with DSU using three 200 MHZ and one 100 MHZ Indigo II machines. Columns 4 and 5 show absolute speedup with respect to the 100 MHZ machine and to the 200 MHZ machine respectively.

Machines	Traces/size	Elapsed time (sec)	Speedup (row 1)	speedup (row 2)
1	500/500	857	-	-
1	500/500	466	-	-
4	500/500	262	3.27	1.7

Table 7.5. Processing 500 traces of 500 samples each with sequence 2. Rows 1 and 2 show the sequential time on 100 MHZ and 200 MHZ machines respectively. Row 3 shows the time obtained with DSU using three 200 MHZ and one 100 MHZ Indigo II machines. Columns 4 and 5 show absolute speedup with respect to the 100 MHZ machine and to the 200 MHZ machine respectively.

Traces/size	susynlv	sushw	sugazmig	Ideal	Real	Overhead
300/300	0.8	0.8	51.17	52.7	58	5.3
500/500	1.5	1.5	221	224	262	38

Table 7.6. Estimating the overhead time with DSU. Columns 2, 3 and 4 show the time taken by each application of sequence 2. Column 5 shows the time expected if we use DSU to execute sequence 2. Column 6 is the real time obtained and column 7 is the difference (overhead).

Table 7.4 and Table 7.5 show the results obtained by processing sequence 2 (sequence 1 with an extra application) with the two different input sizes. As in the previous tables, rows 4 and 5 show the absolute speedup obtained by executing sequence 2 under DSU with respect to the sequential time in two different machines. By comparing the speedups obtained in Table 7.2 with the one in Table 7.4 (similarly, the ones in Table 7.3 with the ones in Table 7.5), we can see that that by adding an application to sequence 1, the speedup notably improved.

We experimented with sequence 2 to estimate the overhead impact when executing sequences of SU applications with DSU. Sequentially, sequence 2 generates input, applies **sushw**, a **sumigps** migration and then two consecutive **sugazmig** migrations to the generated input data. Under DSU, the last three migration processes are executed simultaneously, therefore we should expect the total time with DSU to be the time to generate the data (**susynlv**), plus the time to execute **sushw**, plus the time to accomplish the most expensive of the three migration processes (since they are done in parallel, only the time of the one lasting the longest is considered), plus the overhead (communication, process spawning) time.

Column 5 of Table 7.6 shows the ideal time (best) we should obtain when executing sequence 2 with DSU. The ideal time is calculated by adding the time taken by

each individual application in the sequence and considering only the time of the one lasting the longest, when several are executed in parallel. For instance, in sequence 2, the last three migration process are executed in parallel, so only the time of the one lasting the most (**sugazmig**) is considered in Table 7.6. Column 6 of Table 7.6 shows the real time obtained in practice. The last column of that table shows the difference between the values in rows 5 and 6. This difference is the overhead time spent by DSU spawning tasks and by the applications communicating the traces. Even though Ethernet was the underlying network, this overhead became only around 1% percent for 300 traces and 6% for 500 traces.

It is important to point out here, that the network utilized to connect the machines used in these experiments was Ethernet, therefore, much better results should be expected if a faster network is utilized.

Chapter 8

SUMMARY AND FUTURE WORK

Users of a well-designed distributed system should perceive a single, integrated computing facility even though it may be implemented by many computers in different locations [Couloris *et al.*, 1994]. Distributed Seismic Unix was designed with this idea in mind. The result has been a friendly interface designed to simplify the task of creating, editing and executing sequences of SU applications on a multi-processor environment. In DSU, the user specifies the sequence of applications by drawing a graph whose nodes represent applications and whose arcs describe the way in which the data flow among the applications.

DSU is a notable enhancement to the capacities provided by the already useful SU software. The system has been carefully designed to be portable and easy to extend. The system was designed following object oriented techniques. As a result, a static and a dynamic model describing the system were generated. These two models represent an external view of the implemented system which can also be used as a tool to understand and extend the implemented system. Without these models, the task of extending a system like this would be very difficult.

The implementation of DSU involved the combination of several aspects of computer science. Concepts like distributed computing, load balancing, graph manipulation, computer graphics concepts for graphical user interface development and graph drawing issues were addressed to obtain a final implementation of the system.

The ability of handling multi-branch sequences of SU applications is one of the most important features provided with the system. This feature allows the user to

simultaneously execute several copies of a single branch sequence of applications, each one with different parameters.

Portability of the final product was always a top priority in both the design and implementation of DSU. Even though the current implementation of DSU utilizes PVM for process handling and the communications issues, an alternative message passing system can be used with little work, since an intermediate library was also developed to avoid embedding direct calls on PVM functions in the code.

DSU is composed of four independent subsystems: the graphical user interface, the saving and loading task, the distributing task and the monitoring task. These tasks can be independently upgraded without substantially affecting the other subsystems. They are intended to handle the structure representing the graph or sequence of SU applications.

When DSU executes a sequence of SU applications, it attempts to execute each application of the sequence on the machine best suited to its needs. The current implementation of DSU provides a mechanism that allows the user to specify the host in which a specific application of the sequence must be executed. This mechanism allows the user to explicitly enforce a better load balancing and resource usage. However, a more effective strategy might be developed. DSU has been designed and implemented so that the introduction of a new strategy in this regard would only affect a few lines of source code of the distributing task. Several simple algorithms based on a cost matrix were tested during the implementation, however, the results obtained did not substantially improve the one obtained by just relying on PVM. Furthermore, load balancing algorithms based on a cost matrix are very much dependent on static information and not on dynamic information (current load of a machine) that is usually the best information for good load balancing. Obtaining dynamic information is not

easy and usually introduces some inconvenience that makes useless the information obtained. For example, at any given moment a machine might be idle and it can be selected to execute a heavy job; however, by the time the task is started, it might be busier than expected and then the time spent selecting that machine is considered useless.

In spite of all the pitfalls for load balancing, we still can claim that DSU helps the user to make a more effective use of the resources available in the sense that disk servers, hard copy servers and machines with high computational capacity can be simultaneously used to execute sequences of SU applications. Even though in this preliminary implementation, the user still has to help in the load balancing, there will still be speedup as we can deduce from the brief performance analysis shown in chapter 7.

8.1 How DSU helps other research

The design, development, implementation and testing of parallel algorithms of geophysical applications is currently an important research area. The tools available for accomplishing those steps are not easy to use and in general do not allow the incorporation of other support applications like the ones provided by SU. In fact, most of the numerically intensive geophysical applications are developed and intended to be used in a stand alone fashion; the data are manually taken to the specific machine where the application was developed, they are processed and the results are sent back to allow continuing with the regular processing sequence. DSU is the starting point to provide an environment for integrating the execution of parallelized numerical intensive applications with other less expensive applications. So far, various 2D and 3D migration applications that were network-parallelized using PVM, can be used

with DSU as any other typical SU application, without the hassle of dealing directly with PVM.

An aspect of particular importance that can be a potential beneficiary of this work on DSU is parallel computers performance analysis. As pointed out by [Bell, 1992], “no matter what measure is used to understand a computer, the only way to understand how a computer will perform is to benchmark the computer with the applications to be used”. Given that in DSU the communication of data among the applications is done through messages, it becomes an effective application for benchmarking communication aspects of a multiprocessor environment whose components are connected with a particular type of network.

8.2 Future research

There are several research directions that can be followed at this stage of the development of Distributed Seismic Unix. A major question to be explored is how the concept of active messages can help to dramatically improve both load balancing and the monitoring facilities in DSU.

Another aspect of chief importance is the design and incorporation of applications whose main purpose is to accelerate the execution of individual SU applications without explicitly parallelizing their algorithms. An example of this kind of application is the **Fork-and-merge** task. A Fork-and-merge task should take an SU application name as an argument and initiate many instances of that application. As traces arrive at the fork-and-merge application they are forwarded to the instances of the SU application in a round-robin fashion. The instances of the SU application sends their output to the next application in the sequence. This kind of task may accelerate the execution of the provided SU application.

Regarding the overall capabilities provided by the system, an important aspect would be to explore the advantages of implementing facilities for handling several sequences simultaneously.

Communication via pipe files has been replaced by communication via explicit messages in DSU. Since pipe files are usually implemented with memory buffers, an important issue that would enormously help to predict the performance of DSU would be to compare, in terms of the number of traces to process, the overhead due to the handling of the memory buffers used to implement the pipe files with the overhead introduced by communicating the data via messages.

REFERENCES

- [Almasi *et al.*, 1993] G. Almasi, D. Hale, J. Bell, and A. Gordon. Parallel distributed seismic migration. *Concurrency: Practice and Experience*, 5:105–131, 1993.
- [BabaOglu *et al.*, 1991] O. BabaOglu, L. Alvis, A. Amoroso, and R. Davoli. Paralex: An environment for parallel programming in distributed systems. *Technical report UB-LCS-91-01*, pages 422–431, 1991.
- [Bell, 1992] G. Bell. Ultracomputers: A teraflop before its time. *Communications of the ACM*, 35:27–47, 1992.
- [Black and Su, 1992] J. Black and C. Su. Performance of parallel downward continuation. *62th Ann. Mtg., SEG, Expanded Abstracts*, 1:326–329, 1992.
- [Carreiro and Gelernter, 1989] N. Carreiro and D. Gelernter. Linda in context. *Communications of the ACM*, 32:10–18, 1989.
- [Cohen and Stockwell Jr., 1991] J. K. Cohen and J. W. Stockwell Jr. Su tutorial. *CWP report*, 1991.
- [Coulouris *et al.*, 1994] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems, concepts and design*. Addison-Wesley, 1994.
- [Flower *et al.*, 1991] J. Flower, S. Bharadwaj, and A. Kolawa. The express way to distributed computing. *Supercomputing review*, pages 54–55, May 1991.
- [Geist *et al.*, 1991] A. Geist, V. Sunderam, and A. Beguelin. Graphical development tools for network-based concurrent supercomputing. *Supercomputing'91 Proceedings*, pages 422–431, 1991.

- [Geist *et al.*, 1994] A. Geist, B. Mancheck, J. Dongarra, and A. Beguelin. *The PVM Book*. MIT-Press, 1994.
- [Gropp *et al.*, 1994] B. Gropp, R. Lusk, N. Doss, and T. Skjellum. Mpich: an implementation of mpi. <http://www.mcs.anl.gov/mpi/>, 1994.
- [Gustafson, 1988] J. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, pages 32–33, 1988.
- [Hembold and McDowell, 1990] D. Hembold and C. McDowell. Modeling speedup (n) greater than n . *IEEE Trans. on parallel and distributed systems*, 1:250–256, 1990.
- [Lewis and El-Rewini, 1992] T. Lewis and H. El-Rewini. *Introduction to Parallel Computing*. Prentice-Hall, 1992.
- [Ortega and Voight, 1991] J. Ortega and J. Voight. Solution of PDE on vector and parallel computers. *SIAM review*, pages 149–150, 1991.
- [Ousterhout, 1994] J. Ousterhout. *TCL and TK toolkit*. Addison-Wesley, 1994.
- [Rochkind, 1985] M. Rochkind. *Advanced UNIX programming*. Prentice Hall, 1985.
- [Rumbaugh *et al.*, 1991] J. Rumbaugh, M. Blaha, Premerlani W., E. Frederick, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.
- [Snair *et al.*, 1996] M. Snair, S. Otto, S. Huss-Lederman, and J. Dongarra. *MPI: the complete reference*. MIT press, 1996.
- [Welch, 1995] B. Welch. *Practical programming in TCL and TK*. Prentice Hall, 1995.

Appendix A

ABOUT SU

This appendix has been entirely extracted from the SU user manual document developed by [Cohen and Stockwell Jr., 1991].

Seismic Unix (SU) is a self-contained software environment for seismic research and data processing used by exploration geophysicists, earthquake seismologists, environmental engineers, software developers and others. It is used by scientific staff in both small companies and major oil and gas companies, and by academics and government researchers.

The SU package is *free software*, distributed quarterly with the full source code, so that users can alter and extend its capabilities. The package permits the exchange of data according to the industry protocol (SEG-Y). It provides a standard environment for the testing of new processing algorithms. It is easy to use because it does not require learning a special language—its application uses only the standard facilities afforded by the UNIX operating system. Once UNIX shell-redirecting and pipes are mastered, there is no further artificial language to learn. The seismic commands and options can be used as readily as other UNIX commands. In particular, the user can write ordinary UNIX shell scripts to combine frequent command combinations into meta-commands (i.e., processing flows). These scripts can be thought of as “job files.”

The seismic processing programs in the package assume that the data are written in SEG-Y format with each trace preceded by an appropriate header. This allows the data information to be read by each program in the processing stream in a consistent manner. The package includes facilities for converting data in several other formats

to the SEG-Y format.

The SU user community accesses the software over the Internet using the commonly available “anonymous ftp” facility. In this way, users obtain the software with no constraints on its use. During installation, the user is given the option of sending an electronic mail request to add them to our user group. Members of the user group receive announcements when updates of the package become available. The SU community is worldwide, spanning six continents, 29 countries and over 400 known installations on a variety of hardware platforms ranging from mainframes to workstations and PC’s.

Parts of SU originated from software developed by students working with the Stanford Exploration Project at Stanford University. The present package was developed and is maintained at the Center for Wave Phenomena (CWP) at the Colorado School of Mines. CWP is an interdisciplinary (geophysics, mathematics) research and educational program in seismic exploration, supported by 25 companies in the oil and gas industry.

A.1 How to Get a Copy of SU

The SU package contains seismic processing programs along with libraries of scientific routines, graphics routines and routines supporting the SU coding conventions. The package is available by anonymous ftp at the site hilbert.mines.edu (138.67.12.63). The directory path is `pub/cwpcodes`. Take the files:

1. `README_BEFORE_UNTARRING`
2. `untar_me_first.xx.tar.Z`
3. `cwp.su.all.xx.tar.Z`

Here the **xx** denotes the number of the current release. An incremental update is also available for updating the previous release **yy** to the current release **xx**. Take the files:

1. README_BEFORE_UNTARRING
2. README_UPDATE
3. untar_me_first.xx.tar.Z
4. update.yy.to.xx.tar.Z
5. update.list

A.2 Requirements for installing the package

The only requirements for installing the package are:

1. A machine running the UNIX operating system.
2. Ten megabytes of disk space for the source and compiled binary.

The package has been successfully installed on:

- IBM RS6000
- Silicon Graphics
- SUN SPARC STATIONS
- HP 9000 series machines
- HP Apollo
- NeXT

- Convex
- DEC

A.3 Some core programs

Reading the self-documentation and trying out the following SU programs will give you a good start in learning SU.

A.3.1 Examining the trace headers

surange — print minimum and maximum values of trace header fields

sugethw — print values of selected header fields

suascii — print header and data values

suxedit — interactively examine headers and traces

A.3.2 Some common processing programs

suacor — compute autocorrelations

sufilter — multipurpose zero phase filter (includes bandpass)

sugain — gain (with lots of options)

sumute — zero samples before a time that depends on offset

sunmo — normal-moveout correction

supef — prediction error filtering

susort — sort traces by values of trace header fields

sustack — stack (sum) traces

suvelan — velocity analysis

suwind — window (i.e., get a subset of) traces

A.3.3 Some common plotting programs

suximage — gray scale X Windows plotting

suxwigg — bit mapped wiggle trace X Windows plotting

supsimage — gray scale PostScript plotting

supswigg — bit mapped wiggle trace PostScript plotting

A.4 A brief tour of the source directories

The SU software is a layered product. The layers correspond to the following directories:

cwp Library of scientific routines (e.g. fft routines) written in “vanilla” C. Utility mains and shells.

par Library supporting the CWP programming style (i.e., self-doc, error reporting, parameter passing). Mains that use (only) these facilities. Shells for maintaining the online documentation database.

su Seismic processing codes that use the SEG-Y trace structure. Subroutines that manage this structure. Codes that buffer the generic graphics routines listed below. Shells that provide backward compatibility with earlier releases.

Graphics libraries

ARTHUR LAKES LIBRARY
COLORADO SCHOOL OF MINES
GOLDEN, CO 80401

1. **psplot**—PostScript graphics:
 - (a) **pscontour**: contour plots
 - (b) **pscube**: 3D data cube
 - (c) **psgraph**: curve plotting
 - (d) **psimage**: raster plotting
 - (e) **psmovie**: supports frames
 - (f) **pswigg**: bit mapped wiggle traces (fast)
 - (g) **pswigg**: polygon wiggle traces (slow)
 - (h) PostScript support programs
2. **xplot**—xlib based X Windows graphics
 - (a) **ximage**: raster plotting
 - (b) **xwigg**: bit mapped wiggle traces
 - (c) X Windows support programs
3. **Xtcwp**—toolkit based X Windows graphics
 - (a) **xgraph**: curve plotting
 - (b) **xmovie**: supports frames
 - (c) X Windows resource files

Appendix B

DSU SOFTWARE ORGANIZATION

This appendix contains general information about DSU. It contains a short introduction to its organization and directions to install it. It provides directions at a level appropriate for users with previous experience with PVM. We have also included directions to get you started on using DSU. Here we do not include directions about how to install PVM or TCL/TK. The following URLs contain comprehensive information about that:

PVM http://www.epm.ornl.gov/pvm/pvm_home.html

TCL/TK <http://www.sco.com/Technology/tcl/Tcl.html>

B.1 Requirements for installing the package

The only requirements for installing the package are:

1. A machine running the UNIX operating system.
2. Seismic Unix V28
3. PVM 3.8 or higher
4. TCL/TK (7.4/4.0)

The package has been successfully installed on:

- Pentium workstations running LINUX

- Silicon Graphics workstations (Indigo II and Indy)
- IBM SP2

B.2 Files in the DSU root directory

DSURC: Used to setup the environment variables needed to run DSU. Among this variables we have: **PVM_ROOT**, **TCL_LIBRARY**, **DSULOG**, etc.

README: Directions to install DSU.

Makefile.config: Modifications to `$CWPROOT/src/Makefile.conf` necessary to compile SU applications under the DSU environment. This file is included by DSU subdirectory Makefiles.

B.3 A brief tour of the source directories

The DSU software is distributed in several subdirectories grouped in a root directory called **dsu**. The directory **dsu** must be placed in the subdirectory **src** of the SU software, so that the root directory of DSU can be referenced as: `$CWPROOT/src/dsu`.

The root directory of DSU is organized as follows:

dsupar: Contains the files describing the parameters names and default values for each SU application. There is a file for each SU application. Additionally, a program tool that allows the extraction of the parameter names of any SU application is also included here;

- graphics:** Contains Makefiles to compile SU graphical applications intended to be used with DSU. It is composed of two subdirectories. **xplot**, which contains a Makefile to generate executables for the Xlib based X-windows applications; and, **psplot**, which contains a Makefile to generate the executables of postscript based graphical applications;
- lib:** Distributed Seismic Unix subroutine library;
- main:** Contains a Makefile to generate executables to be used with DSU, for the SU core applications;
- main_dst:** Contains PVM based applications that can be used as any other SU application. It contains the appropriate Makefile;
- gui:** Contains the graphical user interface (GUI). PVM3 and TCL/TK must be available before compiling the GUI. The Makefile in this directory must be customized before installing. The executable is called **dsu** and is placed in the directory **\$CWPROOT/bin**.

B.4 Brief summary of the steps to install DSU

1. Install **SU V28**; **PVM 3.8** or higher; **TCL/TK (7.4/4.0)**
2. Obtain the tar file with dsu (**dsusrc.tar.gz**). The URL is <http://landau.mines.edu/pvm>.
Or if you prefer <ftp://ftp.cwp.mines.edu/cwpcodes/pvm>.
3. **cd \$CWPROOT/src**
4. Create the root directory for DSU by untarring the distribution file in this way:
(this creates the directory: **\$CWPROOT/src/dsu**)
gunzip -c dsusrc.tar.gz | tar xvf -

5. `cd dsu` (or `cd $CWPROOT/src/dsu`)
6. Make sure the following environment variables are set:

`CWPROOT` (CWP SU main directory)
`PVM_ROOT` (PVM main directory)
`PVM_ARCH` (PVM main directory)
7. Customize the file `Makefile.config`. Since SU was previously installed, not too much need to be done here. Check the `CTARGET` for your machine.
8. Make sure the pointers to the TCL and TK libraries and include files are correct in the file: `$CWPROOT/src/dsu/gui/Makefile`
9. Type: `make INSTALL`

B.5 How to run dsu

To use DSU, follow these steps:

1. Customize the file `$CWPROOT/src/dsu/DSURC` according to your site and ADD the following line to your `.cshrc` file:

```
source \ $CWPROOT/src/dsu/DSURC
```

(if you are not using `csh` or `tcsh` the customization of the corresponding files is very similar). If `TCL/TK` are not installed in the default places, the environment variables `TCL_LIBRARY` and `TK_LIBRARY` must be pointing to the places where they were actually installed;

2. Put the hostname of the machines you intend to use with PVM in the file **\$HOME/.dsu_hosts**. DSU will setup a PVM virtual machine with the machine whose names are included in this file;
3. Make sure that **\$CWPROOT/bin** is in your path;
4. Invoke DSU by typing: **dsu**

Appendix C

TCL/TK OVERVIEW

The information contained in this appendix was entirely extracted from the TCL/TK WWW home page. To obtain more detailed information about TCL/TK, please see this World Wide Web site:

<http://www.sco.com/Technology/tcl/Tcl.html>

This is a relatively short introduction to TCL/TK, which is designed to get you started in using TCL/TK. It is not a manual, or a comprehensive book on TK, or a book on programming, but attempts to introduce TCL/TK at a level which will be appropriate for most readers with some previous programming experience. Further information on TCL/TK can be found in the books by Ousterhout [Ousterhout, 1994] and [Welch, 1995]. Some readers may also find that examining the source files (such as the widget demo program) supplied with the distribution, will be helpful. In addition, there is a newsgroup for exchange of information about TCL.

TCL/TK is a programming system developed by John Ousterhout at the University of California, Berkeley,, which is easy to use, and which has very useful graphical interface facilities. TCL is the basic programming language, while TK is a ToolKit of widgets, which are graphical objects similar to those of other GUI toolkits, such as Xlib, Xview and Motif. Unlike many of the other toolkits, it is not necessary to use C or C++ in order to manipulate the widgets, and useful applications can be built very rapidly once some expertise of the TCL/TK system has been gained.

Some users will naturally wish to use the widgets with C or C++. The TCL/TK system can be configured to work co-operatively with other programming languages

such as C or C++, and facilities to support this are described in section 3 of Ousterhout's book.

The TCL language is normally interpreted, so TCL applications will normally not run as fast as equivalent C programs. For a large class of applications this is not a disadvantage, however, since the speed of processing of modern computer systems is more than adequate. Where speed of processing is essential, use can be made of a TCL compiler, or processing can be carried out in a compiled language, such as C or C++, and the user interface written in TCL.

There are versions of TCL for different hardware systems, and for different operating systems, so TCL is to a large extent portable. However, it has already been noted that where TCL programs are used to access operating systems feature, portability will be sacrificed to convenience on the user hardware.