University of Alberta

Library Release Form

Name of Author: Meng Ding

Title of Thesis: Trellis-SDP: File-Level Data Parallelism

Degree: Master of Science

Year this Degree Granted: 2005

Meng Ding
2A 9010 112st
Edmonton, Alberta
Canada, T6G 2C5

Date: ____________________

University of Alberta

TRELLIS-SDP: FILE-LEVEL DATA PARALLELISM

by

Meng Ding

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2005

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Trellis-SDP: File-Level Data Parallelism** submitted by Meng Ding in partial fulfillment of the requirements for the degree of **Master of Science**.

Paul Lu
Supervisor

Mario A. Nascimento

Andrzej Czarnecki

Date: ____________________

To mom, dad and my sister.

Abstract

Some datasets and computing environments are large and inherently distributed. For example, image data may be gathered and stored at different locations for later processing. Although *data parallelism* is a well-known computational model, there are few programming systems that are both easy to program (for simple applications) and able to work across administrative domains.

We introduce Trellis-SDP, a simple data-parallel programming system that facilitates the rapid development of data-intensive applications. Trellis-SDP is layered on top of the Trellis infrastructure, a software system for creating *overlay metacomputers*: user-level aggregations of computer systems. Trellis-SDP is based on file-level data parallelism and provides a Master-Worker programming framework in which the worker components can run self-contained, new or *existing* binary applications. We describe the design and implementation of Trellis-SDP interfaces, including data-parallel interfaces and collective-communication interfaces. We evaluate our programming system with three simple data-parallel applications and one non-trivial seismic data processing application.

Acknowledgements

I would like to thank my supervisor Dr. Paul Lu for his guidance, his patience and his understanding during this challenging time. Without his help and support, my thesis would not have been possible. I would also like to thank Yang Wang, Mike Closson, Nicholas Lamb, Danny Ngo, and all the other members of the Trellis group – Nolan Bard, Mark Goldenberg, Morgan Kan, Mark Lee, Chris Pinchack, Ron Senda, Jeff Siegel and Dr. Edmund Sumbar – for their excellent work in Trellis, and for all the help they have given me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data parallelism is a well-known programming model [37]. However, with the development of distributed computing platforms (such as clusters, metacomputers over a wide-area network (WAN) [12], and grids [14]), it can be difficult to write and deploy even a simple data-parallel application. This is unfortunate because many problems are naturally data parallel. For example, many problems in information retrieval, sorting, and searching have inherently data-parallel phases and regular communication patterns, which are characteristics of simple data-parallel applications. Parallelizing an existing application in these areas may require porting (e.g., using a message-passing system) and access to source code. However, message passing can be complicated, and the applications may be in binary-only form. With the prevalence of inherently data-parallel applications and distributed computing platforms, there is a need for a simple data-parallel programming system for simple data-parallel problems. Furthermore, as disks become larger and cheaper, scientific applications are using this advantage to generate and consume more datasets. When the total amount of data becomes overwhelming and grows beyond the transmission capacity of the underlying network infrastructure, certain data-handling mechanisms must be taken into consideration to accommodate this change [23].

We introduce Trellis-SDP, a simple data-parallel programming system that facilitates the rapid development of data-intensive applications. Trellis-SDP is layered on top of the Trellis infrastructure, a software system for creating *overlay metacomputers*: user-level aggregations of computer systems. Trellis-SDP is based on file-level data parallelism (i.e., data parallelism within files instead of in-memory data structures (Section 3.1)) and provides a Master-Worker programming framework in which the worker components can run self-contained, unmodified, new or *existing* binary applications.

The main design goal of Trellis-SDP is to make the programming of data-parallel and data-intensive applications over a metacomputer as simple as possible. Based on our experience, a large number of meaningful scientific applications are well-suited to be parallelized. However, the effort required to port these computations over the metacomputers may be daunting. For example, without a proper data-parallel programming system, an application as simple as a distributed `grep` (a regular expression pattern matcher on distributed data) can be difficult to write. By using Trellis-SDP, we are able to implement this distributed `grep` in less than 40 lines of code.

Figure 1.1 shows the complete code for the master component of the `trellis grep` program. At line 12 and 16 , a `grep` operation and a metadata file, a physically-distributed but logically-contiguous file (discussed in detail in Section 4.2) are provided. At line 20, a call to the `trellis_scan()` function is made to invoke the `grep` operation on remote hosts. At line 25, the `trellis_scan_readall()` function is called to read in the results. We will explain this code in more detail in Section 4.3.1.

In this thesis, we describe the design and implementation of Trellis-SDP, including the background concepts and the application programming interface (API), e.g., the data-parallel interface and the collective-communication interface. We evaluate our programming system with three relatively simple data-parallel applications (i.e., `trellis grep`, Content-Based Image Retrieval (CBIR) and Parallel Sorting by Regular Sampling (PSRS)) and one non-trivial seismic data processing application with 6 GB of input data.

```
 1 #include <string>
 2 #include <stdio.h>
 3 #include <trellis.h>
 4 #include <trellis_sdp.h>
 5
 6 int main(int argc, char * argv[]){
 7
 8      Trellis_Request request;
 9      Trellis_Status status;
10      void * buffer;
11      char * grep_arg = argv[1];
12      char * metafile = argv[2];
13      string op;
14      int items_read = 0;
15
16      op = "grep " + string(grep_arg);
17
18      trellis_init(argc, argv);
19
20      if(trellis_scan(metafile, op.c_str(), &request)<0){
21            fprintf(stderr, "Scan Failed\n");
22            exit(-1);
23      }else{
24
25            items_read = trellis_scan_readall(&buffer, Trellis_CHAR, request);
26            trellis_scan_wait(request, status);
27      }
28
29      trellis_finalize();
30      if(items_read > 0)
31              printf("%s\n", (char *)buffer);
32
33      return 0;
34 }
```

Figure 1.1: The complete code for the master component of the `trellis grep` program. The worker components are Unix `grep` executables. Section 4.3.1 provides a more detailed explanation of this code.

1.1 The Trellis Project

Trellis-SDP is designed to be a part of the whole Trellis metacomputing system. A Trellis metacomputer is a virtual, batch-processing and capacity-oriented computer [7]. The current Trellis metacomputer contains a batch-scheduler system based on placeholder scheduling for CPU allocation [29], a security infrastructure based on the Secure Shell (`ssh`) software system for authentication and authorization (i.e., Trellis Security Infrastructure [26]), and a distributed file system for global file sharing (i.e., TrellisNFS [7]). Trellis-SDP is our effort to integrate a simple programming system into Trellis.

1.2 Motivating Application Domain

Applications in many research areas involve processing of a large amount of data. A typical domain of application is information retrieval. As a concrete example, let us suppose that a company is providing a service for content-based music retrieval (CBMR), which takes a clip of singing or humming from a client and then searches through the music database to find the top 10 most-similar songs. If the database is too large to fit on one system and/or is already distributed, it would be impractical for the server to read in all the data and perform pitch/rhythm extraction [22] and comparison algorithms on a single site. Instead, one can choose to ship the music feature-extraction and feature-comparison functions to the sites where data resides, and perform the operations there. This *function shipping* and remote execution mechanism (Section 2.1.2) not only makes full use of the computational power on each site, but also greatly reduces the traffic over the WAN.

Another similar example is the content-based image retrieval (CBIR) application, which takes a sample image and returns the top `N` matching images from the image database. We discuss the CBIR application further in Chapter 5.

It is for the purpose of *function shipping* and remote execution that we initiated the research on Trellis-SDP, which is designed to support the easy and efficient programming of applications such as CBMR and CBIR to handle large collections of distributed datasets.

1.3 Contributions

There are a number of existing parallel programming frameworks and each of them targets different applications and platforms. The contributions of our programming framework are the following:

1. Trellis-SDP provides a *simple* Master-Worker programming framework (Section 3.3) that facilitates the rapid development of data-intensive and naturally data-parallel applications on a wide-area network.

2. Trellis-SDP introduces the metadata file that represents the naturally-distributed data. This facilitates the writing of a non-trivial data-parallel application with data-parallel and collective-communication phases.

3. For many data-parallel codes, Trellis-SDP allows the loosely-coupled workers to run *existing*, *sequential* and *unmodified* binaries; the master and worker binaries can be separate. In contrast, many parallel programming systems require the application to be recompiled into a single, tightly-coupled binary (e.g., typical OpenMP and MPI (Message Passing Interface) applications).

1.4 Concluding Remarks

In this chapter, we discussed our motivation for building Trellis-SDP and gave an example of how Trellis-SDP can be used. We also presented the complete sample code of a distributed `grep` application (Figure 1.1) to show that it can be simple to implement a non-trivial data-parallel application using Trellis-SDP. In the next chapter, we discuss the background knowledge on which our programming system is based and review some of the related work in this field.

Chapter 2

Background and Related Work

In the previous chapter, we introduced the Trellis project and discussed the motivation for developing the Trellis simple data-parallel programming framework, Trellis-SDP. In this chapter, we present some related background concepts that influenced the design and implementation of our work. We also outline several previous projects that are relevant to our field of interest.

2.1 Background Concepts

2.1.1 Data Parallelism vs. Task Parallelism

To achieve good performance on distributed-memory multicomputers, two parallel programming paradigms are most-commonly used: *task parallelism* and *data parallelism*. In the task-parallel programming paradigm, a program consists of a set of dissimilar (or similar) parallel functions that interact with each other via explicit communications and synchronizations. In the data-parallel programming paradigm, a program consists of a series of operations that are applied identically to all elements of a large data set, which can be decomposed and distributed among multiple machines. Figure 2.1 shows examples of a task-parallel workflow and a data-parallel workflow.

The major advantages of task parallelism are its generality and flexibility. Task parallelism emphasizes the communication between, and coordination of different tasks, making it more applicable for exploring applications that use irregular data structures. The disadvantage is that extra effort may be required for the programmer to explicitly create parallel tasks and manage all the communications and synchronizations. Changing the communication pattern of a program may entail significant modifications to the program source code.

The major advantages of data parallelism are its simplicity and scalability. Since operations are applied identically to data items in parallel, the amount of parallelism is primarily determined by the data size of the problem. Higher amounts of parallelism may be exploited by simply expanding the size of the problems. In practice, many scientific applications are naturally data-parallel at different levels of the storage hierarchy, from instruction-level data parallelism to file-level data parallelism. We will discuss file-level data parallelism in more detail in Section 3.1. The major disadvantage

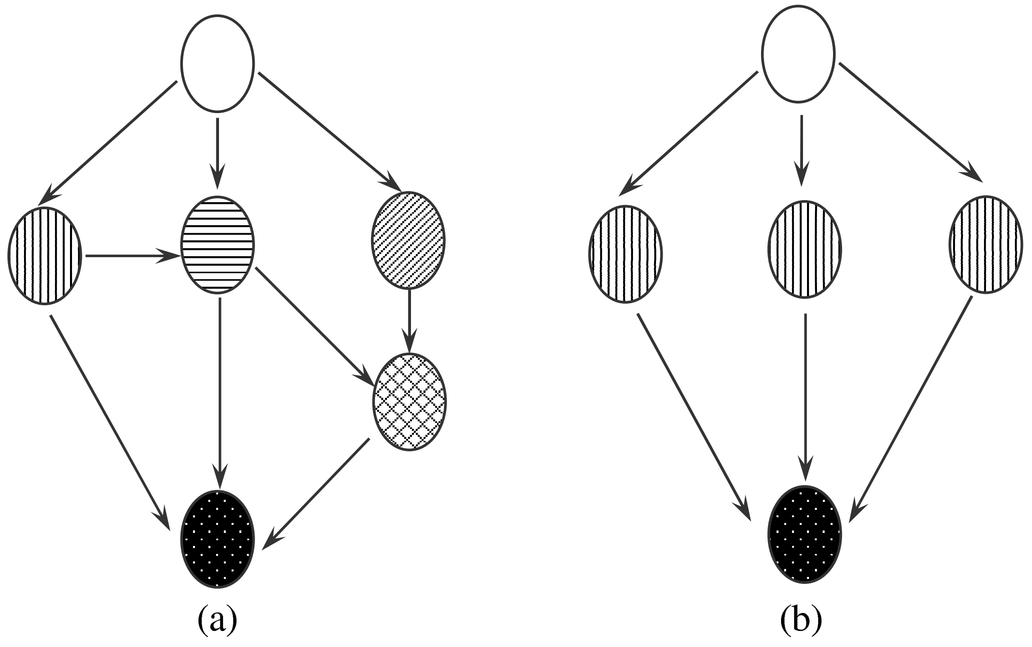


Figure 2.1: Task Parallelism vs. Data Parallelism. (a) Task parallelism workflow; (b) Data parallelism workflow.

of data parallelism is that it is not as general as task parallelism. For applications with different operations on different data, it is probably easier to utilize task-parallel languages.

As task parallelism and data parallelism each have strengths and weaknesses, it can be interesting to integrate these two when solving a complicated application. For example, in large-scale simulations, there may be multiple models simulated simultaneously via task parallelism. Within each simulated model, the computation could involve significant data parallelism.

The major differences between task parallelism and data parallelism are summarized in Table 2.1. Note that some of the comparisons are generalizations. For example, data-parallel applications, in general, operate on regular data structures; however, there are also data-parallel applications with irregular data structures, such as sparse-matrix multiplications. Similarly, there could be task-parallel applications with very large data sizes.

In this thesis, we explore primarily single program, multiple data (SPMD) programs with very large data-parallel phases.

2.1.2 Data Locality and Function Shipping

One critical issue related to data parallelism and task parallelism is the location and size of the data to be processed. It is quite common in a metacomputing environment that the executable and the corresponding data are not located at the same computing site. In this situation, one can choose to move the data to where the executable is located, or vice versa, in order to maximize performance. In a case where the data size is large and the processing is relatively simple, it is wise to move

	Task Parallelism	**Data Parallelism**
Definition	Each processor performs a different task	Each processor works on a different part of the same data
Data Structure	Irregular or regular	Regular or irregular
Common Data Size	Relatively small	Large
Advantages	Flexible and general	Simple and scalable
Disadvantages	Extra effort to manage communications and synchronizations	Less general; limited applicability
Typical Examples	Traveling Salesman Problem; Game Tree Search	Database Search; Matrix Multiplication

Table 2.1: A comparison between Task Parallelism and Data Parallelism.

the executable to where the data is located. In this way, not only can the computing power on individual hosts be fully utilized, the traffic over the wide-area network is also significantly reduced. This technique is called *function shipping* and remote execution (Figure 2.2), and is actually being implemented at different levels of the storage hierarchies. A typical example of *function shipping* is Active Disks [30], which implements this idea at the disk level. Active Disks is a storage system that consists of significant processing power and on-disk memory capacity (Figure 2.3). Application-level processing can be performed on the Active Disks, which can potentially reduce the traffic over the system bus, especially under I/O-bound workloads. Common applications are database operations (such as *select*, *join* and *aggregation*) or any filtering-type operations. There are several existing programming models proposed for Active Disks [1].

2.1.3 Master-Worker Programming Model in a Metacomputing Environment

Master-Worker computing is a widely-used form of a parallel application programming model. It is conceptually simple, and involves dividing a problem into a number of smaller independent worker units, which can be distributed to remote worker processes for computation in parallel. In this thesis, we use the Master-Worker programming model to implement data parallelism with function shipping and remote execution. Before the computation is started, we assume that the data is already distributed across the computing sites, and that the same executable codes are shipped to these sites. The master then triggers the executable in remote computing hosts, where I/O-intensive operations are performed locally, and only a small amount of data is transferred back to the master for subsequent processing.

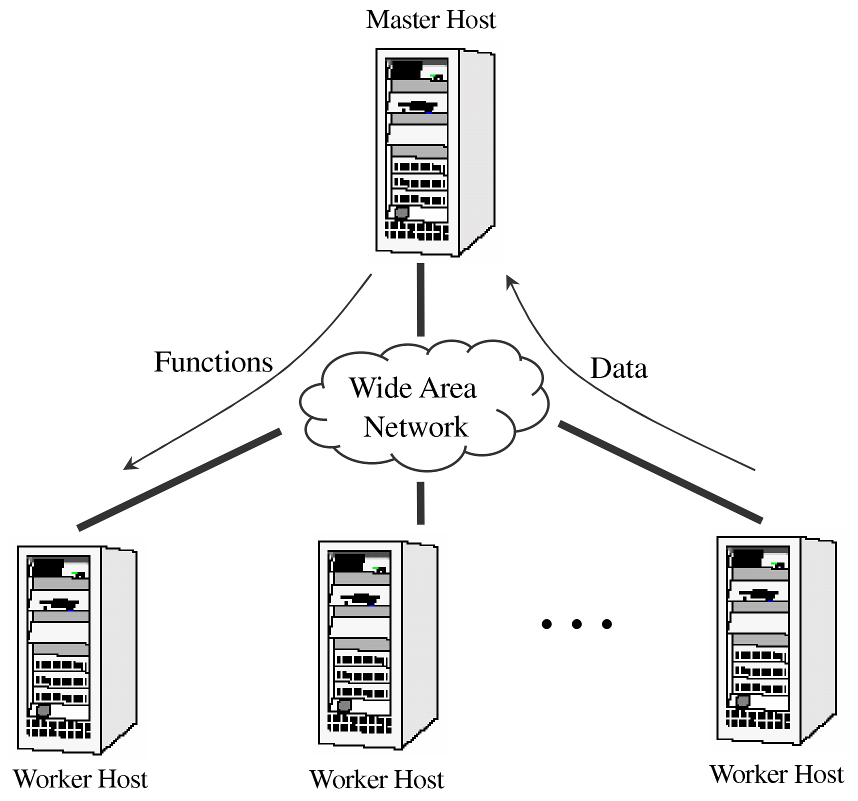


Figure 2.2: Function Shipping and Remote Execution on the WAN: executables are moved to the host where the data resides.

2.2 Related Work

The need to reduce the complexity of data-parallel programming has led to a large amount of work in the area of application-specific toolkits. These include application-specific or domain-specific languages and libraries, programming frameworks, and problem-solving environments [23]. Most of these toolkits have been adapted from traditional parallel and distributed computing systems; only a few are designed for grid computing or metacomputing [11]. Existing programming tools include message-passing libraries, object-oriented tools, and middleware systems. We highlight several major advantages and disadvantages of these work in Table 2.2.

2.2.1 Message Passing Models

MPICH-G2 [27] is a “grid-enabled implementation of the Message Passing Interface (MPI) that allows the programmer to run MPI programs across administrative domains using almost the same commands that would be used on a cluster of workstations” [24]. More specifically, MPICH-G2 is a complete implementation of the MPI-1 standard that uses services provided by the Globus Toolkit [25] to extend the MPICH implementation of MPI for Grid execution. The significant advantage of MPICH-G2 is that the programmer can reuse existing MPI code without having to learn the specific details of each site. A pragmatic disadvantage is that MPICH-G2 requires the Globus

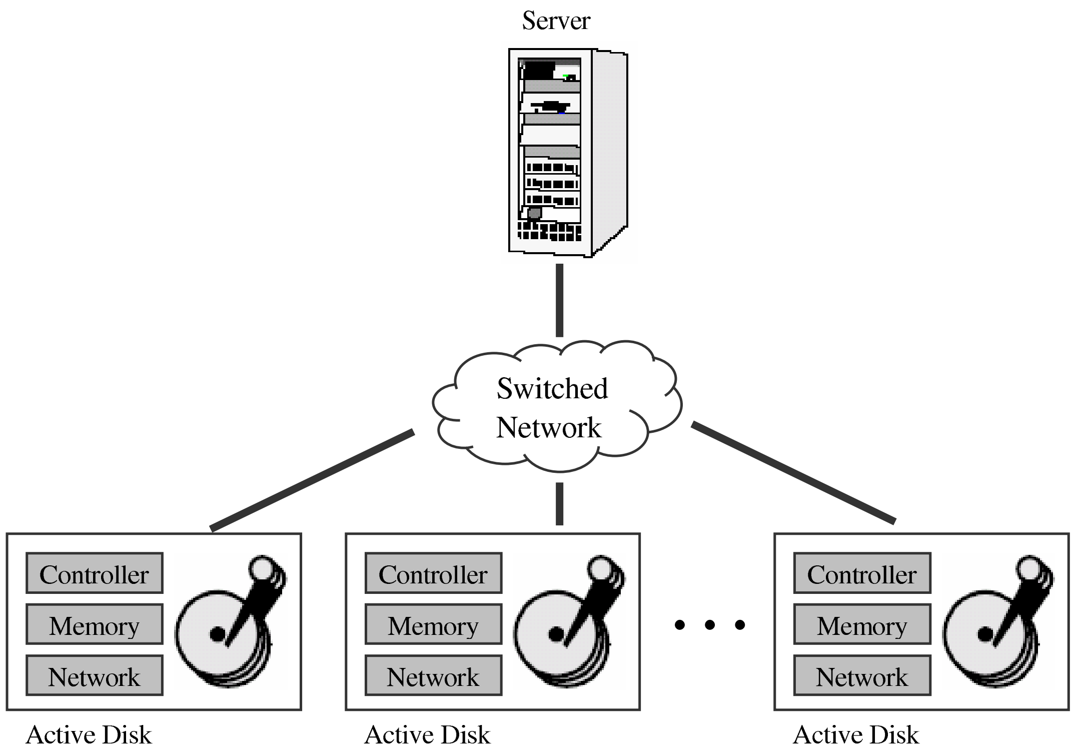


Figure 2.3: Active Disks architecture: applications can be downloaded to the disks, where significant processing power and on-disk memory is available.

toolkit to be installed in all the administrative domains, to address the issues of security, remote process creation, process monitoring, process control, redirection of standard input and output, remote file accesses and cross-domain communications (e.g., Grid Security Infrastructure (GSI), Grid Resource Allocation and Management Protocol (GRAM), Monitoring and Discovery Service (MDS) and Global Access to Secondary Storage (GASS) [25]).

The popularity of MPI has spawned a number of variants that address grid-related issues, such as dynamic process management and more efficient collective operations. The MagPIe library, for example, implements MPI's collective operations – such as broadcast, barrier, and reduction operations – with optimizations for wide-area systems as grids [20]. Existing parallel MPI applications can be run on grid platforms using MagPIe, by relinking with the MagPIe library. MagPIe contains a simple API through which the underlying grid computing platform provides information about the number of clusters in use, as well as which process is located in which cluster.

2.2.2 Framework Models

DataCutter [17] proposes a *filter-stream* programming model (originally designed for Active Disks [1, 30]) in a grid environment. In this programming model, an application is decomposed into a set of *filters* among which the communication is carried out via *streams*. As with Trellis-SDP, DataCutter pushes the computation to the data, instead of migrating the data to the computation [17]. DataCutter does not include the concept of a metadata file. Before running any applications on DataCutter, a `dird` program must be started to maintain a list of nodes that DataCutter applications may be distributed across, and any node wishing to be utilized by DataCutter must run an `appd` program to control the DataCutter applications running on the nodes. All filter *placements* must be specified in the program. Furthermore, the filter does not support unmodified binaries, meaning that programmers will probably have to rewrite their data-intensive components according to the filter specifications.

MapReduce [10] is a programming model developed at Google to process large data sets. It provides a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations. Built upon the Google file system, MapReduce carries out a number of practical design decisions and fine-tunings to achieve maximum performance.

MapReduce is relevant to Trellis-SDP in that both systems are designed to make it simple for application programmers to implement simple data-parallel applications. The abstraction of MapReduce is based on the *map* and *reduce* primitives present in many functional languages. The *map* function processes a key/value pair to generate a set of intermediate key/value pairs, and the *reduce* function merges all intermediate values associated with the same intermediate key. The programmer needs only to implement the *map* and *reduce* functions, and the data distribution and resource allocation are taken care of by the programming framework. Trellis-SDP, however, is more similar to imperative programming, where the number of processors is known before the computation, and the data is already distributed.

MapReduce has been demonstrated to be applicable to a wide range of real problems specific to Google, such as machine learning, clustering, web crawling and graph computation.

MW [15] is a software framework that allows users to parallelize scientific applications on a computational grid, using the Master-Worker programming model. This framework is designed to facilitate Master-Worker applications requiring a reliable delivery of large amounts of computational capacity. MW provides two sets of programming interfaces: an *Infrastructure Programming Interface* that ports the MW framework to a grid software toolkit such as Condor [8] or Globus, and an *Application Programming Interface* that enables the Master-Worker paradigm. In both cases, the user needs to re-implement a number of virtual functions to address low-level details – such as resource request and detection, remote execution and communication. In addition, the programmer needs to re-implement the workers using MW-specific classes – MWTask and MWWorker.

AppLeS Master Worker Application Template (AMWAT) [34] is a middleware approach to Master-Worker application development that aims to achieve three design goals: performance, portability and reasonable effort. This programming framework can be separated into three distinct groups: the *base* group that provides interfaces to perform the initialization of the basic computational activities of the application; the *transfer* group that provides interfaces to perform the data transfers; and finally, the *control* group that provides interfaces to perform the scheduling functions.

AMWAT uses the AppLeS Portable Services components for communications. These services contain common communication approaches, such as MPI, PVM (Parallel Virtual Machine), Unix Sockets and System V IPC (interprocess communication), plus some less-common approaches, such as Globus Input/Output. As with MW, application programmers need to fill in the application templates with source codes, making it difficult to reuse existing application binaries.

Unlike DataCutter and Trellis-SDP, both MW and AMWAT assume that data could/should be moved during the computation; for example, data is not distributed before the computation is started. Also, AMWAT allows dynamic selection of master and worker processes to maximize performance. This is very practical for computationally-intensive applications, but not for data-intensive applications.

2.2.3 RPC Models

Grid Remote Procedure Call (RPC) [16] is an RPC model and API for grids. It offers a relatively simple programming paradigm for programming on the grid. Besides providing standard RPC semantics with asynchronous, coarse-grained, task-parallel execution, it provides a high-level abstraction whereby many details of interacting with a grid environment can be hidden. However, the Grid RPC programming model is not suited for applications with data-intensive or I/O-intensive phases. Also, since the result of the computation is transferred back to the client side, there would be a problem if the data size is large and the network bandwidth is low.

Java Remote Method Invocation (RMI) enables a programmer to create distributed Java-based applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts. The main advantages of RMI are that it is truly object-oriented, that it supports all the data types of a Java program, and that it is garbage collected. These features allow for a clear separation between the caller and callee, and the development and maintenance of distributed systems are thus made easier.

2.3 Concluding Remarks

In this chapter, we began by comparing two typical parallel programming paradigms: *task parallelism* and *data parallelism*. We discussed their advantages and disadvantages under various circumstances. Naturally, for both types of applications, under data-intensive workloads, the location and size of the data will have a large impact on the application performance; this is why we introduced the concept of *function shipping* and remote execution. We implemented this concept using the common Master-Worker programming model, which is simple and easy to manage.

We also reviewed some previous projects from the field of programming models on a wide-area network. Each model has different design goals and is suited to certain types of workloads. This is also true for our system; our target applications are mainly data parallel with I/O-intensive workloads.

Related Work	Advantages	Disadvantages
MPICH-G2 [27]	Allows reuse of existing MPI code; Works across administrative domains.	Requires the installation of the Globus toolkit in all administrative domains; Low-level and complicated; Does not support unmodified binaries.
DataCutter [17]	Simple stream-based Master-Worker programming model; Supports *function shipping* and remote execution.	Current version does not work across administrative domains; No global naming of files; Does not support unmodified binaries.
MapReducei [10]	Simple and powerful Master-Worker programming model; Based on functional programming, the programmer does not need to take care of data distribution and processor allocation; Provides fault tolerance when worker or master fails.	Does not work across administrative domains; Does not support unmodified binaries.
MW [15]	Master-Worker programming framework; Suitable for computational intensive applications; Works across administrative domains.	Does not support unmodified binaries; Need to install grid software toolkit such as Condor and Globus.
AMWAT [34]	Master-Worker programming framework; Suitable for computational intensive applications; Provides basic performance predictions; Allows dynamic selections of master and worker processes.	Current version does not work across administrative domains; Complicated porting of source codes to templates.
GridRPC [16]	RPC programming extended to grids; Suitable for coarse-grained task-parallel applications; Works across administrative domains.	Low level; Not suitable for data-intensive or I/O-intensive applications; Requires complicated porting.

Table 2.2: A comparison of related work.

Chapter 3

Trellis-SDP: File-Level Data Parallelism

In the previous chapter, we introduced the background knowledge and concepts related to our research. We also reviewed several well-known programming systems. In this chapter, we focus on the concepts, as well as some practical problems that have not been addressed by previous systems.

We start by extending the concept of data parallelism to file-level data parallelism. We then present several important design issues in the development of our programming system. These include security, global naming, resource specification and design philosophy. We also illustrate the Trellis-SDP execution environment and briefly discuss the programming interfaces we provide.

3.1 File-Level Data Parallelism

We extend the traditional concept of data parallelism (e.g., the traditional data-parallel languages such as Fortran90 and High Performance Fortran(HPF)) to file-level data parallelism. That is, our programming framework is targeted at data parallelism and collective communications within files in a metacomputing environment, instead of in-memory data structures, as illustrated in Figure 3.1.

Based on our experience in the programming in a metacomputing environment, we claim that there are several advantages to exploring data parallelism and collective communications at the file level:

1. **Using unmodified binaries**: Working at the file level makes it easy to use sequential, unmodified binaries, or make as few changes as possible to the existing sequential/shared-memory applications, when porting these applications to a wide-area network. As long as the sequential executable guarantees that it takes the input from a file or the standard input, and generates the output to a file or the standard output, it can be integrated into the whole computation easily and smoothly.

2. **Master-Worker and batch-pipelined execution model**: Working at the file level makes it easy to implement parallel applications using Master-Worker and batch-pipelined (multiple

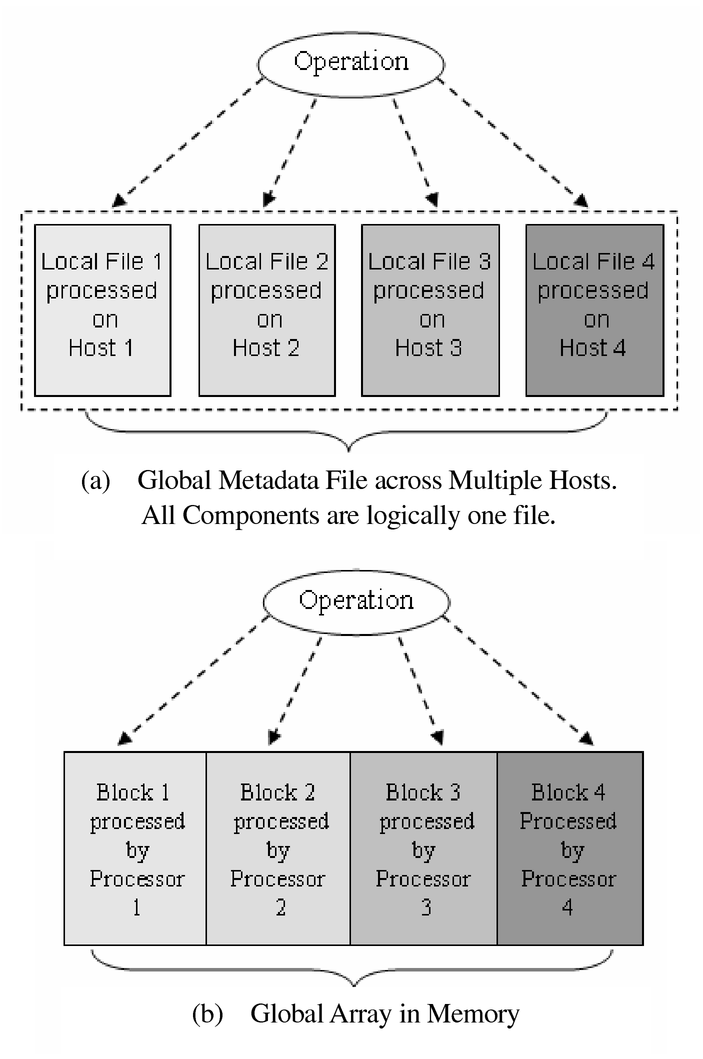

Operation
Local File 1 processed on Host 1
Local File 2 processed on Host 2
Local File 3 processed on Host 3
Local File 4 processed on Host 4
(a) Global Metadata File across Multiple Hosts. All Components are logically one file.
Operation
Block 1 processed by Processor 1
Block 2 processed by Processor 2
Block 3 processed by Processor 3
Block 4 Processed by Processor 4
(b) Global Array in Memory

Figure 3.1: Data Parallelism at different storage hierarchies: (a) Data parallelism at the file level : data-parallel operations on global files across multiple hosts; (b) Data parallelism at the memory level: data-parallel operations on a global array in main memory.

phases) execution model. That is, the output of one computational phase is the input of the next computational phase.

3. **File-level collective communication**: When performing collective communications at the file level, we impose less-strict requirements on the synchronization of processes than are imposed by memory-level collective communications. This is because intermediate computing results can be stored on disks until all collective-communication processes are ready to read and exchange them. Prior to that time, it is possible for the scheduler to schedule jobs on the idle hosts that have already produced the intermediate results.

The concepts of file-level data parallelism, function shipping and remote execution, and Master-Worker programming paradigm provide the foundation for our programming system.

3.2 Design Issues

The design and implementation of Trellis-SDP addresses several important issues in metacomputing programming [11]. We have successfully run Trellis-SDP applications on a wide-area network (Section 5.2 provides some preliminary results). However, for reasons of controllability and reliability, most of our benchmarks are performed on clusters of workstations. We will discuss this further in Chapter 5.

3.2.1 Security: The Trellis Security Infrastructure

Our programming system takes advantage of the underlying Trellis Security Infrastructure (TSI) [26], which is layered on top of the `ssh` software system [9, 3], for authentication and secure communications across different administrative domains. TSI allows single sign-on (a form of authentication that enables a user to authenticate once and gain access to multiple systems) capability by configuring and launching `ssh-agent` processes on all participating hosts. Unlike the Globus Security Infrastructure (GSI) [19], which places most of the configuration and authentication work onto the system administrator, TSI manages these tasks at the user level. The system administrator needs only to give each user an account, and install `ssh` – which has already been widely deployed on most platforms. Figure 3.2 illustrates the process of launching an `ssh` overlay by a user: the user runs a `launchAgents` tool (not shown in the figure), which invokes `ssh-agent-remote` and `ssh-add-remote` for all participating hosts, and loads the remote `ssh-agent` processes with a common key. The user then types in only one passphrase and the `ssh` overlay is established. This means that any of these participating hosts can access one another without a password or passphrase.

3.2.2 Global Naming: Secure Copy Locator

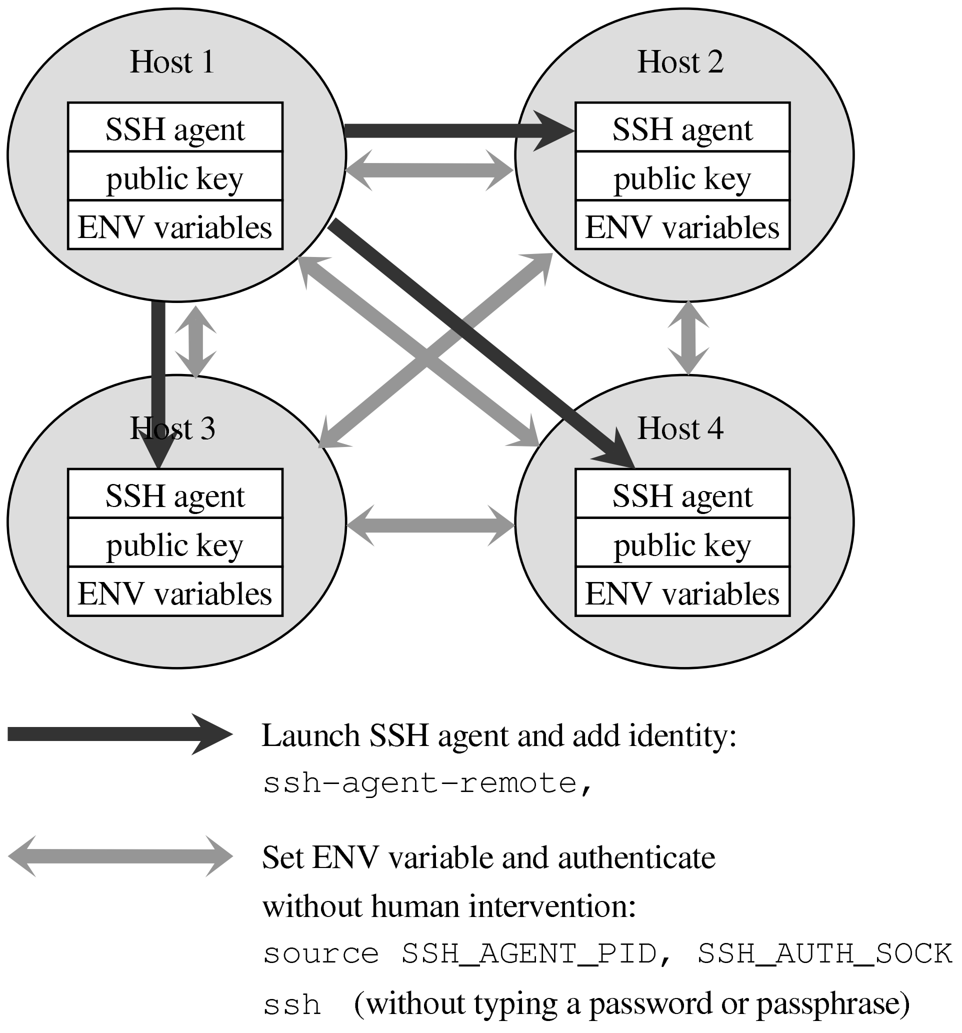
We use Secure Copy Locators (SCL) [35] as the filenames in the global namespace. By using SCL, the Trellis file system can access the remote data by first copying it onto a local disk and then accessing the local cached copy of the remote file. Our programming system extends this concept by function shipping the computation to the remote host (as discussed in Section 2.1.2). For example, a file named

```
scp:ading@cleardale.cs.ualberta.ca:~/worker.exe
```

can be uniquely identified as the file `worker.exe` in the home directory of account `ading` at host `cleardale.cs.ualberta.ca`.

3.2.3 Resource Specification: XML-based Metadata Schema

We represent the program resource (i.e., program data) by a metafile. A metafile is a file that is logically contiguous, but (perhaps) physically distributed across a network (Figure 3.1 (a)). An Ex-



Launch SSH agent and add identity:
`ssh-agent-remote,`

Set ENV variable and authenticate without human intervention:
`source SSH_AGENT_PID, SSH_AUTH_SOCK`
`ssh` (without typing a password or passphrase)

Figure 3.2: Launching the `ssh` overlay involves two steps: (1) Launching the `ssh` agent and adding identities at remote hosts. (2) Setting `ssh` environment variables and authenticating without human interventions.

tensible Markup Language (XML)-based metadata file is used by Trellis-SDP to describe a metafile (Figure 4.1), which includes the location (expressed as an SCL) and the size of the distributed blocks.

3.2.4 Using Existing, Sequential, Unmodifi ed Binaries

One key design philosophy of Trellis-SDP is to make it as simple as possible to write a data-intensive parallel application, which is why Trellis-SDP allows the use of existing, sequential, and unmodified binaries. As discussed in Section 1.2, this functionality may be useful if the programmer has existing binaries or binaries from a third party. By using or reusing existing binaries, the whole application development cycle could be dramatically simplified.

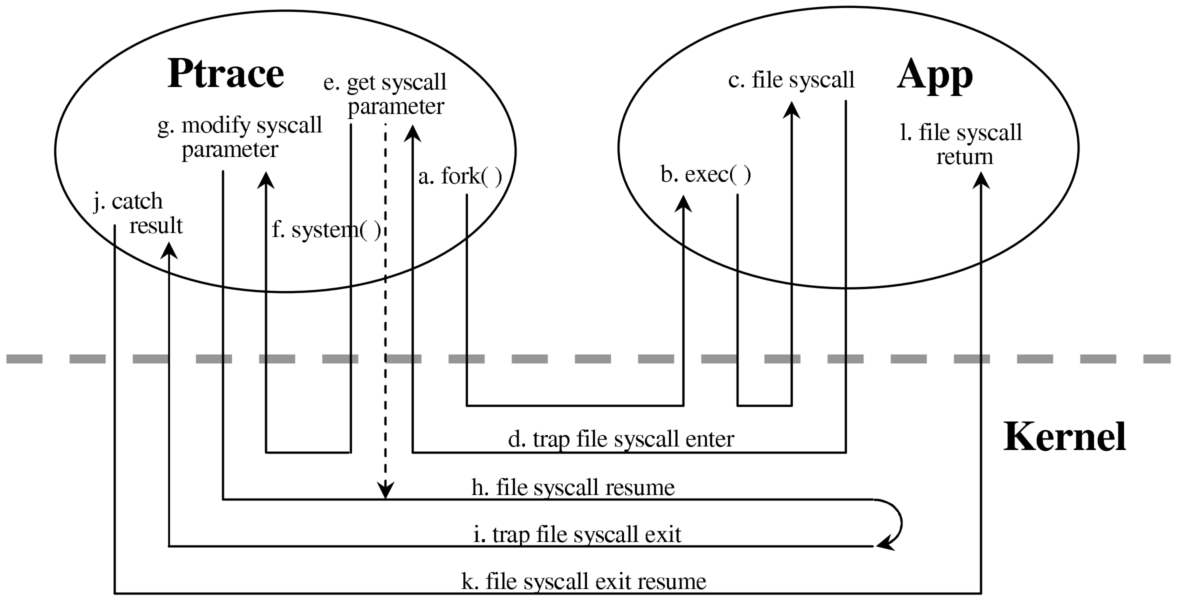


Figure 3.3: The interposition of an unmodified binary using the `Ptrace` program.

In order to use existing binaries, we must ensure that the binaries can access remote files. This is done by the system call tracing in Trellis-SDP. As illustrated in Figure 3.3, we write a program called `Ptrace` (which uses the `ptrace()` system call) to intercept the file system calls in the application, modify the system call parameters, execute some scripts, and resume the system call. For example, here is the sequence when the application opens a file:

1. `Ptrace` intercepts the entry to the system call `open()` (c and d in Figure 3.3).

2. `Ptrace` retrieves the first parameter of the `open()` system call (e in Figure 3.3). If it is a local file, go to 5 (dashed line in Figure 3.3). If it is a remote filename expressed as an SCL, go to 3.

3. `Ptrace` executes a script which uses the Trellis file system library to cache the remote file to local disk (f in Figure 3.3).

4. `Ptrace` modifies the first parameter of the `open()` system call to be the filename of the local cached file (g in Figure 3.3).

5. `Ptrace` resumes the stopped system call (h in Figure 3.3).

Following these steps, the application will access the local cached file. If the file is modified, the `Ptrace` program will intercept the `close()` system call and write the local file back to the remote host.

We use the `Ptrace` program in the implementation of Parallel Sorting by Regular Sampling application described in Section 5.4, where remote file access is required. Appendix C lists the source code for this application.

Besides system call tracing, there are other ways for an application to access a remote file. For example, by installing an NFS-to-Trellis gateway, NFS (Network File System) clients can mount a volume exported by the gateway, so each access to a remote file can be translated by the gateway into a remote data access via the Secure Copy ([7]).

3.3 Trellis-SDP Execution Environment

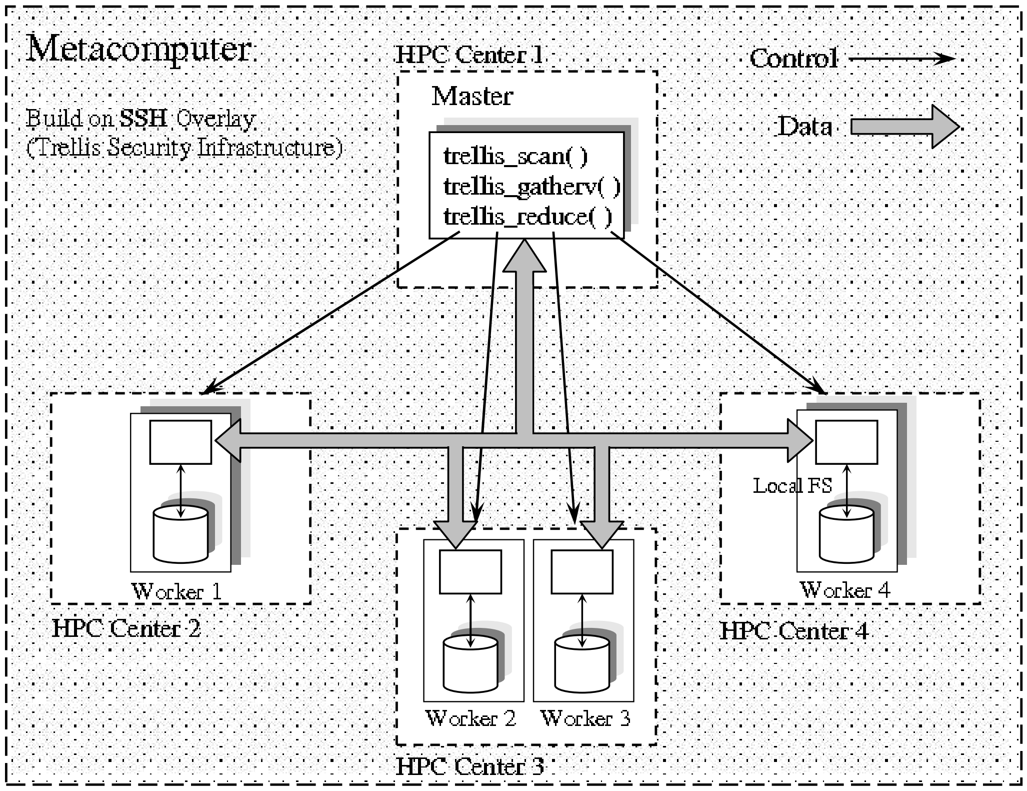


Figure 3.4: The execution environment of the Trellis-SDP programming system.

Having introduced related concepts and design issues, we now present the high-level overview of the Trellis-SDP execution environment. As mentioned earlier, the main design goal of Trellis-SDP is to facilitate the programming of data-intensive applications with coarse-grained communication patterns on a wide-area network. For more fine-grained and complicated message patterns, we also support certain types of group communication. Of course, the overall performance depends on the amount and type of communication in the application.

Trellis-SDP is well-suited to applications where it is either easy to decompose the application into master and worker components, or where the worker component already exists (e.g., as a sequential, binary executable). In both cases, it is the worker component that performs the data-intensive operations near the data, and it is the master component that synchronizes the computation and collects the results. For most data-parallel applications, the amount of data transferred between the master and worker should be *minimal*.

The programmer is responsible for identifying which part of the application can be parallelized and which part cannot, and if necessary, extracting the I/O-intensive cores in the application into one or multiple stand-alone phases, so that the additional communication introduced by the decomposition does not penalize the overall task completion time.

Figure 3.4 illustrates the execution environment of our programming system. Currently, we support the following functions:

1. **Data-Parallel Functions – `trellis_scan()`**: Inside a Trellis-SDP program, a worker process is invoked by a call to the `trellis_scan()` library function. This function takes a metadata file and an operation string as input parameters, and takes a handle to the scan object as the output parameter, which will store the result of the `trellis_scan()` operation. The worker processes on remote hosts perform the specified operations and either generate the results on their local disks or return the results back to the master process via streams. In the former case, intermediate files generated by different worker processes can also be described using a metadata file. This intermediate metadata file can be used in a different `trellis_scan()` phase, or it can be saved to disk. This is useful in a batch-pipelined workload [13], where the output of one worker process may be the input of a succeeding worker process. Figure 3.5 illustrates a batch-pipelined workload with three phases and four metadata files.

2. **Collective-Communication Functions – `trellis_reduce()` and `trellis_gather()`**: Note in Figure 3.4 that, if necessary, group communication can be performed among worker processes. Two group-communication functions are implemented: `trellis_gather()` which performs an all-to-all communication, and `trellis_reduce()` which performs a global reduction operation, such as global sum and global minimum/maximum. These operations are at the file level instead of the memory level because they take metadata files as the input parameter.

3. **Initialization and Finalization Functions**: Two basic run-time functions are provided: `trellis_`

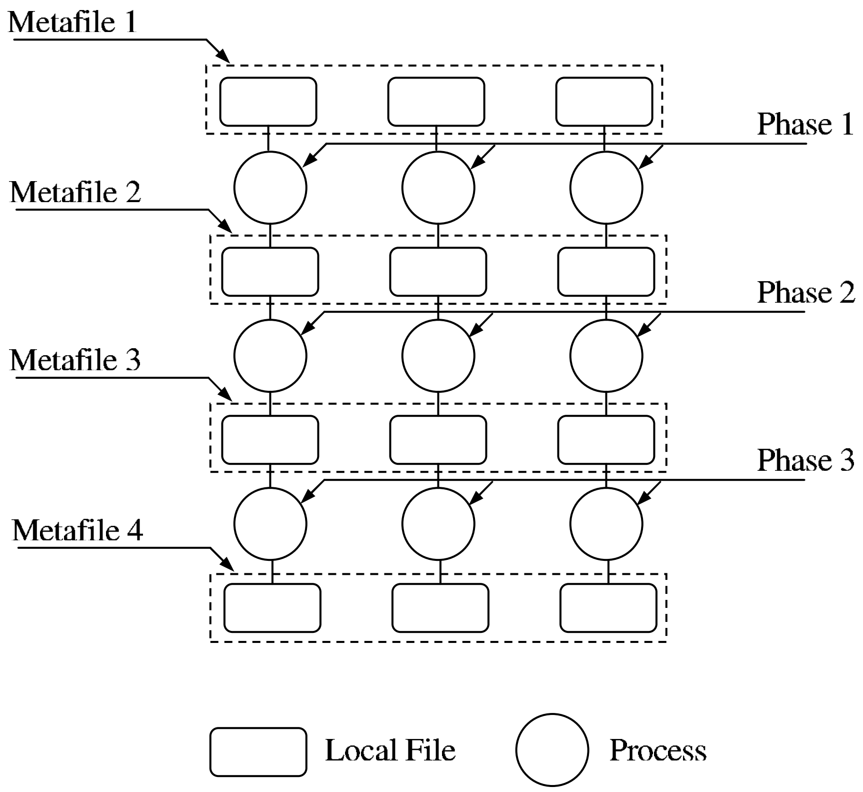


Figure 3.5: Metadata files in a batch-pipelined workload.

`init()` and `trellis_finalize()`. All Trellis functions should be called between `trellis_init()` and `trellis_finalize()`. If profiling capability is turned on, `trellis_finalize()` will generate basic performance data.

4. **Profiling Functions**: Since Trellis-SDP requires that an application be decomposed into multiple phases, it is relatively easy to collect some basic run-time performance data. For example, we can record the execution time of each phase of an application during multiple runs. This information can be useful for predicting the execution time of the application in future runs. We discuss our preliminary study of the application profiling in Appendix A.

We summarize all Trellis-SDP functions in Table 3.1. As a comparison, we also list the related MPI functions, together with Trellis-SDP functions. Note in the table that MPI does not provide data-parallel functions and the capability to use user-defined binary operations.

3.4 Concluding Remarks

In this chapter, we introduced the concept of file-level data parallelism and collective communication. We explained the motivation behind file-level data parallelism and demonstrated how our programming system can benefit from this idea in various ways, such as unmodified binaries, batched-pipelined execution model and simple application profiling.

Trellis-SDP	**MPI**	**Comments**
Data Types		
`Trellis_Datatype`	`MPI_Datatype`	
`Trellis_Op`	`MPI_Op`	
`Trellis_Request`	`MPI_Request`	Communication opaque object
`Trellis_Status`	`MPI_Status`	
Message Datatypes		(Equivalent C Types)
`Trellis_CHAR`	`MPI_CHAR`	`signed char`
`Trellis_INT`	`MPI_INT`	`signed int`
`Trellis_FLOAT`	`MPI_FLOAT`	`float`
`Trellis_DOUBLE`	`MPI_DOUBLE`	`double`
`Trellis_LONG`	`MPI_LONG`	`signed long int`
Predefined Reduction Operations		
`Trellis_ADD`	`MPI_SUM`	Global sum
`Trellis_MAX`	`MPI_MAX`	Finding Maximum
`Trellis_MIN`	`MPI_MIN`	Finding Minimum
Basic Functions		
`trellis_init()`	`MPI_Init()`	
`trellis_finalize()`	`MPI_Finalize()`	
Data-Parallel Functions		
`trellis_scan()`		Remote execution in parallel (non-blocking)
`trellis_scan_read()`		Read in `trellis_scan()` result
`trellis_scan_readidx()`		
`trellis_scan_readall()`		
`parseMetafile()`		Helper function
`single_file_scan()`		Helper function
`multi_file_scan()`		Helper function
Synchronization Functions		
`trellis_scan_wait()`	`MPI_Wait()`	
Collective-Communication Functions		
`trellis_gather()`	`MPI_Alltoall()`	
`trellis_reduce()`	`MPI_Reduce()`	Predefined reduction operations
`trellis_reduce1()`		User defined reduction operations (in binary form)
Profiling Functions		
`registerPhaseStart()`		Discussed in Appendix A
`registerPhaseEnd()`		

Table 3.1: An overview of the Trellis-SDP functions with the analogous MPI functions.

We dealt with several practical design issues during the implementation of our programming system – such as security, global naming and resource specification. These design decisions were made to ensure that the system is simple and reliable to use.

We illustrated the execution environment of the Trellis-SDP programming system and summarized all core Trellis-SDP functions, comparing these functions with related MPI functions. Details of the Trellis-SDP functions will be explained in the next chapter.

We also performed some preliminary studies on the profiling and performance prediction of applications written in Trellis-SDP. These are discussed briefly in Appendix A. We hope to integrate this work into our future scheduler.

Chapter 4

Implementation Details

In the previous chapter, we discussed the idea of file-level data parallelism and the execution environment of the Trellis-SDP system. In this chapter, we discuss the implementation details of our system, especially how the design issues presented in the previous chapter are put into practice.

4.1 Assumptions

For the current implementation of Trellis-SDP, we have several requirements for the whole computation setup. Before the computation is started, Trellis-SDP assumes that:

1. The data needed by the computation is already distributed across the metacomputer. This is a common case for wide-area data-intensive applications such as operations on a federated database [18] or information retrieval applications (e.g., CBIR application) on a distributed database. If the data is not distributed, tools are provided for scattering and gathering the data.

2. The metadata file, identifying the distributed data, already exists (Section 4.2). In our current implementation, the metadata file contains the location and size of the data on each participating host. If the computation contains multiple phases, then the input (or output) metadata file for all phases should all be made ready before the computation.

3. The executable code for the worker components is already distributed across the metacomputer. At this time, we require the worker components to be available at each participating host. The programmer must stage the executables to each host, if they are not there. In the future, we may support automatic staging of executables. For example, a potential strategy would be to append the address of the executable (in SCL format, as discussed in Section 3.2.2) to the corresponding file/data to be processed in the metadata file. Or, if the executable is a script, we could directly embed the script into the metadata file.

```
<?xml version="1.0"?>
<BlockList>
  <DataBlock>
    <Locator>scp:ading@jasper-00:/usr/scratch/data.1</Locator>
    <Size>108003000</Size>
  </DataBlock>
  <DataBlock>
    <Locator>scp:ading@jasper-01:/usr/scratch/data.2</Locator>
    <Size>108003000</Size>
  </DataBlock>
  <DataBlock>
    <Locator>scp:ading@jasper-02:/usr/scratch/data.3</Locator>
    <Size>108003000</Size>
  </DataBlock>
  <BlockSize>32</BlockSize>
</BlockList>
```

Figure 4.1: An example of a metadata file that describes a file that is logically contiguous but physically distributed across 3 nodes over a local-area network.

4.2 The Metadata File

As discussed earlier, a metafile is a file that is logically contiguous, but (perhaps) physically distributed across a network. As with other index-based file-allocation schemes, a Trellis-SDP metadata file specifies the name and location of the distributed blocks of a logical file. The master component can either access the file as if it was a single, logical file, or use the `trellis_scan()` function to perform a data-parallel operation on the physically-distributed blocks. Although it is assumed that the logical file is already distributed, a separate tool is provided to distribute (i.e., scatter) the data and create a corresponding metadata file. Another tool can take a metadata file and gather the distributed blocks into a single file on a local file system.

To make the representation of the program data human readable and extensible [11], the metafile is written in XML, as illustrated in Figure 4.1. In the metafile, each block is specified with a `DataBlock` node that contains a `Locator` (a string in SCL format) node and a `Size` (an integer specifying the size of each block, in bytes) node.

In practice, the programming system will create an in-memory metadata object corresponding to a metadata file. This is analogous to an in-memory version (i.e., metadata object) of a Unix i-node (i.e., metadata file). Upon object creation, all of the information in the metadata file is parsed and cached in the object (Section 4.3 provides further details on the metadata object). It is also possible to export a metadata object to disk, in XML format.

4.3 Main Trellis-SDP APIs

In this section, we explain in detail the implementation of the main Trellis-SDP APIs. In addition to the explanation, we give several examples.

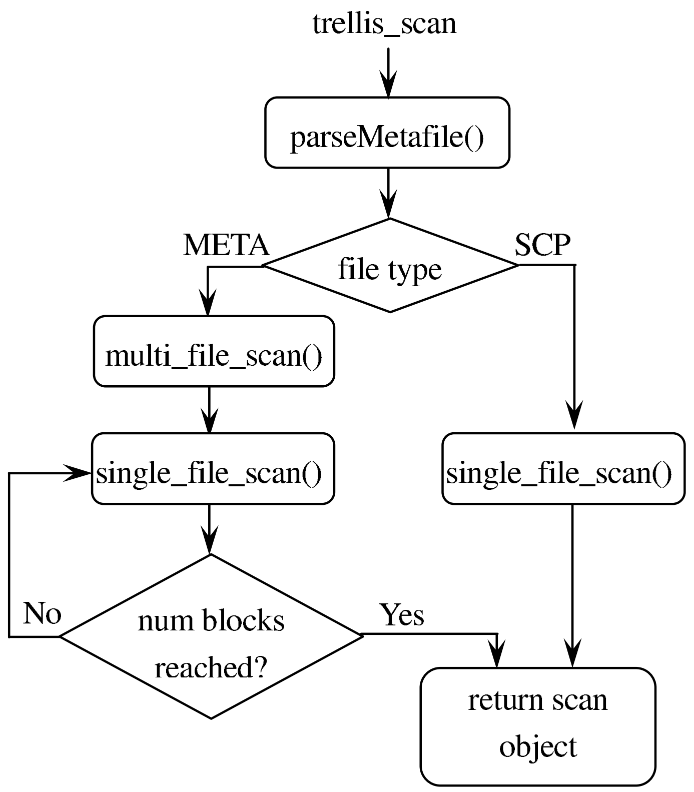


Figure 4.2: The control flow inside `trellis_scan()`.

4.3.1 Trellis Scan

`trellis_scan()` is the main data-parallel API we introduced and implemented in Trellis-SDP. The declaration of `trellis_scan()` is:

```
int trellis_scan(const char * metafile, string op, Trellis_Request *
                 request);
```

`trellis_scan()` is (typically) called in the master process, and takes two input parameters and one output parameter. For input, there is a metadata filename (or a regular SCL) and an operation string. For output, `trellis_scan()` will create an opaque communication object (called `Trellis_Request`, similar to the `MPI_Request` object in MPI) and return a handle to it via the last parameter.

The control flow inside `trellis_scan()` is shown in Figure 4.2. Upon calling of the function, the input file is parsed. Then, depending on the type of the input file, two helper functions (visible only to Trellis-SDP, not to the programmers) are called:

```
int multi_file_scan(MetaHandler * meta, string op, Trellis_Request
                    * request);
```

```
FILE * single_file_scan(string path, string op, int rank);
```

If the type of the input file is SCP, it means that there is only one worker process that needs to be initiated, so the `single_file_scan()` function is called. This function starts up the worker process and builds a data stream between the master and worker process.

If the type of the input file is a metadata file, the information of the metadata file is analyzed and cached in an in-memory metadata object, which is passed as one of the parameters to the `multi_file_scan()` function. This function will then call the `single_file_scan()` function for each data block in the metadata file.

The results of the `multi_file_scan()` or `single_file_scan()` function calls are stored in the `Trellis_Request` object. `Trellis_Request` provides two types of member functions:

1. Read Functions:

   ```
   int trellis_scan_read(void * buffer, Trellis_Datatype datatype,
                         int nmemb, Trellis_Request request);

   int trellis_scan_readidx(void * buffer, Trellis_Datatype datatype,
                      int nmemb, int index, Trellis_Request request);

   int trellis_scan_readall(void ** buffer, Trellis_Datatype datatype,
                            Trellis_Request request);
   ```

   These functions allow the master process to read and store the data returned from worker processes. The programmer can either choose to read a specified number of bytes from all worker processes (`trellis_scan_read()`) or from a single worker process (`trellis_scan_readidx()`), or choose to read all the data that is available from all worker processes (`trellis_scan_readall()`). The type of the data to be read is determined by the `Trellis_Datatype`.

2. Synchronization Function:

   ```
   int trellis_scan_wait(Trellis_Request request, Trellis_Status status);
   ```

   This function ensures that all communications between master and worker processes are finished and the data streams opened by `trellis_scan()` are closed. If there are multiple phases in the program (which means multiple `trellis_scan()`s will be called in the master process), `trellis_scan_wait()` can serve as a barrier function between phases.

As an example, Figure 4.3 implements a data-parallel `trellis grep`, which is a `grep` operation on the distributed data (described as a metadata file, as shown in Figure 4.4). The code shown is the *complete code* for the master component, illustrating how simple a program can be if the problem is simple. For the worker component, we use the unmodified Unix `grep` program. The `trellis_scan()` takes in a metadata file and starts up the worker processes in each remote host to perform `grep` on its local data (line 20). The master process then reads in the results through the

```
 1 #include <string>
 2 #include <stdio.h>
 3 #include <trellis.h>
 4 #include <trellis_sdp.h>
 5
 6 int main(int argc, char * argv[]){
 7
 8      Trellis_Request request;
 9      Trellis_Status status;
10      void * buffer;
11       char * grep_arg = argv[1];
12      char * metafile = argv[2];
13      string op;
14      int items_read = 0;
15
16      op = "grep " + string(grep_arg);
17
18      trellis_init(argc, argv);
19
20      if(trellis_scan(metafile, op.c_str(), &request)<0){
21            fprintf(stderr, "Scan Failed\n");
22            exit(-1);
23      }else{
24
25            items_read = trellis_scan_readall(&buffer, Trellis_CHAR, request);
26            trellis_scan_wait(request, status);
27      }
28
29      trellis_finalize();
30      if(items_read > 0)
31              printf("%s\n", (char *)buffer);
32
33      return 0;
34 }
```

Figure 4.3: The sample code for the `trellis grep` program (the master component). The worker components are Unix `grep` executables. The location and distribution of the data is abstracted by the metadata file shown in Figure 4.4.

communication object by calling the `trellis_scan_readall()` function (line 25). Note that the `trellis grep` program performs most of its data-intensive operations on the remote hosts and transfers only a small amount of data (with type `Trellis_CHAR`) back to the master process. At the end of the program, the master calls `trellis_scan_wait()` (line 26) to close all open data streams.

Another sample code of a data-parallel application written in Trellis-SDP can be found in Appendix B. Section 5.3 provides more details of the description and evaluation of this application.

4.3.2 Trellis Gather

As discussed, `trellis_scan()` establishes communication channels between the master and worker processes. This interface is sufficient for embarrassingly data-parallel applications with no communications among worker processes. However, some complex parallel and distributed applications do require group communications. Thus, we also propose and implement two group-communication interfaces, one of which is called `trellis_gather()`. This interface is similar to the MPI collective-communication interface `MPI_Alltoall()` and `MPI_Alltoallv()` [28].

There are several papers on collective communications on a wide-area network, including the issues of performance and fault tolerance [2, 4, 20]; however, our efforts focus mainly on the API issues at this time. We also touch upon a bit of the performance issue and will discuss this in Chapter 5.

`trellis_gather()` is called in the master process and has the following declaration:

```
int trellis_gather(const char * metafile_source, const char * metafile_dest,
                   int ** index_table, Trellis_Datatype datatype);
```

The function takes a source metadata file, a destination metadata file, an index table specifying how data should be exchanged, and the datatype. The semantics of this function is the all-to-all communication among worker processes, where each worker process sends distinct data to all other worker processes.

As an example, Figure 4.5 illustrates an all-to-all data exchange among three worker hosts, according to the index table (*index_table* in Figure 4.5) provided by the programmer. Each row in the *index_table* specifies which data within one worker host needs to be sent to other worker hosts. For instance, worker host 1 indicates that data 3, 4 should be sent to worker host 2, and that data 5, 6 should be sent to worker host 3. Based on the *index_table*, Trellis-SDP will generate a new index table (*index_table2* in Figure 4.5) to determine the locations within each worker host where the data received from other worker hosts should be stored. For instance, after the data exchange, data 7 from worker host 2 will be stored at index 2 in the receiving file in worker host 1, and data 13, 14 and 15 from worker host 3 will be stored at index 3, 4 and 5 in the receiving file in worker host 1.

Figure 4.6 shows the control flow inside `trellis_gather()`. First, both the source metadata file and the destination metadata file are checked to make sure they have the correct file types. Then, as explained before, indices are calculated (i.e., from *index_table* to *index_table2*) to determine where the exchanged data should be stored. Finally, a simple helper program `sendfile` is initiated to perform the partial file transfer between worker hosts (i.e., to retrieve a portion of a remote file using `scp` for data transport).

As discussed earlier, the semantics of `trellis_gather()` are similar to those of `MPI_`

```
<?xml version="1.0"?>
<BlockList>
  <DataBlock>
    <Locator>scp:ading@nexus.westgrid.ca:~/data.1<</Locator>
    <Size>108003000</Size>
  </DataBlock>
  <DataBlock>
    <Locator>scp:ading@lattice.westgrid.ca:~/data.2</Locator>
    <Size>108003000</Size>
  </DataBlock>
  <DataBlock>
    <Locator>scp:ading@blackhole.westgrid.ca:~/data.3</Locator>
    <Size>108003000</Size>
  </DataBlock>
  <BlockSize>32</BlockSize>
</BlockList>
```

Figure 4.4: The metadata file used by the `trellis grep` program shown in Figure 4.3. The data is distributed on three hosts across three administrative domains: the University of Alberta, the University of Calgary and Simon Fraser University.

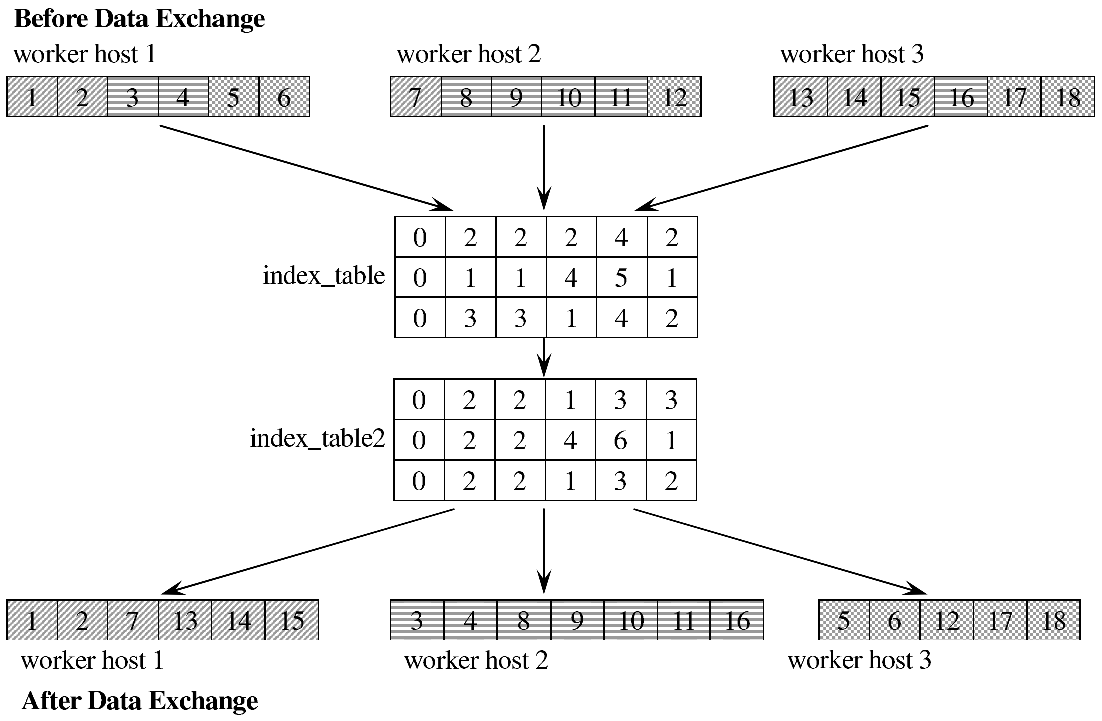


Figure 4.5: The illustration of the all-to-all communication.

`Alltoall()`. The major difference is that in `MPI_Alltoall()`, both the sending data and the receiving data reside in the memory, regardless of whether it is in a remote host or a local host, while in `trellis_gather()`, both the sending and receiving ends are files stored on disks, and are specified by metadata files. Therefore, the numbers in the *index_table* represent the offsets relative to files instead of displacements relative to memory buffers.

Figure 4.7 shows the sample code of Phase Three of the Parallel Sorting by Regular Sampling (PSRS) application (discussed in detail in Section 5.4). This is an example of how `trellis_gather()` is used. Appendix C lists the complete code for the PSRS application.

4.3.3 Trellis Reduce

The other collective-communication API we introduced and implemented is called `trellis_reduce()`. The inclusion of the function is again based on our hands-on experience with real scientific applications, and we found this API necessary when designing a parallel seismic data processing application (Section 5.5). `trellis_reduce()` is similar to MPI's `MPI_Reduce()` and performs global reduction operations across all worker processes. The reduction operation can be either: one of a predefined list of operations (such as global sum, maximum or minimum) or a program/executable provided by the user. Again, the difference between `trellis_reduce()` and `MPI_Reduce()` is that the former works at the file-level while the latter works at the memory-level.

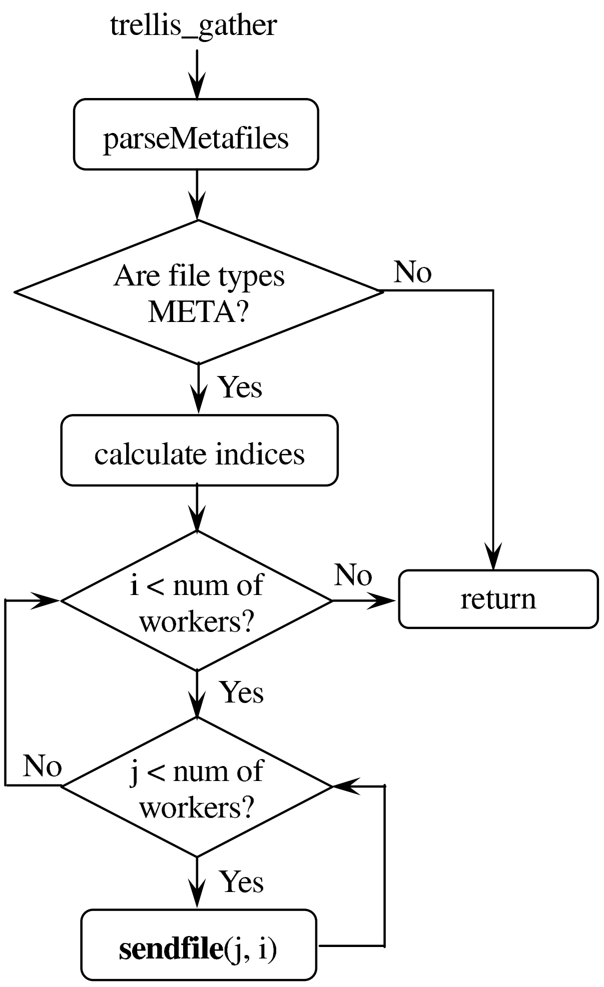


Figure 4.6: The control flow inside `trellis_gather()`.

The declaration of `trellis_reduce()` is:

```
int trellis_reduce(const char * metafile_source, const char * metafile_dest,
                   int count, Trellis_Datatype datatype, Trellis_Op op);

int trellis_reduce1(const char * metafile_source, const char * metafile_dest,
                    const char * reduce_op);
```

There are two reduce functions: `trellis_reduce()` and `trellis_reduce1()`. Both functions take a source metadata file and a receiving metadata file as input parameters. The difference is that `trellis_reduce()` performs the system-defined reduction operations `Trellis_Op`, while `trellis_reduce1()` performs the user-defined reduction operations (in the form of a stand-alone executable). In this section, we focus on the description of `trellis_reduce1()`.

Figure 4.8 illustrates the control flow inside `trellis_reduce1()`. The basic steps are :

1. A local reduction operation is performed at worker processes to reduce all the data that is local to the remote host.

```
...

int phase3(const char * metafile_localsorted, const char * metafile_gather,
                                        int ** index_table, Trellis_Datatype datatype)
{
        int rval;
        rval = trellis_gather(metafile_localsorted, metafile_gather, index_table, datatype);
        return rval;
}

int main(int argc, char * argv[]){

        int *sample_array;
        int **index_table;

        char metafile_in[] = "input.meta";
        char metafile_localsorted[] = "localsorted.meta";
        char metafile_gather[] = "sortgather.meta";
        char metafile_globalsorted[] = "sorted.meta";

        ...

        trellis_init(argc, argv);
        sample_array = phase1(metafile_in, metafile_localsorted);
        index_table = phase2(metafile_localsorted, sample_array);
        rval = phase3(metafile_localsorted, metafile_gather, index_table, Trellis_INT);

        ...

        trellis_finalize();
        return 0;

}
```

Figure 4.7: The sample code of Phase Three of the Parallel Sorting by Regular Sampling (PSRS) application showing how `trellis_gather()` is used.

2. A global reduction operation is performed at the master process to reduce all intermediate results from worker processes.

For step 1, we extend the definition of a metadata file so that the data block can also be a directory instead of just a file. For example, Figure 4.9 shows the metadata file we use for the seismic data processing application – `trellis LSAVA` (Section 5.5). Each block in the metadata file is a directory containing all the local files to be reduced.

In addition, we write two helper programs called `reduce_daemon` and `reduce`. The usage of the two programs are:

```
reduce_daemon reduce_op directory

reduce sendfile1 ... sendfileN receivefile reduce_op
```

The `reduce_daemon` program is initiated by the call to the `trellis_scan()` function inside the `trellis_reduce1()` function. The program in turn calls the `reduce` program to reduce all local data using the user-provided `reduce_op` program. The intermediate results are stored in files at remote hosts and the pathnames of these files are passed back to the master process via the `Trellis_Request` object.

In step 2, the master process simply calls the `reduce` program again, and reduces all intermediate results into the destination metadata file, using the same `reduce_op` program.

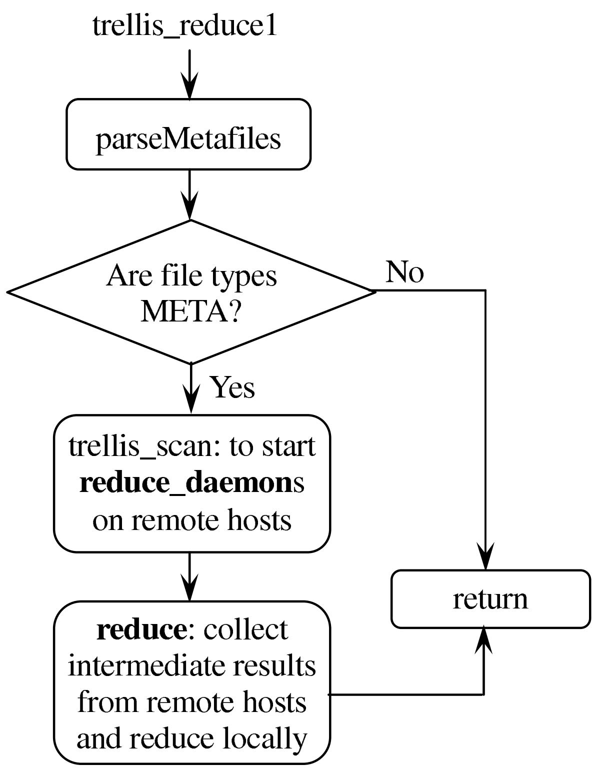

trellis_reduce1
parseMetafiles
Are file types META?
No
Yes
trellis_scan: to start **reduce_daemon**s on remote hosts
return
reduce: collect intermediate results from remote hosts and reduce locally

Figure 4.8: The control flow inside `trellis_reduce()`.

The reason we use a stand-alone helper program, instead of a helper function, is that it is simple and it is consistent with our ideas of using unmodified binaries in the programming system. In addition, if we want to make changes to these helper programs (e.g., to optimize the reduce algorithm in the `reduce` program), we do not need to rebuild the entire programming system.

Figure 4.10 shows several lines of codes taken from the `trellis LSAVA` program, which uses `trellis_reduce1()`. This function takes an existing reduce program `susum` to merge multiple source files into a single destination file. The complete code for the `trellis LSAVA` program is listed in Appendix D.

The Trellis-SDP functions we have proposed and implemented so far are mainly modelled after a subset of MPI functions targeted at data-parallel applications. We believe these functions should be able to handle most of the common data-parallel applications. Of course, there are cases we have not considered due to the limited number of benchmarking applications. We hope to extract more interesting APIs when we explore more applications in the future.

4.4 Concluding Remarks

In this chapter, we discussed the implementation of three major Trellis-SDP APIs: `trellis_scan()`, `trellis_gather()`, and `trellis_reduce1()`. These APIs are based on real applications and we believe they are able to handle most of the common data-parallel applications.

```
<?xml version="1.0"?>
<BlockList>
  <DataBlock>
    <Locator>scp:jasper-04:/usr/scratch/ading/SEISMIC/input/</Locator>
    <Size>65536</Size>
  </DataBlock>
  <DataBlock>
    <Locator>scp:jasper-07:/usr/scratch/ading/SEISMIC/input/</Locator>
    <Size>65536</Size>
  </DataBlock>
  <SpaceBlock>
    <Locator>scp:jasper-01:test/free/</Locator>
    <Size>512</Size>
  </SpaceBlock>
  <BlockSize>32</BlockSize>
</BlockList>
```

Figure 4.9: The metadata file used by the `trellis LSAVA` program shown in Appendix D. The data block is a directory containing all the local files to be reduced.

```
...

int main(int argc, char * argv[]){

      ...

      char * reduce_op = "susum";
      char * metafile_input = argv[2];
      char metafile_output[] = "LSAVA.data/LSAVA_OUTPUT.meta";

      trellis_init(argc, argv);

      ...

      if(trellis_reduce1(metafile_input, metafile_output, reduce_op) < 0)
      {
            fprintf(stderr, "Reduce Failed\n");
            exit(-1);
      }

      trellis_finalize();
      return 0;
}
```

Figure 4.10: The sample code of the seismic data processing application (`trellis LSAVA`) showing how `trellis_reduce1()` is used.

To summarize the main features of the Trellis-SDP APIs:

1. Trellis-SDP works at the file-level; the sending data and receiving data are stored on disks and represented by metadata files.

2. Trellis-SDP is able to use existing executables, not only for `trellis_scan()`, but also for collective-communication interfaces (`trellis_gather()` and `trellis_reduce1()`; both use several helper programs instead of helper functions).

Chapter 5

Applications and Empirical Evaluations

In this chapter, we present the empirical evaluations of Trellis-SDP. We evaluate the performance of four data-intensive applications written in Trellis-SDP, including execution times, speedups, and the breakdown of the execution times. We also measure the overhead in the execution times incurred by Trellis-SDP and show that, for naturally data-parallel applications with coarse granularity, our programing system is easy to use and has reasonably good performance.

To measure the `trellis_scan()` performance only, we use the distributed `trellis grep` (Section 4.3.1) and the content-based image retrieval (CBIR) application. To measure the performance of `trellis_scan()` and `trellis_gather()`, we use the Parallel Sorting by Regular Sampling (PSRS) application, which contains an all-to-all communication phase. To measure the performance of `trellis_scan()` and `trellis_reduce()`, we use a seismic data processing application (3D LSAVA migration application) developed at the Department of Physics, University of Alberta. We take the original OpenMP implementation of the 3D LSAVA migration application and convert it into a distributed-memory application using Trellis-SDP.

5.1 Experimental Methodology and Platform

All applications are run on the local-area network because a LAN is a more controlled environment for benchmarking applications with collective communications. To demonstrate that Trellis-SDP works across administrative domains, we present the benchmarking results for the `trellis grep` application (Section 4.3.1) run on a wide-area network.

All experiments run on the LAN use the same hardware configuration. We use AMD AthlonXP MP 1800+ processors running at 1.5 GHz, each with 1.5 GB of RAM. All local disk drives interface with the computer using a SCSI (Small Computer System Interface). The nodes are connected with a 100 Mbps, switched Fast Ethernet network. All nodes run Linux, with kernel version 2.4.18.

For the WAN settings, the remote nodes we use are located at the University of Calgary, the

Host Name	Administrative Domain	Configuration
nexus.westgrid.ca	University of Alberta	MIPS R16000 IP35 700MHz, 8GB RAM, Irix
lattice.westgrid.ca	University of Calgary	Alpha ES45 1GHz, 4GB RAM, Tru64
blackhole.westgrid.ca	Simon Fraser University	AMD Opteron 2.4GHz, 4GB RAM, Linux

Table 5.1: The WAN settings for benchmarking the `trellis grep` application. The machines are located at three different administrative domains: the University of Alberta, the University of Calgary and Simon Fraser University.

University of Alberta and Simon Fraser University. Table 5.1 provides the detailed information of these hosts.

Finally, as mentioned in Section 4.1, all the data needed for computations in the experiments are manually distributed at the start since we are targeting naturally-distributed applications. In addition, metafiles describing the distribution of the data are ready before the computation.

5.2 Distributed Grep

In this experiment, we benchmark the distributed `grep` (`trellis grep`) application introduced in Section 4.3.1. As described earlier, `trellis grep` performs a `grep` operation on distributed data. The source code of the master component is listed in Figure 4.3. The worker component of the application is the Unix `grep` program.

We perform this benchmark in both the LAN and WAN environments. For the LAN environment, we use the metadata file listed in Figure 4.1; for the WAN environment, we use the metadata file listed in Figure 4.4. We test three sets of input data in both cases, and the total size of the data are 309MB, 618MB, and 927MB, respectively. The data is uniformly distributed across three worker hosts. All experiments are run 10 times, and only the average numbers are reported. The standard deviations are less than 3% of the average.

Table 5.2 shows the execution times of `trellis grep` observed from the master host and each worker host for all three data sets in the LAN environment. Since the LAN environment is homogeneous, the results observed from all worker hosts are almost identical. However, there is a difference (about 0.7 seconds) between the execution times observed from the master host and the worker hosts. This is due mainly to the overhead of the programming system (e.g., `ssh` startup time). Additional discussion on the overhead is provided in later sections.

Table 5.3 shows the execution times of `trellis grep` observed from the master host and each worker host for the same input data sets in the WAN environment. This time, the execution times at different worker hosts are significantly different (for example, with 309MB input data, the execution time at lattice.westgrid.ca is 7.81 seconds, while the execution time at blackhole.westgrid.ca is only 0.25 seconds), as each worker host has different hardware and software configurations. For all tests, the total execution time observed at the master host is always greater than the maximum execution time of all worker hosts. This is the effect of synchronization at the master host, plus the overhead of the programming system.

Hosts	Execution Times (in seconds)		
	Data Size: 309MB	Data Size: 618MB	Data Size: 927MB
jasper-03 (sequential)	1.48	3.01	4.73
jasper-03 (master)	1.25	1.77	2.28
jasper-00 (worker)	0.53	1.07	1.58
jasper-01 (worker)	0.53	1.04	1.55
jasper-02 (worker)	0.53	1.05	1.51

Table 5.2: The execution time of the `trellis grep` application, as observed from the master host and each worker host in a LAN environment, with three different sets of input data.

Hosts	Execution Times (in seconds)		
	Data Size: 309MB	Data Size: 618MB	Data Size: 927MB
cleardale.cs.ualberta.ca (master)	9.65	17.37	25.01
nexus.westgrid.ca (worker)	5.15	10.27	15.44
lattice.westgrid.ca (worker)	7.81	15.54	23.19
blackhole.westgrid.ca (worker)	0.25	0.49	0.69

Table 5.3: The execution time of the `trellis grep` application, as observed from the master host and each worker host in a WAN environment, with three different sets of input data.

The purpose of this experiment is to demonstrate that for simple data-parallel applications, Trellis-SDP works in both the LAN and WAN environments. We present additional performance-related metrics in later experiments.

5.3 Content-Based Image Retrieval

In this experiment, we examine additional performance metrics of `trellis_scan()` and related functions. We implement a typical information retrieval application: content-based image retrieval (CBIR).

5.3.1 Application Description

For a computer, retrieving images based on image content is a difficult task. Unlike human beings, who may easily recognize objects in an image – say, "a red car"– computers do not understand the contents of the image. Researchers in different disciplines (e.g., computer vision, signal processing, biology, neuroscience) have proposed various algorithms in this area [31]. It is ideal to parallelize a

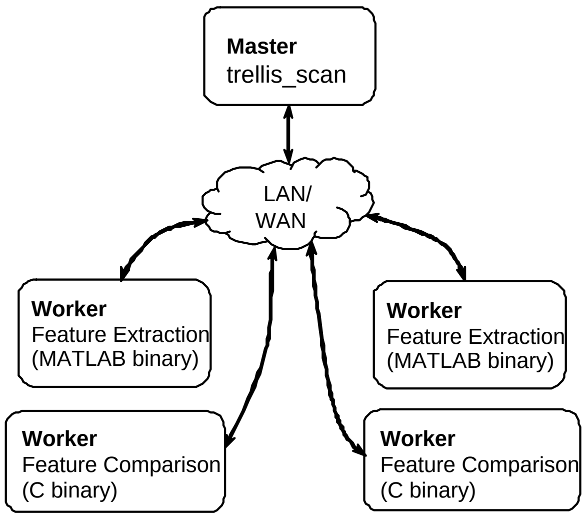


Figure 5.1: The control flow of the content-based image retrieval (CBIR) application

CBIR application using our programming system because it is data-intensive, is easy to decompose, and the computation is embarrassingly parallel.

The process of writing a distributed CBIR application using Trellis-SDP is similar to that of the CBMR example described in Section 1. The sequential CBIR application takes a sample query image and performs a feature-extraction algorithm on the image to generate a multidimensional feature vector (e.g., color, edge and texture information are vector components). The feature vector is then searched through the feature space to find the top n most-matched feature vectors. That is, the feature space is formed by all the feature vectors that have been generated by preprocessing all images in the image database. New feature vectors are continuously added to the feature space during the query processes. It should be noted that there are certain issues related with distributed CBIR applications, such as local relevance versus global relevance [5]. But we use this application mainly for benchmarking purposes and do not look at these issues.

5.3.2 Experimental Setup

To write a distributed version of CBIR, the application is first decomposed into a master component and two worker components: feature extraction and feature comparison. The number of worker components depends on how the image database is distributed. Figure 5.1 depicts the control flow of the distributed CBIR application.

As shown in the figure, the two worker components are written using different tools. We build the feature-extraction component using MATLAB – since it greatly simplifies matrix-based programming – while we build the feature-comparison component using standard C. In practice, when using Trellis-SDP, a programmer may choose to write the worker component using his/her favorite

Image Database	Sequential		2 Worker Hosts		4 Worker Hosts		8 Worker Hosts	
Size	Time	σ	Time	σ	Time	σ	Time	σ
60,000	101.67	0.87	56.40	1.28	28.77	1.03	16.19	1.04

Table 5.4: The raw execution time of the CBIR application: Time is the average execution time (5 repeated runs) in seconds, σ is the standard deviation.

language, to speed up the software development process.

This experiment is performed in a LAN environment and we use up to 8 computing nodes (as described in Section 5.1). The image database contains 60,000 images with a total feature space of 600 MBytes (i.e., all the feature vectors take 600 MBytes).

5.3.3 Experimental Results

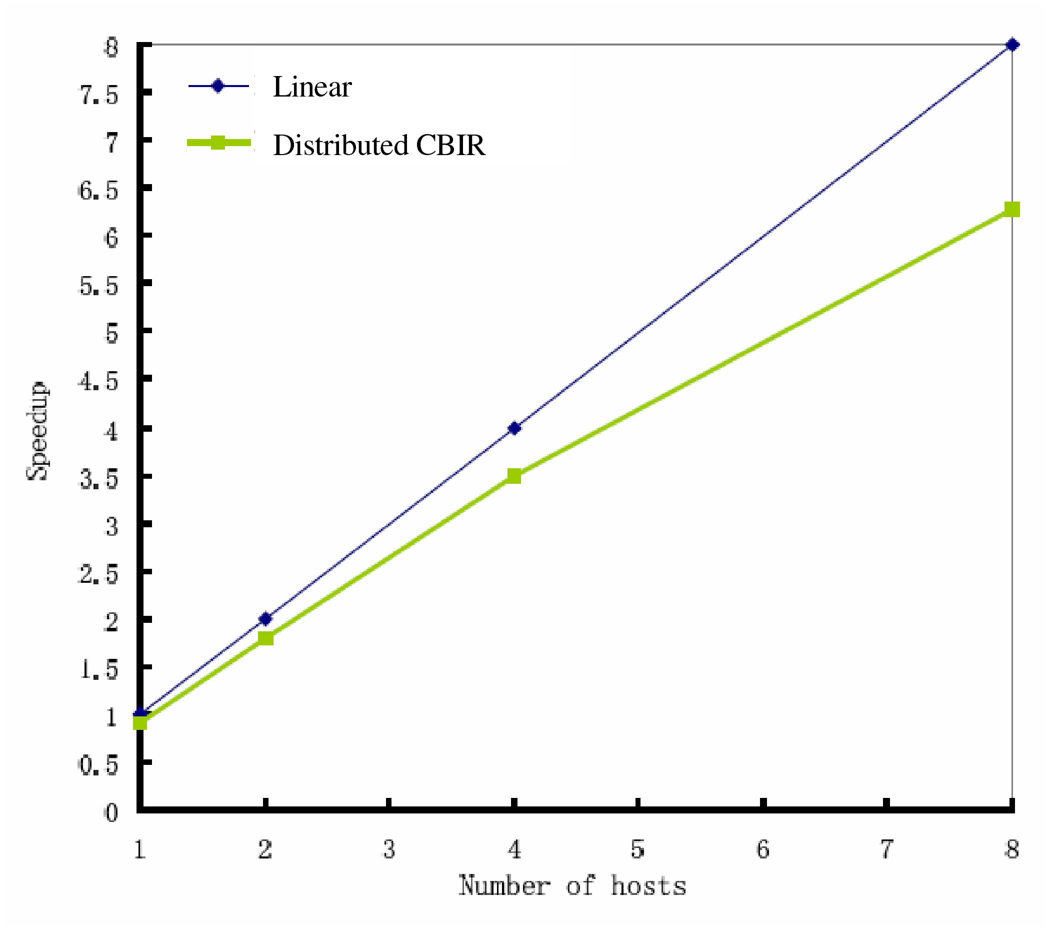


Figure 5.2: The speedup of the distributed CBIR application. The size of the image database is 600MB.

The main experiment is the scalability test, which involves distributing the image database onto different numbers of nodes. This is shown in Table 5.4 where the average raw execution times of

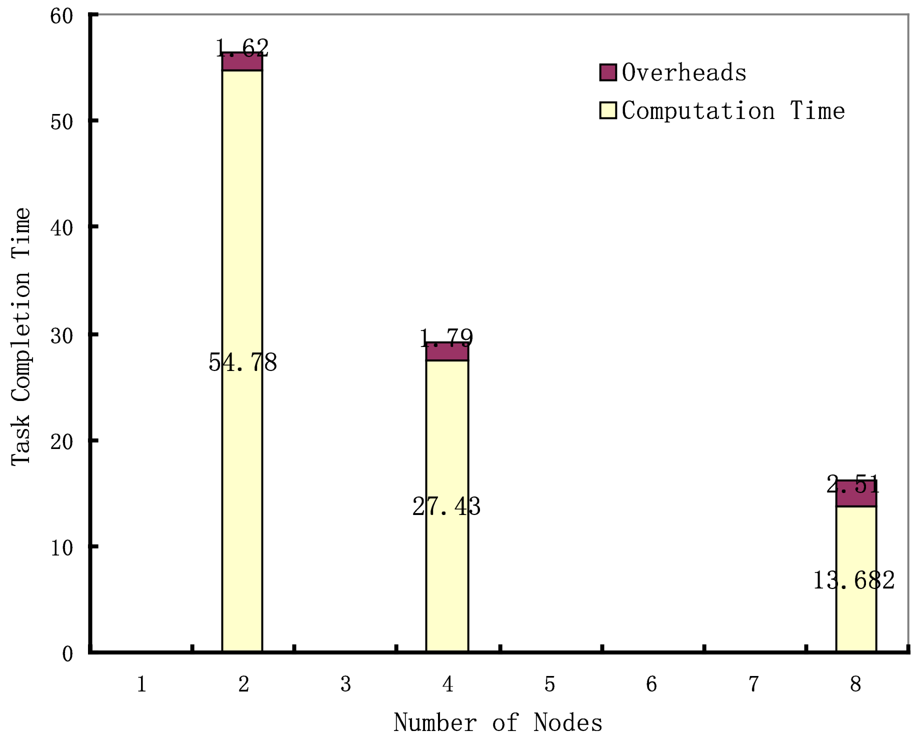


Figure 5.3: The overheads of the programming system in the CBIR application.

the CBIR application on 2, 4 and 8 worker hosts, plus the sequential execution time, are given. We also present a speedup graph to further illustrate the scalability of the application (Figure 5.2).

The distributed CBIR application shows good scalability when the number of participating nodes increases. This is expected, since the distributed CBIR is naturally parallel. The contribution of Trellis-SDP is in simplifying the implementation of the CBIR application (Appendix B) and in minimizing the overheads that detract from linear speedup.

To gain some insight into the overheads (e.g., the startup time of `ssh` connections and the encryption of the communication channel), we measure and factor out the `ssh` startup times, compared to the overall execution time (Figure 5.3). The worst case overhead is 15.5% when the number of nodes is 8. This is understandable since the number of `ssh` calls and connections grows linearly for CBIR, with the number of nodes. As shown with the next application (Section 5.4), `ssh` startup overheads can become a bottleneck as the number of worker processes grows, especially when group communication is involved.

5.3.4 Discussion and Conclusion

Content-based image retrieval is a typical data-intensive and data-parallel application. In practice, many real applications fall into this category, for example, distributed database operations, image processing and data mining. Trellis-SDP is designed for this kind of application and our experimental results demonstrate that the overheads introduced by Trellis-SDP are minimal.

5.4 Parallel Sorting by Regular Sampling

We use the Parallel Sorting by Regular Sampling (PSRS) application to benchmark the `trellis_gather()`, in addition to `trellis_scan()`. The main purpose of the PSRS experiment is to show that our programming system works for parallel applications with all-to-all communication phases.

5.4.1 Application Description

Parallel Sorting by Regular Sampling is an algorithm that is suitable for many parallel architectures. It has "good load balancing properties, modest communication needs, and good locality of references" [21]. To sort the data distributed on p hosts, the algorithm divides the whole process into four phases, which fits well with our programming system.

In Phase One, each worker component sorts its local data using `quick sort`. Then, regular samples are collected from each sorted local data and merged together in the master component. Merged regular samples are also sorted using `quick sort`. In Phase Two, $p - 1$ pivots are found from the sorted regular samples and sent back to each worker component, which partitions its local data according to the pivots. In Phase Three, there is a communication-intensive data exchange where the i^{th} partition in each worker component is transferred to the i^{th} worker. Finally, in Phase Four, the exchanged partitions in each worker are merged using n-way `merge sort`, and the algorithm ends.

Figure 5.4 shows the control flow of PSRS using Trellis-SDP. To simplify the implementation, we create three worker components on each remote host: the first component performs the local sort and collects samples; the second reads in pivots and generates the partition index information; the last component exchanges the data partitions using `trellis_gather()`, and does a final local merge sort. The sorted data still resides in remote hosts and is represented by a metadata file in the master host.

5.4.2 Experimental Setup

This experiment is performed in a LAN environment. The experimental setup is the same as the one described in Section 5.3.2, except that the dataset used contains 1 GB of unsorted (binary) integers (i.e., 256 million keys), in total. While benchmarking, no other applications are run on the same cluster.

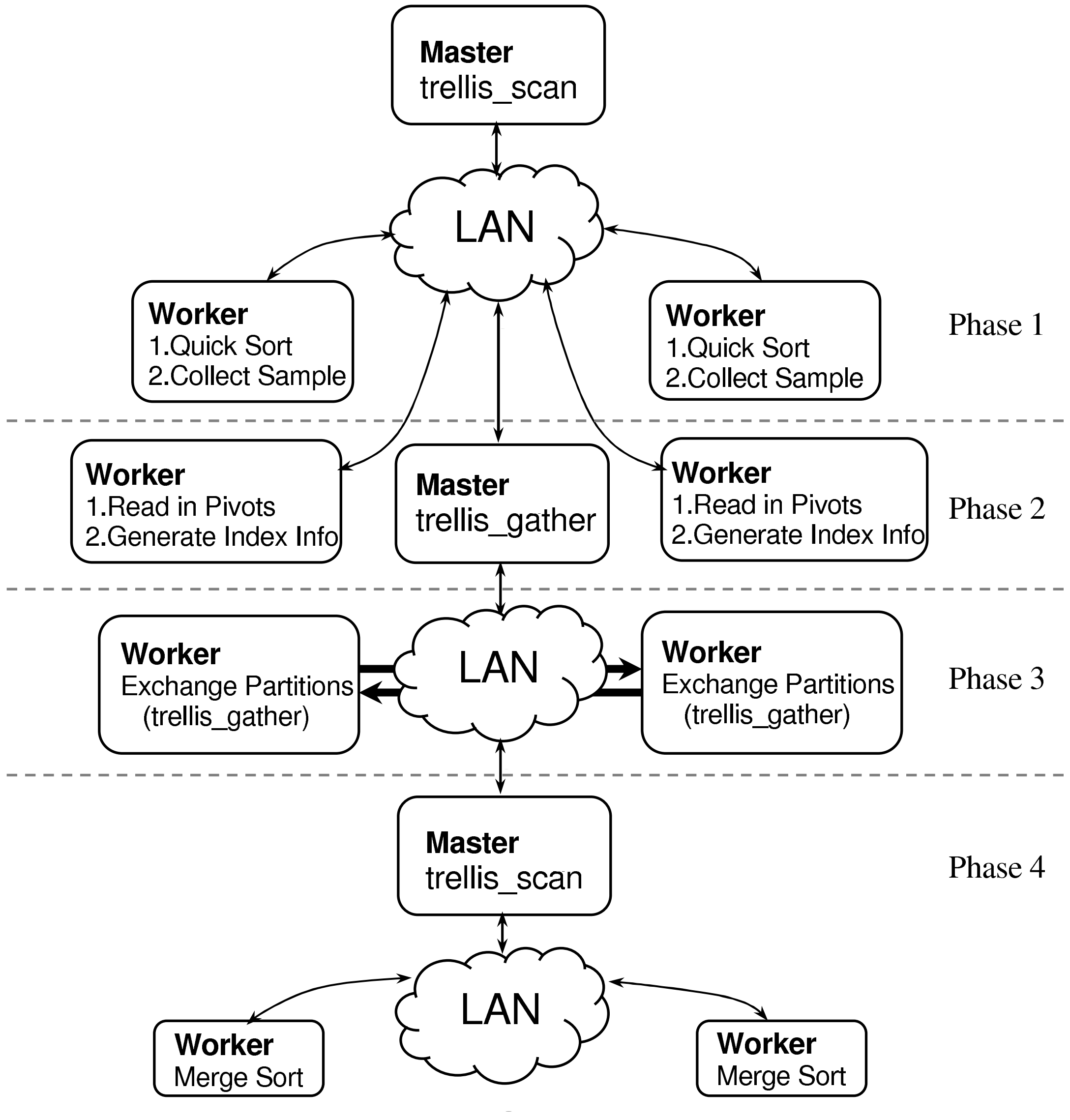


Figure 5.4: The control flow of the PSRS application.

Data Size in Total	Total Real Execution Times (in seconds)			
	Sequential	2 Worker Hosts	4 Worker Hosts	8 Worker Hosts
1GB	184.62	149.34	73.61	48.88

Table 5.5: The raw execution time of the PSRS application on 2, 4 and 8 worker hosts.

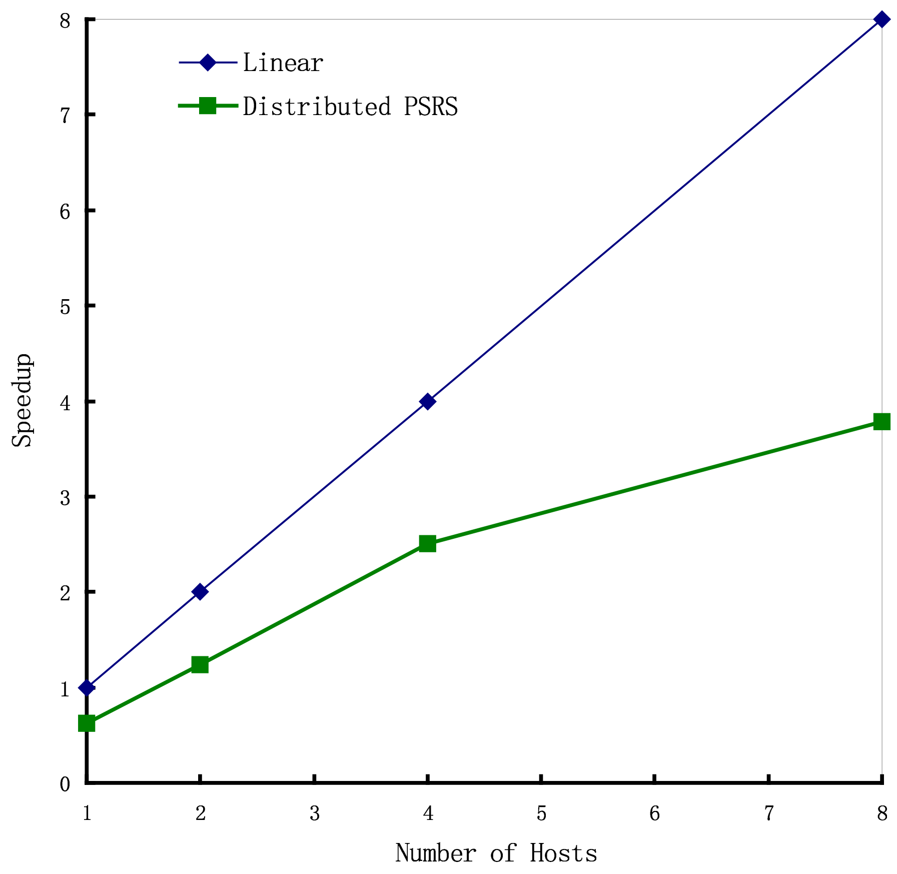


Figure 5.5: The speedup of the PSRS application.

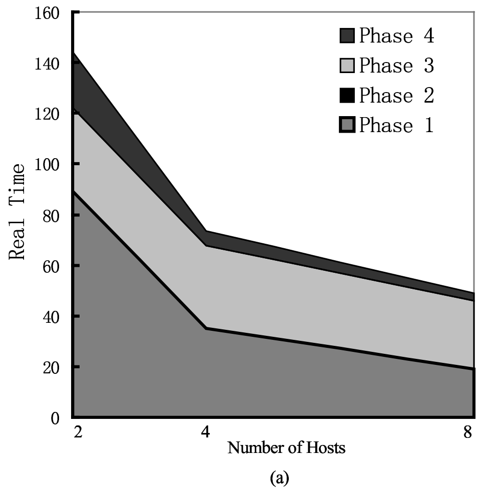
5.4.3 Experimental Results

Scalability

The raw execution time and speedup graphs of the distributed PSRS application are given in Table 5.5 and Figure 5.5. The execution time is an average of 5 repeated runs. As seen from the figure, for 8 worker hosts, we obtain a speedup of 3.7. This is not high, compared with the previous CBIR experiment but, considering the all-to-all communications, and a secure data transfer, the result is reasonable. In fact, we are more interested in identifying the overheads of Trellis-SDP for group communications.

Execution Time Breakdown

We use the phase-by-phase analysis to quantify the execution times in each phase. Figure 5.6 illustrates the breakdown of the execution time of PSRS. As expected, Phase Three becomes a performance bottleneck when the number of worker hosts increases. For example, when there are only two worker hosts, Phase Three is 22% of the total execution time. But, when the number of worker hosts increases to eight, Phase Three grows to 55%.



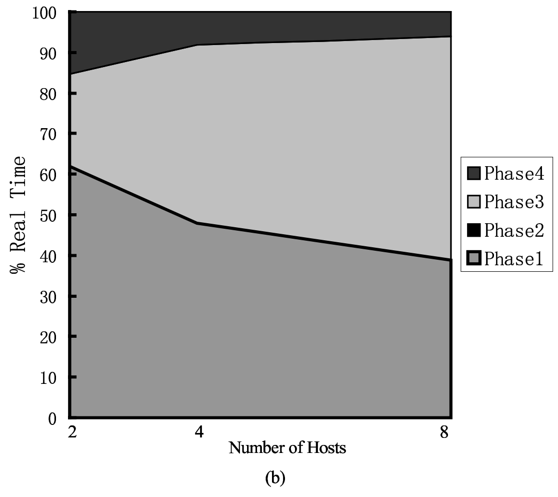


Figure 5.6: The breakdown of the execution time of the PSRS application with default `ssh`. (a) phase-by-phase with real time; (b) phase-by-phase with percentage of real time.

The major reasons for this bottleneck are the saturation of the network bandwidth (i.e., exchanging millions of keys), the number of `ssh` connections, and the data encryption overheads. For all-to-all communications among n worker components in Phase Three, there are $O(n^2)$ `ssh` connections.

To further quantify the overhead, we perform an additional test by replacing all `ssh` connections in Phase Three with `rsh` (which is faster than `ssh` since `rsh` uses clear text channels). Figure 5.7 shows the new breakdown of the execution time of PSRS with `rsh` enabled in Phase Three. With `rsh`, both the total execution time and the percentage of the execution time for Phase Three are reduced. Figure 5.8 more directly shows the impact of the choice of the underlying communication mechanism.

5.4.4 Discussion and Conclusion

In this section, we have shown the experimental results for the PSRS application. As expected, due to an all-to-all communication phase in the application, the overhead introduced by Trellis-SDP is significantly larger than that in the CBIR application. In the future, we plan to explore the communication optimization of `ssh` for large data transfers.

5.5 Seismic Data Processing by 3D LSAVA Migration

We use the 3D LSAVA migration application to test `trellis_scan()` and `trellis_reduce()`. For this application, we test two sets of input data: one small data set (32 MB) and one large data set (6 GB). Since a considerable amount of resources are required to process the large data set (i.e., it takes more than two weeks to process the 6 GB data on 8 worker hosts), it is impractical to com-

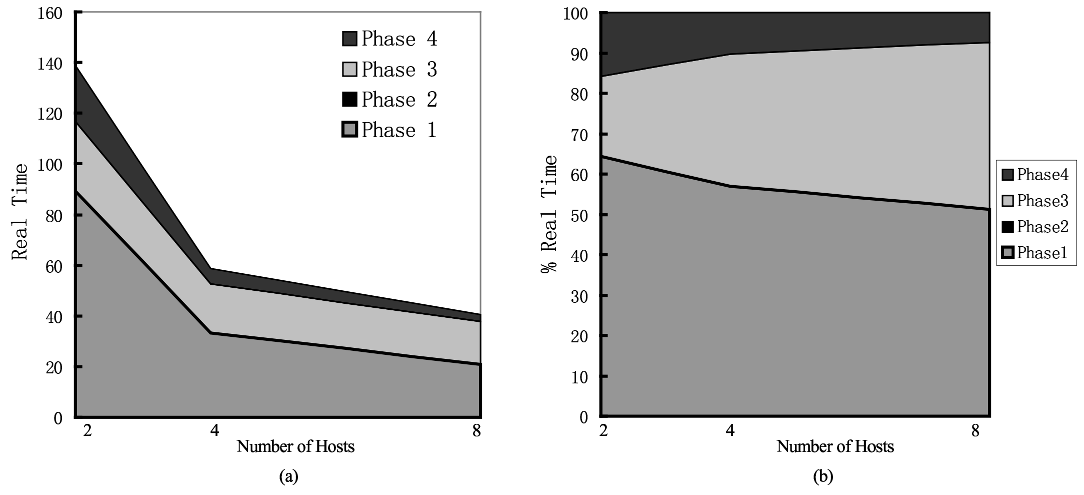


Figure 5.7: The breakdown of the execution time of the PSRS application with `rsh` enabled in Phase Three. (a) phase-by-phase with real time; (b) phase-by-phase with percentage of real time.

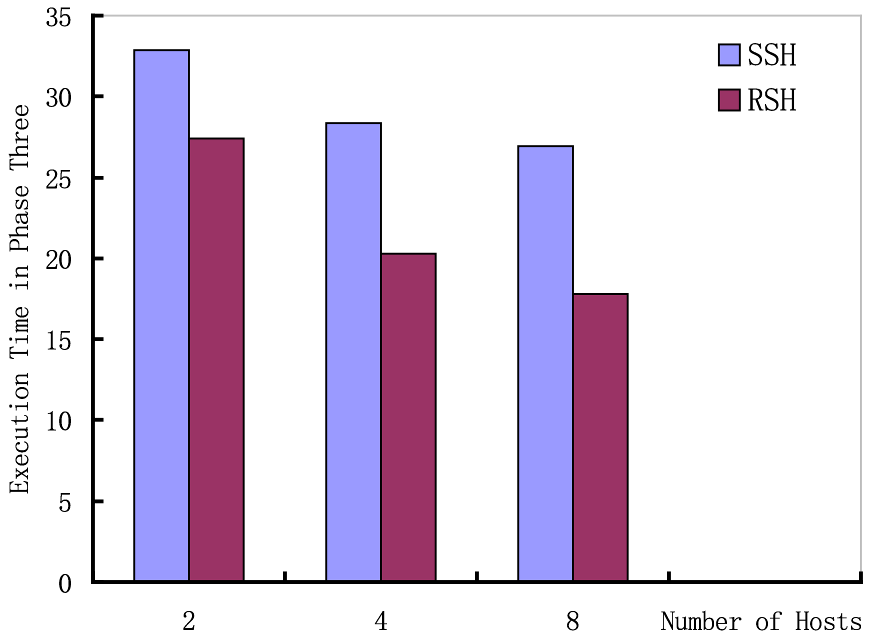


Figure 5.8: The overheads of the programming system in the PSRS application with different underlying communication mechanisms.

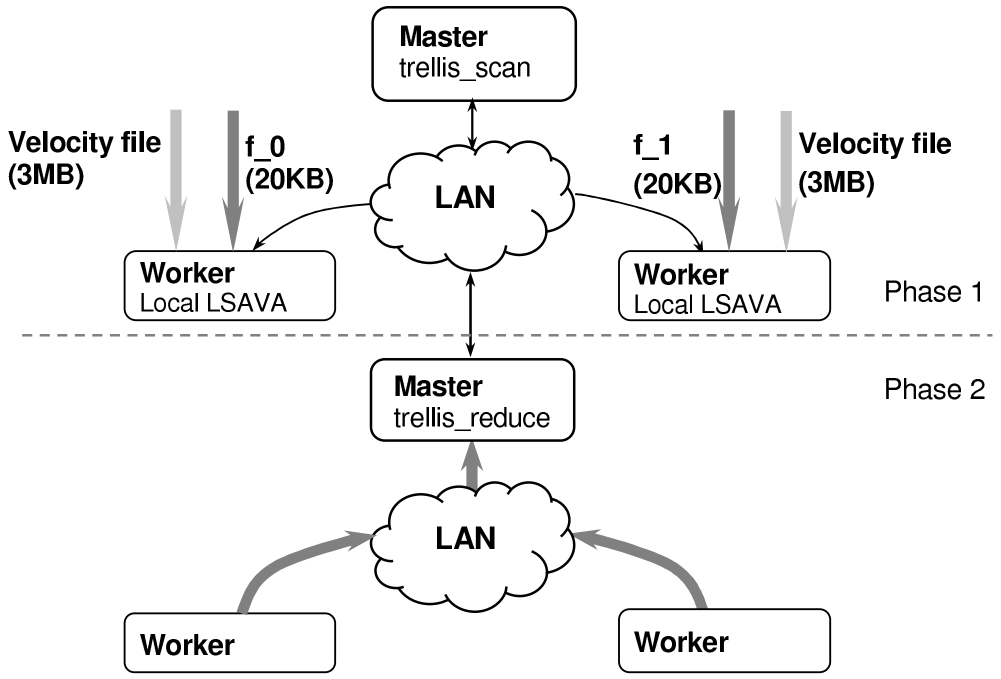


Figure 5.9: The control flow of the 3D LSAVA migration application with small input data set.

plete the entire computation in a reasonably short time. Nevertheless, it is still demonstrated that Trellis-SDP works for the application that handles large amounts of data, and we provide all the experimental results collected.

5.5.1 Application Description

To demonstrate that Trellis-SDP is applicable to large-scale scientific applications, we collaborate with the research group in the Seismic Image Processing Lab, Department of Physics, University of Alberta. We port their novel 3-dimensional least-square amplitude versus angle (3D LSAVA) migration application for seismic data processing to clusters of workstations using Trellis-SDP.

The goal of the 3D LSAVA migration application is to process the raw seismic data, and make good-quality images of the earth's interior. The raw seismic data is collected by the reflection seismic prospecting technology. For example, at first, a survey area is defined at the earth's surface; then, a mesh of sources (i.e., the equipment for impulsive sound waves) is deployed and activated across the area; finally, echoes of the sources (arriving from multiple directions) are recorded by the receivers (i.e., geophone or hydrophone) nearby [6].

To process the data, first, the data is transformed from time-domain into frequency-domain using Fast Fourier Transform (FFT). Then, for each frequency unit, each layer is iteratively processed by the 3D LSAVA migration algorithm. The result of each frequency unit is a fix-sized two-dimensional

	size of input	number of frequencies	number of layers	size of global model
small data	32 MBytes	99	130 (4 meters/layer)	1.94 MBytes
large data	6 GBytes	178	210 (20 meters/layer)	9 GBytes

Table 5.6: The parameters of the small data set and the large data set for the 3D LSAVA migration application.

matrix. The final model is generated by globally summing up all matrices [36].

The original 3D LSAVA migration application was implemented using the shared-memory OpenMP system. In this implementation, the global model resides in the main memory and is shared by all processes. Each process updates this memory region after processing a frequency unit. To port this application to a distributed-memory environment, we factor out into a separate executable the function to process all frequency units in parallel via `trellis_scan()`, and generate the result for each frequency unit locally. As illustrated in Figure 5.9, each worker process takes in an input frequency file and a velocity file, and generates a two-dimensional model onto the local disk. When all local models are generated, a global sum (i.e., a merge operation) is invoked by calling `trellis_reduce()`, which produces the global model. Note that the reduction operation uses an existing binary taken directly from the Center for Waveform Phenomena/Seismic Unix (CWP/SU) package [32]. The final global model can be visualized (i.e., the seismic image) and examined by experts.

5.5.2 Benchmark Setup

The experiments are run in a LAN environment only; the LAN settings are described in Section 5.1.

We use two sets of input data to evaluate this application. One is a small synthetic data set, which contains 32MB of input data, and the final model is approximately 2MB. We perform simple speedup tests and measure the breakdown of the execution time, using the small data set.

The other input data used is a 6GB 3D SEG/EAGE (i.e., Society of Exploration Geophysicists/European Association of Geoscientists and Engineers) salt model dataset, which has been widely used by the oil and gas industry for the research of three-dimensional seismic surveys. For 8 worker hosts, the processing of this data will produce 198GB of output data on each worker host before final reduction. Since there is not enough disk space to accommodate the data, we choose to make some modifications to the code. Instead of computing all layers in a row, and generating a large set of local models for each worker host, we have each worker process compute only a single layer for all frequencies at a time. After each layer is completed at all worker processes, a reduction operation is performed and an intermediate model for that layer is generated and moved to a backup storage. Then, the computation and reduction for another layer is initiated until we finish processing all layers. Finally, a global reduction is performed on all intermediate models to generate the final global model. In this way, we require less than 4GB of disk space to process one layer for

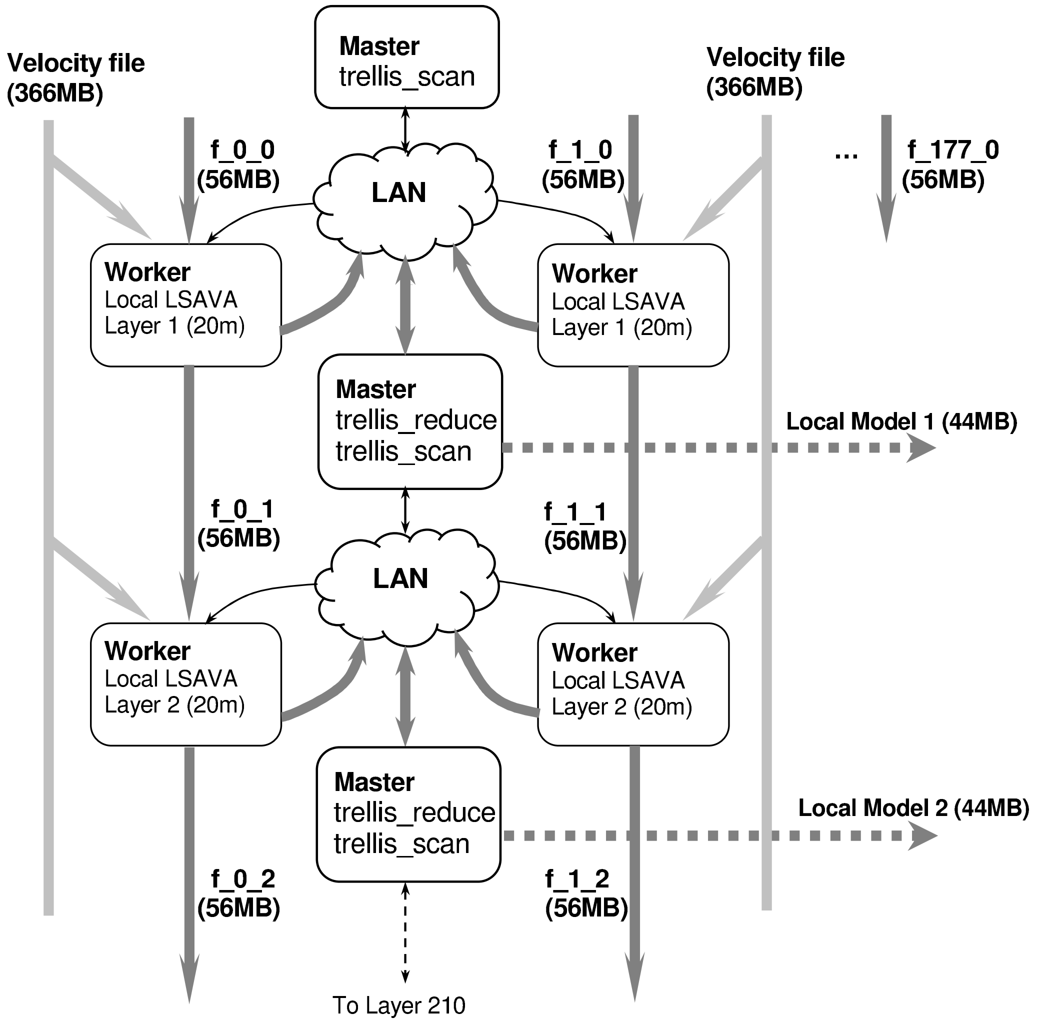


Figure 5.10: The control flow of the 3D LSAVA migration application with large input data set.

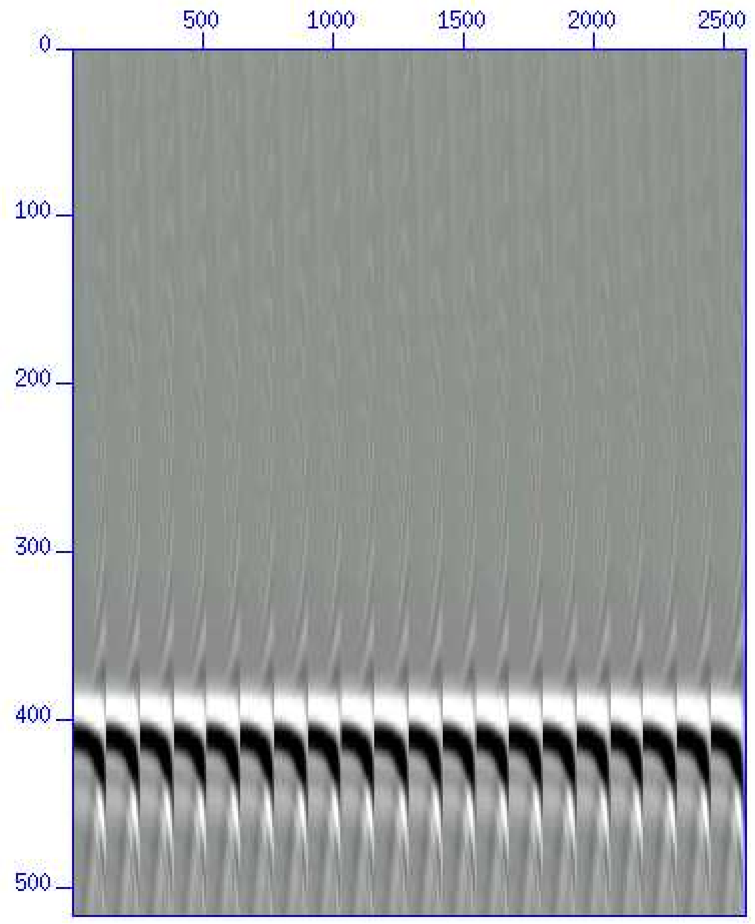


Figure 5.11: The visualization of the 3D LSAVA migration application for the small input data set.

each worker host. The control flow of the modified 3D LSAVA migration application is shown in Figure 5.10.

Some detailed parameters of the two data sets are listed in Table 5.6.

5.5.3 Experimental Results

Small Synthetic Data

Figure 5.11 illustrates the visualized result of the 3D LSAVA migration application for the small data set. Since this is only a synthetic data set, it does not have real geographic meanings. However, in order to study the properties of the application, it is still valuable to measure the scalability, the breakdown of the execution times of different phases, and the `ssh` startup overheads. The results are shown in Figures 5.12, 5.13 and 5.14, respectively.

Figure 5.12 indicates that the distributed 3D LSAVA migration application has good scalability. For example, when the number of worker hosts is 8, we get a speedup number of 7.3. This is because the majority of the computation is done in the scan phase (i.e., phase one), and the reduction phase (i.e., phase two) takes up to only 5% of the total execution time, as shown in Figure 5.13. The `ssh`

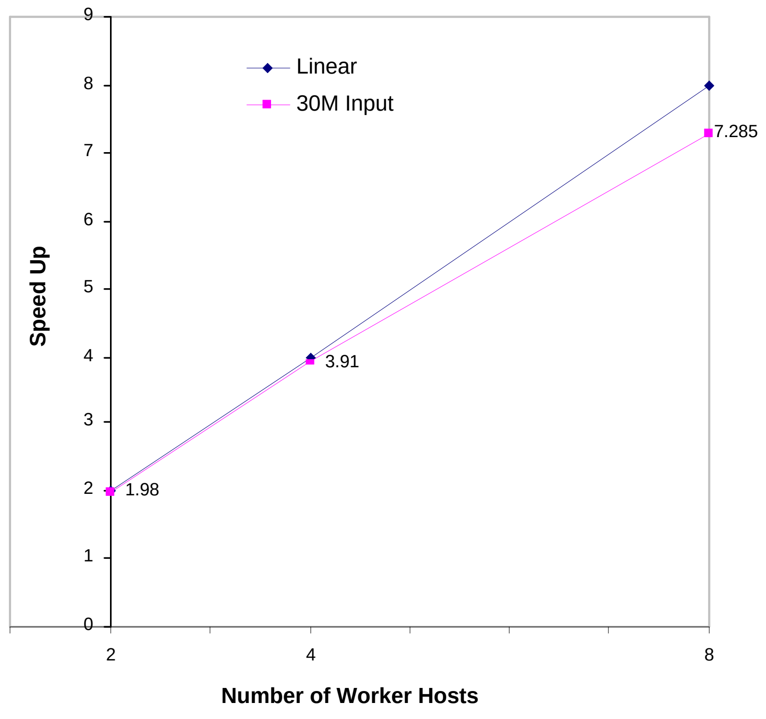


Figure 5.12: The speedup of the 3D LSAVA migration application in a LAN environment.

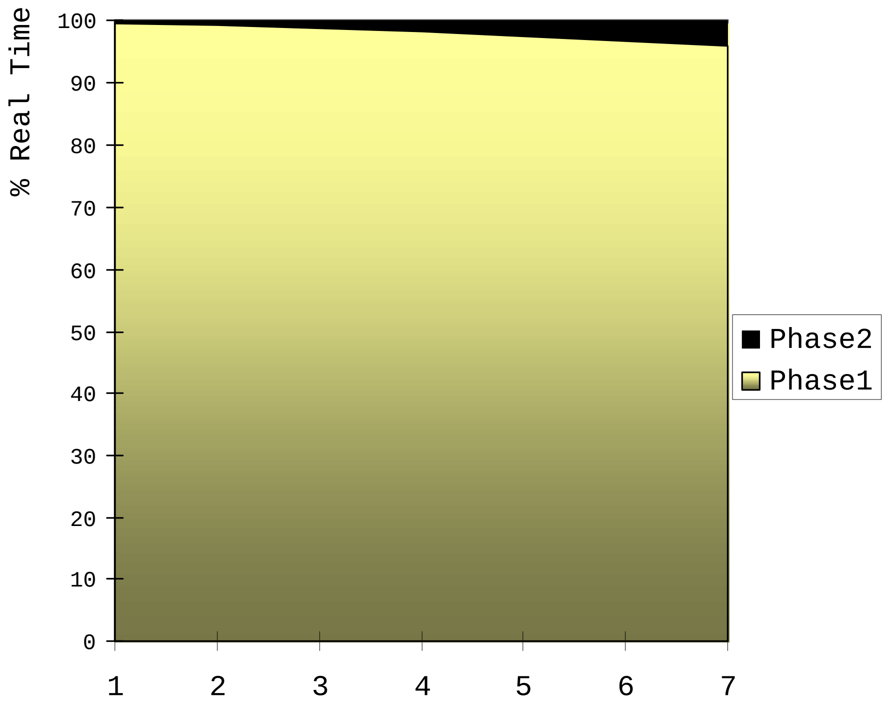


Figure 5.13: The breakdown of the execution time of the 3D LSAVA migration application in a LAN environment.

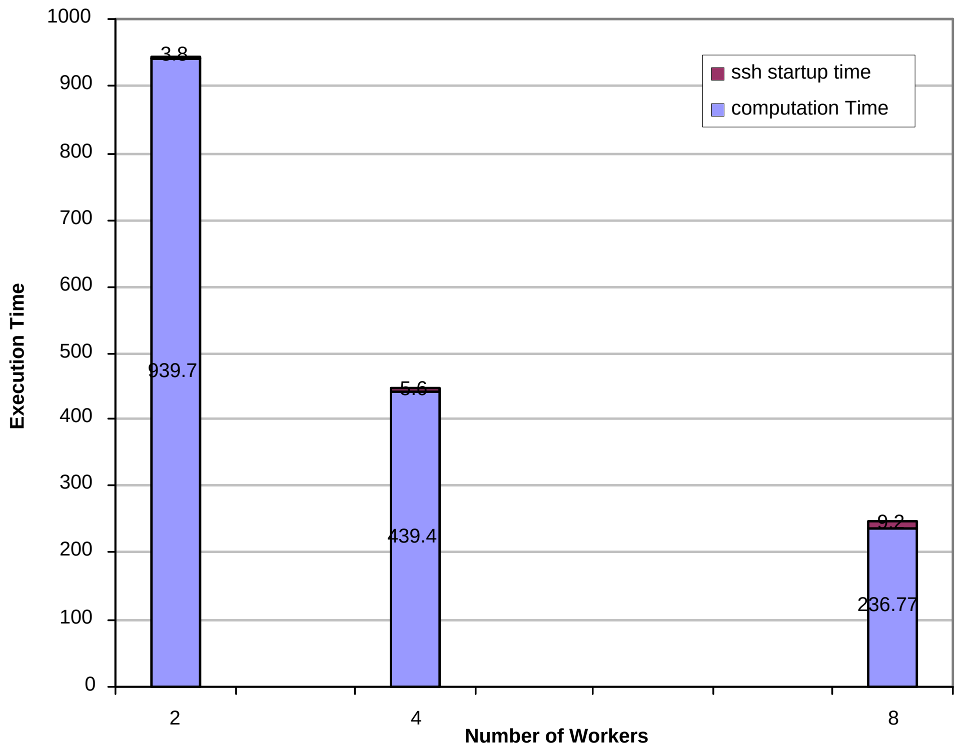


Figure 5.14: The `ssh` startup overhead of the 3D LSAVA migration application in a LAN environment.

	time on scan operations	time on reduction operations	total execution time
120 layers	210.67 hours	1.93 hours	212.6 hours

Table 5.7: The total time spent to generate 120 layers of the final model of the 3D LSAVA migration application using 8 worker hosts. The size of the input data is 6GB.

startup overhead is also negligible, according to Figure 5.14. The worst case overhead is 3.7% of the total execution time when the number of worker hosts is 8.

3D SEG/EAGE Salt Model Dataset

We complete the processing of 120 layers (2400 meters) out of 210 layers of the global model, and the result has been verified to be correct. Figure 5.15 illustrates one profile of the 3D model that is generated. In the figure, the noticeable curves are reflections resulting from different properties of adjacent strata.

The total time spent in generating the 120 layers of the global model using 8 worker hosts is given in Table 5.7. The times spent on all scan phases and all reduction phases are also reported in the same table. Since a considerable amount of time is required to complete the whole computation, given the current system configurations, it is impractical to measure speedups at this problem scale. (For example, 212.6 hours (8.86 days) are spent to process 120 layers; at least another estimated 160 hours will be needed in order to complete the entire 210 layers). However, from Table 5.7 we can see that the reduction operations take less than 1% of the total execution time. This indicates that the 3D LSAVA migration application on the large data set has the potential to have good scalability if we add more processors to the computation.

There are several factors contributing to the long running time of the modified 3D LSAVA migration application (Section 5.5.2) in the current system configurations. In addition to the time for the additional I/O (writing to and reading from disks) between scan phases, and the time for the extra reduction phases that generate the intermediate models, there are two other major overheads introduced by the modification:

1. If we process one layer at a time, we need to read in the velocity file (366MB) for each layer, which takes approximately 10 seconds each time. If we process all layers in a row (before modification), we need to read in the velocity file only once.

2. If we process one layer at a time, we also run `fftw_create_plan()` for each layer, which takes approximately 33 seconds each time. If we process all layers in a row, we need to perform `fftw_create_plan()` only once.

Given the above facts, if we had sufficient storage on each worker host and processed all layers in a row, we would expect to reduce the total execution time by at least 31.5 hours on 8 worker hosts.

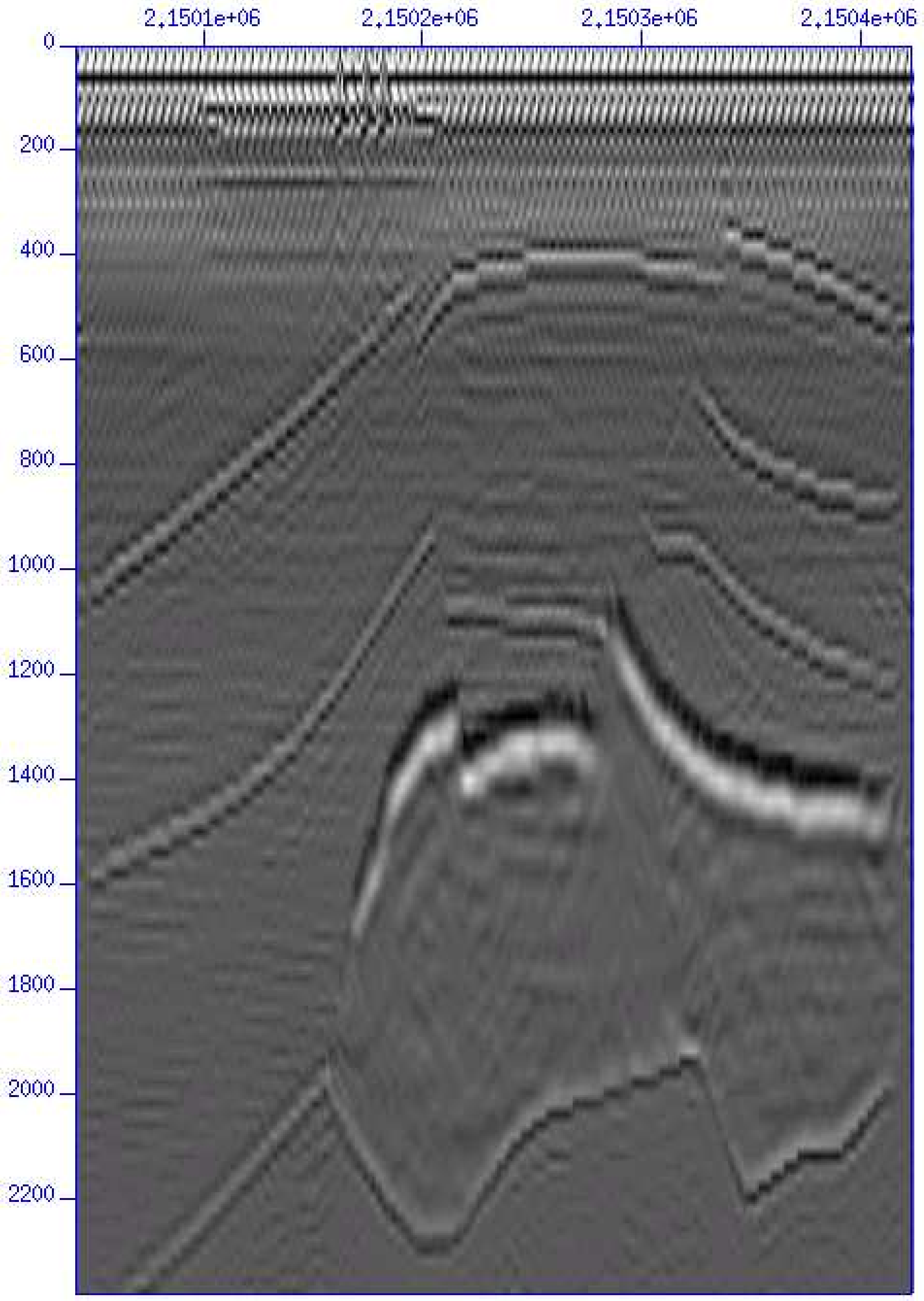


Figure 5.15: The visualization of the 3D LSAVA migration application for the 3D SEG/EAGE salt model data set.

5.5.4 Discussion and Conclusion

In this section, we presented the experimental results of the 3D LSAVA migration application, a project carried out in cooperation with the researchers in the Department of Physics, University of Alberta. The significant features of this experiment are:

1. The 3D LSAVA migration application is a non-trivial application, developed independently at the Department of Physics, University of Alberta.

2. The amount of data that was processed is extremely large.

3. Although the porting of the original OpenMP program to the clusters of workstations needs source code modifications, we have managed to make the changes as minimal as possible by using Trellis-SDP. In fact, the only change required is the way the original application outputs the results. The OpenMP program writes the local models to a large shared-memory region, and performs the global sum at the same time; the modified program (the Trellis-SDP version) first generates the local models to files, then all the local models are merged by the global reduction.

We have verified that the trellis version of the 3D LSAVA migration application produces the correct results for both small and large data sets, and have also demonstrated that it has good scalability.

5.6 Concluding Remarks

In this chapter, we have given the empirical evaluations of our Trellis-SDP system. We studied four applications: distributed `grep`, content-based image retrieval, Parallel Sorting by Regular Sampling, and seismic data processing by 3D LSAVA migration. The results we obtained are summarized as follows:

1. Trellis-SDP can work across administrative domains (Section 5.2).

2. For simple data-parallel applications with coarse-grained communication patterns, Trellis-SDP does not introduce excessive overhead and the application shows good scalability (Section 5.3).

3. Trellis-SDP works for parallel applications with all-to-all collective-communication phases. However, the collective-communication phases may become a performance bottleneck in the applications (Section 5.4).

4. Trellis-SDP works for applications that need to process very large data sets. Due to limited resources, we were not able to scale the application beyond 8 worker hosts (Section 5.5).

Chapter 6

Conclusion

In this thesis, we presented the design and implementation of Trellis-SDP, a simple data-parallel programming system. One major contribution of this work is that Trellis-SDP enables fast development of data-intensive and naturally data-parallel applications in a metacomputing environment. Another major contribution of Trellis-SDP is that it allows worker processes to run existing, unmodified binaries. Trellis-SDP also uses the metadata file to represent files or directories that are distributed across the wide-area network.

Trellis-SDP is built upon the existing Trellis project and provides a Master-Worker programming framework. Function shipping and remote execution strategies are adopted to move the executables to the worker hosts where the data resides. To integrate the metadata file into our programming system, we introduced the file-level data parallelism and file-level collective-communication concepts, enabling the data-parallel and collective-communication operations to be performed at the file level, instead of at the memory level. This offers several advantages, such as facilitating a batched-pipelined execution model and requiring less-strict synchronization of parallel processes, especially in a metacomputing environment.

We discussed all the major application programming interfaces that we proposed and implemented, and gave detailed examples of how these interfaces can be used to write non-trivial data-parallel applications. We demonstrated that if the problem itself is simple, the implementation can also be simple.

We evaluated Trellis-SDP using four applications: the distributed `trellis grep` application demonstrates that Trellis-SDP works over a wide-area network; the content-based image retrieval application demonstrates that for data-parallel application without collective-communication phases, the overhead introduced by Trellis-SDP is minimal, and the application shows good scalability; the Parallel Sorting by Regular Sampling application and the 3D seismic data processing application demonstrate that Trellis-SDP works for applications with collective-communication phases, and that Trellis-SDP is reliable when the application is required to process very large data sets.

Future research directions may include the investigation of other data-intensive applications to further improve the programming system with regard to simplicity and efficiency; the design of abstractions (e.g., metafiles); the implementation of library functions (e.g., `trellis_scan()`, `trellis_gather()` and `trellis_reduce()`), and the evaluation of techniques to create data-parallel applications.

Bibliography

[1] A. Acharya, M.Uysal, and J.Saltz. Active Disks: Programming Model, Algorithm and Evaluation. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 81–91, San Jose, California, United States, 1998.

[2] M. Banikazemi, V. Moorthy, and D. Panda. Efficient Collective Communication on Heterogeneous Networks of Workstations. In *Proceedings of International Conference on Parallel Processing*, pages 460–467, Minneaplis, Minnesota, USA, August 1998.

[3] D.J. Barrett and R.E. Silverman. *SSH, the Secure Shell: The Definitive Guide, 2nd Edition*. O'Reilly and Associates, 2005.

[4] M. Bernaschi and G. Iannello. Collective Communication Operations: Experimental Results vs. Theory. *Concurrency: Practice and Experience*, 10(5):359–386, April 1998.

[5] S. Berretti, A. Del Bimbo, and P. Pala. Collection fusion for distributed image retrieval. In *Proceedings of ACM SIGIR Workshop on Distributed Information Retrieval*, pages 70–83, Toronto, Canada, August 2003.

[6] J. F. Claerbout and J. L. Black. Basic earth imaging, version 2.4. `http://sepwww.stanford.edu/sep/prof/toc_html/bei/toc_html/index.html`.

[7] M. Closson. The Trellis Network File System. Master's thesis, Department of Computing Science, University of Alberta, 2004.

[8] Condor. `http://www.cs.wisc.edu/condor`.

[9] SSH Communication Security Corp. Enabling Virtual Private Networks with Public Key Infrastructure, 2004. `http://www.ssh.com`.

[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 137–150, San Francisco, California, USA, December 2004.

[11] C. Lee et al. A Grid Programing Primer, August 2001. Advanced Programming Models Working Group, Global Grid Forum, `http://www.gridforum.org/`.

[12] C. Pinchak et al. The Canadian Internetworked Scientific Supercomputer. In *Proceedings of 17th Annual International Symposium on High Performance Computing Systems and Applications (HPCS)*, pages 193–199, Sherbrooke, Quebec, Canada, May 11–14, 2003.

[13] D. Thain et al. The Architectural Implications of Pipeline and Batch Sharing in Scientific Workloads. Technical Report UW-CS-TR-1463, Computer Sciences Department, University of Wisconsin, United States, January 2003.

[14] I. Foster et al. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, 2002. Open Grid Service Infrastructure WG, Global Grid Forum, `http://www.globus.org/`.

[15] J. P. Goux et al. An Enabling Framework for Master-Worker Applications on the Computational Grid. In *Proceedings of 9th International Symposium on High Performance Distributed Computing*, pages 43–50, Pittsburgh, Pennsylvania, United States, August 2000.

[16] K. Seymour et al. GridRPC: A remote procedure call API for grid computing. https://forge.gridforum.org/projects/gridrpc-wg/.

[17] M. Beynon et al. Design of a Framework for Data-Intensive Wide-Area Applications. In *Heterogeneous Computing Workshop*, pages 116–130, Cancun, Mexico, 2000.

[18] M. Stonebraker et al. A Wide-Area Distributed Database System. *Very Large Data Bases (VLDB)*, 5(1):48–63, 1996.

[19] R. Butler et al. A National-Scale Authentication Infrastructure. *IEEE Computer*, 33(12):60–66, 2000.

[20] T. Kielmann et al. MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems. *ACM SIGPLAN Notices*, 34(8):131–140, 1999.

[21] X. Li et al. On the Versatility of Parallel Sorting by Regular Sampling. *Parallel Computing*, 19(10):1079–1103, 1993.

[22] J. Foote. An Overview of Audio Information Retrieval. *Multimedia Systems*, 7(1):2–10, 1999.

[23] I. Foster. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1998.

[24] I. Foster and N. Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In *Proceedings of Supercomputing '98*. ACM Press, Orlando, Florida, United States, 1998.

[25] Globus. `http://www.globus.org/`.

[26] M. Kan, D. Ngo, M. Lee, P. Lu, N. Bard, M. Closson, M. Ding, M. Goldenberg, N. Lamb, R. Senda, E. Sumbar, and Y. Wang. The Trellis Security Infrastructure: A Layered Approach to Overlay Metacomputers. In *18th International Symposium on High Performance Computing Systems and Applications (HPCS)*, pages 109–117, Winnipeg, Manitoba, Canada, May 16–19, 2004.

[27] N. T. Karonis. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5):551–563, May 2003.

[28] Message Passing Interface Standard 1.1. `http://www-unix.mcs.anl.gov/mpi/`.

[29] C. Pinchak. Placeholder Scheduling for Overlay Metacomputing. Master's thesis, Department of Computing Science, University of Alberta, 2003.

[30] E. Riedel and G. Gibson. Active Disks - Remote Execution for Network-Attached Storage. Technical Report CMS-CS-99-177, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, United States, November 1999.

[31] Y. Rui, T. S. Huang, and S. Chang. Image Retrieval: Past, Present, and Future. In *Proceedings of International Symposium on Multimedia Information Processing*, pages 211–220, Taipei, Taiwan, December 1997.

[32] Seismic Unix (CWP/SU). `http://www.cwp.mines.edu/cwpcodes`.

[33] K. C. Sevcik. Application Scheduling and Processor Allocation in Multiprogrammed Parallel Processing Systems. *Journal of Performance Evaluation*, 19:107–140, 1994.

[34] G. Shao. *Adaptive Scheduling of Master/Worker Applications on Distributed Computational Resources*. PhD thesis, Department of Computer Science, University of California at San Diego, 2001.

[35] J. Siegel and P. Lu. User-Level Remote Data Access in Overlay Metacomputers. In *Proceedings of the 4th IEEE International Conference on Cluster Computing*, pages 480–483, September 2002.

[36] J. Wang, H. Kuehl, and M. D.Sacchi. Least-squares wave-equation avp imaging of 3D common azimuth data. In *Proceedings of the 73rd Annual International Meeting, Society of Exploration Geophysicists*, Dallas, USA, October 2003.

[37] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, 2nd Edition*. Prentice Hall, 2005.

Appendix A

Preliminary Study of Application Profiling and Performance Prediction

In this thesis, our programming system assumes static data distribution and resource allocation. This means that the selection of master and worker hosts are determined before the computation starts, and there are no data movements involved. In practice, if we want to run an application across a wide-area network, a scheduler is almost always needed to ensure that limited resources are allocated to different workloads or jobs in a reasonable fashion. In this chapter, we briefly discuss our preliminary effort on the application profiling model and performance prediction model, based on Trellis-SDP, for our future scheduler.

Scheduling in a metacomputing environment is challenging. The ability to make good scheduling decisions relies largely on the amount and accuracy of the system and the application information. However, the dynamically-changing system information and the detailed application information may not be readily available, or may be too expensive to collect.

We assume that the research on the scheduling of data-parallel applications is a practical starting point because data-parallel applications are simple, and have coarse-grained communication patterns. More specifically, we focus on the load balancing problem for data-parallel applications written in Trellis-SDP. Figure A.1 illustrates the control flow for the load balancing of a Trellis-SDP application. First, the application and system profiling data from previous runs are collected and analyzed. Then, a performance prediction model is built upon the profiling information, and provided to the scheduler. The scheduler, aware of the current system status, obtains the estimated execution time from the performance prediction model and, finally, generates the best scheduling scheme.

A.1 Application Profiling

Since Trellis-SDP works at the file level, and Trellis-SDP applications are written in data-parallel and collective-communication phases, we can naturally extend our system to integrate application profiling capabilities at a coarse-grained level. Currently, we are interested in the number of phases in

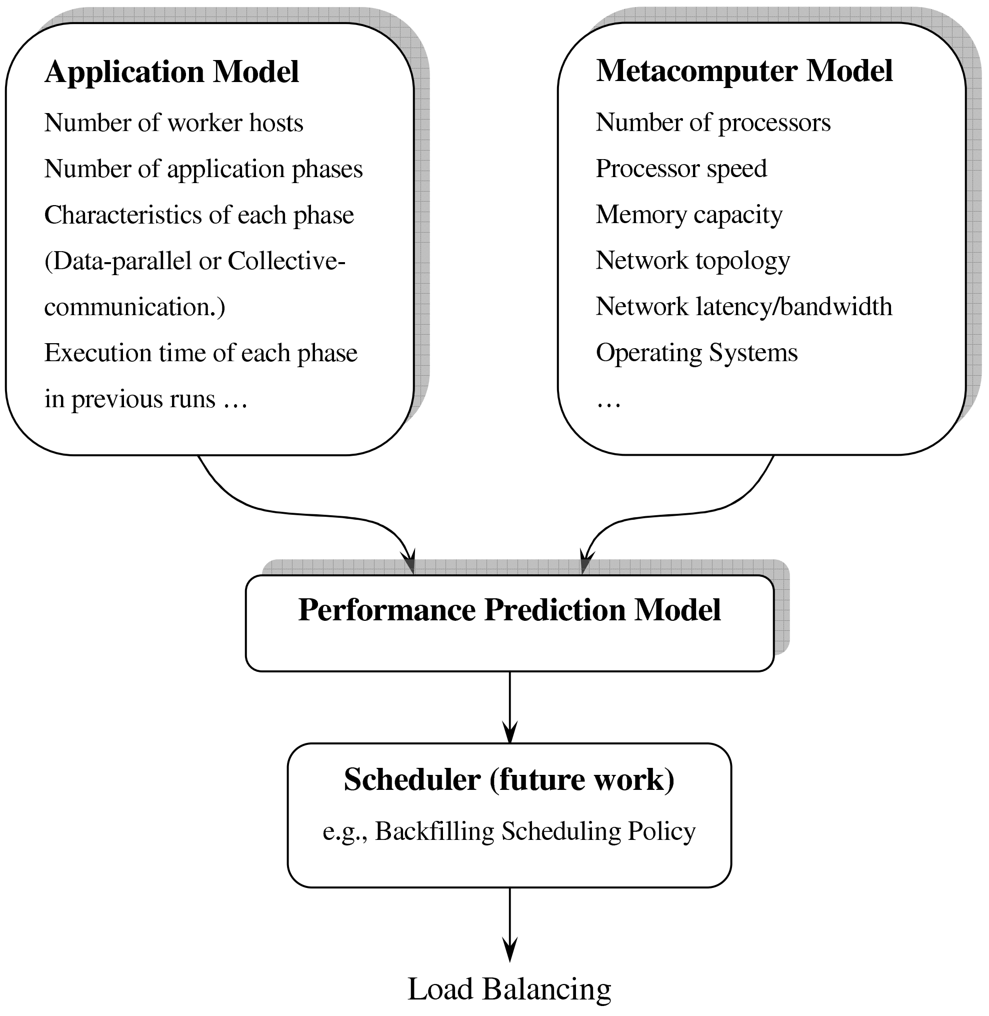


Figure A.1: The control flow for the load balancing of Trellis-SDP applications

the application, the characteristics of each phase (data-parallel phases or collective-communication phases), the execution time of each phase, and the number of worker hosts. Our programming system maintains a static profiling object. It provides two functions:

```
void Profile::registerPhaseStart(struct timeval start_time,
                                 const char * phase);

void Profile::registerPhaseEnd(struct timeval end_time);
```

These two functions are called at the beginning and the end of each phase, respectively, if the user turns on the profiling capability of the programming system. In this way, the number and types of phases, as well as the time spent in each phase, are recorded during each run of the application. This information will be analyzed and fed into the performance prediction model.

A.2 Performance Prediction and Scheduling

A.2.1 Introduction

The performance prediction model is a metric used to predict the expected application performance at future runs. The performance model uses the application profiling data and system configuration data gathered from previous runs to make the estimation. Currently, we are experimenting with a simple performance prediction model based on homogeneous architectures.

Our performance prediction model may be useful if the scheduler is adopting a backfilling scheduling policy where there might be some "holes" in the job waiting queues (Figure A.2). If the scheduler predicts that the run-time of a later submitted application can be fit into the time slot of the "holes", that application can be scheduled earlier.

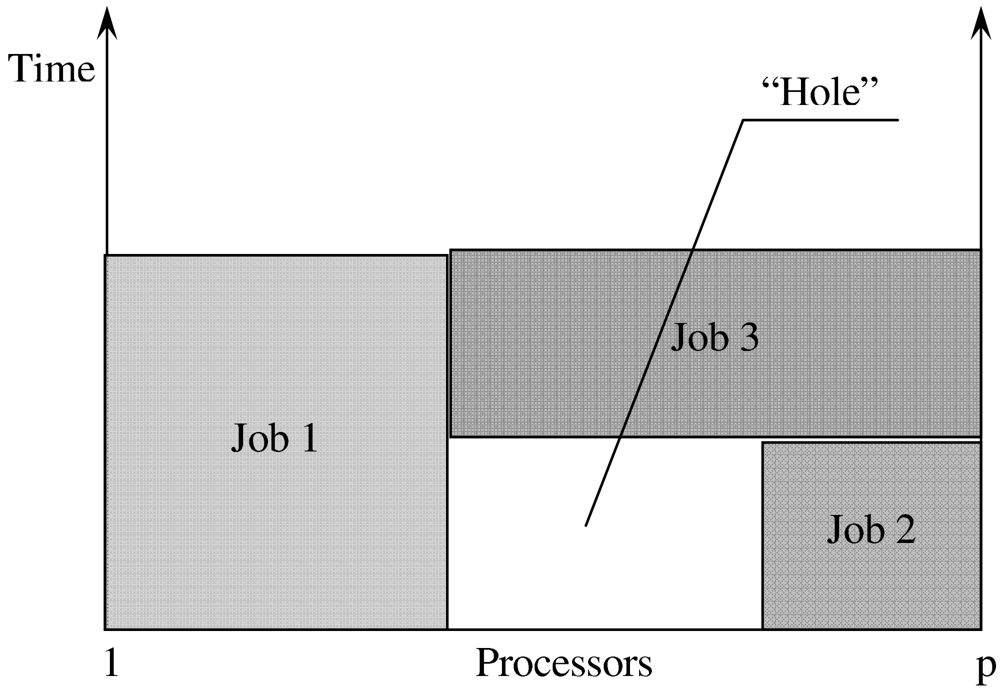


Figure A.2: The backfilling scheduling policy. There might be "holes" in the job waiting queues.

A.2.2 Prediction of the Execution Time

The model we used for execution time prediction is taken from Sevcik [33]. This model takes into account the following major issues:

1. The essential computational work of the application (e.g., the sequential running time of the application).

2. The imbalance with which the essential work is distributed across the processors.

3. The overhead introduced for parallel processing (e.g., the overhead of the programming system/framework).

4. The communication and congestion delays.

The original model is defined as:

$$T_j(p) = \phi_j(p)\frac{W_j}{p} + \alpha_j(p) + \beta_j(p)$$

where W represents the sequential computational work of the application, p is the number of processors, $\phi(p)$ represents the unevenness among the work distributed across p processors, α represents the overhead introduced by the parallel processing, and $\beta(p)$ represents the communication and congestion delays (a function of p).

In our study, we assume that the data is always evenly distributed across processes and we consider only homogeneous architecture, so $\phi(p)$ equals to 1. We further simplify the model by removing the factors for communication and congestion delays, since we assume data-intensive operations of the application are performed locally by worker processes, and only a small amount of data needs to be transferred over the network. The final model we use is:

$$T(p) = \frac{W}{p} + N(1 + 0.2p)$$

where W is still the sequential computational work of the application and p is the number of processors. $N(1+0.2p)$ is our version of the overhead introduced by parallel processing, determined by the historical profiling of the application, where N is the number of phases in the application and $(1 + 0.2p)$ is the average overhead for each phase (basically `ssh` startup overheads, which has been discussed in Chapter 5).

A.3 Preliminary Evaluation Results

In the previous section, we presented our simple performance prediction model for homogeneous architectures by the following equation:

$$T(p) = \frac{W}{p} + N(1 + 0.2p)$$

We evaluate this equation by the CBIR application described in Section 5.3. The experiment is performed in the LAN environment with 2, 4 and 8 worker hosts. We measure the actual execution time of the application running on a given number of worker hosts, and use the equation to derive the predicted execution time of the application running on different number of worker hosts. The results are shown in Table A.1.

From the table, we can see that the worst-case prediction error is less than 5%, which indicates that our performance prediction model is relatively accurate and practical for simple data-parallel applications.

2 worker hosts			4 worker hosts			8 worker hosts		
actual (s)	predicted (s)	error	actual (s)	predicted (s)	error	actual	predicted	error
55.78	–	–	28.5	28.99	1.7%	15.45	16.1	4.2%
55.78	54.8	1.8%	28.5	–	–	15.45	15.9	2.9%
55.78	53.5	4.1%	28.5	27.6	3.2%	15.45	–	–

Table A.1: The prediction of the execution time of the CBIR application on different numbers of worker hosts.

Appendix B

Source Code for CBIR Application

```
#define DISABLE_MACRO_REPLACEMENT

#include <string>
#include <trellis.h>
#include <trellis_sdp.h>
#include <iostream>
#include <stdlib.h>
#include <assert.h>

struct top{
      float distance;
      int index;
} * toplist = NULL;

int main(int argc, char * argv[] ){

      Trellis_Request request;
      Trellis_Status status;

      char * sample_disk = argv[1];
      char * sample_offset = argv[2];
      char * top_n = argv[3];
      char * metafile = argv[4];
      int items_read = 0;
      string op;
      float * buffer;

      op += "query " + string(sample_disk) + " " + string(sample_offset)
            + " " + string(top_n);

      trellis_init(argc, argv);

      if(trellis_scan(metafile, op.c_str(), &request) < 0){
            fprintf(stderr, "Scan Failed\n");
            exit(-1);
      }else{
            memset(buffer,0, BUF_SIZE);
            items_read = trellis_scan_readall(&buffer, Trellis_FLOAT,
                         request);
            trellis_scan_wait(request, status);

            assert(items_read > 0);
            toplist = (struct top *)malloc(items_read / 2 *
                      sizeof(struct top));

            for(int i = 0; i < items_read/2; i++){
                  toplist[i].distance = buffer[i*2];
                  toplist[i].index = (int)(buffer[i*2+1]);
            }

            quicksort(toplist, 0, items_read/2 - 1);
            for(int i = 0; i < items_read/2; i++)
                  printf("%f %d\n", toplist[i].distance,
                          toplist[i].index);
      }

      trellis_finalize();

      if(toplist!=NULL){
            free(toplist);
            toplist = NULL;
      }

      return 0;
}
```

Appendix C

Source Code for PSRS Application

```
#define DISABLE_MACRO_REPLACEMENT

#include <trellis.h>
#include <trellis_sdp.h>
#include <string>
#include <iostream>
#include <stdlib.h>
#include <unistd.h>
#include "quicksort.h"

using namespace std;

#define PTRACE_PATH "ptrace"
#define LOCALSORT_PATH "/usr/scratch/ading/PSRS/localsort"
#define GSAMPLE_PATH "/usr/scratch/ading/PSRS/gathersample"
#define COLLECTIDX_PATH "/usr/scratch/ading/PSRS/collectidx"
#define MERGESORT_PATH "/usr/scratch/ading/PSRS/mergesort"

int DISK_NUM;
int SUBARRAY_SIZE;
int SAMPLE_EACH;
int INDEX_PIVOTS;

int * phase1(const char * metafile_in, const char * metafile_localsorted){

      char sample_each_string[1024], subarray_num_string[1024];
      int * sample_array = NULL;
      int items_read;
      string op1, op2;
      Trellis_Request request;
      Trellis_Status status;

      sprintf(sample_each_string, "%d", SAMPLE_EACH);
      sprintf(subarray_num_string, "%d", SUBARRAY_SIZE);
      op1 = string(LOCALSORT_PATH) + " " + string(subarray_num_string);
      op2 = string(GSAMPLE_PATH) + " " + string(subarray_num_string)+ " "
                                          + string(sample_each_string);

      if(trellis_scan(metafile_in, op1, &request) < 0){
            fprintf(stderr, "Scan Failed\n");
            exit(-1);
      }
```

```
	trellis_scan_wait(request, status);

	if(trellis_scan(metafile_localsorted, op2, &request) < 0){
		fprintf(stderr, "Scan Failed\n");
		exit(-1);
	}
	sample_array = (int *)malloc(DISK_NUM * SAMPLE_EACH * sizeof(int));
	items_read = trellis_scan_read(sample_array, Trellis_INT,
	                               DISK_NUM * SAMPLE_EACH, request);
	trellis_scan_wait(request, status);

	return sample_array;
}

int ** collect_index_info(const char * metafile_localsorted,
                                                string pivots_file_name){

	char disk_num_string[1024], subarray_num_string[1024];
	int ** index_table = NULL;
	string op;
	Trellis_Request request;
	Trellis_Status status;
	int items_received;

	sprintf(disk_num_string, "%d", DISK_NUM);
	sprintf(subarray_num_string, "%d", SUBARRAY_SIZE);

	op = string(PTRACE_PATH) + " " + string(COLLECTIDX_PATH) + " " +
	       string(subarray_num_string) + " " + string(disk_num_string)
	         + " " + pivots_file_name;

	if(trellis_scan(metafile_localsorted, op, &request) < 0){
		fprintf(stderr, "Scan Failed\n");
		exit(-1);
	}

	index_table = (int **)malloc(scan->GetDisknum() * sizeof(int *));
	for(int key = 0; key < scan->GetDisknum(); key++){
		index_table[key] = (int *)malloc(scan->GetDisknum() * 2 *
		                     sizeof(int));
		items_received = trellis_scan_readidx(index_table[key],
		                 Trellis_INT, scan->GetDisknum() * 2, key,
		                 request);
		if(items_received < 0){
			fprintf(stderr, "Scan Failed\n");
			exit(-1);
		}
	}

	trellis_scan_wait(request, status);
	return index_table;
}

int ** phase2(const char * metafile_localsorted, int * sample_array){

	int * pivots_array = NULL;
	int i, j;
	int ** index_table;
	FILE * pivots_file_fp;
	string pivots_file_name;
	string pivots_file_scl;

	quicksort(sample_array, 0, DISK_NUM * SAMPLE_EACH - 1);
	pivots_array = (int *)malloc((DISK_NUM-1)*sizeof(int));

	for(i = 1; i < DISK_NUM; i++){
		j = i * DISK_NUM + INDEX_PIVOTS - 1;
		pivots_array[i-1] = sample_array[j];
	}

	pivots_file_name = string(getenv("HOME")) + "/pivots_file";
	pivots_file_fp = fopen(pivots_file_name.c_str(), "w");
	if(pivots_file_fp == NULL){
		fprintf(stderr, "Open Pivots File Error\n");
		exit(-1);
	}
	for(i = 0; i < DISK_NUM - 1; i++)
		fprintf(pivots_file_fp, "%d\n", pivots_array[i]);
	fclose(pivots_file_fp);

	pivots_file_scl = "scp:"+string(getenv("HOST"))+":pivots_file";
	index_table = collect_index_info(metafile_localsorted,
			pivots_file_scl);
	free(pivots_array);

	return index_table;
}


int phase3(const char * metafile_localsorted, const char *
	metafile_gather, int ** index_table, Trellis_Datatype datatype){

	return trellis_gather(metafile_localsorted, metafile_gather,
					index_table, datatype);
}

void phase4(const char * metafile_gather, const char *
					metafile_globalsorted){

	string op;
	Trellis_Request request;
	Trellis_Status status;

	op = string(MERGESORT_PATH);
	if(trellis_scan(metafile_gather, op, &request) <0 ){
		fprintf(stderr, "Scan Failed\n");
		exit(-1);
	}
	trellis_scan_wait(request, status);

	return;
}

int main(int argc, char * argv[] ){

	int * sample_array = NULL;
	int ** index_table;
```

```
      char metafile_in[] = "tosort.meta";
      char metafile_localsorted[] = "localsorted.meta";
      char metafile_gather[] = "sortgather.meta";
      char metafile_globalsorted[] = "sorted.meta";

      int size;

      DISK_NUM = 8;
      SUBARRAY_SIZE = 33554400;
      SAMPLE_EACH = DISK_NUM;
      INDEX_PIVOTS = DISK_NUM/2;

      trellis_init(argc, argv);

      sample_array = phase1(metafile_in, metafile_localsorted);
      index_table = phase2(metafile_localsorted, sample_array);
      phase3(metafile_localsorted, metafile_gather, index_table,
             Trellis_INT);
      phase4(metafile_gather, metafile_globalsorted);

      for(int i = 0; i < DISK_NUM; i++)
            free(index_table[i]);
      free(index_table);
      free(sample_array);

      trellis_finalize();

      return 0;

}
```

Appendix D

Source Code for 3D_LSAVA Migration Application

```
#define DISABLE_MACRO_REPLACEMENT

#include <string>
#include <string.h>
#include <stdio.h>
#include <trellis.h>
#include <trellis_sdp.h>

int main(int argc, char * argv[]){

      Trellis_Request request;
      Trellis_Status status;

      char * metafile_scan = argv[1];
      char * scan_op = "/usr/scratch/ading/SEISMIC/LS_AVA ";
      char * reduce_op = "susum";
      char * metafile_input = argv[2];
      char * host_num = argv[3];
      char scan_command[1024];
      char metafile_output[] = "LSAVA.data/LSAVA_OUTPUT.meta";

      trellis_init(argc, argv);

      /* Trellis Scan */

      memset(scan_command, 0, 1024);
      strcat(scan_command, scan_op);
      strcat(scan_command, host_num);
      if(trellis_scan(metafile_scan, scan_command, &request) <0 ){
            fprintf(stderr, "Scan Failed\n");
            exit(-1);
      }

      trellis_scan_wait(request, status);

      /* Trellis Reduce */

      if(trellis_reduce1(metafile_input, metafile_output, reduce_op)<0){
            fprintf(stderr, "Reduce Failed\n");
            exit(-1);
```

```
    }

    trellis_finalize();
    return 0;
}
```