Trellis-SDP: A Simple Data-Parallel Programming Interface

Meng Ding and Paul Lu
Department of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8
Canada
{ading|paullu}@cs.ualberta.ca

Abstract

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

We have designed and implemented a simple programming system, called Trellis-SDP, 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 provides a master-worker programming framework where the worker components can run self-contained, new or existing *binary applications. We describe two interface functions, namely trellis_scan() and trellis_gather(), and show how easy it is to get reasonable performance with simple data-parallel applications, such as Content Based Image Retrieval (CBIR) and Parallel Sorting by Regular Sampling (PSRS).*

1. Introduction

Data parallelism is a well-known programming model. However, it can be difficult to write and deploy a simple data-parallel application, which is unfortunate because many problems are naturally data parallel. For example, information retrieval, sorting, and searching have inherent data parallel phases. Parallelizing an existing application in these areas may require porting (e.g., to use a message-passing system) and access to source code. However, message-passing can be complicated and the applications may be in binary-only form. Similarly, hardware developments in workstations and networks have encouraged distributed computing platforms, such as clusters, metacomputers over wide-area networks (WAN) [20], and grids [9, 11]. With the prevalence of inherently data-parallel applications and platforms, there is a need for a simple data-parallel programming system for simple data-parallel problems.

Suppose a company is providing a service for content based music retrieval, which takes a clip of singing 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 [8] and comparison algorithms on a single site. Instead, one can choose to ship the music feature extraction and comparison functions to the sites where data resides and perform the operations there. This function shipping and remote execution mechanism not only makes full use of the computational power on each site, but it also greatly reduces the traffic over the WAN. However, the problem is, how to support the easy and efficient programming of these applications that handle large collections of distributed datasets?

We have designed, implemented, and performed a preliminary evaluation of a programming system, Trellis-SDP, for data-intensive applications. Trellis-SDP is designed to work across WANs, and we have successfully run Trellis-SDP jobs across multiple administrative domains, but our empirical evaluation in this paper is limited to the controlled environment of a network of workstations. Nonetheless, the design and implementation of Trellis-SDP addresses several important issues in metacomputing programming [16]:

Security Trellis-SDP is based on the previous work of the Trellis Project [20, 19, 23]: a software infrastructure for user-level overlay metacomputers. Our programming system takes advantage of the underlying Trellis Security Infrastructure (TSI) [13], which is layered on the Secure Shell (SSH) [7, 3], for authentication and secure communication across different administrative domains.

Resource Specification In Trellis-SDP, a metafile is a file that is logically contiguous, but (perhaps) physically

distributed across a network. An XML-based metadata file is used to describe a metafile (e.g., Figure 2), including the location and the size of the distributed blocks.

Global Naming and Remote Data Access We use Secure Copy Locators (SCL) [23] as the filenames in the global namespace. Using SCL, Trellis can access remote data by first copying it into 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.

Usability One of our key design philosophies is to make the programming system as simple as possible while maintaining the basic metacomputing functionality.

We present and evaluate two basic programming interfaces in Trellis-SDP, namely *trellis_scan()* and *trellis_gather()*, which allow remote execution and group communication in WAN. As is, Trellis-SDP is not feature-rich enough to support all data-parallel applications, but it is capable of implementing a large class of applications with minimal effort.

The main contributions of our work are:

1. Trellis-SDP provides a **simple** master-worker programming framework (Figure 1) that facilitates the rapid development of data-intensive applications. With a metadata file (Figure 2) representing the naturally-distributed data, one can easily write a non-trivial data-parallel application using Trellis-SDP (Figure 3).

2. 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 applications).

2. Related Work

The need for reducing the complexity of data-parallel programming has led to a great deal of work in application-specific toolkits. These include application or domain-specific languages and libraries, programming frameworks and problem-solving environments [9]. Most of these toolkits have been adapted from traditional parallel and distributed computing systems; only a few are designed specifically for grid computing or metacomputing [16]. Existing programming tools include message-passing libraries, object-oriented tools, and middleware systems:

MPICH-G2 [14] is a "grid-enabled implementation of the Message Passing Interface (MPI) that allows the programmer to run MPI programs across administrative domains using the same commands that would be used on a cluster of workstations" [10]. The significant advantage of MPICH-G2 is that the programmer can reuse existing MPI code. A pragmatic disadvantage is that MPICH-G2 requires the Globus toolkit to be installed in all the administrative domains, to address the issues of security, remote process startup, and cross-domain communications.

DataCutter [5] proposes a *filter-stream* programming model (originally designed for Active Disks [1, 21]) in a grid environment. In this programming model, an application is decomposed into a set of *filters* among which the communication is done via *streams*. As with Trellis-SDP, DataCutter also pushes the computation to the data, instead of migrating the data to the computation [5]. DataCutter does not have the concept of a metadata file. All filter *placements* must be specified in the program. Also, the filter does not support unmodified binaries. In other words, programmers must rewrite their data-intensive components according to the filter specifications.

MW [12] is a software framework that allows users to parallelize scientific applications on a computational grid using the master-worker programming model. MW provides two sets of programming interfaces: An *Infrastructure Programming Interface* that ports the MW framework to a grid software toolkit such as Condor [6] and 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.

3. The Trellis-SDP Approach

The main design goal of Trellis-SDP is to facilitate the programming of data-intensive applications with coarse-grained and simple communication patterns. For more fine-grained and complicated message patterns, we also support group communication. Of course, the overall performance depends on the amount and type of communication in the application, but Trellis-SDP is designed to be easy-to-use for easy-to-parallelize applications.

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.

If necessary, the programmer is responsible for identifying the I/O-intensive cores in an application and extracting them into stand-alone components. We try to simplify this

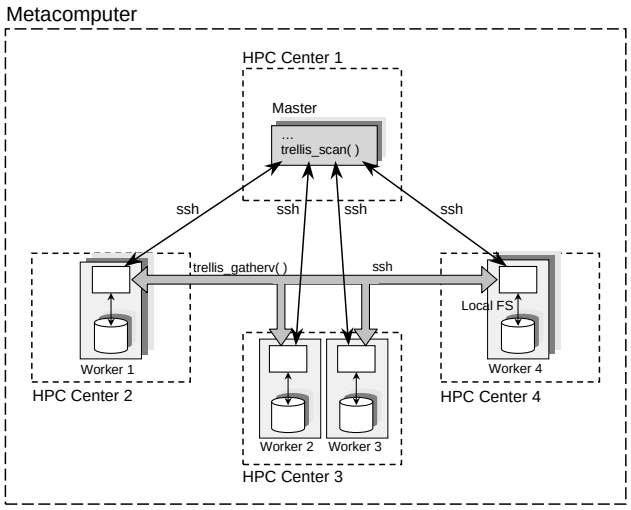

Figure 1. Programming System Execution Environment

process by allowing worker components to run arbitrary executables. For example, for a distributed sorting application, the programmer can choose to reuse an existing sequential sorting application as the worker component.

Before the computation is started, Trellis-SDP assumes:

1. The data needed by the computation is already distributed across the metacomputer, which is a common case for wide-area data-intensive applications (Figure 1). Trellis-SDP provides basic tools for scattering and gathering the data; the tools are used in our testing.

2. The executable code for the worker components is already distributed across the metacomputer.

3. The metadata file, identifying the distributed data, already exists (Section 4.1). In our case, the metadata file contains the location and size of the data on each participating host.

Figure 1 illustrates the execution environment of our programming system. Inside a Trellis-SDP program, a worker process is invoked by a call to the *trellis_scan()* library function. It takes an object, representing the metadata file, and the specified operation as input parameters. The worker components on remote hosts perform the operations and either generate the results on their local disks or return the results back to the master component via streams. In the former case, intermediate files generated by different worker components can also be described using a metadata file. This intermediate metadata file can be used in a different *trellis_scan()* or it can be saved to disk. This is especially useful in a batch-pipelined workload [24], where the output of one worker component may be the input of a succeeding worker component. Note in Figure 1 that, if necessary, a *trellis_gather()* can be used to perform group communication among worker components.

4. Implementation

4.1. Metafiles: A Metadata File Approach

As discussed earlier, a metafile is a file that is logically contiguous, but (perhaps) physically distributed across a network. As with other indexed-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. Given a contiguous file, Trellis-SDP provides a tool 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.

The metafile is written in XML, as is illustrated in Figure 2. Each block in the file 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 programmer creates an in-memory metadata object corresponding to a metadata file. This is analogous to an in-memory version (i.e., metadata object) of an 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. The object can then be passed to *trellis_scan()*. It is also possible to export a metadata object to disk, in XML format.

4.2. Trellis_Scan

trellis_scan() is the main data-parallel application programming interface (API) in Trellis-SDP. The declaration of *trellis_scan()* is:

```
int trellis_scan(MetaHandler* meta, char* operation,
ScanHandler** scan);
```

As shown in Figure 3, *trellis_scan()* is (typically) called in the master component and it takes two input parameters and one output parameter. For input, there is a handle to a metadata object and an operation string. For output, *trellis_scan()* will create a scan object and return a handle to it via the last parameter. The scan object stores the results from the different worker components. As well, the scan object provides functions to examine data streams from remote computing hosts in an arbitrary order so that the mas-

```
<?xml version="1.0"?>
<BlockList>
  <DataBlock>
    <Locator>scp:ading@cleardale.cs.ualberta.ca:/usr/scratch/feature.1</Locator>
    <Size>5945000</Size>
  </DataBlock>
  <DataBlock>
    <Locator>scp:ading@sullivan.cs.ualberta.ca:/usr/scratch/feature.2</Locator>
    <Size>4830000</Size>
  </DataBlock>
  <BlockSize>32</BlockSize>
</BlockList>
```

Figure 2. Example of a Metadata File

ter component can coordinate and merge these information for future processing.

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

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

     ScanHandler * scan = NULL;
     MetaHandler * meta = NULL;
     /* Result buffer */
     char buffer[1024];
     /* Argument for grep */
     char * grep_arg = argv[1];
     /* Metadata File */
     char * metafile = argv[2];
     /* Operations */
     string op = "grep " + string(grep_arg);

     /* Create Metadata Object */
     meta = new MetaHandler(metafile);
     /* Trellis Scan */
     if(trellis_scan(meta, op.c_str(), &scan)<0){
           fprintf(stderr, "Scan Failed\n");
           delete meta;
           exit(-1);
     }else{
           memset(buffer, 0, 1024);
           /* Read Result */
           scan->Read(buffer, 1024);
           printf("%s\n", buffer);
           delete scan;
           delete meta;
     }

     return 0;
}
```

Figure 3. Sample Code for `trellis grep` (master component). Worker components are Unix `grep` executables. Location and distribution of data is abstracted by the metadata file.

Figure 3 implements a data-parallel **trellis grep**, which is a grep operation on distributed data. 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()* reads in metadata information and starts up the worker component in each remote host to perform "grep" on its local data. The master component then reads in the results through the scan object. Note that **trellis grep** performs most of its data-intensive operations on the remote hosts and transfers only a small amount of data back to the master host.

The scan object also provides a synchronization operation that waits for the data transmission to complete in all streams. The synchronization is done automatically when the scan object is deallocated.

4.3. Trellis_Gather

As discussed, *trellis_scan()* establishes communication channels between the master and worker components. This interface is sufficient for embarrassingly data-parallel applications with no communications among worker components. However, some complex parallel and distributed applications do require group communications. Thus we also propose a group communication interface called *trellis_gather()*. This is quite similar to the MPI collective communication interface MPI_Gather/MPI_Gatherv [18]. There are several papers on collective communications in wide-area networks, including performance issues, fault tolerance issues, etc [2, 4, 15]. Our efforts focus mainly on the API issues at this time. We also touch upon a little bit of the performance issue and will discuss this in the Application section.

trellis_gather() has the following declaration:

int trellis_gather(MetaHandler meta, void* recvbuf, int* recvbytes, int* starts, int datatype, int root);*

The semantics of *trellis_gather()* are similar to those of MPI_Gather. The difference is that in MPI_Gather, the remote data resides in the memory of a remote host, but in *trellis_gather()*, the remote data is stored on disk and is specified by a metadata file. Note that the first parameter to *trellis_gather()* is a metadata object that has already been bound to the metadata file. Therefore, the input parameter *starts*, which is an integer array, represents the offsets relative to files instead of displacements relative to memory

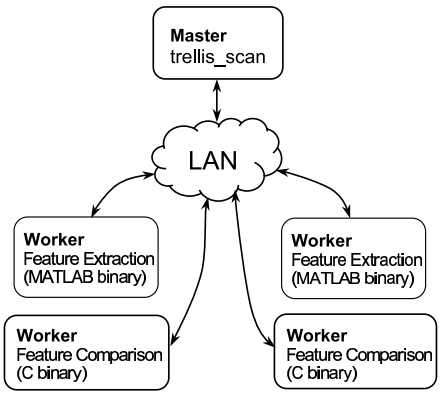

Figure 4. Control flow of CBIR application

buffers. To initiate a group communication, all participating worker components should call *trellis_gather()*. By default, data is transferred using `ssh`-protected channels and received in each worker component's *recvbuf*.

5. Applications

In this section, we describe two applications: Content Based Image Retrieval (CBIR) and Parallel Sorting by Regular Sampling (PSRS). We perform some initial benchmarks on the application and give a discussion on the preliminary performance results.

5.1. Content Based Image Retrieval

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, neuro-science, etc. [22]) have proposed various algorithms in this area.

5.1.1 Implementation

Writing a CBIR application using our programming system is quite similar to the CBMR example we 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. The feature vector is then searched through the feature space (i.e., the feature vectors of all the images in the image database) to find the top n most-matched feature vectors.

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 4 depicts the control flow of the distributed CBIR application. Note in the figure that the interconnection between the different components is show as being over a LAN (instead of a WAN), since all the experiments we performed are with a LAN. We will extend our experiments to WAN in the future.

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. And, we build the feature comparison component using standard C. In practice, one may choose to write the worker component using one's favorite language, to speed up the software development process.

5.1.2 Experimental Results

The experiment is carried out using a cluster of workstations with dual 1.5 GHz AMD Athlon CPUs, 1.5 GB RAM memory per node, and running the Linux kernel 2.4.18. All nodes are connected via Fast Ethernet. Our image database contains 60,000 images with a total feature space of 600MB.

Image Database Size	Sequential		2 Hosts		4 Hosts		8 Hosts	
	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

Figure 5. Raw execution time of CBIR application: Time is the average execution time (5 repeated runs) in seconds, σ is the standard deviation.

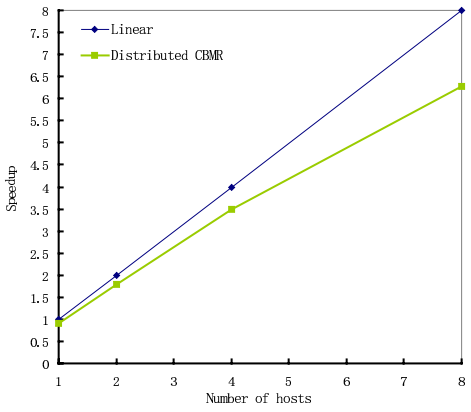

Figure 6. Speedup of distributed CBIR application. Image database size: 600MB

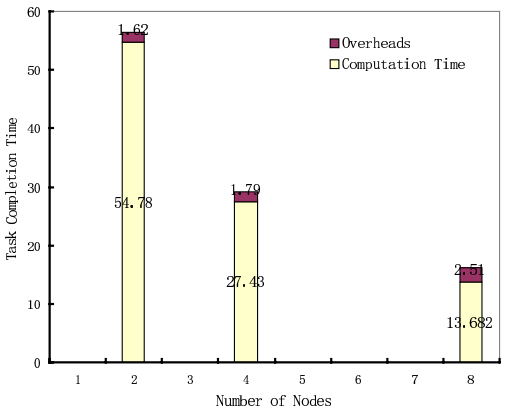

Figure 7. Overheads of the programming system in CBIR application

The main experiment we performed is the scalability test by distributing the image database onto different numbers of nodes. This is shown in Figure 5 where the average raw execution times of the CBIR application on 2, 4 and 8 hosts (excluding the master host) plus the sequential execution time are given. We also present a speedup graph to further illustrate the scalability of the application (see Figure 6).

The distributed CBIR application shows good scalability when the number of participating nodes increases. This is expected since the distributed CBIR is intrinsically embarrassingly parallel. The contribution of Trellis-SDP is in simplifying the implementation of the CBIR application 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 measured and factored out the `ssh` startup times, as compared to the overall execution time (Figure 7). 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, at least for CBIR, with the number of nodes. As shown with the next application, `ssh` startup overheads can become a bottleneck, especially when group communication is involved.

5.2. Parallel Sorting by Regular Sampling

Parallel Sorting by Regular Sampling(PSRS) [17] is a parallel sorting algorithm that is suitable for many parallel architectures. It has a "good load balancing properties, modest communication needs and good of locality references". To sort the data distributed on p hosts, the algorithm divides the whole process into **four phases**, which fits well with our remote execution 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 partition 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. The main purpose of the PSRS experiment is to show that our programming system works for non-embarrassingly parallel applications.

5.2.1 Implementation

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 component reads in pivots and generates 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.

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

Figure 8. Raw execution time of PSRS

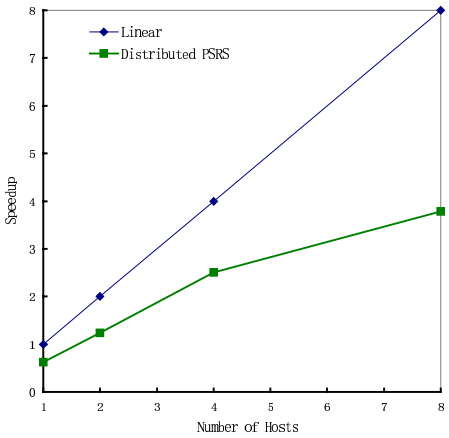

Figure 9. Speedup of PSRS application

5.2.2 Experiments and Discussions

The experiment setup is the same as the one described in Section 5.1.2 except that the dataset we use now contains 1

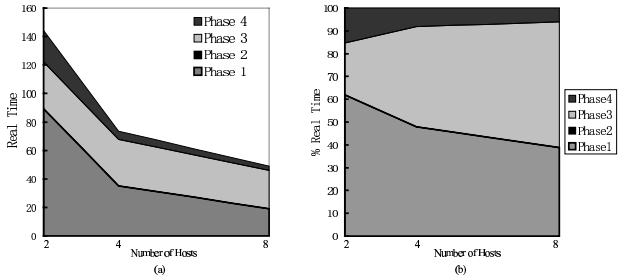

Figure 10. Breakdown execution time of PSRS application with default `ssh`. (a) phase-by-phase with real time. (b) phase-by-phase with percentage of real time.

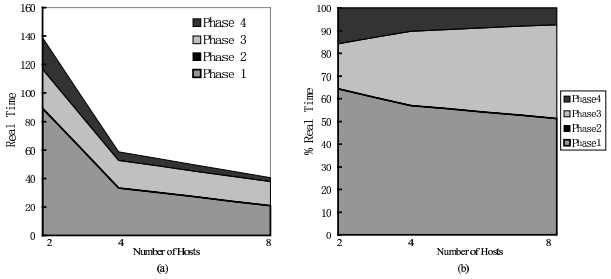

Figure 11. Breakdown execution time of PSRS application with `rsh` enabled in phase three. (a) phase-by-phase with real time. (b) phase-by-phase with percentage of real time.

GB of unsorted (binary) integers (i.e., 256 million keys) in total.

Scalability The raw execution time and speedup graphs of the distributed PSRS application are given in Figure 8 and Figure 9. The execution time is an average of 5 repeated runs. As seen from the figure, for 8 hosts, we get a speedup of 3.7. This is not "high" as compared with the previous CBIR experiment, but considering the all-to-all communications, the result is reasonable. In fact, we are more interested in identifying the overheads of Trellis-SDP for group communication.

Breakdown Execution Time We use phase-by-phase analysis to quantify the execution times in each phase. Figure 10 illustrates the breakdown of execution time of PSRS. As expected, phase three becomes a performance bottleneck when the number of hosts increases. For example, when there are only two hosts, phase three is 22% of the total execution time. But, when the number of hosts increases to eight, phase three grows to 55%.

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 communication among n worker components in phase three, there are $O(n^2)$ `ssh` connections.

To further quantify the overhead, we perform another test by replacing all `ssh` connections in phase three with `rsh`, which is faster than `ssh` since `rsh` uses cleartext channels. Figure 11 shows the new break down execution time of PSRS with `rsh` enabled in phase three. With `rsh`, both the total execution time, and percentage of time for phase three, is reduced. Figure 12 more-directly shows the impact of the choice of underlying communication mechanism. In the future, we plan to explore the communication optimization of `ssh` for large data transfers.

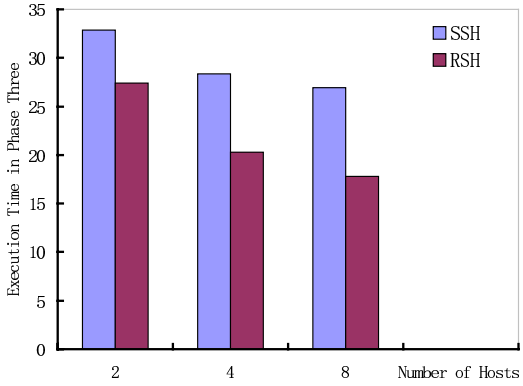

Figure 12. Overheads of the programming system in PSRS with different underlying communication mechanism

6. Concluding Remarks

We present the design and implementation of a simple programming system, called Trellis-SDP, that facilitates the rapid development of data-intensive applications in a user-level metacomputing environment. Trellis-SDP is built on top of the existing Trellis system and provides a master-worker programming framework where the worker components can run self-contained, purely sequential and existing (i.e., unmodified) binary applications.

Three data-intensive applications (i.e., grep, image retrieval, sorting) that make use of Trellis-SDP are described and the performance results for two applications are discussed. The results show that for naturally data-parallel applications, our programming system is easy to use and has reasonable performance, especially when considering the amount of algorithmically-necessary data communication. In the future, we will continue to investigate other data-intensive applications to further improve our system with

regard to simplicity and efficiency. Our on-going goals are to design abstractions (e.g., metafiles), provide library functions (e.g., *trellis_scan()* and *trellis_gather()*), and evaluate techniques to create data-parallel applications.

7. Acknowledgements

This research was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC), SGI, the Alberta Science and Research Authority (ASRA), and C3.ca.

References

[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 *International Conference on Parallel Processing*, pages 460–467, 1998.

[3] D. Barrett and R. Silverman. *SSH, the Secure Shell: The Definitive Guide*. O'Reilly and Associates, 2001.

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

[5] M. Beynon, T. M. Kurc, A. Sussman, and J. H. Saltz. Design of a Framework for Data-Intensive Wide-Area Applications. In *Heterogeneous Computing Workshop*, pages 116–130, 2000.

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

[7] S. C. S. Corp. Enabling Virtual Private Networks with Public Key Infrastructure, 2004. `http://www.ssh.com`.

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

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

[10] I. Foster and N. Karonis. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. In *Proceedings of SC'98*. ACM Press, 1998.

[11] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. 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/`.

[12] J.-P. Goux, S. Kulkarni, J. Linderoth, and M. Yoder. An Enabling Framework for Master-Worker Applications on the Computational Grid. In *Proc. 9th Int'l Symposium on High Performance Distributed Computing (HPDC-9)*, pages 43–50, 2000.

[13] 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.

[14] 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.

[15] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems. *ACM SIGPLAN Notices*, 34(8):131–140, 1999.

[16] C. Lee, S. Matsuoka, D. Talia, A. Sussman, M. Mueller, G. Allen, and J. Saltz. A Grid Programming Primer, August 2001. Advanced Programming Models Working Group, Global Grid Forum, `http://www.gridforum.org/`.

[17] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. On the Versatility of Parallel Sorting by Regular Sampling. *Parallel Computing*, 19(10):1079–1103, 1993.

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

[19] C. Pinchak, P. Lu, and M. Goldenberg. Practical Heterogeneous Placeholder Scheduling in Overlay Metacomputers: Early Experiences. In *Proc. 8th Workshop on Job Scheduling Strategies for Parallel Processing*, Edinburgh, Scotland, UK, July 2002.

[20] C. Pinchak, P. Lu, J. Schaeffer, and M. Goldenberg. The Canadian Internetworked Scientific Supercomputer. In *17th Annual International Symposium on High Performance Computing Systems and Applications (HPCS)*, Sherbrooke, Quebec, Canada, May 11–14, 2003. `http://www.cs.ualberta.ca/~ciss`, `http://www.cs.ualberta.ca/~paullu`.

[21] 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.

[22] Y. Rui, T. S. Huang, and S.-F. Chang. Image Retrieval: Past, Present, and Future. In *International Symposium on Multimedia Information Processing*, December 1997.

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

[24] D. Thain, J. Bent, R. Arpaci-Dusseau, A. Arpaci-Dusseau, and M. Livny. The Architectural Implications of Pipeline and Batch Sharing in Scientific Workloads. Technical Report UW-CS-TR-1463, Computer Sciences Department, University of Wisconson, January 2003.