University of Alberta

Library Release Form

Name of Author: Nicholas Peter David Lamb

Title of Thesis: Data-Conscious Scheduling of Workflows in Metacomputers

Degree: Master of Science

Year this Degree Granted: 2005

Nicholas Peter David Lamb
221 Athabasca Hall
University of Alberta
Edmonton, Alberta
Canada, T6G 2X5

Date: _____

University of Alberta

DATA-CONSCIOUS SCHEDULING OF WORKFLOWS IN METACOMPUTERS

by

Nicholas Peter David Lamb

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

in

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 **Data-Conscious Scheduling of Workflows in Metacomputers** submitted by Nicholas Peter David Lamb in partial fulfillment of the requirements for the degree of **Master of Science** in the *Department of Computing Science*.

Paul Lu (Supervisor)

Mike Carbonaro (External)

Pawel Gburzynski

Date: _____

Abstract

We develop the Trellis Driver package for integrating Java applications with Trellis metacomputers, which are user-level aggregations of hosts. Using `TrellisDriver.exec()` calls in place of `Runtime.exec()` calls, applications can distribute their workflows across metacomputers. For example, Proteome Analyst (PA) is a high-performance bioinformatics tool that executes a workflow of jobs to annotate proteomes. Running all workflow jobs on a single server severely restricts throughput for large analyses. Empirical results show that Trellis Driver's job scheduling overheads can be amortized by batching together many jobs, leading to linear speed-up of application phases.

We further investigate techniques to optimize PA's performance by reducing data movement between workflow jobs. We test our new Data-Conscious (DC) scheduling policy for Trellis in a simulation study. Simulation results show that DC scheduling is most beneficial when co-locating jobs and data offers considerable savings in either network overheads, or overheads due to application file sizes.

Acknowledgements

This thesis is the result of more than my own efforts. Many others played an important role in both the technical and non-technical aspects of this work.

First, I wish to thank my supervisor, Dr. Paul Lu, for his guidance, encouragement, insight, and constant patience throughout the prolonged timeline of this research. I extend my sincere appreciation to the Trellis research and development team, including Nolan Bard, Michael Closson, Meng Ding, Morgan Kan, Mark Lee, and Yang Wang. I gratefully acknowledge the technical and moral support I received from the Proteome Analyst team, including Dr. Duane Szafron, Alona Fyshe, Danny Ngo, and Brett Poulin. I thank Anne Nield for her prompt and helpful proof-reading work. I offer all parties my best wishes for continued success in the field of computing science.

Others outside of our department played a significant role as well. I thank my non-computing science friends, who kept me sane, if distracted, throughout my time at the University of Alberta. Finally, I wish to thank my family for their continued interest and support of my graduate studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many areas of science, including bioinformatics, chemistry, and physics, exhibit problems that require years of computing cycles to solve. Examples of such problems include protein annotation, thermodynamic molecular modelling, simulating ion transport, and simulating protein folding [26]. Regardless of the advances in central processing unit (CPU) power, or the data processing and storage capabilities of even the most powerful supercomputers at any point in time, there are always scientific problems whose resource demands exceed the capacity of any single machine or even a collection of dedicated, high-end servers from one computing site. For example, the study of the chiral recognition surfaces of two chiral molecules involves calculating interaction energies at many different separations and orientations [34]. The computation for one data point can take up to four hours on a single server, using present-day technology. A chiral recognition experiment with 10,000 data points would, for example, require 40,000 CPU hours, or approximately 4.6 CPU years. There is, then, a persistent need to aggregate resources from multiple computing sites to improve throughput – the amount of work completed over a certain time period – and significantly reduce the turnaround time of the computation. By doing so, systems integration is able to further a particular area of science.

Software applications that are written to solve large-scale scientific problems typically execute multiple programs, each of which performs a small part of the overall computation. Sometimes, each part of the computation can be performed independently. Often, however, there is a pattern of dependencies between different parts of the computation that forms a *workflow*, which is a directed acyclic graph (DAG) of interdependent application tasks. Workflows are explained in detail in Section 1.1.1. As an external example from the literature, which we will discuss only here, we can consider the workflow of an atmospheric modelling application [1, 2], shown in Figure 1.1.

In the first stage of the workflow, environmental data is acquired from a distribution of sensors, and then fed to a general circulation modelling task, which computes global weather patterns. The circulation modelling task sends its output data to a regional weather model, which produces wind data for a fixed area, using some additional sensory data. Output values from the regional weather model are then sent to several pollution modelling tasks, including a photo-chemical model, a parti-

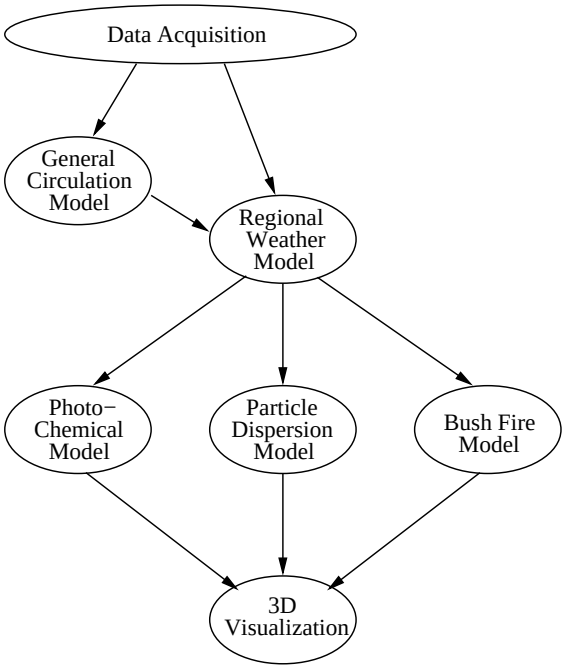


Figure 1.1: Atmospheric Sciences Workflow

cle dispersion model, and a bush fire model. The outputs from the various pollution models are then passed to a 3D visualization task, which provides a graphical representation of atmospheric trends.

In Figure 1.1, the direction of the arrows indicates the flow of data. For instance, the general circulation model task passes data to the regional weather model task. Therefore, the former task must complete before the latter may begin. Thus, the execution of jobs proceeds downwards in the diagram.

We refer to a single unit of computation, or task, as a *job*. Typically, there is a master or driver job that manages the other jobs. This master job starts the other jobs, monitors their progress, and combines their individual results to generate a complete solution. Thus, the master job controls the overall execution of the workflow. Although Figure 1.1 does not show a master job, a master job could be added to this workflow to perform, for example, the transferring of data from the sensors to the hosts that execute the various modelling programs, as well as the invocation of all data acquisition, modelling, and visualization jobs.

A *metacomputing* system is a common platform for executing scientific applications with large and complex workflows. In a metacomputing system, resources belonging to different computing centres that are managed by different administrators, are aggregated to boost computing capacity. Running applications on a metacomputing platform poses several technical challenges – namely, transparency of data access, security of communications, fault tolerance and, finally, scheduling of the resulting workload across multiple sites.

Here the term *scheduling* refers to the mapping of individual jobs to specific processors. The

problem of job scheduling has been a topic of interest in high-performance scientific computing for decades. Much work to date has focused solely on maximizing application throughput. While this metric is important in determining the overall effectiveness and user acceptance of a particular scheduling algorithm, the magnitude of data that must be transferred between networks is another essential performance factor in scheduling in metacomputing. Undue data movement imposes a performance penalty on workflow execution, as individual jobs must wait for their typically large input files to be transferred across shared networks before they can run. An ideal policy for scheduling scientific workflows places the computation near the data whenever it is feasible to do so, while still ensuring high throughput.

1.1 Key Concepts

Having explained the main motivation behind this research, we now provide background information on the key concepts of workflows, metacomputing, data consciousness in scheduling, and mechanisms versus policies. Finally, we explain how these concepts are combined to form the basis of the experimental and development work presented in this thesis. A solid understanding of these concepts and how they fit together is necessary to fully appreciate the contributions of this work and its place in the broad research area of high-performance scientific computing.

1.1.1 Workloads and Workflows

Certain applications in science perform a large number of independent computations. We refer to the set of all computations, or jobs, that must be run within an application as the *workload*. One specific type of workload is an *embarrassingly parallel* computation, in which no communication between individual jobs is required. Parameter space studies are a good example of embarrassingly parallel workloads [23]. In such experiments, each job is an invocation of the same calculation with a unique data point, and has no influence on the outcome of other calculations. Scheduling of such workloads is easy since there are no constraints on job ordering; the scheduler can assign any job to any available host.

Other applications, however, perform computations that have a pattern of interdependencies. The workloads of these applications constitute *workflows*, which are control-flow or data-flow structures of typically heterogeneous tasks that collaboratively solve a large-scale problem. Workflows are conveniently expressed as DAGs [13]. Data analysis applications with discernible computational phases are good examples of applications that execute workflows. Recall our earlier example of the atmospheric modelling program whose workflow was shown in Figure 1.1. The first application phase of data acquisition generates environmental data that is consumed by the general modelling phase that follows. This second phase produces data that is consumed by the regional weather modelling phase, and so on. The data flow between phases places constraints on the order in which individual jobs may run. For instance, any combination of the data acquisition tasks may run concur-

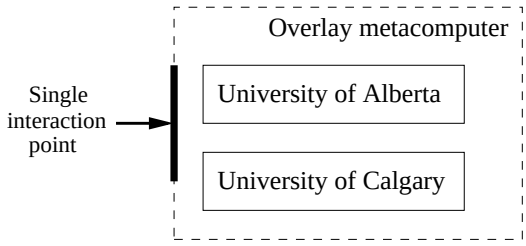


Figure 1.2: Overlay Metacomputer with Two Administrative Domains

rently, but all data acquisition tasks must complete before the general modelling task can run. The scheduling of these types of workloads is more complicated than it is for embarrassingly parallel workloads.

1.1.2 Metacomputing

Academic researchers and industry professionals who run scientific applications whose workflows contain hundreds or thousands of jobs often wish to harness the power of multiple independent servers to maximize the throughput of their applications. The servers used may fall under different *administrative domains*, which are autonomous computing centres where local administrators enforce their own policies concerning resource usage. It is common for users to obtain accounts at several high-performance computing centres (HPCCs) so that they may aggregate resources to provide the needed hardware and software capacity. The difficulty in executing workflows over multiple HPCCs is that jobs and data must be passed between administrative domains, each of which runs its own batch scheduling software, and may sit behind its own firewall.

For effective and convenient utilization of all available resources, users should be able to submit their jobs to one instance of a scheduling service that automatically distributes jobs over multiple execution hosts (or servers). Metacomputing aims to abstract a collection of individual computing servers by providing the user with a view of a single supercomputer. *Overlay metacomputers* provide this abstraction by building on top of existing infrastructure with software that performs cross-domain job scheduling.

Users can create their personal metacomputer by combining specific machines from any computing centre to which they have access. Consider the schematic diagram of a metacomputer shown in Figure 1.2. In this case, the user has access to two different administrative domains from two different universities. The separate resources from these two systems appear as one large system to the user. Overlay metacomputer software provides the user with a single point of control for job admission and management.

A common solution for providing metacomputer-wide scheduling is to implement a user-specific *metaqueue*. A metaqueue is a central depository for jobs that are to be run over a metacomputer. Metaqueues are similar to global queues in the sense that they provide high-level job queuing. Jobs

in a metaqueue are not bound to any particular host or HPCC either at the time they are submitted, or while they wait in the queue for resources to become available. Only when a job is selected for running is that job bound to a specific machine or group of machines. Based on a well-defined set of criteria, the metacomputing system decides onto which HPCC to offload a job, and automatically handles program and data shipping between domains. Users submit and monitor jobs through the metaqueue only; they need not interact with local job scheduling systems to handle the exact placement of individual jobs. This feature makes submitting jobs intuitive and straightforward to users, and allows local administrators to retain their autonomy.

1.1.3 Data Consciousness in Scheduling

Certain scheduling algorithms focus exclusively on maximizing application throughput. Throughput is defined as the number of work units completed in a fixed time interval. These algorithms, however, are often unaware of the location of data files required by individual jobs. If a large percentage of jobs must copy their data from a remote host before executing, the cumulative data movement when executing a workflow of thousands of jobs could be substantial. Moving such a large amount of data between machines across separate networks may limit the attainable throughput, since a greater proportion of execution time is now spent in communication as opposed to computation.

We aim to define a scheduling strategy that is *data conscious*, which we define as a strategy that considers the location of a job's input data when evaluating that job's suitability for execution on a specific processor or group of processors. Data consciousness is an important goal for job scheduling in metacomputing environments. Metacomputers comprise hosts from different administrative domains, possibly at geographically distant sites, so there is often considerable overhead in fetching the input data of a job from a remote host.

While data affinity (i.e., enforcing data locality through job assignment decisions) is the primary design goal of the scheduling policy we develop, there are other job-specific traits that can influence performance of workflow execution. Consider the importance of the pattern of inter-job dependencies. The completion of certain jobs may free up or unblock more waiting jobs than would the completion of others. Jobs with a higher number of dependents should be preferred for running over those with fewer dependents. The length of time a given job has been in the work queue should also be considered, in the interest of limiting the maximum time for servicing jobs. By favouring jobs with a long waiting time for execution, the scheduler prevents starvation of jobs deemed less important by other measures.

Typically, schedulers consider a variety of job criteria and map the jobs to a single priority value; the job with the highest priority is then chosen for running. Thus, our fundamental challenge in designing an efficient scheduler for a metacomputing environment is to define a set of equations that takes quantified measures of all of the aforementioned job characteristics for a given job, weighs each of them appropriately, and returns a numeric value representing that job's priority.

1.1.4 Mechanisms vs. Policies

It is important to distinguish between the notions of *mechanisms* and *policies*. A mechanism provides the infrastructure or means for carrying out a task, whereas a policy outlines rules on how that task is to be carried out. Thus, policies formulate heuristics (i.e., rules of thumb) to make intelligent decisions as to how to best use mechanisms to accomplish their tasks. Consider the task, required by an operating system, of providing users with access to a file. The `open()` system call provided by Unix systems is a mechanism for accomplishing this task. Any prefetching and caching techniques used by a particular version of Unix for optimizing file access are a matter of policy. Separation of the mechanism from the policy is advantageous because systems programmers may change one without affecting the other. For instance, implementors of `open()` may integrate a new prefetching policy into this routine without modifying its semantic behaviour or syntactic signature, both of which are examples of mechanism.

Although designing and implementing a mechanism for executing workflows with inter-job dependencies is non-trivial, solutions that are both efficient and scalable do exist [13]. Such mechanisms provide a means for users to submit jobs, and have those jobs run on remote hosts. Choosing the best job to assign to a free host, however, is a matter of policy.

A policy for scheduling workflow jobs in metacomputers defines a method for choosing the most appropriate job to run next, from a group of several candidate jobs. During the execution of any non-trivial workflow, there are often multiple jobs that can be run at a given moment due to flexibility in the pattern of dependencies between jobs. A greater number of candidate jobs offers the scheduler more flexibility in assigning jobs to hosts, which in turn can lead to better job assignments.

1.1.5 Integration of Key Concepts

The individual concepts discussed above fit together in a specific way within the context of this thesis. We address scientific applications performing large-scale computations that entail running a workflow of jobs, and that require high-performance computing resources to complete within a timeframe acceptable to the researchers who depend on those computational results. Our execution platform is an overlay metacomputer, which is a convenient aggregation of hosts from multiple administrative domains. Overlay metacomputers are a simple yet effective way of obtaining the necessary computing capacity for such applications.

Given a metacomputing environment and a workflow of stand-alone jobs, each with its own input data needs, we seek a strategy of assigning jobs to hosts that places jobs with their input data when practical to do so. We focus on developing a scheduling policy that upholds data consciousness by weighing the cost of moving files across networks against other job-specific factors.

1.2 Contributions and Outline

Having introduced the computing science concepts that are fundamental to this work, and subsequently explained how these concepts relate to each other, we now highlight the main contributions of this work, and provide an overview of the organization of this thesis.

1.2.1 Contributions

There are three main contributions of this thesis:

1. **Development of a Java Package to Transform Process-level Concurrency into Workflow Concurrency Within Metacomputers**

   We use the bioinformatics tool Proteome Analyst (PA) as a guiding application. PA's workflow contains several opportunities for job parallelism, but exploitation of this parallelism requires integrating the application with the cross-domain scheduling service typical of a metacomputer. We develop a new software module called Trellis Driver that allows Java applications, such as PA, to integrate with the existing Trellis metacomputing system.

2. **Implementation of Job Batching Strategies to Amortize Scheduling Overheads**

   The additional software layer between the PA application and Trellis metacomputing infrastructure imposes a non-negligible latency on the launching of external jobs. We describe our support for batching together jobs of either the same or a different type to amortize job scheduling overheads. Empirical results obtained in a local area network (LAN) setting show that job batching provides linear speed-up (i.e., a 4 times speed-up on 4 processors) of data-parallel phases.

3. **Development and Evaluation of a Data-Conscious Scheduling Policy that Reduces Workflow Turnaround Time**

   After evaluating PA's performance in a real setting, we develop a simulator that models the PA workflow and the underlying Trellis system. We use this simulator to explore the effects of executing PA's workflow over a wide area network (WAN), which has higher latencies than a LAN, and increasing the sizes of the workflow job input files.

   The increased data movement cost in such scenarios leads us to develop the Data-Conscious (DC) scheduling policy that considers the location of input data when assigning jobs to hosts. Through our simulator, we compare the performance of our DC policy against two existing, widely-used scheduling mechanisms. Our results show that DC scheduling produces notably shorter workflow turnaround times when the the network communication costs or the file sizes are large.

1.2.2 Outline

PA is a bioinformatics tool that provides a high-performance classification framework for protein sequences. It consists of a Java driver program that invokes other helper programs to analyze and infer functional characteristics of individual proteins. Prior to this work, PA ran all helper jobs on the same server. Test trials of the original version of PA indicate that for large inputs there are phases that take roughly five hours to complete. Using the newly-developed Trellis Driver Java package, we integrate PA with the Trellis metacomputing system – thereby allowing PA to distribute its workflow of jobs across aggregations of servers comprising an overlay metacomputer. Empirical results show that linear speed-ups can be achieved for data-parallel phases.

Once PA is integrated with metacomputing, we next optimize the execution of PA's workflow by improving the scheduler within the Trellis system. We define the DC scheduling policy that places jobs on hosts where their input data resides, when deemed practical, so that the cumulative data movement is reduced. We develop a simulator that models the execution of PA's workflow over a Trellis metacomputer. Simulation results indicate that DC scheduling places the maximum possible percentage of jobs with their input data, and reduces makespan (i.e., turnaround time) by up to 53% against First Come First Served (FCFS) in a WAN setting when the file sizes are sufficiently increased.

Chapter 2

Motivation

Chapter 1 explained the fundamental problem in high-performance scientific computing that our research addresses, and highlighted the key concepts of workflows, metacomputing, data consciousness, and scheduling policies. In this chapter, we provide the motivation for our choice of a guiding scientific application, the development of a software module that enables the simple integration of our application with Trellis metacomputers and, finally, the consideration of input data location when mapping jobs to processors.

Initially, we describe our motivating application and its workflow, and explain why we believe this application can benefit from being run across an overlay metacomputer. Next, we address the shortcomings in the current mechanism our application uses for running external jobs, and explain the need for a replacement module that allows jobs to be handed off to a metacomputing system – thereby implicitly scheduling the jobs across many remote hosts. Finally, we provide a detailed illustrative example of a simple workflow that can benefit noticeably from a scheduling policy that places jobs with their input data.

In Chapter 3, we review existing solutions for parallelizing workflows, integrating resources from multiple computing sites, running external jobs through language mechanisms, and considering data location in job scheduling. In Chapter 4, we provide an architectural overview of all components involved in the integration of our chosen application with metacomputing. We describe how communication is carried out between the scientific application, the metacomputing integration module, and the metacomputing system itself.

2.1 Example Application: Proteome Analyst

We chose to use an existing scientific application with an established user community as a guide in developing our metacomputing integration software, and our Data-Conscious (DC) scheduling policy. This way, we can be assured that the resulting products works in real-world situations. We also have the privilege of contributing to another discipline in science by providing improved computational support.

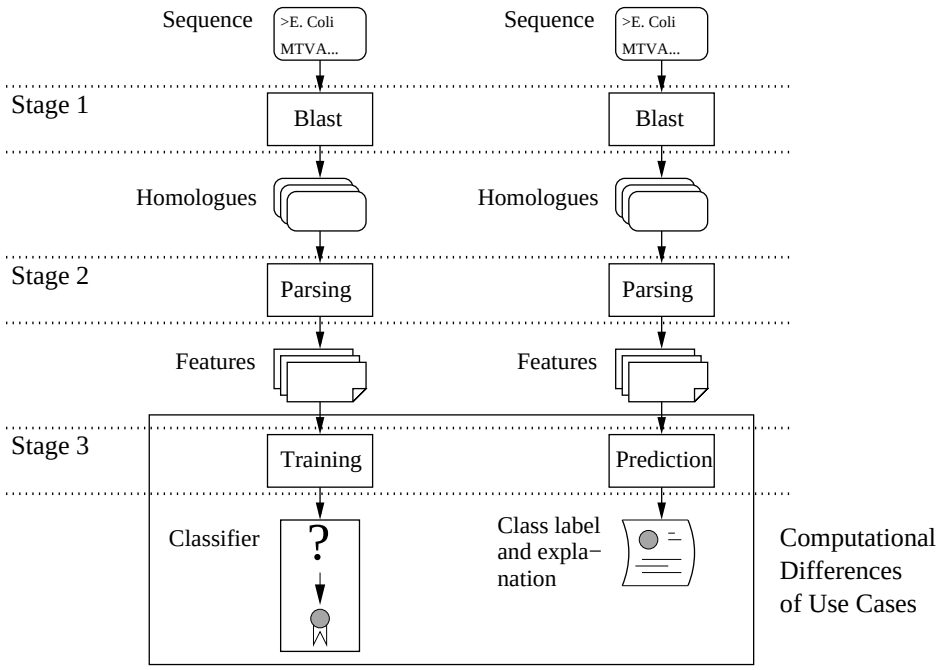


Figure 2.1: Job Pipelines for Training and Prediction

Among the many biological problems that require years of computing power to solve is that of annotating large sets of protein sequences. Currently, there are more than 1,200 genomes analyzed and stored in public databases [9]; some of these genomes have up to tens of thousands of sequences. The sheer volume of this biological data necessitates the development of tools that can extract meaningful data trends and thereby convey useful biological knowledge to researchers.

Proteome Analyst (PA) is a bioinformatics application that provides detailed annotations of proteomes (sets of protein sequences) [29]. Annotations offered include general function, which indicates the primary role the protein plays in the cell; and subcellular localization, which indicates the location within the cell where the protein performs its primary role.

PA uses the machine learning technique of classification to make predictions or annotations of a new protein [20]. Classification answers the question: Given a data sample and a finite set of distinct categories, of which of these categories is the data sample a member? One use for predicting the properties of a protein is in suggesting the kinds of physical experiments that would be most valuable for empirically confirming the predictions made about that protein.

Although the above summary provides an accurate snapshot of PA, the PA application has many more capabilities. PA can create a custom classifier to predict a new property, from a set of labeled proteins. PA's graphical exposition of annotation results offers a "what-if" analysis, in which users may see the effects of altering an individual feature on a protein's predicted properties.

For our purposes, we focus on two distinct aspects of PA, both of which have similar workflows that are computationally-intensive. Figure 2.1 shows the workflows for these two common use

cases. First, PA can machine-learn a classifier as part of a training process (left side of Figure 2.1). PA accepts a training set, which is a set of proteins whose class labels have been assigned by domain experts through experimental analysis, and from this training set automatically builds a Naive-Bayes classifier in a learn-by-example fashion [20]. Second, PA can use an existing classifier to predict annotations of new (i.e., previously unseen) proteins (right side of Figure 2.1). PA accepts a query sequence, performs a lexical analysis on this sequence, and then assigns it to a functional category. Note that when describing PA's inputs, we use the terms "protein" and "sequence" interchangeably.

Both the training and prediction processes produce workflows that are three-stage pipelines. Conceptually, a protein is either part of a training set or it is a query sequence that is being analyzed for prediction. Each protein in the input proteome produces one instance of the pipeline.

During the training process, the initial input to a pipeline is a protein from the training set, which has an assigned class label. In Stage 1, a text version of the protein, which is a string representation of its primary structure, is compared against the Swiss-Prot biological database of sequences, using the Basic Local Alignment Search Tool (BLAST) toolset [4]. Swiss-Prot is a high-quality, curated database of known proteins and their various properties [5]. The output from this string-matching step is a set of homologues, or proteins with a similar primary structure. In Stage 2, the known information about the homologues is parsed to extract features (descriptive keywords) from the Swiss-Prot database. After the Parsing job is complete, we possess a keyword summary of a particular protein. In Stage 3 of training, a mapping function from the features to a class label is machine-learned, and used in the construction of a new classifier.

During the prediction process, the initial input is an unknown protein, which we call a query sequence, whose class we wish to determine. The first two stages of the prediction pipeline are identical to those in the training pipeline: A string representation of the sequence is fed to the BLAST utility, which produces homologues. These homologues are then given to the parsing utility, which produces features. Stage 3 is a prediction step in which the extracted features, or keywords, of the query sequence are given to an existing classifier for analysis. The classifier produces a class label, thereby assigning that sequence to an ontological (i.e., metaphysical) category.

In either use case, the jobs from all three pipeline stages are "small" in the sense that they have a short running time, typically on the order of seconds, and work with relatively small input and output data, typically of size tens of kilobytes or less. A single pipeline therefore requires few computing cycles and minimal data storage space. However, some proteomes have thousands or tens of thousands of sequences. The human genome, for example, consists of approximately 23,000 sequences [31], This results in roughly 23,000 pipeline instances being executed. Thus, the collective computational and data storage requirements for large analyses can be quite high.

To assess and illustrate the need for workflow concurrency in such cases, we measured runtimes of various application phases in a test program using the original PA. In this experiment, we trained and validated a new classifier using a moderately-sized training set based on a collection of proteins

Test Program Phase	Workflow Stage	Runtime (H:MM:SS)
BLAST	Stage 1	4:51:26
Feature Extraction	Stage 2	0:08:33
Machine Learning	Stage 3 (left side)	0:05:43
Resubstitution	Stage 3 (right side)	0:18:34
Total		**5:24:16**

Table 2.1: Phase Times for Training and Validation of a New Classifier Based on Gram Negative Bacteria, Using Original PA

from several gram negative bacteria that consists of 3,916 protein sequences and 1,531 features (keywords). PA was run on a single Linux box with two AMD Athlon MP 1800+ processors, 1.5 GB main memory, and Red Hat Linux 7.1.

Table 2.1 shows the correspondence between each of the four phases of our test program and the previously illustrated workflow pipelines, as well as the runtimes from these phases. Note the long BLAST runtime of nearly 5 hours. Recall our earlier example of the human genome with its 23,000 proteins, which is roughly six times the size of the sample proteome used here. In that case, the BLAST phase would require roughly 30 hours.

The original PA uses Java's `Runtime.exec()` facility to invoke the external BLAST program (Stage 1 in the pipeline). Unfortunately, all these jobs are run locally (i.e., on the same server as the main PA process). Thus, the application's performance is limited to that of the local server, which must execute every job in every pipeline. The BLAST phase's long runtime, shown in Table 2.1, empirically confirms the disadvantage of running all these jobs onto the local server.

Since the first two stages in each pipeline perform an independent analysis on one particular protein, we can execute an arbitrary number of BLAST and Parsing jobs concurrently, without sacrificing application coherence. For this reason, Stages 1 and 2 are embarrassingly parallel. Distributing BLAST jobs from distinct pipelines over the constituent hosts of a metacomputer can significantly increase PA's throughput (i.e., work units completed in a fixed time interval) for large or even moderately-sized proteomes, including that used in the test program above. Enabling a Java application for metacomputing however, requires a replacement for `Runtime.exec()` that passes a job to a metacomputer scheduler instead of running that job locally. We describe the new software module we developed to achieve this goal in Section 2.2.

With the improvement in application throughput that a metacomputing platform offers comes the challenge of avoiding excessive data movement between administrative domains. As mentioned earlier, undue data movement imposes a performance penalty on PA. To effectively parallelize PA, the assignment of jobs to hosts must fit well with the pattern of data flow between jobs.

When executing PA's workflow, we should strive to assign jobs from Stages 1 and 2 of a common pipeline to the same machine. The homologue file outputted by a BLAST job (Stage 1) must be present on the machine on which the subsequent Parsing job (Stage 2) runs. Moving a few such files

between hosts has minimal impact on the workflow's execution time. Moving the homologue file of every protein in a proteome as large as the human genome, however, produces a high volume of total data movement. In some cases it may be worthwhile to delay execution of a Parsing job until the machine that ran the corresponding BLAST job becomes available. The value of delaying the execution of a job in the interest of placing that job near its data is further discussed in Section 2.3.1.

2.2 System Support: Trellis Driver

Metacomputers offer a suitable parallel execution platform for resource-intensive scientific applications. While there are many existing metacomputing solutions, there is an ongoing need for a mechanism that provides the integration of applications with a metacomputer scheduler.

PA and other scientific applications consist of a main job that invokes, or drives, other jobs. At various points in execution, the main job invokes external programs to run specialized "helper" jobs that perform a specific part of the computation. If there are many of these helper jobs, or if they are resource-intensive, it can be desirable to run several of them concurrently across multiple servers to maximize throughput.

PA, which is written in Java, uses the Java 2 Platform application programming interface (API) function `Runtime.exec()` [28] to launch the BLAST jobs. As illustrated in Figure 2.1, BLAST constitutes Stage 1 of all pipelines. In the original version of PA, all BLAST jobs are run locally, which means that on a single-processor machine, these jobs must contend with the main driver program for the same processor. On a multi-processor machine, it would be possible to run several helper jobs simultaneously on separate processors, but this only achieves job parallelism within the local host. In the case of the human genome with its 23,000 odd proteins, restricting all 23,000 of the required BLAST jobs to a single server would seriously limit the attainable throughput and hence increase the turnaround time for the analysis of this proteome.

A more practical solution is to modify the job dispatching functionality within a driver-based application to use the resources of a metacomputer. Users get the benefit of executing several workflow jobs simultaneously, without having to manually place individual jobs on remote machines.

Given the existing PA application, which is already driver-based, and the existing Trellis metacomputing system [23, 22], we develop a new software module called Trellis Driver that integrates these two components. As much as possible, Trellis Driver behaves as a drop-in replacement for `Runtime.exec()`. Trellis Driver provides a method `TrellisDriver.exec()`, for instance, that takes a command string to be run as a separate process. However, rather than passing that command string to the local operating system for execution, `TrellisDriver.exec()` sends the command string to the Trellis scheduling service for execution in an underlying metacomputer. Basing the command interface and the functionality of Trellis Driver on that of the `Runtime` class simplifies modifying PA (and other Java applications) to use Trellis Driver. We further discuss Trellis Driver in Chapter 4.

2.3 Optimization of Workflow Execution: Data-Conscious Scheduling Policy

Integrating PA with the Trellis metacomputing system enables the parallel execution of PA's workflow. Such a measure reduces the time needed for processing proteomes, possibly by a wide margin. There is, however, a contrast between enabling more brute force computing power and using the existing power more wisely. Performance can also be improved by a metacomputing scheduling policy that considers the location of the input files of jobs. For reasons argued above, we wish to avoid shipping too many files between the hosts that comprise a user's overlay metacomputer.

Although individual homologue files are relatively small, and consequently, impose little data movement overhead, recall that every input protein produces one instance of the analysis pipeline. PA's workflow, then, becomes quite data-intensive for large input proteomes.

In this section, we first explain the trade-off between data affinity and throughput, and provide an intuitive argument of how a scheduling policy can achieve both these objectives in a manner that maximizes application performance. Next, we quantify our previous argument by providing an illustrative example of the execution of a small workflow, in which data file sizes and job runtimes are taken from PA. We show that exploiting data locality improves performance noticeably, even in this simple case.

2.3.1 Data Locality vs. Throughput

Scheduling policies used in any metacomputing environment must address the central goal in capacity computing, that of maximizing throughput. When scheduling scientific workflows, the placement of jobs near their typically large input data becomes important. Blindly assigning any job to any available host may produce a substantial amount of data movement, which can significantly slow the overall computation. One strategy for achieving data locality is to withhold jobs from execution when their data is not locally accessible, which has the undesirable effect of lowering central processing unit (CPU) utilization.

Sometimes, it may be desirable not to assign a job to any presently available host, but wait until another host, on which a copy of that job's input data resides, becomes free. The delay in starting the job's execution may be compensated for by the service time reduction due to the avoidance of having to transfer a large amount of data across the network. A job's service time is defined as the time to fetch any and all input files from a remote host, if necessary, plus the time to execute that job. Waiting too long for a desired host to become free may delay the time of an individual job's start (and completion) to the point where data locality produces no net savings in service time. Additionally, a scheduler's frequent refusal to dispatch jobs to hosts because they do not have the desired data may result in reduced job parallelism.

There is then, a trade-off in preserving data locality and maintaining high throughput. Figure 2.2 provides a conceptual graph of this compromise. This graph shows the throughput for varying

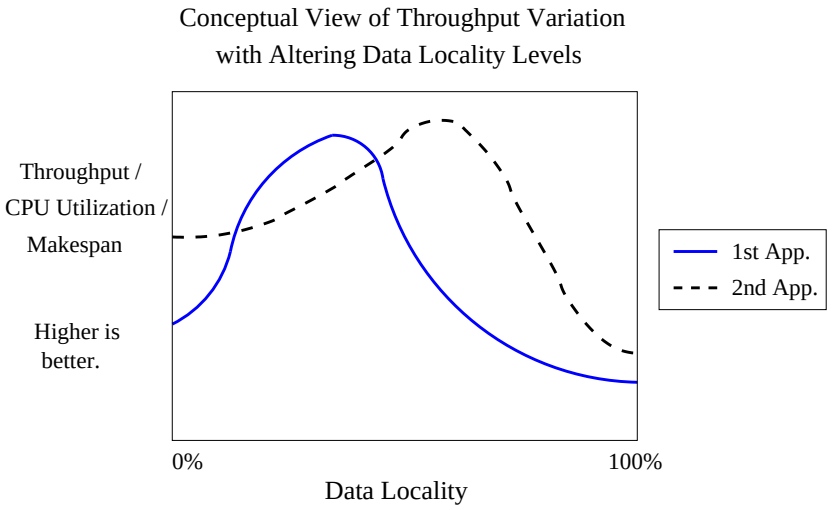


Figure 2.2: Conceptual Trade-off Between Data Locality and Throughput

data locality levels for two different applications. The curves drawn in Figure 2.2 serve merely as illustrative examples, and do not represent any actual or expected universal workflow behaviour.

Recall that throughput refers to work units completed in a fixed time interval, and that work refers solely to computation and not communication. Thus, if a job is idle for a fixed amount of time because it is waiting for a copy operation to finish importing its required input files, the job's throughput is zero for that time interval.

Data locality refers to the proportion of jobs that are placed with their data. This metric is a property of the scheduler's policy settings, and cannot be directly set. By varying the emphasis placed on input data location in the scheduler's job placement decisions, we can alter the level of data locality realized when executing workflows, and observe the corresponding effects on application throughput. In the graph shown above, a data locality level of 0% means no jobs are placed on a host where their data resides. A data locality level of 100% means every job is placed on a host that has a copy of its input data. In practice, both these extreme cases might never occur.

From Figure 2.2, we see that when the scheduling policy provides no data locality (i.e., the 0% case), throughput is not that high for either application. Every job must copy its input files from a remote host before it can run, therefore a considerable portion of every job's service time is spent in communication as opposed to computation. The data movement overhead seriously limits throughput. As data locality is increased by the scheduling policy, throughput at first improves, reaches a maximum, and then steadily declines, with both applications. A higher degree of data locality implies that at any given time, a greater proportion of jobs are actively working on their computations, and not waiting for their input data to be retrieved from a remote host. Hence, throughput is higher.

However, there comes a peak in performance. This occurs when the penalty to job service time, caused by scheduler's refusal to assign jobs to available hosts who do not have the appropriate data,

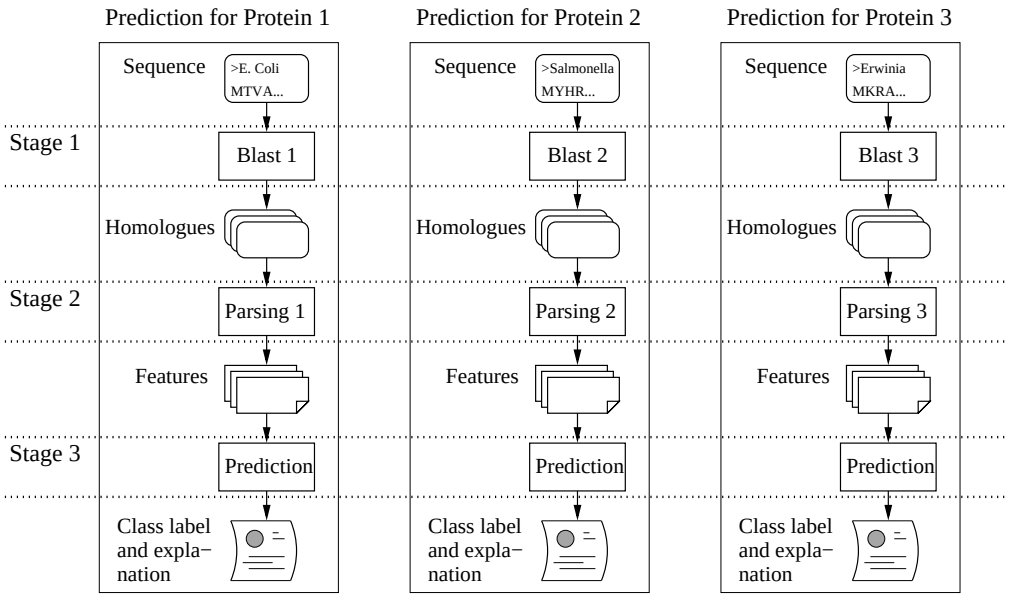


Figure 2.3: Multiple Instances of Proteome Analyst's Pipeline Workflow

Pipeline Event	Protein 1	Protein 2	Protein 3
BLAST	3.8	3.3	3.0
Data Transfer	1.1	1.1	1.0
Parsing	0.5	0.6	0.7

Table 2.2: Runtimes and Data Transfer Times for All Pipelines.
All times shown in seconds.

begins to outweigh the savings from avoiding data movement by handing out a job later. Note the differences between the two applications in terms of the exact data locality value that delivers peak performance. The second application is likely more data-intensive than the first, and thus, requires a higher level of data locality to achieve maximum throughput.

When the scheduler pursues the goal of data locality too aggressively, it is more likely to avoid assigning a job to an available host, choosing instead to wait for the host with the required data to become free. With many hosts frequently idle, fewer workflow jobs are running simultaneously. In the extreme case of data locality being enforced in every job assignment (i.e., the 100% case), throughput is quite low for both applications.

2.3.2 Scheduling Example

After explaining the trade-off between data locality and throughput, we now provide a detailed scheduling example based on the behaviour of PA when run on a real metacomputer that spans a wide area network (WAN). Our example demonstrates the performance gains that exploitation of data locality offers.

In our example use case, three proteins whose pipelines are shown in Figure 2.3 are given as input as part of a prediction process. As explained earlier, it is desirable to run Stages 1 and 2 of a common pipeline on the same machine to avoid transferring the homologue file, which is passed between these two pipeline stages, between metacomputer hosts. In this scenario, there are two hosts in our user's metacomputer that are connected via a WAN; we refer to these as hosts A and B, respectively. We assume that the two hosts have identical hardware and therefore the runtimes of any job are precisely the same on both hosts. We also assume that the text files for all three input proteins, which contain the string representation of the sequence structure, are available on either host. Thus, the input to the BLAST job, which is the initial input to a pipeline, is always stored locally.

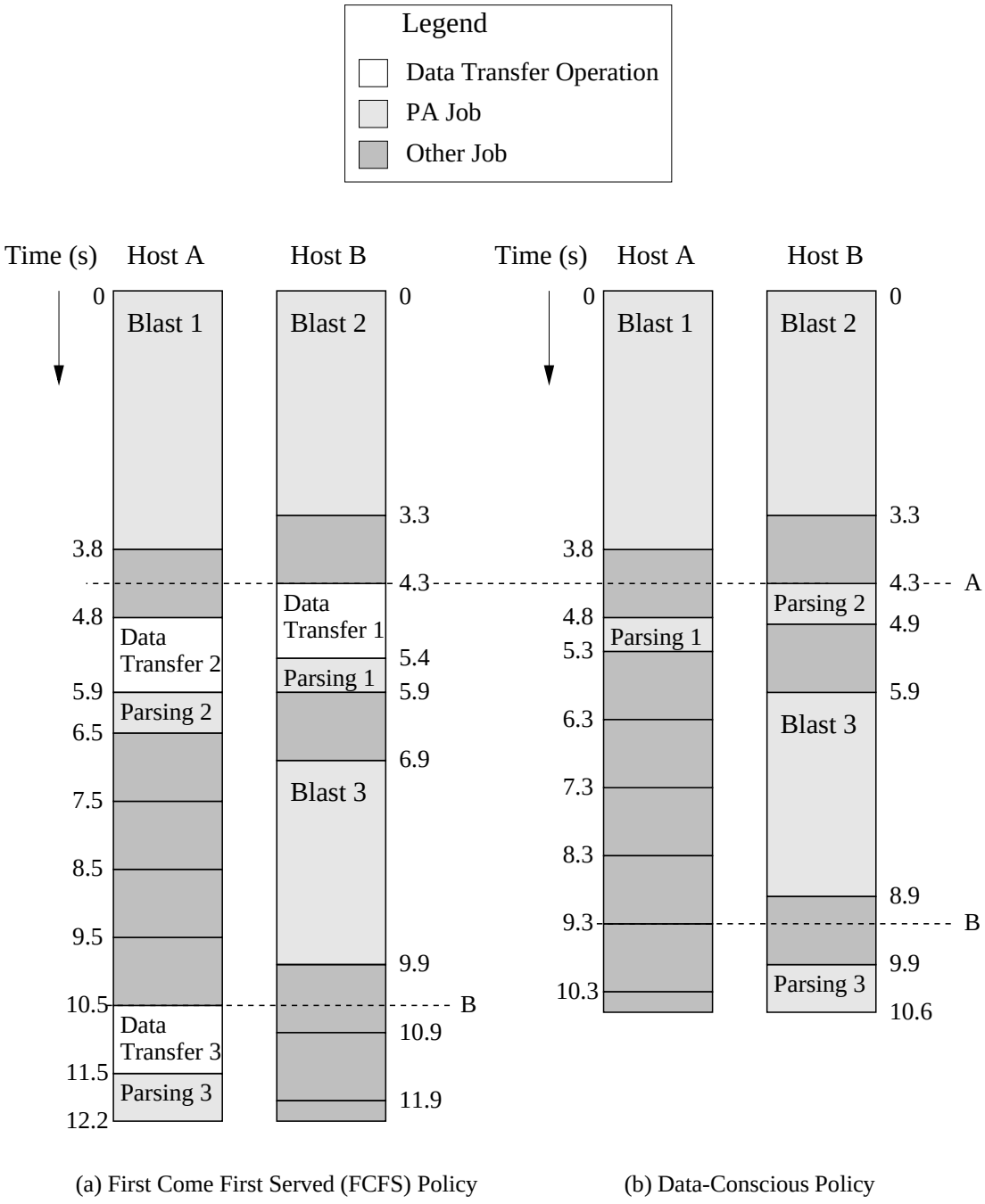
Table 2.2 shows the non-negligible data transfer times between the hosts, for the homologue files that are produced by Stage 1 and consumed by Stage 2, as well as the runtimes of the BLAST and Parsing jobs. Values are given for all three pipeline instances from the workflow shown in Figure 2.3, and are all shown in seconds. We see that communication time is significant compared to computation time: The data transfer time is roughly 1 second, and the combined processing times for BLAST and Parsing jobs are roughly 4 seconds.

Typically, the PA user does not have exclusive access to the hosts comprising their overlay metacomputer. Given the shared nature of metacomputing resources, we simulate another user's jobs, which run on both machines utilized by our PA user, throughout our sample workflow's execution. For simplicity, we assume that all the other user's jobs take exactly 1 second to complete. We also assume that the batch scheduler on both hosts constantly alternates between executing a job belonging to our PA user, and executing a job from the other user. When no jobs from our PA user are available to run, the local schedulers simply run one of the other user's jobs.

In our example, and in the real PA, the PA driver process creates pipelines of jobs in a serial manner. That is, PA jobs are issued to the metacomputer scheduler in the following order: the BLAST job from the first pipeline (BLAST 1), the Parsing job from the first pipeline (Parsing 1), the BLAST job from the second pipeline (BLAST 2), and so on. In addition, a job's submission includes inter-job dependency information – so the scheduler knows, for example, that it must wait for a BLAST job to complete before it can run the corresponding Parsing job.

Figure 2.4 (a) shows the job mappings for our sample workflow resulting from a First Come First Served (FCFS) metascheduling policy that does not consider the location of a job's input data. Figure 2.4 (b) shows the job mappings resulting from a data-conscious metascheduling policy that places jobs with their input data.

Note that the initial job mappings produced by both policies are identical. The two policies behave differently when choosing a job to run on host B after 4.3 seconds, which is marked as Decision A in Figure 2.4. The FCFS scheduling policy chooses Parsing 1 over Parsing 2 since the former comes earlier in the job ordering. Parsing 2 is chosen to run on host A somewhat later. Both job



(a) First Come First Served (FCFS) Policy

(b) Data-Conscious Policy

Figure 2.4: Job Assignments for Sample Multi-Pipeline Workflow

assignments entail fetching the homologue file of the relevant protein from the other metacomputer host, on which the corresponding BLAST job ran. A noticeable amount of time (1.1 seconds in both cases) is required to transfer the data files before the Parsing jobs can run.

The data-conscious scheduling policy chooses Parsing 2 to run on host B at Decision A, instead of Parsing 1, since the former job's input data is already on this host, having been generated by the BLAST 2 job. Parsing 1 is then chosen to run on host A, since BLAST 1 ran on this host. Since no data movement is required by either job assignment, these first two parsing jobs complete 1.2 seconds earlier than in the FCFS scenario.

The two policies next differ when scheduling the last workflow job, Parsing 3, which is marked as Decision B in Figure 2.4. In the FCFS scenario, the Parsing 3 job is run on host A as soon as this machine becomes available after BLAST 3 has completed. This entails transferring the homologue file of protein 3 from host B. In the data-conscious scenario, the scheduler decides, instead, to wait for host B to become available. The delay in starting this final workflow job is more than made up for by the time savings from the subsequent avoidance of data movement. Such a scheduling decision requires a strategy for knowing when specific metacomputer hosts will become free. We discuss our chosen strategy in Chapter 6.

The makespan, or turnaround time, for the entire workflow is 12.2 seconds in the FCFS scenario and 10.6 seconds in the data-conscious scenario. Since there were three proteins analyzed, the throughput is 3 / 12.2 $\approx$ 0.25 proteins per second for the FCFS case and 3 / 10.6 $\approx$ 0.28 proteins per second for the data-conscious case. The improvement in performance becomes evident with larger proteomes. For instance, given a proteome with 1,000 sequences, the FCFS scheduler will complete the analysis in 1,000 / 0.25 = 4,000 seconds (1:06:40) whereas the data-conscious scheduler will require only 1,000 / 0.28 = 3,572 seconds (0:59:32).

To make the scheduling decisions necessary for maximizing utilization and throughput, the scheduler must successfully weigh the benefits of exploiting data affinity against the cost of delaying a job's execution. The scheduler, then, requires accurate estimates on the times at which various metacomputer hosts will become free, the job's expected runtime, the size and location of all the job's input files, and the data transfer rate. We explain our sources for obtaining estimates of these parameters in Chapter 6.

Chapter 3

Related Work

The previous chapter provided the motivation for using metacomputing as an execution platform for scientific applications, developing a software package that allows Java programs to distribute jobs across metacomputers, and developing a scheduling policy that enforces data locality. To date, a considerable amount of research has been done in metacomputing, job launching mechanisms, and scheduling policies.

In this chapter, we review existing solutions to the aforementioned problems. We then explain why the discussed solutions do not adequately solve our overall problem of minimizing turnaround times for scientific workflows that are executed over metacomputers.

First, we discuss existing strategies to parallelize the BLAST program within bioinformatics workflows, and justify our method of parallelism. Second, we outline two existing systems for distributing jobs over servers at distinct computing sites, and explain why we use Trellis, instead, to execute Proteome Analyst (PA)'s workflows. Third, we review existing programming language mechanisms for launching external jobs from within an application. We explain how none of these legacy mechanisms provide implicit job scheduling across multiple hosts. Finally, we discuss previous work in data consciousness in inter-host job scheduling.

This chapter serves to justify the technology choices we make in our work. The remaining chapters describe, in detail, the software tools and the scheduling policy we develop to efficiently execute PA's workflows over metacomputers.

3.1 Parallelization of BLAST in Bioinformatics Workflows

Improving the performance of applications such as BLAST, which are common in bioinformatics workflows, is a topic of considerable interest to system researchers, particularly as the size of biological databases increases. The BLAST phase is the performance bottleneck of PA, as indicated in Chapter 2. We now discuss previous solutions to parallelizing BLAST, and justify our decision of running a sequential (i.e., non-parallel) version of BLAST.

Braun *et al.* [6] discuss three approaches to parallelizing BLAST, each at a different level of

granularity. The first two fine-grained approaches to parallelism involve partitioning the biological database, and distributing the database subsets over a cluster of workstations. These strategies seek to reduce the time for a single similarity search. The third coarse-grained approach involves replicating the database on several workstations or hosts, and partitioning the set of input queries among the hosts. This strategy seeks to improve the throughput of multiple sequence similarity searches.

Users of PA are primarily concerned with the throughput of many (i.e., possibly thousands) of sequences, and not the response time for an individual sequence. We therefore prefer the replication approach over the partitioning approach. Although this method of parallelism requires storing the entire database at each host, the Swiss-Prot [5] database that PA uses is a mere 64 MB in size, which entails negligible storage requirements for modern workstations.

Meyer *et al.* [19] explore the performance gains of alternate data distribution strategies on a three-phased protein modelling workflow. The workflow is a pipeline of jobs consisting of two initial phases – BLAST and a sequence filtering step – followed by an optional third phase, in which either of two programs – Modeler or Threader – might be run. The selection of which program to run, if any, in the third phase is based on the results of the second phase.

The authors report on experiments in which the workflow is parallelized at a coarse level by running multiple pipelines on separate hosts concurrently. The input data set is evenly partitioned, and the resulting query subsets are distributed by a master node to the BLAST jobs on worker nodes (i.e., execution hosts) at the start of execution. Unfortunately, reductions in turnaround time level off significantly when more than four hosts are used. This limit on performance improvement was attributed to load imbalance. The Threader program (the third phase) has by far the longest runtime, but is only run for 3 out of 66 input sequences. Often, one host ends up executing two, or even all three, of the Threader programs.

The above results demonstrate that statically distributing all the BLAST jobs (and pipelines) to the hosts can lead to a load imbalance when the runtimes of different pipelines vary drastically, as is the case in PA. We believe that a dynamic scheduling strategy, in which jobs are assigned to hosts based on availability, has greater potential to improve the performance of such pipelined bioinformatics workflows.

Wang *et al.* [33] propose the BLAST++ tool that exploits common substructures (words) within multiple input sequences to provide efficient processing of batched queries. BLAST++ follows the same basic algorithm for finding similar sequences as standard (i.e., non-batched) versions of BLAST. In standard BLAST, however, batches of queries are processed simply by running BLAST on query sequences one at a time. BLAST++ processes a batch of queries in one execution run by formulating a single virtual query from the concatenation of all input queries, and then searching the database for sequences similar to the virtual query. For N query sequences, then, BLAST++ scans the entire database only once for similar words as opposed to N times, as does the standard BLAST.

Empirical results demonstrate that BLAST++ reduces the time for processing query sets deci-

sively – by at least 50% – against the benchmark, non-batched BLAST. The performance gains of BLAST++ are particularly evident in trials using the human genome dataset, demonstrating that the batching approach scales well to large datasets. Batching the execution of multiple jobs can benefit PA by reducing the communication between hosts, and by amortizing scheduling overheads.

Another version of BLAST, MEGABLAST [35], also optimizes the processing of queries in batches. However, Ma *et al.* [17] investigate MEGABLAST and find that the algorithm trades sensitivity, which is the percentage of actual sequence similarities found, for efficiency (i.e., processing time). Thus, while MEGABLAST runs faster, it may miss out on answers found by the original BLAST.

We choose to use the non-batched BLAST [4] from the National Center for Biotechnology Information (NCBI) because it is used by domain experts (i.e., biologists). Furthermore, PA views BLAST as a black box, meaning that PA is not dependent on the particular implementation of BLAST. Thus, our current BLAST utility could be substituted with BLAST++, if desired. To obtain high throughput, we parallelize BLAST by distributing the individual jobs over an aggregation of servers, such as a grid or a metacomputer. Metacomputers offer performance scalability, since more hosts can be added as needed to augment the available computing power.

3.2 Resource Integration: The Virtual Supercomputer

A common strategy for boosting computing capacity is to aggregate resources from multiple sites and make them appear as a single resource. Workflow-based applications such as PA can then hand off jobs to an underlying job scheduling service for automatic placement and execution, instead of manually placing jobs on specific hosts, which is cumbersome for large workflows.

In the following subsections, we first review two existing systems that provide job scheduling across administrative domains, and then briefly explain why we adopt the Trellis system as a parallel processing platform for PA.

3.2.1 Globus

The Globus Alliance [11] is a well-known effort to develop the fundamental technologies for building computational grids in which computing power is traded as a resource, much like electricity in a power grid. The Globus toolkit provides software for building grid systems and applications. Globus offers the essential services of resource discovery and management, and cross-domain security.

As discussed by Lamb and Lu [16], Globus is feature-rich and powerful in concept, but has a number of shortcomings: 1) Administrators from all participating sites must negotiate service-level and security agreements; 2) The Globus middleware is non-trivial to set up; and 3) The middleware must be installed and managed by administrators with privileges. We therefore seek a much simpler solution for resource integration.

3.2.2 Condor

Condor [30] is a distributed job batching system designed for running compute-intensive jobs over collections of servers. Although originally developed for harnessing central processing unit (CPU) cycles of idle workstations, Condor supports a variety of distributed systems, including clusters and multiprocessors. Condor's "flocking" technology [10] allows it to work across administrative domains, creating a grid-like computing environment.

Unfortunately, Condor suffers from a significant drawback: All workstations that are to be a part of the Condor pool must have the Condor software installed by system administrators. We prefer to aggregate resources through overlay metacomputing, which does not require administrator privileges to set up.

3.2.3 Trellis

The Trellis metacomputing system [22] is a simple way to integrate resources from multiple high-performance computing centres (HPCCs) to boost computational capacity. Unlike the two systems described above, Trellis does not require all participating parties to adhere to a specific set of standards, or adopt new infrastructure, and does not require administrator privileges to set up and use. Based on the concept of overlay metacomputers, which was explained in Section 1.1.2, Trellis does not replace existing local schedulers, but builds on top of them with a centralized scheduler that hands off jobs to specific hosts, based on availability. No new hardware or software need be adopted by local administrators, except for the Secure Shell (SSH) software suite, which is used for secure inter-domain communication [21]. SSH is open-source software that is easy to deploy and use, and is already in use worldwide.

3.3 Language Support for Running External Jobs

After having justified our choice of Trellis as a solution for integrating disparate HPCC resources, we now seek a means of invoking Trellis to run external jobs from within an application. Trellis is potentially easy to use from a programming standpoint because it offers a single point of control for administering jobs over collections of hosts from multiple sites.

As indicated earlier, driver-based applications need a way of efficiently calling out to external programs from within the master or driver process. We now review existing programming language mechanisms for running external jobs, and explain our choice of Trellis Driver for carrying out this task.

3.3.1 Subprocess Creation: Running Jobs Locally

Nearly all modern scripting and programming languages provide one or several application programming interface (API) functions that allow one process to start another process on the same

machine. ANSI C [15] and scripting languages with a similar syntax, such as Python [32], provide the commands of `system()` and `popen()` for creating child processes, or subprocesses. The `system()` function takes a command string, passed in as an argument, and executes that command string in a separate process. `system()` blocks until the underlying subprocess completes, at which time its exit status is returned to the caller. Thus, `system()` runs subprocesses in a synchronous fashion.

In workflow-based execution, however, we often wish to run subprocesses asynchronously. For instance, to achieve maximum concurrency within the BLAST phase of PA, the driver should be able to start all the BLAST jobs and then await their completion before proceeding to the next phase. A PA driver process using `system()` to launch a workflow job would be stalled until that one job completes. This serializes the execution of workflow jobs, eliminating the possibility of job concurrency. Thus, `system()` is not useful in the asynchronous execution of jobs.

ANSI C provides the `popen()` function as a non-blocking way of launching a new process. `popen()` runs a given command string in a separate process and creates a pipe for communication between the parent and child process. While an open inter-process communication channel is sometimes desirable, it is not necessary for workflow-based applications. Often, a driver merely starts the workflow jobs, and checks for their completion at a later time. Moreover, `popen()` is too low-level a mechanism for our work. Using this function to run PA jobs over multiple hosts would require specifying the host that is to run each job. Manual placement of jobs is onerous for large workflows.

In common versions of Unix and Linux, the `system()` and `popen()` functions are implemented using the lower-level functions of `fork()`, `exec()`, and `wait()`. The Unix and Linux APIs provide access to these three functions as a means of manually starting subprocesses [27]. The `fork()` command creates a child process that is an exact clone of the parent process. Typically, a call to `fork()` is followed by a call to `exec()` in the child process, which starts running a new command in that child's executable image. The parent may await completion of the child with the function `wait()`, which returns the subprocess' exit status.

A driver process can run external jobs asynchronously by starting jobs, as desired, through `fork()` and `exec()` invocations, and then later awaiting the completion of jobs through `wait()` calls. However, starting subprocesses with `fork()` and `exec()` requires a later call to `wait()` for every issued workflow job, which is inconvenient from a programming perspective. A more practical alternative would be to use a job barrier function that blocks until every outstanding subprocess has completed. This way, drivers could synchronize with the completion of an entire workflow phase with one function call.

As mentioned earlier, the PA application is written in Java and uses the Java API function `Runtime.exec()` to start external jobs, such as the BLAST jobs in Stage 1 of the protein analysis pipelines (Figure 2.1). `Runtime.exec()` automates many of the steps involved in creating a new process. `Runtime.exec()` also has the benefit of being highly portable, since Java 2 plat-

form functions are constant across different architectures. Unfortunately, like the combination of `fork()`, `exec()`, and `wait()`, the `Runtime.exec()` facility has no job barrier functions. A driver process must then make a separate call to `Process.waitFor()` to await the completion of every workflow job it initiated.

In addition to lacking workflow barrier functions, `Runtime.exec()` and all the other language mechanisms described above do not provide integration with job schedulers. All issued jobs are run locally, meaning it is only possible to exploit process-level concurrency within a single server. The large-scale scientific applications we are addressing in this research [26] require a high degree of job-level concurrency, attainable only by distributing their workflows across multiple hosts, to complete within an acceptable time frame (i.e., hours or days, as opposed to weeks or months).

Programmers could pass to job launching mechanisms, such as `Runtime.exec()`, command lines that invoke remote shell technologies, such as SSH, to run workflow jobs on remote hosts. However, as with `popen()`, this strategy requires deciding in advance where individual jobs will run. To successfully load balance an application's workflow, which means distributing the computation among the hosts in a manner that reflects the current resource usage, a runtime binding of jobs to machines is more appropriate than *a priori* job assignments, since the latter can only be based on a static system view. Thus, we desire a language mechanism that passes jobs to a scheduler that offloads those jobs onto available hosts.

3.3.2 Remote Method Invocation: Running Jobs Remotely

The Java 2 Platform includes the Remote Method Invocation (RMI) package [28] that allows objects in one program to invoke the services of objects in a different virtual machine and hence on a different host. Theoretically, RMI could be used to place objects that run workflow jobs on remote machines, and have the PA driver invoke these remote objects as needed. Unfortunately, RMI is specific to Java, and therefore only works between Java objects and processes. PA requires a way of calling out to non-Java utilities, such as BLAST, to perform protein analyses. We must instead seek a solution that supports non-Java executables as well.

3.3.3 Trellis Driver: Running Jobs Over Metacomputers

We develop the Trellis Driver package as a means for PA and other Java applications to run jobs over aggregations of hosts comprising a Trellis metacomputer. Trellis Driver sends PA-issued jobs to the Trellis metascheduler, which provides cross-domain scheduling of the jobs.

Trellis Driver allows the PA driver process to invoke, through `TrellisDriver.exec()` calls, external programs to run workflow jobs remotely. The external programs invoked may be Java or non-Java executables. `TrellisDriver.exec()` executes jobs asynchronously, meaning that PA is not blocked until the newly-issued job completes, and may, then, run multiple workflow jobs

concurrently. Trellis Driver's API hides the details of subprocess creation, such as the selection of an execution host, and offers a high-level interface for job dispatching. Additionally, Trellis Driver provides workflow synchronization functions, so that applications may start any number of jobs, and then later issue a single function call to await the completion of all outstanding jobs.

3.4 Co-Locating Jobs and Data

Trellis Driver enables concurrency within PA's workflow by allowing PA to execute many jobs simultaneously across Trellis metacomputers. While workflow concurrency can greatly enhance performance, further performance optimization is possible by adapting the metascheduler to enforce data locality, which entails placing jobs near their input data files. We now review a previous approach to integrating scheduling and data placement.

Shan *et al.* [24] investigate the impact of file sizes and network communication costs on workload turnaround time in a computational grid. The authors employ distributed scheduling, in which multiple peer Grid Schedulers (GS'), located on disparate servers, exchange resource availability information to decide on which server a job should be run. Through simulation, the authors model the execution of large workloads consisting of up to tens of thousands of jobs, at multiple sites connected via a wide area network (WAN).

The first simulation results indicate that the distributed scheduling algorithm produces an average response time that is almost as good as that attained by a baseline centralized strategy. Shan *et al.* do not consider the centralized strategy further; they deem this strategy impractical due to a lack of fault-tolerance and scalability. Althrough Trellis uses centralized scheduling, Trellis' design is fault-tolerant because the failure of one or many execution hosts does not affect the continued execution of the workload jobs by the remaining hosts. Previous real-life experiments [23] that entailed executing a scientific application over 18 administrative domains have demonstrated Trellis' scalability.

Additional results presented by Shan *et al.* show that when input data sizes are not considered, there is a severe performance penalty for large file sizes. When the file sizes are inflated by a factor of 10, average response times increase more than ten-fold over the case with the original file sizes. Average response times also increase substantially when the servers are spread over fewer sites and multiple servers at each site must contend for the slower WAN links. From these last results, it appears that when the workload is distributed over multiple sites, WAN connections can become performance bottlenecks. The above findings suggest that file sizes and network transfer rates should be considered when scheduling large workloads over aggregations of servers from multiple sites, which metacomputers constitute.

Chapter 4

Trellis Driver: Architecture and Functionality

This chapter describes the architecture and functionality of the Trellis Driver software layer. We first demonstrate how the Proteome Analyst (PA) application uses Trellis Driver to execute its workflow over metacomputers. We then describe the design of both the Trellis Driver module and the Trellis metacomputing system, upon which the former module builds.

Chapter 2 explained how metacomputing can boost the performance of scientific applications through increased throughput. We gave an illustrative example of how scheduling that is data conscious further increases performance in metacomputers that span wide area networks (WANs). Chapter 3 contrasted and justified our approach to parallelizing BLAST against previous solutions. We justified our choice of Trellis as an execution platform for PA, and Trellis Driver as a means for distributing jobs over multiple hosts. Finally, we reviewed related work in data-conscious scheduling.

We now describe the components involved in integrating PA with Trellis. Initially, we show code snapshots from the original and modified versions of PA to provide a clarifying example of how Java programs use Trellis Driver. Next, we introduce the Trellis metacomputing system, and explain the placeholder scheduling mechanism. Next, we provide an architectural overview of the Trellis Driver module. Finally, we describe the Trellis Driver application programming interface (API).

In Chapter 5, we provide the implementation details of Trellis Driver, describing both the job barrier functions and the job batching mechanisms contained therein. Chapter 6 describes our simulation program, which models a WAN-based metacomputing environment, and serves as a testbed for developing our DC scheduling policy.

4.1 Sample Usage

Existing Java applications, such as PA, that use the `Runtime.exec()` facility to run external jobs can easily be adapted to use Trellis Driver. Figures 4.1 and 4.2, respectively, show the original PA

code that runs the BLAST jobs locally, and the modified version of the same PA code that runs the BLAST jobs across metacomputers, using Trellis Driver. Numbers at the start of each line of text in these two figures represent the actual line numbers within the PA source files. Adapting PA to use Trellis Driver requires adding 6 new lines of code and replacing 13 existing lines with 6 new ones, in the section of code shown here. Consequently, the changes required to port PA to metacomputing are modest.

In Figure 4.1, code is taken from two program segments of the original PA: the `execute()` method in the `ClassifierWrapper` class, and the `executeBlast()` method in the `BlastTaskPredictor` class. The former routine coordinates the four major activities involved in the building and validation of a new classifier based on a training set. These four activities are listed inside comments in the code, from lines 316 to 319 in Figure 4.1. The first task of feature collection actually entails the two steps of finding homologues using BLAST and feature parsing, of which only the code for the first is shown. As shown on lines 375 to 382, PA iterates through the training set and executes a training task (i.e., a BLAST analysis) on each protein.

The function called in the loop body (line 381, Figure 4.1) triggers a chain of function calls, of which only the top-level call is shown, that together formulate and execute the appropriate BLAST command for the current protein. The `executeBlast()` routine, shown on lines 141 to 153 in the latter code segment in Figure 4.1, makes the actual call to `Runtime.exec()` that invokes the BLAST utility (line 146). After the BLAST job is started through `Runtime.exec()`, the `Process.waitFor()` method is called (line 147) to await the job's completion. The execution of BLAST jobs is synchronous, since the program starts each BLAST job, and then waits for it to complete before proceeding to the next analysis. The original PA, therefore, has no concurrency between BLAST jobs.

Figure 4.2 shows the same two program segments from the modified (i.e., parallelized) version of PA, which we refer to as *PA-Trellis*. Note that there is an extra line of code at the beginning of the `ClassifierWrapper.execute()` function (line 326, Figure 4.2) that obtains a reference to the associated Trellis Driver object, which is stored in the variable `td`. The loop that iterates through the training set and starts the BLAST task for every protein (lines 379 to 386) is unchanged from the original PA version. There is, however, an extra line of code just before the loop (line 376) that instructs Trellis Driver to group the execution of two BLAST jobs on the same metacomputer host whenever possible. Chapter 5 shows how batching multiple jobs together amortizes scheduling overheads.

The `BlastTaskPredictor.executeBlast()` routine is changed substantially from the original version. A reference to the `TrellisDriver` object (as opposed to the `Runtime` object) that is associated with this Java virtual machine (JVM) instance is first obtained (line 141, Figure 4.2). The call site of `td.exec()` (line 146), which replaces the original call to `Runtime.exec()`, starts a BLAST analysis on the current protein as a Trellis job. PA-Trellis

```
ClassifierWrapper.execute():

313    public void execute() {
314      /*
315       * this method accomplishes the following tasks:
316       *   1. collects features for the proteins for training
317       *   2. starts the classifier's training process
318       *   3. creates and runs the resubstitution cardSet
319       *   4. creates the Summary.html page for this classifier
320       */
321      ReportTimer timer = new ReportTimer(
322          "Classifier Training", this.classifierId,
323          ReportTimer.CLASSIFIER_TRAINING);
324      timer.startEvent("Classifier Training");

364        SequenceList seqs = new SequenceList(this.user, this.listId);
365        numTrainingInstances = seqs.getProteins().size();
366
367        timer.startEvent(
368          "Homologue Finding (BLAST)", "Classifier Training");
369
370        Protein protein;
371        Task trainTask = this.trainingPolicy.getTrainingTask();
372        Task filter = this.trainingPolicy.getFilter();
373
374        /* 1. Get some features */
375        for (Iterator proteinIter = seqs.getProteins().iterator();
376             proteinIter.hasNext();
377             ) {
378          protein = (Protein) proteinIter.next();
379
380          //      analyze proteins
381          trainTask.execute(protein, protein.getOutputDir());
382        }
383        timer.stopEvent("Homologue Finding (BLAST)");

BlastTaskPredictor.executeBlast():

141        Runtime rt = java.lang.Runtime.getRuntime();
142
143        String commandline = this.command + " -i "
144               + protein.getFile();
145
146        Process p = rt.exec(commandline);
147        int error = p.waitFor();
148        if (error != 0) {
149          System.err.println(
150              "BlastTaskPredictor.executeBlast() Non-zero exit code: "
151              + error);
152        }
153        p.destroy();
```

Figure 4.1: Code for BLAST in Original (Sequential) Version of Proteome Analyst

```
ClassifierWrapper.execute():

313    public void execute() {
314      /*
315       * this method accomplishes the following tasks:
316       *   1. collects features for the proteins for training
317       *   2. starts the classifier's training process
318       *   3. creates and runs the resubstitution cardSet
319       *   4. creates the Summary.html page for this classifier
320       */
321      ReportTimer timer = new ReportTimer(
322          "Classifier Training", this.classifierId,
323          ReportTimer.CLASSIFIER_TRAINING);
324      timer.startEvent("Classifier Training");
325
326      TrellisDriver td = TrellisDriver.getDriver();

366        SequenceList seqs = new SequenceList(this.user, this.listId);
367        numTrainingInstances = seqs.getProteins().size();
368
369        timer.startEvent(
370          "Homologue Finding (BLAST)", "Classifier Training");
371
372        Protein protein;
373        Task trainTask = this.trainingPolicy.getTrainingTask();
374        Task filter = this.trainingPolicy.getFilter();
375
376        td.setGroup("BLAST", 2);
377
378        /* 1. Get some features */
379        for (Iterator proteinIter = seqs.getProteins().iterator();
380             proteinIter.hasNext();
381            ) {
382          protein = (Protein) proteinIter.next();
383
384          //      analyze proteins
385          trainTask.execute(protein, protein.getOutputDir());
386        }
387        int[] exitcodes = td.waitForAll();
388
389        timer.stopEvent("Homologue Finding (BLAST)");

BlastTaskPredictor.executeBlast():

141        TrellisDriver td = TrellisDriver.getDriver();
142
143        String commandline = this.command + " -i "
144               + protein.getFile();
145
146        td.exec(commandline, "BLAST");
```

Figure 4.2: Code for BLAST in Modified (Parallel) Version of Proteome Analyst

eliminates the call to `Process.waitFor()` since this version does not wait for one BLAST job to complete before starting the next. With this asynchronous job launching strategy, all BLAST jobs are issued before any results are collected. PA-Trellis can then obtain a maximum degree of concurrency equal to the number of placeholders in the underlying metacomputer.

After having iterated through the training set and started the appropriate BLAST job for each protein, the `ClassifierWrapper.execute()` routine must await the completion of all the BLAST jobs before it may proceed with the feature parsing step, the code for which is not shown in the figure due to space constraints. The call site `td.waitForAll()` (line 387, Figure 4.2) causes the application to block until all outstanding BLAST jobs have finished, and returns an array containing the exit codes of these jobs.

We have shown the front end of Trellis Driver to emphasize the ease of porting existing Java applications to metacomputers through this module. We next provide the back-end details by giving an architectural overview of the Trellis metacomputing system and the Trellis Driver module itself in Sections 4.2 and 4.3, respectively. The command interface of the `TrellisDriver` Java package is further described in Subsection 4.3.1.

4.2 Trellis Metacomputing System: Overview

The Trellis system [23, 22] is a metacomputing solution aimed at scientific researchers who require the computational resources of many high-end servers. Developed in a Unix environment, Trellis provides a convenient user-level aggregation of high-performance computing centres (HPCCs) that are contained within separate administrative domains, possibly spanning multiple platforms, operating systems, and job queuing systems. Trellis is layered on top of existing infrastructure, offering job management services that provide the abstraction of a single, powerful resource from multiple resources. As mentioned in Section 3.2, the only new software required is the widely-available Secure Shell (SSH) security tool [21]. Moreover, Trellis is deployed entirely at the user-level, meaning users may install and configure their own instance without administrator support.

Users can define a personal metacomputer by combining specific hosts from any of the HPCCs to which they have access. Note that users need not have exclusive access to these constituent hosts. HPCC resources are typically shared among members of a dedicated research or industry community. Local job batching systems enforce equitable usage of their respective computing systems, according to the policies of local administrators. All that users of Trellis require is a regular, non-privileged account on any HPCC whose resources they wish to use. Users get the benefit of utilizing resources from multiple HPCCs while local administrators retain their autonomy.

After specifying the participating hosts in their metacomputer, users can create one or more metaqueues to which they can deposit jobs that are to be run over the Trellis system. Multiple metaqueues are supported so that users can run and monitor multiple applications at the same time. In addition, users may create metaqueues with differing properties, such as job ceiling time and

the maximum or minimum number of processors bound to placeholders. A metaqueue may be customized to the needs of a specific application.

Trellis is implemented as a thin layer of software that sits between applications and the infrastructure of HPCCs. The Trellis platform consists of multiple components that handle all the essential tasks involved in the cross-domain scheduling and execution of large workflows with inter-job dependencies and per-job data requirements. In particular, Trellis includes:

1. **Trellis File System (Trellis FS)**
   As job migrate from one metacomputer host to another, a practical problem that occurs is that of ensuring the required data is always available. Trellis FS [25] provides transparent access to remote data and supports many common file system operations, such as sparse access and caching.

2. **Trellis Security Infrastructure (Trellis SI)**
   Since metacomputer hosts span multiple HPCCs, security across administrative domains is a primary concern. Trellis SI [14], layered on top of SSH, provides single sign-on and handles cross-domain authentication, authorization, and data management.

3. **Trellis Metascheduler and Command Line Server (CLS)**
   The Trellis Metascheduler includes all commands necessary for specifying and managing hosts and metaqueues, as well as those for submitting and monitoring jobs. Contained in the metascheduler is the CLS, which carries out the distribution of jobs to metacomputer hosts. The metascheduler is the Trellis component this work focuses on.

4.2.1 Placeholder Scheduling

In Trellis, jobs are assigned to specific hosts through the mechanism of *placeholder scheduling* [22], which follows a "pull" model. In placeholder scheduling, local batch queues interact with metaqueues to retrieve and execute jobs on demand. A placeholder is formally defined as a unit of potential work. For a given unit of work, which in this case constitutes a job in an application's workflow, it is possible for any placeholder within all of those previously launched to actually complete that work. Each placeholder is associated with a particular metaqueue and is submitted to a particular host by the user. One can then view a placeholder as a special-purpose job that is submitted to the batch queue of a remote host.

Often, placeholders are implemented as job scripts for batch schedulers. From the perspective of the local batching system, then, placeholders are simply regular, non-privileged user jobs that must wait in the queue for their turn to run.

Figure 4.3 illustrates the placeholder scheduling mechanism. Each Trellis user installs and runs one instance of Trellis CLS, create metaqueues as desired, and submits placeholders to hosts as is necessary to achieve the desired level of concurrency. When a placeholder job runs, it first contacts

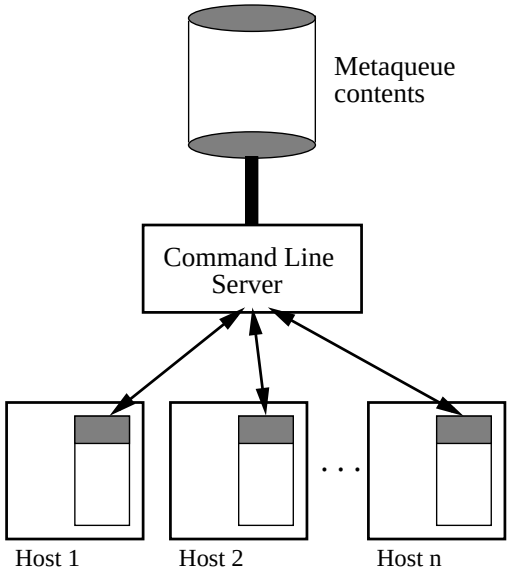


Figure 4.3: Placeholder Scheduling Mechanism

its associated metaqueue, via the Trellis CLS, and asks for a new command line. A command line is the means of specifying a unit of actual work (a job) in a Unix environment. Thus, the binding of jobs to hosts occurs at placeholder execution time (not submission time), under the control of the CLS. When assigned a new job, the placeholder simply runs the corresponding command in its local environment, and then reports the job's completion, via the Trellis CLS, to the metaqueue. When there are no jobs in the metaqueue that are presently available to run, the CLS informs the placeholder of this, and the placeholder can either re-submit itself to its local batch queue or take itself offline.

This scheduling mechanism follows a "pull" model, in which jobs on remote hosts initiate communication by requesting work, and pull jobs out of a metaqueue and onto their local computer system. This is in contrast to a "push" model, in which a scheduling process offloads jobs onto individual computers according to a work distribution algorithm.

While simple in nature, the pull model in Trellis effectively solves the problem of load imbalance, which stems from an uneven distribution of jobs over the available resources. Hosts that are heavily loaded will have fewer placeholders asking for work over a given time interval and will pull fewer jobs from the metaqueue. Hosts with a lighter load will have more placeholders asking for work and pull more jobs from the metaqueue. Placeholder scheduling therefore achieves load balancing across all hosts.

4.2.2 Job Control in Trellis CLS

The names and semantics of the CLS job management commands are based on those of existing batch queuing systems, such as the Portable Batch System (PBS) [3] and the Sun N1 Grid Engine [7]

job schedulers. PBS provides the commands `qsub`, `qdel`, and `qstat` to add jobs, remove jobs, and list the contents of a local job queue, respectively. A PBS queue functions much like a printer queue. New jobs are added at the back of the queue and must wait until all previous jobs finish before they are processed (i.e., executed). By analogy, Trellis' placeholder scheduling is similar to having a pool of printers (in this case hosts) pull jobs from a queue instead of having a printer daemon find idle printers and push jobs onto those printers. Trellis CLS provides the commands of `mqsub`, `mqdel`, and `mqstat` to add jobs to a metaqueue, remove jobs, and query the metaqueue status, respectively.

Following are examples of usages of the three Trellis CLS commands for job management:

1. `mqsub -b -p protein1BLAST protein1Parse "java FeatureParser 1 15 1 0.01 ~/output/proteome15/protein1"`

   Adds a job to the default Trellis metaqueue. The new job belongs to the target "protein1Parse" and has the command line "java ca.pence.utils.FeatureParser 1 15 1 0.01 ~/output/proteome15/protein1", as given by the last two arguments.

   A *target* refers to a group of workflow jobs that are mutually independent and can, therefore, be executed in parallel. The -p flag allows Trellis users to specify inter-target dependencies. In this case, "protein1BLAST" is a prerequisite target. The newly-submitted job will not run until all jobs in the "protein1BLAST" target have completed.

   The -b flag specifies that the command runs in blocking mode, meaning the caller is suspended until Trellis completes the job.

2. `mqsub -q gram_neg protein1BLAST "BLASTpgp -E 0.001"`

   Adds a job belonging to the target "protein1BLAST", which has no prerequisites, to the "gram_neg" metaqueue named with the -q flag.

3. `mqdel -q plant1 20`

   Deletes the job whose numeric identifier is 20 from the "plant1" metaqueue.

4. `mqstat -q animal1 -t squirrelBLAST -s E`

   Displays information on the jobs in the "animal1" metaqueue. The -t flag selects those jobs belonging to the "squirrelBLAST" target, while the -s flag and E parameter specify only jobs that are currently executing.

4.3 Trellis Driver

Trellis Driver is a Java module that integrates applications with the Trellis metacomputing system. Trellis Driver allows applications to launch external jobs and transparently schedule these jobs across the hosts of an overlay metacomputer. Since jobs are no longer restricted to the local server, the computational capacity of the application's execution environment can be greatly increased, leading

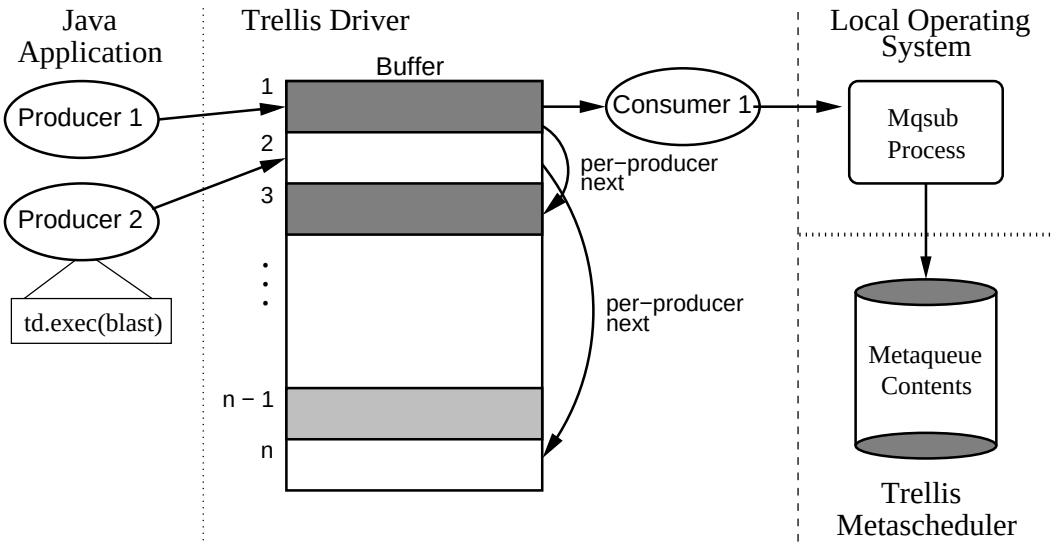


Figure 4.4: Bounded Buffer Approach to Metacomputing Integration

to increase throughput and reduce turnaround times. The fundamental concept of Trellis Driver is general enough to lend itself to any programming language that provides functions for running command lines as separate processes.

PA, and other Java programs, import Trellis Driver as a Java package. The front end of Trellis Driver is a well-defined API, designed to be similar to that of the `Runtime` class (and its associated functions such as `Runtime.exec()`) for ease of portability. A detailed listing of all API functions is presented in Subsection 4.3.1. Briefly, Trellis Driver's API provides functionality for specifying job grouping patterns, starting jobs through the function `TrellisDriver.exec()`, and synchronizing with their completion. The back end of Trellis Driver interacts with the relevant metaqueue by issuing blocking `mqsub` calls, through the local operating system. Accordingly, there must be an instance of the CLS actively running on the same machine on which the Trellis Driver-enabled application runs.

Trellis Driver adds to the Trellis metascheduler, and CLS, the following three features:

1. **Job Launching Through a Library Function**

   Although a driver process could call out to the `mqsub` utility via a library function such as `system()`, specifying a new job in this manner often requires passing several parameters, as seen in the example `mqsub` use cases shown in the previous subsection. Trellis Driver automates and simplifies the submission of jobs to `mqsub` by abstracting cumbersome `mqsub` command details, such as the flags and prerequisite target list. External jobs may be started from within an application through Trellis Driver library functions, which take command strings as arguments.

2. **Job Batching to Amortize Scheduling Overhead**

   Programmers may provide hints about the grouped nature of the application's workflow jobs. Trellis Driver automatically batches the execution of a preset number of jobs from a specified job group to amortize the overhead of creating a new `mqsub` process for running these jobs. The functions for enabling job batching are given in Table 4.1 while the purpose and concept are explained in Chapter 5.

3. **Job Barriers for Driver Synchronization with Workflow**

   Since scientific workflows can consist of hundreds or even thousands of jobs that can run independently of the driver process, support for computational synchronization is necessary. The driver process can await the completion of some or all workflow jobs through the job barrier functions provided by Trellis Driver. These same functions provide a means of collecting the exit codes of jobs that were executed asynchronously.

Figure 4.4 shows the various software components in our metacomputing integration solution. The Trellis Driver component (horizontally central) is newly developed, while the other components were implemented prior to this work. The data flow resulting from multiple job submissions follows a producer-consumer pattern. Threads within the application (producers) generate jobs by calling `TrellisDriver.exec()`, abbreviated as `td.exec()` here. Threads within Trellis Driver (consumers) process jobs by calling `mqsub` to send them to the Trellis metascheduler, which executes on the same server as the application. `mqsub` inserts the command line into the desired metaqueue for processing by the Trellis system, as seen in the bottom right corner of Figure 4.4.

Each application thread or producer can call `td.exec()` multiple times, since a thread may launch as many external jobs as it desires. Figure 4.4, however, shows the data flow situation that occurs when two different producers each make only one call to `td.exec()`. At the same the two producers are pushing jobs into the buffer, there is one consumer that is pulling jobs out of the buffer. Dark gray entries represent jobs submitted by Producer 1, white entries jobs submitted by Producer 2, and the light gray entry a job submitted by a third producer that is not currently adding a job to the buffer. When Producer 1 calls a work barrier function to wait on all its outstanding jobs, Trellis Driver walks the linked list of jobs belonging to Producer 1 by following the "per-producer next" links shown, waiting for each job to complete in turn. In the example above, Trellis Driver waits at buffer entry 1 until that job has completed before examining buffer entry 3. Only after the job at entry 3 has completed does the job barrier function return.

To decouple the number of producer threads from the number of consumer threads, Trellis Driver uses a standard bounded buffer to store incoming jobs. The bounded buffer is a limited-size storage space for command lines. The advantage of this design is that we keep the generation of jobs separate from the execution of jobs. We wish to avoid starting a new consumer thread for every incoming job, since this could flood the JVM with hundreds or even thousands of threads. Having this many

threads active at the same time would seriously undermine performance. There is no limit on the number of producer threads, since this depends on the application code. Maintaining a fixed number of consumer threads, that is independent of the number of producers, ensures scalability.

Trellis Driver allows users to set the size of the bounded buffer and the number of consumer threads. Thus, if an application programmer is conscious of resource usage within the JVM, they may specify a relatively small buffer size and low number of consumers. If the programmer has many thousands of jobs to run within a short time frame, and sufficient JVM resources, they may specify a relatively large buffer size and high consumer count. Programmers can also specify which Trellis metaqueue to use, so that they may pick a metaqueue that is tailored to their current application.

Applications can submit jobs either synchronously or asynchronously through Trellis Driver. In synchronous mode, the calling producer thread blocks until Trellis completes the given job. In asynchronous mode, the caller continues after submitting a Trellis job, and confirms the completion and collects the exit status of that job at a later time. Implementing the work barrier functions requires keeping track of which jobs are submitted by which producers. Buffer entries maintain the special-purpose "per-producer next" links to maintain this information.

4.3.1 Trellis Driver API

The Trellis Driver API provides a command interface through which Java programs can hand off jobs to the Trellis metascheduler for remote execution. The API provides methods for defining the Trellis Driver execution environment (e.g., specifying which metaqueue to use), starting and stopping the job scheduling mechanism, running jobs synchronously and asynchronously, and awaiting the completion of one or many outstanding Trellis jobs. The complete Trellis Driver API specification is given in Tables 4.1, 4.2, and 4.3.

API Function	Description
`getDriver()`	Returns the Trellis Driver object associated with the current application. The Trellis Driver object is then used for all subsequent communication between the application and Trellis Driver. Analogous in purpose to the `Runtime.getRuntime()` method.
`getConsumersCount()`, `setConsumersCount(` *`count`* `)`	Retrieve and specify the number of consumers, respectively. Acceptable values for consumer count currently range from 2 to 512.
`getBufferSize()`, `setBufferSize(` *`size`* `)`	Retrieve and specify the number of buffer entries, respectively. Acceptable values for the buffer size currently range from 100 to 10,000.
`getQueueName()`, `setQueueName(` *`metaqueue_name`* `)`	Retrieve and specify the name of the Trellis metaqueue to which jobs are submitted, respectively.
`setGroup(` *`group_name`*`,` *`batch_factor`* `)`	Registers a new job group, labeled as *group_name*, with Trellis Driver. The numeric value *batch_factor* indicates how many such jobs are to be batched together into a single `mqsub` command.
`createPipeline(` *`pipeline_name`*`,` *`length`* `)`	Defines a new job pipeline, labeled as *pipeline_name*, that contains as many stages as specified by *length*. This means that *length* jobs, possibly of different type, are to be batched together into a single `mqsub` command.
`start()`	Informs consumers to start running jobs.
`stop()`	Informs consumers to stop running jobs.

Table 4.1: Trellis Driver API: Configuration Functions

API Function	Description
`execSynch(` *`command_line`* `)`	Runs the given command (i.e., *command_line*) as an external Trellis job, in synchronous mode. The function call blocks until Trellis completes the job, at which time the exit code of the `mqsub` process that ran this job is returned.
`exec(` *`command_line`*`,` *`command_group`* `)`	Runs the given command (i.e., *command_line*) as an external Trellis job, in asynchronous mode. The optional parameter *command_group* specifies the job group or pipeline this job belongs to. The function call returns when the job is inserted into the bounded buffer. The function returns a key that the caller can use to reference the newly-submitted job later.
`exec(` *`prod_id`*`,` *`command_line`*`,` *`command_group`* `)`	Identical in purpose and behaviour to the version of `exec()` shown above, except for an extra initial parameter that identifies the producer that is submitting this job. This version of `exec()` allows different producers to separate their jobs from one another.

Table 4.2: Trellis Driver API: Job Launching Functions

API Function	Description
`waitForOne( key )`	Waits for the Trellis job associated with the given reference key (i.e., *key*) to finish. The exit code of the `mqsub` process that ran the specified job is returned to the caller, after Trellis finishes executing the job.
`waitForAll()`	Waits for all outstanding Trellis jobs to finish, regardless of which producer submitted them. An array of exit codes from the respective `mqsub` processes that ran the various jobs is returned to the caller, as soon as the last job finishes executing.
`waitForAll( prod_id )`	Waits for all Trellis jobs that were submitted by the specified producer (i.e., *prod_id*) to finish. An array of exit codes from the respective `mqsub` processes that ran all of this producer's jobs is returned to the caller, as soon as this producer's last job finishes executing.

Table 4.3: Trellis Driver API: Workflow Synchronization Functions

In the next chapter, we follow the computational flow within Trellis Driver that occurs when PA launches a series of external jobs. For now, we provide an example of the steps involved in starting and monitoring jobs through Trellis Driver:

1. **Configuration**

   The programmer first calls the functions shown in Table 4.1 to configure the Trellis Driver execution environment. The runtime parameters of number of consumers, buffer size, and metaqueue name are set through calls to the appropriate three functions. Workflow information is then conveyed to Trellis Driver with calls to `setGroup()` and `createPipeline()`, which the programmer uses, respectively, to define new job groups and pipelines. The programmer then calls `start()` (only once) to enable job processing by Trellis Driver.

2. **Job Launching**

   Through calls to `TrellisDriver.exec()` (Table 4.2), the programmer starts all desired external jobs. Multiple calls to `TrellisDriver.exec()`, which replace any former `Runtime.exec()` invocations, achieve job parallelism in the Trellis environment. Workflow jobs may also be executed synchronously via `TrellisDriver.execSynch()`, in which case only one external job runs at a time.

3. **Workflow Synchronization**

   Using the functions listed in Table 4.3, the programmer enforces synchronization between the driver process and the workflow jobs that were launched by the driver. Workflow synchronization is achieved by imposing work barriers that block the application until some or all outstanding workflow jobs have completed. Calls to `TrellisDriver.waitForAll()` allow the application to block until all jobs started by a given producer thread, or all jobs

started by any thread, complete. Calls to `TrellisDriver.waitForOne()`, which are analogous to `Process.waitFor()` calls, allow the application to await the completion of individual jobs. Finally, the programmer calls `stop()` (only once) to end the job processing by Trellis Driver, and terminate the JVM threads within Trellis Driver.

4.3.2 Comparisons with MPI

Programmers familiar with the popular Message Passing Interface (MPI) message passing library standard [18, 8] will note the similarities between the functionality offered by our Trellis Driver package and the message passing services of MPI. Note that we did not design Trellis Driver to behave like MPI or be a replacement for MPI. MPI is a standard for exchanging messages in distributed systems, whereas Trellis Driver is a Java interface that links applications with an actively running Trellis metascheduler. However, analogies can be made between the semantics of MPI functions as seen by the programmer, and the internal mechanisms of Trellis Driver, which help provide this module's visible semantics.

Command lines issued by a sender (producer thread) can be thought of as messages sent to a receiver (consumer thread), which oversees the execution of that command in the Trellis environment. MPI manages communication in the form of messages sent between processes running in separate memory spaces. Trellis Driver manages communication in the form of command lines passed between separate Java threads executing in the same virtual machine instance. Table 4.4 provides further descriptions of the six Trellis Driver API routines that handle this command line communication, and also indicates to which MPI routine each one corresponds.

4.4 Concluding Remarks

In this chapter, we presented our solution for porting driver-based Java applications, such as PA, to metacomputing. We described the functionality and the architecture of the Trellis Driver Java module that is used in place of the `Runtime.exec()` facility. We explained the Trellis placeholder scheduling mechanism, which provides load balancing of jobs across all hosts.

Having integrated PA with metacomputing through Trellis Driver, we now wish to study the cost of moving data over the network connecting the individual metacomputer hosts. When the metacomputer spans a local area network (LAN), the metascheduler's policy towards data locality is not critical to application performance. In Chapter 5, we experimentally confirm this assertion. When the metacomputer spans a wide area network (WAN), we expect and observe, in Chapter 6, that the data movement overhead becomes high enough to necessitate the use of a scheduling policy that co-locates jobs and data. We develop a new scheduling policy for Trellis that is shown experimentally to reduce data movement overheads by placing jobs on hosts that hold their input data. Thus, the next two chapters serve to demonstrate the substantial performance benefit of running PA's workflow over a metacomputer, whether the metacomputer deployed spans a LAN or a WAN.

Trellis Driver Function	MPI Function	Semantic Similarities
`start()`	`MPI_Init()`	Initializes the execution environment. In Trellis Driver, the environment depends on parameters preset by the application such as buffer size and the number of consumers. In MPI, the environment depends on command line settings such as stated process priority and debugging settings.
`stop()`	`MPI_Finalize()`	Terminates the execution environment. No more jobs may be started or messages sent after a call to either of these functions. A single call must be made at the end of the master or driver process to properly clean up all program threads.
`execSynch()`	`MPI_Send()`	Starts a synchronous message send operation. The caller is blocked until it receives notice that the operation has completed. In Trellis, a message constitutes a command line and gets processed by a consumer thread, which executes the specified job in the underlying metacomputer. In MPI, a message constitutes a chunk of program data that is processed and acknowledged by a designated recipient.
`exec()`	`MPI_Isend()`	Starts an asynchronous message send operation. The caller returns right away and is given a handle through which they may verify the result of this operation later. In Trellis, the handle is a numeric key that refers to a job in the buffer. In MPI, the handle is a request object that is modified when the status of the communication changes.
`waitForOne()`	`MPI_Wait()`	Provides a synchronization barrier for one outstanding message, which may be either an unfinished Trellis job or an MPI communication that is still active.
`waitForAll()`	`MPI_WaitAll()`	Provides a synchronization barrier for all outstanding messages, whether they are unfinished Trellis jobs or active MPI communications.

Table 4.4: Semantic Comparison of Trellis Driver and MPI methods.

Chapter 5

Trellis Driver: Implementation and Empirical Evaluation

Chapter 4 presented an illustrative example of how Proteome Analyst (PA) uses the newly-developed Trellis Driver module to run multiple workflow jobs concurrently over a Trellis metacomputer. The preceding chapter then outlined Trellis' placeholder scheduling mechanism. We then gave an overview of Trellis Driver's bounded buffer architecture, and explained the intrinsic producer-consumer data flow pattern. Finally, we annotated the Trellis Driver application programming interface (API).

In this chapter, we provide full implementation details on the Trellis Driver Java module. We describe the computational flow within this software layer throughout a job's lifetime. In our running example, we follow the pathway of function calls that are triggered when the PA application specifies the properties, submits, and collects the results of BLAST jobs through Trellis Driver. After describing the functionality of each class in Trellis Driver, we explain the two techniques offered for grouping multiple workflow jobs into a common `mqsub` process, which address two different inherent scheduling overheads.

We then present empirical results from our test case of PA. We observe that while `mqsub` imposes significant latency in the scheduling of jobs, batching many jobs of the same type together successfully amortizes `mqsub` latencies, leading to linear speed-up of the BLAST phase. The relationship between the number of placeholders, the number of jobs batched, and load balancing is then explained through an illustrative example. Finally, we examine the difference in speed-up trends for the BLAST and Parsing phases.

5.1 Trellis Driver: Implementation as a Java Package

Trellis Driver consists of four distinct Java classes that are grouped into a common package: `TrellisDriver`, `ConsumerThread`, `BoundedBuffer`, and `BufferEntry`. Figure 5.1 provides an illustrative diagram of the interaction between objects from these four classes and the

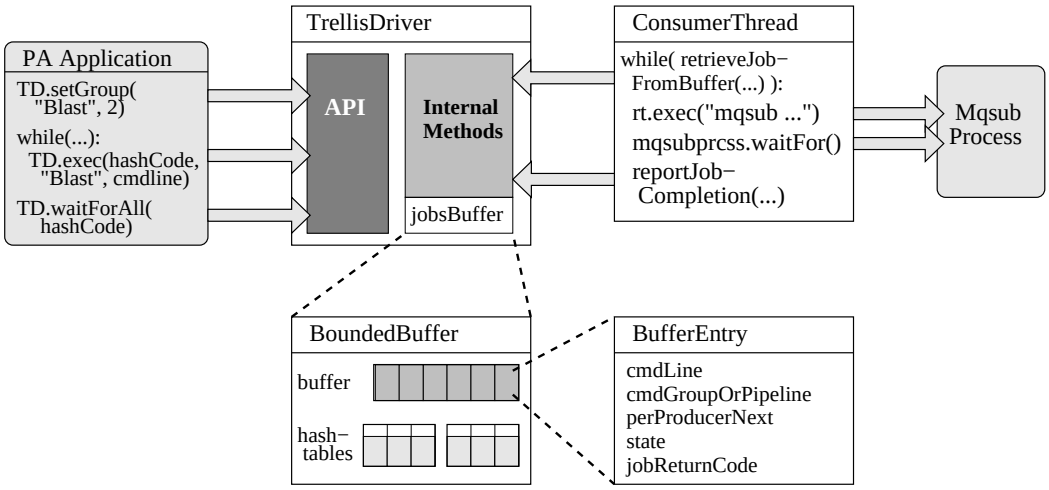


Figure 5.1: Pathway of Function Calls During Execution of Jobs in Trellis Driver

external components, during the execution of jobs. Arrows in the figure represent invocations of one object's methods by the code within another object. Dotted lines represent an expanded view of a component from an instance of a certain class, such as the buffer for storing job information that is contained in the singleton `TrellisDriver` object. We abbreviate `TrellisDriver` with `TD`.

Trellis Driver was previously described as a software module that integrates applications with the Trellis metacomputing system. To clarify this conceptual definition, the left side of Figure 5.1 shows the part of the PA application that utilizes Trellis Driver's job scheduling services. More specifically, the box in the upper left corner shows calls to the API functions, such as `TD.setGroup()`, discussed in the previous chapter. The right side of the figure shows one `mqsub` process that is created when a `ConsumerThread` instance retrieves one set of jobs from the buffer, and sends these jobs to Trellis for execution. The `mqsub` process shown here lives until all the latest jobs sent to Trellis finish executing.

An instance of each of the PA application and `mqsub` utility are shown in Figure 5.1 merely to show the external programs that are involved in the execution of workflow jobs. We avoid discussing the internal workings of either PA or `mqsub` here. We presented the computational pipelines for the training and prediction use cases of PA in Section 2.1, and explained how `mqsub` adds jobs to a metaqueue so that these jobs are later assigned to placeholders in Section 4.2. We now present a step-by-step description of the processing of a new job by Trellis Driver.

5.1.1 Functional Pathway During Job Execution in Trellis Driver

This section provides a walkthrough of all the tasks involved in executing the workflow jobs from the BLAST phase in the PA application. As discussed in Section 2.1, both the training and prediction use cases of PA entail running BLAST as the first step in the analysis pipeline of an input protein.

On the left side of Figure 5.1, we see some of the call sites to key API functions from the part of PA that interacts with Trellis Driver. Although not shown in this diagram for the sake of compactness, it is always necessary to call `TrellisDriver.getDriver()` to obtain a reference to the single `TrellisDriver` instance associated with the current application. This reference is stored in a program variable, represented as `TD` in the PA box in Figure 5.1, and is then used to invoke API methods in all subsequent Trellis Driver operations. The application may also set the buffer size, number of consumers, and metaqueue name, and must call the `start()` function to enable the job processing mechanism within Trellis Driver. The call sites to the relevant configuration functions are not shown in the interest of compactness.

To enable the reduction of job scheduling overheads within Trellis Driver, PA must take the important step of registering the BLAST job group. The `TD.setGroup()` call informs Trellis Driver of a new job group, entitled "BLAST", for which multiple jobs of this common type are to be bundled together in a single `mqsub` process, and therefore, run on the same machine. Section 5.3.1 explains how batching together multiple jobs of the same type amortizes scheduling overheads. In this example, the PA application specifies a *batching factor* of two (2), which means that two BLAST jobs are to be grouped into a common `mqsub` process. The programmer may, in fact, choose any value within a certain legal range, which we discuss later in Section 5.2.

After registering the BLAST job group, the PA application then iterates over all proteins in the input proteome and starts a BLAST analysis on each one. This is done through the `TD.exec()` invocations within the `while()` loop, shown in the centre of the PA box in Figure 5.1. Trellis Driver stores the command line of each incoming BLAST job in its own buffer entry, which is then protected from being overwritten with another command line until the application has acknowledged the completion of that particular job. Details on the transitioning of buffer entry states throughout a job's lifetime are given in Section 5.2.

The first argument in the call to `TD.exec()` shown in this example, `hashCode`, is a numeric value that uniquely identifies the producer. One source of unique numeric values for individual producers is the hash code of the Java virtual machine (JVM) thread encompassing this producer. Recall, from Figure 4.4, that Trellis Driver keeps track of which jobs belong to which producers by setting "per-producer next" links between the relevant buffer entries.

The second argument given to `TD.exec()` is a string containing the BLAST job group label. Trellis Driver uses this string as a key in the job group hashtable to find the batching factor for BLAST jobs (in this case two). When storing command lines of BLAST jobs in the buffer, Trellis Driver forms sublists of two buffer entries containing BLAST command lines. Consumer threads can then iterate over these sublists to recruit two BLAST jobs at a time for execution.

The third and final argument to `TD.exec()` is the command line itself. In PA, BLAST jobs are specified by the absolute path of the BLAST binary followed arbitrary arguments, which specify properties of the BLAST search.

Execution Parameter	Lower	Upper
Buffer Size	100	10,000
Number of Consumers	2	512
Job Group Batching Factor	1	*Buffer Size*
Job Pipeline Length	1	*Buffer Size*

Table 5.1: Upper and Lower Bounds on Trellis Driver Execution Parameters

After starting BLAST analyses for all proteins in the input proteome, the PA code calls `TD.waitForAll()`, giving the hash code of the calling producer as the only argument. Trellis Driver then walks the corresponding per-producer job list, blocking at each buffer entry in that list until the associated BLAST job has been marked as completed by a consumer.

Including the JVM thread-specific hash code in all `TD.exec()` calls when submitting jobs, and providing this same hash code in the subsequent `TD.waitForAll()` call, allows multiple producers to execute separate pieces of an application's workflow without interfering with each other's jobs. Although the above example shows only one producer, there could in fact be multiple copies of this same PA code running BLAST analyses for distinct proteomes concurrently.

5.2 Trellis Driver Classes

1. **`TrellisDriver`**

   The `TrellisDriver` object handles all communication with the client Java program, as well as with the other objects in the Trellis Driver package. As described previously, programmers must first obtain a reference to the singleton `TrellisDriver` object belonging to their application, and then may invoke API methods on this object as needed to run external jobs. The `TrellisDriver` object also stores the bounded buffer that holds the command lines, and manages the pool of consumer threads that process jobs.

   As a pre-processing step, an input validity check is performed in the four configuration functions of `setBufferSize()`, `setConsumersCount()`, `setGroup()`, and `createPipeline()`. Table 5.1 shows the fixed ranges for each of these execution parameters. Although currently not adjustable by the user, the parameters shown could easily be made so.

2. **`ConsumerThread`**

   `ConsumerThread` objects communicate with the underlying metacomputer by continually pulling command lines out of the buffer and submitting them to the desired Trellis metaqueue with `mqsub`. During workflow execution, `ConsumerThread` objects follow a basic cycle of fetching, executing, and marking jobs as complete.

3. **`BoundedBuffer`**

   The single `BoundedBuffer` instance holds the command lines of all jobs issued to Trellis

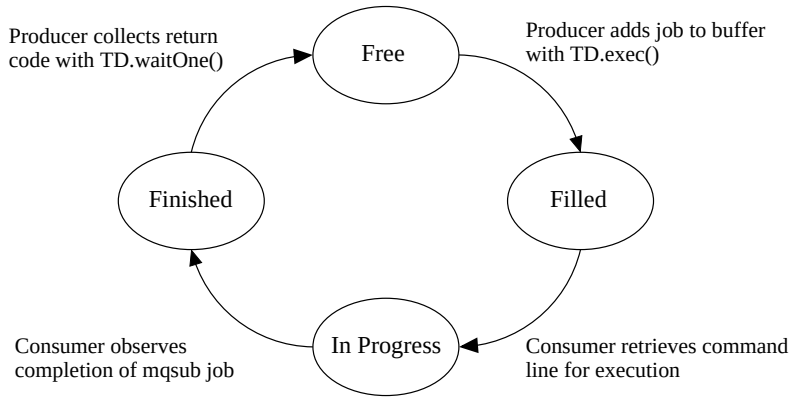


Figure 5.2: State Transitions and Associated Operations for Bounded Buffer Entries

Driver. For performance scalability, the `BoundedBuffer` object uses a buffer that is of fixed size (i.e., bounded) to decouple the number of consumers from the number of producers.

Each incoming command line is placed in a buffer entry that was previously marked as Free. Trellis Driver keeps up-to-date linked lists of buffer entries in each of the four states, which are outlined in the subsection immediately following. All buffer manipulation functions are synchronized Java methods, so mutual exclusion is enforced.

The `BoundedBuffer` object also contains hashtables for storing information on producer threads, job groups, and job pipelines.

4. **`BufferEntry`**

   An array of `BufferEntry` instances implements the bounded buffer for storing command lines. Individual `BufferEntry` objects contain multiple fields for recording all pertinent job information. When required, the label of the job group is stored. The job's command line and the return code of the `mqsub` process that passes this command line to the Trellis metascheduler are also stored.

   Figure 5.2 provides a state transition diagram for a bounded buffer entry. In this example, one producer is executing one job asynchronously. When the producer invokes `TD.exec()`, the new job's command line is written to a previously Free buffer entry, which then moves to the Filled state. A consumer thread retrieves the command line and passes it as an argument to the `mqsub` utility for execution, changing the state of that entry to InProgress. The `mqsub` process passes its return code back to the consumer upon the job's completion. The consumer records this return code in the buffer entry, which is moved to the Finished state. Finally, the producer calls `TD.waitOne()` to synchronize with the completion of this individual job and collect the `mqsub` exit code. The buffer entry's state then moves back to Free.

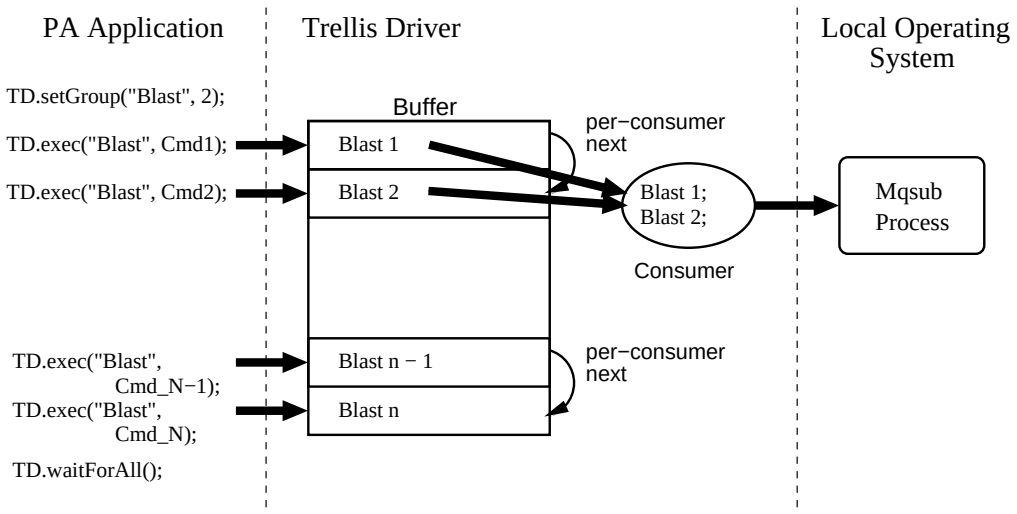


Figure 5.3: Homogeneous Batching of BLAST Jobs Within Trellis Driver

5.3 Batching of Multiple Jobs

The following subsections describe our two techniques for batching jobs of the same and different types, respectively, into a common `mqsub` process. The two batching strategies described here amortize different sources of overhead, both of which are inherent in workflow scheduling.

5.3.1 Job Groups: Homogeneous Batching of Jobs

In PA, the short runtimes of BLAST and Parsing jobs make for poor job granularity. There is an inherent overhead in moving a command line through Trellis Driver and in starting the associated `mqsub` process. Likewise, there is an overhead for a `mqsub` process to terminate and pass its exit code back to Trellis Driver. Aggregating several BLAST or Parsing jobs into one `mqsub` process can greatly amortize the Trellis Driver and `mqsub` latencies.

Trellis Driver supports the grouping of multiple jobs from a common workflow phase to reduce scheduling overheads. We refer to this job batching strategy as *homogeneous batching* since the jobs bundled together by Trellis Driver are always of the same type (i.e., they are invocations of the same external program).

Figure 5.3 illustrates the flow of BLAST command lines within Trellis Driver when homogeneous batching is used. The left side of the diagram provides a snapshot of PA code from the BLAST phase. Here, the workflow is specified in terms of control flow, so PA starts all BLAST jobs, invokes a workflow barrier function to await their completion, and then repeats this process for the Parsing phase.

When PA calls `TD.setGroup()`, Trellis Driver registers the BLAST job group by adding an entry to the job group hashtable (not shown), which maps the BLAST label to the given batching

factor of two. The `TD.exec()` calls send the BLAST command lines to Trellis Driver, which stores them in separate buffer entries, as seen in the centre of the diagram. A series of "per-consumer next" links is maintained to form job groups of the desired size.

On the right, we see one consumer recruiting two BLAST command lines from the buffer and passing them to an `mqsub` process. The command line recruiting proceeds in three steps:

1. The consumer reads the BLAST job group label from the top buffer entry shown and consults the job group hashtable to find that the desired batching factor is two.

2. The consumer follows the per-consumer next links until it has grabbed two BLAST jobs, or until a fixed timeout has expired.

3. The consumer passes the concatenation of the two BLAST command lines to `mqsub`.

Having a timeout prevents the starvation of an incomplete job group at the end of a workflow phase. For instance, if PA executes 1,000 BLAST jobs and the BLAST batching factor is 16, this produces 62 groups of 16 – accounting for the first 992 jobs – and one remaining smaller group of 8 BLAST jobs. With no timeout, the consumer recruiting the last few BLAST command lines would wait indefinitely for 8 more jobs that would never come.

When a placeholder is given a job bundle (i.e., a set of jobs grouped together by homogeneous batching), it receives one command string that consists of multiple BLAST command lines separated by semi-colons. As per Unix convention, placeholder scripts execute these command lines individually. With homogeneous batching, the exact order of job execution within a single job bundle is irrelevant, since the jobs come from the same workflow phase and are, therefore, mutually independent.

5.3.2 Job Pipelines: Heterogeneous Batching of Jobs

Although Trellis Driver and `mqsub` impose significant overhead for jobs with short runtimes, the cost of transferring data between jobs from successive pipeline stages that are run on different hosts can also entail considerable overhead. The late binding of jobs to placeholders in Trellis means that BLAST and Parsing jobs from a common pipeline can be run on different hosts, particularly at high placeholder counts. In such cases, the homologue file outputted by the BLAST job must be transferred to the machine where the successor Parsing job runs.

To avoid data movement overheads, Trellis Driver supports the bundling of jobs from different workflow phases. We refer to this strategy as *heterogeneous batching* since the jobs bundled together are of distinct types (i.e., they are invocations of different external programs).

Figure 5.4 illustrates the flow of protein pipeline jobs within Trellis Driver when heterogeneous batching is used. The left side of the diagram shows a code extract from the overlapped BLAST and Parsing phases. Here the workflow is specified in terms of data flow, so PA issues the BLAST

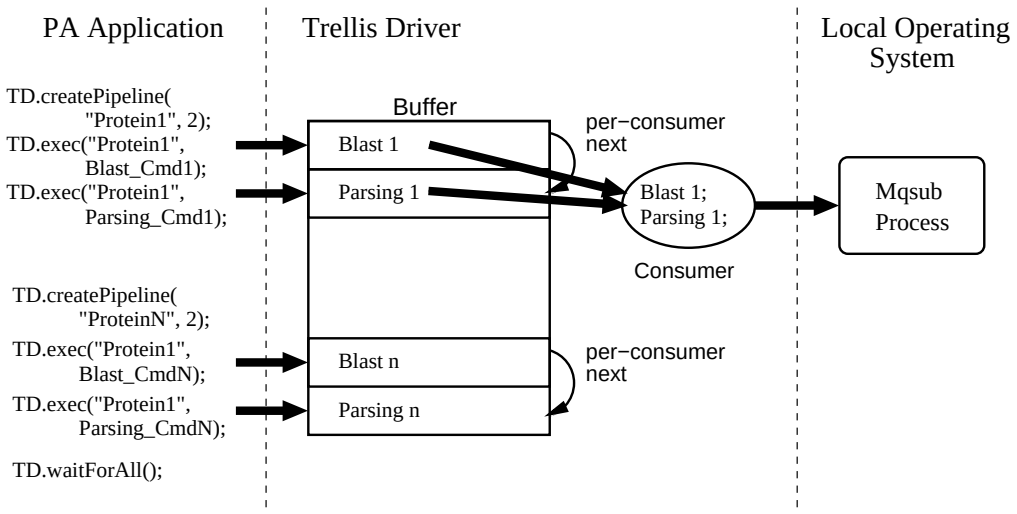


Figure 5.4: Heterogeneous Batching of Protein Pipelines Within Trellis Driver

job and the Parsing job from the first pipeline, then the BLAST and Parsing jobs from the second pipeline, and so on.

Before executing each pipeline, however, PA calls `TD.createPipeline()` to register the next job pipeline with Trellis Driver. This function call adds a new entry to the job pipeline hashtable (not shown), which maps pipeline labels to their given lengths, which are always two in this workflow. The `TD.exec()` calls add the command lines of the BLAST and Parsing jobs to the buffer, as seen in the centre of the diagram. The per-consumer next links are set according to the specified pipeline length.

The consumer recruits command lines in a manner similar to that used in homogeneous batching. The difference is that there is no timeout for recruiting jobs. Thus, a consumer that finds one BLAST job, which is part of a two-stage pipeline, will wait indefinitely for the accompanying Parsing job. This approach is used because the application is expected to supply the correct number of jobs for every pipeline.

Data flow dependencies between the BLAST and Parsing jobs are respected, since the command strings given to placeholders always specify the command lines in the same order in which the corresponding jobs were issued. For instance, in Figure 5.4, the consumer constructs a command string containing the first BLAST command line, followed by a semi-colon, and then the first Parsing command line. Following Unix convention, the placeholder receiving the command string shown in the above figure executes BLAST 1 before Parsing 1.

5.4 Experimental Results

In Section 2.1, we explained the three stages of PA's job pipelines. Stages 1 and 2, BLAST and Parsing, respectively, are identical in both the training and prediction use cases. Table 2.1 presented

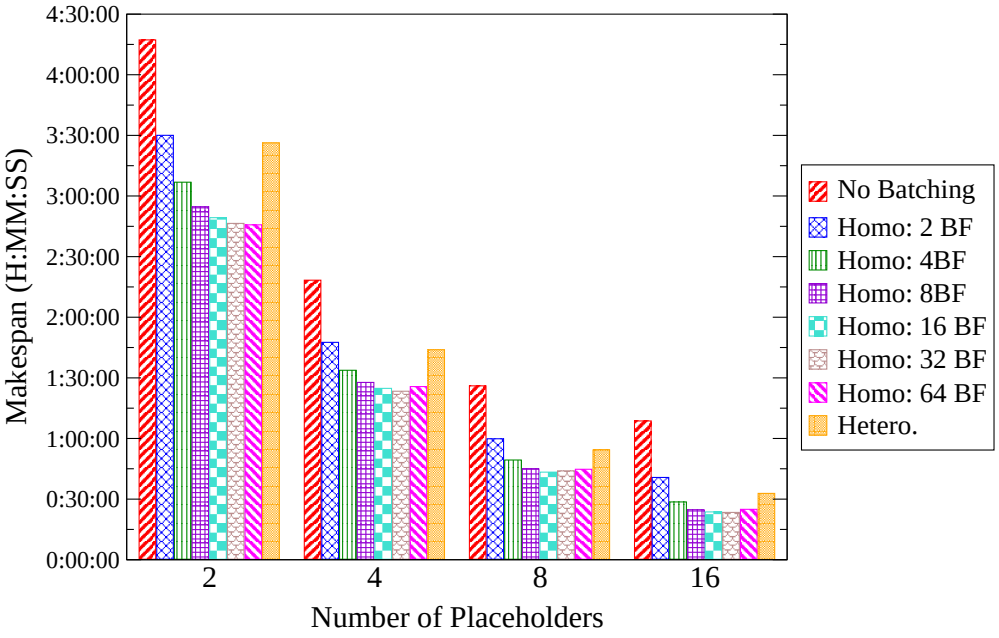


Figure 5.5: PA-Trellis Makespan with Varying Batching Strategies
BLAST and Parsing Phases Run Across a **LAN**

the runtimes of all four phases in our PA test case, in which we trained and validated a new classifier. The BLAST phase had by far the longest runtime. Although Parsing had a much shorter runtime, like the BLAST phase, Parsing is also embarrassingly parallel, meaning any number of jobs from either Stage 1 or Stage 2 may execute concurrently. Accordingly, we parallelized both the BLAST and Parsing phases of PA.

Experimental results presented in this chapter show:

1. Homogeneous batching is more effective than heterogeneous batching (Figure 5.5). The former strategy reduces makespan to a much greater degree than the latter, suggesting that in this PA workflow, `mqsub` imposes greater overhead than data transfer.

2. Linear speed-up of the BLAST phase is approached or achieved at all placeholder counts.

3. The Parsing phase actually slows down when parallelized at low placeholder counts. Even at high placeholder counts, speed-ups are nowhere close to linear.

We parallelized the Parsing phase of PA the same we did for the BLAST phase. Recall the code snapshots from the original and parallel PA (PA-Trellis), presented in Section 4.1, in the latter of which we replaced `Runtime.exec()` calls with `TrellisDriver.exec()` calls.

Num. PHs	No Batching	Hetero. Batching	Homo. Batching					
			2 BF	4 BF	8 BF	16 BF	32 BF	64 BF
2	4:17:14	3:26:17	3:30:02	3:06:50	2:54:43	2:49:16	2:46:27	**2:45:44**
4	2:18:20	1:43:55	1:47:35	1:33:47	1:27:44	1:24:48	**1:23:22**	1:25:42
8	1:26:03	1:00:32	0:59:52	0:49:21	0:44:55	**0:43:21**	0:43:57	0:44:46
16	1:08:41	0:40:14	0:40:42	0:28:36	0:24:45	0:23:29	**0:23:24**	0:24:56

Table 5.2: Combined BLAST and Parsing Makespan for Varying Batching Strategies and Factors (H:MM:SS)
Original (Sequential) BLAST and Parsing Makespan was **4:59:59**

To benchmark PA-Trellis against the original PA, we trained a new classifier with the same input proteome of 3,916 sequences and 1,531 features that we used in our test case, presented in Section 2.1. In all experiments discussed below, PA-Trellis was run over a cluster of Linux machines, connected via Fast Ethernet. Thus, the underlying metacomputer spanned a local area network (LAN). Each constituent host was a Linux box that had two AMD Athlon MP 1800+ processors, 1.5 GB main memory, and ran Red Hat 7.1. The BLAST executable and the Swiss-Prot database were replicated on all metacomputer hosts. The sequence text files (i.e., the BLAST inputs), were stored on an NFS-mounted volume, making for transparent and fixed-time data access across all hosts. There was no need to use the Trellis File System (Trellis FS) for data access.

5.4.1 Homogeneous vs. Heterogeneous Batching

Figure 5.5 shows the combined makespan for the BLAST and Parsing phases of PA-Trellis with varying batching strategies and factors. The "No Batching" series represents a base case where the BLAST and Parsing jobs were all run individually, and thus there was no amortization of `mqsub` or data movement overheads. Here, batching factor is abbreviated as `BF`.

We immediately see that either batching strategy decreases makespan noticeably against the base case at all placeholder counts. For clarity, Table 5.2 displays the values of the combined makespan of the BLAST and Parsing phases. The label `PH` is an abbreviation for placeholder. These numbers show that homogeneous batching reduces makespan considerably more than does heterogeneous batching. Already at a batching factor of 4, homogeneous batching wins easily against heterogeneous batching in terms of makespan. For instance, at 16 placeholders, the heterogeneous makespan is 0:40:14, which is a 41.4% reduction of the base case makespan of 1:08:41. However, the homogeneous makespan with a batching factor of only 4 is 0:28:36, which is a much larger 58.4% reduction against the base case makespan. Similar trends can also be seen with fewer than 16 placeholders. At all higher batching factors, the homogeneous strategy outperforms heterogeneous even more decisively, at all placeholder counts. Thus, `mqsub` appears to be a far greater source of overhead than data movement, irrespective of the number of placeholders.

With homogeneous batching, there are diminishing returns in makespan improvements, which eventually go negative, as the batching factor increases. This is evident in the flattening out of

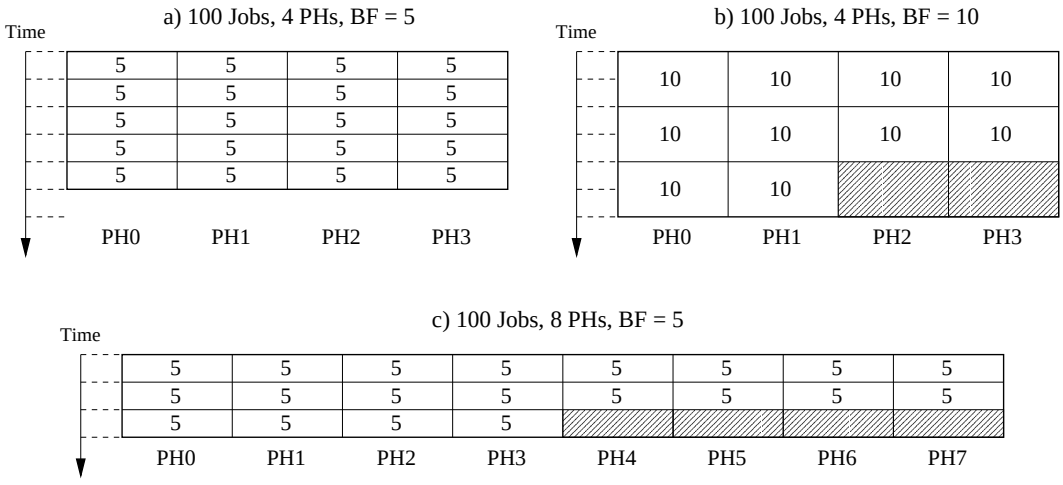


Figure 5.6: Load Balancing with Varying Placeholder Counts and Batching Factors

successive makespan bars in Figure 5.5 (except for the heterogeneous bars on the far right). Note that homogeneous makespan is reduced considerably when going from a batching factor of 2 to 4, but only slightly, if at all, when going from a batching factor of 16 to 32. The batching factor that produces the lowest makespan varies with the number of placeholders. From Table 5.2, it appears that larger batching factors lead to better performance at lower placeholder counts. We discuss this phenomenon in the following subsection.

5.4.2 Causes of Load Imbalance

Having established that homogeneous batching works best for our test PA workflow, we now examine the different characteristics of the BLAST and Parsing phases. We begin by investigating the relationship between the number of placeholders, the batching factor, and makespan.

As an illustrative example of inherent load imbalance in some situations, Figure 5.6 shows three different load balancing scenarios, each of which has 100 jobs but a distinct combination of placeholder count and batching factor. Columns in the diagrams illustrate the assignment of job bundles to specific placeholders, which are labeled at the bottom. Numbers in the boxes represent the size of the job bundles. Rows represent job assignment "rounds", in which the Trellis scheduler hands off one bundle of jobs to each placeholder. Time proceeds downwards in the diagrams, meaning that the first row represents the first round of job assignments, the second row the second round, and so on. Dashed lines from the time arrow to the box of job assignments provide a visual measurement aid.

Figure 5.6(a) shows a scenario in which the number of placeholders times the batching factor

Num. PHs	2 BF	4 BF	8 BF	16 BF	32 BF	64 BF
2	2:47:08	2:35:35	2:29:09	2:27:11	2:25:38	**2:25:12**
4	1:23:26	1:17:15	1:14:26	1:12:53	**1:12:32**	1:15:19
8	0:42:49	0:39:07	0:37:26	**0:37:07**	0:37:58	0:38:56
16	0:23:33	0:20:09	**0:19:24**	0:19:41	0:19:51	0:21:33

Table 5.3: BLAST Makespan with Varying Homogeneous Batching Factors (H:MM:SS)
Original (Sequential) BLAST Makespan was **4:51:26**

Num. PHs	2 BF	4 BF	8 BF	16 BF	32 BF	64 BF
2	1.74	1.87	1.95	1.98	2.00	**2.01**
4	3.49	3.77	3.92	4.00	**4.02**	3.87
8	6.81	7.45	7.79	**7.85**	7.68	7.49
16	12.38	14.47	**15.03**	14.81	14.68	13.52

Table 5.4: Speed-up of Parallel BLAST over Sequential BLAST with Varying Homogeneous Batching Factors

evenly divides the number of jobs, and thus perfect load balancing is achieved. In Figure 5.6(b), when the batching factor is 10, there is some load imbalance since the placeholders do not all run the same number of job bundles. The workflow's execution takes longer in this second scenario, requiring six time units as opposed to the five required in the first scenario.

Figure 5.6(c) shows the same workflow as in (a) but with eight placeholders instead of four. Again, we see some load imbalance, since the placeholders run different numbers of job bundles. In this case, doubling the number of placeholders does not cut turnaround time in half (i.e., by 50%). The workflow execution takes three time units as opposed to five, or 60% of the original runtime.

Admittedly, if there were 120 jobs instead of 100 in the above example, perfect load balancing would be achieved with a batching factor of 10, or with 8 hosts. However, the above example is intended to show that as the job bundle granularity or the number of placeholders increases, it becomes less likely to be able to assign each placeholder the same number of jobs. From the three scenarios illustrated, we see that either a high number of placeholders or a large batching factor can throw off load balancing, thereby prolonging makespan.

5.4.3 Differences in BLAST and Parsing Phases

Table 5.3 displays the BLAST makespans for the PA-Trellis workflow introduced in the previous section, with the best makespans obtained at each placeholder count highlighted. Note that the optimal homogeneous batching factor decreases as the number of placeholders increases. We believe this is due to the load balancing trends discussed previously. At 2 placeholders, when half the workflow jobs (50%), on average, are assigned to each placeholder, a batching factor of 64 produces the shortest makespan. At 16 placeholders, when only one sixteenth of the workflow jobs (roughly 6%), on average, are assigned to each placeholder, the optimal batching factor is 8. Thus, the

Num. PHs	2 BF	4 BF	8 BF	16 BF	32 BF	64 BF
2	0:42:54	0:31:16	0:25:34	0:22:05	0:20:49	**0:20:32**
4	0:24:09	0:16:31	0:13:19	0:11:56	0:10:50	**0:10:22**
8	0:17:03	0:10:14	0:07:29	0:06:14	0:05:58	**0:05:50**
16	0:17:09	0:08:27	0:05:22	0:03:59	0:03:33	**0:03:23**

Table 5.5: Parsing Makespan with Varying Homogeneous Batching Factors
(H:MM:SS)
Original (Sequential) Parsing Makespan was **0:08:33**

Num. PHs	2 BF	4 BF	8 BF	16 BF	32 BF	64 BF
2	0.20	0.27	0.33	0.39	0.41	**0.42**
4	0.35	0.52	0.64	0.72	0.79	**0.82**
8	0.50	0.84	1.14	1.37	1.43	**1.47**
16	0.50	1.01	1.60	2.15	2.41	**2.53**

Table 5.6: Speed-up of Parallel Parsing over Sequential Parsing with Varying Homogeneous Batching Factors

combination of large job bundles and a high number of hosts prolongs makespan. This observation is consistent with our earlier illustrative load balancing scenarios.

Measuring speed-up ratios for the BLAST and Parsing phases allows us to assess how effectively we are amortizing the overheads of `mqsub` and data transfer. Table 5.4 shows speed-ups for the BLAST phase. Not surprisingly, as the batching factor increases and hence, the overheads of `mqsub` are amortized more, speed-ups improve. For the 2-placeholder case, we achieve linear speed-up with batching factors of 32 and 64. For the 4-placeholder case, linear speed-up is achieved with batching factors of 16 and 32. For the 8 and 16-placeholder cases, we fall just short of linear speed-up. We believe that contention for the common Trellis Command Line Server (CLS), which must process requests for a larger number of placeholders, is the reason for this slightly weaker performance. We explore this issue further in Chapter 6.

Table 5.5 displays the Parsing makespans for the PA-Trellis workflow. Table 5.6 shows the corresponding speed-ups. Immediately, we note that parallelizing this phase does not produce any worthwhile speed-up. At 2 or 4 placeholders, there is actually an increase in the Parsing times, regardless of how high we set the batching factor. Even at 8 or 16 placeholders, when we see some reduction in phase time at higher batching factors, the speed-ups obtained are quite low, well short of linear.

We believe this poor speed-up is simply due to the small granularity of the Parsing jobs. The short runtimes for individual Parsing jobs, which we estimate to be within the 0.5 to 0.7 second range in Chapter 6, makes for a low ratio of computation to communication in this phase. Here, computation refers to the time spent actually executing Parsing jobs, and communication refers to all overheads of Trellis Driver, `mqsub`, and even placeholders contacting the CLS.

5.5 Summary of Results

In Chapter 4, we outlined Trellis Driver's scalable, bounded buffer architecture as well as its API, which is based on that of `Runtime.exec()`. The code extracts from the original PA and PA-Trellis demonstrated how Trellis Driver functions as a drop-in replacement for `Runtime.exec()`, with the added feature of workflow barrier functions.

In this chapter, we described the implementation of Trellis Driver. We followed the computational flow as Trellis Driver processes all the BLAST jobs launched by the PA application. We explained our two different job batching strategies, which address two different inherent scheduling overheads. Our empirical results showed that the overheads of `mqsub` and data movement can be amortized by batching. In particular, homogeneous batching leads to linear speed-up of the BLAST phase in our test case. The speed-up values showed that the optimal batching factor varies with the number of placeholders; we believe this is due to load balancing trends. The Parsing phase obtains low speed-ups due to small job granularity.

Chapter 6

Data-Conscious Scheduling Policy: A Simulation Study

This chapter investigates the circumstances under which the Proteome Analyst (PA) application can benefit from a scheduling policy that assigns jobs to hosts that hold their input data. We implement and refine our new scheduling policy in the context of a simulation study, which provides a controlled environment for easily altering key parameters such as file sizes and communication overheads.

Chapter 5 presented empirical results showing the effectiveness of homogeneous job batching at amortizing job scheduling overheads. We saw that linear speed-up for the BLAST phase of PA is achieved at higher batching factors, even though the job scheduling strategy used by Trellis is First Come First Served (FCFS), which does not consider data movement.

The results from the previous chapter demonstrated that good speed-ups are achievable without data consciousness in scheduling. However, three important questions are left unanswered:

1. While PA achieves impressive performance gains when the metacomputer spans a local area network (LAN), what are the gains when the metacomputer spans a wide area network (WAN)?
2. What are the effects of using scheduling policies other than FCFS within the metacomputer?
3. What influence do application and metacomputing parameters, such as file sizes or contention for the command line server (CLS) among multiple placeholders, have on performance?

This chapter explores the above questions. We first explain the design of our new Data-Conscious (DC) scheduling policy. We then determine the cases in which co-locating jobs and their input data improves performance significantly. Finally, we explore the impact on performance of parameters such as job batching strategy and data transfer rate.

6.1 Why Use Simulation?

The previous chapter presented results from experiments that were run in a real setting involving a LAN-spanning metacomputer whose resources were all contained within a single administrative

domain. Since metacomputing works across multiple administrative domains that may be geographically dispersed, we wish to measure the performance of PA-Trellis on a metacomputer that spans a WAN as well. However, WAN environments are hard to regulate for our own use. Many factors that can impact performance, such as sustainable cross-domain bandwidth, lie beyond our control. Obtaining exclusive access to all the resources over which we wish to distribute our metacomputer can be difficult since it requires the cooperation of all parties who share those resources.

A more practical alternative is to use simulation to model the execution of PA-Trellis over a WAN-spanning metacomputer. A simulator program allows us to reproduce the PA-Trellis application and its execution environment in sufficient detail, while controlling all factors that influence performance, including data transfer times. Additionally, we can adjust the settings of the relevant model parameters to explore their theoretical impact on performance. For example, we can inflate the file sizes to see how well our new scheduling policy handles data-heavy workflows.

Two more properties, common to all simulators, greatly aid our work. First, virtual experiments executed by a simulator are carried out in virtual time and not real time. Thus, we can run a range of simulated PA-Trellis experiments in a few seconds of real time where equivalent experiments with the actual application would take hours to complete. Second, the important scientific objective of reproducibility of results is easily met in simulation. The deterministic nature of many simulators, including the one we develop, means that precisely the same performance results are observed in successive trials with identical inputs.

6.2 Smurph Simulation

We use the Smurph simulation package [12], developed at the University of Alberta, to model the execution of the PA application's workflow over a Trellis metacomputer. The Smurph utility is designed for writing discrete event simulations of packet-based network protocols. We found Smurph to be easily adaptable for creating an accurate simulation of the PA workflow and the Trellis placeholder scheduling environment.

Figure 6.1 provides a diagram of the flow of control and data between the key objects in our Smurph simulation. The thick grey arrows indicate data flow or the passing of information, such as command lines, between objects. The thin black arrows indicate control flow between objects, typically done through signalling. In our simulation, we model the batch queues on hosts within the metacomputer, which control the execution of local placeholders (also modelled), and the CLS process, which stores information on all jobs injected into the Trellis system via `mqsub` calls.

There is a single Root or master object whose code is executed at the beginning of every simulation. At start-up time, the Root object reads one text file specifying the placeholder-to-host assignments. Based on this information, the Root creates the appropriate batch queue and placeholder objects. The Root then opens another file containing the list of workflow jobs to be executed.

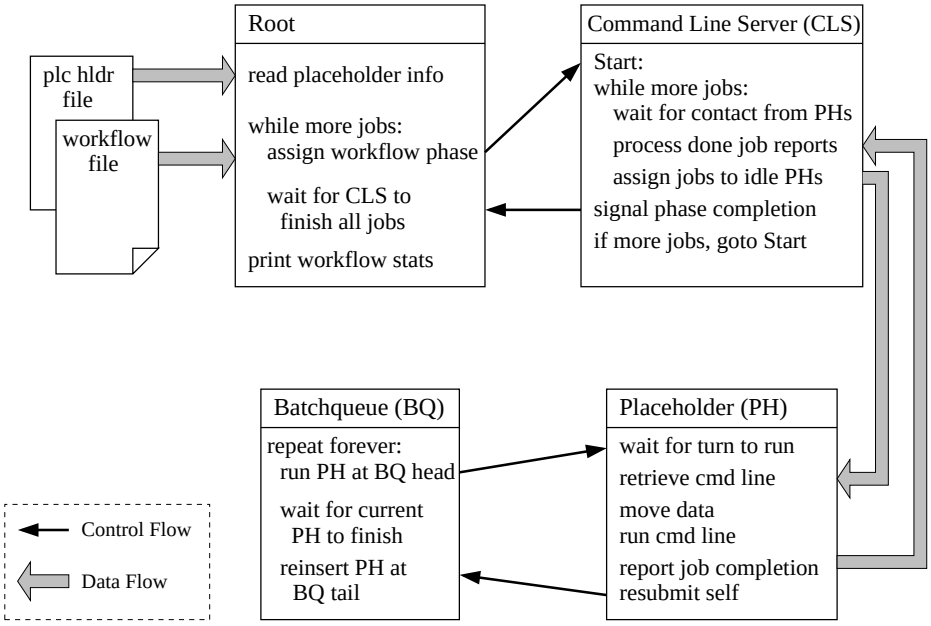


Figure 6.1: Control and Data Flow in Smurph Simulator

Our simulator then enters its main workflow processing loop, which functions in three steps:

1. The Root reads through the list of workflow jobs until it reaches the end of the current phase. For each job read in, the Root submits that job to the appropriate CLS object, creating that CLS object first if necessary. The handing of jobs to CLS instances is our way of simulating the `mqsub` invocations by consumer threads within Trellis Driver.

   In all our simulations, we assume only one user running a single copy of PA. We therefore create only one CLS instance that processes all the workflow jobs.

2. The Root sleeps to allow the single CLS instance to distribute the workflow jobs among the placeholders. When the CLS has recorded the completion of all its jobs, it signals the Root.

3. The Root resumes reading the workflow file at the point just after the end of the previous phase. If there are more jobs listed (i.e., there is another phase in the workflow), the program's execution repeats from Step 1. If the end of the file is reached, the Root prints out workflow execution statistics, including makespan and total data movement, and the simulation ends.

During the execution of a workflow phase, as in any real Trellis setting, placeholders initiate contact with the CLS by sending messages either reporting the completion of jobs or asking for work. First, the CLS processes any outstanding reports of finished jobs and sends acknowledgments to the relevant placeholders. Second, to each placeholder asking for work, the CLS sends a reply message containing the command line of a new job. This activity repeats itself until every job in the

current workflow phase is recorded as complete, at which point the CLS signals the Root and awaits the arrival of a new phase of jobs, as discussed above.

Each batch queue object simulates a local batching system that executes on one host and controls when each placeholder bound to that particular host runs. Local placeholders are, in essence, queued jobs. Batch queues follow a repetitive process of: signalling the placeholder at the front of the queue that it may run; waiting for that placeholder to retrieve the command line of a new job and when necessary the input data, execute the job, and report the job's completion to the CLS; and finally, reinserting that placeholder at the back of the queue.

Placeholder objects model the movement of data files between metacomputer hosts by sleeping a fixed amount of virtual time related to the input data retrieval time as well as to the network type (i.e., LAN or WAN). Likewise, placeholders "execute" jobs simply by sleeping for a number of virtual time units that corresponds to the job's runtime. Thus, the execution of workflow jobs proceeds almost instantaneously in real time since the simulator program has only to advance virtual time.

Included in the job information passed from the Root to the CLS is the inter-job dependency information reflecting the data flow between BLAST and Parsing jobs, for each protein pipeline. The simulator keeps track of the hosts on which particular BLAST jobs run and generate their homologue files. When modelling the actions of our data-conscious scheduler, the simulator considers whether or not the placeholder that is currently asking for work is running on the same host that stores the homologue input file of any ready-to-run Parsing job (i.e., any Parsing job whose prerequisite BLAST job has completed). Parsing jobs whose input is available locally, if bound to the current placeholder, are given higher priorities than those whose input data must be copied from a remote host. We now describe the equations used for job priority calculations within our new scheduling policy.

6.3 Job Priority Calculations

When we introduced the notion of data consciousness in scheduling in Section 1.1.3, we raised two important points: First, although data affinity is the primary objective of our new scheduling policy, there are other job-specific characteristics that are relevant when assigning jobs to hosts within a metacomputer. Second, schedulers usually map each candidate job to a numeric priority value, so that the highest ranked job may be chosen for execution.

In our DC scheduling algorithm, we employ a job priority calculation scheme that examines three properties for each job. The abstract form of our priority equation is:

$$priority = data\ affinity + dependent\ jobs + queue\ time \tag{6.1}$$

Equation 6.1 shows that the priority assigned to a candidate job is the linear sum of three numeric values, each representing one of the relevant job properties. **Data affinity** is a measure of the potential data movement savings. Higher values indicate that little or no data movement is required should

the given job be assigned to the presently-available placeholder as compared to another placeholder in the near future. **Dependent jobs** is the number of other workflow jobs that cannot run until this particular job has completed. Thus, jobs that are bottlenecks in the workflow have higher values for this second term. **Queue time** is a measure of how long this job has been in the metaqueue; higher values indicate a job has waited in the queue for a long time. Including this third term prevents starvation of jobs with heavy data movement requirements or few dependents.

In practice, the priority of a candidate job x for a given current placeholder *currentPH* and a set of upcoming placeholders *futurePHs*, is defined as follows:

$$priority(x,\ currentPH,\ futurePHs) = \\ Data\_Affinity\_R\_C\ (x,\ currentPH,\ futurePHs) + \frac{queue\_time\ (x)}{placeholder\_latency} \tag{6.2}$$

The function *Data_Affinity_R_C* measures the relative cost of data affinity for job x. The relative cost of data affinity is a function of the candidate job x, the current placeholder *currentPH*, and the set of all future placeholders *futurePHs*, which constitutes those placeholders expected to ask for work in the near future. We explain the intuition behind the relative cost metric and how this value is calculated below. Before, however, we discuss two important characteristics of Equation 6.2.

First, note that there is no term for dependent jobs. While the argument of favouring the execution of jobs that are workflow bottlenecks is quite valid in the context of workflow scheduling, the PA workflows we consider do not contain any such bottleneck jobs. The BLAST jobs that comprise the first pipeline stage always have only one dependent job, which is the Parsing job from the same pipeline. The Parsing jobs comprising the second pipeline stage have zero dependent jobs, since in our work we execute the third pipeline stage within the driver process. Rudimentary testing showed that the inclusion of the dependent jobs metric had no significant effect on the performance of any of our workflows. For this reason, we exclude the dependent jobs count from our priority calculations.

Second, observe that queue time is divided by what we call "placeholder latency". Here, queue time is a function of job x and placeholder latency is a constant measuring the time for one-way communication between placeholders and the CLS, in the specific network environment (i.e., LAN or WAN). For example, when a placeholder contacts the CLS to ask for work, there is a finite amount of time for that request to traverse the network and reach the CLS. Placeholder latency is a measure of this communication time.

We perform the division in the second term of the equation to scale down the impact of queue time on final priority. Through preliminary evaluations of job priorities, we found that using queue times directly made the value of this second term too large against the metric of relative cost. Consequently, the goal of co-locating jobs and their data would often be eclipsed by the goal of minimizing the queue times of jobs. Normalizing queue times by placeholder latency brings the values of this job property in line with those of relative cost.

We chose placeholder latency as a scaling factor because we wanted to reduce queue times in

a manner that was conscious of the underlying network. For example, in WAN environments, the dispatching of jobs proceeds more slowly than in LANs due to the increased communication times between the placeholders and the CLS. This slower execution of the workflow reflects the properties of the metacomputing environment and not the actions of the scheduler. The larger placeholder latency value in WAN environments then scales well with the inevitably longer queue times of jobs.

We now explain our metric of relative cost, defined as:

$$Data\_Affinity\_R\_C\ (x,\ currentPH,\ futurePHs) = \\ Min\ (cost\_of(x,\ PH_i)) - cost\_of(x,\ currentPH) \\ \forall PH_i,\ where\ PH_i \in futurePHs = \{PH_1, PH_2, \ldots PH_n\} \tag{6.3}$$

Equation 6.3 defines the relative cost of data affinity for a job x as the minimum cost of assigning x to one of the first n upcoming placeholders (PH_i *for 1 to n*), minus the cost of assigning x to the current placeholder (*currentPH*). In practice, both these quantities are expressed in simulation time units. Conceptually, cost can be thought of as the performance penalty incurred by having a particular placeholder run the given candidate job. A job's relative cost tells us whether it is better to run that job now or later.

The cost of running job x on the current placeholder is calculated in a similar manner to that of running x on an upcoming placeholder. As Equation 6.4 will show, measuring the cost on the current placeholder is a special case of measuring the cost on an upcoming placeholder.

When the policy predicts a higher cost of binding job x to an upcoming placeholder as opposed to the current placeholder, the value of the first term in Equation 6.3 will be greater than the value of the second term. Relative cost will then be positive, indicating that it is advantageous to assign job x to the current placeholder rather than an upcoming placeholder. When the policy predicts a lower cost for assigning job x to an upcoming placeholder, the first term will be smaller. Relative cost will then be negative, indicating that it is desirable to delay the execution of x.

We take the minimum value in the set of job assignment costs for the respective upcoming placeholders so that in our priority calculations we always consider the most promising upcoming placeholder. Thus, even if there is only one upcoming placeholder that can guarantee data affinity for job x, we compare the performance penalty of assigning x to the current placeholder with that of assigning x to the best-fitting upcoming placeholder. Whenever assigning x to the current placeholder entails data movement, we will delay the execution of x when there is any opportunity to place x on a host that stores its data in the near future. This is assuming that the future job assignment occurs soon enough that the delay in execution does not overwhelm the data movement savings.

Following is the equation for calculating the cost of assigning job x to either the current or an upcoming placeholder:

$$cost\_of(x,\ PH_i) = arrival\_time(PH_i) + penalty * data\_transfer(x,\ PH_i) + runtime(x) \tag{6.4}$$

The *arrival_time(*PH_i*)* term in Equation 6.4 is an estimate of when placeholder PH_i will next ask for work. For calculating the cost of running a job x on the current placeholder, $currentPH$ is used instead of PH_i, and arrival time is zero, since the current placeholder can run a job immediately.

Arrival times in Equation 6.4 are measured in the simulator's virtual time units and are always relative to the present. We make our predictions on future placeholder arrivals based on information of past arrivals. We keep track of the average time between successive arrivals for each placeholder, and use this metric to estimate a given placeholder's next arrival time (*arrival_time(*PH_i*)*). Although using average arrival times works in our simulation study, in general, computing future arrival times is quite difficult. For instance, the variation in job runtimes over different platforms must be considered in a heterogeneous setting.

By examining the next arrival times for all placeholders, we can construct a list of upcoming placeholders and then sort that list in the order in which those placeholders will ask for work. Inspecting the first n entries of this list then tells us the next n placeholders that will request work. In practice, we have found that considering the next 32 placeholder arrivals (using n = 32 in Equation 6.3) produces a high degree of data affinity without requiring the simulation program to examine an unnecessarily long next arrivals list.

The *data_transfer* function, the second term in Equation 6.4, measures the virtual time required to transfer job x's input data to the host on which placeholder PH_i runs. Section 6.4 describes the model parameters, pertaining to the network environment, that are used to calculate data transfer time. As mentioned in Section 6.2, our simulator tracks which BLAST jobs run and produce their homologue files on which hosts. The workflow file read in at the beginning of the simulation contains the names and sizes of the homologue input files for each Parsing job. Using the location of homologue files and information from the workflow file, our simulator can determine how much data, if any, would need to be transferred if job x was assigned to placeholder PH_i.

Note that we inflate the cost of data transfer through multiplication by a constant (*penalty*). This way, the final priority of a candidate job is more dependent on whether or not that job's input data needs to be shipped across the network. This helps our scheduling policy better attain its primary objective of co-locating jobs with their data. Also, users of Trellis can control the degree to which data affinity is enforced by adjusting the value of the *penalty* constant. In our simulations, we have found that using *penalty* = 25 is sufficient to achieve high data affinity levels.

The final term in Equation 6.4 measures the runtime of job x. We assume job runtimes are constant across all hosts. In a real setting, the runtime for a given job may vary greatly, depending on the placeholder to which that job is bound – given that metacomputing aggregates heterogeneous resources. However, the assumption of runtime uniformity is acceptable in the context of our motivating application, PA, whose workflow is often distributed over clusters of hosts with similar hardware and software capabilities. For instance, we have observed no significant variation in the runtimes of either the BLAST or Parsing jobs on different hosts in the LAN environment used in the

Model Parameter	Lower	Upper
Sequence File Size (B)	250	850
BLAST Runtime (s)	2.0	8.0
Homologue File Size (B)	5,000	85,000
Parsing Runtime (s)	0.5	0.7

Table 6.1: Approximation of PA Workflow Parameter Ranges Based on Measurement

PA-Trellis experiments of the previous chapter.

There is obviously some room for variation in the priority calculation scheme presented above. In Equations 6.2 and 6.3, for instance, one could add numeric coefficients to the various terms to alter the weight placed on the different job characteristics. Likely, there are several coefficient values that could lead to data affinity being well-balanced against limiting maximum queue time. However, the values as shown in the equations above make the priority calculation simple, and are known to work in practice.

6.4 Model Parameters

Before we could carry out any experiments contrasting the performance of competing scheduling algorithms in simulated LAN and WAN settings, we first needed to determine appropriate constant values for all model parameters.

To properly characterize real PA workflows, we measured the input sizes and runtimes of the BLAST and Parsing jobs from a handful of the proteins from the same proteome used in Chapter 5. Table 6.1 lists the ranges for the four workflow parameters of interest. We base the values of our these Smurph model parameters on the characteristics of five different proteins whose sequence specification files are varied in size, and cover (roughly) the overall range of sequence file sizes that we have observed in proteomes analyzed by PA.

Recall that the analysis of a single protein entails executing a short pipeline of jobs, which was shown in Figure 2.1. When generating synthetic PA workflows for our simulation, we select the size of the sequence file, which constitutes the initial input to a pipeline, from a uniform probability distribution spanning the range of sequence file sizes shown in Table 6.1. Following our observations of real PA input file sizes and job runtimes, we generate the runtimes of the BLAST and Parsing jobs, and the homologue file sizes based on the sequence file sizes.

In addition to replicating the PA workflows in our simulation, we must also replicate the execution environment of those workflows. Table 6.2 shows measurements for the four relevant parameters of the Trellis placeholder environment, for both LAN – and WAN – metacomputers.

The first parameter listed in Table 6.2, CLS Service Invocation Time, is a measure of the time for interaction between placeholders and the CLS. To measure this parameter, we launched a single placeholder whose script was modified to report the turnaround times for the CLS functions called

Model Parameter	LAN	WAN
CLS Service Invocation Time (s)	0.66	1.30
Queuing Delay (s)	0.35	0.35
SCP Latency (s)	0.58	1.20
SCP Transfer Rate (B/s)	20,000	5,000

Table 6.2: Smurph Simulation Parameters Based on Measurement

by placeholders for job retrieval and job completion reporting. As the table shows, the time for placeholder interaction with the CLS in a WAN setting is roughly double that in a LAN setting.

The second parameter listed, Queuing Delay, is a measure of the processing time of a single placeholder request by the CLS. An important consequence of running multiple placeholders is that often more than one placeholder contacts the CLS simultaneously. All placeholders executing workflow jobs from the active PA instance share a common metaqueue, and so individual placeholder requests must be queued at the CLS as placeholders await their turn for accessing that metaqueue.

To measure Queuing Delay, we launched a "dummy" placeholder that constantly pulls command lines out of the metaqueue but does not execute them, thereby continuously interacting with the CLS. One placeholder that was modified to report the turnaround times of the two job management functions but still executed command lines given to it, was also launched. Turnaround times for the job management functions in this case were compared against those in the case with no contention. The average difference in these time values was taken as a measure of the queuing delay. This parameter is the same in a LAN or WAN setting, as shown in Table 6.2, since the queuing delay is purely at the CLS end, and is independent of the network type.

The remaining two parameters of SCP Latency and SCP Transfer Rate measure the time required to transfer an input file between metacomputer hosts. Secure Copy (SCP), part of the SSH suite [21], provides file transfers over an encrypted channel. SCP Latency represents the start-up cost of transferring a file over the network using SCP. The per-file latency in a WAN environment is roughly double that in a LAN environment.

The SCP Transfer Rate represents the average bandwidth attained during the copying of a file from one metacomputer host to another. LAN transfer rates are based the time to copy homologue files between hosts in the Linux cluster we used in our PA-Trellis experiments. WAN transfer rates are representative of actual file transfer times between high-performance computing centres (HPCCs) in Edmonton and Calgary, and between HPCCs in Edmonton and New Brunswick. Note the four-fold reduction in the data transfer rate going from a LAN to a WAN environment. To estimate this final simulation parameter, we examined the differences in transfer times of the homologue files as compared to their size differences, for each of the five proteins previously mentioned.

The bandwidth measures shown in Table 6.2 may seem rather small given the speed of modern LANs and WANs. There are three reasons why we use these numbers: First, they are based on real transfer times of homologue files taken from an actual PA workflow. Second, SCP has an encryption

overhead that reduces sustainable bandwidth. Third, the maximum size of any homologue file is only 85,000 B. Given the properties of modern networks, PA or any application is unlikely to achieve maximum bandwidth for such small messages.

6.5 Practicality of Data Consciousness in Scheduling

An important first step in developing a new scheduling policy that considers data placement is to answer the question: Under what circumstances does intentionally placing jobs with their data notably improve application performance? Clearly, input file sizes are a central factor in the scheduling decisions. When the input files are of a sufficient size, the scheduler may choose to delay the execution of a job for the sake of achieving data affinity, depending on how far into the future the desired host will become available.

The results presented in this section show:

1. DC scheduling offers a moderate reduction in makespan of almost 15% over FCFS for PA workflows based on the original homologue file sizes.

2. The makespan reduction achieved by DC scheduling increases to 53% over FCFS as the file sizes are scaled up sufficiently.

3. Our DC algorithm places the maximum possible percentage of jobs on hosts that hold their input data, regardless of file sizes.

We simulated the execution of our synthetic workflow by reproducing the job placement decisions of the three scheduling algorithms of FCFS, Shortest Job First (SJF), and DC. The FCFS algorithm simply assigns the job at the front of the queue to the first available machine. Thus, jobs are bound to hosts in the precise order that they enter the Trellis metaqueue. The SJF algorithm walks the list of available jobs, examining their expected runtimes, and always chooses the shortest job to run. We discuss the situations in which SJF is desirable, in Section 6.7. As Section 6.3 explained, our DC algorithm calculates job priorities based on multiple job characteristics (the most important being location of input data) in order to reduce data movement and, hence, application response time.

The following three subsections respectively discuss the performance improvements achieved by DC over FCFS and SJF for virtual experiments using the original file sizes (i.e., those that were observed in practice), file sizes inflated by 10, and file sizes inflated by 100.

6.5.1 WAN Results: Original File Sizes

Figure 6.2 shows the makespan of our synthetic 1,000-pipeline-instance workflow, further explained below, for the three competing scheduling algorithms. Values in this graph, and all upcoming graphs of makespan and data affinity, are the average of ten trials with different random number

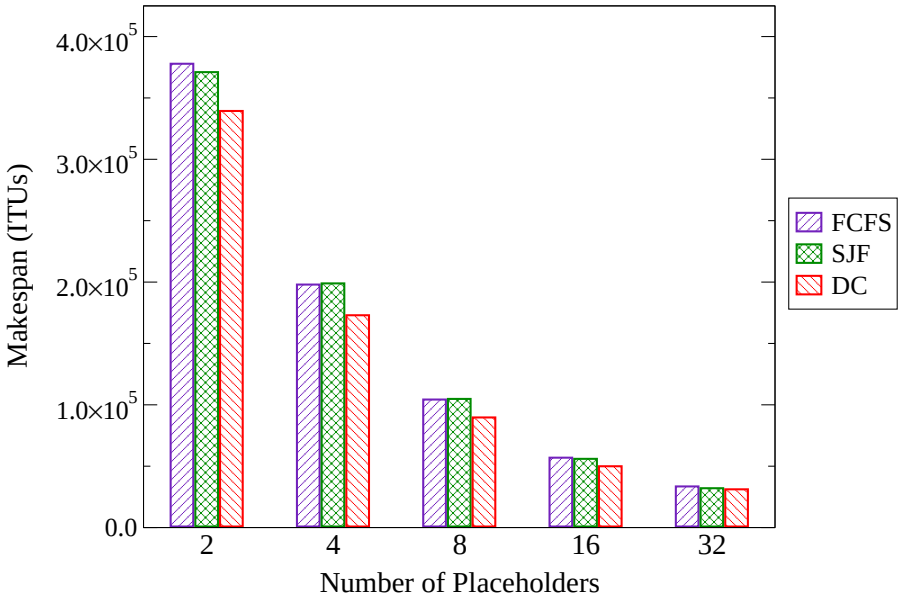


Figure 6.2: **WAN** Makespans for Scheduling Algorithms Using **Original Files**

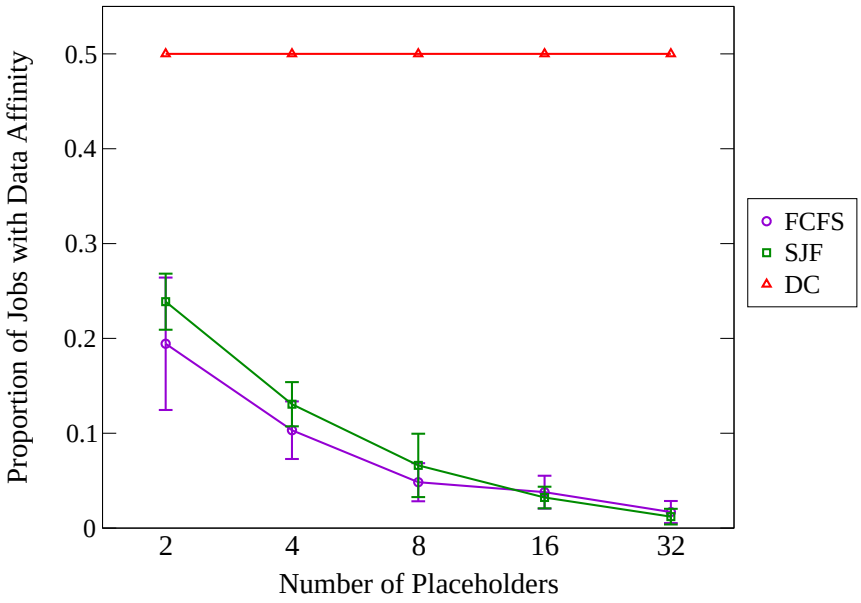


Figure 6.3: **WAN** Data Affinities for Scheduling Algorithms Using **Original Files**

seeds, each of which produces a 1,000-pipeline workflow with unique file sizes and job runtimes. We use homogeneous batching of 16 jobs within Trellis Driver since this is known to be an effective batching factor (i.e., one which provides good speed-up ratios) in real PA workflows, as seen in Chapter 5.

The number of placeholders is plotted along the horizontal axis in Figure 6.2. Makespan measurements are plotted along the vertical axis and are given in Smurph indivisible time units (ITUs), which are the simulation's virtual time units. Here, makespan refers to the elapsed virtual time from when the first BLAST job enters the metaqueue to when the final Parsing job, and hence the entire workflow, completes. We specified all model parameters and job information in such a way that a single Smurph ITU corresponds to 0.01 seconds in a real setting. So a makespan of 3.00×10^5 ITUs conceptually represents a real makespan of 3.00×10^3 seconds.

Figure 6.2 shows that the DC algorithm provides a minor performance benefit against FCFS and SJF. The differences in the magnitude of makespans between the FCFS, SJF, and DC cases are more pronounced for smaller numbers of placeholders, although DC continues to outperform the other two algorithms, even when 32 placeholders are used. The greatest proportionate reduction in makespan occurs with 8 placeholders, when the DC makespan of 9.00×10^4 ITUs is 13.5% and 14.3% less than the respective makespans of 1.04×10^5 ITUs for FCFS and 1.05×10^5 ITUs for SJF.

As the preceding numbers suggest, FCFS and SJF makespans are roughly the same, while the DC makespans are somewhat less, at all placeholder counts except 32. At 32 placeholders, the makespans for all three algorithms are within a much narrower range of one another, with the DC makespan being only 6.9% less than that of FCFS.

We claim contention for the CLS as the main reason for DC's weaker performance at 32 placeholders. The greater the number of placeholders, the more likely their individual communications with the common CLS are to overlap. Since the CLS can process only one placeholder request at a time, multiple requests are frequently queued up, job dispatching is slowed, and makespan is prolonged. We believe that contention affects the performance of DC more than for the other two algorithms. In Section 6.6, we discuss this phenomenon in detail, and present results showing that if the queuing delay is reduced even slightly, DC outperforms FCFS and SJF by a reasonable margin with 32 placeholders.

In Section 5.4.2, our illustrative example of load imbalance demonstrated how high placeholder counts can lead to an uneven job distribution, which prolongs makespan. With 32 placeholders, for example, minor discrepancies in the number of jobs executed on each host are more likely to occur than at 4 placeholders. We believe, however, that imperfect load balancing has only a minor impact on the 32-placeholder case, and so we do not further investigate load balancing effects.

Figure 6.3 shows the proportion of jobs for which data affinity was achieved (i.e., jobs that were placed with their input data) for the three scheduling algorithms. The horizontal axis plots the

number of placeholders while the vertical axis plots the degree of data affinity attained.

We intentionally placed all protein sequence specification files, which are the inputs to the BLAST jobs (Stage 1 in Figure 2.1), on a remote host that was not part of the simulated metacomputer. Thus, the first job in a pipeline is never local to its data, analogous to an unavoidable cold cache miss on the first data access made by a program. Given that each pipeline has two stages, and that all input files for the first stage must be copied from a host that is external to the metacomputer, the maximum possible data affinity for our workflow is 0.5, or 50% of all jobs.

Figure 6.3 shows that the DC algorithm achieves the maximum 50% data affinity level at all placeholder counts. In practice, we do not consider every job that is ready to run, since walking a long list of candidate jobs slows down our simulator program. We have found that considering up to 128 ready jobs is sufficient to consistently find a job whose input data is local to the current placeholder.

The data affinity measurements in Figure 6.3 confirm that DC meets its objective of co-locating jobs and their data for this particular combination of batching strategy, input file sizes, and network type. The figure also shows that FCFS and SJF attain much lower levels of data affinity. This is not surprising since neither scheduling strategy considers the location of input data.

Although SJF may appear to have higher data affinity levels than FCFS at lower placeholder counts, it is important to understand that the job assignments made during the workflow's execution vary greatly between different random seed trials. To emphasize this last point, we calculated the standard deviation in the ten data affinity values from the individual random seed trials whose averages form the data points on the curves in Figure 6.3. The overlapping of the standard deviation bars between the values on the FCFS and SJF curves indicates that there is no statistical difference between the data affinity levels obtained by these two strategies.

While DC outperforms both FCFS and SJF by a wide margin for data affinity, it is evident from the preceding makespan measurements that there is not a large performance gain to be achieved by using scheduling that is data conscious, when the virtual workflows use the original homologue file sizes. This is understandable, given that the maximum size of any homologue file passed between pipeline stages is only 85,000 B, as shown by Table 6.1. Although present-day WANs do impose non-negligible latencies, the homologue files in our virtual PA workflow are simply not large enough to give DC a strong performance edge over its two competitors.

The total quantity of potential data movement also depends on the size of the workflow. For these experiments, we generated a synthetic PA workflow consisting of the first two stages (BLAST and Parsing) of 1,000 protein analysis pipelines. Our choice of 1,000 pipelines is intended to produce a workflow that entails non-trivial data movement, but is still within the scope of real PA use cases. In practice, PA users have analyzed some proteomes containing only 1 sequence and others containing up to 100,000 sequences. Thus, 1,000 proteins entails a moderate amount of data movement.

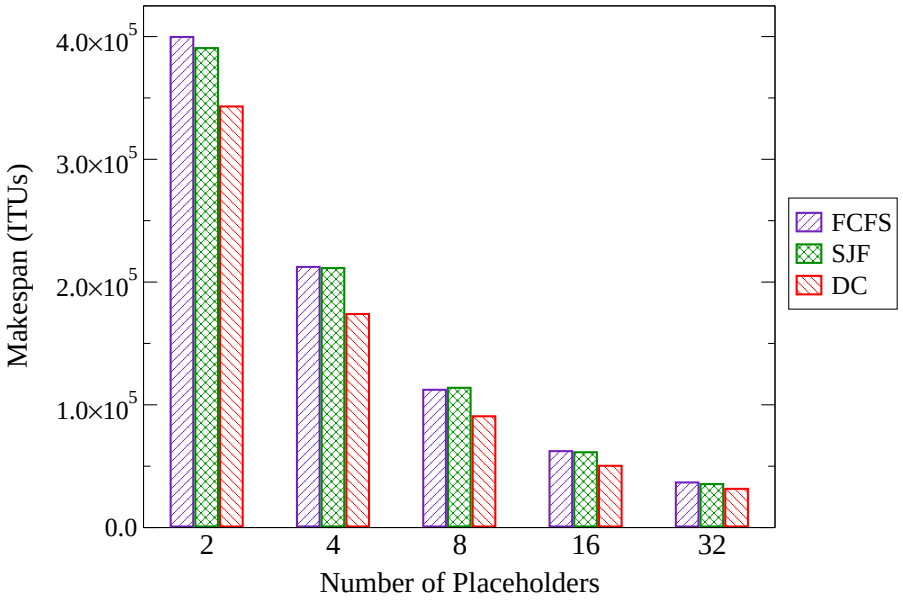


Figure 6.4: **WAN** Makespans for Scheduling Algorithms Using **Files Inflated by 10**

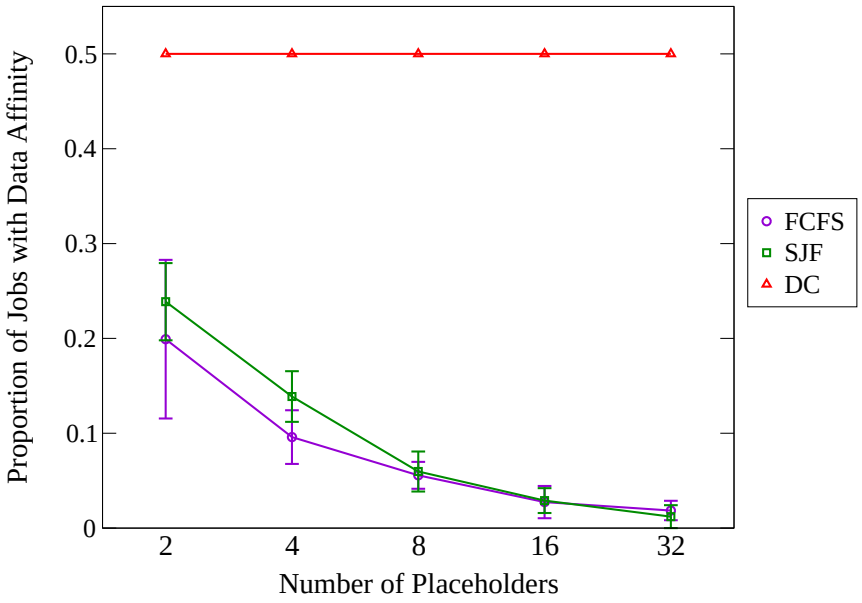


Figure 6.5: **WAN** Data Affinities for Scheduling Algorithms Using **Files Inflated by 10**

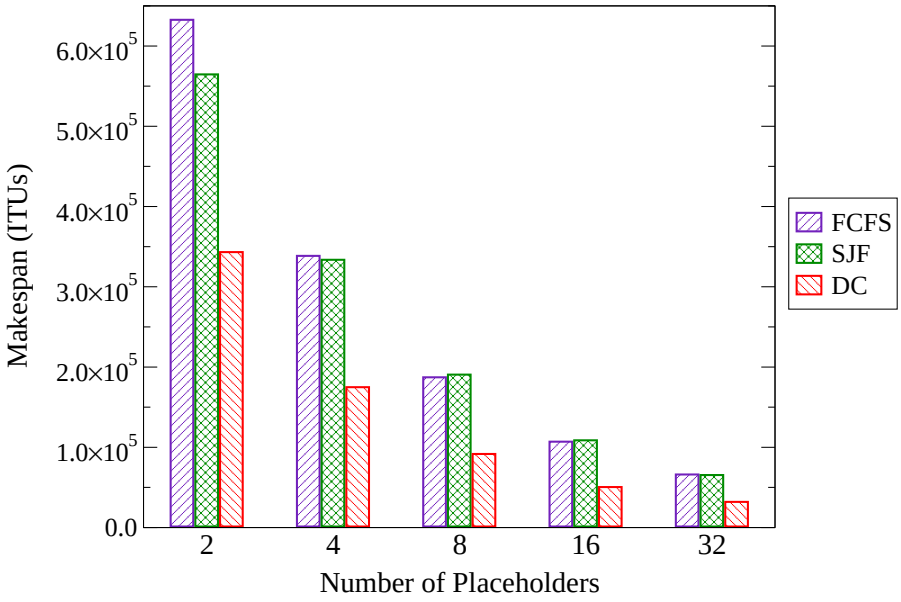


Figure 6.6: **WAN** Makespans for Scheduling Algorithms Using **Files Inflated by 100**

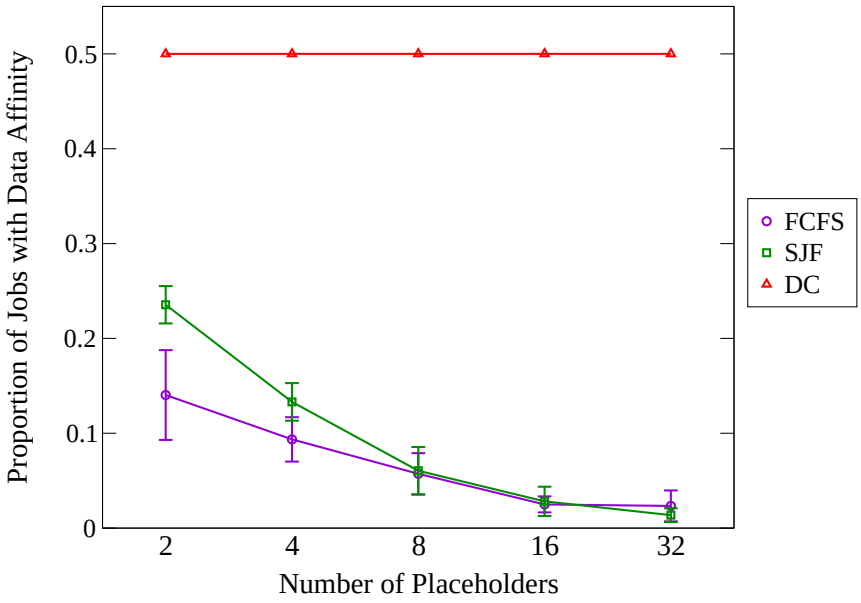


Figure 6.7: **WAN** Data Affinities for Scheduling Algorithms Using **Files Inflated by 100**

6.5.2 WAN Results: File Sizes Inflated by 10

Following the modest performance improvements in our first set of experiments, we inflated the homologue file sizes so that the potential amount of data to be transferred became large enough to affect performance significantly.

Figure 6.4 shows makespans obtained with the three scheduling algorithms for the same PA workflow using homogeneous batching of 16 jobs, but with file sizes multiplied by 10. As expected, makespans for DC are lower than those of FCFS and SJF at all placeholder counts by a wider margin than in the original trials. The greatest proportionate reduction occurs with 8 placeholders, when the DC makespan of 9.07×10^4 ITUs is 19.0% less than the FCFS makespan of 1.12×10^5 ITUs, and 20.4% less than the SJF makespan of 1.14×10^5 ITUs. Thus, DC has a moderate performance gain over FCFS or SJF when the file sizes in our workflow are multiplied by 10.

Figure 6.5 shows data affinity levels. DC achieves the maximum level of 50% at all placeholder counts, far outperforming FCFS and SJF. As in the previous data affinity graph (Figure 6.3), the curve for FCFS appears to be higher than that of SJF at lower placeholder counts. However, there is once again overlapping between the bars marking the standard deviations in the measured values for FCFS and SJF, meaning there is no statistical difference between the data affinity levels obtained by these two algorithms.

6.5.3 WAN Results: File Sizes Inflated by 100

Figure 6.6 contrasts makespans of the competing scheduling strategies for the same PA workflow, using the same job batching strategy, but with file sizes multiplied by 100. Not surprisingly, there is a dramatic reduction in makespan for DC against FCFS or SJF at all placeholder counts. The greatest savings occur with 16 placeholders, when the DC makespan of 5.05×10^4 ITUs is 52.8% and 53.7% lower than the respective makespans of 1.07×10^5 ITUs and 1.09×10^5 ITUs for FCFS and SJF. DC, then, has a large performance advantage over the other two algorithms when the file sizes are multiplied by 100. Makespan values for FCFS and SJF are quite close except when 2 placeholders are used.

Data affinity levels are shown in Figure 6.7. As before, DC achieves the maximum level of 50% at all placeholder counts, with FCFS and SJF attaining much lower data affinity levels. We see the same wide standard deviation bars on the average data affinity values plotted by the FCFS and SJF curves. Note, though, that there is no overlapping of bars for the 2-placeholder case, suggesting FCFS fares worse than SJF here.

While the cause of this performance discrepancy may be of general interest in workflow scheduling, we have not yet found a satisfactory reason for it. We do not investigate this phenomenon any further, since our data affinity results achieve their main purpose of demonstrating that DC attains higher data affinity levels than either FCFS or SJF.

6.5.4 Summary of Results

The makespan measurements from the preceding three sets of empirical trials indicate that data-conscious scheduling provides a minor reduction in makespan against FCFS and SJF for synthetic PA workflows using file sizes taken from a real setting. We would expect a real PA workflow of roughly 1,000 pipelines, then, to benefit only slightly from our new algorithm when run in a WAN environment. However, when we multiply the file sizes by 10, and especially by 100, we see much larger proportionate makespan reductions. As expected, workloads that are more data-intensive perform notably better with DC than with FCFS or SJF.

The preceding results demonstrate that data-conscious scheduling is most useful in situations when the amount of data passed between interdependent workflow jobs, and hence the amount of time spent transferring data, is significant compared to the overall makespan.

The data affinity measurements from the preceding trials indicate that the DC algorithm achieves optimal data affinity levels, irrespective of file size. Thus, in a WAN environment, DC minimizes the number of job assignments that result in data transfer, even when the total data movement savings are small.

6.6 LAN vs. WAN: Differences in Performance

In our first sets of experiments, we simulated a WAN because of the high data movement overhead in such an environment. The WAN experiments, then, formed a suitable testbed for determining the circumstances in which data-conscious scheduling is desirable. However, it is common for PA workflows (and those of other scientific applications) to be run over LANs. One reason for using a LAN is that it offers a controlled environment in which application administrators can eliminate unwanted contention for computing resources and network bandwidth.

The results in this section show:

1. Performance benefits of DC scheduling in a simulated LAN follow the same trends observed in a WAN. As file sizes in the PA workflow are increased, makespan reductions of DC over FCFS and SJF increase notably, reaching a respectable 25.0% when file sizes are inflated by 100.

2. The DC algorithm places a slightly lower percentage of jobs with their data in LANs that it does in WANs, when the original file sizes are used.

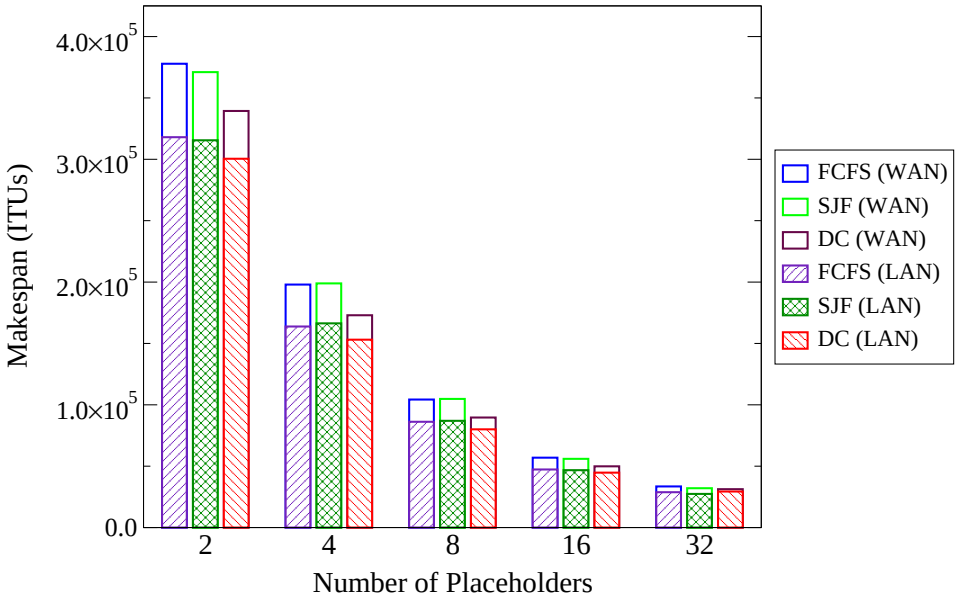


Figure 6.8: **WAN and LAN** Makespans for Scheduling Algorithms Using **Original Files**

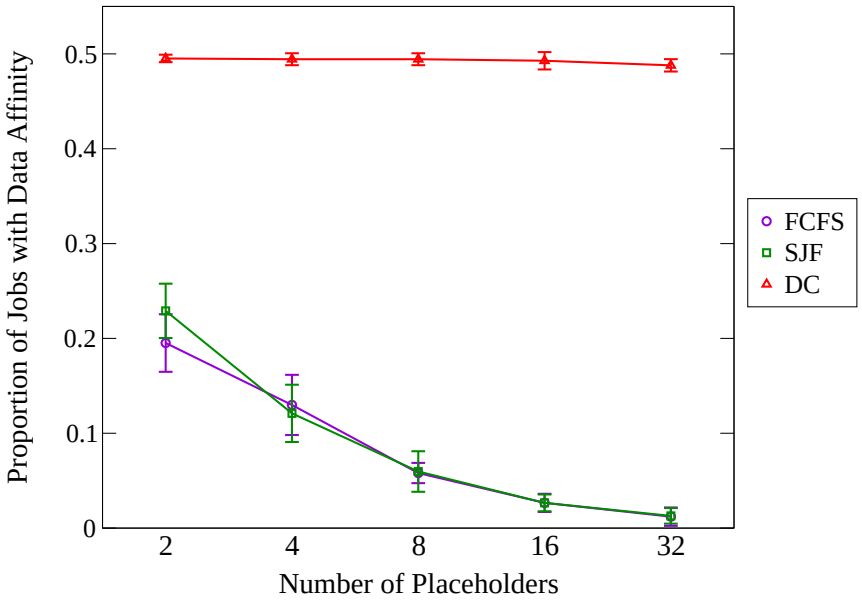


Figure 6.9: **LAN** Data Affinities for Scheduling Algorithms Using **Original Files**

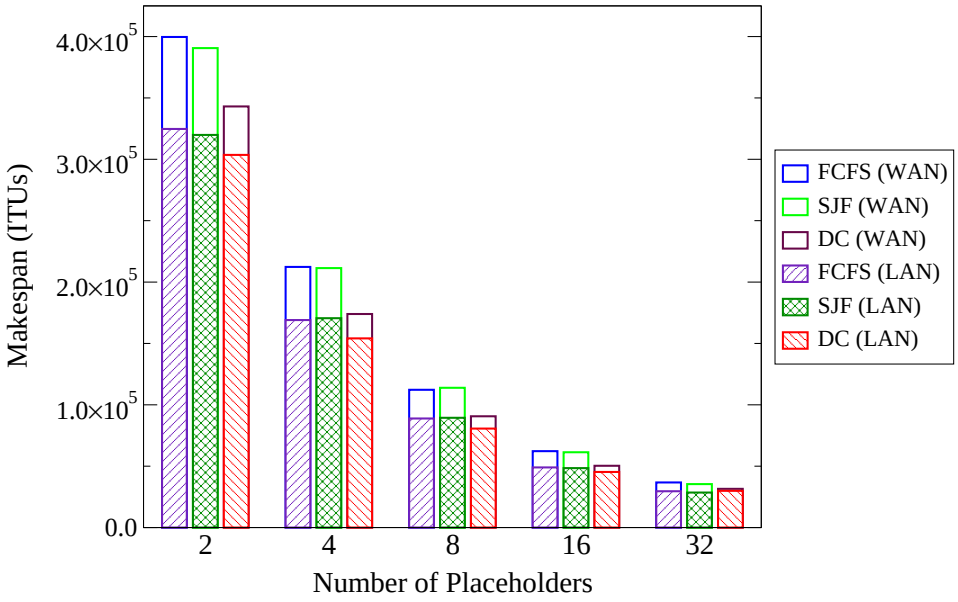


Figure 6.10: **WAN and LAN** Makespans for Scheduling Algorithms Using **Files Inflated by 10**

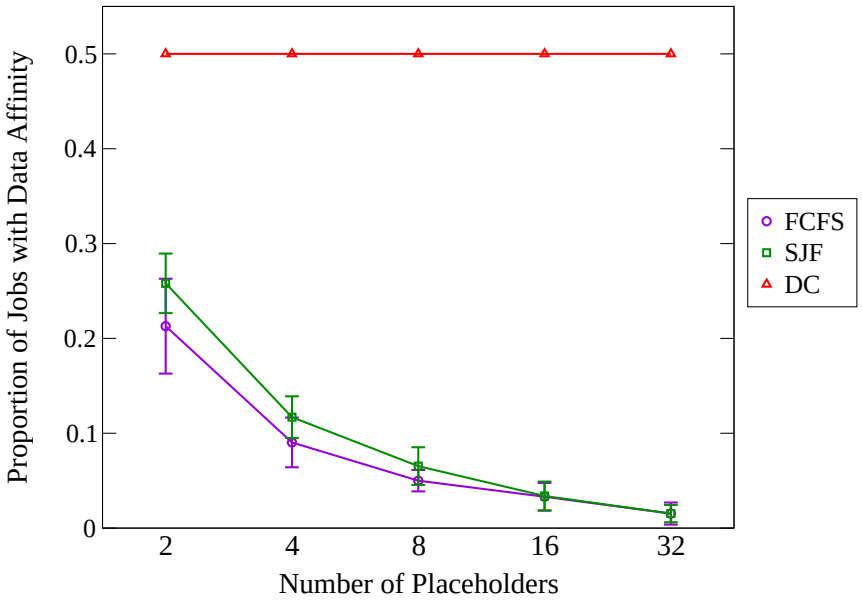


Figure 6.11: **LAN** Data Affinities for Scheduling Algorithms Using **Files Inflated by 10**

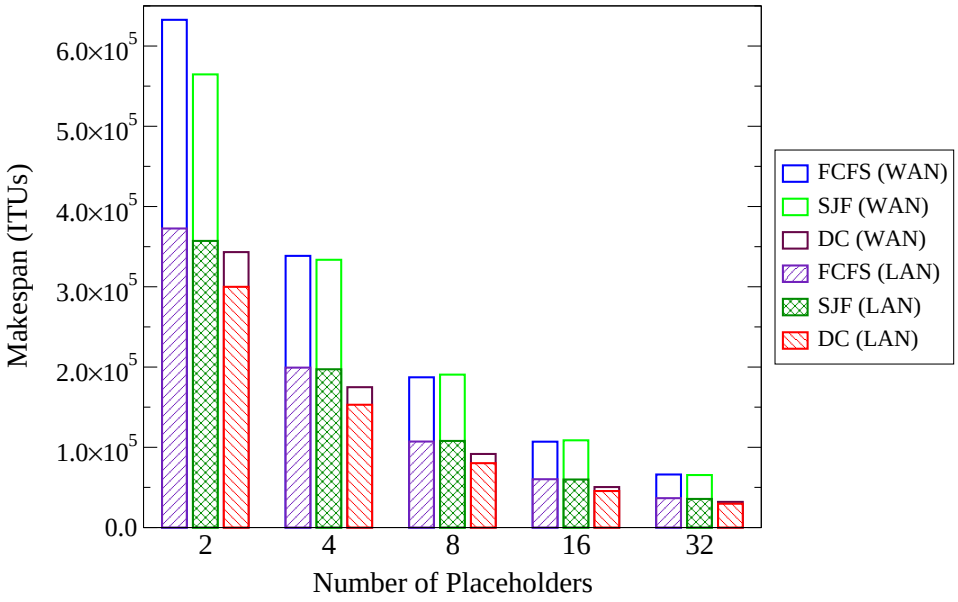


Figure 6.12: **WAN and LAN** Makespans for Scheduling Algorithms Using **Files Inflated by 100**

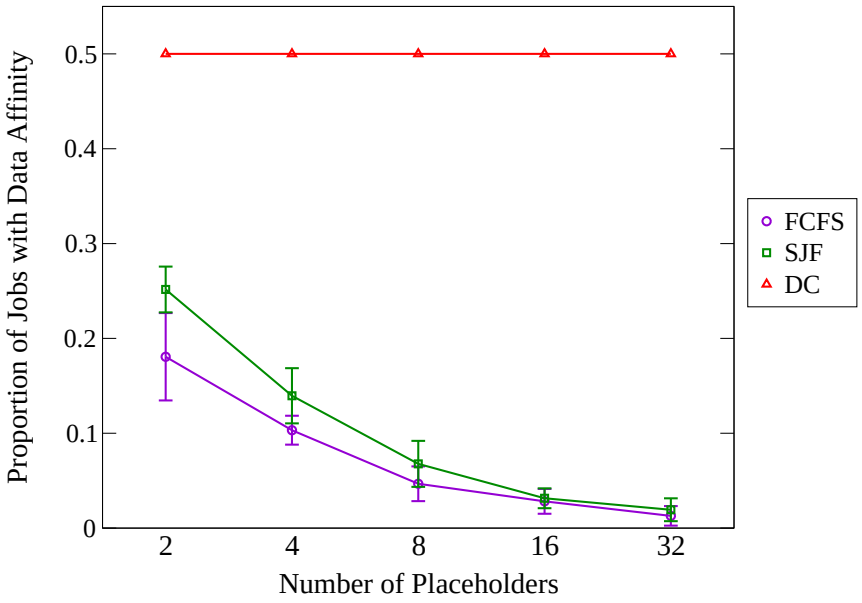


Figure 6.13: **LAN** Data Affinities for Scheduling Algorithms Using **Files Inflated by 100**

Figure 6.8 shows the makespan obtained with the three scheduling strategies at the same five placeholder counts used in the WAN trials, with homogeneous batching of 16 jobs. We illustrate both LAN and WAN makespans (the latter borrowed from Figure 6.2) to highlight the differences in makespans obtained in the two network environments. With 2 placeholders, the FCFS makespan is 3.18×10^5 ITUs in a LAN setting, as compared to 3.78×10^5 ITUs in a WAN setting – which is a reduction of 15.9%. The SJF makespan of 3.15×10^5 ITUs in a LAN is 15.1% less than the 3.71×10^5 ITUs makespan in a WAN. Finally, the DC-LAN makespan of 3.00×10^5 ITUs is 11.5% less than the DC-WAN makespan of 3.39×10^5 ITUs.

The makespan reduction for DC is somewhat smaller than that for the two other algorithms because the only source of makespan savings for DC when run over a LAN versus a WAN is the decreased cost of placeholder communication with the CLS, as listed in the first row of Table 6.2. Since DC minimizes data movement in any network environment, the total overhead due to data transfer experienced by DC is much less than that experienced by either FCFS or SJF. The makespan reduction values above suggest that placeholder communication imposes significant overhead.

Migrating from a WAN to a LAN environment, we continue to see the same trends in both makespan reduction and the relative performances of the various algorithms at all placeholder counts, except for 32. At 32 placeholders, the DC-LAN makespan of 2.91×10^4 ITUs is a mere 6.4% less than the DC-WAN makespan of 3.11×10^4 ITUs. This DC makespan also is slightly higher than the makespans of FCFS and SJF, which are 2.88×10^4 ITUs and 2.74×10^4 ITUs, respectively.

In Section 6.5.1, we claimed that the main reason for DC's drop in performance at 32 placeholders was contention for the common CLS. We reasoned that a high placeholder count causes individual placeholder requests to overlap often, resulting in one placeholder being serviced by the CLS, while the others idly await their turn. This frequent contention increases the average turnaround time for the two placeholder operations of retrieving a new command line and reporting a job's completion, thereby slowing the workflow execution.

The waiting time for queued placeholders depends on the Queuing Delay parameter, the concept of which was explained in Section 6.4 and for which a value of 0.35 seconds was given in Table 6.2. We ran our simulation with a decreased Queuing Delay value of 0.25 seconds to verify whether reduced queuing times significantly decrease the performance penalty imposed by contention. We obtained FCFS-LAN and DC-LAN makespans of 2.83×10^4 ITUs and 2.76×10^4 ITUs, respectively, for our synthetic workflow at 32 placeholders. Further reducing the Queuing Delay value to 0.15 seconds widened the gap between FCFS and DC makespans in a LAN, producing values of 2.79×10^4 ITUs and 2.67×10^4 ITUs, respectively. These results indicate that reducing the Queuing Delay even moderately causes the makespan of DC to drop below that of FCFS, suggesting that contention is indeed the main cause of DC's slightly weaker performance at 32 placeholders.

Figure 6.9 plots the data affinity levels for the LAN trials using the original file sizes. DC obtains data affinity of almost 50% at all placeholder counts, far outperforming FCFS and SJF. The slightly

suboptimal data affinity levels are not problematic. The lower cost of data movement in a LAN means that the scheduler sometimes chooses not to co-locate jobs and their data.

Despite the high data affinity attained by DC, however, the makespan results show that there is at best a minor performance improvement and, at 32 placeholders, a slight drop-off in performance when DC is used instead of FCFS or SJF.

As we increase the file sizes, DC's performance benefits gradually increase. Figures 6.10 and 6.11 provide the makespan and data affinity results from the LAN experimental trials with file sizes inflated by 10. The DC makespans are only slightly less than that of FCFS and SJF, even though DC attains maximum data affinity at all placeholder counts. Figures 6.12 and 6.13 provide the results from the LAN trials with file sizes inflated by 100. Here, DC begins to offer a noticeable performance benefit over FCFS and SJF. The greatest reduction occurs with 8 placeholders, when the DC makespan of 8.02×10^{4} ITUs is 25.0% less than the FCFS makespan of 1.07×10^{5} ITUs.

As expected, increasing the file sizes does not hurt performance substantially in a LAN. This last observation is evident in the increasing gaps between the makespan values from the two trial sets as we scale up the file sizes. WAN makespans increase significantly while LAN makespans increase moderately. With file sizes inflated by 100, the FCFS-LAN makespan at 2 placeholders is 3.73×10^{5} ITUs, a steep 41.1% less than the FCFS-WAN makespan of 6.33×10^{5} ITUs.

We conclude that although the benefits of DC scheduling follow the same trends in a simulated LAN as in a WAN, the potential data movement savings are too small to benefit makespan to any significant degree, unless the file sizes are inflated by at least 100.

6.7 Advantages of SJF

Thus far, we have emphasized the performance benefits of DC primarily over FCFS, which is the scheduling mechanism currently used in Trellis. We include SJF metrics in our makespan and data affinity results for three reasons: 1) SJF is a widely-used job scheduling strategy that is easy to implement; 2) SJF, like FCFS, is unaware of input data location and is therefore a suitable second baseline strategy to compare our DC algorithm against; and 3) SJF minimizes mean response time, which is a standard performance measure in scheduling. Mean response time refers to the average time elapsed from when a job is submitted to the Trellis system to when that job finishes executing. In this section, we show that despite the tendency of SJF to minimize mean response times, DC actually produces lower mean response times than SJF in WANs for all file sizes, and in LANs when the files are inflated by 100.

PA users are concerned primarily about the end-to-end time for training a new classifier or using an existing one to classify an entire proteome, and not individual job response times. However, some scientific applications have workflows that consist of a single pipeline of jobs. In these cases, the mean response time determines the rate at which the results are produced and made available to the user. The SJF strategy would then be a logical scheduling strategy to use.

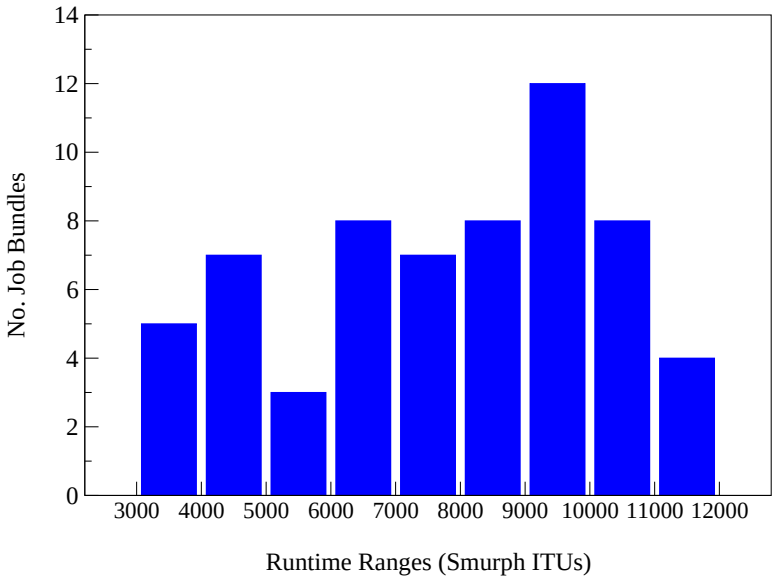


Figure 6.14: Distribution of Trellis Job Runtimes
Homogeneous Batching of 16 Jobs, Random Seed = 1

The benefits of SJF are most apparent when there is a substantial range in service times. This is because the algorithm avoids executing the longest-running jobs, which can tie up all other jobs, until the end of the workflow's execution. Figure 6.14 plots the distribution of runtimes for the Trellis jobs comprising the BLAST phase. These results are based on a homogeneous batching factor of 16, with a random seed value of 1. Examination of the workflow files produced by different seed values revealed a similar variation in job runtimes.

In Table 6.1, we saw there was considerable variation in the runtimes of individual BLAST jobs (between two and eight seconds). When multiple BLAST jobs are bundled into a common Trellis job with homogeneous batching, the discrepancies in runtimes evidently become magnified.

Figures 6.15 and 6.16 illustrate the mean response time across the BLAST and Parsing phases at our five standard placeholder counts in WAN and LAN settings, respectively. The results in both figures indicate that FCFS has the highest mean response time for any number of placeholders in either network type. This is not surprising since FCFS is designed to be fair and not to minimize job response times. The WAN results shown in Figure 6.15 indicate that only at 2 placeholders does SJF produce a lower mean response time than DC. At higher placeholder counts, DC outperforms SJF. The data movement savings realized by a data-conscious scheduling policy, even when using the original, relatively small files, evidently pay off against the baseline SJF strategy in a WAN environment.

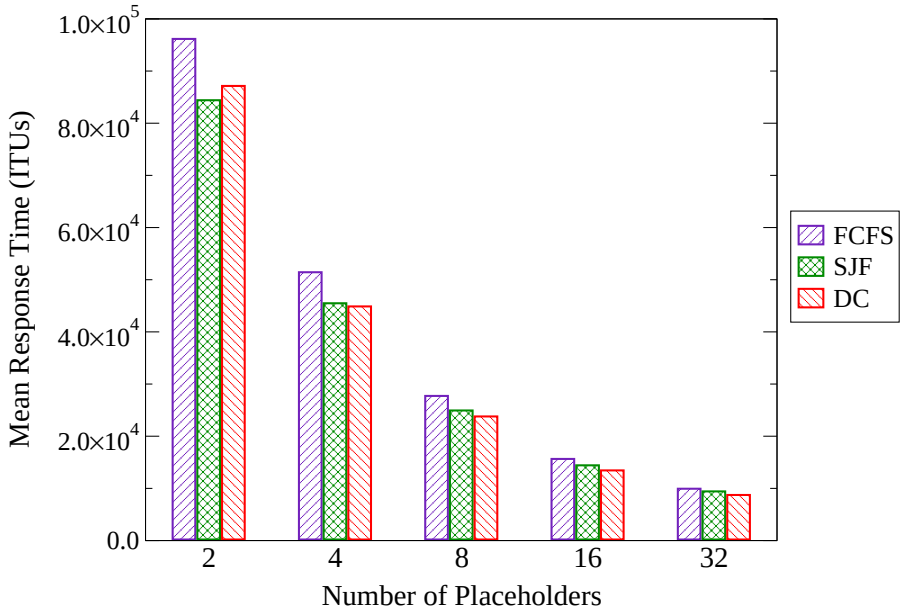


Figure 6.15: **WAN** Mean Response Times for Scheduling Algorithms Using **Original Files**

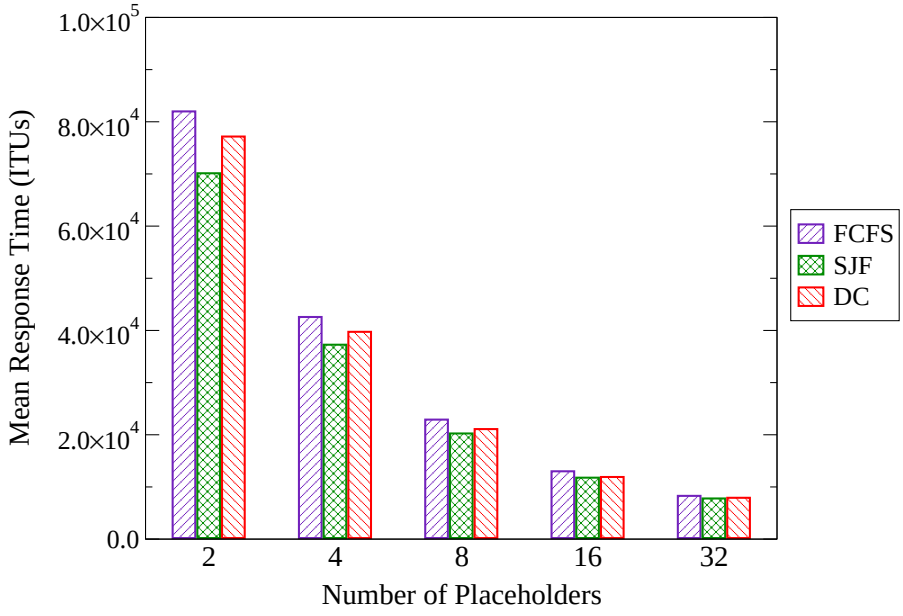


Figure 6.16: **LAN** Mean Response Times for Scheduling Algorithms Using **Original Files**

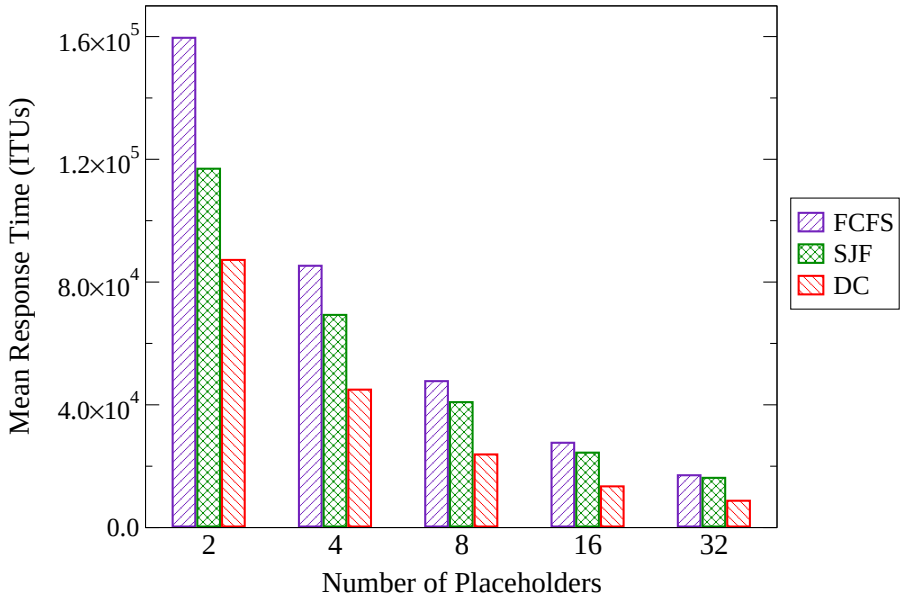


Figure 6.17: **WAN** Mean Response Times for Scheduling Algorithms Using **Files Inflated by 100**

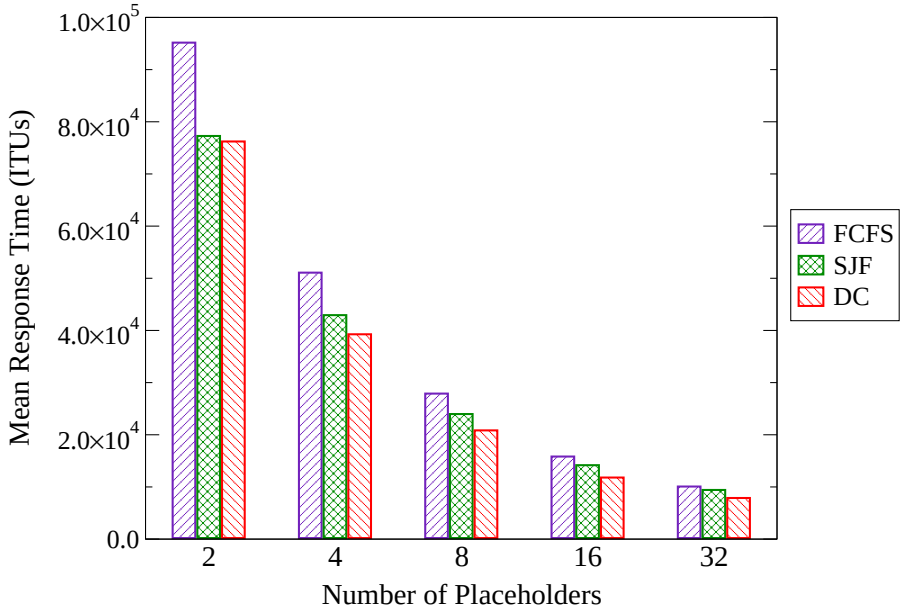


Figure 6.18: **LAN** Mean Response Times for Scheduling Algorithms Using **Files Inflated by 100**

The results shown in Figure 6.16 indicate that SJF outperforms DC in terms of mean response times in a LAN, albeit by a very narrow margin at 16 and 32 placeholders. The performance gap between SJF and DC narrows as more placeholders are added because the data affinity achieved by chance for SJF linearly decreases with the number of placeholders. For instance, with 2 placeholders, there is a 1 in 2, or 50% probability, that any given Parsing job will be placed on the host that holds its input data. However, with 32 placeholders, there is only a 1 in 32, or roughly 3% probability, that a Parsing job will be placed with its data. Inspecting the data affinity curves for SJF in Figure 6.9 confirms this assertion.

Despite the inevitable rise in total data movement as the number of placeholders increases, the overall cost of data movement in a LAN does not become so great that DC ever outperforms SJF for mean response time, even at 32 placeholders. These results reveal that if the PA user desires short mean response times, SJF is a better choice than DC when PA's workflow is distributed across a LAN.

Even when the file sizes are inflated by 10, mean response times achieved by the various scheduling algorithms follow the same trends as in the previous trials, in both LAN and WAN settings. For this reason, we do not present the corresponding graphs. When the file sizes are inflated by 100, however, DC outperforms SJF in either network type.

Figures 6.17 and 6.18, respectively, show the mean response times in WAN and LAN settings with file sizes inflated by 100. Not surprisingly, FCFS again has the worst performance at all placeholder counts in either setting. Figure 6.17 shows that DC mean response times are considerably lower than those of SJF in a WAN, with the relative performance gap between the two algorithms widening as the placeholder count is scaled up. This is understandable given that the percentage of jobs placed with their data by chance drops for SJF as placeholders are added, as explained above, but DC achieves high levels of data affinity at all placeholder counts, as shown by Figure 6.7. The greatest gap in performance occurs at 32 placeholders, when the DC mean response time of 8.75×10^{3} ITUs is 46.0% lower than the SJF measure of 1.62×10^{4} ITUs.

Figure 6.18 shows that in a LAN, with file sizes inflated by 100, DC outperforms SJF for mean response time, unlike in the LAN trials with smaller files. Again we see a widening in the relative gap for SJF and DC values as the placeholder count is increased. Here, the greatest performance gap occurs at 16 placeholders when the DC mean response time of 1.18×10^{4} ITUs is 16.9% less than the SJF measure of 1.42×10^{4} ITUs. Thus, when the potential data movement becomes high enough, DC outperforms SJF in mean response time, even in LANs where data transfer costs are not prohibitively high. From this set of experiments, we conclude that for all file sizes in a WAN, and for sufficiently large files in a LAN, DC produces somewhat shorter job response times than SJF.

6.8 Summary of Results

The PA-Trellis experiments carried out over an actual LAN, whose results were presented in Chapter 5, left some unanswered questions. Namely, what are the effects of running PA-Trellis over a WAN, or increasing the file sizes? In this chapter, we conducted a simulation study to explore these questions and in so doing, determined the circumstances in which data-conscious scheduling improves PA's performance by a fair margin.

We explained our new DC policy's method of assigning job priorities, which essentially measures the trade-off between running a candidate job immediately and running that job later for the sake of data affinity. We found that DC produces somewhat lower makespans than either FCFS or SJF in a WAN, when the original file sizes are used. When the homologue file sizes are inflated by 10 or 100, DC's makespans are considerably lower than those of the other two scheduling algorithms. In a LAN setting, makespan values follow the same trends as in a WAN, except the benefits of DC over FCFS are less pronounced since data movement overheads are lower in LANs. In either network type, we found that DC places all or nearly all the workflow jobs with their data, when possible. Lastly, we examined mean response times produced by the three algorithms and found that DC edges out SJF at all file sizes in WAN, and for large files in a LAN.

Chapter 7

Conclusions

Scientific applications that are resource-demanding can benefit significantly by executing their workflows, in parallel, over aggregations of servers, specifically metacomputers. Our application of choice, Proteome Analyst (PA), was a representative example of a scientific computation that can notably reduce its turnaround time by load balancing its workflow across metacomputers. The two PA use cases we focused on, training and prediction, both entail executing a fixed pipeline of jobs for every protein sequence in the input proteome, which may contain tens of thousands of sequences. In either the training or the prediction pipeline, the first two stages of BLAST and Parsing are embarrassingly parallel, making them suitable candidates for metacomputing.

Trellis Driver integrated PA with metacomputing easily, thereby enabling PA to distribute its workflow of jobs over a local area network (LAN) cluster of hosts. The code snapshots from the original PA, which ran all BLAST jobs locally, and the parallel PA (PA-Trellis), which used Trellis Driver to run its BLAST jobs across metacomputers, demonstrated that Trellis Driver conveniently functions as a drop-in replacement for the `Runtime.exec()` mechanism. Only 6 new lines of code were added, and 13 existing lines replaced with 6 new ones to parallelize the BLAST phase of PA. The bounded buffer architecture ensures that no matter how many BLAST jobs PA starts, the resources consumed by Trellis Driver will be restricted.

With the benefit of increased throughput that metacomputing offers comes some unavoidable overheads in scheduling jobs across multiple hosts. We developed two distinct batching strategies for Trellis Driver that address the separate overheads of `mqsub` scheduling and data movement. Homogeneous batching, which groups together jobs of the same type, was shown experimentally to be much better at speeding up PA's workflow than heterogeneous batching, which groups together jobs of different types. Thus, for our 3,916-sequence test case, `mqsub` appears to be a much greater source of overhead than file transfers between pipeline stages.

Empirical results from the LAN experiments revealed significant performance variations between the BLAST and Parsing phases. The BLAST phase attained linear speed-ups with higher batching factors at lower placeholder counts, suggesting that homogeneous batching is quite effective at amortizing `mqsub` overheads. The optimal batching factor varies with the number of

placeholders, which we attribute to load imbalance under certain conditions. The Parsing phase had low speed-ups at all placeholder counts, regardless of the batching factor. The performance differences between the parallel BLAST and Parsing phases indicate that job granularity greatly influences speed-up.

The data flow dependencies between the first and second pipeline stages may necessitate transferring many files over the network. We developed a simulation of PA-Trellis and an underlying Trellis metacomputer to investigate the performance benefits of a data-conscious scheduling policy, which aims to minimize the number of files transferred. Our simulation reproduced the job placement decisions of three scheduling strategies, and allowed us to explore the effects of factors such as network type and file sizes. When modelling a wide area network (WAN) setting, we found that our new Data-Conscious (DC) policy significantly reduces makespan (by roughly 50%) against the existing First Come First Served (FCFS) mechanism, when the file sizes are inflated by 100. In a LAN setting, makespan reduction is smaller, but still respectable (roughly 25%), for the same inflated file sizes. Thus, in workloads with high data movement costs, whether due to large file sizes or high network latencies, DC scheduling is reasonably beneficial.

In Chapter 1, we stated three main contributions of this thesis:

1. The development of a Java package to enable workflow concurrency within metacomputers;
2. The implementation of job batching strategies to amortize scheduling overheads; and
3. The development of a data-conscious scheduling policy that reduces turnaround time.

The design and implementation overview of our new Trellis Driver module showed how existing Java programs can easily be ported to metacomputing by replacing `Runtime.exec()` calls with `TrellisDriver.exec()` calls. Our empirical evaluation of Trellis Driver showed that batching multiple jobs together does indeed reduce `mqsub` overheads, leading to linear speed-up of embarrassingly parallel application phases. The simulation experiments demonstrated that our DC scheduling policy, which places jobs with their input data, when practical, notably reduces makespan, or turnaround time, when the data movement costs are high enough.

Bibliography

[1] Abramson, D. GriddLeS: Grid Enabling Legacy Software, [Online 2005]. http://www.csse.monash.edu.au/~davida/griddles/applications.htm.

[2] Abramson, D., Kommineni, J., McGregor, J.L., and Katzfey, J. An Atmospheric Sciences Workflow and Its Implementation with Web Services. *Future Generation Computer Systems (FGCS)*, 21:69–78, 2005. Also appeared in International Conference on Computational Science (ICCS04), Krakow, Poland, June 2004.

[3] Altair Engineering, Inc. Altair PBS Pro, [Online 2005]. http://www.altair.com/pdf/PBSPro.pdf.

[4] Altschul, S.F., Madden, T.L., Schaffer, A.A., Zhang, J., Zhang, Z., Miller, W., and Lipman, D.J. Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. *Nucleic Acids Research (NAR)*, 25(17):3389–3402, September 1997.

[5] Bairoch, A., and Apweiler, R. The SWISS-PROT protein sequence data bank and its supplement TrEMBL. *Nucleic Acids Research (NAR)*, 25(1):31–36, 1997.

[6] Braun, F., Pedretti, K., Casavant, T., Scheetz, T., Birkett, C., and Roberts, C. Parallelization of local BLAST service on workstation clusters. *Future Generation Computer Systems (FGCS)*, 17(6):745–754, 2001.

[7] Bulhões, P., Byun, C., Castrapel, R., and Hassaine, O. N1 Grid Engine 6 Features and Capabilities, [Online 2005]. http://www.sun.com/products-n-solutions/edu/whitepapers/pdf/N1GridEngine6.pdf.

[8] Dongarra, J., Otto, S., Snir, M., and Walker, D. An Introduction to the MPI Standard. Technical report, Knoxville, Tennesse, USA, 1995. http://portal.acm.org/citation.cfm?id=898812.

[9] EBI (2005). EMBL-EBI: European Bioinformatics Institute, [Online 2005]. http://www.ebi.ac.uk/Databases.

[10] Epema, D., Livny, M., van Dantzig, R., Evers, X., and Pruyne, J. A worldwide flock of Condors: Load sharing among workstation clusters. *Future Generation Computer Systems (FGCS)*, 12:53–65, 1996.

[11] Foster, I., Kesselman, C., Nick, J., and Tuecke, S. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, June 2002. http://www.globus.org/alliance/publications/papers.php#OGSA.

[12] Gburzynski, P. An Overview of Smurph: an Object-oriented Configurable Simulator for Low-level Communication Protocols, [Online 2005]. http://www.cs.ualberta.ca/~pawel/SMURPH/report.html.

[13] Goldenberg, M. Trellis DAG: A System for Structured DAG Scheduling. Master's thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, April 2003. http://www.cs.ualberta.ca/~goldenbe/papers/thesis.ps.gz.

[14] Kan, M., Ngo, D., Lee, M., and Lu, P. The Trellis Security Infrastructure: A Layered Approach to Overlay Metacomputers. In *Proc. 18th International Symposium on High Performance Computing Systems and Applications (HPCS)*, pages 109–117, Winnipeg, Manitoba, Canada, May 2004. http://www.cs.ualberta.ca/~paullu.

[15] Kernighan, B.W., and Ritchie, D.M. *The C Programming Language*. Prentice Hall, 1998.

[16] Lamb, N., Lu, P., and Fyshe, A. Trellis Driver: Distributing a Java Workflow Across a Network of Workstations. In *Proc. 6th International Workshop on High Performance Scientific and Engineering Computing with Applications (HPSEC-04)*, Montreal, Quebec, Canada, August 2004. http://www.cs.ualberta.ca/~nlamb/lambn_trellis_hpsec04-14.pdf.

[17] Ma, B., Tromp, J., and Li, M. PatternHunter: Faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, March 2002. http://citeseer.ist.psu.edu/ma02patternhunter.html.

[18] Mathematics and Computer Science (MCS) Division, Argonne National Laboratory. The Message Passing Interface (MPI) standard, [Online 2005]. http://www-unix.mcs.anl.gov/mpi.

[19] Meyer, L., Rössle, S., Bisch, P., and Mattoso, M. Parallelism in Bioinformatics Workflows. In *Proc. Vector and Parallel Processing 2004 (VECPAR 2004)*, pages 583–597, Valencia, Spain, June 2004.

[20] Mitchell, T.M. *Machine Learning*. McGraw-Hill, New York, New York, USA, 1997.

[21] Open BSD. Open SSH Project, [Online 2005]. `http://www.openssh.org`.

[22] Pinchak, C., Lu, P., and Goldenberg, M. Practical Heterogeneous Placeholder Scheduling in Overlay Metacomputers: Early Experiences. In *Proc. 8th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 85–105, Edinburgh, Scotland, UK, July 2002. Also published as Springer-Verlag LNCS 2537 (2003), pages 205–228. `http://www.cs.ualberta.ca/~paullu`.

[23] Pinchak, C., Lu, P., Schaeffer, J., and Goldenberg, M. The Canadian Internetworked Scientific Supercomputer. In *Proc. 17th International Symposium on High Performance Computing Systems and Applications (HPCS)*, pages 193–199, Sherbrooke, Quebec, Canada, May 2003. `http://www.cs.ualberta.ca/~paullu`.

[24] Shan, H., Oliker, L., Smith, W., and Rupak, B. Scheduling in Heterogeneous Grid Environments: The Effects of Data Migration. In *Proc. Advanced Computing and Communications 2004 (ADCOM 2004)*, Ahmedabad, Gujurat, India, December 2004. `http://crd.lbl.gov/~oliker`.

[25] Siegel, J., and Lu, P. User-Level Remote Data Access in Overlay Metacomputers. In *Proc. 4th IEEE International Conference on Cluster Computing (Cluster 2002)*, pages 480–483, Chicago, Illinois, USA, September 2002. `http://www.cs.ualberta.ca/~paullu`.

[26] Snow, C.D., Zagrovic, B., and Pande, V.S. The Trp Cage: Folding Kinetics and Unfolded State Topology via Molecular Dynamics Simulations. *Journal of the American Chemical Society*, 2002. `http://folding.stanford.edu/Snow_Pande_JACS_2002.pdf`.

[27] Stevens, W.R. *Advanced Programming in the Unix Environment*. Addison-Welsey Publishing Company, 1992.

[28] Sun Microsystems, Inc. Java 2 Platform Standard Edition v1.4.2, API Specification, [Online 2005]. `http://java.sun.com/j2se/1.4.2/docs/api/index.html`.

[29] Szafron, D., Lu, P., Greiner, R., Wishart, D.S., Lu, Z., Poulin, B., Eisner, R., Anvik, J., and Macdonell, C. Protein Analyst – Transparent High-throughput Protein Annotation: Function, Localization and Custom Predictors. *12th International Conference on Machine Learning, Workshop on Machine Learning in Bioinformatics (ICML Workshop–Bioinformatics)*, August 2003.

[30] Thain, D., Tannenbaum, T., and Livny, M. Condor and the grid. In Berman, F., Fox, G., and Hey, T., editor, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.

[31] U.S. Department of Energy Genome Programs. Genomics and Its Impact on Science and Society: The Human Genome Project and Beyond, March 2003. `http://www.ornl.gov/sci/techresources/Human_Genome/publicat/primer2001/primer11.pdf`.

[32] van Rossum, G. Python Library Reference, [Online 2005]. `http://docs.python.org/lib/lib.html`.

[33] Wang, H., Ong, T., Chin Ooi, B., and Tan, K. BLAST++: A Tool for BLASTing Queries in Batches. In *Proc. First Asia-Pacific Bioinformatics Conference (APBC 2003)*, pages 71–79, Adelaide, Australia, 2003. Australian Computer Society.

[34] Xu, Y., Huckauf, A., Jäger, W., Lu, P., Schaeffer, J., and Pinchak, C. The CISS-1 Experiment: *ab initio* Study of Chrial Interactions. In *Proc. 39th International Union of Pure and Applied Chemistry (IUPAC) Congress and 86th Conference of The Canadian Society for Chemistry*, Ottawa, Ontario, Canada, August 2003. `http://www.cs.ualberta.ca/~paullu`.

[35] Zhang, Z., Schwartz, S., Wagner, L., and Miller, W. A Greedy Algorithm for Aligning DNA Sequences. *Journal Of Computational Biology*, 7:203–214, 2000.