Trellis Driver: Distributing a Java Workflow Across a Network of Workstations

Nicholas Lamb, Paul Lu, and Alona Fyshe
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada, T6G 2E8
{nlamb|paullu|alona}@cs.ualberta.ca

Abstract

Some applications in science and engineering consist of a main job that invokes, or drives, other jobs. For example, a server process may receive a request, then invoke a workflow of stand-alone scripts or executables to handle the request, and then generate the final response. Java's `Runtime.exec()` function allows jobs to be invoked from within a master Java program. However, these jobs are usually restricted to the same machine. If the number of jobs in the workflow is large, then it can be desirable to load balance the workload across different servers to maximize throughput.

We describe the design and implementation of the Trellis Driver, a newly-developed Java module that runs jobs using `TrellisDriver.exec()` and allows jobs to be scheduled across clusters and metacomputers (i.e., aggregations of servers). Using a Java-based bioinformatics application as a case study, we evaluate the performance improvement Trellis Driver offers through workflow parallelism.

1. Introduction and Related Work

1.1. Motivation

Some applications in science and engineering consist of a main job that invokes, or drives, other jobs and executables. For example, a server process may receive a request, then invoke a workflow of scripts to handle the request, and then generate the final response. Often, each script is a stand-alone application, executed as a separate process or job, that is invoked by the driver process. For example, there may be a pre-processing application, a main-computation application, then a post-processing application, organized as a pipeline. Overall control of the workflow, whether it is a pipeline or other shape, is often handled by a master script or driver.

Nearly all scripting languages and modern programming languages provide an Application Programming Interface (API) function for an application to run a command string as a separate process. ANSI C and Unix provide the `fork()` function, which creates a new operating system process, and the `exec()` function, which starts running a new program. The `system()` function automates creating and running a new process, and waiting for that process' termination status. However, `system()` blocks the caller until the command has been completed by the shell. Notably, Unix also provides `popen()` as a non-blocking, asynchronous function for running a command string. Similarly, Java provides the `Runtime.exec()` method to invoke executables that are external to the Java application [8]. In combination with Java threads, it is possible to develop concurrent Java applications to exploit multiprocessors.

Regardless of the specific application or language, if there are many jobs in the workflow, or if the jobs are very resource-intensive, then throughput may be improved by load balancing the jobs across a cluster or some aggregation of different servers. Restricting the concurrent processes or threads to a single server limits overall performance. Therefore, a variation of `system()` or `Runtime.exec()`, with similar semantics, which is integrated with a metacomputing system such as Trellis [12, 6], can improve performance.

1.2. Example Application: Proteome Analyst

Among the many optimization strategies applied to large scientific and engineering problems is workflow parallelization: exploiting concurrency between (**not** within) jobs with control-flow (or data-flow) dependencies. Although exploiting workflow concurrency is a well-known strategy, parallelizing large scientific applications at the job-level continues to be a challenge. In particular, the scheduling

and placement of hundreds or thousands of jobs in a workflow is typically handled by a batch scheduler, a grid infrastructure, or a metacomputer system, which are more complicated than simple library functions like `system()` or `Runtime.exec()`.

Proteome Analyst (PA) is a bioinformatics tool that provides a high-performance proteome classification framework [4] [13]. It consists of a Web interface and a driver program, written in Java, that invokes other bioinformatics and machine-learning applications. In bioinformatics, a classifier can be used to make predictions or annotations of a new protein. The value of predicting the properties of a new protein is in filtering the vast amounts of bioinformatics data and in suggesting the kinds of physical experiments that might be most valuable for a given protein. Currently, PA exploits process-level concurrency within a single server, but large analyses can require hours of computation and could benefit from aggregating multiple compute servers.

PA accepts a proteome (i.e., a set of protein sequences) in the form of text strings and then, based on a homology search and the extraction of information from databases of known proteins, makes predictions about properties of the new query sequence. When describing PA, we use the terms "sequence" and "protein" interchangeably. PA currently provides predictions or annotations for general protein function (i.e., what does the protein do) and subcellular localization (i.e., where in the cell the protein performs its main function), and will provide additional annotations in the future.

There are two distinct aspects to PA, but both have similar workflows and both are computationally-intensive (discussed in detail later and illustrated in Figure 3). First, PA can machine-learn a classifier as part of a training process. Using a training set of sequences with *known* annotations, PA automatically machine-learns a Naive Bayes (NB) classifier in a learn-by-example fashion. Second, PA can use a trained classifier to predict an annotation (or class label) for a new query sequence with an *unknown* class label.

From a high-level algorithmic standpoint, both training and prediction processes have three phases (Figure 1): First, the sequence (i.e., the primary structure of the protein represented as a string) is compared against the Swiss-Prot biological database of known sequences [1] using the well-known Blast toolset [3]. Swiss-Prot is a high-quality, curated database of known proteins and their various properties. During the training process, the sequence is from the training set. During the prediction process, the sequence is a new query sequence. The output from this string-matching step is a set of homologues, or proteins with similar primary structure. Second, the known information about the homologues is parsed to extract features or keywords from the Swiss-Prot database. Third, during training, a mapping function from the features to a class label is machine-learned. The result is a classifier. During prediction, a previously-trained classifier is used to map the features to a class label.

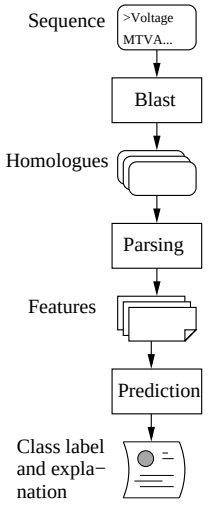

Figure 1. Job Pipeline for Classification

As shown in Figure 1, the prediction process consists of a sequential pipeline of jobs in which the output from one job is the input to the next. Each job is "small" in the sense that it has a short running time and small input and output data sizes. However, some proteomes have thousands or tens of thousands of proteins and so the challenge is in efficiently "calling out" to the different applications (e.g., Blast, the parsing tool) in an efficient manner and in a way that can exploit metacomputers for workflow concurrency.

1.3. Java Support for Workflow Parallelism

Previous to the development of Trellis Driver, PA used the Java 2 Platform API function `Runtime.exec()` to run Blast jobs for individual proteins. `Runtime.exec()` is non-blocking and returns a Process object. The caller may then invoke the `Process.waitFor()` method to wait until that process has completed and obtain its exit code. Thus, PA could theoretically issue many calls to `Runtime.exec()` and start multiple Blast jobs in parallel, and later call `Process.waitFor()` on each process to verify its outcome.

Unfortunately `Runtime.exec()` has the drawback that processes are usually run locally. An individual proteome may consist of tens of thousands of sequences, implying there may be tens of thousands of job pipelines for classification launched by PA. A single workstation is incapable of executing a workload of this size in less than a few hours, which is an unacceptably high turnaround time. To

effectively parallelize PA, the distribution of individual jobs over several machines is required.

We therefore need a replacement for `Runtime.exec()` that transparently schedules a job on any one of several available machines in a pool. While Java does offer the Remote Method Invocation (RMI) package that allows programs to invoke methods on remote objects, RMI only works between Java methods and programs. `Runtime.exec()` supports non-Java executables, as does our new Java package.

1.4. Trellis Driver

We have developed the Trellis Driver Java package to function as a drop-in replacement for `Runtime.exec()`. Trellis Driver provides a Java API that includes functionality for launching a new process, waiting for that process to complete, and obtaining its exit status. Trellis Driver is essentially a layer of software between a Java application (e.g., PA) and our underlying Trellis metacomputing system. Trellis Driver works across local and wide-area networks, offering security and load balancing. In contrast to mechanisms such as `fork()` or `system()`, which create new processes only within a single server, Trellis Driver distributes jobs across a network of workstations.

Trellis does work across geographically-distributed administrative domains, but our experimental results, in this paper, are limited to a local-area network. Specifically, we use a cluster of workstations as the hardware platform.

1.5. Related Work

The Globus Alliance is a well-known research and development effort to define new protocols and standards for grid computing [10, 11, 9]. Although powerful in concept and feature-rich, the Globus solution has the following drawbacks: 1) The Globus infrastructure is non-trivial to set up; 2) Administrators from all participating high-performance computing (HPC) centres must negotiate service level agreements among each other; and 3) Administrators must install and manage the grid software. For these reasons, we seek a simpler solution for resource integration.

In the absence of any grid infrastructure, there is a need to create an abstraction of a virtual supercomputer. The goal of *metacomputing* is to combine resources from multiple HPC centres, not by replacing existing infrastructure with a computational grid, but by building on top of the individual HPC systems. *Overlay metacomputing* provides resource aggregation at the user-level, meaning users may configure and use their own metacomputer without any system administrator support.

There are several automated job batching systems in widespread use. The Portable Batch System (PBS) is a feature-rich, multi-platform job queueing system [2]. Sun's N1 Grid Engine [7] is a resource management tool for distributed environments that includes a job queueing system. N1 Grid Engine allows the creation of grids that allow users to share resources across multiple domains. The problem with basing Trellis Driver on any one particular batch scheduler system is that it requires system administrators from all HPC centres to use the same local batch scheduler, which is an unrealistic assumption. Globus offers the promise that heterogeneous batch schedulers can be used within a single computational grid, but once again, all sites must first deploy the Globus infrastructure.

The purpose of this paper is two-fold. First, we describe the design and implementation of our Trellis Driver package, which integrates the PA application with the Trellis metacomputing environment. Second, as a case study, we describe the performance improvement Trellis Driver offers PA through workflow parallelism. Our experiments show Trellis Driver is effective at launching multiple jobs in a short time frame (typically on the order of seconds), and can provide, as expected, almost linear speed-up of data parallel application phases. Our main conclusion is that the PA system can be effectively integrated with the Trellis system, in order to take advantage of load balancing and other benefits, and that the Trellis overheads can be fully amortized.

2. Trellis System Architecture

2.1. Overview

Trellis is a thin layer of software that sits between applications and the infrastructure of HPC centres. Communication between domains is accomplished via message passing over Secure Shell (SSH) channels. Trellis uses the OpenSSH implementation of SSH.

The Trellis scheduler provides load balancing of workloads across multiple administrative domains. The Trellis system allows users to submit jobs to one or more *metaqueues* without concern about where and when those jobs will be run. A metaqueue is a queue of pending Trellis jobs that is visible to all hosts in the metacomputer. Users can create distinct metaqueues for different workloads, allowing them to run and monitor multiple applications concurrently.

Jobs are assigned to specific hosts through *placeholder scheduling*, whereby local batch queues interact with metaqueues to retrieve and execute jobs on demand. Placeholders are special-purpose programs running on a user's behalf on host machines. Each placeholder is associated with a specific metaqueue at the time it is started. The placeholder scheduling strategy follows a "pull" model in which jobs are retrieved from the metaqueue by programs on host machines. This is in contrast to a "push" model in which a central job server offloads jobs onto individual computers it

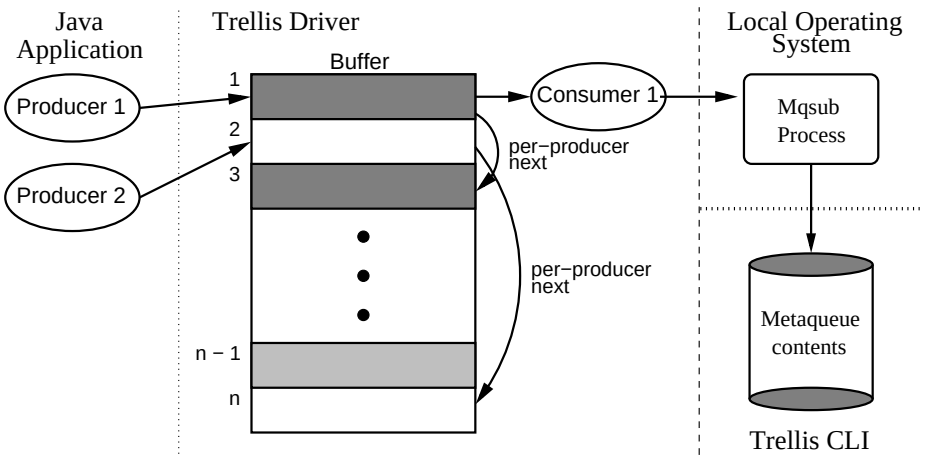

Figure 2. Bounded Buffer with Per-Producer Links

detects are idle. Since jobs are assigned on demand, 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 implicit load balancing across all hosts.

The Trellis Command Line Interface (CLI) is the front end of the Trellis job scheduler. The names and semantics of the Trellis CLI commands for submitting and monitoring jobs are based on those of the PBS batch scheduler. PBS provides the commands qsub, qdel, and qstat to add jobs, remove jobs, and list the contents of a local job queue, respectively. Trellis CLI provides the equivalent commands of mqsub, mqdel, and mqstat.

2.2. Trellis Driver Architecture

Trellis Driver is implemented as a Java package. Java applications import this package and call `TrellisDriver.exec()` to run jobs in the Trellis metacomputing environment. Communication between the higher-layer application code and the Trellis Driver is carried out through a well-defined API, which is described further below. Briefly, to launch a job, application code passes a command line to Trellis Driver through API calls. The application can then obtain that job's exit status through the API.

The dataflow resulting from a job submission can be viewed as a producer-consumer situation. Figure 2 illustrates the implementation. Threads within the application (producers) generate jobs while threads within Trellis Driver (consumers) process jobs by sending them to the underlying metascheduler. There is no limit on the number of producer threads, since this depends on the application code. We wish to avoid starting a 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. To decouple the producer count from the consumer count, Trellis Driver uses a standard bounded buffer to store incoming jobs. The bounded buffer is simply a limited-size storage space for command lines. Trellis Driver allows users to set the size of the bounded buffer and the number of consumers threads.

Applications can submit jobs either synchronously or asynchronously using 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 can later collect results from any or all jobs it previously submitted. This last task is done through work barrier functions provided by the Trellis API. Implementing work barrier functions requires keeping track of which jobs are submitted by which producers. Buffer entries contain special links to maintain this information, which we refer to as "per-producer next" links.

In Figure 2 we see there are two producers pushing jobs into the buffer while one consumer 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 new job to the buffer.

When Producer 1 calls a work barrier function to wait on all its outstanding jobs, Trellis Driver iterates through the linked lists of jobs belonging to Producer 1, waiting for each job to be marked as complete before moving onto the next one. 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 work barrier function return.

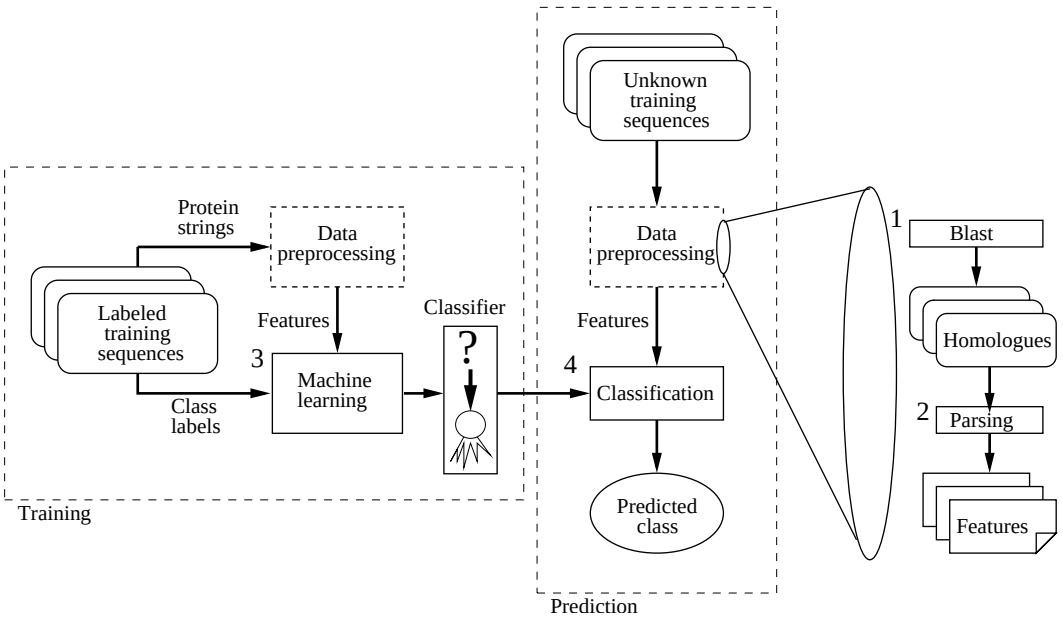

Figure 3. Training and Prediction Activities of Proteome Analyst

2.3. Trellis Driver API

The key Trellis Driver API functions are:

- `setGroup(`*`group_name, batch_factor`*`)`: Register a new job group (i.e., *group_name*), with a number indicating how many of such jobs are to be grouped together in a single mqsub command (i.e., *batch_factor*). This is further explained in Section 3.2.
- `exec(`*`command_line`*`)` [1]: Run the given command (i.e., *command_line*) in synchronous mode.
- `execAsynch(`*`command_line, prod_id`*`)`: Run the given command on behalf of the specified producer (i.e., *prod_id*), in asynchronous mode.
- `waitForOne(`*`key, prod_id`*`)` : Wait for the Trellis job with the given reference (i.e., *key*) that was submitted by the specified producer to finish.
- `waitForAll(`*`prod_id`*`)`: Wait for all Trellis jobs that were submitted by the specified producer to finish.

Application code that uses the standard Java runtime environment to run jobs in parallel can be ported to the Trellis environment with relative ease. The programmer first defines any job groups they desire using `setGroup()`. It is sometimes desirable to batch multiple small jobs of the same type (into a single call to mqsub) to amortize the overhead of starting a new job in Trellis. The programmer can issue `exec()` and `execAsynch()` calls as needed to run jobs either synchronously or asynchronously. The programmer can call `waitForOne()` or `waitForAll()` to await completion of one or several jobs at appropriate execution points.

[1] For consistency in function naming, we offer an alias to this method called `Trellis.execSynch`

3. Empirical Results: Parallelizing Proteome Analyst

Note that, although the Trellis system supports metacomputing across administrative domains and across a wide-area network, this following empirical results focus on a network of workstations hardware platform.

3.1. Proteome Analyst: Application Details

Proteome Analyst conveys information on the biological properties of protein sequences through classification. Each input sequence is mapped to a class that groups other sequences with similar properties. From an application perspective, classification involves two major activities: 1) Training, in which a classifier is built based on a set of proteins whose class memberships are known; and 2) Prediction, in which the classifier is used to predict the classes of previously unseen proteins.

Before any classifier can be trained or used for prediction, PA must perform data preprocessing to map each input

protein to a set of features or keywords that describe the protein. The data preprocessing consists of two substeps: finding homologues and extracting features. For every protein, PA first runs an application called Blast to find homologues, which are proteins with a common structure. PA then parses the homologues to extract a set of features. After data preprocessing, PA can use the extracted features along with the class labels from proteins in a training set to build a new classifier. Alternatively, PA can apply the feature lists to an existing classifier to predict class memberships of unlabeled proteins.

Figure 3 shows the flow of data and computation for the two main activities of PA. Rectangular boxes (e.g., Data preprocessing, Machine learning) represent application tasks while rounded boxes and icons (e.g., Labeled training sequences, Classifier) represent data elements that are passed from one task to the next. Note that the tasks shown for prediction are essentially a reproduction of the job pipeline shown in Figure 1.

To assess and improve the performance of all tasks involved in either training or prediction, we trained and validated a new classifier with a moderately-sized training set containing 3,916 sequences and a total of 2,632 different features. In practice, PA users have used training sets with up to 100,000 sequences and 6,500 features.

There are four phases of computation in this use case: 1) Blast; 2) Feature Extraction; 3) Machine Learning; and 4) Resubstitution. In the first phase, PA iterates over all sequences in the training set and runs Blast on each one, outputting the matching homologues in HTML files. In Feature Extraction, PA parses the HTML files to extract features corresponding to each input sequence. During the Machine Learning phase the classifier is constructed using the extracted features and class labels found in the training set. The fourth and final phase of Resubstitution entails running all of the sequences from the training set through the newly-built classifier to measure the error on the training set itself. Tasks in Figure 3 are labeled with the number of the corresponding computational phase.

Table 1 shows the average phase times and average total running time over five execution trials of the original PA, which executes all four phases sequentially. PA was run on a single Linux machine with two AMD Athlon MP 1800+ processors (1.533 GHz), 1.5 GB main memory, and Redhat v7.1. Although PA is multi-threaded, the key Blast phase discussed in this paper is single-threaded (i.e., it runs on only one processor). The HTML files outputted by phase one were stored on a locally-mounted disk volume. The feature lists produced by phase two were stored in a MySQL 4.0 database that was stored locally and had a server process running on the same machine as PA. The total running time of 6:52:55 for our moderately-sized training set provides empirical evidence for the need to reduce running time through parallelism. Notice that the first phase has the longest running time by far. In this paper, we focus on parallelizing Blast.

Phase	Run time (HH:MM:SS)
Blast	5:19:27
Feature Extraction	0:06:37
Machine Learning	1:08:17
Resubstitution	0:18:34
Total	**6:52:55**

Table 1. Phase Times for Training and Validation Pipeline (Sequential Version) of Proteome Analyst

3.2. Homogeneous Job Batching

The first two phases of PA preprocess the string representations of proteins to produce a list of features. Given the potentially large sizes of input proteomes, it is possible for PA to invoke Blast tens of thousands of times (i.e., once per sequence).

The original PA runs Blast on one protein at a time. The short running time of Blast on an individual protein makes for poor job granularity. In other words, the amount of computation in a single Blast execution is too short to justify running a job through Trellis. The ability to launch multiple Blast jobs in a single command line greatly amortizes the overhead of mqsub. We refer to this strategy of grouping together calls to the same program as *homogeneous job batching*. We refer to the number of jobs in a group as the *batching factor*.

In addition to job batching, the Blast phase can be accelerated through parallelism. Since this phase entails applying the same algorithm on multiple independent proteins, any number of Blast jobs may be run in parallel. The optimized Blast phase uses a combined strategy of homogeneous job batching and parallelism.

The original PA iterates through the training set, calling `Runtime.exec()` to run Blast on each sequence. Thus, the Blast executions are serialized. The parallel version of PA first registers the Blast job group with `TrellisDriver.setGroup()`, specifying the desired batching factor. Parallel PA then iterates through the training sequences and calls `TrellisDriver.execAsynch()` to start each Blast job. The actual concatenation of Blast commands into one command string that is passed to mqsub is done by Trellis Driver, and is thus transparent to the PA application. PA has only to specify in any call to `TrellisDriver.execAsynch()` that the incoming command is of type Blast. Finally, PA calls `TrellisDriver.waitForAll()` to await the com-

No. Place-holders	Phase Times (HH:MM:SS) for Varying Batching Factors (Original Blast time was 5:19:27)				
	1	2	4	8	16
2	3:22:05	3:02:13	2:49:06	2:44:09	2:40:34
4	1:42:34	1:30:50	1:25:03	1:21:47	1:20:33
8	0:55:31	0:46:47	0:42:58	0:41:10	0:40:55

Table 2. Phase Times for Parallel Blast

No. Placeholders	Batching Factor				
	1	2	4	8	16
2	1.58	1.75	1.89	1.95	1.99
4	3.11	3.52	3.76	3.91	3.97
8	5.75	6.83	7.43	7.76	7.81

Table 3. Speed-ups of Parallel Against Sequential Blast

pletion of all Blast jobs before proceeding onto the next phase.

Table 2 shows the average runtimes from five execution trials for the Blast phase in the parallel version of PA for varying batching factors and numbers of placeholders. Table 3 show speed-ups for the Blast phase. A batching factor of one means all Blast jobs run individually (i.e., no batching was done). Our platform was a cluster of Linux hosts connected by Fast Ethernet. Each host had the exact hardware and software specifications as the host used for measuring phase times for the sequential version of PA, as described earlier in Section 3.1.

From Table 3, we see that the speed-ups are reasonable for a data-parallel phase such as Blast. We do not achieve linear speed-up. A trade-off is made between the overheads of mqsub and the extra functionality provided by the Trellis system, especially in terms of load balancing the workload across multiple servers. Thus, mqsub's advantages outweigh the increased cost of scheduling jobs using Trellis.

Notice that there are diminishing returns as we increase the number of placeholders. With a batching factor of one (no batching), we obtain a speed-up of 1.58 when two placeholders are used. The speed-up of the Blast phase increases by 97% (i.e., almost doubles) as we move to four placeholders, reaching a respectable 3.11. With eight placeholders however, we achieve a speed-up of 5.75, a mere 85% increase over the four placeholder case.

One reason for this drop-off could be that the single Trellis CLI server, which constantly accepts requests for work by placeholders and hands out jobs, has a heavier computational load with a higher number of placeholders. Thus, the CLI server may be slower in dispatching jobs to placeholders, increasing the amount of time it takes for placeholders to pull jobs off of the server. Another explanation could be the overhead of SSH, which is magnified when more placeholders (and more SSH channels) are communicating with the CLI server simultaneously.

Performance scalability improves as the batching factor is increased. Not only do we approach linear speed-up, but the speed-ups scale with the number of placeholders. With a batching factor of eight, we observe a doubling of speed-ups as we move from two to four placeholders, and our speed-ups almost double as we move from four to eight placeholders. The speed-up is slightly lower than we would expect for the case of eight placeholders and a batching factor of 16. However, the phase time of 0:40:55 is just 12.8% of the original Blast time of 5:19:27.

These results show that homogeneous job batching achieves its objective of amortizing the mqsub overhead. Moreover, the job batching is transparent to the application, making it easy for the programmer to take advantage of this feature of Trellis Driver.

4. Future Work

In this work, we parallelized the Blast phase only. However, the sequential phase time for Machine Learning, which entails constructing the classifier, was also quite long (i.e., over an hour) relative to the remaining phases. Parallelization of classifier construction is currently underway [5].

We explained and motivated the need for homogeneous job batching. For reasons of data affinity, it is often practical to group together jobs of different types that constitute a job pipeline (i.e., heterogeneous job batching). This is of particular interest to Proteome Analyst, given the pipeline shape of its workflows. We are currently implementing this feature as well.

Finally, testing in an actual metacomputing environment that consists of an aggregation of geographically distributed servers is necessary to further verify the effectiveness of Trellis Driver, and to explore any issues related to job scheduling through Trellis over wide-area networks.

5. Concluding Remarks

Many applications in science and engineering have natural workflow parallelism. Some of these, including Proteome Analyst, consist of a driver process that creates thousands of processes or jobs, and could benefit from the ability to be placed and scheduled on multiple servers.

We have described the design and implementation of Trellis Driver. As a Java module, Trellis Driver has been used to parallelize the Proteome Analyst application by providing a simple API to link the familiar, existing process-based parallelism with the Trellis metacomputing system.

By exploiting the combined power of different servers, the PA and Trellis Driver combination can obtain reasonable throughput speed-ups of up to 7.81 on 8 processors (with 8 placeholders) for large phases such as Blast.

Our case study with Proteome Analyst suggests that driver-based applications can potentially have significant amounts of workflow parallelism that are well-suited for load balancing across a network of workstations.

6. Acknowledgements

Thank you to Duane Szafron, Russ Greiner, David Wishart, Brett Poulin, Roman Eisner, Zhiyong Lu, John Anvik, Cam Macdonell, David Meeuwis and the rest of the Proteome Analyst group.

This research was partially funded by research or equipment grants from the Protein Engineering Network of Centres of Excellence, the Natural Sciences and Engineering Research Council of Canada, Alberta Ingenuity Centre for Machine Learning, Sun Microsystems, SGI, and the Alberta Science and Research Authority.

References

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

[2] Altair Grid Technologies, [Online 2004]. `http://www.openpbs.org/`.

[3] S. Altschul, T. Madden, A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25:3389–3402, 1997.

[4] D. Szafron, P. Lu, R. Greiner, D.S. Wishart, Z. Lu, B. Poulin, R. Eisner, J. Anvik, and C. MacDonell. 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)*, 2003.

[5] N. Lamb. Scheduling Policies for Overlay Metacomputers. Master's thesis, Department of Computing Science, University of Alberta, 2004. in preparation.

[6] C. Pinchak, P. Lu, and M. Goldenberg. Practical Heterogeneous Placeholder Scheduling in Overlay Metacomputers: Early Experiences. In *Proc. 8th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 85–105, Edinburgh, Scotland, UK, July 24, 2002.

[7] Sun Microsystems Inc., [Online 2004]. Available: `http://www.sun.com/software/grid/SunClusterGridArchitecture.pdf/`.

[8] Sun Microsystems Inc. Java 2 Platform Standard Edition v1.4.2, API Specifi cation, [Online 2004]. Available: `http://java.sun.com/j2se/1.4.2/docs/api/index.html/`.

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

[10] The Globus Alliance, [Online 2004]. Available: `http://www.globus.org/research/papers/ogsa.pdf/`.

[11] The Globus Alliance. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, [Online 2004]. Open Grid Service Infrastructure WG, Global Grid Forum, `http://www.globus.org/`.

[12] Trellis Project, [Online 2004]. `http://www.cs.ualberta.ca/~paullu/Trellis`.

[13] Z. Lu, D. Szafron, R. Greiner, P. Lu, D.S. Wishart, B. Poulin, J. Anvik, C. Macdonell, and R. Eisner. Predicting Subcellular Localization of Proteins Using Machine-Learned Classifi ers. *Bioinformatics*, 20(4):547–556, 2004.