Biocompute: Towards a Collaborative Workspace for Data Intensive Bio-Science

Rory Carmichael
Bioinformatics Core Facility
University of Notre Dame
Notre Dame, IN 46556
rcarmich@nd.edu

Patrick Braga-Henebry
IMC Financial Markets
233 South Wacker Drive
#4300
Chicago, IL 60606
pbh101@gmail.com

Douglas Thain
Dept of Computer Science
and Engineering
University of Notre Dame
Notre Dame, IN 46556
dthain@nd.edu

Scott Emrich
Dept of Computer Science
and Engineering
University of Notre Dame
Notre Dame, IN 46556
semrich@nd.edu

ABSTRACT

The explosion of data in the biological community demands the development of more scalable and flexible portals for bioinformatic computation. To address this need, we put forth characteristics needed for rigorous, reproducible, and collaborative resources for data intensive science. Implementing a system with these characteristics exposed challenges in user interface, data distribution, and workflow description/execution. We describe several responses to these challenges. The Data-Action-Queue metaphor addresses user interface and system organization concepts. A dynamic data distribution mechanism lays the foundation for the management of persistent datasets. The Makeflow workflow facilitates the simple description and execution of complex multipart jobs. The resulting web portal, Biocompute, has been in production use at the University of Notre Dame's Bioinformatics Core Facility since the summer of 2009. It has provided over seven years of CPU time through its three sequence search modules — BLAST, SSAHA, and SHRIMP — to ten biological and bioinformatic research groups spanning three universities. In this paper we describe the goals and interface to the system, its architecture and performance, and the insights gained in its development.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Design, Human Factors, Performance, Experimentation

HPDC'10, June 20–25, 2010, Chicago, Illinois, USA.

1. INTRODUCTION

The field of biology is becoming increasingly reliant on computation. Data collection machines and methods continually decline in cost and increase in throughput, generating a profusion of data. This explosion of data has grown the collaborative field of bioinformatics, which has in turn generated many powerful tools for the exploration of large biological datasets. However, despite the convenient parallelism and demanding computational requirements present in many of these applications, there continues to be a dearth of easily deployable parallel bioinformatics tools. Though many popular resources for bioinformatic analysis exist, data acquisition has rapidly outpaced their analysis and curation capabilities.

We propose a more collaborative way to do data intensive scientific computing. Our key goals are to provide collaborative, rigorous, and repeatable analytic results. To achieve these goals we propose a system in which user data are primary concerns, computational resources can be shared and managed, data and computational workflows are parallel, distributed, and replicable, and parameters of analysis are persistently maintained. By allowing scientific communities to pool and curate their data and resources in this way, we hope to increase the scalability of scientific endeavors beyond the scope of monolithic central resources.

To support this collaborative structure, we suggest the Data-Action-Queue interface metaphor. This model provides users with a view of their past and current analyses, their raw data, and the tools they use to analyze that data. Sharing capabilities allow users to provide their collaborators with access to user data and job information. The modular design suggested by this metaphor permits tool developers to provide the community with new Actions without having to re-implement interfaces to source data or job runtime information.

In pursuit of these goals, we have implemented the Biocompute web portal. Biocompute serves community needs by providing biologists and their bioinformatician collaborators a Data-Action-Queue based environment where datasets can be readily shared and analyzed, results are automatically documented and easily reproducible, and new tools can be readily integrated. It runs on top of Chirp and Condor to facilitate distributed storage and computation on shared resources.

Dataset distribution, effective description and execution of workflows, and user interface all proved to be challenging in this context. In this paper, we provide a detailed description and evaluation of our data distribution methods. We document our use of the Makeflow [15] abstraction as a solution to the workflow description and execution problem, and we discuss the Data-Action-Queue interface model.

As of the writing of this paper, Biocompute is in production use by the University of Notre Dame's Bioinformatics Core Facility. Since its initial deployment in the summer of 2009, it has provided over seven years of CPU time through its three sequence search modules — BLAST [1], SSAHA [11], and SHRIMP [12] — to ten biological and bioinformatic research groups spanning three universities.

2. SYSTEM GOALS

There exist bioinformatics portals for a variety of biological data. These range from broad-base resources, such as NCBI [7], to more specific community level resources, such as VectorBase [8], down to organism or even dataset level web portals. While these portals all rely, at least in part, on the scientific communities they support for data and analysis, they share the characteristics of centralized computation and curation. Additionally, many existing portals suffer from imperfect data transparency or job parameter storage, reducing the rigor and reproducibility of the results generated. As increasing number of organisms are sequenced, smaller and less well funded biological communities are acquiring and attempting to analyze their data. These communities rarely have the resources to support the development of specialized community web portals, and find portals such as NCBI insufficient for their computation, customization, or collaboration needs.

It seems natural, then, to turn to a more rigorous, reproducible, and collaborative way to do data intensive science. We believe the following characteristics to be vital to these goals.

1. User data must be able to integrate with the system just as well as curator provided data.
2. Data management should be simple for owners as well as site administrators.
3. Sharing of data and results should be straightforward.
4. Persistent records of job parameters and metadata need to be kept.
5. Jobs should be easily resubmitted in order to reproduce results.
6. System resources should be shared fairly, productively and transparently.
7. The resources providing computation and data storage must be scalable and shareable.

A system meeting these characteristics should permit a user community to develop and improve a shared resource that is capable of meeting their computational needs and contains the data they require. Further, it will allow users to maintain a clear, traceable, record of the precise sources of datasets and results.

3. DATA-ACTION-QUEUE

Having the functionality required in section 2 is insufficient if users cannot effectively use the resource. To provide the requisite interface, we employ the Data-Action-Queue (DAQ) interface metaphor. Like Model-View-Controller, this suggests a useful intellectual structure for organizing a program. However, DAQ describes an interface, rather than an implementation.

The DAQ metaphor rests on the idea that users of a scientific computing web portal will be interested in three things: their data, the tools by which they can analyze that data, and the record of previous and ongoing analyses. We believe this metaphor interacts well with the laid out requirements. The Data view provides users with means to think about, act on, and share their custom data. The Action view provides users access to the analysis tools in the system. This also suggests a modular design for the implementing system. If tool developers need only specify the interface for the initial execution of their tool — without having to concern themselves with the incoming data, or with recording parameters and tracking progress — it greatly simplifies the addition of new actions to the system. The Queue view documents job parameters and meta-information, and permits users to drill down to a single job in order to resubmit it or retrieve its results. Since the Queue also shows the ongoing work in the system, it gives users a simple way to observe the current level of resource contention in the system, and the degree to which they contribute to that contention. Figure 1 shows this model.

4. IMPLEMENTATION

Here we present our preliminary implementation of these ideas into the Biocompute web portal. First, we describe our implementation of the DAQ interface metaphor. Second, we describe the system architecture underlying the metaphor. Third, we discuss our use of the workflow engine Makeflow [15] to provide tool developers with a simple and expressive interface to Notre Dame's distributed computing and storage resources. We then describe our solution to the data distribution challenges introduced by our choice to include the BLAST [1] tool in Biocompute's available actions, and, finally, provide performance results for this solution.

4.1 Software Architecture

4.1.1 System Interface

In accordance with the DAQ model, users have three separate views of Biocompute: a filesystem like interface to their data, a set of dialogues permitting users to utilize actions

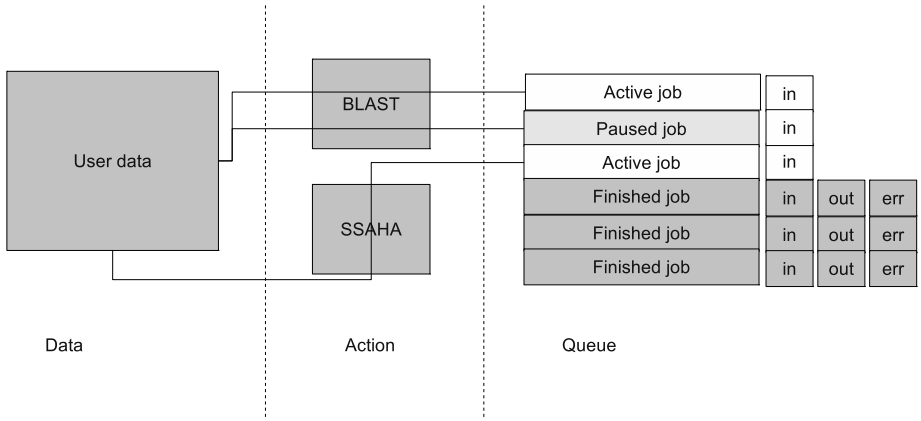

Figure 1: Biocompute interface model.

provided by the system, and a queue storing the status, inputs, and outputs of past and current jobs. Figure 1 lays out this model, and the following sections describe the interaction and sharing characteristics of its components.

4.1.2 Data

Our system stores, generates, and manipulates data. Users interface with their uploaded data much as they would in a regular filesystem. They can move files between folders, delete them, and perform other simple operations.

The same interface used to perform these operations can be used to promote files to datasets. To perform such a promotion, a user enters into a dialogue customizable by Biocompute tool developers. This screen permits users to set metadata as well as any required parameters. Once the selected file has been processed by the appropriate tool, the resulting data are distributed to a subset of the Biocompute Chirp [14] stores. The meta-information provided by the users is stored in a relational database, along with other system information such as known file locations. An example in which a tool uses metadata to improve performance is documented in section 4.4.

The data stored in biocompute and the datasets generated from it are often elements of collaborative projects. User files and datasets may be marked public or private. A publically available dataset is no different from an administrator created one, allowing users to respond to community needs directly. This primarily facilitates collaboration between biologists.

4.1.3 Actions

Tools provide the core functionality of Biocompute. As Biocompute grew from a distributed BLAST web portal to a bioinformatics computing environment, we quickly realized a need for a flexible and low maintenance method for integrating new tools into the system. Specifically, we needed to provide application developers with simple hooks into the main site while providing enough flexibility for including diverse applications. Further, it was important that application developers be provided with a simple and flexible tool for describing and exploiting parallelism provided by the underlying distributed system.

From these requirements, modules emerged. Conceptually, each module consists of an interface and an execution unit. The interface provides a set of php functions to display desired information to users via the web portal. The execution component is a template for the job directory used by Biocompute to manage batch jobs. So far, each Biocompute module utilizes a local script to generate and run a *Makeflow* [15] for execution on the distributed system. Further discussion of this tool can be found in section 4.3.

In creating this system we greatly facilitated the collaborations between biologists and bioinformaticians. A tool developer need not be intimately familiar with biocompute in order to develop a module. In fact, two of the three available modules have been developed by computer science grad students not otherwise involved with biocompute.

4.1.4 Queue

While the distributed system beneath Biocompute may not be of general interest to its users, the details of job submissions and progress are important. Biocompute provides methods for users to recall past jobs, track progress of ongoing jobs, and perform simple management such as pausing or deleting jobs. Users may view currently running jobs or look through their past jobs. Each job has drill down functionality, permitting a user to look through or download the input, output, and error of the job, and to see pertinent meta-data

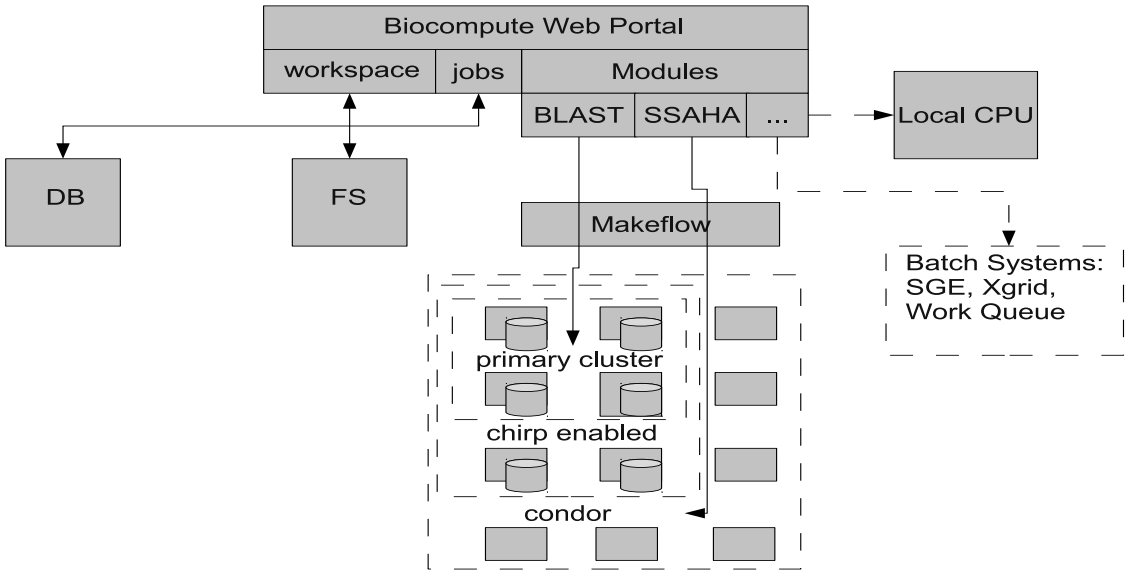

Figure 2: Architecture of Biocompute. The dotted lines indicated possible but unimplemented components.

such as job size, time to completion (or estimated time to completion if the job is still running), and parameters used to generate results.

For system administrators and developers, a page showing the status of the Condor queue for Biocompute jobs and status messages for all the computers advertising a willingness to run Biocompute jobs is also available.

The Queue facilitates collaboration in two primary ways. As with user data, queue jobs may be marked public or private. Making a job public effectively exposes the source data, parameters, and results to inspection and replication by other Biocompute users. Furthermore, the currently running Queue and the Condor queue detail pages provide curious users with a way to evaluate the current resource contention in the system. We discuss the impact of this and other feedback mechanisms in section 4.6.

4.2 System Description

Biocompute is arranged into three primary components. A single server hosts the website, submits batch jobs, and stores data from those jobs such as input and output files. A relational database stores metadata for the system such as user data, job status, runtime and disk usage statistics. Each dataset is stored on a Chirp [14] cluster of 32 machines that has been integrated into the Condor distributed computing environment. These machines serve as a primary runtime environment for batch jobs and are supplemented by an extended set of machines running Chirp that advertise availability for Biocompute jobs using the Condor classad system.

4.3 Workflow Engine for Modules

Makeflow [15] provides application developers needed simplicity and flexibility for developing distributed applications. Briefly, a developer uses makeflow with make-like syntax to describe dependencies and execution requirements for their tool (see figure 3 for an example makeflow). The makeflow engine is capable of converting this workflow into the necessary submissions and management of batch jobs. Though Biocompute currently relies on Condor [9], makeflow can use diverse batch systems and therefore provides the mechanism for our system to extend to other batch systems with minimal changes. Makeflow also provides mechanisms for fault tolerance that have proven invaluable for maintaining correctness in large Biocompute jobs.

4.4 Data Management for BLAST

Here, we describe developing and improving our BLAST functionality. We briefly describe the nature of the BLAST tool and the data management problem it presents. Next, we present the distributed BLAST tool used in Biocompute. Finally, we describe the performance impact of our approach and discuss ways of mitigating possible negatives in future versions of the Biocompute system.

4.4.1 Problem Description

Alignment is a process by which sequences—discrete strings of characters representing either the DNA bases or amino acids of proteins derived from an organism—are compared to one another. Alignments are accompanied by a score

```
2111.input.0 2111.input.1 : 2111.input job.sched gen_submit_file_split_inputs.pl
	LOCAL ./gen_submit_file_split_inputs.pl 2111

2111.output.0 2111.error.0 2111.total.0: 2111.input.0 distributed.script job.params
	./distributed.script 0

2111.output.1 2111.error.1 2111.total.1: 2111.input.1 distributed.script job.params
	./distributed.script 1

2111.output: 2111.output.0 2111.output.1
	LOCAL touch 2111.output;find . -name "2111.output.*" -exec cat {} \\\; >> 2111.output

2111.error: 2111.error.0 2111.error.1
	LOCAL touch 2111.error;find . -name "2111.error.*" -exec cat {} \\\; >> 2111.error

2111.total: 2111.total.0 2111.total.1
	LOCAL touch 2111.total;find . -name "2111.total.*" -exec cat {} \\\; >> 2111.total

2111.complete: 2111.output 2111.error 2111.total finishjob.sh
	LOCAL ./finishjob.sh 2111
```

Figure 3: Example Makeflow file.

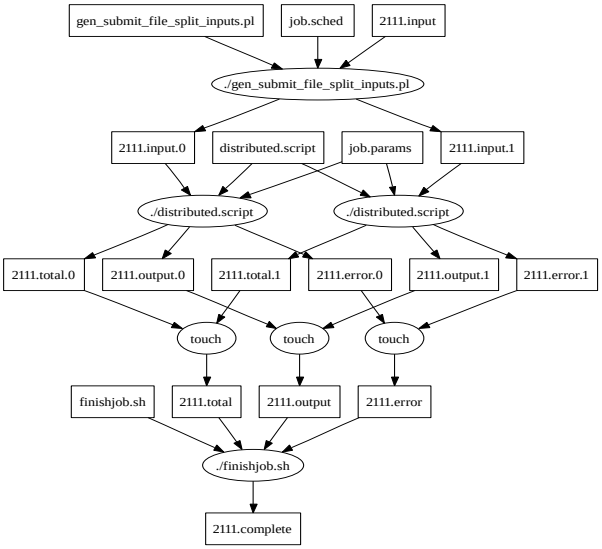

Figure 4: Graph of the execution of a small workflow. The distributed acyclic graph of this job is typical of Biocompute jobs .

noting the similarity of the two sequences, and a visual representation that lines up corresponding characters in both strings. Such alignments can be invaluable to biologists attempting to identify the function, origin, or structure of biological sequences through comparison to other well-studied sequences. While optimal alignments can be performed algorithmically using a technique known as Smith-Waterman [13], the quadratic runtime of this technique makes it impractical for large problems.

BLAST [1], or Basic Local Alignment Search Tool, is a commonly used bioinformatics tool that implements a substantially faster heuristic to align a set of one or more query sequences against a set of reference sequences. The sequences can be either DNA or amino acid strings, and BLAST has options that permit comparison of any query type to any type of preformatted database. Because biologists often wish to compare hundreds of thousands of sequences against databases containing gigabytes of sequence data, these jobs can take prohibitively long if executed sequentially. However, BLAST jobs are conveniently parallel in that the output of a single BLAST is identical to the concatenation of the output from BLASTs of all disjoint subsets of desired query sequences against the same reference database. Because most BLAST jobs are run against the same small subset of databases, we avoid considerable data transfer overhead by pre-staging BLAST databases where computation is expected to take place. This allows us to run a distributed BLAST across the campus grid.

While pre-staging is a simple solution to the data transfer problem, it falls short in a number of ways. First, it is impractical and expensive to stage databases on every machine available. Even if this was possible, it would be wasteful as the majority of jobs on our Biocompute system can be run quickly on our dedicated cluster of 32 machines. Further, some user generated databases are used rarely while common system databases are heavily used.

Initially, we sought to address these difficulties by only running BLAST jobs on a cluster of 32 machines administered by the Cooperative Computing Lab (CCL) at Notre Dame. We modified the Condor settings on these machines to give priority to jobs initiated by Biocompute. BLAST databases were prestaged to this cluster. While this solution was simple, it became insufficient as the demands on the system increased. Disk failures on some older machines in the cluster quickly undermined an assumption of data homogeneity, and resulted in mismatches and unused machines. Network outages, machine downtime, and other factors helped prevent uniform distribution of newly created databases. Finally, a lack of a coherent system to track the locations of working databases made maintaining consistency in the cluster very difficult.

The next iteration of the system used a set of tables in Biocompute's relational database to store database sizes and locations. Datasets were tracked generically, with a separate table storing BLAST-specific metadata. The addition of a few simple screens based on these tables permitted users to track detailed information regarding their own databases. This modification, along with a few simple scripts to distribute datasets to new machines, provided additional flex-

ibility and power when based on a host cluster. Shortly after implementing this solution, however, a number of new machines were added to our campus grid. As these machines had Chirp and Condor resources, it made sense to leverage these underutilized resources for Biocompute's computational tasks. Because these systems were spread between three physically distinct locations, we supplemented our static distribution and tracking scheme with dynamic data distribution to adjust to a larger, more heterogenous computing environment.

Given new resources, it seemed reasonable to examine moving databases at runtime. By transferring required databases on a per distributed task basis we hoped to avoid unacceptable costs in terms of bandwidth and disk utilization. We wanted to avoid unnecessary file transfers, and efficiently utilize existing resources whenever possible. Additionally, we wanted to be sure that the network load incurred by runtime file transfers was well distributed and that failures didn't result in data corruption or any future errors.

We achieved these goals by carefully modifying the script used by the BLAST module. The previous script used a simple check to determine presence or absence of any of the databases needed. A failure to find the appropriate database was reported such that the Condor job would be evicted and rescheduled rather than being considered complete with error. This provided a failsafe if the assumption of uniform database distribution was violated (as it often was in our experience). We used Chirp's put and get functions to perform data transfers and performed several tests for correctness. BLAST's wrapper script was modified to exit with the "database not found" error code if the data transfers failed.

While these modifications ensured that failed data transfers didn't leave a lasting impact, they did nothing to mitigate spurious data transfers. To minimize the amount of unnecessary data transfer, we decided to experiment with Condor's rank functionality. Condor provides a convenient option by which users can define a rank equation for a given job. This rank equation calculates the value of a given machine based on characteristics advertised. Jobs are run on the highest ranked available machine. We found that using the dataset metadata stored in our relational database, we could rank machines by the number of bytes already pre-staged to that machine. This rank function schedules jobs to the machine requiring the least dynamic data transfer. While this approach naturally ties our BLAST module to Condor, similar concepts could be used to provide job placement preferences in other batch systems.

This technique requires that the list of database locations be correct. To maintain correct lists, we use the following method. Any successful job was run on a machine containing all required databases. Jobs returning with the database not found error are known to lack one of the needed databases. Therefore, we parse job results and update the mysql database with data locations accordingly. For those jobs that run against only one database, this is sufficient information to update the database, however in the general case we cannot simply determine the appropriate action and therefore make no modifications to the database.

To balance load, we transfer databases from a randomly selected machine in the primary cluster. If the chosen machine does not to have the appropriate database, the transfer fails and the job is rescheduled.

Even with random source selection load balancing, potential network traffic is significant. For example, the largest BLAST database hosted in Biocompute is over 8 GB. Further, many Biocompute jobs run in hundreds or even thousands of pieces, and therefore it is possible that a job could request 8GB transfers to dozens or even hundreds of machines simultaneously. To mitigate this unreasonable demand we limit the number of running jobs to 300. Additionally, we cap transfer time at 10 minutes. This effectively prevents us from transferring files overly large for the available bandwidth. Under high load or over low speed networks only small files will be transferred, whereas connections with abundant bandwidth will support the movement of significant databases. Concisely put, the process is as follows:

1. Check for database in local Chirp store
2. if missing database initiate database transfer from a random host, else run normally
3. if transfer successful run move the database to the expected location in the Chirp server and set ACLs to permit local read and execute.
4. repeat 1-3 until all necessary databases are acquired or one of 2, or 3 fail.
5. if failure, return with database not found error code, else run executable
6. On logfile parse (triggered by user accessing a job page), mark all machines hosting successful runs as possessing all databases used by job.

4.5 Semantic Comparison of Manual and Dynamic Distribution Models

The transition from manual to dynamic distribution required a shift in dataset storage semantics within Biocompute. This shift was brought about by the new authentication requirements introduced by node to node copying, and by the automatic updating characteristics provided by our logfile inspection technique. In Table 1 we document the characteristics of datasets before and after dynamic distribution. In Table 2 we describe the failure and execution semantics for several common dataset operations.

4.5.1 Performance of Data Management Schemes

In this section we will explore the performance characteristics of Biocompute, and evaluate the impact of data distribution model, and data availability on these characteristics.

Table 3 illustrates the cost of the current timeout policy for the dynamic distribution model, as compared to the original static distribution model. Figure 5 shows the runtime of the dynamic BLAST query, which was executed against the NCBI Non-Redundant (NR) BLAST database. This dataset is 7.5GB - well outside the transferable size for our network. This effectively causes any job assigned to a machine without the database to immediately reschedule itself (in the

Table 1: Object Characteristics Before and After Implementing Dynamic Distribution

Object	Before Dynamic		After Dynamic	
	record	permissions	record	permissions
New Primary Machine Replica	local fs	Biocompute, local	global	Biocompute, local, nd
Audited Primary Machine Replica	global	Biocompute, local	global	Biocompute, local, nd
New Dynamic Replica	n/a	n/a	global	Biocompute, local

Table 2: Operation Characteristics Before and After Implementing Dynamic Distribution

Operation	Before Dynamic					After Dynamic				
	atomic	synchronous	recorded	times out	automatic	atomic	synchronous	recorded	times out	automatic
Create	no	yes	yes	no	no	no	no	yes	no	no
Delete	yes	yes	yes	no	no	yes	yes	yes	no	no
Make Replica	no	yes	yes	no	no	yes	no	yes	yes	yes

Table 3: Worst case cost of dynamic distribution. While the dynamic case shows a 167% increase in badput, it only suffers an 18% increase in runtime

Distribution Method	Execution Time (hours)	Badput (hours)
dynamic	17.09	517.3
static	14.49	193.7

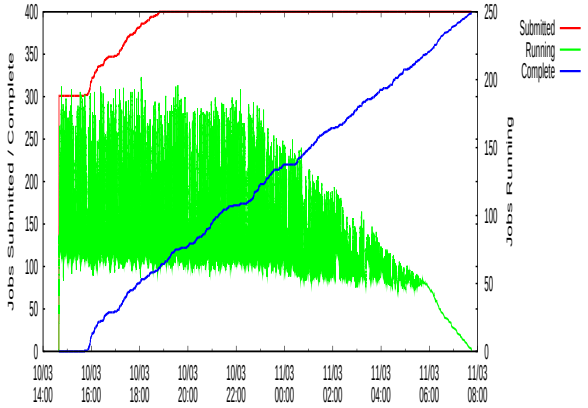

Figure 5: Worst case dynamic distribution run. The high variation in number of jobs running is a result of failed data transfer attempts.

static case) or to wait for some time and then reschedule (in the dynamic case). Though one might expect this behavior to significantly increase the overall runtime, we only observe an 18% increase in runtime for the dynamic distribution worst case run. The characteristics of the rank function ensure that dynamic jobs are only assigned to database-less machines when all of the usable machines are already occupied. In the static case, these jobs would simply wait in the queue until a usable machine became available. Essentially, the change to dynamic distribution forces a transition to a busy-wait. While this increases badput, and is perhaps suboptimal from the perspective of users competing with Biocompute for underlying distributed resources, it has a limited impact on Biocompute users.

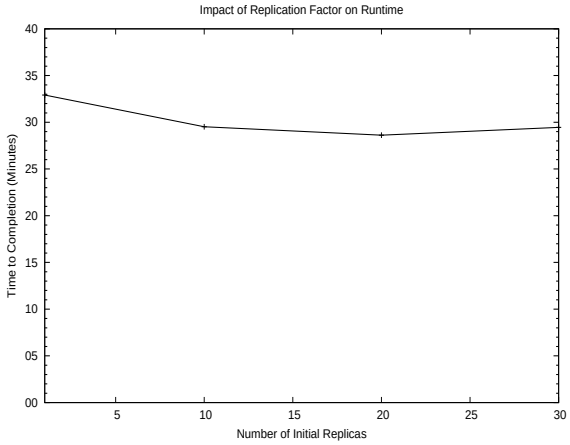

Figure 6: Runtime of BLAST versus a medium sized (491 MB) database for varying initial dataset replication level

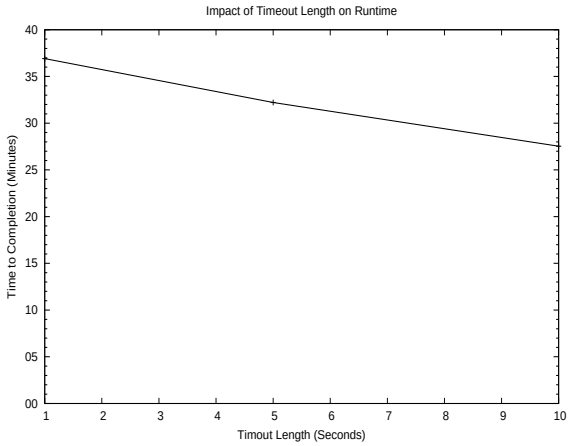

Figure 7: Runtime of BLAST versus a medium sized (491 MB) database for varying timeout lengths.

Figure 6 shows the impact of initial replication level on the runtime of a BLAST job. It is important to note that the Rank function remains unmodified throughout the runtime of a particular job, so the number of jobs arriving at machines which have acquired the required databases during runtime is statically determined. The naivety of our source selection algorithm ensures that many transfer attempts will fail if the primary cluster is saturated. This reduces our ability to grow the level of parallelism as the job progresses. However, because 300 subjobs may be submitted at any one time, it is likely that each wave of execution will propogate at least a few databases. The results demonstrate that over the course of a job the distribution task is accomplished quickly.

Figure 7 shows the impact of varying timeout times on the runtime of a BLAST job. For this test, each database was initially distributed to the 32 core nodes and, depending on timeout, potentially distributed to any of the nodes open to Biocompute jobs. The lowest timeout time never transfers the target database, the middle one sometimes succeeds and sometimes fails, and the final one always succeeds. As expected, the increased parallelism generated by successful distribution reduces runtime.

In both of our experiments, long tail runtimes for some subjobs created limitations on the benefit of increased parallelism. A more sensitive input splitting scheme might be better able to control for this.

Our final timeout value was set high in order to maximize the transferrable database size, as the impact on performance was acceptable even for untransferable databases. Our final replication value was set to 32, the size of our dedicated cluster. Since the introduction of dynamic distribution, some datasets have been spread to up to 90 machines, tripling the parallelism available for those data.

4.6 Social Challenges of Biocompute

Up to this point, we have only addressed how Biocompute meets the purely technical challenges described in section 2. However, as a shared resource and a nexus of collaboration, Biocompute requires several mechanisms to balance the needs of all of its stakeholders. We believe that this is best achieved through fair policies and transparency. In this section we discuss the policies and tools we have used to facilitate the management of our two most limited resources - data and compute time.

4.6.1 Data

While the Center for Research Computing (CRC) at Notre Dame generously provides campus groups with several terabytes of storage at no cost, the sheer size and number of available biological datasets require consideration of disk space costs of Biocompute. The two simplest ways of achieving this are motivating users to control their disk consumption, and convincing our users to provide or solicit funding to increase available space. In either case, it is necessary to provide users with both comparative and absolute measures of their disk consumption. To this end we track disk space consumption and CPU utilization for all users, and publish a "scoreboard" within Biocompute. Pie-charts have thus far provided particularly heavy users with sufficient motivation to delete jobs and data that they no longer need, and have given us useful insights into common use cases and overall system demand.

4.6.2 Compute Time

Resource contention also comes into play with regards to the available grid computing resources. Utilization of Biocompute's computational resources tends to be bursty, generally coinciding with common grant deadlines and with the beginnings and ends of breaks when users have extra free time. These characteristics have produced catastrophic competition for resources during peaks. Without a scheduling policy, job portions naturally interleaved, resulting in the wall clock time of each job to converge on the wall clock time necessary to complete all the jobs. To combat this problem we implemented a first in first out policy for large jobs, and a fast jobs first policy for small (less than five underlying components) jobs. This has allowed quick searches to preempt long running ones, and prevents the activities of users from seriously impacting the completion timeline for previously started jobs. This modification also made Biocompute's job progress bars into accurate predictors of remaining runtime, rather than a simple visualization of work already completed. Significantly, system users were much more satisfied (as measured in substantially reduced complaints) following these changes.

5. RELATED WORK

The provenance system at work in Biocompute bears similarity to the body of work generated by The First Provenance Challenge [2, 3, 5, 10]. For biologists, the queue and detail views provide extremely high level provenance data. At the user interface level, we were most interested in giving the user an easily communicable summary of the process required to produce their results from their inputs. To this end, Biocompute displays the command that would be used to execute the equivalent BLAST job using only the unmodified serial executable. This model hides the complexity of the underlying execution, and gives our users a command that they can share with colleagues working on entirely different systems. Obviously, such information would be insufficient for debugging purposes, and Makeflow records a significantly more detailed record of the job execution, including the time, location and identity of any failed subjobs, and a detailed description of the entire plan of execution. However, while the records necessary to support detailed provenance queries are available, we have thus far relied exclusively on manual inspection for debugging and troubleshooting. In contrast, formal provenance systems are expected to have a mechanism by which queries can be readily answered [10]. We selected Makeflow as our workflow-engine for its simplicity, and its ability to work with a wide variety of batch systems. A similarly broad workflow and provenance tool could be selected in its place without modifying the architecture of Biocompute.

BLAST has had many parallel implementations [4,6]. While the BLAST module of Biocompute is essentially an implementation of parallel BLAST using makeflow to describe the parallelism, we do not mean to introduce Biocompute as a competitor in this space. Rather, we use BLAST to highlight common problems with the management of persistent datasets, and show a generic way to ease the problems gen-

erated without resorting to database segmentation or other complex and application-specific techniques. The BLAST module, like all of Biocompute's modules, uses an unmodified serial executable for each subtask.

6. CONCLUSIONS

6.1 Usability

As with any service, the true metric of Biocompute's success is the degree to which it assists those it is meant to serve. During the past year, Biocompute has grown from a fledgling tool, only used by its bioinformatician developers, to a workhorse for a wide variety of users. Regular users range from faculty to support staff to students, and cover all areas of computational expertise from faculty in Computer Science to undergraduate biology majors; they span 10 research groups and 3 universities. The SSAHA and SHRIMP modules were developed by computer science grad students without prior knowledge of the properties of SSAHA, SHRIMP, or Biocompute. In a year we provided our users with more than seven years of CPU time, and enabled them to perform research using datasets that are not available or explorable anywhere outside of Notre Dame.

6.2 Challenges Revisited

The primary technical challenges encountered in Biocompute stemmed from the absence of effective systems to manage data and execution. Our work with the Makeflow abstraction has convinced us that generally deployable tools are needed. Decoupling parallelization of Biocompute's modules by basing them on Makeflow has provided consistent semantics and interfaces for our batch jobs for independent tool development.

The burden of managing dataset replicas, however, has required module specific modifications and a significant amount of custom development, which would have to be repeated in the current implementation. The addition of dynamic data transfer impacted the permissions semantics for primary replicas. Permission complications and monitoring requirements limited our ability to communicate and rely on data at various levels. For instance, tasks are unable to modify their execution parameters to take advantage of new information regarding dataset locations. While the concepts learned from our approach to dataset management for the BLAST module are generally applicable, the implementation of a system using those ideas is essential for the continuing development. Such a system would require a simple tool for the replication, distribution, location, verification, and access control of large files in a distributed system in addition to providing hooks to facilitate on-demand replication.

The requirements of modularity also proved a significant source of complexity. For a modular system to be effective it must encapsulate many possible concepts within a single usable framework - essentially an abstraction. The components of our module abstraction - a parameter file, an execution workflow, and a set of presentation functions - form the rough shape of any non-interactive program. Adding interactive functionality would present a significant challenge to Biocompute, and would likely require an alternative system architecture. Furthermore, the constraints on the interface between module and system increased the complexity of permissions management significantly. Consider that permitting a module's sub-jobs the ability to intelligently update module specific tables at runtime would require granting not only the module, but all distributed machines in the grid system, the ability to write to some elements of our system infrastructure. While this would undeniably provide additional power to modules, it would also incur significant risk - even within the boundaries of Notre Dame's internal network.

6.3 Closing Observations

Finally, we return to our motivating goals. We stated that our system should integrate user data, and make its management simple. Job parameters and meta-information should be kept and results easily shared and resubmitted. System resources should be scaleable, and fairly, productively and transparently shared.

Our system has, at least in pilot form, achieved these goals. User datasets are indistinguishable from administrator created datasets. Job parameters and meta-information are kept and shared, and jobs can be resubmitted with two clicks. Condor and Chirp, our system backends, are time-tested scaleable shared resources. We report relative usage of both space and computation, and maintain a fair scheduling scheme. We facilitate access to this system with the Data-Action-Queue interface.

However, our solutions are for the most part initial steps. By implementing the current version of Biocompute we have exposed a need for a much more complex dataset management tool. With the addition of such a tool, Biocompute's ability to usefully classify user data and vary its storage and presentation policies based on these classifications could be expanded to any kind of data. Our first attempt at implementing a Data-Action-Queue interface came out effective, but complex. Finally, we find that user data is constrained by website upload speeds. User data cannot become fully integrated until it can be accessed directly from the source machines. At Notre Dame, in-house biological data generation is commonplace, and an effective system to pipeline this data directly into biocompute would be a very useful to our user community. Likewise, mechanisms to efficiently import data from online resources would be welcomed.

Biocompute addresses a broad spectrum of challenges in scientific computing web portal development. It illustrates some of the technical solutions to social challenges presented by collaborative and contentended resources, implements techniques for mitigating the performance impact of large semi-persistent datasets on distributed computations, and provides a useful framework for exploring and addressing open problems in cluster storage, computation, and socially sensitive resource management.

7. ACKNOWLEDGEMENTS

We would like to acknowledge the hard work of undergraduates Joey Rich and Ryan Jansen, who greatly assisted in the implementation of Biocompute. We would also like to thank Andrew Thrasher and Li Yu for their development of the SSAHA and SHRIMP modules, respectively. This work is supported by the University of Notre Dame's strategic

investment in Global Health, Genomics and Bioinformatics and by NSF grant CNS0643229.

8. REFERENCES

[1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[2] R.S. Barga and L.A. Digiampietri. Automatic capture and efficient storage of e-Science experiment provenance. *Concurrency and Computation*, 20(5):419, 2008.

[3] B. Clifford, I. Foster, J.S. Voeckler, M. Wilde, and Y. Zhao. Tracking provenance in a virtual data grid. *CONCURRENCY AND COMPUTATION*, 20(5):565, 2008.

[4] A.E. Darling, L. Carey, and W. Feng. The design, implementation, and evaluation of mpiBLAST. *Proceedings of ClusterWorld*, 2003, 2003.

[5] S.B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD Conference*, pages 1345–1350. Citeseer, 2008.

[6] M. Dumontier and C.W.V. Hogue. NBLAST: a cluster variant of BLAST for NxN comparisons. *BMC bioinformatics*, 3(1):13, 2002.

[7] M. Johnson, I. Zaretskaya, Y. Raytselis, Y. Merezhuk, S. McGinnis, and T.L. Madden. NCBI BLAST: a better web interface. *Nucleic acids research*, 2008.

[8] D. Lawson, P. Arensburger, P. Atkinson, N.J. Besansky, R.V. Bruggner, R. Butler, K.S. Campbell, G.K. Christophides, S. Christley, E. Dialynas, et al. VectorBase: a home for invertebrate vectors of human pathogens. *Nucleic Acids Research*, 2006.

[9] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[10] L. Moreau, B. Lud ”ascher, I. Altintas, R.S. Barga, S. Bowers, G. Chin, S. Cohen, S. Cohen-Boulakia, B. Clifford, S. Davidson, et al. The first provenance challenge. *Concurrency and Computation: Practice and Experience*, 20(5):400–418, 2007.

[11] Z. Ning, A.J. Cox, and J.C. Mullikin. SSAHA: A fast search method for large DNA databases. *Genome Research*, 11(10):1725, 2001.

[12] S.M. Rumble, P. Lacroute, A.V. Dalca, M. Fiume, A. Sidow, and M. Brudno. SHRiMP: accurate mapping of short color-space reads. *PLoS computational biology*, 5(5), 2009.

[13] TF Smith and MS Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[14] D. Thain, C. Moretti, and J. Hemmes. Chirp: A practical global filesystem for cluster and grid computing. *Journal of Grid Computing*, 7(1):51–72, 2009.

[15] L. Yu, C. Moretti, A. Thrasher, S. Emrich, K. Judd, and D. Thain. Harnessing Parallelism in Multicore Clusters with the All-Pairs, Wavefront, and Makeflow Abstractions. *Journal of Cluster Computing*, 2010.