Taming Complex Bioinformatics Workflows with Weaver, Makeflow, and Starch

Andrew Thrasher, Rory Carmichael, Peter Bui, Li Yu, Douglas Thain, and Scott Emrich
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN

***Abstract*—In this paper we discuss challenges of common bioinformatics applications when deployed outside their initial development environments. We propose a three-tiered approach to mitigate some of these issues by leveraging an encapsulation tool, a high-level workflow language, and a portable intermediary. As a case study, we apply this approach to refactor a custom EST analysis pipeline. The Starch tool encapsulates program dependencies to simplify task specification and deployment. The Weaver language provides abstractions for distributed computing and naturally encourages code modularity. The Makeflow workflow engine provides a batch system agnostic engine to execute compiled Weaver code. To illustrate the benefits of our framework, we compare implementations, show their performance, and discuss benefits derived from our new workflow approach relative to traditional bioinformatics development.**

I. Introduction

The rapid production of Bioinformatics applications by academic institutions has led to the development of many powerful and useful tools. Many of these tools require significant computational resources for any nontrivial task. Further, increasing numbers of research groups are deploying externally developed bioinformatics applications in their own computing environments and modifying or building upon them.

Unfortunately, deployment and modification of bioinformatics applications has proven challenging. Commonly encountered challenges (Section II) include dependence on specific distributed computing resources, complex program dependencies, and intermingling of conceptually distinct tasks within code. The first two problems severely impede deployment efforts, while the third undermines customization, debugging and code maintenance.

Here, we propose to separate these problems and address each with a different layer of a software stack. Using a production pipeline described in Section III, we implement the three tiers described in Section IV: an encapsulation layer to reduce complex webs of coordinated programs and libraries, a high level workflow language to concisely and intuitively describe this pipeline, and a portable low level workflow language and execution environment.

In Section V, we describe three tools developed by the University of Notre Dame's Cooperative Computing Lab which mitigate the previously mentioned problems. Starch provides a method for packaging complex program dependencies into a single executable archive. Weaver is a Python-based high level workflow description language with support for concise implementation of many common distributed computing patterns. Weaver programs are compiled into make-like low level workflow descriptions that can be executed through the highly portable Makeflow workflow engine. In Section VI and VII we leverage these tools to refactor an internally developed EST analysis pipeline into a maintainable and easily deployed workflow, and compare this refactored pipeline to its predecessor with respect to its conciseness, maintainability, robustness, and portability.

II. Common Challenges in Bioinformatics Applications

A. Portability

When leveraging the parallelism in their software, many development sites focus on a particular distributed resource because it is the only resource accessible to them, and they don't possess the resources or expertise to develop interfaces for more systems. As a result, few applications are developed with the flexibility to utilize a variety of distributed resources. This introduces serious challenges when organizations attempt to deploy computationally intensive tools without access to the same computing resources as the original developers.

B. Software Maintainability

Bioinformatics users often need to handle hypotheses and data outside of the application's original scope. Such difficulties are particularly pronounced in code bases that lack modularity, implement their control logic at very low levels (rather than describing it through higher level work patterns or abstractions), or perform their function indirectly through the runtime generation of code. Some programs achieve a modest degree of modularity, but are implemented with low level control and programmatically-generated intermediate executables [1], [2].

C. Dependency Management

Many bioinformatics applications feature tasks with a high degree natural parallelism. Naturally, bioinformaticians (people developing tools for biologists) take advantage of this by running work on many nodes, often in grid or cloud settings. The execution of such workloads depends on the ability to transport the required dependencies for each task to each worker node. However, many bioinformatics tools rely on third-party applications and libraries to function. This leads to

difficulty in adapting them to run in distributed environments, especially heterogeneous systems such as Condor. It is typically impractical to guarantee the existence of required software on all computation nodes because of diverse execution environments. As a result, inconsistencies in available libraries and applications among worker nodes often lead to failures when attempting to run distributed bioinformatics applications.

III. EST PIPELINE DESCRIPTION

Expressed Sequence Tags (ESTs) are an important source of bioinformatics data. EST sequencing involves extracting and sequencing representatives of the mRNA of a cell, which represent the expressed portion of an organism's genes. Since ESTs capture the expressed genes of an organism, biologists are able to glean significant amounts of information from diverse species, including organisms where whole-genome sequencing resources are not yet available. This is a particularly useful method for research on non-model organisms, including those important in studying ecological and environmental questions [3].

Due to the effectiveness of these data for ecological questions, we created specialized tools for analyzing EST data from natural populations of butterflies [4]. These tools were created to be run as a pipeline of custom Ruby scripts written by a member of the Notre Dame Bioinformatics Lab; however, each step in the analysis reported in [4] (including several large BLAST jobs) was run individually and manually. These scripts have many dependencies in the form of Ruby libraries, both custom and public. These dependencies made the subtasks of this workflow difficult to run on the various resources available, or even to share among researchers. These challenges were exacerbated when analysts attempted to run independent components in parallel.

Our initial efforts began with a step familiar to many bioinformatics developers—the creation of a wrapper Perl script. Because of our previous experience, we chose to use a Perl wrapper script to generate a Makeflow [5], and through it execute tasks. This process is similar to the mechanism by which InterproScan [2] generates makefile descriptions of its workflows. However, this particular workflow was based on Ruby and specific Ruby libraries such as BioRuby, which were not available on many of the machines available in our distributed system (Condor). To overcome this, cumbersome rules were written to specify library dependencies, and the workflow subtasks could only be executed on machines with access to the Ruby interpreter. Furthermore, the wrapper script was hard to read and modify because the functionality it was providing was buried in the print statements it used to generate the workflow.

In our experience many pipelined bioinformatics tools are released as such: functional but inelegant. They work on a single system, rely heavily on custom wrapper scripts, and have complex dependencies and specifications.

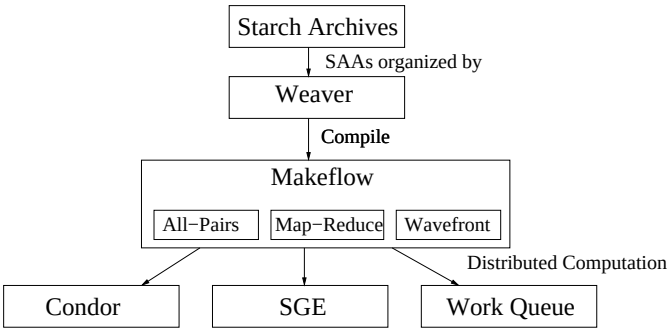

Fig. 1. Stack of applications.

IV. SOLUTION STRATEGY

Based on these limitations we explored an alternative framework, using a layered approach (Figure 1) to address each of our problems separately. The clearest challenge we were experiencing was that of dependency management, but we had already suffered from coding errors because of the nature of our wrapper script. Further, we hoped to expand to more computational resources.

A. Portable low level workflow engine

We required the ability to execute our workflows on a variety of systems. To achieve this we needed an engine that could support execution of the same workflow in multiple batch execution environments. Such an engine would ideally provide us with execution robustness and detailed logs of runtime behaviors to increase reliability and ease debugging.

B. High level workflow specification language

While the low level language of Makeflow provided the desired portability and logging, we knew from our initial pipeline that the Perl wrapper script necessary to generate makeflows was difficult to understand and modify. A high level workflow language that would express the structure of the workflow programmatically, modularly describe units of work, and provide abstractions for common patterns of work would greatly increase the transparency and maintainability of the code.

C. Encapsulation

We quickly recognized that our dependency problems could be resolved with some sort of encapsulation strategy. To be effective, such a strategy needed to reduce the entire web of dependencies to a single package. Additionally, the execution of that package needed to be as simple as possible. Such a change would also conveniently simplify the specification of tasks in a workflow specification language.

V. SOLUTION TOOLS

A. Makeflow

Makeflow [6] is a workflow engine targeted at execution on clusters, grids and clouds. It accepts a specification of a workflow and parallelizes the execution on multiple cores

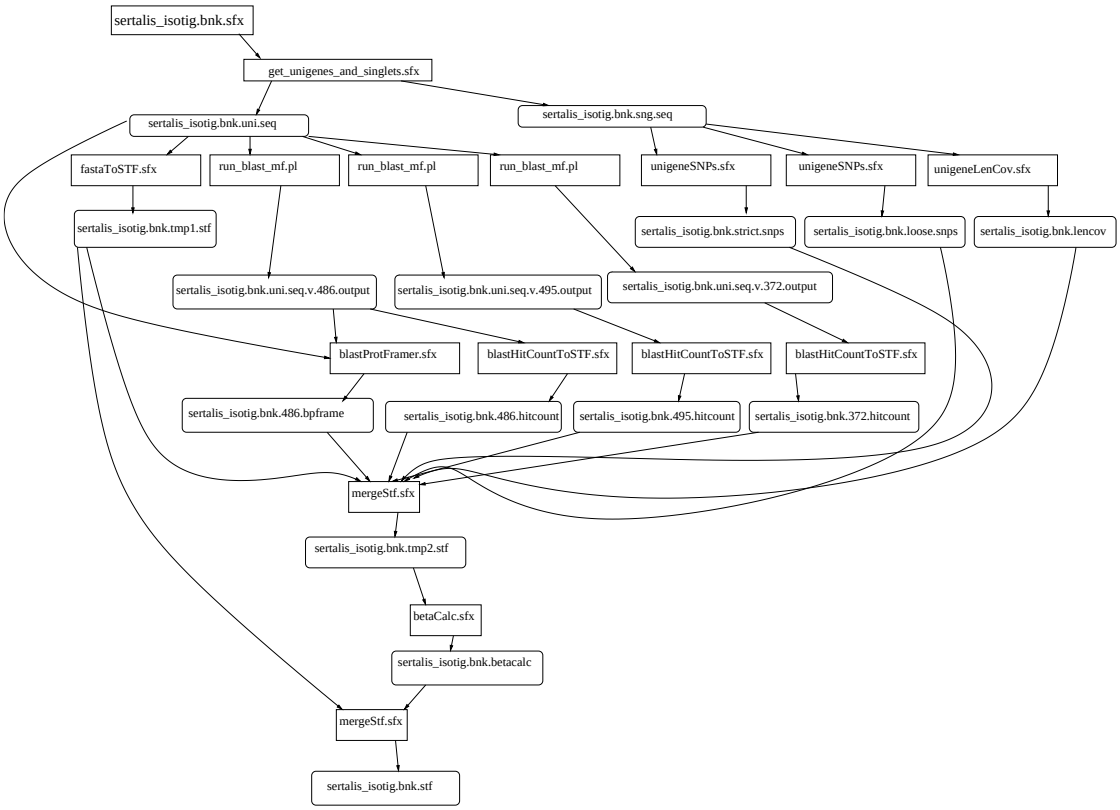

Fig. 2. The EST pipeline takes an AMOS bank from an assembly and three BLAST databases.

or machines whenever possible. The syntax of a Makeflow workflow specification is similar to the traditional UNIX Make program. It consists of a list of rules where each rule contains a set of source files, a set of target files and a command to generate those target files from the source files. The dependencies among the rules are implicitly expressed in the source and target files. For example, if a source file of rule A is also a target file of rule B, then Makeflow would know that rule A depends on rule B. Thus, all the dependencies in the workflow can be visualized in a directed acyclic graph (DAG). Because Makeflow knows the DAG structure of the workflow, it can readily determine which rules have had their dependencies satisfied thus far, and execute such rules in parallel. An example makeflow DAG describing the EST analysis pipeline is shown in Figure 2.

The Makeflow specification can work on different computer systems, such as multicore machines, Condor and Sun Grid Engine (SGE) batch systems, and the bundled Work Queue system. For example, a user can combine some number of Condor nodes with some number of SGE nodes to work on the same workflow using Work Queue. As a workflow engine that drives various parallel systems, Makeflow provides fault-tolerance across all the underlying systems that it supports. If a workflow fails or stops during the execution, Makeflow will continue from where it left off upon resuming.

B. Weaver

Although makeflows (and other DAG-based workflow specifications such as DAGman) are relatively straightforward to construct for small applications, they can be cumbersome and difficult to program and maintain for the large pipelines commonly found in bioinformatics. Rather than generating Makeflows manually or using ad-hoc scripts, we utilize the Weaver distributed computing framework that allows us to develop workflows in the Python programming language [7].

While many workflow-type languages enforce a graph-based or agent-based programming paradigm, Weaver allows users to program in a variety of common styles such as imperative, functional, and object-oriented while exposing a simple programming model consisting of datasets, functions, and abstractions. In Weaver, datasets are simply collections of input files. These can be a Python list of file names, a generator function, or even a SQL query. Functions in the Weaver programming model are formal specifications of the interface of executables. Weaver functions may be external programs or they may be inlined Python code. To develop with Weaver, the user simply specifies their datasets and functions and then organize them into pipelines using Weaver's features.

By default, Weaver also includes the following abstractions: Map, All-Pairs [6], Wavefront [6], and Map-Reduce [8]. Users may combine any of these abstractions in their workflows to construct sophisticated pipelines.

To run a workflow written in Weaver, the programmer must

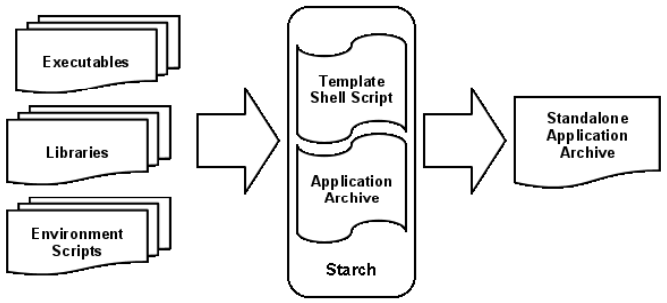

Fig. 3. Users provide a set of executables, libraries, and environment settings to Starch, which internally produces an application archive. This is then appended to a template shell script to produce a standalone application archive that can be executed as a normal application.

first compile it using the Weaver compiler. The user constructs the workflow as a Python script and passes it to Weaver, which then compiles the code into a workflow sandbox. This sandbox encapsulates the Makeflow DAG file and any input data and executables specified in the Weaver script. To run the workflow, the user simply executes Makeflow from inside the sandbox.

Weaver provides a high-level language in which to specify our workflow. It possesses the necessary abstractions and modularity support to create transparent and maintainable code.

C. Starch

A common problem in large distributed applications is the packaging and management of individual application components. While Weaver and the underlying Makeflow system support specifying dependent files and environmental variables, it is often necessary to test these individual application components independently. To solve this problem, Weaver includes a tool named Starch to create standalone application archives (SAA).

To create a SAA, the user simply specifies a list of executables and libraries to include in the execution image along with the command run when the SAA is executed by the user. To aid in packaging, Starch will automatically search for any dynamically linked libraries required by the executables specified and include those in the list of libraries to embed in the SAA. If the application requires any special input data files, they may also be included. Likewise, for special environmental variables and other runtime configuration options, Starch allows users to include environment scripts that will be imported before the application's command is executed.

The general process for creating a SAA is shown in Figure 3. Once the user specifies all of the necessary options, the executables, libraries, and environment scripts are compressed and archived as a UNIX tarball. This application archive is then appended to a template shell script to generate a standalone application archive. When the SAA is executed the wrapper shell script will automatically extract the embedded archive, configure the environment, and run the user specified command.

The benefit of using Starch to package applications is threefold: First, it allows for complex applications with multiple dependencies to be bundled as a single self-contained executable. This is important for bioinformatics workflows where a single application may require a wrapper script and multiple external programs and libraries to execute properly. Second, because Starch produces a standalone application archive, the individual application component is naturally versioned, and easier to test and share among researchers. Finally, the self-contained nature of the SAAs also facilitates deployment in distributed systems where it is not known if the target working environment contains all the necessary libraries and programs. This clearly satisfies our encapsulation needs.

VI. Final EST pipeline

We used these tools to implement the final version of our EST pipeline. Starch enabled us to encapsulate large complex steps into more convenient archives that are easy to distribute. We used Weaver rather than Perl to express our workflow. This compiled into a Makeflow program capable of running on a variety of batch systems.

Figure 4 is an excerpt of Weaver code for running the EST pipeline. It is compiled into Makeflow code describing the DAG shown in Figure 2. From the figure, we notice immediately that there are several steps that can be run in parallel. Each of the run_blast_mf.pl steps generates several thousand intermediate steps as they run BLAST subjobs in parallel.

VII. Pipeline Characteristics

The final pipeline demonstrates the promising technical characteristics of workflows implemented using this stack.

A. Provenance

While they do not support provenance queries, Starch, Weaver, and Makeflow provide a powerful tool for gathering provenance information.

First, Starch naturally provides versioning and encapsulation. Once a Starch archive is created, it remains the same regardless of changes to the source executables, providing a working frozen copy of a program. This facilitates debugging and helps users avoid many of the problems associated with upgrades and updates in complex dependency environments.

Each run of a Weaver program produces a different makeflow, describing the specific steps about to be executed and the full set of dependencies for each step. Further, upon execution, a makeflow can generate a great deal of provenance information, including the node of execution, start and end time, and exit status of each substep of the makeflow. Existing tools for analyzing and displaying runtime have been adapted to Makeflow logs (Figure 5).

B. Encapsulation

Starch takes only two minutes to create the packages required for running this EST pipeline. One such archive contains 14519 files, clearly motivating the need for encapsulation.

```
def run_blast_mf(r):
     run = Function('run_blast_mf.pl')
     run.output_string = lambda i: i
     run.command_string = lambda i, o: \
    'LOCAL ./run_blast_mf.pl ' + r
     run.add_functions('blastwrapper.pl')
     run.add_functions('tee')
     run.add_functions('blastall')
     return run

def unigeneSNPs(r):
     run = Function('unigeneSNPs.sfx')
     run.output_string = lambda i: i
     run.command_string = lambda i, o: \
     './unigeneSNPs.sfx > ' + o
     run.add_functions(bank+'.sfx')
     return run

def unigeneLenCov(r):
     run = Function('unigeneLenCov.sfx')
     run.output_string = lambda i: i
     run.command_string = lambda i, o: \
     './unigeneLenCov.sfx > ' + o
     run.add_functions(bank+'.sfx')
     return run
```

Fig. 4. Excerpt from the Weaver EST pipeline code.

C. Performance

Weaver also generates makeflows, so we did not expect performance gains relative to our original Perl. As expected, a Weaver makeflow executes in approximately the same time as a Perl-generated makeflow (Table I).

The EST pipeline, however, provided significant advantages over its manual incarnation. Even with the assistance of campus computing resources for BLAST the pipeline suffered from turnaround times of approximately one work week. With the creation of an automated pipeline aided by Starch, turnaround time for previous data has been reduced to under an hour.

Weaver enables programmers to take advantage of native abstractions. For BLAST (Table I), the Weaver map variation is slightly faster than a highly optimized non-Weaver implementation [5]. Other programs may see more pronounced performance gains from abstractions [7].

D. Portability

We have run our pipeline using a variety of batch systems, and even by combining multiple systems (SGE and Condor). This flexibility is derived from the technical characteristics of Makeflow (see Section V).

E. Fault Tolerance

In a distributed system, there are numerous ways a task can fail. Makeflow provides batch system-independent fault tolerance. In the worst case, Makeflow can emit both regular and debugging output. This assists the programmer, or user, in investigating the causes of failure. It also contributes to the body of provenance information generated by the system.

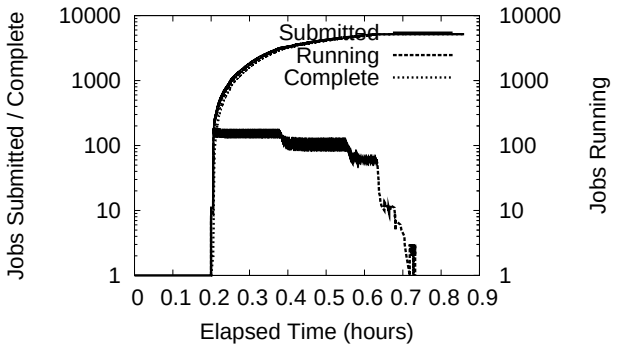

Fig. 5. EST pipeline runtime graph that can be generated from the log output of Makeflow.

TABLE I
RESULTS FROM RUNS OF TWO DIFFERENT APPLICATIONS USING WEAVER AND USING PERL.

Application	Weaver	Non-Weaver
EST pipeline	2,529s	2,882s
BLAST	665s	677s

VIII. DISCUSSION

We see a number of improvements related to the development, use, and maintenance of our pipeline.

We have found our Weaver scripts to be both more concise and understandable. For example, our Weaver implementation of BLAST is approximately one-third shorter than our Perl implementation. More importantly, Weaver more closely resembles general purpose programming languages, which in our experience has helped development and maintenance of bioinformatics tools. This advantage is increased by the availability of built-in abstractions, which in our experience has reduced code volume and increased readability as MapReduce [8] has for other applications.

The object oriented nature of Weaver helps bioinformatics programmers to write modular code, as individual makeflow steps are coded as functions that run on objects. Unlike many typical bioinformatics applications, bioinformaticians are encouraged to create sets of objects and functions that can be reused. This also enables many different types of analysis to be performed or updated from these core sets of functions and objects.

In addition to making our workflow-description code more concise, the use of Starch reduces the overall size by a third (and as a result reduces complexity) while still producing the same correct output. By packaging programs that depend on many libraries or subprograms into a Starch archive, we can simply specify the archive and then execute it on a remote machine.

This increased level of software engineering should help to alleviate some of the issues currently associated with running bioinformatics applications. Weaver applications tend to be more readable than corresponding Perl scripts. This allows for easier modification and maintenance of existing software.

Weaver does not necessarily provide any direct performance benefit. These makeflows, however, can by executed in multiple distributed environments at the same time and built-in abstractions are also available. Traditionally, programmers have had to implement abstractions to take advantage of the parallelism available or use specific abstractions such as MapReduce [8]. With Weaver, the programmer can simply call multiple abstractions as needed to achieve parallelism with relative ease.

IX. Conclusion

The rapid production nature of bioinformatics applications gives rise to useful and powerful tools. However many of them would benefit from improved encapsulation, clarity of code, and portability. We believe that Starch, Weaver, and Makeflow begin the process of addressing some of these problems. Starch encapsulates program dependencies to simplify task specification and deployment. Weaver provides a natural way to express workflows modularly, and at a high level. Makeflow provides a batch-system agnostic engine to execute compiled Weaver workflows.

X. Related Work

There is certainly no shortage of distributed systems and workflow tools available for bioinformatics. The single most popular tool in distributed systems recently is the MapReduce [8] paradigm. However, MapReduce is only a single type of abstraction for distributed computing; it is not sufficient for all tasks. Therefore Weaver implements a MapReduce function, but many other abstractions as well. Unlike the Hadoop MapReduce, the Weaver implementation of MapReduce does not take advantage of data locality.

The Swarm [9], Taverna [10], Pegasus [11] and Galaxy [12] services take advantage of several computing resources, similar to Makeflow's capabilities. However, they are designed to manage many users' jobs and require additional infrastructure to host. Makeflow is designed to be run on a per-user basis and requires no special infrastructure.

The Kepler [13] workflow system provides a method to generate scientific workflows. However it is not designed to assist the developer in creating parallel workflows from third party executables. Weaver and Makeflow encourage the developer to parallelize third party executables with minimal effort.

The SAGA [14] system is similar to our Makeflow layer in that it provides the abstraction to the execution layer. However, Makeflow is workflow description language and execution engine. SAGA is only an API for interacting with various distributed computing systems. Makeflow has this as a component, but provides much more to the user.

Makeflow collects much provenance data. It would be extremely useful to develop a system to query this data as suggested in the First Provenance Challenge [15]. Makeflow is already able to emit the DAG that it constructs so the user can see the steps that lead to each output.

Acknowledgments

This work is supported by the University of Notre Dames strategic investment in Global Health, Genomics and Bioinformatics and and the National Institutes of Health NIAID contract HHSN272200900039C. The authors would also like to thank Shawn O'Neil for his original EST pipeline scripts that formed the foundation for this study.

References

[1] E. W. Myers, G. G. Sutton, A. L. Delcher, I. M. Dew, D. P. Fasulo, M. J. Flanigan, S. A. Kravitz, C. M. Mobarry, K. H. Reinert, and K. A. e. a. Remington, "A Whole-Genome Assembly of Drosophila," *Science*, vol. 287, no. 5461, pp. 2196–2204, 2000.

[2] E. M. Zdobnov and R. Apweiler, "InterProScan - an integration platform for the signature-recognition methods in InterPro," *Bioinformatics*, vol. 17, no. 9, pp. 847–848, 2001.

[3] J. Vera, C. WHEAT, H. FESCEMYER, M. FRILANDER, D. CRAWFORD, I. Hanski, and J. MARDEN, "Rapid transcriptome characterization for a nonmodel organism using 454 pyrosequencing," *Molecular Ecology*, vol. 17, no. 7, pp. 1636–1647, 2008.

[4] S. O'Neil, J. Dzurisin, R. Carmichael, N. Lobo, S. Emrich, and J. Hellmann, "Population-level transcriptome sequencing of nonmodel organisms Erynnis propertius and Papilio zelicaon," *BMC genomics*, vol. 11, no. 1, p. 310, 2010.

[5] R. Carmichael, P. Braga-Henebry, D. Thain, and S. Emrich, "Biocompute: Towards a Collaborative Workspace for Data Intensive BioScience," *Emerging Computational Methods for Life Sciences Workshop at High Performance Distributed Computing (HPDC)*, 2010.

[6] 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," *Cluster Computing*, vol. 13, pp. 243–256, 2010, 10.1007/s10586-010-0134-7.

[7] P. Bui, L. Yu, and D. Thain, "Weaver: Integrating Distributed Computing Abstractions into Scientific Workflows using Python," *CLADE 2010*.

[8] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *COMMUNICATIONS OF THE ACM*, vol. 51, no. 1, p. 107, 2008.

[9] S. Pallickara, M. Pierce, Q. Dong, and C. Kong, "Enabling Large Scale Scientific Computations for Expressed Sequence Tag Sequencing over Grid and Cloud Computing Clusters," in *PPAM 2009 Wroclaw, Poland*. Citeseer, 2009.

[10] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li, and T. Oinn, "Taverna: a tool for building and running workflows of services." *Nucleic Acids Research*, vol. 34, no. Web Server issue, pp. 729–732, July 2006.

[11] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. Su, K. Vahi, and M. Livny, "Pegasus: Mapping scientific workflows onto the grid," in *Grid Computing*. Springer, 2004, pp. 131–140.

[12] J. Goecks, A. Nekrutenko, J. Taylor *et al.*, "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences," *Genome biology*, vol. 11, no. 8, p. R86, 2010.

[13] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: An extensible system for design and execution of scientific workflows," in *Scientific and Statistical Database Management. Proceedings. 16th International Conference on*. IEEE, 2004, pp. 423–424.

[14] H. Kaiser, A. Merzky, S. Hirmer, and G. Allen, "The SAGA C++ Reference Implementation," in *Second International Workshop on Library-Centric Software Design (LCSD'06)*. Citeseer, p. 101.

[15] L. Moreau, B. Ludäscher, I. Altintas, R. Barga, S. Bowers, S. Callahan, G. JR, B. Clifford, S. Cohen, S. Cohen-Boulakia *et al.*, "Special issue: The first provenance challenge," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 409–418, 2008.