CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2012; **24**:1685–1707
Published online 9 November 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1871

Scripting distributed scientific workflows using Weaver

Peter Bui*,†, Li Yu, Andrew Thrasher, Rory Carmichael, Irena Lanc, Patrick Donnelly and Douglas Thain

Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, USA

SUMMARY

Weaver is a high-level distributed computing framework that enables researchers to construct scalable scientific data-processing workflows. Instead of developing a new workflow language, we introduce a domain-specific language built on top of Python called Weaver, which takes advantage of users' familiarity with the programming language, minimizes barriers to adoption, and allows for integration with a rich ecosystem of existing software. In this paper, we provide an overview of Weaver's programming model, which allows users to organize and specify scientific workflows by using a collection of datasets, functions, and abstractions. We also explain how these workflow specifications are compiled into a directed acyclic graph that is used by the Makeflow workflow manager to dispatch work to a variety of distributed execution platforms. To demonstrate the power and benefits of using the framework in constructing scientific research applications, the paper examines four distinct real-world applications scripted using Weaver and analyzes the performance, scalability, and impact of the distributed generated scientific workflows.

Received 8 February 2011; Revised 24 June 2011; Accepted 29 August 2011

KEY WORDS: scripting; workflow; distributed systems; weaver; makeflow; python

1. INTRODUCTION

In recent years, the increase in the availability of vast amounts of distributed computing resources has led to the development of new programming tools and systems that simplify and ease the use of such resources. Primarily, these tools have come in the form of distributed computing abstractions, such as MapReduce [1] and All-Pairs [2], that optimize specific patterns or models of computation. These new systems have been useful in enabling the development of high-performance/high-throughput distributed scientific applications.

Unfortunately, although these abstractions have been successful in improving specific patterns of computation, they often fail to encompass large and sophisticated scientific workflows. This is because whereas many computational data-processing workflows consist of a series of separate computational stages, these tools normally focus on a single particular stage. For instance, in the case of biometrics as shown in Figure 1, a basic experimental workflow consists of selecting a subset of data from a repository, transforming the raw data into an intermediate form suitable for processing, and finally performing the experiment. Such multi-stage workflows are often too complicated to be performed in a single abstraction and may in fact require the use of multiple computational abstractions.

*Correspondence to: Peter Bui, Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, USA.
†E-mail: pbui@cse.nd.edu

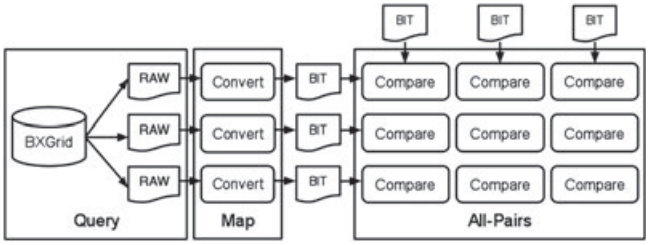

Figure 1. Biometrics workflow.

To address this problem a few workflow systems have been developed such as Dryad [3], DAGMan [4], and Pegasus [5], which allow users to specify a pipeline of computational tasks. These specifications usually consist of relationships between tasks and the data inputs and outputs, and are used by the software tools to construct a directed acyclic graph (DAG) representing the flow of data through the pipeline. Distributed computing abstractions are incorporated into these systems by implementing the abstraction directly as nodes in the DAG or by using a specialized implementation as a single node in the graph. Once a DAG has been formed, it can be processed by a workflow manager, which dispatches tasks to a distributed computing engine such as Condor [6].

The main disadvantage of these DAG-based systems is that they require end users to explicitly construct workflow graphs, which is often cumbersome and complex for larger sophisticated workflows [7–9]. For example, it is not uncommon for scientific workflows to consist of thousands to millions of tasks, where each job must be specified as a DAG node. Recent research projects such as Swift [10] tackle the problem of efficiently specifying scientific workflows by proposing new programming languages. In these high-level languages, the DAG is implicitly constructed by a compiler that parses and processes a workflow specification written in a high-level scripting language. The advantage of this approach is that it allows for rapid construction of sophisticated distributed applications in concise and maintainable workflow scripts.

The introduction of a new workflow scripting language, however, has significant adoption and usage challenges [11]. For instance, it is not always possible to deploy the new system onto unprivileged distributed resources, and it may be difficult to convince non-expert users to adopt the new language due to the unfamiliar syntax or programming model. Rather than developing a new language, we address the problem of efficiently constructing distributed workflows by building a distributed computing framework named **Weaver** [9] on top of an existing general-purpose programming language, Python [12]. By providing a framework that facilitates the development of distributed applications as a domain-specific language in Python, we enable researchers to effectively and efficiently script scientific distributed workflows using a familiar and ubiquitous programming language with a rich ecosystem of existing software, documentation, and community.

Using the Weaver framework, research scientists develop distributed applications by specifying the properties of their workflow in a Python script. The framework provides a set of high-level constructs such as **datasets**, **functions**, and **abstractions** that implement the domain-specific language and allows the users to rapidly construct their applications. Once these workflow scripts are complete, they are processed by the Weaver compiler to produce a DAG that is then utilized by a workflow manager such as Makeflow [13]. This workflow manager, in turn, executes the application on a variety of distribute execution engines such as Condor or Sun Grid Engine (SGE) by dispatching jobs on the basis of the tasks specified in the generated DAG.

This paper is an extended version of our workshop paper introducing Weaver [9]. Since this initial work, we have developed additional **datasets** and **functions** to support the new applications presented in this paper. Moreover, we have extended the programming model with the concept of **nests** that enables the user to construct hierarchical workflows for large sophisticated data-processing pipelines. Additionally, we also developed the Starch application packager, which was introduced at a recent biometrics workflow workshop [14], as a complementary part of the Weaver framework.

DOI: 10.1002/cpe

This new tool enables creating portable standalone application archives (SAAs) and therefore provides users a simple means of managing application dependencies separate from the entire workflow. Finally, Weaver has been utilized in a variety of different distributed applications ranging from biometrics experiments and multimedia transcoding to bioinformatics analysis pipelines, allowing us to evaluate Weaver's effectiveness in enabling both distributed computing experts and non-specialists to effectively script distributed scientific workflows.

The remainder of this paper explores the details of Weaver. In Section 2, we explain the programming model used in Weaver and identify what components are provided by the framework and how they fit together. Section 3 discusses the execution model used by Weaver, describes the complete software application stack, and examines how the different pieces integrated into the framework. Section 4 provides an evaluation of four applications developed using Weaver along with analysis of the effectiveness of the framework in developing these workflows. This is followed by a short discussion of work related to Weaver. Finally, we conclude the paper with an exploration of possible future work and our summary of the current version of Weaver.

2. PROGRAMMING MODEL

Weaver is a distributed computing framework that enables researchers to construct scalable scientific workflows in the Python programming language. This framework utilizes a simplified programming model that consists of **datasets**, **functions**, **abstractions**, and **nests** as shown in Table I. These concepts are the fundamental building blocks of the Weaver application programming interface and are implemented as a collection of custom Python modules, classes, and functions that end users combine and extend to define their workflows. This section provides a high-level overview of these components and explains how they work together to enable the construction of distributed workflows.

2.1. Datasets

Many scientific computing tasks involve processing a repository or collection of experimental data, which is normally stored as files on a physical filesystem. In the Weaver programming model, collections of data objects are organized into **datasets**, where each object's string method returns the location of the file that contains the data. This simple convention allows for datasets to take the form of a Python list, set, generator function, or any other Python object that implements the language's iteration protocol.

Although specifying a **dataset** can be as straightforward as defining a list of file paths, Weaver provides a collection of custom *DataSet* objects that simplify the specification and selection of input data. Each object in these *DataSet* collections contains the path to the data file as required by the programming model, along with a set of attributes shared by all the members of the dataset. This common set of metadata properties can be accessed and manipulated by the user through the *Query* function, which will be explained shortly.

2.1.1. File DataSets. One example of a *DataSet* provided by the Weaver framework is the *FilesDataSet*. Since the most common type of dataset is simply a group of data files, Weaver provides the *FilesDataSet* constructor, which given a file path pattern, this dataset builder will return the set of file objects that match the specified pattern. Another common method for keeping track of files

Table I. Weaver programming model.

Component	Summary
Datasets	Collections of data objects that represent physical files.
Functions	Specifications of executables used to process data.
Abstractions	Patterns of execution that define how functions are applied to datasets.
Nests	Combination of workspace and directed acyclic graph that defines the workflow namespace.

DOI: 10.1002/cpe

is to store the list of data paths in a text file. Weaver provides a simple *FileListDataSet* object for this type of collection. Each object in both of these collections contains the location of the file, along with relevant filesystem metadata of each file such as size and timestamps. An example of the *FilesDataSet* is shown below in Listing 1.

```
# Define dataset using Files constructor
files_ds = FilesDataSet('/path/to/files/*.txt')

# Filter files dataset for sizes > 1024
my_fds = Query(files_ds, files_ds.c.size > 1024)

# Define dataset using SQL constructor
sql_ds = SQLDataSet('db', 'biometrics', 'irises')

# Filter SQL records based on eye color
my_sds = Query(sql_ds,
               Or(sql_ds.c.EyeColor == 'Blue', sql_ds.c.EyeColor == 'Green'))
```

Listing 1. Weaver datasets examples.

2.1.2. Structured Query Language DataSets. In addition to data files stored on a filesystem, another common source for scientific data is a Structured Query Language (SQL) database. Weaver provides a simplified database querying interface that facilitates accessing information stored in conventional SQL databases such as MySQL or SQLite. Besides specifying the details about how to connect to the database as shown in Listing 1, the user only needs to define a `file_path` method, which returns the location of the data file on the basis of the object record returned by an SQL query. The user may either directly map the database record to a file on disk, or the user may materialize a file containing information from the database record and return the path to that generated file. In either case, it is up to the user to specify how to translate the database record to a physical data file as demanded by the Weaver programming model.

2.1.3. DataSet Queries. Sometimes it is necessary to filter or select a subset of data from a large collection before processing it. For instance, a scientific database may contain thousands of records, but the user is only interested in a specific subset for experimentation. To facilitate this selection operation, Weaver provides a *Query* function that allows the user to filter a dataset in a manner similar to the SQLAlchemy expression language [15], a popular Python object-relational mapping (ORM) system. Users can specify queries on arbitrary datasets using SQL-like operations on datasets regardless of whether the underlying data is an actual database or collection of Python objects as shown in Listing 1.

Because objects in Weaver *DataSet* collections contain metadata information common to each item in the set, it is possible for the user to filter these sets of objects based on their attributes. The Weaver *Query* function provides this selection operation by implementing an SQL-like query expression language, which translates the user-defined queries into an appropriate form for the underlying data collection. For datasets that are actual databases, the function will translate the ORM query expression into the appropriate SQL expression and use the generated SQL to perform the query on the database server. In the case of datasets that are collections of Python objects, the ORM query expressions are translated into filter functions that are applied to each object in the dataset to produce the desired subset of the data collection. To use the *Query* function, the user simply specifies the name of the dataset, followed by an ORM query expression. As can be see in Listing 1, this allows users to filter and select all their datasets in a simple and consistent manner.

In summary, users may specify **datasets** by using normal Python collections such as lists, tuples, or sets, or they may use one of the provided Weaver *DataSet* constructors such as *FilesDataSet*, *FileListDataSet*, and *SQLDataSet*. Utilizing one of the Weaver *DataSet* constructors further enables selection and filtering using a simple ORM system provided through the *Query* function.

DOI: 10.1002/cpe

2.2. Functions

The second major component of most scientific workflows are the executables used to process the data. The Weaver programming model accounts for these executables by providing the notion of a **function** specification wrapper, which is a Python object that defines the interface to an external application or embedded script. Like objects in a dataset, each Weaver function also corresponds to a physical file, in this case an executable or script, on the filesystem.

As with datasets, Weaver provides a set of custom Python components designed to expedite and simplify the specification of workflow functions. The base object is a generic *Function* object that contains the path to the executable as well as a couple of methods: `command` and `output`. The first method specifies how to generate the appropriate shell command string needed to execute the task given a set of input and output files, whereas the second method allows the user to provide a default means of naming the output files of the function.

Users may further modify the specification of a *Function* by passing a `cmd_format` argument or a `cmd_args` parameter. The former allows the user to specify the order in which the executable, arguments, inputs, and outputs are to be placed in the `command` string. The latter allows the user to specify additional non-file arguments as shown in Listing 2. In the example, the `grep` *Function* is formed using the base *Function* constructor, and the `cmd_args` parameter is used to specify the *needle* to search for during the operation. The reason for this separate `cmd_args` parameter is that the inputs and outputs of Weaver *Functions* must refer to physical file locations.

```
def sum_all_floats(*args):
    for f in args:
        print sum(map(float, open(f).readlines()))

grep       = Function('grep', cmd_args = 'needle')
cat        = StreamFunction('cat', out_suffix = 'txt')
img_to_png = SimpleFunction('convert', out_suffix = 'png')
float_sum  = PythonFunction(sum_all_floats)
script     = ScriptFunction('#!/bin/sh\nchmod 664 && mv -f $1 $2\n')
```

Listing 2. Weaver functions examples.

To further simplify the definition of functions, Weaver includes a set of constructors that build upon base *Function* constructor.

2.2.1. StreamFunction. A common type of executable is one that takes one or more input files and outputs the results to standard output (`stdout`). An example of this is the `cat` utility, which takes a set of input files and concatenates their contents to the output stream. Such an application can be specified in Weaver by using the provided *StreamFunction* constructor. Because these executables require capturing standard output in order to generate a output file, the *StreamFunction* object modifies the `command` method to reflect this requirement.

2.2.2. SimpleFunction. Some executables simply require the user to explicitly specify the input and output files as arguments in the shell command. Weaver provides the *SimpleFunction* constructor for these types of applications. The *SimpleFunction* is similar to the *StreamFunction* except that the shell command generated by the `command` method does not capture standard output to a file because the executable requires an explicit output file to be specified.

As shown in Listing 2, both the *StreamFunction* and *SimpleFunction* constructors allow the user to specify a default output suffix by passing the `out_suffix` keyword argument to the function constructor. This suffix is used by the *output* method of each constructor to generate an output file name on the basis of the executable and the input filename. This generated output file name is then used by the `command` method to generate an appropriate shell command.

DOI: 10.1002/cpe

2.2.3. PythonFunction. A third *Function* object provided by Weaver is the *PythonFunction* constructor, which allows for Python functions embedded in the specification script to be invoked as normal executables. As shown in Listing 2, this is done by simply passing the desired Python function to the *PythonFunction* constructor. Since all functions in the Weaver programming model must be manifested as physical files on the filesystem, the constructor marshals the specified Python function and embeds it into a template Python script that is materialized on the filesystem. Any arguments passed to this generated script, such as the input and output files, are passed to the marshaled function as normal Python function arguments. The availability of this constructor allows users to specify complete workflows entirely in the Python programming language, which is useful for prototyping or experimentation.

2.2.4. ScriptFunction. Along the same lines, sometimes it is useful specify a short sequence of shell commands rather than a single executable command. For these situations, Weaver provides the *ScriptFunction* constructor. Like *PythonFunction*, *ScriptFunction* allows workflow designers to embed shell scripts in their workflow specification. These scripts are materialized on the filesystem upon compilation and work as normal executables. Listing 2 provides an example of a simple *ScriptFunction*. In this short shell sequence, the function changes the permissions of the input file and moves the input file to an output target location. This is a useful script because it atomically modifies the properties of a file and then renames it.

All together, these Weaver *Function* constructors, *StreamFunction*, *SimpleFunction*, *PythonFunction*, and *ScriptFunction*, simplify the task of specifying and defining **functions** that are to be used in a distributed scientific workflow.

2.3. Abstractions

The third component in the Weaver programming model is the notion of **abstractions**, which are patterns or models of computation with a precise set of semantics. Unlike most other high-level workflow frameworks, Weaver provides a built-in collection of distributed computing abstractions to the end user as higher-order functions that the user explicitly invokes in order to utilize the pattern in a workflow [13].

Each abstraction takes in a set of datasets and functions as arguments and applies those functions to the input data in a particular pattern. For the development of pipelined workflows, the output of each abstraction is another collection of data objects, thus enabling the output of one abstraction to be used as the input to another. If a collection of output files is not desired, the user may choose to merge these output files into a single file by specifying an `output` file when invoking the abstraction.

As with datasets and functions, Weaver includes a library of readily available *Abstractions*. A summary of these abstractions can be found in Table II. The following subsections describe the five patterns of computation commonly found in scientific workflows that are included as a part of the Weaver framework.

2.3.1. Run. The most basic pattern is simply executing a single task on the basis of a set of inputs and outputs. In Weaver, this is the *Run* abstraction, which takes a single function and applies the

Table II. Weaver abstractions.

Abstraction	Description
Run (`function, inputs`)	Apply `inputs` to `function`.
Map (`function, inputs`)	For each `input` in `inputs`, apply `function`.
MapReduce (`mapper, reducer, inputs`)	For each `input` in `inputs` apply `mapper`, sort intermediate outputs, and then apply `reducer`.
AllPairs (`function, inputs_a, inputs_b`)	Apply `function` to all combinations of `inputs_a` and `inputs_b`.
Wavefront (`function, matrix`)	Compute recurrence relation by applying `function` to `matrix` in a wave pattern.

DOI: 10.1002/cpe

inputs to generate the outputs. Although very simple, this abstraction is useful for constructing more complicated patterns of execution.

2.3.2. Map. The *Map* abstraction is a common pattern used for work that exhibits data parallelism. *Map* in an input function, which is applied to each item in the input dataset. The results of each function application is stored in a collection of output data objects or as a single data file if the user specifies an output target. Because each function application is independent of other function executions, the individual tasks in this pattern are data parallel and thus can be executed concurrently.

2.3.3. MapReduce. Another common data-processing abstraction provided by Weaver is *MapReduce* [1]. In this pattern, a `mapper` function is applied to the initial set of inputs to generate a group of intermediate output files that are partitioned, sorted, and then passed to the `reducer` function for aggregation. All the tasks in both the `mapper` and `reducer` phases exhibit data independence and therefore can be run in parallel.

2.3.4. All-Pairs. An abstraction that is frequently used in fields such as biometrics and data-mining is *All-Pairs* [2]. In this pattern of work, each member of one dataset is compared with each member of another dataset to produce a matrix that contains the scores for each comparison. Like the previous abstractions, the individual comparison tasks can execute independently of each other, which allows the jobs to be scheduled to run concurrently.

2.3.5. Wavefront. An abstraction used in game theory and gene sequencing applications is *Wavefront* [13], which computes a two-dimensional recurrence relationship where each cell in the output matrix is generated by a function whose arguments are the values in the cell immediately to the left, below, and diagonally left and below. Although some cells can be processed in parallel, due to the recurrence relationship, special care must be taken to ensure the proper ordering of dependent cell computations.

By default, Weaver includes all five of these **abstractions** as a part of the framework. However, because abstractions are just normal Python functions, it is possible for users to extend the existing ones or define their own abstractions specific to their workflow.

2.4. Nests

The final component of the Weaver programming model is the concept of **nests**. In Weaver, all workflows consist of a workspace and a DAG. The workspace is normally a directory that serves as a reserved storage area for outputs and any intermediate workflow artifacts, whereas the DAG encodes the relationships between tasks in the workflow. *Nests* are Weaver objects that represent both a workspace and a DAG. Whenever an abstraction is processed, it is done so in the context of a particular *Nest*, which captures any tasks produced by the abstraction being executed.

Users may utilize multiple *Nests* by using the *SubNest* constructor to produce additional sub-workflows with separate namespaces. Using a *SubNest* allows for the creation of hierarchical workflows, which means that a single workflow can be composed of a set of smaller workflows. Because each *Nest* has its own workspace, outputs and artifacts of each workflow are isolated, preventing naming collisions. Moreover, each sub-workflow contains its own DAG and thus runs independently from each other, which aids in scaling larger workflows that can be broken up into smaller data-processing pipelines.

A simple example of the use of *SubNests* is shown in Listing 3. In this example, we define two datasets (images on the local machine) and three functions (image transcoders). We then create separate *SubNests* and perform *Map* operations. In the first case, we create a `png_nest` explicitly and pass the new *SubNest* as the `nest` argument to the *Map* abstraction. All abstractions in Weaver take this `nest` argument; if it is not specified, then the default `CurrentNest` object is accessed from the global environment. After explicitly defining and passing a nest, we then create a hierarchy of nests by using Python's `with statement` syntax and Weaver's *SubNest* constructor. When the

DOI: 10.1002/cpe

```
# Datasets
pngs = FilesDataSet('/usr/share/pixmaps/*.png')
xpms = FilesDataSet('/usr/share/pixmaps/*.xpm')

# Functions
png_to_jpg = SimpleFunction('convert', out_suffix='jpg')
xpm_to_ppm = SimpleFunction('convert', out_suffix='ppm')
ppm_to_gif = SimpleFunction('convert', out_suffix='gif')

# Explicitly define and set subnest
png_nest = SubNest('pngs')
jpgs = Map(png_to_jpg, pngs, nest=png_nest)

# Implicity define subnests into hierarchy
with SubNest('xpms'):
    ppms = Map(xpm_to_ppm, xpms)
    with SubNest('ppms'):
        gifs = Map(ppm_to_gif, ppms)
```

Listing 3. Weaver subnest example.

`with statement` is executed in conjunction with the *SubNest* constructor, a new nest is created and immediately set as the `CurrentNest`. Because of this context created by the Python `with statement`, we do not have to pass the newly constructed *SubNest* explicitly to the proceeding *Map* operations. Note that child *SubNests* can reference datasets defined by ancestor *Nests*. Once we are out of the scope of the `with statement`, the previous nest is restored as the `CurrentNest`.

As explained, the Weaver programming model consists of datasets, functions, abstractions, and nests. Datasets identify a set of input files to be processed, whereas functions define executables that are used to process such files. Abstractions are higher-order functions that govern the pattern in which functions are applied to datasets and allow the user to take advantage of data parallelism to achieve increased performance. Because the input and output of each abstraction is simply another dataset, different abstractions can be pipelined together to form sophisticated scientific workflows. Nests allow for organizing multiple workflows together in a hierarchical manner.

3. EXECUTION MODEL

To construct a distributed workflow using Weaver, the user programs a specification in Python that invokes the various dataset, function, abstraction, and nest components that are provided by the Weaver framework as described earlier. Once the specification is complete, the user processes the script by using the Weaver compiler, which generates a sandbox (directory) that contains a DAG detailing the relationships between each task in the workflow. Additionally, if the user requested for executables and input data to be copied, then these files are also placed in the sandbox along with any scripts that were generated by the compiler. Using the sandbox as a workspace, a workflow manager schedules the tasks specified in the generated DAG by dispatching the jobs to a distributed execution engine. This section further examines the components of the Weaver execution model in detail.

3.1. Compiler

The core of the Weaver framework is a collection of Python packages and modules that provide the various datasets, functions, and abstractions described previously. To construct a workflow, users simply import and utilize the components provided in the Weaver libraries in addition to any existing Python modules that they wish to utilize. To generate a DAG for use with a workflow manager, the user processes the workflow script with the Weaver compiler.

As shown in Figure 2, the output of the Weaver compiler is a sandbox that primarily contains a DAG. If specified by the user, various input and executable files are also copied to this workspace. As such, the sandbox is the physical manifestation of the *Nest* concept described previously. When

DOI: 10.1002/cpe

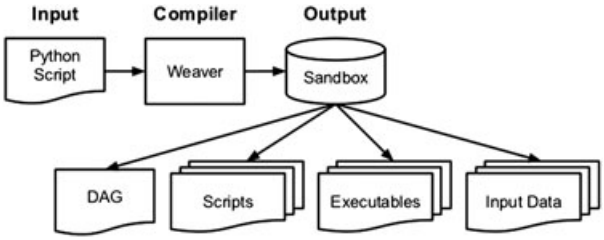

Figure 2. Weaver compiler.

the compiler begins, it creates an initial sandbox, which serves as the default location for the compiler's output and as the environment for the workflow manager. To construct the DAG that encodes the relationship between the various tasks specified by the workflow script, the compiler sets up an initial Weaver environment by importing the various components of the framework and setting required paths, and then, it evaluates the Python script using Python's `execfile` function.

During the evaluation of the workflow script, the compiler tracks the task generated by the abstractions. That is, whenever users invoke one of the Weaver abstractions in the Python script, *Functions* are applied to *DataSets* in the pattern proscribed by the *Abstraction*, and a sequence of tasks in the form of `(command, inputs, outputs)` tuples is registered with the current *Nest*. If the user specifies a *SubNest*, then a new sub-workflow is initialized and subsequent tasks are associated with this new *Nest*. This continues until the *SubNest* is out of scope or no longer specified. The tracking of *Nests* enables Weaver to construct hierarchical workflows that are isolated from each other. Likewise, the compiler also tracks what files need to be copied (executables or input data if specified) and which files require materialization (*PythonFunctions* and *ScriptFunctions*), and performs these operations while processing the script. When the whole script has been evaluated, the compiler then processes the generated task list for each *Nest* and generates a DAG description file in a format appropriate for the target workflow manager.

In other words, a DAG of tasks is generated as a side effect of the evaluation of the imperative Python script, which is similar to GRID superscalar [16] except that Weaver is a compiler, rather than a run-time system. In Weaver, the compiler tracks all these tasks and allows the users to perform various optimizations. For instance, rather than having a single DAG node responsible for a single task, the user may wish to aggregate a group of tasks into one single super DAG node. This is an example of clustering, which was shown by the Pegasus project to have significant performance benefits on the Montage and Tomography applications [17]. Weaver allows users to do this easily by simply *chunking* their datasets into sub-collections (i.e., lists of lists), which will notify the compiler to group generated tasks into super nodes. Another possible optimization technique is the use of instruction selection, which is discussed in detail in Section 3.3. Through the use of the various Python libraries provided by Weaver, users are able to quickly construct scientific workflows and then fine-tuned them with various optimization techniques.

3.2. Workflow manager

The output of the Weaver compiler is a sandbox environment containing a DAG and various files required for proper execution of the workflow. This workspace is used by a workflow manager as the storage area for the outputs of the workflow and any intermediate workflow artifacts generated during execution.

Currently, Weaver supports the Makeflow [13] workflow manager. When a workflow script is compiled with Weaver, a Makeflow DAG is generated in the sandbox directory. This DAG contains rules similar to those found in a normal Makefile that describe tasks in terms of the input and output dependencies and is used by Makeflow to form an internal graph of the entire pipeline. In this graph, the nodes are the data to be processed and the tasks to be executed with this data, and the links are the relationships between the tasks and the necessary input and output files. By forming this directed graph, Makeflow can accurately determine which tasks depend on others and can schedule

DOI: 10.1002/cpe

the work appropriately to optimize concurrency. Once the DAG is generated by Weaver, users pass it to Makeflow to perform the actual execution of the workflow.

The relationship between the user-defined Weaver workflow script, and the generated Makeflow DAG is shown in Table III. In this simple example, we wish to convert a thousand JPG images to PNG format. We can specify this workflow in three lines of Weaver as shown on the left side of the table. If we run the workflow script through the Weaver compiler, we get the Makeflow DAG displayed on the right side of Table III. Note that this is only an abbreviation of the Makeflow DAG, as the whole DAG would be 2000 lines, which is much longer than the three line Weaver specification.

The complete Weaver software stack is composed of three layers as shown in Figure 3. The first layer is the Weaver framework, which is used to generate a DAG. The second layer is the Makeflow workflow manager, which processes the DAG and dispatches jobs to the third layer, the various distributed execution engines. As a flexible DAG-based workflow manager, Makeflow provides access to multiple execution engine targets such as Condor [6], SGE, Work Queue [13], and local Unix processes. To perform workflow execution, Makeflow utilizes the master–worker paradigm to perform task scheduling and resource allocation.

Because Weaver generates Makeflow DAGs rather than directly scheduling for a specific execution engine, users of the framework can easily take advantage of multiple execution environments by simply selecting the appropriate platform at run-time. This flexibility allows users to adapt to the resources available to them without having to modify their workflow specification. Additionally, Makeflow utilizes a journal or log to record the progress of a workflow. This journal is stored as a plain text file in the Weaver workspace and can be used to collect provenance information such as the number of tasks failures, attempts, execution times, and so on. The log also enables Makeflow to resume or restart a failed workflow without rescheduling already completed tasks. Batch system specific logs such as a Condor log file are also stored in the Weaver workspace and can also be used to collect provenance information.

This system architecture is similar to other workflow systems such as Swift and Pegasus. For instance, Swift relies on Community Grid (CoG) Karajan execution engine [10, 18] to dispatch tasks and perform resource allocation, while Pegasus relies on Condor DAGMan [4, 5] for distributed execution. In this case, Weaver depends on a workflow manager such as Makeflow to perform the actual

Table III. Weaver and Makeflow directed acyclic graph (DAG) example.

Weaver source	Makeflow DAG

`jpgs=[str(i)+'.jpg' for i in range(1000)]`
 	
`conv = SimpleFunction('convert', out_suffix='png')`
`pngs = Map(conv, jpgs)`
	`0.png: 0.jpg /usr/bin`
	`  /convert`
	`/usr/bin/convert 0.jpg 0.png`
	`1.png: 1.jpg /usr/bin`
	`  /convert`
	`/usr/bin/convert 1.jpg 1.png`
	`2.png: 2.jpg /usr/bin`
	`  /convert`
	`/usr/bin/convert 2.jpg 2.png`
	`...`
	`999.png: 999.jpg /usr/bin`
	`  /convert`
	`/usr/bin/convert 999.jpg`
	`  /999.png`

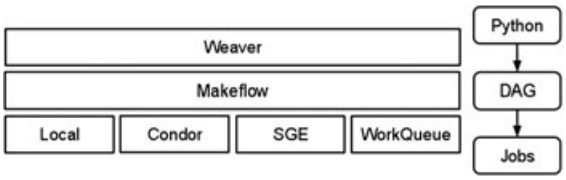

Figure 3. Weaver software stack.

DOI: 10.1002/cpe

execution of the workflow. The power and utility of Weaver is that it allows for users to rapidly construct and configure workflows in a high-level language. This is important because it is not always clear what the best decomposition of a workflow should be. Using Weaver, users can prototype and optimize their workflows in a concise and effective manner.

3.3. Optimized tools

As discussed earlier, Weaver implements a variety of distributed computing abstractions such as *All-Pairs*. Normally, Weaver *Abstractions* generate a sequence of task tuples in order to implement the appropriate execution pattern while relying on the workflow manager to assemble a DAG and determine proper execution order. This allows for the specification of sophisticated workflows consisting of high-level abstractions that are completely agnostic of the underlying execution engine. Therefore, one can view the abstraction primitives provided by Weaver as generic operations that work on any distributed execution platform.

Unfortunately, these generic operations are not necessarily the most optimal or efficient implementations of the particular abstraction. One reason is that the Weaver programming model requires input and output data to be manifested as physical files. In workflows that involve many short running applications, this model will yield an inefficient implementation because each data record will need to be instantiated on the filesystem and each application will appear as a separate task in the DAG. Depending on the execution engine, the latency for each job start up can greatly diminish the performance of such a workflow.

Another reason why the generic implementations may be non-optimal is that some abstractions require intimate knowledge of the underlying distributed system to be effective. For instance, the MapReduce implementation presented by Google is successful not only because of the data-parallel task scheduling but also because of its ability to take advantage of data locality.

Because there already exists optimized tools that implement some of the abstractions provided by Weaver and because the generic implementations may be non-optimal, Weaver allows users to easily specify which version of the abstraction to utilize. To use an optimized native version of an abstraction, the user sets the `use_native` keyword argument in the abstraction's constructor to `True`. If the native version is available for that abstraction, then Weaver will use that optimized tool instead of a generic version.

This ability to choose between a generic operation and a specialized operation is similar to the use of single instruction, multiple data (SIMD) instructions with computer processors. In a conventional compiler, operations are compiled using the lowest-common-denominator instruction set for a particular platform. However, if the user requests an architecture-specific optimization, such as SIMD instructions, the compiler will output an optimized program code that takes advantage of the hardware. Weaver behaves in a similar manner. By default, it generates DAGs with generic implementations of any requested abstraction. If the user specifies the use of a native tool, then Weaver will forgo the generic version and instead use the optimized tool. As shown in Figure 4,

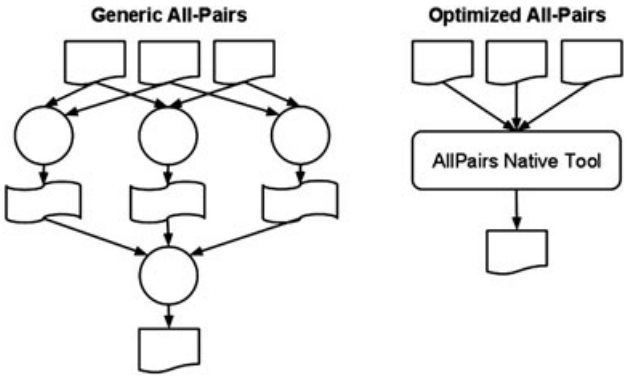

Figure 4. Generic versus optimized All-Pairs directed acyclic graphs.

DOI: 10.1002/cpe

a native tool can greatly reduce the size of the DAG because it no longer needs to implement the whole abstraction as a set of task rules and instead uses the specialized software as a single optimized task.

The ability to choose between a generic implementation and an optimized one makes Weaver flexible and attractive for exploring new abstractions. For instance, new patterns of execution can be quickly developed as a set of DAG relationships and tested on a variety of execution engines. If the new abstraction proves useful, then an optimized implementation can be developed and then plugged into Weaver seamlessly.

3.4. Application packager

Although Weaver supports specifying the file dependencies of a particular function, it is often desirable to execute these individual application components independently (i.e., separate from the workflow). Moreover, some applications require data files or configuration that are difficult or cumbersome to specify in Weaver. To solve this problem, Weaver includes an application packager named **Starch** [14] to create SAAs, which is single file that contains all of the dependencies, configuration, and environment variables necessary to execute the application.

To create an SAA using Starch, the user specifies a list of executables and libraries to include in the target execution image along with the appropriate shell command to run when the SAA is executed by the user. By default, Starch will automatically search for any dynamically linked libraries required by the executables specified and embed those in the SAA. If any special input data files are required by the application, they may also be specified to be included in the SAA. Additionally, special environmental variables and other run-time configuration options can be stored in the SAA through the use of user-defined environment scripts that will be imported before the SAA's command is executed.

Once all of the necessary options are specified, the executables, libraries, and environment scripts are archived and compressed as a standard Unix tarball. This tarball is then appended to Starch's template shell script to generate an SAA. When the SAA is executed, the wrapper shell script will automatically extract the embedded archive, configure the environment, and run the user-specified shell command. To integrate Starch-generated application packages, users simply replace the normal executables with the constructed SAA's in their function specifications.

Altogether, the separation of workflow specification, task management, and execution engine make the Weaver execution model quite flexible. The compiler reads in a specification and outputs a sandbox containing a DAG. This in turn is used by a workflow manager, currently Makeflow, which dispatches jobs onto a variety of execution platforms. To allow users to take advantage of existing optimized abstraction software, Weaver provides a mechanism for providing hints to the compiler to use the specialized tool rather than a generic implementation. Likewise, to simplify the distribution of applications and to enable testing of individual workflow tasks, Weaver provides Starch an applications packager that creates SAAs. With the described Weaver software stack, it is possible to script a workflow specification in Python and effectively and reliably execute it on multiple distributed systems.

4. APPLICATIONS

In this section, we analyze and evaluate four different real-world scientific workflows performed at the University of Notre Dame. Each of these distributed applications was scripted using the Weaver framework and was executed on our local Condor and SGE clusters, which are composed of hundreds of 64-bit Red Hat Linux (RHEL5) machines. These clusters are composed of a heterogeneous mixture of machines with AMD Opterons to Intel Core 2 Duos and an assorted mixture of memory configurations. The purpose of the analysis is to evaluate the effectiveness of scripting large distributed workflows by examining different throughput and scalability measurements produced by running the four workflows generated by Weaver on our campus grid. In doing so, we identify the various strengths and weaknesses of our framework as discovered through practice and experience.

DOI: 10.1002/cpe

4.1. Biometrics

One of the primary uses of Weaver is to construct and streamline biometric experiments, which usually consist of multiple data-processing stages as described in Section 1. First, the interested dataset must be selected or extracted from the original data repository. In our case, this is the BXGrid biometrics repository, which contains over 8 TB of image and video data collected by our collegues in the Computer Vision Research Lab [19]. Next, the raw data must be transformed into a format suitable for experimentation. Finally, the experiment is performed, which, in the case of biometrics, involves comparing all members of the dataset with each other to produce a matrix of match scores.

Listing 4 provides a Weaver implementation of the biometrics workflow depicted in Figure 1 as just described. To specify the interested dataset, the user first instantiates a database connection to BXGrid biometrics repository and, then, uses the *Query* function to select records with the desired properties. In this example, we use the built-in ORM query system (the *Query* and *Or* functions) to select irises that are marked *enrolled* and have the color *green* or *blue*. These selected files can then be converted to an appropriate format by using the *Map* abstraction. The results of the conversions form a new dataset that is then used as arguments to the *All-Pairs* abstraction, which schedules tasks to compare each member of the converted dataset with each other. As can be seen, it is quite easy to reuse the same workflow for different sets of records. The user only needs to modify the *Query* and recompile using Weaver to produce biometrics workflow custom-tailored to the interested dataset.

While utilizing this workflow script on our campus grid, we noticed some interesting behavior regarding the scalability and performance of using the native optimized All-Pairs tool versus the generic implementation produced by Weaver. Figure 5 shows the results of biometric experiments with the native tool optimization option enabled and disabled.

In these experiments, we varied the size of the workloads by using datasets consisting of 10 to 1000 301-KB images. Each of these image files was converted to a specialized bit format using the *Map* abstraction. These bit files were then compared with each other using *All-Pairs* to generate a

```
db      = SQLDataSet('db', 'biometrics', 'irises')
irises = Query(db, db.c.state == 'Enrolled',
                Or(db.c.color == 'Blue', db.c.color == 'Green'))

iris_to_bit  = SimpleFunction('convert_iris_to_template', out_suffix='bit')
compare_bits = SimpleFunction('compare_iris_templates')

bits = Map(iris_to_bit, irises)
AllPairs(compare_bits, bits, bits, output='scores.txt')
```

Listing 4. Weaver biometrics experiment source.

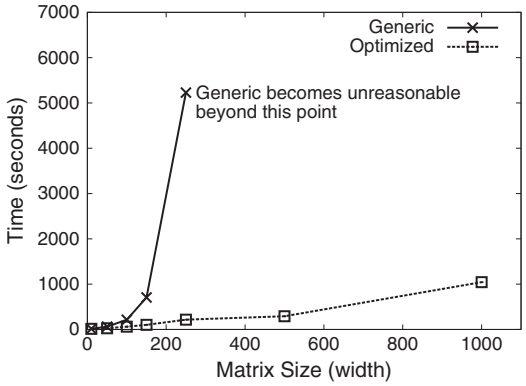

Figure 5. Biometrics workflows using generic versus optimized abstractions.

DOI: 10.1002/cpe

scoring matrix. For each dataset, we generated workflows with and without the optimized native implementations using Weaver's SIMD-like compiler mechanism and executed both on our campus cluster. From the graph, it is clear that the generic version scales poorly as the number of images increases, whereas the optimized native version scales almost linearly. The generic implementation is several orders of magnitude slower because it must use files for intermediate storage and is unable to take advantage of specific execution engine environment features. For instance, the optimized *All-Pairs* tool uses the Work Queue execution engine to enable low-latency work dispatching and takes advantage of multi-core systems by intelligently scheduling tasks to multiple cores. Moreover, the native tool stores the intermediate outputs in memory and outputs the results as a single file.

The results from the biometrics scaling and throughput experiments show that there is a performance penalty associated with using the generic abstraction implementations. The Weaver programming model requires that input and output data must be stored as files, which greatly constrains the performance of certain types of workflows as demonstrated earlier. In these cases, a specialized native tool easily outperforms the generic implementation because it is not constrained by the programming model and can take advantage of particular platform features. Fortunately, Weaver provides a mechanism to take advantage of these optimized abstraction implementations, allowing for specialized versions to be used when possible. Moreover, it is important to note that the availability of generic abstraction implementations is useful and necessary for the cases where a native implementation does not exist or does not match the semantics of the user's workflow. This allows users to still take advantage of a particular pattern of work, even if the tool is not available for their particular distributed computing platform.

4.2. Data transcoding

The second major application of Weaver is transcoding over 500,000 multimedia files contained in the BXGrid biometrics data repository. The BXGrid system is composed of two main distinct parts: an object storage system based on rich object archival system [20] and a Web portal front–end to the data archive [19]. One of the key user interactions with the BXGrid Web portal is viewing snapshots of the collected image and video data. This allows researchers to quickly and easily enroll, verify, and modify data in the repository for purposes of maintaining data quality and integrity [21].

Because some of the raw biometric data is quite large (e.g., 2-GB high-definition videos), it is impossible to present the user with data directly from the storage system. Instead, smaller thumbnails must be generated from the raw data and used by the Web portal as shown in Figure 6. Initially, this thumbnail generation was performed on-demand by the Web server; that is, a thumbnail was created whenever a user browsed a collection and required viewing an image or video file. This stressed the Web server and, at times, crippled it because of the excessive bandwidth and CPU required to generate thumbnails on the raw data. Moreover, for large files such as videos, this on-demand transcoding lead to unacceptable latencies (e.g., several minutes to produce a thumbnail) that negatively impacted the user experience on the Web site. Our solution to this problem was to use Weaver to periodically generate distributed workflows that transcoded the raw data from the repository into smaller thumbnails and cache them on the Web server.

Listing 5 is the Weaver script used to construct these distributed transcoding workflows. As can be seen, BXGrid has four major types of files: point data contained in *abs* format, various compressed *images*, *raw* digital photos, and different types of videos. Each file type requires a separate transcoding tool that is specified in the workflow along with the appropriate ORM query necessary to select the particular files. The Weaver script constructs the workflow in the following manner:

1. For each file type, find the missing thumbnails. Note that, in BXGrid, we generate three thumbnail sizes (small, medium, and large) and, thus, we check the cache for each size.
2. Check if the number of missing files is larger than a user-defined `CHUNK_SIZE`. If it is, then split the larger dataset into smaller sized chunks.
3. For each chunk or dataset, create a *SubNest*, and schedule the thumbnail generation tasks. After the thumbnail is created, move it to the Web server cache using a *ScriptFunction* that automatically sets the appropriate file permissions and moves the file.

DOI: 10.1002/cpe

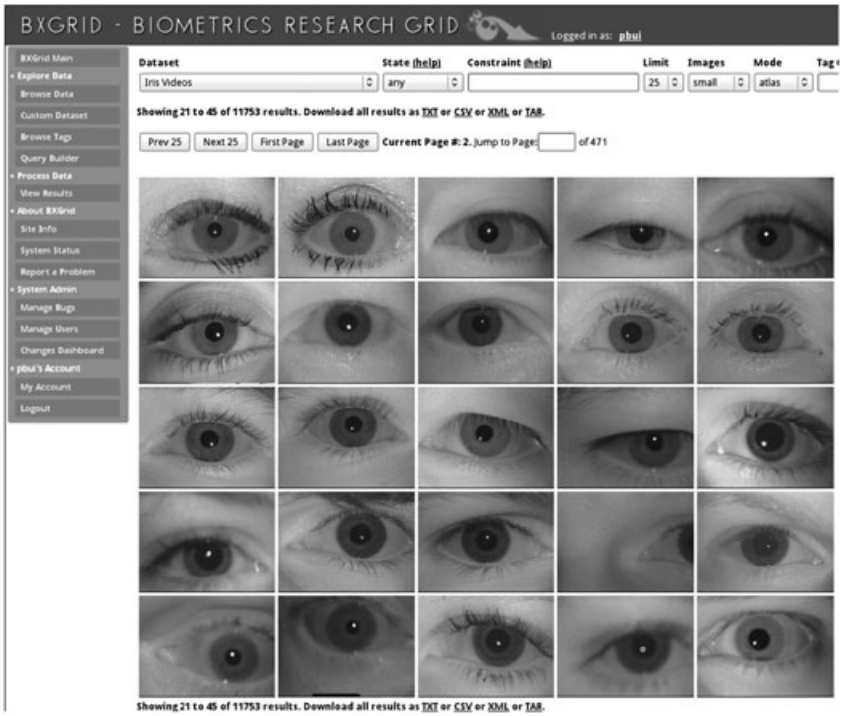

Figure 6. BXGrid Web portal screenshot.

```
from transcode import BXGrid, GenerateThumbnails, find_missing_thumbnails

CHUNK_SIZE = 2500
FILE_TYPES = (
    ('abs',
     'convert_abs_to_gif_animation',
     lambda db: Query(db, Or(db.c.type == 'gz', db.c.type == 'abs.gz'))),
    ('image',
     'convert_image_to_jpg',
     lambda db: Query(db, Or(db.c.type == 'JPG', db.c.type == 'ppm',
                             db.c.type == 'tiff'))),
    ('raw',
     'convert_raw_to_jpg',
     lambda db: Query(db, Or(db.c.type == 'NEF', db.c.type == 'cr2'))),
    ('video',
     'convert_video_to_gif_animation',
     lambda db: Query(db, Or(db.c.type == 'avi', db.c.type == 'mp4',
                             db.c.type == 'MPG', db.c.type == 'ts'))),
)

bxgrid = BXGrid()

for file_type, command, query in FILE_TYPES:
    missing = [find_missing_thumbnails(f) for f in query(bxgrid)]
    missing = filter(lambda x: x, missing)

    if len(missing) > CHUNK_SIZE:
        with SubNest(file_type):
            for i, chunk in enumerate(Chunk(missing, CHUNK_SIZE)):
                with SubNest('%s.%04X' % (file_type, i)):
                    GenerateThumbnails(file_type, command, chunk)
    else:
        with SubNest(file_type):
            GenerateThumbnails(file_type, command, missing)
```

Listing 5. Weaver data transcoding source.

DOI: 10.1002/cpe

The use of *SubNests* allows us to create a hierarchical workflow, where the sub-workflow for each file type is isolated from the others. This allows for easier monitoring and provenance tracking while increasing scalability. By creating a hierarchy of workflows, we can execute multiple transcoding workflows concurrently and thereby increase our overall throughput. That is, by using the *SubNest* mechanism, we create isolated and independent sub-workflows. These sub-workflows can then be scheduled and executed concurrently, with different masters or controllers running on different machines, which thus increases scalability as workers will not be bottlenecked by a single-master/controller compute node.

It is for this reason that we also split large datasets by a user-defined `CHUNK_SIZE`. From our experience, breaking a dataset transcoding workflow into separate, smaller sub-workflows increases the performance and reliability of our distributed application. Not only does the hierarchical design allow for workflows to be executed in parallel, but it also means that the workflow manager, in this case Makeflow, has to keep track of a fewer amount of tasks and files and can thus schedule faster and use less resources.

To automate our transcoding workflow, we schedule the Weaver script to run every night via `cron` on the Web server where it generates the appropriate workflows for any image or video data ingested into the BXGrid system the previous day. When the Weaver script runs, it checks the cache for missing thumbnails and queues tasks to generate those files as explained earlier. Next, workers are dispatched to our local campus grid. After this, the main workflow is executed, and it in turn executes the sub-workflows for each of the file types. Each of these sub-workflows is in charge for a subset of the thumbnails and shares the workers from the initial pool.

Table IV provides the summary of three separate transcoding activities performed using Weaver. Most of the image transcoding work on files that are relatively small (less than 1 MB), and thus, the batch job overhead incurred for scheduling and dispatching the task is often greater than the cost of simply fetching the file and transcoding it locally. However, because Work Queue workers serve as glide-in jobs, we can utilize Work Queue low-latency batch job dispatching mechanism to minimize these overheads. As Table IV shows, for larger file types such as raw or video, the Weaver-generated workflows are able to achieve a higher average data throughput. Overall, the automated data transcoding with Weaver has been a relative success. The system described has been in place for about 8 months and runs daily on our live BXGrid Web portal. The generation of thumbnails ahead of time has been the key in maintaining a positive interactive experience for the biometrics researchers who regularly use the Web site. Because Weaver supports nested workflows through the hierarchical *SubNest* system, we are able to split larger datasets across smaller and more manageable workflows. Likewise, this hierarchical design allows us to run multiple sub-workflows concurrently. Finally, because Weaver is designed to be agnostic about the target distributed system, we can target multiple distributed execution platforms. In this case, we utilize the Work Queue master/worker framework for its low-latency dispatching due to the large number of tasks required by transcoding workflows.

4.3. Expressed sequence tag pipeline

Another scientific workflow that utilizes Weaver at the University of Notre Dame is a biological computing pipeline used by our bioinformatics colleagues to analyze the expressed sequence tags (ESTs) of various biological entities such as butterflies [14]. In an EST pipeline, ESTs are first collected from a genome assembler and then sequenced. Next, the generated sequences are compared and analyzed by using Basic Local Alignment Search Tool (BLAST) [22] with multiple databases.

Table IV. Sample transcoding metrics.

File type	No. of files	Avg. file size	Run-time	Data throughput	Task throughput
Image	20 388	0.4 MB	74 min	1.8 MB/s	5.32 tasks/s
Raw (CR2)	3 541	19.0 MB	56 min	20.0 MB/s	2.09 tasks/s
Video (MP4)	1 680	13.0 MB	46 min	8.4 MB/s	1.20 tasks/s

DOI: 10.1002/cpe

The comparison outputs from BLAST are combined with statistical information computed by additional software to form the final merged analysis result.

Initially, our bioinformatics colleagues created an EST pipeline composed of an *ad hoc* mixture of Ruby and shell scripts, which had to be run manually. Unfortunately, this workflow was cumbersome and took over a week to execute. Eventually, this analysis pipeline was organized by a PERL wrapper script, a common technique amongst bioinfomaticians. This PERL script soon became unwieldy and difficult to maintain or extend. Moreover, because of the intricate dependencies of the different scripts and tasks, it was difficult to isolate an individual function from the whole workflow and thus impractical to test or debug. It was also difficult to port the workflow to the various campus grid resources available because many parts of the PERL wrapper were specific to their local cluster environment. To rectify this, the group rewrote their pipeline using Weaver.

Although the users were new to the Python programming language, they were able to reconstruct their bioinformatics workflow in less than a month using Weaver. The result of this port is the EST pipeline depicted in Figure 7. To tackle the problem of complex dependencies, they packaged individual parts of their pipeline using the Starch application packager. This resolved their complex dependency problems and allowed them to test individual workflow functions outside the scope of their pipeline. Moreover, because of the self-contained nature of the SAAs produced by Starch, they could easily share the packaged applications with each other and thereby increase their collaboration and productivity. This is particularly important because parts of their pipeline depended on the BioRuby library, which is not available on most of the cluster machines. Because Starch supports embedding libraries into the application package, this allows for the executable to be run portably on any of the cluster machines.

Table V compares the performance of the Weaver-generated workflow against that of the PERL pipeline. As can been seen, Weaver provides slightly better performance, though not by much, for the overall EST pipeline and for the BLAST sub-workflows. According to the bioinformatics

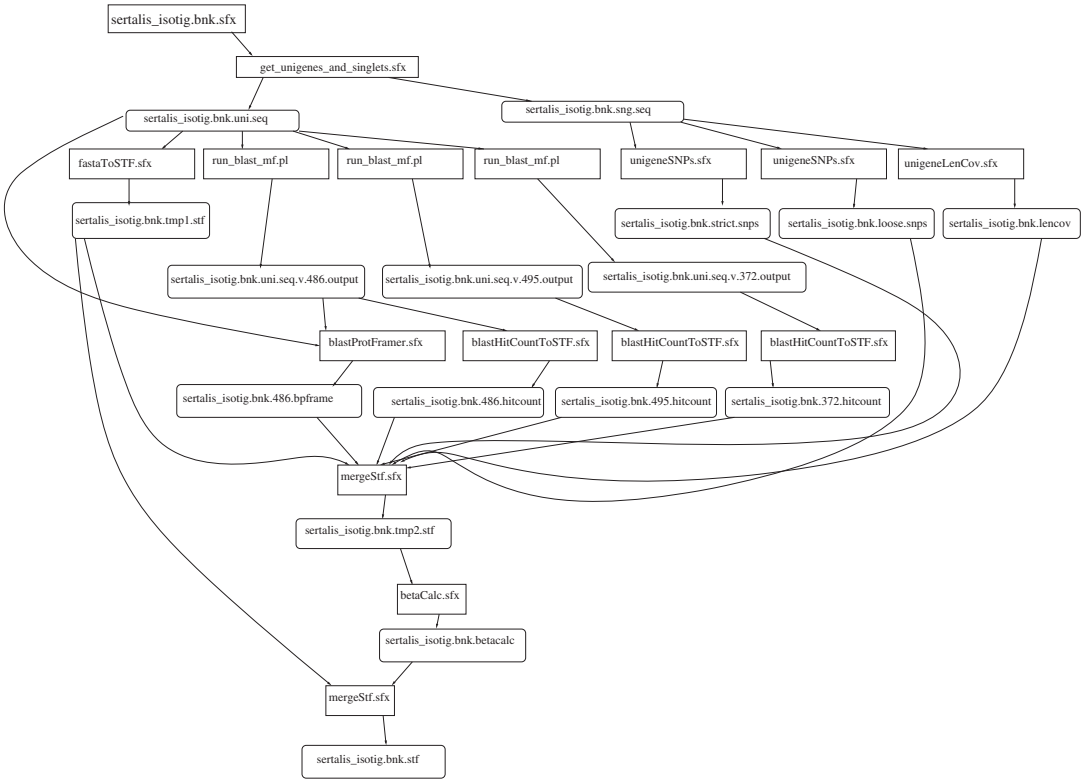

Figure 7. Expressed sequence tag pipeline.

DOI: 10.1002/cpe

Table V. Expressed sequence tag (EST) workflow run-times.

Application	Weaver (s)	PERL (s)
EST pipeline	2529	2882
BLAST	665	677

researchers, the main advantages of using the Weaver framework instead of the custom PERL script were the following:

1. **Encapsulation**. Using Starch allowed the researchers to create application packages of their executables to hide the complexity of their programs. In some cases, archives contained over 14,000 files. The SAAs enabled testing of individual functions and eased deployment on different distributed systems because of their portable nature.
2. **Portability**. Because Weaver does not rely on any particular feature of a distributed execution engine, it can produce platform agnostic workflows. As such, the EST pipeline generated by Weaver was able to be deployed on different cluster configurations and with different batch systems.
3. **Code clarity**. The Weaver workflow script ended up being approximately a third shorter than the PERL wrapper. Moreover, the researchers found the resulting Weaver script to be much more concise and understandable despite their lack of familiarity of Python. Additionally, Weaver's component library also helped encourage modular code and object-oriented design, which the researchers felt would improve the long-term maintainability of the EST pipeline.

All of these features enabled the bioinformatics researchers to successfully use Weaver to port their *ad hoc* set of scripts to a more maintainable code base. The final workflow constructed using Weaver was slightly more efficient than their hand-coded system, and offered additional advantages such as testing of individual segments of their pipeline, greater portability, and easier maintenance.

4.4. Paired-end mapper pipeline

The final workflow presented here is the application of Weaver to paired-end mapper (PEMer) [23], a structural variation detection workflow. Given a reference genome and a set of mated pair sequence queries, the PEMer analysis pipeline is designed to infer genomic structural variants from the sequencing data using the following steps:

1. Preprocessing on a list of paired DNA sequences to generate a set of mate pairs.
2. Independently align mate pair ends using a tool such as MegaBLAST [24].
3. Find optimal placement of mate pair reads according to alignments that seek to minimize the occurrence of outliers.
4. Identify the mate pair outliers.
5. Categorize sets of outlier mate pairs as structural variations if N or more independent paired ends can be clustered according to each variation.

The stock PEMer pipeline outlined earlier executes well for smaller datasets but quickly becomes intractable in terms of running time on larger datasets. For instance, running the PEMer genome analysis on the water flea *Daphnia pulex*, which has a 227 million-character genome, sequentially on an 8-core 32-GB machine would require more than 2 weeks to complete. In order to address this problem, one of our bioinformatics colleagues decided to use the Weaver framework to refactor PEMer so that parts of the pipeline would execute on a distributed system. Although she was a distributed systems and Python novice, she was able to complete this port on her own in less than 2 months and achieve significant throughput improvements.

Figure 8 displays the workflow our bioinformatics colleague constructed to make PEMer execute on distributed systems. Step 1 in the diagram is the first phase of the distributed PEMer workflow and corresponds to the first four steps of the normal PEMer pipeline, while step 2 is the last step. Both of these two steps are separate workflows but run end-to-end. The reason for this separation is

DOI: 10.1002/cpe

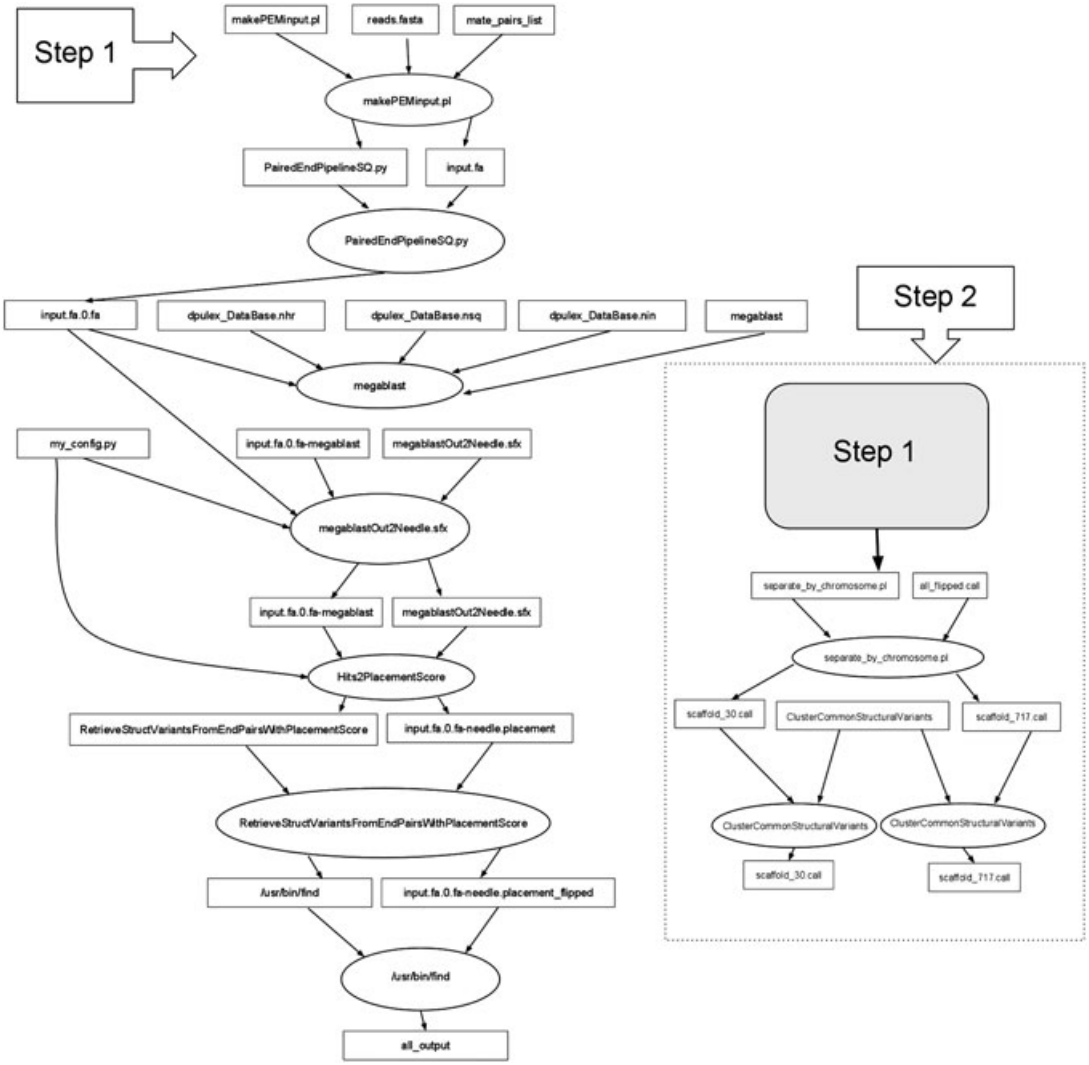

Figure 8. PEMer pipeline.

that Weaver currently only supports static workflows, meaning that all of the inputs and outputs of workflow must be known at compile time and that new dependencies and tasks cannot be generated dynamically from the same script. Recall that the last step in the PEMer pipeline is to categorize the outlier mate pairs on the basis of clustering behavior. Unfortunately, it is not known ahead of time how many outliers are present, therefore, this categorization step cannot be specified in the first workflow. A workaround, as implemented by our colleague, is to simply have the last tasks in the first workflow use Weaver to compile a new workflow DAG at run-time on the basis of the generated outliers found in the initial step and then execute the dynamically generated workflow to complete the analysis. Although relatively inelegant, the solution is successful and works around a current limitation in Weaver.

A summary of the results of distributed PEMer workflow are shown in Table VI. To evaluate the workflow, our colleague executed the original pipeline on an 8-core 32-GB machine. After 2 weeks, the pipeline had not advanced beyond the third step, and it was stopped. Using 2 weeks of running time as the lower bound then, our colleague ran the workflow on our campus cluster using both the Condor and Work Queue execution engines. As can be seen in Table VI, the Weaver-generated workflows were able to complete in less than a day. Using Condor, she was able to get 91.5× speedup with 100 nodes and 211.5× speedup with 300 nodes. With Work Queue, she achieved 85.0× speedup

DOI: 10.1002/cpe

Table VI. Paired-end mapper workflow run-times.

System	No. of Workers	Running time	Speedup
8-core single node	1	>2 weeks	1.0
Condor	100	19:16	91.5
Condor	300	7:45	211.5
Work Queue	100	23:44	85.0
Work Queue	300	11:06	169.0

with 100 workers and 169.0× speedup with 300 workers. This means that the workflows were able to scale up with additional workers or compute nodes.

Using Weaver, our bioinformatics colleague, who was a novice to both distributed systems and Python, was able to successfully convert a complex genome analysis pipeline to a distributed workflow. Although she had to use some workarounds to overcome some limitations in the Weaver framework, she was still able to achieve significant performance improvements and produce a scalable distributed PEMer workflow.

Altogether, these four different scientific applications demonstrate the different aspects of Weaver and how they can be used to solve real-world problems. The biometrics experiment pipeline took advantage of Weaver's ability to combine multiple abstractions and plug in optimized native implementations when available. The multimedia transcoding application utilized Weaver's hierarchical workflow feature to construct large automated data-processing workflows that ran effectively and reliably on multiple distributed platforms. The EST pipeline showed that novices could port existing legacy workflows to the Weaver framework and gain in maintainability and code clarity. Additionally, the use of Starch-enabled packaging complex applications into isolated, portable, and testable standalone archives for use in distributed workflows. The PEMer pipeline demonstrated that non-experts can effectively port data-processing pipelines to distributed systems by using Weaver and achieve significant performance improvements. After evaluating these applications, it is clear that Weaver is a capable means of scripting together scientific workflows for use on distributed systems.

5. RELATED WORK

Because distributed computing resources can be quite complex to use and often require significant effort and knowledge to utilize effectively, there has been a large amount of previous and ongoing research focused on providing users with simplified and efficient programming interfaces to these systems. Like Weaver, these distributed computing frameworks provide a compact programming model where the user specifies their workflow that is then translated into a set of tasks to be performed by a distributed execution engine.

Pig [25] and Sawzall [26] are two languages that provide a high-level interface to MapReduce [1]. The former targets the open source Hadoop [27] MapReduce platform, whereas the latter runs on the Google's proprietary MapReduce system. Both of these languages provide a simplified programming model composed of datasets and functions that is presented as new declarative programming languages with an SQL-like syntax. Because these languages are tightly tied to the MapReduce abstraction, the user is constrained in the types of workflows that they can efficiently specify. Cascading [28] is a Java library built on top Hadoop that allows users to explicitly construct dataflow graphs in order to program data-parallel pipelines that run on Hadoop's MapReduce. FlumeJava [7] is another Java library that runs on top of Hadoop and also supports constructing data-processing pipelines by performing operations on a set of parallel collections provided by the library. Because of the nature of the Hadoop platform, it can be difficult to integrate legacy or external software.

For more traditional workflows, there has been an ongoing research into specifying distributed workflows as DAGs. DAGMan [4] and Pegasus [5] are two such workflow specification languages that allow the user to specify a set of tasks to compute and the relationship between each task. Each of these systems provide a custom programming language and a compiler or interpreter that takes

DOI: 10.1002/cpe

the job specification and produces a static workflow graph. A similar type of workflow system is Kepler [29], which is a sophisticated scientific workflow application that allows users to construct workflows using a graphical interface. Oozie [30] is an XML-based DAG workflow system that runs on top of Hadoop. Although these tools are effective and scale well to large workflows, from our experience, the manual construction of DAGs can be tedious and error-prone.

Dryad [3] is another attempt at simplifying the construction of distributed workflows through the construction of graphs. Because the work of building a workflow graph is rather low-level and complex, the authors of Dryad suggest the use of various higher-level tools such as DryadLINQ [8]. This programming construct takes advantage of the LINQ programming idiom in Microsoft's .NET system to allow the specification of MapReduce-type workflows using a single LINQ [31] expression. Another language built on top of Dryad is SCOPE [32], which is a declarative scripting language where programs are written in a variant of SQL. Like Pig and Sawzall, these Dryad-based languages are tied to their distributed computing platform and are limited to the MapReduce programming model.

Swift [10] also tackles the problem of specifying diverse scientific workflows but does so by providing a general-purpose programming language complete with a data type system. In Swift, users construct data structures representing their input and output data and specify functions that operate on these structures in a custom programming language. This specification is then compiled into a set of abstract computation plans, which is processed by the CoG Karajan [18] execution engine, which works in conjunction with the Swift run-time system to execute the plans.

GRID superscalar [16] demonstrates the use of an imperative programming language to implicitly construct workflows. In the GRID superscalar programming environment, users utilize either C/C++ or PERL in conjunction with Common Object Request Broker Architecture interface definition language specifications of the tasks to automatically generating a task data-dependent workflow graph. This workflow generation an execution is accomplished using a run-time library that dispatches tasks in a master–worker paradigm.

Overall, Weaver shares many of the important features present in these projects. Like traditional grid workflow systems such as Condor DAGMan, Pegasus, and TAVERNA, Weaver utilizes a DAG-based workflow engine and provides a user-directed language-based approach to constructing distributed workflows [33]. Because it does not force users to define workflows in terms of graph nodes and links, Weaver is most similar to DryadLINQ, Swift, and FlumeJava in providing a high-level programming interface to the underlying distributed systems. Like FlumeJava but unlike Swift, Weaver builds on top of an existing programming language Python rather than introduce a new one. This takes advantage of Python's user-friendliness and allows programmers to utilize the plethora of existing Python software. Likewise, Weaver is not restricted to a single programming construct as in DryadLINQ, Pig, and Sawzall, but it encompasses a whole library of components that form a domain-specific language for distributed computing. Finally, unlike most of the projects mentioned previously, Weaver supports multiple distributed computing abstractions as standard components, enabling scientific researchers to incorporate powerful distributed computing tools into their workflows. All of these features make Weaver a unique and powerful framework for scripting together distributed applications.

6. CONCLUSIONS AND FUTURE WORK

Despite Weaver's usefulness and success, there are still a few promising areas for further research and work. For instance, it would also be interesting to expose the notion of data locality in the framework. Currently, Weaver collects the input data in a sandbox. When Makeflow is executed, the data and executables are sent to the appropriate computational nodes for processing. For large datasets, this data transfer can be problematic and hinder throughput (as in the case of transcoding large sets of videos). If an active storage system such as Hadoop or Chirp [34] were available to the user, then it would be better to only send the executables to where the data is and perform the processing there. Some initial work has been performed on the lower software layers to address this problem, but it is not clear how this locality information should be exposed at the Weaver level or how to take full advantage of it.

DOI: 10.1002/cpe

Another possibility for future work is to consider more dynamic workflows. Currently, Weaver is used to compile a workflow, which is then executed using Makeflow. Although this works well for static pipelines, there are some applications that require more dynamic workflows such as the PEMer pipeline evaluated in this paper. Another common example of this type of dynamic application is a simulation where the same tasks are repeated with small changes between each iteration. To account for this type of work, users can conceivably utilize Weaver as an interpreter with just-in-time compilation to iteratively generate workflows at run-time rather than producing a single static workflow. Another approach would be to implement a method of persistently check-pointing a Weaver script. This would require defining a consistent workspace convention for keeping track of the progress of workflow execution and repeatedly calling the same Weaver script. On reinvocation, the Weaver script could trace the progress of its execution and jump to where it left off.

As demonstrated by the applications presented in this paper, Weaver enables researchers to solve real-world problems by scripting distributed scientific workflows in Python. In this paper, we presented the Weaver programming model, which consists of datasets, functions, abstractions, and nests. We also explained the Weaver execution model and discussed the software stack that consists of the Weaver compiler, the Makeflow workflow manager, native optimized tools, and the Starch application packager. To demonstrate the use of the framework, we provided four applications constructed using Weaver and evaluated the effectiveness of the framework in the context of scripting scientific workflows for distributed systems. Overall, Weaver is a capable and effective framework for constructing distributed scientific workflows.

ACKNOWLEDGEMENTS

We gratefully acknowledge the support of the US Department of Education through a Graduate Assistance in Areas of National Need (GAANN) Fellowship for Peter Bui (award P200A090044). This work was also supported in part by National Science Foundation grants NSF-CNS-0643229, NSF-CCF-0621434, and NSF-CNS-0855047.

REFERENCES

1. Dean J, Ghemawat S, Google Inc. MapReduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. USENIX Association, 2004.
2. Moretti C, Bulosan J, *et al.* All-Pairs: an abstraction for data intensive cloud computing. *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Miami, FL, 2008; 1–11.
3. Isard M, Budiu M, *et al.* Dryad: distributed data parallel programs from sequential building blocks. *Proceedings of EuroSys*, Lisbon, Portugal, 2007.
4. The directed acyclic graph manager, 2002. Available from: http://www.cs.wisc.edu/condor/dagman.
5. Deelman E, Singh G, *et al.* Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal* 2005; **13**(3):219–237.
6. Thain D, Tannenbaum T, *et al.* Condor and the grid. In *Grid Computing: Making the Global Infrastructure a Reality*, Berman F, Fox G, Hey T (eds). John Wiley: Chichester, 2003.
7. Chambers C, Raniwala A, *et al.* FlumeJava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '10. ACM: New York, NY, USA, 2010; 363–375.
8. Isard M, Yu Y. Distributed data-parallel computing using a high-level programming language. In *SIGMOD '09: Proceedings of the 35th SIGMOD International Conference on Management of Data*. ACM: New York, NY, USA, 2009; 987–994.
9. Bui P, Yu L, *et al.* Weaver: integrating distributed computing abstractions into scientific workflows using Python. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10. ACM: New York, NY, USA, 2010; 636–643.
10. Zhao Y, Dobson J, Foster I, Wilde LMM. A Notation and System for Expressing and Executing Cleanly Typed Workflows on Messy Scientific Data. *SIGMOD Record.* 2005; **34**:37–43.
11. Barker A, van Hemert J. Scientific workflow: a survey and research directions. In *Seventh International Conference on Parallel Processing and Applied Mathematics, Revised Selected Papers*, Vol. 4967, Wyrzykowski R *et al.* (eds), LNCS. Springer: Berlin, Germany, 2008; 746–753.
12. Python Programming Language, 2010. Available from: http://www.python.org/.
13. Yu L, Moretti C, *et al.* Harnessing parallelism in multi-core clusters with the all-pairs, wavefront, and makeflow abstractions. *Cluster Computing* 2010; **13**:243–256.

DOI: 10.1002/cpe

14. Thrasher A, Carmichael R, *et al.* Taming complex bioinformatics workflows with Weaver, Makeflow, and Starch. *Workflows in Support of Large-Scale Science (WORKS), 2010 5th Workshop on*, New Orleans, LA, 2010; 1–6.
15. SQLAlchemy, 2010. Available from: http://sqlalchemy.org/.
16. Sirvent R, Pérez JM, *et al.* Automatic grid workflow based on imperative programming languages: research articles. *Concurrency Computing: Practice and Experience* August 2006; **18**:1169–1186. DOI: 10.1002/cpe.v18:10.
17. Singh G, Kesselman C, *et al.* Optimizing grid-based workflow execution. *Journal of Grid Computing* 2005; **3**:201–219.
18. von Laszewski G, Hategan M. Workflow concepts of the Java CoG kit. *Journal of Grid Computing* 2005; **3**:239–258. DOI: 10.1007/s10723-005-9013-5.
19. Bui H, Kelly M, *et al.* Experience with BXGrid: a data repository and computing grid for biometrics research. *Journal of Cluster Computing* 2009; **12**(4):373.
20. Bui H, Bui P, *et al.* ROARS: a scalable repository for data intensive scientific computing. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10. ACM: New York, NY, USA, 2010; 766–775.
21. Bui H, Wright D, *et al.* Towards long term data quality in a large scale biometrics experiment. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10. ACM: New York, NY, USA, 2010; 565–572.
22. Altschul S, Gish W, *et al.* Basic Local Alignment Search Tool. *Journal of Molecular Biology* 1990; **3**(215):403–410.
23. Korbel J, Abyzov A, *et al.* PEMer: a computational framework with simulation-based error models for inferring genomic structural variants from massive paired-end sequencing data. *Genome Biology* 2009; **10**(2):R23. DOI: 10.1186/gb-2009-10-2-r23.
24. Zhang Z, Schwartz S, *et al.* A greedy algorithm for aligning DNA sequences. *Journal of Computational Biology* 2000; **7**:203–214.
25. Olston C, Reed B, *et al.* Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. ACM: New York, NY, USA, 2008; 1099–1110.
26. Pike R, Dorward S, *et al.* Interpreting the data: parallel analysis with Sawzall. *Scientific Programming Journal*; **13**(4):227–298.
27. Hadoop, 2007. Available from: http://hadoop.apache.org/.
28. Cascading, 2010. Available from: http://www.cascading.org/.
29. Ludäscher B, Altintas I, *et al.* Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience* 2006; **18**(10):1039–1065.
30. Oozie, 2010. Available from: http://yahoo.github.com/oozie/.
31. Meijer E, Beckman B, *et al.* LINQ: reconciling object, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06. ACM: New York, NY, USA, 2006; 706–706.
32. Chaiken R, Jenkins B, *et al.* Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment* 2008; **1**:1265–1276.
33. Yu J, Buyya R. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Record* 2005; **34**:44–49.
34. Thain D, Moretti C, *et al.* Chirp: a practical global filesystem for cluster and grid computing. *Journal of Grid Computing* 2009; **7**(1):51–72.