Weaver: Integrating Distributed Computing Abstractions into Scientific Workflows using Python

Peter Bui
University of Notre Dame
pbui@nd.edu

Li Yu
University of Notre Dame
lyu2@nd.edu

Douglas Thain
University of Notre Dame
dthain@nd.edu

ABSTRACT

Weaver is a high-level framework that enables researchers to integrate distributed computing abstractions into their scientific workflows. Rather than develop a new workflow language, we built Weaver on top of the Python programming language. As such, Weaver takes advantage of users' familiarity with Python, minimizes barriers to adoption, and allows for integration with existing software. In this paper, we introduce Weaver's programming model, which consists of datasets, functions, and abstractions that users combine to organize and specify large-scale scientific workflows. We also explain how these specifications are compiled into a directed acyclic graph used by a workflow manager that dispatches the work to a variety of distributed computing engines. To examine how Weaver is used in scientific research, we present three example applications that demonstrate Weaver's ability to integrate into existing workflows and incorporate optimized distributed computing abstraction tools.

1. INTRODUCTION

The increase in availablity 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 [2] and All-Pairs [6] that optimize specific patterns or models of computation. These new systems have been useful in enabling the development of high performance/throughput distributed scientific applications.

Unfortunately, while these abstractions have been successful in improving specific patterns of computation, they often fail to encompass large and complicated scientific workflows. This is because many computational science workflows consist of a pipeline of separate computational stages, but these tools normally focus on a single particular stage. For instance, in the case of biometrics, a normal 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.

To address this problem a few programming systems have been developed such as Swift [13], DAGMan [1], Pegasus [3] 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 [11].

These previous projects tackled the problem of specifying scientific workflows and integrating distributed computing abstractions by proposing new programming languages. The introduction of a new language, however, has significant adoption and usage challenges. Rather than developing a new language, we decided to pursue a different route by building a distributed computing workflow system named **Weaver** on top of an existing general purpose programming language, Python [9]. Developing a workflow framework that facilitates the use of distributed computing abstractions in Python provides the following advantages over constructing a new language:

1. **Familiarity:** Most scientific workflows generally consist of a set of *ad hoc* scripts that organize the various stages of a computational pipeline. Building a distributed computing workflow framework on a general purpose language such as Python takes advantage of this familiarity with scripting. Rather than force users to adapt to a new programming language, Weaver allows users to augment their existing toolset and takes advantage of their existing programming expertise.

2. **Deployment:** Since we build on top a ubiquitous programming language, deployment becomes a non-issue. Most Linux systems already include Python, which means Weaver can run on these platforms easily without much administrative time or costs, significantly lowering the barrier to adoption and distribution.

3. **Extensibility:** Using a general purpose programming language also allows users the ability to leverage existing software that is not a part of the Weaver frame-

CLADE 2010, Chicago, IL USA

work. Additionally, since the framework is constructed as a library in the native language, end users may extend the system to meet their needs and add their own extensions in a straightforward and reusable manner.

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 and describes the complete software application stack. Section 4 provides some examples of applications developed using Weaver along with some experimental results. This is followed by a short discussion of related work. Finally, we conclude the paper with possible issues to explore in the future.

2. PROGRAMMING MODEL

Weaver is a high-level framework that enables scientific researchers to construct distributed workflows in the Python programming language. This framework provides a simplified programming model composed of **datasets**, **functions**, and **abstractions**. These concepts are the fundamental building blocks of the Weaver application programming interface (API) and consist of a collection of custom Python modules, classes, and functions that end users combine and extend to define their workflows. This section introduces these components and explains how they work together to enable the incorporation of distributed computing abstractions into large-scale scientific applications.

2.1 Datasets

Most scientific computing tasks involve processing some sort of collection of 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. These datasets can take the form of a Python list, set, generator function, or any other Python object that implements the language's iteration protocol.

To aid the user in specifying datasets, Weaver provides a collection of custom *DataSet* objects that simplify the enumeration 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 is exposed to the user through the *Query* function, which will be explained shortly.

2.1.1 Files Dataset

An 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. Each object in this collection contains the location of the file, along with relevant filesystem metadata of each file such as size and timestamps.

2.1.2 SQL Dataset

In addition to files stored on a filesystem, another common source for scientific data is a SQL database. Weaver provides a simplified database abstraction interface that facilitates access to 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 based on the object record returned by a SQL query. The user may either directly map the database record to a file on disk, or the user may materialize a file on-demand 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 Query

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 [10], a popular 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.

Since the objects in the Weaver *DataSet* collections contain metadata information common to each item in the set, it is possible for the user to filter these objects based on these attributes. The *Query* function provides this selection mechanism by implementing a SQL-like query expression language which translates the queries into an appropriate form for the underlying data collection. For datasets that are actual databases, the function will form the appropriate SQL expression and use that to perform the query. In the case of datasets that are Python lists, sets, or generators, the user-specified filters will be applied to each object in the dataset to produce the appropriate subset. To use the *Query* function, the user first specifies the name of the dataset, followed by the filters to be applied.

```
# 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 Example.
This shows how to define a dataset in Weaver using the Files and the SQL dataset constructors and how to filter the collection using the Query function.

Examples of the use of the *Query* function are shown in Listing 1. In the first example, the *FilesDataSet* constructor is used to enumerate a dataset of files. The *Query* function is then used on this collection to select files that are of a particular size (> 1024 bytes). This is done using an SQL-like query syntax even though the collection itself is not a database. The second example shows the construction of a *SQLDataSet* and the formation of a subset by using the

Query function to only select records with an *EyeColor* of *Blue* or *Green*.

Including a query expression language in the Weaver framework is useful for a couple of reasons. First, it provides a unified way of constructing subsets of larger datasets. The user simply defines the whole data repository and can then apply filters to extract the interesting subgroup in a consistent and repeatable manner. Second, by implementing the expression language as a domain specific language, the user never needs to leave the Python language environment. Instead, the user always interacts with the dataset as collections of Python objects.

In summary, datasets in the Weaver programming model are simply collections of objects with a corresponding manifestation in the filesystem. To simplify and expedite the specification of these datasets, Weaver provides a library of constructors such as *FilesDataSet* and *SQLDataSet*. These Weaver *DataSets*, in conjunction with the *Query* function allow the user to quickly and easily specify large datasets and perform filtering operations to selectively group data for processing.

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**, which is a Python object that defines the interface to an executable. Like objects in a dataset, a Weaver function corresponds to a physical file on the filesystem.

As with datasets, Weaver provides a set of 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_string` and `output_string`. The first method specifies how to generate the appropriate string used to execute the task given a set of input and output files, while the second method allows the user to provide a default means of naming the output files of the function. To further simplify the definition of functions, Weaver includes a set of objects that build upon this generic *Function*.

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

Cat     = StreamFunction('cat',
                         out_suffix = 'txt')
Img2Png = SimpleFunction('convert',
                         out_suffix = 'png')
PyFcn   = PythonFunction(f)
```

Listing 2: Weaver Functions Example.
This shows a few examples of how to specify executables using StreamFunction, SimpleFunction, and PythonFunction constructors provided by Weaver.

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. Since these executables require capturing standard output in order to generate a output file, the *StreamFunction* object modifies the `command_string` 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 to the command. Weaver provides the *SimpleFunction* constructor for these types of applications which is similar to the *StreamFunction* except the `command_string` method does not capture standard output to a file since the executable uses an explicit output file.

As shown in Listing 2, both the *StreamFunction* and *SimpleFunction* constructor also 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_string* method to generate an output file name based on the executable and the input filename.

2.2.3 PythonFunction

A final *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.

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

2.3 Abstractions

The final 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 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 [12].

Each abstraction takes in a set of datasets and functions as arguments and applies those functions to the input data in a particular pattern. To enable 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*. The following is a description of four patterns of computation included in the Weaver framework and commonly found in scientific workflows.

2.3.1 Map

The *Map* abstraction is a common pattern used for work

that is naturally parallel and takes the form:

```
Map(function, inputs)
```

The abstraction takes 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. Since each function application is independent of other function executions, the individual tasks in this pattern are data parallel and thus can be executed concurrently.

2.3.2 MapReduce

MapReduce [2] is another common abstraction that is used for large data processing pipelines. This pattern usually takes the following form:

```
MapReduce(mapper, reducer, inputs)
```

In this pattern, a `mapper` function is applied to the initial set of inputs to generate a group of intermediate output files which are partitioned, sorted, and then passed to the `reducer` function for aggregation. All the tasks in the both the `mapper` and `reducer` phases exhibit data independence and therefore can be run in parallel.

2.3.3 All-Pairs

All-Pairs [6] is an abstraction that is frequently used in fields such as biometrics and data-mining. In this pattern of work each member of one dataset is compared to each member of another dataset to produce a matrix that contains the scores for each comparison. This abstraction takes the following form:

```
AllPairs(function, inputs_a, inputs_b)
```

Like the previous abstractions, the individual comparison tasks can execute independently of each other, which allows the jobs to be scheduled to run in parallel.

2.3.4 Wavefront

Wavefront [12] is an abstraction used in game theory and gene sequencing applications and takes the following form:

```
Wavefront(function, matrix)
```

This abstraction 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.

As explained, the Weaver programming model consists of datasets, functions, and abstractions. Datasets identify a set of input files to be processed, while functions defined executables that are used to process such files. Abstractions are high 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.

3. EXECUTION MODEL

To take advantage of Weaver's programming model, the user programs a workflow specification in Python that invokes the various dataset, function, and abstraction components described above. Next, the user processes the script using the Weaver compiler which produces a sandbox (directory) that contains a DAG detailing the relationships between each task, and links to the executables and input data specified by the user. The generated DAG is then used by a workflow manager that dispatches tasks in proper order to a distribute execution engine. This section examines the main components of the Weaver execution model.

3.1 Compiler

The Weaver framework is implemented in Python as a collection of libraries that implement the various components of the programming model and a compiler that processes the user-defined scripts. To construct a specification, a user simply creates a Python script that invokes the components provided in the Weaver library to describe the desired workflow and then runs the Weaver compiler on the script.

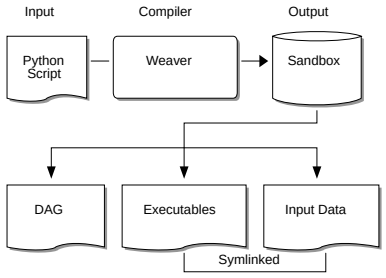

Figure 1: Weaver Compiler.
Users specify their workflow in a Python script that is compiled using Weaver to generate a sandbox containing a DAG and links to the specified executables and input data.

As shown in Figure 1, the output of the Weaver compiler is a sandbox that contains a DAG and the various input and executable files specified by the script. When the compiler begins, it creates this sandbox which serves as the location for the compiler's output and as the environment for the workflow manager. To form the DAG, the compiler evaluates the user's Python script using Python's `execfile` function. Internally, when users invoke one of the Weaver abstractions, a list of tasks in the form of `(command, inputs, outputs)` tuples is generated. During the evaluation of a user's script, the compiler tracks these tuples and at the end outputs them into a DAG. Additionally, the compiler also keeps track of the user specified input and executable files and symbolically links them (or copies if the user desires) into the sandbox so that the workflow manager can locate them later.

3.2 Makeflow

The result of the compilation phase is a sandbox environment containing a DAG and links to various files required for proper execution of the workflow. Currently, Weaver outputs Makeflow [12] DAGs which contain rules similar to those found in a normal Makefile that describe tasks in terms of the inputs and output dependencies. This information is used by Makeflow to form a directed acyclic graph of the entire pipeline, where the nodes are the data to be processed and the tasks to be executed, and the links are the relationships between the tasks and the necessary input and output files. By forming this directed graph, Makeflow can deter-

mine which tasks depend on others and schedule the work appropriately. Once the DAG is generated, users pass it to Makeflow to begin the actual execution of the workflow.

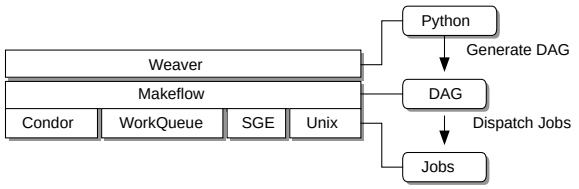

Figure 2: Weaver Software Stack.
The complete Weaver execution model is composed of three layers. The first is the Weaver framework which is used to generate a DAG. The second layer is the workflow manager processes the DAG and dispatches jobs to the third layer, the distribute execution engine.

The whole Weaver software application stack is shown in Figure 2. As a flexible DAG-based workflow manager, Makeflow provides access to multiple execution engine targets such as Condor[11], Sun Grid Engine (SGE), WorkQueue [12], and local Unix processes. Because Weaver generates Makeflow DAGs rather than direct schedules for a specific execution engine, users of the framework can take advantage of multiple execution environments. This flexibility allows users to adapt to the resources available to them without having to modify their workflow specification.

3.3 Optimized Tools

As discussed earlier, we implemented a variety of distributed computing abstractions such as *All-Pairs* in Weaver. By design, the framework uses a task tuple list as the means of implementing an abstraction's workflow pattern and relies on the workflow manager to assemble a DAG to formulate proper execution order. This allows for the specification of sophisticated workflows that are completely agnostic of the underlying execution engine. One can view the abstraction primitives provided by Weaver, then, as generic operations that would 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 data to be manifested as files on the filesystem. In workflows that involve many short running applications processing small pieces of data, this model will yield an inefficient implementation since 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 for why the generic implementations are non-optimal is that some abstractions require intimate knowledge of the underlying execution engine to be effective. For instance, the original MapReduce 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 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 specify which version of the abstraction to utilize. To use a 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.

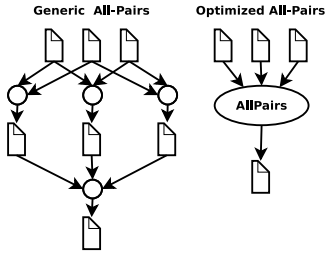

Figure 3: Generic vs. Optimized All-Pairs DAG.
The DAG on the left represents the generic All-Pairs implementation produced by Weaver, while the one on the right shows the DAG when the optimized version is used instead.

This ability to choose between a generic operation and a specialized operation is similar to the use of SIMD instructions. 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 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 optimized tool, then Weaver will forgo the generic version and instead use the optimized tool. As shown in Figure 5, a native tool greatly reduces the size of the DAG since 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 that takes advantage of existing tools makes Weaver flexible and great 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.

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, in this case 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 non-optimal generic implementation.

4. APPLICATIONS

To illustrate how Weaver is used in scientific workflows, we present three applications constructed using the framework. The first application is the canonical word count workflow, which represents many common text processing tasks. The second example demonstrates a data analysis pipeline used

to examine the results of molecular dynamics simulations. The third application is a common workflow used to perform experiments on large volumes of biometric data.

4.1 Word Count

The first example demonstrates the Weaver version of the common MapReduce word count application. The goal of this program is to process a set a files to produce a count for each word in the set. This is done by performing a MapReduce on the files: the map function splits any input string into tokens and emits them, while the reduce function accumulates the total number of tokens found for each word.

```
def wc_mapper(key, value):
    for w in value.split():
        print '%s\t%d' % (w, 1)

def wc_reducer(key, values):
    print '%s\t%d' % \
        (key, sum(int(v) for v in values))

MapReduce(mapper  = wc_mapper,
          reducer = wc_reducer,
          input   = FilesDataSet('weaver/*.py'),
          output  = 'wc.txt')
```

Listing 3: Word Count Example.
This shows a Weaver implementation of word count using the MapReduce abstraction and where the mapper and reducer functions are defined in Python.

Listing 3 contains the Weaver specification for the word count application. In this example, the map and reduce functions are implemented directly in the script as Python functions and are passed to the *MapReduce* abstraction along with the *FilesDataSet* of the Weaver source code. The final output is collected in the `wc.txt` file specified in the example. As can been seen, the specification of the entire workflow is rather compact and straightforward. Because Weaver uses Makeflow as the task manager, the MapReduce workflow can run on a variety of distributed execution engines that do not normally support such a pattern of computation.

4.2 Molecular Dynamics

Another typical use of Weaver is to construct data analysis pipelines for large volumes of experimental results. In such situations, there is usually a large repository of data that needs to be analyzed using one or more analysis programs. Normally these datasets can be processed independently from each other and thus fit into the *Map* abstraction.

```
db    = SQLDataSet('db', 'proj_100000', 'status')
files = Query(db, db.c.jobid == -1)
rmsd  = StreamFunction('rmsd')

Map(rmsd , files , output = 'rmsd.results')
```

Listing 4: Molecular Dynamics Example.
In this example, unprocessed molecular dynamics results are selected and then processed with the rmsd analysis tool using the Map abstraction.

Listing 4 shows an example of a data analysis workflow used to process biological data generated by researchers in the Folding@Home project. Periodically, this script is executed by Weaver to schedule tasks that need to be analyzed and produces an appropriate workflow. As can be seen in the code example, a database is used to keep track of which data has already been processed. The *Query* function is used to filter the database for data files that have yet to be scheduled for processing (jobid is -1). Once this selection is formed, the *Map* abstraction is used to apply the analysis tool to each of these unprocessed files and to collect the results into a single output file. After the Weaver compiler is run on this specification and generates a DAG, Makeflow can then be used to perform the actual computation.

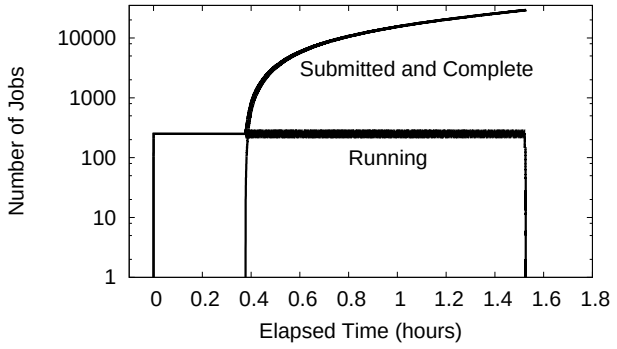

Figure 4: Molecular Dynamics Workflow Timeline.
This shows the progress of the tasks in a molecular dynamics data analysis workflow involving 25,843 work units. The complete pipeline was completed in about 90 minutes.

Figure 4 presents the execution timeline of a data analysis workflow generated by weaver using a specification similar to that in Listing 4. In this experiment, 25843 Folding@Home data units were analyzed in about 90 minutes using SGE. Normally, each work unit takes about 2.5 seconds, which means that DAG generated by Weaver was able to achieve a 12× speedup when executed on a SGE cluster composed of a mixture of 2GHz+ dual-core and quad-core machines. Beyond enabling an increase in throughput, the use of Weaver also provided the researchers with a consistent and simple means of scheduling their data analysis work. The success of the Weaver framework as demonstrated in the graph enabled the researchers to replace their set of *ad hoc* scripts in favor of Weaver, which proved to be more reliable and simpler to use and maintain.

4.3 Biometrics

Weaver is also used to conduct biometric experiments, which usually consist of multiple data processing stages as described in the introduction. First, the interested dataset must be selected or extracted from the original data repository. Next, the raw data must be transformed into a format suitable for experimentation. Finally, the experiment is performed, which in biometrics, involves comparing all members of the dataset to each other.

Listing 5 provides the Weaver implementation of such a biometrics workflow. To construct the interested dataset, the user first specifies the original data repository, in this case a biometrics database, and uses the *Query* function to select records with the desired property. 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.

Figure 5 shows the results of experiments constructed us-

```
db    = SQLDataSet('db', 'biometrics', 'irises')
nefs = Query(db,
             db.c.state == 'Enrolled',
             Or(db.c.color == 'Blue',
                db.c.color == 'Green'))
conf = SimpleFunction('convert_iris_to_template',
                      out_suffix = 'bit')
cmpf = SimpleFunction('compare_iris_templates')
bits = Map(conf, nefs)
AllPairs(cmpf, bits, bits, output = 'matrix.txt')
```

Listing 5: Weaver Biometrics Example.
In this example, images from a biometrics data repository are selected, and transformed into a specialized bit format, and then compared to each other.

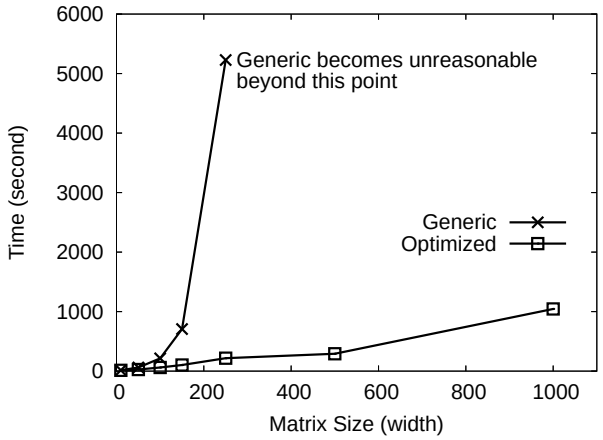

Figure 5: Generic vs. Optimized All-Pairs Timeline.
This figure shows the time to complete a biometrics pipeline using the generic All-Pairs implementation versus the optimized version. The generic implementation is considerably more expensive, because it uses files for output storage and is unable to dispatch sub-problems to multi-core processors.

ing the workflow specifications just described with the optimization options turned on and off. In these experiments, we varied the size of the workloads by using datasets consisting of 10 to 1000 images, where each file was 301 KB. Each of these image files was converted to a specialized bit format that was about 1.2 KB using the *Map* abstraction. These bit files were then compared to each other using *All-Pairs* to generate comparison matrices. For each dataset, we generated workflows with and without the optimized native implementations using Weaver's SIMD-like compiler mechanism and executed both. From the graph, it is clear that the generic version scales poorly as the number of images increases, while 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 WorkQueue 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 experiments show that while generic implementations of various abstractions are useful since they can be adapted to run on multiple distributed systems, there is a performance penalty for this generality. Weaver's programming model's requirement that input and output data must be stored as files can greatly constrain the performance of certain types of workflows as demonstrated above. In these cases, a specialized native tool can easily outperform the generic implementation because it is not constrained by the programming model. Fortunately, Weaver provides a mechanism to take advantage of these optimized abstraction implementations, allowing for specialized versions to be used when possible.

5. RELATED WORK

Because distributed computing abstractions can be quite complex and require significant effort to utilize effectively, there has been a variety of previous work aimed at providing users with simplified and efficient interfaces to these systems. Like Weaver, these high level languages provide a compact programming model where the user specifies their workflow, which is then translated into a set of tasks to be performed by a distributed execution engine.

Pig [7] and Sawzall [8] are two languages that provide a high-level interface to MapReduce [2]. Both of these languages provide a simplified programming model composed of datasets and functions that is presented as new declarative programming languages with a SQL-like flavor. Since these languages are tightly tied to the MapReduce abstraction, the user is constrained in the types of workflows they can efficiently specify.

To allow for more types of workflows, there has been research into specifying distributed workflows as directed acyclic graphs. DAGMan [1] and Pegasus [3] 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 the job specification and produces a static workflow graph.

Dryad [4] 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 [5]. This programming construct takes advantage of the LINQ programming idiom in Microsoft's .NET system to provide a native interface to MapReduce type workflows.

Swift [13] also tackles the problem of specifying diverse scientific workflows. 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 execution engine which works in conjunction with the Swift run-time system to execute the plans.

Weaver shares many of the important features present in these projects such as using a DAG-based workflow engine and providing users a simplified interface to constructing distributed workflows. Because it does not force users to define workflows in terms of graph nodes and links, Weaver is most similar to DryadLINQ and Swift in providing a high-level programming model. Unlike Swift, however, Weaver builds on top of an existing programming language, Python, rather than introduce a new one. Likewise, Weaver is not restricted to a single programming construct as in DryadLINQ, but encompasses a whole library of components that form a domain specific language distributed computing. Finally,

unlike 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.

6. FUTURE WORK

Although Weaver is in the early stages of development, it is already being used by various research groups at the University of Notre Dame. There are, of course, a variety issues left to be explored. One possible line of inquiry is to further investigate the trade-offs involved in using dedicated optimized tools rather than generic implementations of abstractions. As demonstrated in the results, a highly optimized abstraction can easily outperform the generic version produced by Weaver. It would be interesting to see if there was any reasonable means of improving the generic versions without removing the flexibility provided in separating the specification and execution layers.

Along those lines, it would also be interesting to try to expose the notion of data locality in the framework and see how that information can be utilized. 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 transfer of data can be a problem. If an active storage system were available, it would be better to only send the executables to where the data is and perform the processing there. It is not readily apparent how this sort of functionality would be exposed in Weaver.

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. While this works well for static pipelines, there are some applications that require more dynamic workflows. A common example of this is a simulation where the same tasks are repeated over and over with small changes in between each iteration. To account for this type of work, users can utilize Weaver as an interpreter with JIT (just-in-time) compilation capabilities to iteratively generate workflows at run-time rather than producing a single static workflow. This ability to use Weaver as an interpreter rather than a compiler opens up the possiblity of more dynamic abstractions and is worth investigating in the future.

7. CONCLUSION

Weaver is a high-level framework that enables scientific researchers to incorporate distributed computing abstractions into their workflows using Python. In this paper, we presented the Weaver programming model, which consists of datasets, functions, and abstractions. We also explained the Weaver execution model and discussed the relationship between the compiler, Makeflow, and the optimized tools. To demonstrate the use of the framework, we provided three applications constructed using Weaver and examine some experimental results.

Overall, Weaver is a useful and effective framework for constructing workflows that incorporate distributed computing abstractions. Its ability to integrate existing optimized tools, as well as provide generic implementations of abstractions makes it a versatile workflow organizer. Because it is built on top of an existing language, it takes advantage of users' familiarity with the language and eases adoption and deployment issues that normally hinder new systems.

Acknowledgments: We gratefully acknowledge the support of the U.S. Department of Education through a GAANN Fellowship for Peter Bui (award P200A090044). This work was also supported in part by NSF grant CNS-06-43229.

8. REFERENCES

[1] The directed acyclic graph manager. http://www.cs.wisc.edu/condor/dagman, 2002.

[2] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Operating Systems Design and Implementation*, 2004.

[3] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, B. Berriman, J. Good, A. Laity, J. Jacob, and D. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13(3), 2005.

[4] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data parallel programs from sequential building blocks. In *Proceedings of EuroSys*, March 2007.

[5] M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 987–994, New York, NY, USA, 2009. ACM.

[6] C. Moretti, J. Bulosan, D. Thain, and P. Flynn. All-Pairs: An Abstraction for Data Intensive Cloud Computing. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–11, 2008.

[7] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. 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*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[8] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming Journal*, 13(4):227–298.

[9] Python Programming Language. http://www.python.org/, 2010.

[10] SQLAlchemy. http://sqlalchemy.org/, 2010.

[11] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley, 2003.

[12] 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. *to appear in Journal of Cluster Computing*, 2010.

[13] Y. Zhao, J. Dobson, L. Moreau, I. Foster, and M. Wilde. A notation and system for expressing and executing cleanly typed workflows on messy scientific data. In *SIGMOD*, 2005.