Flexible Partitioning of Scientific Workflows Using the JX Workflow Language

Tim Shaffer
tshaffe1@nd.edu
University of Notre Dame

Nathaniel Kremer-Herman
nkremerh@nd.edu
University of Notre Dame

Douglas Thain
dthain@nd.edu
University of Notre Dame

ABSTRACT

Scientific workflows are typically expressed as a graph of logical tasks, each one representing a single program along with its input and output files. A conventional workflow manager transforms each logical task into a discrete batch job and submits it to an underlying execution system. However, converting every logical task into one batch job is not necessarily the most efficient partitioning of a workflow. By grouping multiple logical tasks into a single batch job, we may decrease data transfer, increase system utilization, and reduce the execution time of a workflow. This paper presents JX (JSON eXtended), a declarative language that can express complex workloads as an assembly of sub-graphs that can be partitioned in flexible ways. We present a case study of using JX to represent complex workflows for the Lifemapper biodiversity project. We evaluate partitioning approaches across several computing environments, including HTCondor at the University of Notre Dame, TACC Stampede2, and SDSC Comet, and show that a coarse partitioning results in faster turnaround times, reduced data transfer, and lower master utilization across all three systems.

CCS CONCEPTS

• **Software and its engineering → Specialized application languages**; *Distributed systems organizing principles*; • **Applied computing** → Computational biology; • **Computer systems organization** → Cloud computing;

KEYWORDS

scientific workflows, high throughput computing, workflow partitioning, configuration

ACM Reference format:
Tim Shaffer, Nathaniel Kremer-Herman, and Douglas Thain. 2019. Flexible Partitioning of Scientific Workflows Using the JX Workflow Language. In *Proceedings of Practice and Experience in Advanced Research Computing, Chicago, IL, USA, July 28-August 1, 2019 (PEARC '19)*, 8 pages.
https://doi.org/10.1145/3332186.3338100

1 INTRODUCTION

Workflows are a widely-used abstraction for representing simulations, data analyses, and other scientific computations. A workflow is commonly represented as a directed acyclic graph (DAG) which provides a static description of a complex pipeline of interdependent steps called tasks. A workflow management system is used to parse this complex DAG to submit each task to an execution engine once that task's dependencies are met.

The tasks that make up the logical structure of the workflow, however, do not necessarily align with the requirements of the physical infrastructure where the workflow executes. A straightforward approach to structuring a workflow can lead to unexpectedly poor performance and wasted resources. Grouping tasks into *partitions* with coarser granularity can improve the data transfer, overhead, and other performance characteristics of the workflow, independent of the results of the computation. Explicitly partitioning the tasks of the workflow allows for precise control over data movement, execution, and error handling in each logical part of the workflow.

We developed JX (JSON eXtended) as a language for expressing workflows that allows for easy manipulations to the structure and partitioning of a workflow. JX extends a JSON representation of the workflow by supporting a Python-like syntax for expressions, allowing for a concise intermediate representation that expands to a normal JSON document. Using JX, it is easy to treat a subset of the workflow as if it were an atomic job that can be dispatched as part of a higher-level application. Templates in JX can expand to complicated nested workflow structures based on parameters, allowing flexible changes to a workflow's partitioning scheme.

We explored schemes for partitioning Lifemapper, a distributed biodiversity modelling application. As a high-throughput application, Lifemapper offers significant freedom in organizing computation beyond simply following data dependency relationships. We observed that the granularity at which we distribute pieces of the workflow has a significant impact on its overall behavior.

We measured the behavior of Lifemapper under two different partition schemes and ran the application on the TACC Stampede2 [11], HTCondor at the University of Notre Dame [10], and SDSC Comet [12] execution sites. We observed substantial differences in performance in terms of execution time and data transfer between configurations when running on the same execution site. Across all three computing sites, there were similar trends in reduced data transfer and execution time with coarser workflow partitioning. There is no single rule for partitioning every workload, but expressing Lifemapper in JX provided enough flexibility to quickly match the partitioning scheme to each environment.

Our contributions are twofold: first, we demonstrate how poor choice of organization for a scientific workflow can result in poor performance on different execution sites. Second, we introduce

PEARC '19, July 28-August 1, 2019, Chicago, IL, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7227-5/19/07…$15.00
https://doi.org/10.1145/3332186.3338100

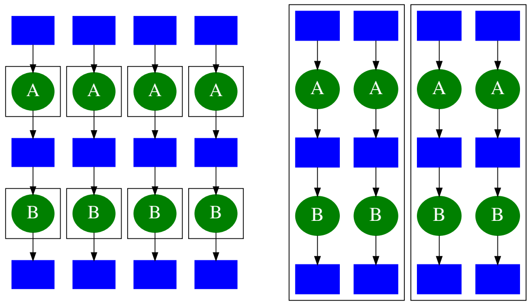


Figure 1: Fine and coarse partitions of a workflow.

On the left, a fine-grained scheme assigns each task to its own partition. On the right, coarser partitions group multiple tasks together.

JX as a language for flexibly expressing workflows and illustrate how we used JX to fit the partitioning of a scientific workflow to improve performance across several execution sites. Since the choice of partitioning scheme depends strongly on the particular application and execution site, JX offers researchers a way to easily make broad structural changes based on knowledge of the application. We demonstrate that effective partitioning can significantly reduce the amount of data transfer and wasted resources without negatively impacting the correctness or run time of the application.

2 WORKFLOW PARTITIONING

Workflow partitioning is the process of splitting a workflow graph into sub-graphs, such that each sub-graph will become a discrete batch job in the target execution system. The workflow manager must dispatch each of these jobs to a batch system in a way that respects the data and control dependencies in the original workflow graph. The most appropriate partitioning depends on many properties of the workflow graph, such as the size of data objects and the execution time of tasks, as well as the performance properties of the execution system. Partitioning workflow graphs in the general case is an active area of research [15]. Without defining an "optimal" strategy, we can approach the problem pragmatically by making some general observations about the granularity of partitioning.

In principle, a workflow system could partition the graph automatically. However, such an approach would require accurate advance information about task runtimes, file sizes, network performance, and other system details. In most production situations, these details are neither static nor known in advance, so a fully automatic approach is not practical. Moreover, graph partitioning is an NP-hard problem [3, 7], so the time costs of determining an optimal schedule might outweigh the cost of the work to be done.

Further, the problem of workflow partitioning also intersects with the problems of job placement and scheduling. As workflow partitions become more coarse, the jobs they generate require more resources, which reduces the set of execution nodes available to satisfy the job, which increases queuing time to run the job. In a similar way, performance may be affected by global system issues such as peak network capacity, utilization of the master node by other users, and batch system scheduling efficiency.

For these reasons, we do **not** seek to find an optimal partitioning scheme for arbitrary workflows. Instead, we propose a semi-manual approach in which the workflow writer indicates natural partitions in the graph by grouping related tasks together. The workflow manager can then be configured at runtime to treat each partition as an atomic job or decompose it further into individual jobs. In our experience, the end user does not often know numerical values for file sizes and job runtimes but does have some sense of which items are *big* vs *small* or *long* vs *short*, which is sufficient to perform a usable partitioning. This is often robust to changes in workflow performance with changing job parameters and new datasets. In Section 4, we give a case study of a specific application, Lifemapper, in which this semi-manual partitioning is an effective approach.

This method maps well to a hierarchical workflow implementation. A top-level workflow manager maintains the entire workflow description with the user-indicated partitions. As the workflow executes, the top-level manager dispatches either single jobs or subgraphs as jobs to the underlying batch system. If a job contains a single task, then it is executed in the ordinary way. If a job contains a complex sub-graph, then it is submitted as an invocation of a workflow manager, given only the relevant sub-graph to execute. The job is sent to the execution node, where the workflow manager is invoked to execute the sub-graph using *only* its local resources. If the sub-graph expresses concurrency, then it can be used to exploit the available resources on the execution node. When complete, only the final results of the sub-graph are returned to the top-level manager. From the batch system's perspective, the sub-graph is a single node job with internal parallelism.

To accomplish this, we must have a workflow representation that can easily express a partitionable workload and a workflow management system which is easily invoked in a hierarchical manner. When dealing with an application partitioned in this way, we expect to repeatedly run workflows that are variations on the same pattern. Thus we would like a way to express a workflow template with parameters, e.g. the number of times to split a reference file or a list of input files to process. We can then use a small set of templates to define complex workflows and quickly adapt to changes. The next section describes JX, a workflow language designed to meet this need that works with the Makeflow [1] workflow system.

3 JX: JSON EXTENDED

JX is a workflow description language based on JSON that supports variable substitution, basic operators, and list comprehensions. JX also supports structured parameters, making it easy to pass in more complicated data such as lists or rule specifications. JX is not a general purpose programming language, but rather a compact representation of nested data structures. When all input parameters are provided and structures evaluated, the result is a static JSON document representing a set of jobs describing a workflow. JX is inspired by Python syntax with the intention of making it easily accessible to a wide variety of programmers.

A motivating use case in designing JX was expressing workflow specifications in Makeflow, a workflow system for executing large, complex workflows on clusters, clouds, and grids. Makeflow was designed to use a Make-like [6] syntax to describe the rules in a workflow. This format is well-known, compact, and easy for novices to write, but it has some limitations. Traditional Make is very particular about the layout of each rule, requiring specific whitespace characters to delimit the fields. Filenames containing whitespace or unusual characters are partially supported at best. It is also difficult to add additional data or fields to a rule in a programmatic way, which is needed to handle partitioning and other workflow transformations.

We designed JX as an alternate workflow representation to be used in parallel with the traditional Make syntax. Users can write in either language, or ask Makeflow to automatically convert from traditional Make into JX. We have found that rather than rewriting their applications to use a new workflow manager or runtime library, users often prefer to start with an existing application in Make syntax and incrementally change to JX as they add parameters and factor out repeated patterns. JSON is used as the basis for JX because it is widely supported and has straightforward rules regarding quoting and character encoding. JSON can also easily express complex and nested structures and allows new fields be added as needed. A single task expressed in JSON looks like this:

```
{
  "inputs": ["japonica.csv"],
  "outputs": ["out/japonica.asc"],
  "command": "./project japonica.csv"
}
```

An entire workflow could be expressed as a sequence of plain JSON records like the above. This might be done if an external script is used to generate and emit complete workflows. To facilitate hand-written workflows, we defined several language features that could be used to express a single compact JX program that can be evaluated into plain JSON. First, JX supports variable substitutions when expanding a document. Whereas the traditional Make format uses shell-style string substitution, JX additionally supports structured values such as numbers and lists. Several common types of operators, such as arithmetic, comparison, and Boolean, are available. These operators function the same as in Python. When writing workflow rules, this allows for simple transformations based on arguments provided to the workflow specification. We can write rule patterns that are used with a set of input variables:

```
{
  "inputs": [SAMPLE + ".csv"],
  "outputs": ["out/" + SAMPLE + ".asc"],
  "command": "./project " + SAMPLE + ".csv",
}
```

If the value `"japonica"` is bound to the variable `SAMPLE`, this template expands to the previous JSON rule. It is also possible to pass in an entire list of inputs, for example, so that an external program can easily modify the connections in the workflow. This program could be a script passing JSON arguments into the workflow, or in the case of nested workflows in Lifemapper it could be a higher-level workflow communicating the details of a partition. JX parameters give a flexible way to customize a workflow specification so that the high-level workflow uses a common template for each partition. The concrete sub-workflows only need to be elaborated at runtime according to the chosen partition. If the higher-level workflow is also expressed in JX, it becomes possible to use the same values to both define the high-level workflow and to pass into each partition. This eliminates the possibility of the partitions falling out of sync with the rest of the workflow.

For some commonly encountered workflow patterns it is necessary to create a rule for each input file or to produce a range of outputs like `file.1` ...`file.n`. To quickly generate a list of items, JX supports Python-style list comprehensions. Assuming that the variable `SAMPLES` contains the list of strings `["japonica", "arboreum"]`, we can produce a pair of rules, one for each sample:

```
[{
  "inputs": [s + ".csv"],
  "outputs": ["out/" + s + ".asc"],
  "command": "./project " + s + ".csv",
} for s in SAMPLES]
```

Since the connections between tasks in a workflow (inputs and outputs) are specified as lists, this gives substantial freedom in programmatically defining the structure of a workflow. In addition, the workflow itself is primarily a list of rules, making it possible to expand a large number of rules from a single template. Rather than passing only numbers or strings, structures such as the list of inputs of a partition can be passed as arguments, with the workflow template expanding to create matching rules. As a simple example of a map-reduce task, consider a workflow that takes a set of `N` input files, `INPUT.0`, `INPUT.1`, ..., processes each, and combines the results into `INPUT.out`. For any value of `INPUT` and `N`:

```
{"define": {
  "TMP": ["out/" + INPUT + "." + i
      for i in range(N)],
}, "rules": [{
  "outputs": t,
  "inputs": basename(t),
  "command": "./proj " + basename(t),
} for t in TMP] + [{
  "outputs": [INPUT + ".out"],
  "inputs": TMP,
  "command": "./merge " + join(TMP),
}]}
```

Here `range()`, `basename()`, and `join()` are built-in functions: `range()` returns a list of numbers up to its argument (just as in Python), `basename()` strips leading directory components (just as the shell utility), and `join()` takes a list of strings and concatenates them (by default separated by spaces). The first part of this template defines the list of `N` intermediate files for the workflow as `TMP`. Then for each file in `TMP`, we add a rule to run `./proj` to create the file. We finally take the entire list `TMP` as the inputs for the reduce step, and use that same list to build the command line. This ensures that the inputs to `./merge` always match the outputs of the map rules.

Combining these features, JX allows us to treat a sub-workflow as a job. A JX template serves to define the structure of the sub-workflow subject to some parameters. We can reuse the same definition to programmatically produce a large number of rules that fit the concrete arguments to each invocation. For a two-level workflow scheme as with Lifemapper, we need only two templates: one for the low-level workflow rules within each partition and one for the high-level workflow that connects the partitions. We can pass the chosen partition into the high-level workflow which can expand a list of pieces into a potentially complicated graph of sub-workflows. The high-level workflow can then repeatedly expand the partition template and produce each sub-workflow as it is executed. If the previous JX template for an example map-reduce workflow is stored in the file `sub.jx`, then we can use a higher-level workflow to take a list of input files, split each into 100 pieces, and process each group as a sub-workflow. The number 100 is hard-coded here, but the number of splits could also be a variable to allow for changes.

```
{"rules": [{
  "inputs": [d],
  "outputs": [d + "." + i
      for i in range(100)],
  "command": "./split " + d,
} for d in DATA] + [{
  "inputs": [d + "." + i
      for i in range(100)],
  "outputs": [d + ".out"],
  command: "makeflow --jx-define N=100"
      + " --jx-define INPUT=" + d
      + " sub.jx",
} for d in DATA]}
```

The list of data files to be processed, `DATA`, could be determined based on the contents of a directory, or it could be passed in from a tool or workflow layered above this one. The first pattern defines a rule for each piece of input data that runs `./split`. A recursive invocation of Makeflow then expands `sub.jx` to fit each group of inputs. A more complete application might submit each group of input files to a batch system to carry out bulk processing on remote workers. We chose features for JX such as variable substitution and list comprehensions that, based on our experience working with scientific applications in Makeflow, capture commonly encountered features of workflows and allow for flexibility in partitioning and recursive subdivisions. Lifemapper is an example of a scientific workload where the features of JX can be put to effective use.

4 LIFEMAPPER

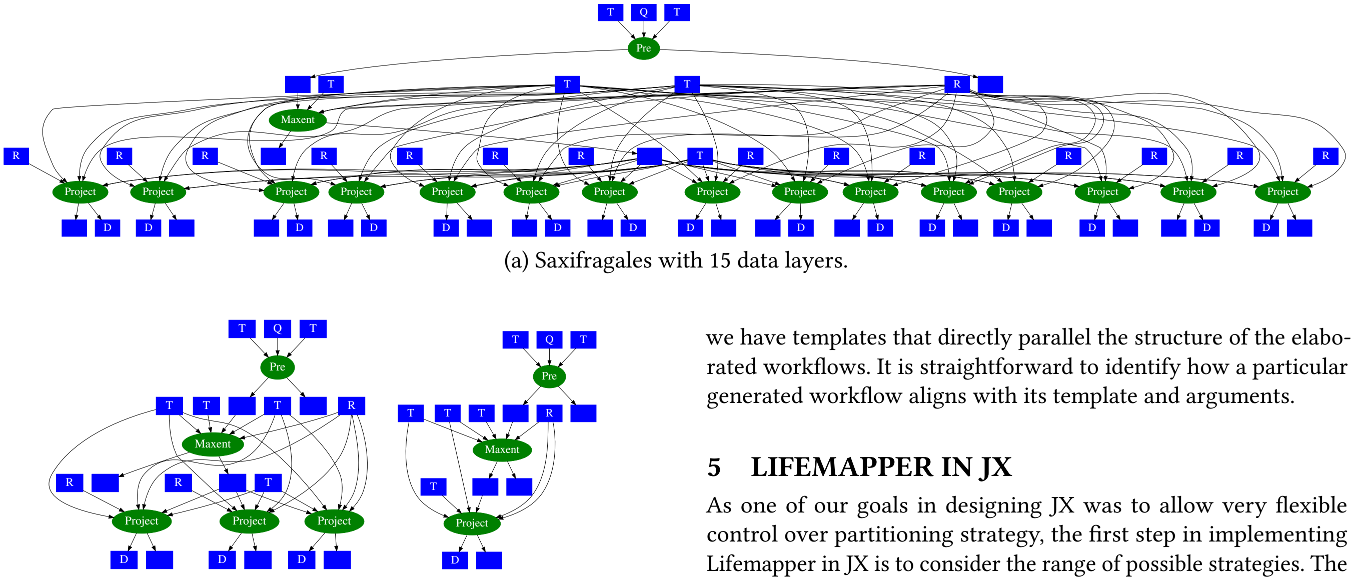
Lifemapper is a biodiversity modelling project based at the University of Kansas. Lifemapper collects geographic and temporal occurrence data for a large number of species to map biodiversity across the world. Using climate, terrain, and landcover data, the project can search for regions where a species is likely to thrive. Lifemapper also projects future species distributions under different environmental models. The Lifemapper project offers infrastructure for biodiversity researchers for running models and organizing parameters and results. Researchers can use the publicly available species occurrence data or upload their own. Researchers then choose a climate model and projected environmental conditions for the model and submit the requested task via a web interface.

The first part of Lifemapper's pipeline provides several fundamental algorithms for building species niche models. Other parts of the pipeline use these modelling plugins by calling into an internal REST API. This portion of the infrastructure uses the openModeller [4] platform for processing museum data to generate a geospatial data archive of predicted species distributions across the globe. Running a complete modelling experiment through Lifemapper's pipeline consists of assembling niche modeling experiments, dispatching them to the openModeller web service, retrieving the results, and cataloging them so that clients can later retrieve them. The infrastructure also includes a number of Python components for handling data formats and connecting components.

Lifemapper offers ample opportunity to parallelize work. Each modelling experiment is independent, so it is possible to run each computation pipeline in parallel. Shared reference data is required at the start of the computation, but there is no communication among instances during analysis. A modelling experiment consists of processing a single taxon based on some set of reference data. The structure of the pipeline for individual taxa is shown in Figure 2. Since researchers can provide their own queries and occurrence data, the web frontend needs to be able to generate workflows dynamically and trigger their execution. A single workflow would not be sufficient since the scale of the workflow varies based on user-provided queries. A simple query might require only a few tasks, while larger datasets and queries can easily produce tens of thousands of tasks. Aside from the commands to be executed, many of the workflow tasks depend on shared reference data. The Lifemapper project provides a set of reference layers to use, but researchers can also upload their own. Large queries can use gigabytes of common reference data, though an individual task might require only a subset consisting of tens to hundreds of megabytes. Thus the computational pipeline for Lifemapper must be flexible enough to handle dynamically generated workflows and must efficiently transfer significant amounts of data among workers.

Lifemapper's existing infrastructure uses Makeflow as a workflow execution engine. Each workflow is written out as a Makefile by a Python script connected to the frontend. At first, these generated scripts were finely partitioned, i.e. they included all the individual tasks associated with each query. The administrators encountered difficulty with error handling in this configuration. Some portion of the tasks in Lifemapper's pipeline can fail due to mismatched input data or intermittent problems with the query. Simply retrying the failed tasks themselves, however, only wastes execution time. With one large, finely partitioned workflow, it was difficult for the administrators to identify failing pieces of the workflow. Removing these pieces and continuing required manual intervention from the administrators. This implementation nonetheless worked for processing queries on their local compute resources.

When moving a portion of the computation to another execution site to take advantage of an XSEDE resource allocation, however, the administrators noticed poor performance and limited utilization of the worker pool. Larger queries were causing resource exhaustion on the master node, while making poor use of the available compute



(a) Saxifragales with 15 data layers.

(b) Saxifragales with 3 data layers.

(c) *Heuchera* with 1 data layer.

Figure 2: Lifemapper queries at three different scales. *Here boxes represent files with arrows connecting to programs, represented as ovals.* ***Q*** *denotes query data,* ***T*** *denotes tools such as Python scripts and Java JAR files,* ***R*** *denotes common reference data, and* ***D*** *denotes output data. The lower workflow (c) shows a small query against a single data layer for the Heuchera dataset. The next workflow (b) shows a larger query against the Saxifragales dataset with three data layers. At the top (a), we duplicated layers to create a twelve-layered query against Saxifragales.*

nodes. Thus the administrators had to restructure their Lifemapper implementation to work with this new computing site. This involved modifying the setup script to partition the workflow into multiple pieces and generate a large number of workflow definition files. The administrators observed marked improved performance with the coarser partitions. This also aided in error recovery, since a failing partition could be removed and retried a a whole.

In adapting to multiple execution sites and varying the structure of the workflow, however, the disadvantages of using generator scripts became apparent. Either a custom generator script is required for each site (and these scripts must be kept in sync), or a single, more complex script can generate workflows for multiple sites. Adding customizable partitioning and sub-workflows further adds to the complexity of the script(s). For large-scale scientific workloads, the size of generated workflows is often too large to manually read and validate. Thus debugging tends to start with running the workflow and observing failures. With a growing number of features and variations to generate, it becomes more difficult to pinpoint the source of such failures and how they were produced by the script. The contents of a generator script often bear little resemblance to the generated workflows, so debugging or modifying them requires a significant investment of time to trace potentially complex logical flows. By expressing workflows in JX, however, we have templates that directly parallel the structure of the elaborated workflows. It is straightforward to identify how a particular generated workflow aligns with its template and arguments.

5 LIFEMAPPER IN JX

As one of our goals in designing JX was to allow very flexible control over partitioning strategy, the first step in implementing Lifemapper in JX is to consider the range of possible strategies. The most fine-grained partitioning approach is to break every single node of the workflow into its own sub-graph and submit that as a single batch job, as depicted on the left of Figure 1. This is a conservative configuration and is the normal mode of operating for most production workflow managers, such as Makeflow [1], Pegasus [5], and Swift [14], which must seek to correctly execute arbitrary workflows, often without detailed advance information about each job. This approach has several advantages: it maximizes the concurrency of jobs submitted to the target batch system, and it minimizes the cost of failure, should a single job fail and roll back to the beginning. On the other hand, it maximizes the amount of data transfer necessary, because each job must have its data transferred in and out of the execution node.

A slightly more coarse approach would be to group a small number of related tasks into one sub-graph, as shown on the right of Figure 1. Viewing this partition itself as a single batch job, the workflow manager only needs to manage the inputs and outputs of the sub-graph, with the tasks and intermediate files comprising this sub-workflow handled locally on a worker. This approach would reduce the concurrency of jobs submitted to the target batch system, however each job would effectively become a mini-workflow with its own internal concurrency that could be exploited on the execution node. The total amount of data transfer could be reduced if tasks in the sub-graph shared common input files or if some files internal to the sub-graph were not needed outside of it.

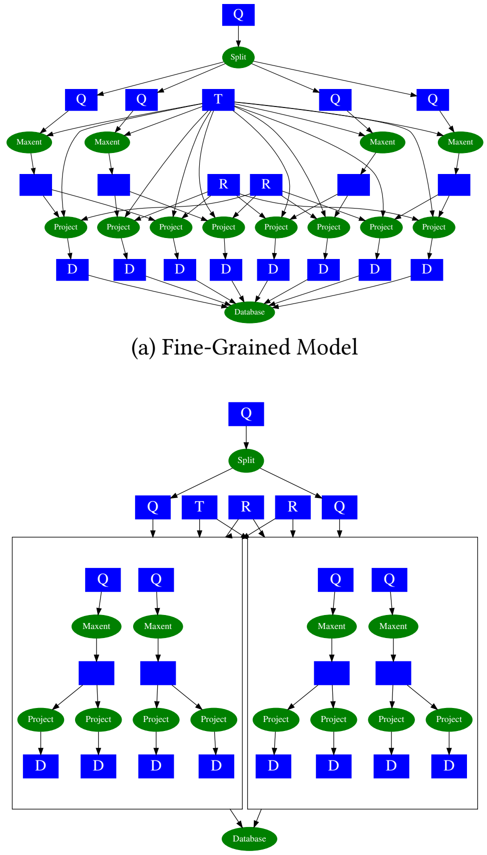
Taking this idea to its limiting case, we might consider a single large partition containing the entire workflow in one batch job. A single large partition results in the absolute minimum amount of data transfer, while minimizing the concurrency of jobs (in this case only a single job) submitted to the batch system. Of course, the single job would be very large, highly concurrent, and very sensitive to node failures. While this may sound extreme, it may in fact be a viable strategy for a workflow with large amounts of internal data transfer if it were assigned to a single large multi-core machine that is expected to be available for the entire workflow.

For a given workflow run, *some* partitioning strategy must be chosen, either manually by the end user or automatically by the workflow system. Most production workflow systems take the conservative approach of fine-grained partitioning, assuming that it is better to ensure forward progress rather than seeking better performance via coarser jobs that may fail to run at all. In a manual approach, the end user could provide grouping information within or alongside the workflow description, allowing the workflow manager to make partitions without changes to the task definitions.

We chose to compare two partitioning strategies for Lifemapper: a fine-grained strategy following the original implementation with each task submitted as a distinct batch job, and a coarse-grained strategy with multiple taxa processed together. Rather than using the setup script to write out multiple workflow definitions, we used two JX templates for all configurations. One specifies the structure for processing an individual taxon. Figure 2 shows several possible structures for a single taxon workflow. The number of tasks in each depends on the query, with more complicated queries producing additional intermediate tasks and data. The partitions of the workflow consist of one or more taxon grouped together. The other template defines the high-level structure of the workflow. Each partition in the workflow has a number of input and output files to be transferred to and from the worker. The high-level template defines the per-partition tasks and passes the list of taxa to the low-level template to generate the workflows on the workers.

Using JX to express both levels of the workflow allows us to avoid the large number of intermediate Makefiles produced by scripts. Instead, we pass structured parameters (a list of taxa) to the high-level workflow template and then expand the low-level template to fit each partition. These JX templates were initially written to work with the Coarse-Grained workflow configuration with multiple partitions, but also support the original Fine-Grained configuration consisting of unpartitioned individual tasks. In this case, we simply place each task of the workflow into its own partition.

Figure 3 compares these workflow configurations. For the low-level workflow definition, we define a pattern for each phase of the per-taxon workflow, e.g. pre-processing, maximum entropy model construction, occurrence projection. The query information (passed in from the high-level workflow) determines the structure of the per-taxon workflow and the connections between phases. The template for the high-level workflow is simpler as it only transforms a list of partitions into sub-workflow invocations. The query details do not affect the high-level structure of the workflow, so this template passes the query down to the taxon template. Using these workflow templates, we can freely adjust the partitioning scheme without modifying the workflow definition. Using JX templates helps to disentangle the design of the pipeline from details of the execution site. We were able to use the same workflow templates to take advantage of several different compute environments by simply adjusting the partitioning parameters. The templates used here do not support every conceivable partitioning scheme. For example, they do not break individual taxa into fragments split across partitions. Choice of partitions would depend on the structure of the particular application. When working with Lifemapper's computational pipeline, the partitioning options we implemented are sufficient to demonstrate the decisions and trade-offs between partitioning schemes. Nonetheless, JX provides enough flexibility to perform more complicated partitions. For our evaluation, we consider only Lifemapper partitioned at the granularity of taxa.



(a) Fine-Grained Model

(b) Coarse-Grained Model

Figure 3: Partitioning Schemes for Lifemapper.

The labels here have the same meaning as in Figure 2. With the Fine-Grained configuration in (a), there is no additional structure within the workflow beyond data dependencies. With the Coarse-Grained configuration in (b), the taxa are arranged into two partitions. Each partition becomes a task in the high level workflow.

In addition to limitations that would prevent the workflow from running, researchers may need to take other factors into considerations when structuring a workflow. Specific sites often require varying resource or queue specifications to work with the batch scheduler or local resources. When moving between the sites evaluated here, JX gave us a way to easily patch these localized changes into a common workflow template shared across all sites. In the case of Lifemapper, it is also important to recover from failures due to invalid input data. This was one of Lifemapper's researchers' initial motivations for breaking a workflow into sub-components. In the Fine-Grained configuration, it is difficult to isolate failed tasks and recover. While the amount of lost work in each failure was smaller in the Fine-Grained configuration, grouping taxa in the Coarse-Grained configuration made it more convenient to repair or discard only the failing pieces. For the particular case of Lifemapper, where failures are an infrequent but regular occurrence that may require manual intervention, researchers preferred to waste

Execution	Batch		Per-node	
Site	Scheduler	Cores	Clock Speed	RAM
Comet	SLURM	24	2.5 GHz	128 GB
Notre Dame	HTCondor	Varies	Varies	Varies
Stampede2	SLURM	68	1.4 GHz	96 GB

Figure 4: Comparison of Execution Sites.

Note that Stampede2 compute nodes are equipped with Intel Xeon Phi 7250 ("Knights Landing") CPUs, with 4 hardware threads per core for a total of 272 hardware threads. Also note HTCondor is a cycle-scavenging batch system, relying on unused compute cycles on commodity hardware in addition to dedicated compute nodes.

somewhat more computational resources so that the majority of sub-workflows completing successfully could finish quickly and return results to end users. Different applications and infrastructures might benefit from other strategies, so it is valuable to give users flexibility in defining workflow structure.

6 EVALUATION

For our evaluation, the Lifemapper project provided us with two sample data sets. Figure 2 shows the structure of a workflow to process a single taxon from these data sets. The smaller dataset, consisting of samples related to the genus *Heuchera*, consists of 51 taxa and generates projections for a single data layer. This dataset is small enough to run on a single computer. The larger dataset consists of samples from the order Saxifragales and contains 838 taxa with queries against three data layers. This bigger dataset is better suited to running on multiple workers in parallel. Larger production queries may include more data layers to be processed for each taxon, shown at the top of Figure 2.

Using the larger of the two data sets, we measured the runtime characteristics of the two workflow configurations on the execution sites listed in Figure 4. Between sites, there are significant differences in types of worker nodes, communication speeds, and system organization. Using Makeflow's logs, we collected detailed data about the total workflow runtime and amount of data transferred over the course of the workflow. The Fine-Grained configuration is the normal mode of operation for workflow managers such as Makeflow, so we initially supposed that this configuration would perform the best overall. The original implementation of Lifemapper also used a Fine-Grained configuration. We wanted to compare this straightforward structure to a Coarse-Grained configuration that partitions the workflow into multiple sub-workflows to run on worker nodes. We suspected that this configuration would result in less data transfer compared to the Fine-Grained configuration.

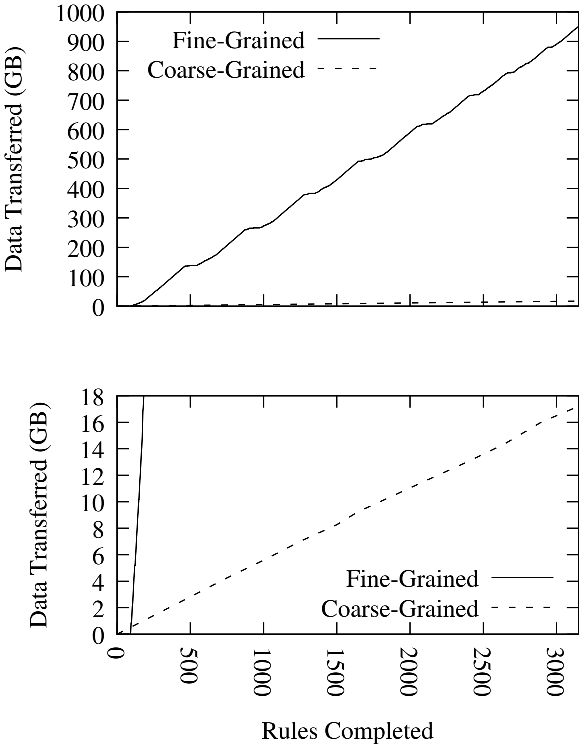


Figure 5: Data Transferred by HTCondor.

Above, the data transfer in the Fine-Grained configuration is far greater than in Coarse configuration; Coarse-Grained is difficult to distinguish from the x-axis. Below, the same graph is zoomed in to show the Coarse-Grained configuration more clearly. Note the difference in y-axis scales between the two.

Execution	Fine-Grained	Coarse-Grained
Site	Configuration	Configuration
Comet	162 min.	116 min.
Notre Dame	86 min.	9.8 min.
Stampede2	171 min.	8.7 min.

Figure 6: Lifemapper Runtime Differences.

Figure 5 shows the amount of data transferred by HTCondor versus the number of individual tasks completed over the course of the workflow execution. For each distinct worker node, the Makeflow requests the transfer a number of necessary components such as scripts, Java programs, and reference data. The other execution sites assume a shared file system, which hides this process of data transfer from the user. In the Coarse-Grained case, many tasks run at once on the same node, sharing input data and only transferring the finished outputs back to the master. In the Fine-Grained case, however, each task is handled independently, so all input data and intermediate files must be transferred to and from the master. As Figure 5 shows, reorganizing the workflow into a Coarse-Grained configuration substantially decreases the amount of data transfer required to compute the same results.

Figure 6 gives the running time for the workflow under both configurations at each site. Across all sites, the Coarse-Grained configuration showed better performance, Notre Dame and Stampede2 showed a more than tenfold increase in performance due to reorganizing the workflow. On Comet, the improvement was more modest, but still significant. We attribute this to variations in system utilization and queue scheduling when running the workflows. We do not present these results as a comprehensive performance analysis; instead, they serve to illustrate that straightforward workflow transformations based on intuitive *big* and *small* pieces can achieve significant improvements in data transfer and performance.

7 RELATED WORK

An important consideration for researchers is whether JX or another workflow language is a good fit for a particular application. There is a large number of existing workflow languages, which vary in expressivity, ease of use, and adoption by the research community. While we cannot hope to discuss every alternative, we can draw comparisons to several popular languages and systems.

Galaxy [8] operates on static workflow "templates" with the runtime generating concrete steps during execution. Galaxy provides users with a graphical interface for combining command line tools to build workflows, which is helpful for researchers with little programming experience. The Common Workflow Language (CWL) [2] is a workflow specification standard for describing command line tools and workflows in a portable manner. It supports similar uses to Galaxy, but is specified as a textual language rather than through a graphical interface. With both Galaxy and CWL, administrators can provide pre-configured tools to use in workflows. Thus researchers may be encouraged to use the workflow tool adopted by their site. Another alternative is Cromwell [13] workflow manager, which operates on the Workflow Description Language (WDL) and CWL. WDL is a more complete imperative language than expression-oriented JX, and is popular in the genomics and bioinformatics communities. WDL and Cromwell have strong support for running genomics workloads in the cloud. Snakemake [9] also supports a Make-like style, but is much more tightly integrated with Python. Thus for applications already written in Python, it may be easier to use Snakemake.

8 CONCLUSIONS

Based on our evaluation with Lifemapper, we demonstrated how the organization of a scientific workflow can affect runtime performance. Despite computing the same results, poor choice of intermediate workflow structure can result in degraded performance. We introduced JX as a language for flexibly expressing workflows that allowed us to fit the partitioning of a scientific workflow to several execution sites. Using JX, it is possible to quickly make broad structural changes to a workflow based on common templates. Our implementation of JX is distributed under the GPLv2 as part of CCTools[1], which also includes Makeflow. A summary of the syntax and workflow representations for JX is also available[2].

While JX can aid in generating workflows according to a partitioning scheme, it does not address the issue of choosing parameters to fit an execution site and workflow. This kind of decision requires the researcher to have some knowledge of the application structure. In general, it is also necessary to run the application on a given site and measure the performance to get an estimate of the best partitioning parameters. We do not make an effort to find optimal partitioning schemes for arbitrary applications. JX offers researchers a way to quickly and flexibly adjust the parameters of a workflow partitioning to fit a site based on knowledge of the application structure and performance measurements.

ACKNOWLEDGMENTS

We would like to thank CJ Grady, Senior Research Software Engineer at the University of Kansas. This work was supported by National Science Foundation grant ACI-1642409 and also used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by NSF grant ACI-1548562.

REFERENCES

[1] Michael Albrecht, Patrick Donnelly, Peter Bui, and Douglas Thain. 2012. Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies (SWEET '12)*. ACM, New York, NY, USA, Article 1, 13 pages. https://doi.org/10.1145/2443416.2443417

[2] Peter Amstutz, Michael R. Crusoe, Nebojša Tijanić, Brad Chapman, John Chilton, Michael Heuer, Andrey Kartashov, Dan Leehr, Hervé Ménager, Maya Nedeljkovich, Matt Scales, Stian Soiland-Reyes, and Luka Stojanovic. 2016. Common Workflow Language, v1.0. (7 2016). https://doi.org/10.6084/m9.figshare.3115156.v2

[3] V. Chaudhary and J. K. Aggarwal. 1993. A generalized scheme for mapping parallel algorithms. *IEEE Transactions on Parallel and Distributed Systems* 4, 3 (Mar 1993), 328–346. https://doi.org/10.1109/71.210815

[4] Mauro Enrique de Souza Muñoz, Renato De Giovanni, Marinez Ferreira de Siqueira, Tim Sutton, Peter Brewer, Ricardo Scachetti Pereira, Dora Ann Lange Canhos, and Vanderlei Perez Canhos. 2011. openModeller: a generic approach to species' potential distribution modelling. *GeoInformatica* 15, 1 (01 Jan 2011), 111–135. https://doi.org/10.1007/s10707-009-0090-7

[5] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, Bruce Berriman, John Good, Anastasia Laity, Joseph Jacob, and Daniel Katz. 2005. Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming Journal* 13, 3 (2005), 219–237.

[6] Stuart I. Feldman. 1979. Make — a program for maintaining computer programs. *Software: Practice and Experience* 9, 4 (1979), 255–265. https://doi.org/10.1002/spe.4380090402

[7] Michael R. Garey and David S. Johnson. 1979. *Computers and intractability.* W. H. Freeman and Co., San Francisco, Calif., New York. x+338 pages. A guide to the theory of NP-completeness, A Series of Books in the Mathematical Sciences.

[8] Belinda Giardine, Cathy Riemer, Ross C Hardison, Richard Burhans, Laura Elnitski, Prachi Shah, Yi Zhang, Daniel Blankenberg, Istvan Albert, James Taylor, Webb C Miller, W James Kent, and Anton Nekrutenko. 2005. Galaxy: a platform for interactive large-scale genome analysis. *Genome research* 15, 10 (2005), 1451–1455.

[9] Johannes Köster and Sven Rahmann. 2012. Snakemake–a scalable bioinformatics workflow engine. *Bioinformatics* 28, 19 (08 2012), 2520–2522. https://doi.org/10.1093/bioinformatics/bts480

[10] M. J. Litzkow, M. Livny, and M. W. Mutka. 1988. Condor-a hunter of idle workstations. In *[1988] Proceedings. The 8th International Conference on Distributed.* IEEE Press, Piscataway, NJ, USA, 104–111. https://doi.org/10.1109/DCS.1988.12507

[11] Dan Stanzione, Bill Barth, Niall Gaffney, Kelly Gaither, Chris Hempel, Tommy Minyard, S. Mehringer, Eric Wernert, H. Tufo, D. Panda, and P. Teller. 2017. Stampede 2: The Evolution of an XSEDE Supercomputer. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact (PEARC17)*. ACM, New York, NY, USA, Article 15, 8 pages. https://doi.org/10.1145/3093338.3093385

[12] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkins-Diehr. 2014. XSEDE: Accelerating Scientific Discovery. *Computing in Science Engineering* 16, 5 (Sept 2014), 62–74. https://doi.org/10.1109/MCSE.2014.80

[13] Kate Voss, Jeff Gentry, and Geraldine Van der Auwera. 2017. Full-stack genomics pipelining with GATK4 + WDL + Cromwell. Poster [version 1; not peer reviewed]. (8 2017). Presented at the 18th Annual Bioinformatics Open Source Conference (BOSC 2017).

[14] Michael Wilde, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz, and Ian Foster. 2011. Swift: A language for distributed parallel scripting. *Parallel Comput.* 37, 9 (2011), 633 – 652. https://doi.org/10.1016/j.parco.2011.05.005 Emerging Programming Paradigms for Large-Scale Scientific Computing.

[15] Chen Wu, Andreas Wicenec, and Rodrigo Tobar. 2018. Partitioning SKA Dataflows for Optimal Graph Execution. In *Proceedings of the 9th Workshop on Scientific Cloud Computing (ScienceCloud'18)*. ACM, New York, NY, USA, Article 6, 6 pages. https://doi.org/10.1145/3217880.3217886

[1] https://github.com/cooperative-computing-lab/cctools

[2] http://ccl.cse.nd.edu/software/manuals/jx-quick.html