Scaling Up Bioinformatics Workflows with Dynamic Job Expansion: A Case Study Using Galaxy and Makeflow

Nicholas Hazekamp, Joseph Sarro, Olivia Choudhury, Sandra Gesing, Scott Emrich, and Douglas Thain
Department of Computer Science and Engineering, University of Notre Dame
Notre Dame, Indiana, United States of America
Email: {nhazekam, jsarro, ochoudhu, sgesing, semrich, dthain}@nd.edu

Abstract*—Logical workflow management systems provide a user-friendly portal through which data can be processed using a sequence of standard tools. These logical workflows are a natural way to express the high level intent of the user, and to share the structure and the results with other users. However, logical workflows are not necessarily suited to expressing parallelism for very large runs. As the amount of data is scaled up, the run time of each node in the logical workflow may become extreme. We propose a technique of job expansion to solve this problem. When job expansion is applied to a logical workflow, each node in the workflow is itself expanded into a large performance workflow that may consist of hundreds to thousands of tasks that can be executed in parallel, thus enabling high concurrency and scalability. From the user's perspective, nothing has changed and the logical workflow remains in its original form. To demonstrate this technique, we have applied job expansion to a selection of bioinformatics applications running in the Galaxy workflow management system. Each job in the workflow is expanded into a highly parallel workflow executed using Makeflow, which is well suited to express high levels of parallelism. Work Queue is then utilized for execution because of its ability to quickly dispatch tasks and cache files for later reuse. After applying job expansion, we improve the execution time of BWA 18X and GATK 402X, with a total speedup of 61.5X on the workflow. We also take a look at the systems behavior since its launch to analyze its effectiveness.

***Index Terms*—Workflow; Galaxy; Bioinformatics; Makeflow;**

I. Introduction

Workflow management systems (WMS) such as Galaxy [1]–[3], Taverna [4], and Kepler [5] allow scientists to easily combine multiple tools into a coherent whole. While a WMS is designed to manage the connected steps, it may also offer a graphical user interface with more intuitive features for job configuration and basic data management. These traits make the systems versatile, and allow integrating tools from different sources to process data in a consistent, reproducible manner across users and environments. We refer to this as a **logical workflow**.

Many WMS assume that each tool is the smallest granularity of computation. As a result, these systems are limited when inputs become larger as there is either limited or no mechanism to further increase parallelism. Scaling in a WMS is not related to resources only, but also the size and complexity of the jobs executing. These complexities come in many different forms from the number of concurrent jobs, interjob dependencies, and the complexity of combining different topical programs. Enabling additional parallelism must then come as either a change to the WMS's job handling, or an extension of the job structure already in place. However, such extensions are invariably tied to the local execution environment, which limits portability and reproducibility.

We propose **dynamic job expansion** as a solution to this problem. In this technique, each job of a logical workflow is expanded at runtime into a **performance workfow** in which the majority of the work is done in parallel, accelerating the performance of the original process. The purpose is to create tools that are indistinguishable from sequential tools, while supporting greater parallelism. Abstracting this parallelism from the user allows dynamic job expansion to be utilized by WMS without user involvement, and enables the parallelization to be specialized to the local environment. Ideally this method could apply to any computation that consists of data-independent sub-groups, which we find to be prevalent in bioinformatics.

For many applications, dynamic job expansion can be implemented with the split-process-join computing pattern. First, the single job is processed by a **job expander** which writes out a performance workflow corresponding to a single job in the logical workflow. The performance workflow then partitions the inputs to prepare for parallel execution, dispatches parallel tasks to a remote cloud or grid, and then joins the results into a format identical to that of the original tool. This allows the user to run the application faster, getting the same results, with no knowledge of the difference in execution.

We have implemented dynamic job expansion using Galaxy as a logical workflow manager, Makeflow [6] as the performance workflow manager, and Work Queue [7] as a remote execution system. Galaxy was used because of its large audience, bioinformatic focus, and GUI workflow construction. Makeflow was utilized because it provides a workflow expression language that simple and easily scripted. Makeflow also provides the ability to directly submit to a number of execution systems. Combining these tools required careful attention to the semantics of dependency management, remote execution, and garbage collection, in order to yield a stable and consistent system. We demonstrate the combined effect of job expansion while running a four-job logical workflow in which the two most time consuming jobs (BWA [8] and GATK [9]) are converted into performance workflows consisting of hundreds of tasks. This resulted in a speedup of 18X and 402X on BWA and GATK individually, and an overall speedup of 61.5X for the entire workflow.

To look at the behavior in a non-ideal setting, we analyzed data logged on our catalog server. This allowed us to see that while the process does reduce the execution time of these tools, there is room for improvement in bring the acheived concurrency closer to the potential concurrency. Despite the difference, we discuss several causes such as workflow makeup and system congestion.

II. Background

In this section we will give a brief introduction to each of the following frameworks, as well as reasons for their selection. Though these were selected, it would be possible to replace these techonologies with other technologies whose benefits are more suited to the application in mind.

A. Galaxy

Galaxy is a web portal that was created to provide biologists with ready access to a number of bioinformatic tools. It creates an environment where the user only supplies inputs and selects the tool to perform computation. This abstraction provides biologists with a generic analysis framework without the need for specifying resources. The administrator of a Galaxy instance configures the underlying infrastructure and provides the tools available to the users. The portal records how and when the tools are run, so that reproduction of an experiment is consistent and easy. Each tool consists of a predefined web page that serves as a launch for the tool, as well as scripts in the background to provide the correct setup.

Galaxy also provides the user with a means of creating DAG-based workflows. Workflows use the data dependencies between tools to determine order, and provide a means of creating a logical flow for processing inputs in a consistent way. Workflows created may be configured with predefined parameter values and can be shared so that analysis can be reproduced easily by numerous parties.

The portal creates an environment through which users can analyze data without the hassle of data and resource management. Galaxy's abstraction makes using tools quick and easy, while offering a platform to create meaningful results. It also provides a means of sharing results and the process required to achieve them. Work can then be verified and analyzed by many people at once, allowing for easy collaboration.

B. Makeflow

Makeflow is a system built to create and run performance workflows, using a syntax that is very similar to that of classic Make. Each rule in the performance workflow must explicitly state input and output dependencies, along with the command to run. Makeflow can dispatch jobs to a variety of execution systems, including Condor, SGE, and Work Queue.

The simple syntax supports DAG-structured workflows, which are ideal for transformation of input to output over a number of predefined steps. This syntax also makes it easy to write or have a script write rules for a workflow based on the inputs. This allows for the dynamic creation of performance workflows based on the provided input, and a means to execute them. Makeflow is great for running workflows on a distributed platform as Makeflow supports several different execution engines. For improved flexibility, Makeflow assumes there is no shared file system and provides the execution engines with the information about file dependencies. These qualities make Makeflow a good intermediate platform between Galaxy and the execution.

Since many workflows are structured as a series of defined tasks that can be mapped accross the input, scripts can readily describe the workflow as a Makeflow. This architecture also performs well as a lightweight workflow manager that can express resources needed by tasks and schedule when these requirements are met. The scheduling algorithm is limited, requiring the user to manage scheduling by how the performance workflow description is written. This design also allows workflows to be submitted to batch systems, but is restricted by the time required for tasks to be scheduled, inputs transferred, and execution on the batch platform.

C. Work Queue

Work Queue is a lightweight master-worker platform. Workers consist of a process that is started at an execution site and communicates with the master process to retrieve, execute, and return tasks. While preparing the task, the required dependencies described by Makeflow are staged and a sandbox is established. To utilize a worker, the site runs the worker, allowing for workers to be created on any supported platform or through a batch system. Work Queue provides several benefits that help performance workflows. Workers are processes that persist outside of the execution of single tasks. This allows the master to cache files on the worker and reuse them to limit multiple transfers to a single worker. Caching helps to limit the execution time as well as the number of workers needed to impact performance. This benefit can be extended by utilizing "multi-slot" workers on multi-core machines. If a task is labeled with resource needs and the worker is larger than the task's requirements, multiple tasks can be scheduled on the same worker. When a worker performs several tasks, they share a cache that limits transfers and total disk usage.

Worker persistence also enables workers to be given tasks as soon as they are available, without the overhead of waiting for the task to be scheduled through the batch system. Sidestepping the batch scheduler allows short tasks to be rapidly scheduled, with of a less penalty. The uninstantiated workers still need to go through the scheduling process, but once at an execution site they can be utilized. Further, a Work Queue pool can be created that scales active workers up and down as need arises. If more workers are needed, the pool will submit them to the specified resource, and lets them time out when more exist than are needed. Work Queue's design benefits performance workflows in several ways, and has a means of managing workers dynamically. These allow for applications such as Galaxy to create and dispatch tasks and have a facility that scales to handle them.

III. Dynamic Job Expansion

Dynamic job expansion is the run-time transformation of a single job in a **logical workflow** into a **performance workflow**. The resulting performance workflow must be logically indistinguishable from the original job by accepting the same input files and generating the same output files, while hiding the complexity from the user. Figure 1 gives an overview of dynamic job expansion.

For each type of logical job to be expanded, we must write a **job expansion tool**. This is a command-line tool that is invoked in an identical manner to the underlying application. Instead of running the application directly, it writes out the desired performance workflow, invokes the performance workflow manager, and waits for it to complete. From the perspective of the logical workflow manager, the job expansion tool *is* the job to be run.

Naturally, job expansion must be specialized to a given application and must take advantage of details of the application's structure and performance. For many applications, a split-process-join pattern is effective. The initial step of the generated performance workflow evaluates the size of the input and split the inputs into appropriately sized pieces. Then, the primary application runs on each split of the input, generating multiple outputs. The final step merges the results into a single output file. In simple cases, this might be concatenation of the outputs, while in more complex cases, it could require recomputing statistical results. The more complex cases rely on knowledge of properly dividing the work, possibly in stages, to maintain equivilance with the original tool.

Using Galaxy, Makeflow, and Work Queue, the process works as follows. For each tool in the logical workflow, Galaxy assigns the job an identifier, specifies input locations, and generates output filehandles. Galaxy creates a working directory for the job and job execution is moved to the directory where the job expansion tool can create the performance workflow. The Galaxy wrapper script links inputs to the directory and copies down the necessary execution files. The job expansion tool writes a performance makeflow based on the input's characteristics.

After the performance workflow is created, the job expander invokes Makeflow with Work Queue as its execution environment. Makeflow verifies that the structure of the performance workflow is correct, and confirms the presence of required input files. The split and join processes, which require most of the existing files, run locally to limit the data transfer. The remaining tasks utilize a subset of the files and are run in parallel. Makeflow sends each task through the Work Queue master to workers as required files become available.

The Work Queue master communicates with workers to send tasks and inputs. When the task is ready at the worker, the process is executed in a task sandbox. The completed task returns the output to the master. Having verified that the task produced a successful return value, the output is collected and returned. Makeflow continues to submit and collect tasks until all the tasks have been completed.

Upon completing the tasks, Makeflow joins the result and the wrapper copies outputs to the specified filehandles. The Galaxy wrapper script completes and Galaxy verifies that the output files have been created. After the verification, Galaxy changes the state of the job to either successful or failed. The results are then delivered to the user, and the Galaxy job is complete.

Dynamic job expansion benefits from several characteristics of this process.

- **Hidden Complexity** Using an expansion tool to write and run a performance workflow alleviates the user need to understand the interprocess complexities and expand within the logical workflow. This allows lay users to quickly pick up a tool and use it interchangably with the original tool, without needing to know the background process.
- **Environment Aware Decisions** Expanding the workflow at the execution environment provides several benefits. Inputs are defined and can be used to make intelligent partitioning decisions [10]. Intermediate files are managed locally and do not fill the data store and history. Processes utilizing many inputs, such as split and join, can be run locally to prevent large amounts of data traffic.
- **Flexible Execution Resources** The ability to utilize resources that are not primarily dedicated provides a flexibility in scaling. This flexibility also provides a means of users coming to a portal with their own resources.

IV. Example Application

To demonstrate job expansion, we show a logical workflow that transforms general genomic data to aligned genotyped data. The working data is a query of 32GB of reads against a reference dataset of 36MB, which are the reference loci of Red Oak. This is done using BWA alignment, SAMtools sort, Picard AddOrReplaceReadGroups, and GATK HaplotypeCaller. When run on the working data, the BWA alignment and GATK genotyping steps take the longest to execute: 19 hours and 12 days, respectively. These two steps were used to demonstrate dynamic job expansion, and remaining intermediate steps were left as sequential jobs.

Figure 2 details how this particular logical workflow was expanded at runtime. The expansion strategy is slightly different for each of the two tools:

BWA employs the Burrows Wheeler Transform algorithm to align genome queries. BWA is a light-weight alignment tool that supports paired-end mapping, gapped alignment, and various file formats like ILLUMINA [11] and ABI SOLiD [12]. The output format is SAM (Sequence Alignment Map), which can be analyzed using a number of tools such as GATK and the SAMtools package [13].

In previous work [14] we observed that the runtime of BWA is roughly proportional to the product of the reference and

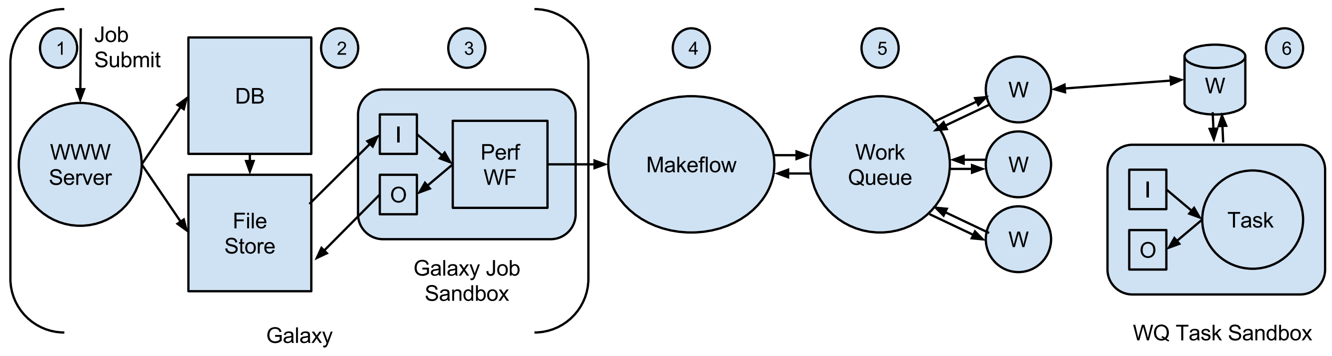

Fig. 1. Dynamic job expansion. In Stage 1, the job has been created by the user from the tool launch page. Once Galaxy gets the launch, the job is given an id, a working directory is created, and the job is added to the history. Stage 2, the files selected at launch are located via the file database, and the location is communicated to the job. Stage 3, inputs are collected, either directly or linked, in the job sandbox. Following setup, a script creats the performance workflow. Stage 4, Makeflow is launched with the performance workflow in the job sandbox, and the workflow begins processing. Stage 5, a Makeflow creates a Work Queue master that communicates with workers to create execution locations. Stage 6, the worker receives task, retrieving the inputs and task information. The task is computed and the output delivered back to Work Queue, who relays this to Makeflow. The performance workflow will move through stages 4, 5, and 6, until the workflow is complete. After completion, stage 4 will finalize the outputs and copy it to the output location defined by Galaxy. If successful, stage 3 is cleaned up, and the wrapper process concluded. At stage 1 Galaxy will change the job status and the user will be informed.

the input size. The input dataset was partitioned into splits of approximately 50K reads each. Partitioning the reference is also possible, but would increase the complexity of joining and negatively impact the use of cached files. We kept the reference whole and distribute it to all nodes of the performance workflow. The joining phase of the BWA workflow separated the header and content of each of the output splits, then generate a single output with one header and concatenated contents.

GATK employs a sophisticated Bayesian algorithm to compare aligned sequences with the reference. In this workflow, GATK's HaplotypeCaller walker was used for it higher sensitivity, when compared with GATK's UnifiedGenotyper. HaplotypeCaller functions by indexing the input set and creating walkers to locate variations between the query and reference. Once a difference is detected, HaplotypeCaller performs local assemblies to fill gaps or correct mistakes. This produces a output that expresses how closely alignments match and other information about the analysis as a whole.

The runtime of GATK is dominated by the size of the reference file. Thus, the job expander splits both the query file by size and the reference by distinct reference contigs. There is added complexity after completion due to correcting quality scores, but affects the runtime of the application less than if the input were a single piece.

Using Galaxy's workflow creation system, we were able to create tools that split and join the input files. However, the tool, in order to be used in a workflow, needed to have a static number of inputs and outputs, limiting the dynamic nature of this approach. This static design requirement is lifted when expanding a job at runtime allowing the number of partition be dynamically based on the input and not an arbitrary decision made when creating a tool.

V. Design Considerations

In the process of implementating job expansion, we encountered several problems that resulted from converting what was previously a locally executed job into a performance workflow: dependency management, remote execution, and garbage collection.

A. Dependency Management

1) Problem: Jobs depend, explicitly and implicitly, on things being in the local environment. An input file is an example an explicit resource and is straight-forward to express. Explicit resources are specified in the command, and expressed by the user at launch. Implicit resources are, by nature, left unspecified and known only by the program. Implicit resources fall into two categories, the first is reference material, such as database indexes, that are expected to exist within the environment. When a user is not expected to specify these resources, modifying a tools to require expression confuses tool implementation. The second category is environment resources, Java being a great example. Java is often required, but the environment's version may not be clearly expressed. This causes confusion for the developer as they attempt to use incorrect or absent versions.

For explicit resources, Galaxy tools clearly request resources using the tool XML launch page. This handles the expression of resources such as inputs and reference files. Implicit resources such as database index files, are added to a default location within the Galaxy instance. Referencing location in the environment allows the indexes to be located by the tools at execution. This solution requires users utilize existing indexes, or add references and indexes, which is perfomed by administrators. Implicit resources, such as Java versions, are more difficult to deal with as they can be dynamic. Versioning,

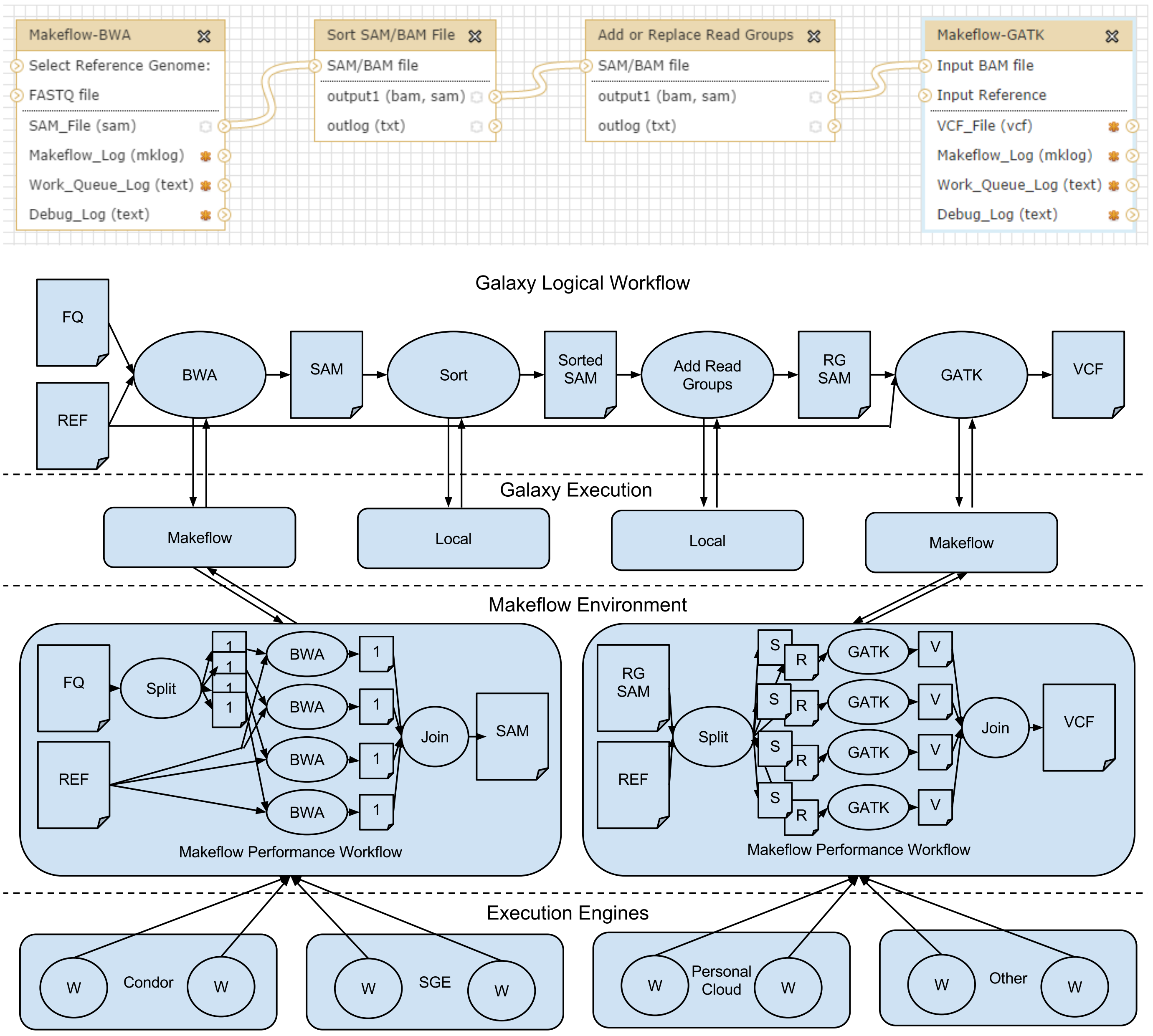

Fig. 2. **Detail of Example Application.** The top level shows a workflow as it is represented in Galaxy. Each box is a tool, with the names and arrows differentiating inputs and outputs. The Galaxy Logical Workflow lever simplifies the Galaxy representation to the simple logical workflow that it implies. This level shows the sequential nature of the jobs. The Galaxy Execution level defines the environment in which they are running, local being on the Galaxy instance and Makeflow denoting that the Makeflow process is local, but is creating tasks for parallelism. The Makeflow Environment level shows the process of expanding a single job. This level clearly shows the split-process-join nature of the performance workflow created. The lowest layer illustrates that workers from a number of systems can be utilized to perform the individual tasks.

in systems like Java, creates incompatitability that are difficult to handle in the program and necessitate the program expresses these requirements. These requirements need to either be provided by the developer or expressed in a manner so tools could access the intended variable or installation.

2) Solution: When designing an expanded tool, both the explicit and implicit resources must be expressed prior to execution. BWA alignment requires its reference be indexed and the index be present. Many users know these files, but often overlook them as they are assumed to be present. Galaxy allows this implementation, assuming that the reference is in a common location. This limits the references that the user can use as this location file needs to be updated for every additional reference, along with having the files moved to a safe location. This was addressed here by creating the index files at job execution. This adds the index files to the local environment, but must be recreated every run. In the ideal situation, the first time an index is created it is stored and the reference is for future use. This option does not require changes to the tool implementation that utilize the indexes, but must be monitored as these files are moved and collect in the system.

Handling environment settings also presents an interesting set of problems. Java is a great example, as many programs that utilize Java require a minimum and maximum allowed version. This is remedied by utilizing a script that unpacks the required version from an archive file and sets the neccesary environment variables to utilize the unpackaged version. This allows the required environment to be created at any execution site specified. Packing the environment with the tools allows freedom of movement for the execution, as well as guarentees that it is utilizing the correct tool. Enviroment replication is a well known issue encountered distributed computing. In our simple example, a python script was created to package Java locally prior to the workflow and one that unpacked Java at the execution site. More complex situation can be handled by tools such as Docker [15] and Umbrella [16].

B. Remote Execution

1) Design: Galaxy assumes that all jobs are run and handled by the machine on which Galaxy is running. This restraint was relaxed with the introduction of CloudMan [17], which allows the user to send jobs to a number of different cloud resources. CloudMan allows for the creation of a runner that submits jobs to a cloud resource. Galaxy's assumption that each task in a workflow is a single job allows for easy mapping from a Galaxy workflow to the scheduled task on a cloud resource. An issue that arose in our implementation is that the tool creates tasks that cannot use cloud resources without having a mechanism to launch them within the Galaxy controlled environment.

2) Solution: This was solved by utilizing Work Queue as an execution engine. Using Makeflow in conjunction with Work Queue, the tool uses a single port on the Galaxy machine that communicates with Work Queue workers on a variety of platforms. A pool of Work Queue workers was used to supply a steady stream of cloud resources. This allows tools to be used at different sites and execution engines. Work Queue utilizes project names and passwords to allow workers to connect without knowing the specific location ahead of time. The project names allows the tools to be configured by allowing Galaxy to name projects dynamically or for the user set it. Allowing users to devote resources to a project, even if the resources on the WMS are limited.

C. Garbage Collection

1) Design: When running an application, it is important to create a clean namespace for the application to work in, and clean the workspace after. Galaxy enforces this by creating a job working directly when a tool is launched. Galaxy runs the entire application from within this newly created directory. Tools limit the amount of space used by specifing the absolute path to the scripts, executables, inputs, and outputs. When using this method, tools make no noticable footprint within it and require no clean up. Some applications create temporary files and miscellaneous outputs by default that are not cleaned by the tool. The tool implemenation creates partitions of the input, as well as the creation of many intermediate files that are ignored by Galaxy, and would exist only at the working directory.

2) Solution: The solution strives to minimize the footprint and clean the directory. At the end of the a tool execution, after the outputs are transferred back, the tool utilizes the Makeflow clean option. This option removes all intermediate files and output files that are located in the current directory. Following this, the only remaining things to clean are executables and inputs that were fully transfered to the directory. It is the responsibility of the developer to clean any files that were explicitly added, as they are neither dynamically created nor removed by Makeflow.

VI. Opportunities

We also observed two opportunities for better integration between Galaxy and Makeflow, but did not address them in this work: expression of job status and checkpointing.

A. Expression of Job Status

1) Problem: Galaxy expresses a job as running, waiting, or complete. However, there is no clear indication of either a job's progress, or why a job is waiting. Additionally, the completion status relays to the user only if a job failed or completed as expected. Failures can be clarified by looking at the logs and system output, but leaves investigation of the cause to the user. The job running and completion status relay to the user the barest amount of information about the job. Some tools, such as GATK, give progress of what is being done and progress through the input. This helps estimate the time needed to finish as well as assure the user that processing continues. These simple estimates would help Galaxy users to better understand the tools and Galaxy's overall progress.

2) Solution: Our tool implementation does not directly address this design choice, but a simple solution may be found. If, within a tool, a status file were designated, the WMS could be configured to follow the tail of the status file. This would allow any tool to create a status file of a supported format to allow the WMS to track progress. A simple example of this would be processes reporting their percentage done. The WMS would need a means of expressing status in a location other than the output files history. The WMS is informed about the location of the working directory and status file to draw from. After this, tools need to simply create a status file to enable status updates. If a tool did not create a status file, then no more information would be relayed than already provided.

B. Checkpoints and Partial Failure

1) Problem: Completion in Galaxy is a binary state, where the job has either failed or completed, but there is no concept of checkpointing or partial failures. This is adaquate for tools that are extensively tested and there is little possibility of system circumstances interferring with completion, such as intermittent network connection. Galaxy treats jobs as local Unix tasks, where the process is either done or not. Performance workflows, however, can be represented as partially finished, as either a result of failure or the workflow is still computing. These partially completed workflows have checkpoints, that allow for the workflow to be moved and restarted. For example, Makeflow can be rerun using the Makeflowlog, when intermediate data is present. Though WMS can preserve the logs, it is difficult to preserve the working directory for reuse. A manner of expressing both checkpoints and non-fatal failures allows the underlying system to better express and handle failures. Also,workflows are easily created and run, but there is no functionality that allows for workflows to be rerun, as you would for a single tool. Utilizing previously computed results in restarting a partial workflow would improve usability and reproducibility.

2) Solution: The solution would be to define a means of returning either the working directory, if the user has machine level access, or creating an archive for the user to modify. The user resubmits the job after determining it was a non-fatal error, or modifies to correct a user error. This package or directory would then be resubmitted through a generic tools that detects the required tool. This solution fundamentally changes how WMS views tools and would require careful thought for correct design.

As Work Queue utilizes network resources, failures that are unrelated to the execution tool may occur. Failures in Work Queue are not necessarily failures in the workflow, but possibly an issue on the resource on which the task ran. In this case, running the Makeflow-Work Queue process again from the same directory and log changes the outcome of the workflow, while not recomputing any previously successful work.

A mechanism within the WMS that allowed directly re-running a workflow introduces a similar issue for workflows. Assuming the WMS allowed rerunning workflow directly, utilizing the previous results in a rerun would be as straight forward as using the links within the workflow to determine the work that has been previously done. The same result could be reached by creating a workflow log that would link previous progress to a new run of the workflow.

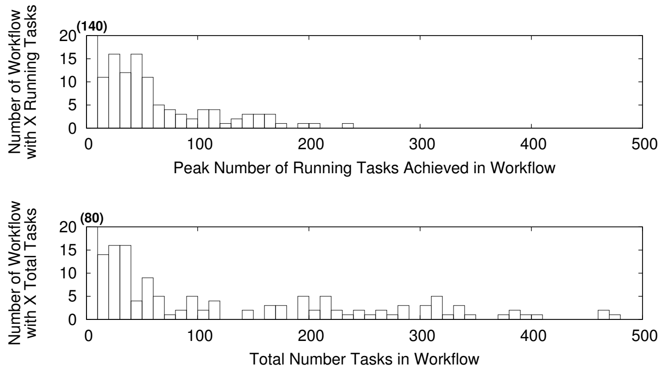

Fig. 4. The top images shows a histogram of the number of dynamically expanded workflows that had a max of X workers running. The bottom image shows the number of total tasks that each dynamically expanded workflow had over the course of its execution. All workflows shown were run with Work Queue and managed through Makeflow, and contains a mix of workflows run concurrently and alone. The difference between the graphs comes from two sources. The first being the tiered nature of the workflows, only allowing a portion of the total tasks being executed at any given point due to dependencies. The second being that for BWA the amount of concurrency was limited to 50 to prevent overloading the network. The last is the potential for better task partitioning and handling in future iterations as not every situation is perfectly matched with workers.

VII. Evaluation

To evaluate the combined system, we implemented dynamic job expansion on BWA and GATK as described above, and then ran the combined workload in four different configurations, using a campus Condor [18] pool to provision workers on demand for the expanded jobs. We varied the amount of query data in the first three configurations, using workers configured to each run a single task. In the fourth configuration (described below), the workers were configured to run four tasks at once, sharing a local cache.

Config	**Query**	**Reference**	**Tasks/Worker**
Small	0.6 GB	36 MB	1
Medium	7.5 GB	36 MB	1
Large	32 GB	36 MB	1
Large-Shared	32 GB	36 MB	4

Figure 3 shows the timeline of execution for each configuration. The thin line of each graph shows the number of tasks available to be executed, the thick line shows the number of tasks executing, and the gray bars show data transfer over a 10 second period. Note that the left axis measures tasks ready/running, while the right axis measures data transfers. In the first graph, dotted lines indicate the phases of the logical workflow. In the later graphs, only the BWA and GATK phases are shown.

On the largest dataset, the sequential version of BWA ran 19 hours, while the expanded version completes in one hour, 3 minutes. This resulted in a speedup of 18X on utilizing

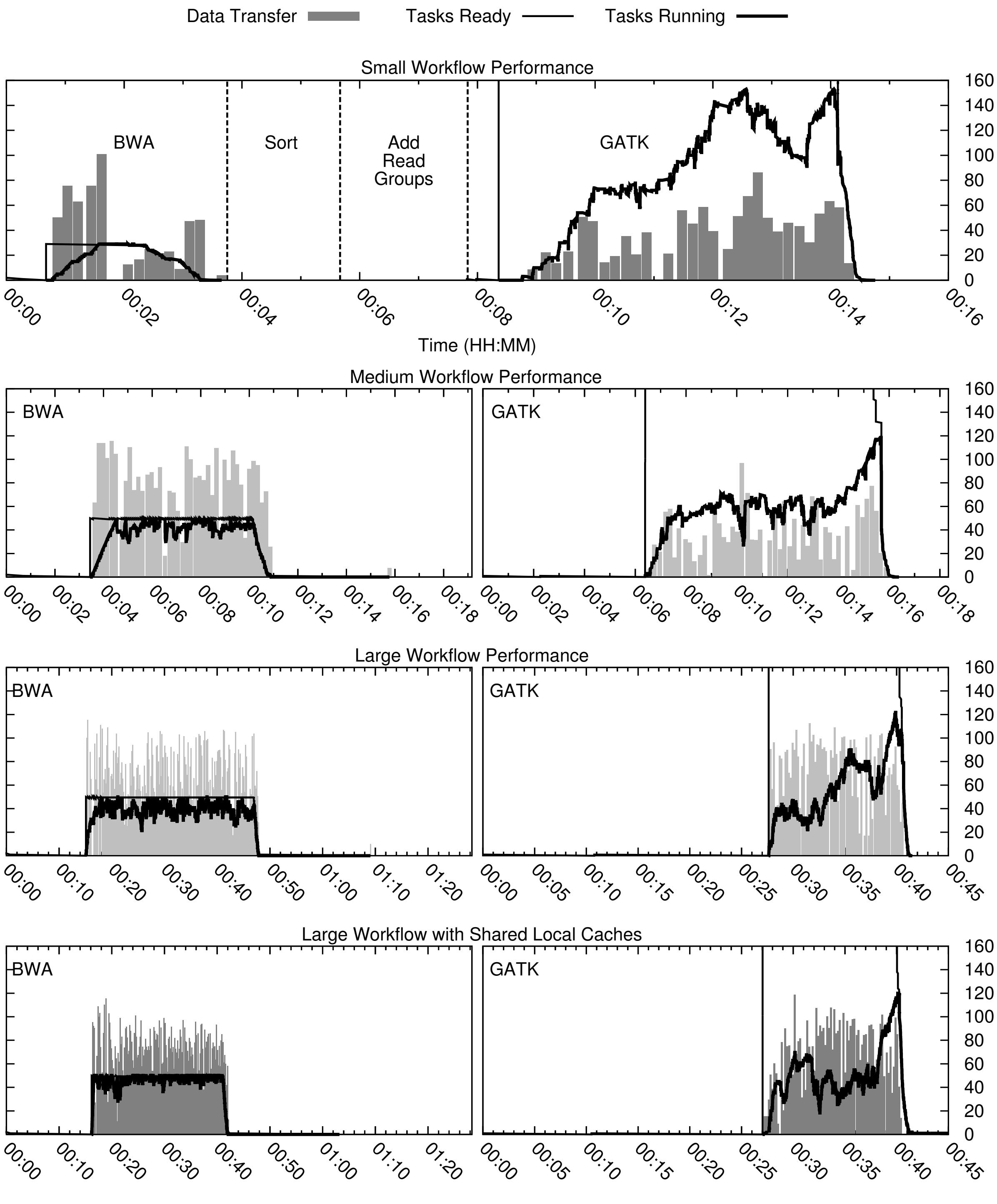

Fig. 3. This shows the execution of BWA, sorting, adding read groups, and using GATK to refine the results. The thin line of each graph shows the number of tasks available to be executed. The thick line shows the number of tasks executing. The gray bars show data transfer during a 10 second period. The left axis corresponds to the two lines, while the right axis corresponds to the data transfer. The four graphs, from top to bottom, show the small, medium, large, and large with shared cache workflows. As you can see we are able to dynamically create partitions that allowed for greater parallelism and performance in both BWA and GATK.

up to 50 workers. Figure 3 Row 4 Column 1 shows BWA, with the bold line in front representing the spliting task, the jagged section running BWA, and the end bold line joining the results. The sequential version of GATK ran for days, while the expanded version completes in 43 minutes for a speedup of 402X utilizing up to 125 workers. (The super-linear speedup comes from keeping the memory consumption of each task within physical memory.) Figure 3 Row 4 Column 2 shows splitting, GATK, and joining similar to BWA. The four-job logical workflow is accelerated by 61.5X overall.

In each configuration, it can be noted that neither the available workers nor the running tasks are ever constant. The workers vary due to competition from other users of the shared Condor pool. The variance in running tasks is due to the structure of the workload, and also due to the data transfer between the master and the workers. That is, the master can only dispatch tasks when the bandwidth can support transfer of necessary data to workers.

The main barrier to scaling up further is data transfer. For both BWA and GATK, not only must the query and reference datasets be sent, but also the software tools and dependencies such as the Java virtual machine. Each of these unique items is cached at each worker node and reused for future tasks. As the workload progresses, more workers have the necessary data cached, and parallelism can increase.

The first three configurations use worker processes that can execute one task at a time. This turns out to be inefficient, because the physical machines are multi-core and often end up running multiple workers simultaneously, each with its own distinct cache that must be managed. To improve this situation, we reconfigured the workers to consume four cores each, thus sharing a single local cache among four running tasks. The result of this can be seen in the bottom graph of Figure 3, where the running tasks grows more quickly and moves less data. In this configuration, we saw an 24 percent reduction in outgoing data from the master to the workers, from 69 GB to 52.1 GB. This is substantial, because as the number of concurrent tasks increases, the scalability is limited by the systems bandwidth.

The ideal scenario shown helps to showcase the benefits of this system design. To understand non-ideal situations, we looked at data gathered from the catalog server, which matchs workflows with workers, over the several month period of operation to see actual behavior. Figure 4 shows a comparison of histograms grouped by the number of tasks in each group, running and total. The disparity between the running and total histograms shows that there are limitations in the current implemenation. This is the result of several design decisions. The first is the limited concurrency through BWA. This only allows the running tasks to reach 50, despite the total number of tasks. This was done to limit the load on the network. The second is caused by general traffic on the machine that limited the ability to supply workers with work. These area provide an opportunity to improve the performance and further limit strain on the system. Figure 5 gives a full view of the system and the comparitive concurrency.

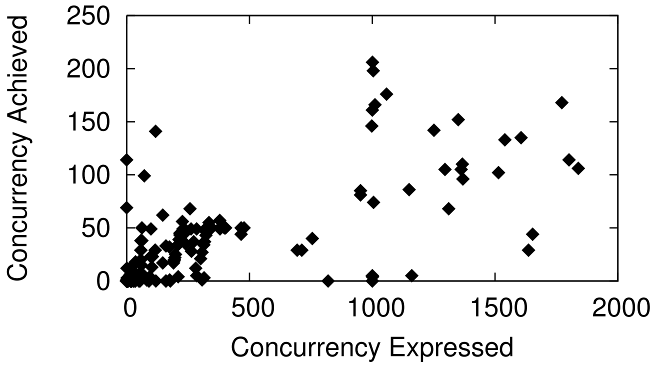

Fig. 5. The above image shows a scatterplot of the concurrency expressed, in terms of number of tasks, opposite of the concurrency achieved, the number of running tasks. As mentioned above, there are several reasons for disparaty between expressed and achieved, but it also showcases area for improvement in workflow creation. This would benefit from modeling functions to better partition the workflows.

Overall, dynamic job expansion has a dramatic impact on the overall performance of the workload, to the point where attention must be paid to improving the performance of the intermediate sequential steps.

VIII. Related Work

The area of workflow design and management is a heavily investigated field. This paper addresses two areas, the use of logical workflows that describe the overview process of applications and the performance workflows that aims to decrease runtime. Though Makeflow was selected for its simple syntax and flexibility, there many other WMS that can be used to create performance workflows, which allow for different control. Taverna [4] is a system that enables the creation of workflows for complex pipelines, and makes sharing and reusing workflows a high priority to enable collaboration and reproducibility. Pegasus [19] workflow management system utilizes directed acyclic graphs (DAG) similar to Makeflow. Pegasus make data flow and movement a priority and attempts to optimize scheduling for this. Other systems, such as Kepler [5] aim to provide tools for creating generic scientific workflows where portability and power are important.

Swift [20], [21] has even been used to investigate similar approaches that can be used to integrate it with Galaxy. The most pronounced difference between many of these approachs and our own is the focus on abstracting the parallelism from the user. Spark [22] is another powerful WMS, that focuses on utilizing the RAM to achieve higher performance, and could be leveraged to further boost performance. A great overview by Wang et al. [23], that looks at many of these systems and there place within the WMS spectrum.

Ideally, any of the above mentioned management systems could be used to replace either Makeflow, Work Queue, or both. However, these were selected due to their interoperability, Makeflows commandline interface, the light weight expression of a workflow, and familiarity with them. Each of these as advantages, that should be considered when using the proposed approach for dynamic expansion.

IX. Conclusion

In conclusion, this paper looked at the structure of Galaxy's Workflows and proposed a manner through which jobs can be partitioned for speed. An additional goal shown, is that the complexity and structure of job expansion is hidden from the user. This allows the user to utilize any number of tools with only the small learning curve of understanding Galaxy's environment and Workflow Management System, without needing to learn about cloud resources to utilize them. This manner of cloud resource utilization shifts the control and care of these resources to the resources provider, allowing to the correct usage and management to be monitored and controlled.

This paper also explores the design decisions of WMS systems, to better accommodate this type of work. In addressing these, WMS designers should look to how the system will be utilized and who should be responsible for different aspects of these systems control. This may lie on the designer to abstract difficult system interactions, on the tool developer to hide complexity and only show relevant choices to the user, or on the user to be educated on the use of a set of tools to better utilize the resources. These should be clearly laid out, so as users come to a system in different roles they are aware of what is abstracted away, and what can be relegated to the user.

X. Acknowledgments and Availability

This work was supported in part by National Science Foundation grant OCI-1148330. This source code is distributed under the GNU General Public License. See the URLs below for source code used in this paper.

`http://ccl.cse.nd.edu/`

`https://github.com/cooperative-computing-lab/cctools/`

References

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

[2] D. Blankenberg, G. V. Kuster, N. Coraor, G. Ananda, R. Lazarus, M. Mangan, A. Nekrutenko, and J. Taylor, "Galaxy: A web-based genome analysis tool for experimentalists," *Current protocols in molecular biology*, pp. 19–10, 2010.

[3] B. Giardine, C. Riemer, R. C. Hardison, R. Burhans, L. Elnitski, P. Shah, Y. Zhang, D. Blankenberg, I. Albert, J. Taylor, W. C. Miller, W. J. Kent, and A. Nekrutenko, "Galaxy: a platform for interactive large-scale genome analysis," *Genome research*, vol. 15, no. 10, pp. 1451–1455, 2005.

[4] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. P. Balcazar Vargas, S. Sufi, and C. Goble, "The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud," *Nucleic Acids Research*, vol. 41, no. W1, pp. W557–W561, 2013. [Online]. Available: http://nar.oxfordjournals.org/content/41/W1/W557.abstract

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

[6] M. Albrecht, P. Donnelly, P. Bui, and D. Thain, "Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids," in *Workshop on Scalable Workflow Enactment Engines and Technologies (SWEET) at ACM SIGMOD*, 2012.

[7] P. Bui, D. Rajan, B. Abdul-Wahid, J. Izaguirre, and D. Thain, "Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications," in *Workshop on Python for High Performance and Scientific Computing (PyHPC) at the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (Supercomputing)* , 2011.

[8] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows–wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[9] M. DePristo, E. Banks, R. Poplin, K. Garimella, J. Maguire, C. Hartl, A. Philippakis, G. del Angel, M. Rivas, M. Hanna *et al.*, "A framework for variation discovery and genotyping using next-generation DNA sequencing data," *Nature genetics*, vol. 43, no. 5, pp. 491–498, 2011.

[10] O. Choudhury, D. Rajan, N. Hazekamp, S. Gesing, D. Thain, and S. Emrich, "Balancing thread-level and task-level parallelism for data-intensive workloads on clusters and clouds," in *IEEE Cluster*, 2015.

[11] P. J. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice, "The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants," *Nucleic acids research*, vol. 38, no. 6, pp. 1767–1771, 2010.

[12] O. Harismendy, P. C. Ng, R. L. Strausberg, X. Wang, T. B. Stockwell, K. Y. Beeson, N. J. Schork, S. S. Murray, E. J. Topol, S. Levy *et al.*, "Evaluation of next generation sequencing platforms for population targeted sequencing studies," *Genome Biol*, vol. 10, no. 3, p. R32, 2009.

[13] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin *et al.*, "The sequence alignment/map format and SAMtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009.

[14] O. Choudhury, N. Hazekamp, D. Thain, and S. Emrich, "Accelerating comparative genomics workflows in a distributed environment with optimized data partitioning," in *C4Bio Workshop at CCGrid*, 2014.

[15] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[16] H. Meng, R. Kommineni, Q. Pham, R. Gardner, T. Malik, and D. Thain, "An invariant framework for conducting reproducible computational science," *Journal of Computational Science*, vol. 9, no. 0, pp. 137 – 142, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877750315000502

[17] E. Afgan, D. Baker, N. Coraor, B. Chapman, A. Nekrutenko, and J. Taylor, "Galaxy cloudman: delivering cloud compute clusters," *BMC bioinformatics*, vol. 11, no. Suppl 12, p. S4, 2010.

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

[19] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny *et al.*, "Pegasus, a workflow management system for science automation," *submitted to Future Generation Computer Systems*, 2014.

[20] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 2011.

[21] K. Maheshwari, A. Rodriguez, D. Kelly, R. Madduri, J. Wozniak, M. Wilde, and I. Foster, "Enabling multi-task computation on galaxy-based gateways using swift," in *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, Sept 2013, pp. 1–3.

[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[23] J. Wang, D. Crawl, I. Altintas, and W. Li, "Big data applications using workflows for data parallel computing," *Computing in Science Engineering*, vol. 16, no. 4, pp. 11–21, July 2014.