Adapting Bioinformatics Applications for Heterogeneous Systems: A Case Study

Irena Lanc
University of Notre Dame
Notre Dame, IN
ilanc@nd.edu

Peter Bui
University of Notre Dame
Notre Dame, IN
pbui@nd.edu

Douglas Thain
University of Notre Dame
Notre Dame, IN
dthain@nd.edu

Scott Emrich [*]
University of Notre Dame
Notre Dame, IN
semrich@nd.edu

ABSTRACT

The advent of new sequencing technologies has generated extremely large amounts of information. To successfully apply bioinformatics tools to such large datasets, they need to exhibit scalability and ideally elasticity in diverse computing environments. We describe the application of Weaver to the PEMer structural variation detection workflow. Because the original workflow has an intractable sequential running time on large datasets, it also has a batch implementation designed for a shared file system. Using scripts provided by the developers of PEMer, along with the Weaver Python module, the Starch archive generator, and the Makeflow workflow engine, we have refactored PEMer for elastic scaling on personal clouds. Our case study describes the various challenges faced when constructing such a workflow, from dealing with failure detection, to managing dependencies, to handling the quirks of the underlying operating systems. The practice of scaling bioinformatics tools is increasingly commonplace. As such, the hands-on application of refactoring techniques to PEMer can serve as a valuable guide for those looking to reconfigure other bioinformatics software. Significantly, our customized Makeflow framework enabled elastic deployment on a wider variety of systems while substantially reducing wall clock runtimes using hundreds of cores.

Keywords

Bioinformatics, Distributed Systems

General Terms

Design, Performance

[*]To whom correspondence should be addressed

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

1. INTRODUCTION

The explosion in the size of biological datasets has fueled interest in exploring the boundaries of scalability for many common bioinformatics tools. While the ability to parallelize existing bioinformatics programs essentially depends on whether the underlying problem lends itself to parallel execution, the majority of genome-focused tools often have an inherent ability to be performed in parallel. For example, alignment and assembly are now frequently performed on multiple cores using frameworks such as MPI [2] and MapReduce [9]. Even so, there are multiple serial implementations targeted towards a particular project or even on a specific dataset that could benefit from scaling in a production setting. For such software it is up to users to refactor existing code such that the functionality of the program is conserved.

Refactoring has been historically aided by the widespread availability of distributed, batch and now cloud computing systems. Frameworks such as our Makeflow engine [12] provide a straightforward route for software with clear data dependencies and simple serial execution. Because a single bioinformatics Makeflow can be thousands of lines long, we have shown that creating scripts and higher level programs is the key for successful and practical parallelization. To this end, we have shown the Weaver tool, together with the Starch archive manager, provides a standardized framework for relatively simple bioinformatics workflows [10].

Here, we refactor the structural variation detection tool PEMer [5] to run on clusters, grids and clouds as a case study for complex bioinformatics workflows. The provided PEMer batch execution script executed quickly on a standard batch system (SGE) with a shared file system (NFS). We apply the Weaver/Starch/Makeflow application stack to extend the scalability to heterogenous systems lacking shared resources typical of personal/ad hoc clouds. We show that this process can be fraught with challenges and pitfalls substantially more complex than our previous results. Significantly, we describe elastic scaling of the PEMer pipeline in this framework and test it on hundreds of processors. Our paper is structured as follows: Section 2 describes the PEMer structural variation tool. Section 3 covers both our approach to implementing the PEMer pipeline as a Makeflow, as well as the general lessons extracted from the process. Section 4 presents the results of executing the pipeline on different batch systems. In Section 5, we contrast our approach with the original bundled batch implementation, and we give our conclusion in Section 6.

2. AN INTRODUCTION TO PEMER

The PEMer pipeline [5] consists of four core steps that work together to discover structural variants given a reference genome and a set of mated pair sequence queries. The general outline of this process is given in Figure 1. This pipeline is trivial to execute sequentially on small tasks but requires significant resources to complete; running PEMer with approximately 136000 mated pair sequences on a small 230 million nucleotide arthropod genome [1] quickly exhausted 32 GB of RAM and would have required weeks of compute time executed sequentially. PEMer serves as a model complex workflow for distributed execution, given the large number of steps involved and their myriad of dependencies, which provide a significant challenge to adapt for heterogenous systems. While it can be implemented on one machine easily, and executes quickly using a shared file system, it proved to be both a challenge and a significant learning experience to adapt for execution on a personal cloud. In this case study, we will share the useful information gleaned during this process. As the practice of scaling even small, customized programs becomes more commonplace given the deluge of available biological data, it is certain that other users will encounter similar challenges. We provide guidance on some necessary adjustments using PEMer as an example.

3. METHODS AND DATA

The effort to scale PEMer began with identifying the steps that would lend themselves well to clusters, grids and clouds. We established that all of the most significant serial steps in the PEMer workflow could be successfully refactored in our established Weaver/Makeflow framework [10] while alleviating the massive memory requirements that had swamped the sequential version. This framework was employed as Weaver is already an extension of the popular programming language Python, which has numerous well-developed resources and a more familiar syntax than a newer language like Swift. In addition, the Weaver stack is already tailored to the resources at our disposal, specifically SGE, Condor and WQ, and so additional installation of tools such as CoG Karajan was avoided. Finally, Weaver did not require any adaptation or expansion of functionality to work, as we did not need to define data types or do any XML, reducing the time needed to develop and test the pipeline.

Given that the abstractions Weaver provides related well to those in PEMer (Figure 1), we found it relatively easy to refactor them into a Makeflow [12]. A sample of the code to do this is given in Table 1. Next, it was necessary to ensure appropriate dependencies when executing remotely. This was achieved using Starch [10], an application manager that allows users to encapsulate all dependencies, executables, environment variables and even multiple commands needed to run a standalone application into simple files called *archives*. This greatly simplified the Makeflow construction by allowing multiple steps to be executed within what is, at face value, a single command and its associated archive. This not only ensured a clean abstraction of the required executables, but also removed the need for the complicated data management strategies usually required in a remote environment typical of personal clouds.

Once the Makeflow script was generated, it was distributed using the Condor batch computing system for testing using both the Makeflow and Work Queue frameworks. The flow of control in the program is illustrated in Figure 2. Each oval represents a job to be executed on the Condor grid; the DAG structure maintains proper execution order, ensuring that input/output dependencies are handled prior to further execution. Condor provides access to a highly heterogeneous collection of remote machines, though it maintains a reasonably high level of fault tolerance by retrying failed jobs and logging execution information and failure states. Makeflow and Work Queue provide similar functionality as frameworks for performing distributed computing on diverse systems, but differ slightly in how they execute remote jobs. This allowed us to test the flexibility of our Makeflow under varying distributed computing models for clusters, grids and clouds. For completeness, we also prepared the PEMmer batch script and our own pipeline for execution on SGE using Weaver/makeflow to collect comparable runtimes.

Note, however, that execution on SGE also afforded us benefits of a shared file system. Under this scenario there was no longer a need to construct Starch archives containing the supplemental scripts, data files, and environmental variables needed to support remote execution. Further, this reduced runtime, as the archives would not be unpackaged each time one of the programs would run. To ensure comparability of runtime results our SGE modifications were made to the original Weaver script, and consisted of simply replacing the names of archives with the names of the programs they encapsulated.

PEMer requires mate pair reads, a mate-pair information file, and a reference genome in order to execute. We used a 2.0 GB file of trimmed mate pair reads that, when combined with the mate-pair information file, created a 231 MB file formatted for input to PEMer. For reference, we used the completed *D. pulex* genome, 222 MB in size. It can be expected that most research into structural variation would use data sets of a similar size or larger. Given that this data set proved large enough to warrant investigation into alternatives to sequential execution, our successful parallelization is sure to have positive repercussions for projects an order of magnitude larger such as locating structural variation between humans (e.g. [8]) important in recent diversity and cancer analyses.

3.1 LESSONS LEARNED

Several changes and modifications had to be implemented to achieve better scalability and overcome certain inherent limitations. The challenges encountered during this Makeflow-based case study led us to extract several general principles that can guide the adaptation to personal clouds:

I: **Determine optimal granularity.**

II: **Understand remote path conventions.**

III: **Be aware of the scalability of native OS utilities.**

IV: **Identify semantic differences between the batch system and local programs.**

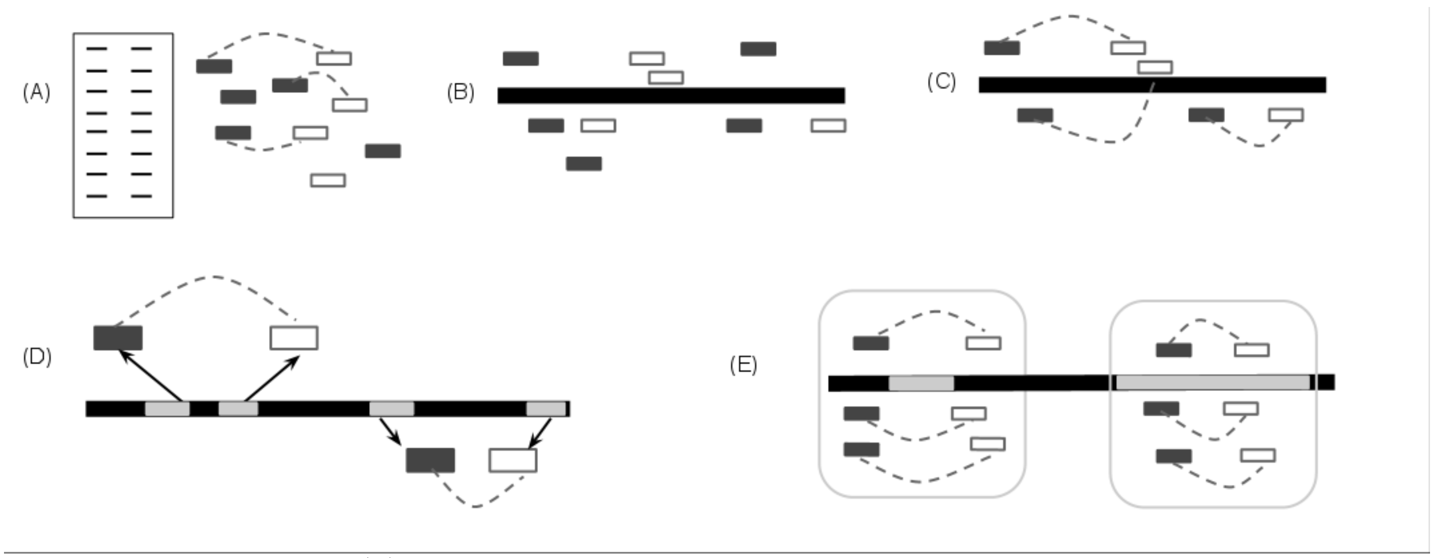

Figure 1: The PEMer pipeline: (A) Preprocessing creates mate pairs given the list of paired sequences and associated set of reads. (B) Mate pair ends independently aligned to reference using Megablast or MAQ. (C) Optimal placement of mate pair reads according to alignments that seek to minimize the occurence of outliers. (D) Identification of mate pair outliers. (E) Sets of outlier mate pairs are categorized as structural variations if N or more independent paired ends can be clustered according to each variation.

`hits2placement = StreamFunction('Hits2PlacementScore_GB.py ,cmd_format='exe input args`	`input.needle.placement:  input.fa.200.fa-needle Hits2PlacementScore_GB.py my_config.py`
`> output', cmd_args= '-d %s -c %s' % ('placement.discards', my_config])`	`Hits2PlacementScore_GB.py input.needle -d placement.discards`
`hits2placement_output = Map(hits2placement, needle_input)`	`-c my_config.py > input.needle.placement`

Table 1: Sample code: The Weaver instructions on the left generate the Makeflow instruction on the righthand side.

V: **Establish the execution patterns of the program or pipeline.**

I. **Determine optimal granularity**
Striking a good balance between the size of each individual job and the total number of jobs can allow users to optimize the trade-off between transmission and execution time. Formal analysis has shown that careful selection of job size can yield distinct performance benefits [6]. A very small job size, whose transmission time exceeds its execution time, can prevent the workflow from reaching full capacity. Jobs that are too large can result in eviction of jobs from remote machines, especially in a highly heterogeneous system such as Condor.

The primary configuration script for PEMer was modified to allow for the splitting function to create appropriately sized subsets of files because extremely small jobs incurred extremely long execution times, and therefore did not scale. Several job sizes were attempted in this study before a size that yielded a reasonable runtime was found.

II. **Understand remote path conventions.**
The idiosyncrasies of batch systems can result in unexpected interpretations of file names and paths on remote machines. This problem becomes more pronounced with the addition of multiple dependency files and programs. Evaluating the required format for input to remote jobs can reveal unexpected formatting requirements that prevent even properly specified files from being correctly transferred and executed on remote machines.

For our PEMer makeflow, the rules were modified to include the full path of needed files because soft links were not interpreted correctly on our remote machines. The syntax requirements were such that the names of all files drawn from higher directories needed to replace forward slashes with underscores. This is not by any means a common feature of other distributed system frameworks, but is illustrative of surprises that can manifest themselves in ad hoc/personal clouds based on even the most well-implemented systems.

III. **Be aware of the scalability of native OS utilities.**
Scripts will oftentimes make use of UNIX functions such as `cat`, `redirect`, `pipe` and `find` to handle simple data manipulation. While these utilities have well-understood behavior within a local system, they may pose problems when integrated into a cloud. The usage of these commands in a parallel environment has been explored in the context of enabling functionality across distributed resources [3]. However, they are also susceptible to errors when applied to extremely large datasets of the kind generated during large-scale, sequence-based bioinformatics. These errors can manifest themselves in the form of line length or argument limits on functions such as `cat`. It is best to avoid this issue entirely by making use of `exec` or `xargs` in any case where it is suspected the number of arguments may exceed system capacity. File limits in folders also pose a potential problem. Extremely large distributed workflows may run up against such constraints in filesystems such as AFS or NFS, potentially hobbling the scalability of the pipeline if executed from that space. Users should be mindful of these limits prior to execution.

For example, we encountered a concatenation step failure due to a limit on the line length limit on arguments in UNIX. This problem did not manifest on small test sam-

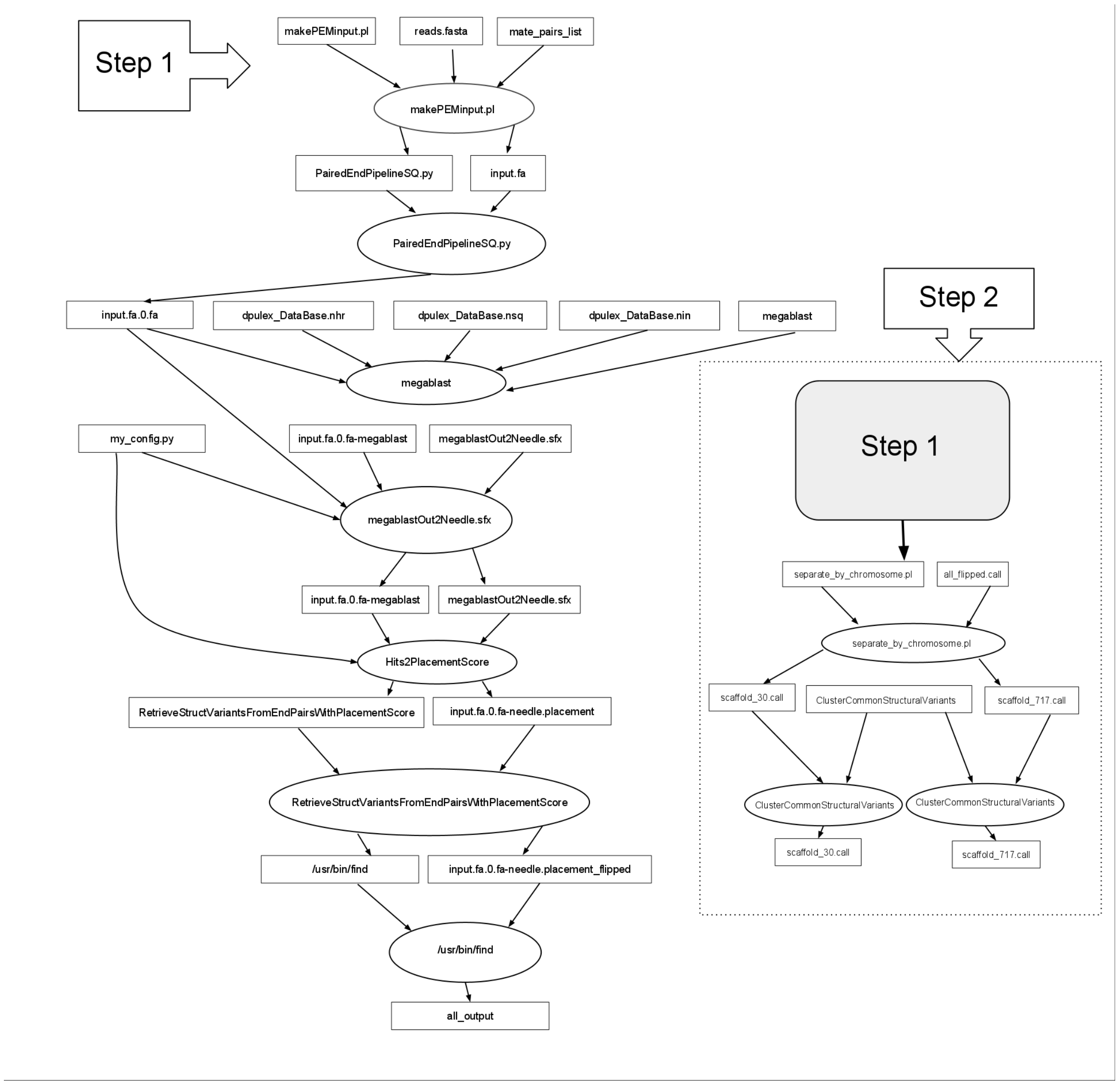

Figure 2: Representing the PEMer pipeline as a set of directed acyclic graphs.

ples; however, as we scaled up the size of the dataset this issue became apparent through the continued failure of a specific step. As suggested above, this was solved by modifying the original concatenation step using `find` with the -exec option, allowing the `cat` command to run on each file separately. While this is not the most efficient approach, in practice it executes very quickly. This particular error illustrates the importance of performing tests on small and large datasets. Between a small, successful deployment and a production-scale job can lie a blind spot where unexpected errors may appear.

IV. **Identify semantic differences between the batch system and local programs.**
The goals of users implementing distributed pipelines can differ both practically and philosophically from those of batch systems. What a batch system may define to be successful completion of a program can differ substantially from the expectations of the user. This disparity between user expectation and system behaviour is seen in other large-scale distributed systems such as AFS, where files that have been written but not yet closed are only visible to the local machine [4]. In the context of the Makeflow framework, "success" is defined as returning a file to the master node, regardless of the file's content. In a pipeline where there are output dependencies between steps, this can pose a significant problem, as users expect correct data to be fed into subsequent steps. Such conflicts of interest can be avoided by working to align the user's priorities with that of the system. This can be as simple as adding a test condition to change the meaning of a job's return status.

In the case of PEMer, a subtle error appeared that circumvented the regular error-correcting mechanisms. Occasionally, empty files from failed instances of the megablastOut2-Needle step were returned, but because they had the correct name they were interpreted as having finished successfully. This would ultimately lead to missing data and incorrectly completed steps later in the pipeline. To avoid painstakingly finding, removing and re-running these steps, we modified the remote job to test completion within our novel Starch executable command line framework. Specifically, we used the simple UNIX system call `stat` to check whether the returned file was of a valid size, indicating a non-empty file. If not, the status of the job was returned as 1 to indicate failure, and 0 otherwise. This ensured that only successfully completed jobs were returned.

V. **Establish the execution patterns of the program or pipeline.**
Analysis of program structure for the purpose of parallelization is an established area of research, with studies focused on bioinformatics-specific applications emerging within the last decade [9, 11]. Recognizing possible abstraction, determining granularity, and analyzing data flow are all necessary for successfully adapting programs for scalable execution. Adapting a program for a workflow framework such as Makeflow relies on the assumption that it will be executing a preset number of known workflow tasks. Some steps, however, hinge on the results of previously processed data in such a way that it is impossible to know *a priori* the number of commands that need to be executed later. In its current form, the Starch/Weaver/Makeflow stack is meant to create a workflow whose format is known ahead of time. While the static nature of this system ultimately proved to be a manageable limitation, it showed that identifying parts of the pipeline where such data dependencies can interfere with the static generation of the Makeflow is vital. When users identify these decision points, they are better able to plan alternate approaches to their workflows to avoid such problems.

We identified four steps of the pipeline that could be successfully executed as independent computational units, and applied the Map abstraction in Weaver to handle certain large subsets of the data. We also implemented a solution that allowed us to adapt generation of Makeflows to the conditional flow of data. The final step of the PEMer pipeline relies on knowing the number of output files created in previous steps. This cannot be determined initially, as it is a function of the particular biological properties of the data set. Given that there is no current way to dynamically generate makeflows based on the results of earlier steps, we wrote a secondary Weaver script for the final step based on results generated by the first script. When this is compiled, it generates a new Makeflow that then completes the last step of the PEMer pipeline in parallel. Though this approach is not ideal, it does enable the entire pipeline to benefit from clusters, clouds and grids.

4. RESULTS

The application of our scalable, distributed pipeline proved to be successful in speeding up PEMer's execution (Table 2). We found that the program scaled well, was easy to execute, and provided definite advantages on both the Makeflow and Work Queue frameworks. For Makeflow, we performed distributed execution under two different conditions, varying the number of remote jobs allowed to execute at a given time from 100 to 300. In both cases, a 12-core machine with 12 GB of RAM was used for makeflow submission. This generic machine was open to all students on campus, and frequently had multiple users on it. Our speedup was calculated by dividing the *goodput*, or total CPU time devoted to successfully completed jobs, by the total wall clock time required to complete execution.

We observed that there was considerable improvement in runtime, and, as expected, scaling to more processors was limited mostly by the ability of the submitting machine to keep up with more workers [7]. We also executed the pipeline via Work Queue using 100 workers, 300 workers, and a dynamic scaling approach where the number of available workers was gradually increased over the course of two and a half hours. The pipeline was begun with 10 workers, and 40 additional workers were added after an hour. Another half-hour after that, 50 workers were again added, and finally 200 workers were started after one more hour. The number of submitted and completed jobs remained low until the addition of the final 200 workers, at which point the rate of completion and submission increase noticeably, and it remained steady for hours.

Implementation	Wall Clock Time	CPU Time	Speedup
Sequential	> 2 weeks	N/A	N/A
Makeflow (100)	0+19:15:32	73+10:29	91.5
Makeflow (300)	0+08:49:57	71+12:43:27	194.36
Work Queue (100)	0+23:44:24	84+17:21:57	85
Work Queue (300)	0+11:5:47	84+17:21:57	169
Work Queue (scaled)	0+10:10:49	73+15:37:24	173
SGE original (100)	0+1:16:33	5+2:9:37	95.7
SGE new (100)	0+18:31:21	73+12:8:54	95.2

Table 2: Runtimes of the various PEMer implementations

We attempted to perform a sequential execution of the pipeline, using an 8-core 32 GB machine with similar user contention as the machine used to submit makeflows. We ran the pipeline for more than a week and a half and were unable to move past the third step of the pipeline. Unfortunately, due to time constraints and disproportionate resource consumption, the job could not be completed. In contrast, all distributed instances of the pipeline managed to complete in under a day, on machines that were being periodically utilized by multiple other individuals.

The visualizations of runtime behaviour for the Condor (Fig 3.) and Work Queue (Fig. 4) implementations of the pipeline indicate that number of jobs submitted and jobs completed during execution follow a similar pattern. Both follow each other closely, and show a steady increase over time, eventually reaching a plateau where all jobs have been submitted, but some remain to be completed. It can be seen from Fig.4 that the submission and maintenance of 300 concurrent workers results in a uneven number of running jobs. Running at this scale results in more competition across the available resource pool, a larger risk of being evicted by the machine's owner in a shared model, or experiencing machine failure. Regardless of this fluctuation, the runtime for the 300-worker instance (9 hours) represents a significant improvement over both the 100-job and sequential executions using a highly heterogenous, volunteered cloud environment.

The scaled Work Queue implementation (Fig 4) displayed much more consistent behaviour as expected. once workers established themselves they continuously field demands for job execution. Thus, they are represented as being constantly running, rather than being subject to the fluctuations of a regular makeflow in a contentious resource pool. The steep tail of running jobs that begins at 1:00 represents the rapid scaling down that occurs when the majority of jobs have finished, unused workers die out, and all that is left are jobs with abnormally long running times. These may have landed on busy machines, though their influence on total runtime can be mitigated by use of the fast abort option

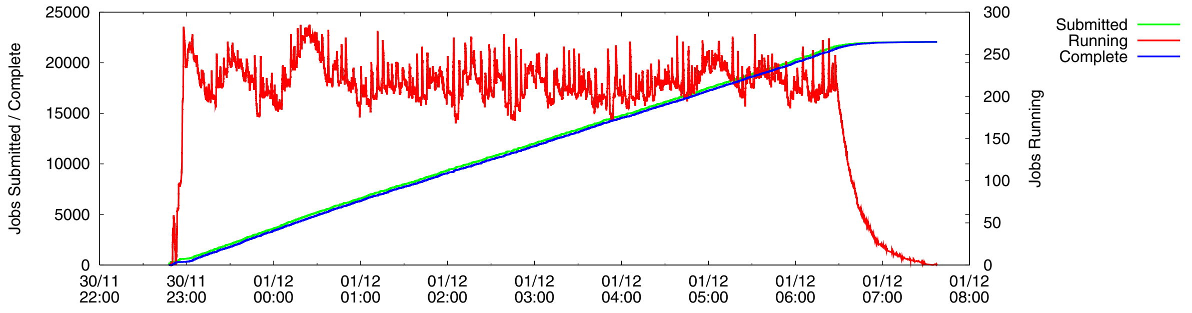

Figure 3: A timeline of execution using Makeflow with Condor, with a maximum of 300 jobs running at a given time.

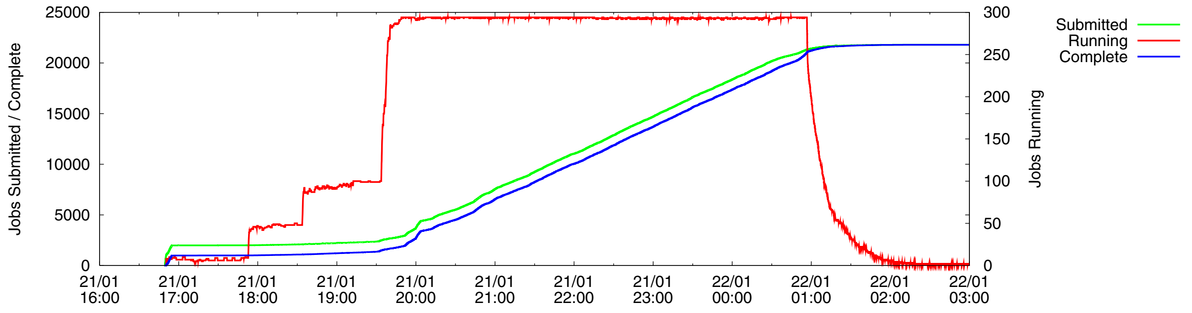

Figure 4: A timeline of execution using Work Queue on Condor, where the number of running workers is gradually scaled up over the course of two and half hours.

of Work Queue. Here, we employed a fast-abort multipier of 300, which was high enough to ensure that all jobs had ample time to complete.

5. COMPARISON TO PROVIDED BATCH EXECUTABLE

The authors of PEMer have provided their own version of a distributed pipeline for their tool, part of which is incorporated into our own pipeline in the form of a splitting script. In their version, this script is run on the pre-formatted PEMer input file, in order to divide it into a user-defined number of sub-files. This also generates a batch submission script, in which a single program encapsulating the four central steps of PEMer is run on each sub-file.

For comparison, we ran the provided pipeline in our Makeflow framework on a shared file system and a slightly modified SGE version of our own pipeline on our personal cloud without a shared filesystem. In support of our generic framework, our Makeflow+PEMmer batch script exhibited very good performance and efficiency; it finished in roughly an hour with a a speedup of 95 using 100 processors (see Table 2). Our personal cloud version showed comparable speedup with respect to total CPU time required (95x speedup) but took about 18 hours to complete in a contentious computing environment. The large difference in time could be attributed to the increased computational and transfer requirements of running without a shared file system, which do not scale with the number of jobs, or an increased risk of preemption associated with attempting to running a large number of jobs on a volunteered computing resource like Condor.

The main advantage of refactoring traditional PEMer as a Makeflow script could enable us to automate submission of the script to multiple batch systems using the Makeflow engine [12]. Further, Makeflow also keeps a detailed log of runtime behavior and failure events, which allows analyzing runtime behavior and error correction. This gives insight into the scaling capabilities of the script, and allows us to track the remote behavior of jobs at a much finer grain than SGE alone. When we used the Work Queue engine, which establishes consistent workers on remote machines and only sends jobs to those workers, we gain additional advantages. Each worker maintains its dependencies locally, which eliminates the need to re-send data used in multiple jobs (the 222 MB genome in this case study, for example). Further, workers can be started on any machine, even independently of batch systems such as Condor. This gives the user even greater degree of freedom in scaling up to more processors via this personal cloud. Significantly, by running our scalable pipeline with both Makeflow and Work Queue we avoided having to submit only to machines with a shared file system. In this experiment, caching of required dependancies does not significantly alter performance (Table 2), but does give a more reliable set of persistent workers.

Secondly, and perhaps most crucially, separating the various components of PEMer into discrete steps rendered the entire pipeline more transparent. Many errors were caught by examining output files of each step as they were returned to our local machine. This would be more difficult if we were to execute all steps via the single consolidated script. Such difficulty would be further compounded without a batch submission method or any pre-staging of work.

Our implementation also parallelizes the final step of the pipeline that clusters found variations according to their location on the genome. This step was not included in the

Attribute	**Provided Batch Script**	**Makeflow**
Requires Shared File System	Yes	No
Deployment Environment	Shared file system batch system, e.g. SGE	Any batch system, e.g. Condor, SGE, Work Queue
Code Encapsulation	A single script that encapsulates all four core programs	The pipeline consists of multiple discrete steps executed consecutively
Logging	Start and stop times, program log captured using stderr and stdout	Detailed execution log, batch system log and optional debugging output

Table 3: Differences between the provided batch script and the Makeflow/Work Queue implementation of the PEMer pipeline

original script because it is not as computationally intensive in comparison to other pipeline steps. However, it provided a valuable opportunity to explore the process of creating nested makeflows where one or more steps is achieved using on-the-fly internal makeflows. At present, makeflows cannot currently be dynamically generated; the file is created users are unable to create rules that rely on data unavailable at runtime. In this particular case, the number of jobs required for the last step of PEMer was dependent upon on the total number of structural variations, which was initially an unknown quantity. To handle this case we created a secondary Weaver script, to be run from within the same sandbox upon completion of the first makeflow. Our setup enables flexibility as users are also able to verify the results of the first deliverable if required.

Finally, refactoring PEMer in makeflow made this pipeline more flexible. We were easily able to run this pipeline on clusters, grids and clouds and with and without shared filesystems without modifying our code. We believe this flexibility is needed to ensure scalability and elasticity on highly heterogeneous systems such our campus grid and personal clouds. Users of our new pipeline can choose to leverage performance enhancing systems such as AFS shared files and persistent workers using work queue but do not require them. For a summary of the differences between the two implementations, see Table 3.

6. CONCLUSIONS

We successfully refactored the PEMer structural variation pipeline for execution on clusters, grids and clouds. Run times were reduced to hours from weeks with high speedup. Weaver proved to be a flexible tool that allowed for the adaptation of each program to the Makeflow in a clean and relatively intuitive manner. The process of scaling this tool was found to be susceptible to a variety of complications, however. These were dealt with through a number of modifications, such as simple UNIX commands and additional custom scripts. An awareness of the potential obstacles, and some possible strategies to deal with them, can ease the process of customizating bioinformatics software for personal/ad hoc clouds. This in turn will lower the barrier to scalable execution, allowing users to better harness the power of heterogeneous distributed systems for their own tools.

7. ACKNOWLEDGEMENTS

We would like to acknowledge the help of Xinmeng Jasmine Mu and Nicholas Carriero, who contribued their time and effort to answering questions and modifying the PEMer source code.

This work was supported in part by National Science Foundation grants NSF-0643229, NSF-0621434, and NSF-08554087.

8. REFERENCES

[1] J. K. Colbourne, M. E. Pfrender, D. Gilbert, W. K. Thomas, and others. The Ecoresponsive Genome of *Daphnia pulex*. *Science*, 331(6017):555–561, 2011.

[2] A. E. Darling, L. Carey, and W. chun Feng. The design, implementation, and evaluation of mpiBLAST. In *In Proceedings of ClusterWorld 2003*, 2003.

[3] P. H.-P. Implementation and E. O. E. Lusk. Scalable unix commands for parallel. In *In Proceedings of the Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 410–418. Springer, 2001.

[4] M. L. Kazar. Synchronization and caching issues in the andrew file system, 1988.

[5] J. Korbel, A. Abyzov, X. Mu, N. Carriero, P. Cayting, Z. Zhang, M. Snyder, and M. Gerstein. PEMer: a computational framework with simulation-based error models for inferring genomic structural variants from massive paired-end sequencing data. *Genome Biology*, 10(2):R23+, 2009.

[6] C. Moretti, J. Bulosan, D. Thain, and P. J. Flynn. All-pairs: An abstraction for data-intensive cloud computing. In *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS*, pages 1–11, 2008.

[7] C. Moretti, M. Olson, S. J. Emrich, and D. Thain. Highly scalable genome assembly on campus grids. In *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers, MTAGS*, 2009.

[8] A. Pang, J. MacDonald, D. Pinto, J. Wei, et al. Towards a comprehensive structural variation map of an individual human genome. *Genome Biology*, 11(5):R52+, 2010.

[9] M. C. Schatz, B. Langmead, and S. L. Salzberg. Cloud computing and the DNA data race. *Nature Biotechnology*, 28(7):691–693, 2010.

[10] A. Thrasher, R. Carmichael, P. Bui, L. Yu, D. Thain, and S. Emrich. Taming complex bioinformatics workflows with Weaver, Makeflow, and Starch. In *In Proceedings of 5th Workshop of Workflows in Support of Large-Scale Science 2010*, 2010.

[11] O. Trelles. On the parallelisation of bioinformatics applications. *Briefings in Bioinformatics*, 2(2):181–219, 2001.

[12] L. Yu, C. Moretti, A. Thrasher, S. J. Emrich, K. Judd, and D. Thain. Harnessing parallelism in multicore clusters with the All-Pairs, Wavefront, and Makeflow abstractions. *Cluster Computing*, 13(3):243–256, 2010.