MAKER as a Service: Moving HPC applications to Jetstream Cloud

Nicholas Hazekamp*, Upendra Kumar Devisetty‡, Nirav Merchant†, and Douglas Thain*
*Department of Computer Science and Engineering, University of Notre Dame
Notre Dame, Indiana, United States of America
Email: {nhazekam,dthain}@nd.edu
‡CyVerse, The University of Arizona
Tucson, Arizona, United States of America
Email: {upendra}@cyverse.org
†The University of Arizona
Tucson, Arizona, United States of America
Email: {nirav}@email.arizona.edu

***Abstract*—As cloud resources become more available as an execution platform, the need to transition applications between HPC and the cloud becomes a necessity. However, because of the complex setup and system specific demands of these applications, transition is difficult and may not scale as desired. Jetstream is a NSF funded cloud service that is aiming to provide these services for users in a dynamical allocated nature. In this work we look at three key areas to focus on when transitioning between resources: providing a portable reproducible environment, scaling between local and remote resources, and using feedback to the user for informing configuration and runtime decisions. Building on the MAKER bioinformatic application, we have deployed WQ-MAKER on the Jetstream cloud platform, helping to annotate over 30 genomes and accelerating performance from days to hours and weeks to days.**

***Index Terms*—Bioinformatics, Workflows, Scalability**

I. Introduction

Today's researcher has access to a large number of computing resources, including multiple specialized high performance machines (HPC), general purpose commodity clusters, and public cloud services. Many users desire the flexibility to move their applications between platforms so as to take advantage of the best cost and performance available to them. This specialization makes traditional HPC applications difficult to move between environments.

Historically, HPC applications have been built in isolated environments supported by professional system administration staff. In order to extract the maximum possible performance from specialized hardware, application creators have relied on custom software stacks, tuned code to rely on high performance network interconnects, and designed I/O behaviors to exploit sophisticated high performance filesystems. As a result, applications often become dependent upon the specific details of the environment in which they were created. Getting an application to run in a new environment is challenging and once running, may not be optimized or configured similarly for the hardware.

As a case study to investigate this problem, we consider the MAKER [1] bioinformatics pipeline. MAKER has a large number of software dependencies that must be installed, limited scalability in high latency environments, and can produce configuration and runtime errors that are difficult to diagnose, all of which are common traits of HPC applications.

In our experience of migrating MAKER, we determined three goals for smooth transition between platforms.

- Portable reproducible environment for HPC, Cloud, and user resources, targeting support with user permissions.
- Ability to leverage resources (threads/MPI) on local and remote resources (multiple non-contiguous machines).
- Provide feedback for scalable and dynamic system, to aid in configuration and runtime decisions.

In order to achieve these goals we used VC3 [2] to build MAKER on site, instead relying on an existing machine or container images with variable availability, to provide a portable reproducible environment. WQ-MAKER, a combination of Work Queue and MPI, was used to leverage local concurrency across a set of distributed resources at different sites, i.e. Jetstream, Condor, and locally. In using VC3 and writing WQ-MAKER, care was taken to determine the origin of errors, direct users on fixing them, and providing runtime stats on performance.

Using these techniques we were able to launch MAKER on multiple platforms, including on the NSF Jetstream cloud facility where WQ-MAKER is currently in production use for bioinformatics research. As more users build the provided software and pipeline, we have been able to leverage available resources in Jetstream to accelerate performance of the analysis of over 30 genomes.

II. Background

A. MAKER

MAKER is a bioinformatic pipeline used to annotate genomic information. MAKER utilizes standard programs in bioinformatics to customize the processing and preparation of the raw data. This includes processes to identify repeats, align ESTs and proteins to a target genomes, predict genes,

and quantify the quality of the results based on the provided evidence. MAKER focuses on automating the entire annotation process to create an easy and consistent initial annotation. MAKER is still under active development and is used in many areas of organism modeling. It can be deployed as a sequential, multicore, or MPI application, depending on the available resources. As a side-effect of utilizing a number of different tools in the pipeline, files are often used as intermediaries on top of the data structures passed by MPI, leading to reliance on shared filesystems on multi-machine MPI runs.

B. Jetstream

Jetstream [3], [4] is a NSF funded cloud service built on OpenStack. Operates similarly to Amazon EC2 and has support for data transfer and storage. Allows users created images to provide consistent platforms for review, comparison, and verification of results. One of Jetstream's goals is to provide a service that focuses on usability and support. As a cloud service, Jetstream is able to create and host custom images and environments that are more difficult to deliver on a more traditional HPC service.

C. Work Queue

Work Queue [5] is a master-worker framework that provides an API for creating tasks and submitting them to a heterogeneous pool of workers. Workers are started as stand-alone processes either submitted to a batch system or run locally, creating a diverse pool of resources. Work Queue makes the assumption that there is no shared filesystem and requires files to be specified for transfer. With the provided inputs, tasks are distributed to workers based on matching the required resources. Theses tasks are placed in a local sandbox and executed with a specified command. Upon completion the output files are collected and returned to the master. Using workers as a pool of resources allows for dynamic addition and removal of workers to scale to execution needs.

Assuming no shared filesystem, each worker creates a local cache to limit repeatedly transferring files that are shared by multiple tasks. This is useful in applications with common reference files, such as many bioinformatics workflows. Additionally, the assumption of no shared filesystem for a cloud environment where files must be transferred between instances for collaboration.

III. Portable Reproducible Environments

Creating and supporting a reproducible environment is a current research topic of relevance with working being done at the platform level of OpenStack and Amazon, the container level by Docker and Singularity, and the deployment level of Jenkins, and Ansible. These three different levels each provide a different way of creating a reproducible environment. As part of this work, we targeted Jetstream, but also our Condor and SGE clusters, as well as user machines. A key consideration was the user's ability to verify a setup and configuration locally prior to moving to larger, possibly costly, resources.

A. Machine Images

A machine image is a pre-built snapshot of a desired software stack. Machine images can come in a variety of formats and are supported by a variable number of platforms, such as OpenStack. Machine images are ideal when working on a singleoperating system (OS) and platform as a base to provide consistent low level integration. However, outside of the scope of a single operation system, images have less portability. This reduced portability requires a developer to maintain machine images for each supported platform. This also precludes using the image at HPC facilities that lack user-level integration with machine images. The machine image would be an ideal target were we not also targeting users without access to systems like Jetstream.

B. Container Images

A container image is, similar to a machine image, a snapshot of a desired software stack. Container technology allows for users to run a container image on a supporting site using programs such as Docker, Singularity [6], and Charliecloud [7]. Container images provide an portability at a higher level than machine images, by running on any system that supports them. This allows for container images of variable OS to run on any supporting resource. Containers are also now beginning to be supported at HPC centers, such Singularity on a number of XSEDE resources.

However, in the case of both Docker and Singularity, super-user privileges are required for installing the software. Therefore leaving systems such as campus resources and local clusters unavailable. Charliecloud, assuming unprivileged user name spaces are enabled in the kernel, does provides a user-level container system. Unfortunately, not all kernels have this enabled by default, and availability varies between resources.

C. Deployment Services

In contrast with images, a deployment services install and organizes independent software packages into a single coherent package. This often includes finding either source code or pre-compiled binaries that are compatible, installing them, and configuring different software packages together. Examples of deployment services are as simple as apt-get and make, up to automated systems such as Ansible, Spack, Homebrew, and server-level orchestration tools such as Jenkins and Puppet.

In contrast with machine images, deployment services are often lightweight and only require a small number of predetermined packages to be installed. This allows for a high level of flexibility when deploying in a diverse set of environments and onto different platforms. While some cloud platforms may offer interoperability due to an underlying OpenStack framework, most platforms will require machine images to be recreated. Using a deployment services alleviates this by adapting to the current system and using generic build information from source where necessary. Deployment services adapt well to changes in versions and allow a user to customize these on the fly and test out different configurations.

Deployment services help to codify required build steps, and when written with multiple OSes in mind can reduce the work of supporting different platforms. However, with this flexibility comes the cost of building the software at each site for each use. Additionally, builds often rely on a remote data such as git repositories or the software's host site, as in the case of MAKER. The large variance in power and scope of these tools results in a number of different situations where super-user privileges may or not be needed.

D. VC3

As mentioned previously, several platforms were targeted including Jetstream, a Condor pool, a SGE cluster, and individual machines. As a result, we targeted a deployment services to allow for flexibility on both the OS and permissions. Some sites, such as local clusters, neither had the required tools installed nor allowed user-level installation. Targeting user-level permissions and OS agnostic features provides flexibility to target users' available resources.

VC3 [2] was used to install and configure MAKER. VC3 creates a sub-shell with a self-contained environment, and organizes software in a consistent, predictable manner. VC3 is based upon the idea of tool recipes, with inspiration taken from NixOS [8]. Each tool description consists of a recipe, dependencies, version, and environment variables.

VC3 has several features ideal for MAKER. The consistent file structure and referencing is important as some MAKER dependencies rely on hard-coded paths and strict relative locations. This allowed for resources to come from a number of configurations with the same structure. VC3 also requires only user level permissions, allowing the portability to any linux platform. VC3's interface allowed invocation of MAKER and organization of input data to be consistent between systems. Though there are other tools that could perform similarly, VC3 was picked for it flexibility, unprivileged operation, and familiarity. Similar solutions were written using Ansible, though this method was only used on Jetstream.

E. Deploying MAKER

When deployed onto Jetstream, MAKER was installed using deployment services to consistently handle the complex setup. VC3 and Ansible were both used for consistent builds on more widely provided base images, such as Centos 7 Developement. This process included installing several of MAKER's required programs and libraries that cannot be distributed in an automated manner due to developer licensing.

Though deployment services provide more flexibility when installing, a machine image provides easier, faster start-up for the users. To accommodate this, a machine image was built with the VC3 package installed, allowing execution to only verify the MAKER install and not have to build it each time. Additionally, Work Queue workers could be launched from the same machine image limiting traffic to only task input and output, rely on required software to be installed. This differed on our Condor cluster where there was no shared file system or container support. As a result a compressed install of VC3 was sent, so MAKER was built for each OS only once.

A build using deployment services provides a great deal of flexibility, but as users primarily used this just for MAKER, the repeated build overhead limited benefits. This was mitigated by using a machine image with MAKER installed using VC3 on Jetstream, and compressed VC3 on Condor. Using VC3 made rebuilding images, targeting new OS, and adding new features simple. By using a static image based on the deployment services, regardless of machine or container image, we can update software without the user needing to build every run.

IV. Scalability

We define scalability as the number of cores that any one project was able to harness at a time. In an HPC context, this translates to the number of cores by the number of machines that were allocated to your job. MPI, being able to work on distributed memory machines, could work across the boundaries of several machines. In a cloud context, jobs are often limited by the size of selectable images. Some cloud platforms allow for creating sub-networks of machines, but the typical user (a researcher trying to run an analysis) will not have the time nor expertise to configuring them.

Scalability was achieved at two levels in this work.

1) Local parallelism, such as MPI, GPUS, or threads.
2) Distributed concurrency, partitioning across machines.

Our scalability goal was to limit both involvement in the provided software and work needed to target a different concurrency model. As such we decided on using the provided concurrency model of MAKER, MPI, to execute on each worker, where a worker is equivalent to a node. Using MPI allows us to scalably utilize resources, but lacks dynamicity as new resources become available. By leveraging the existing concurrency model we avoid the complexity of interfacing with MAKER's internal architecture. This also allows for smooth transitions between versions of MAKER and any underlying programs. This, based on experience from our previous tightly interfaced work [9], makes transitioning between versions, configurations, and platforms difficult, requiring repeated almost equal effort for each transition.

Relying on the underlying concurrency model for local parallelism lets Work Queue reason about the scaling and distribution of work to all available resources. In contrast with concurrency models like MPI and threads, Work Queue does not rely on having a statically determined set of resources. Orchestrating the work distribution with Work Queue allows users to add workers to increase resource pool, use resources from several allocations or sites, and provides fault tolerance to the application as a whole. Relying on MPI for local scalability, WQ-MAKER uses Work Queue to dynamically schedule on new resources.

A. MAKER's MPI Behavior

MAKER utilizes MPI as the primary means for scalability. Concurrency in bioinformatics is often available at the sequence (contig/scaffold) level. This is a division commonly

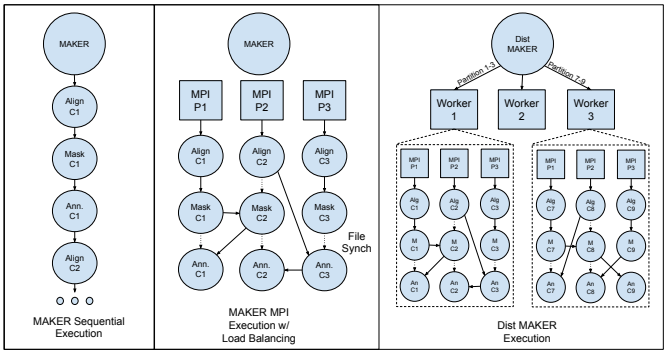

Fig. 1: MAKER, MPI MAKER, and WQ-MAKER models *MAKER, without MPI, runs the sub-process analysis sequentially. MPI MAKER executes by sharing work, with MPI processes going to the pool of ready tasks and executing them. These processes are synchronized using data-structures(in MAKER) and data files(in MAKER's sub-tools) passed between sub-process (see dotted lines). WQ-MAKER partitions the data and sends it to separate workers. Each worker executes MAKER locally using MPI MAKER.*

used for partitioning data, as each sequence is a unique piece of data analyzed separately from the other sequences. MAKER then creates an additional level of concurrency using each analysis tool as a sub-process in the pipeline. This allows the burden of longer running sequences to be shared between multiple cores on the same machine and allows load balancing with smaller computational chunks. However, the secondary level of concurrency can rely on intermediate files, for locks and tool specific data, to exist in shared space between tasks. This is not a requirement of MPI, which discourages this, though some MAKER's sub-processes rely on files for information and state. As a result, execution must also be located in a shared filesystem to allow for the outputs to be coordinated between all MPI processes.

B. WQ-MAKER

WQ-MAKER is built using the Work Queue API. The work is partitioned in different sizes, anywhere from individual sequences to the entire query file. WQ-MAKER does not split the work past the sequence level, as MAKER does with MPI, to prevent communication overhead from sub-processes. Each partition is a self-contained computational chunk that is distributed and organized after completion.

WQ-MAKER utilizes Work Queue's resource interface to allocate resources based on the partitions size and structure. Controlling at the task level allows for handling based on structure, such that long scaffolds are handled differently than short contigs. Using the resources allocated to a task by Work Queue, the worker can assign the appropriate amount of cores for MPI. To do this accurately assign resources a model is being developed as part of future work. Employing MPI on larger task, which occupy the entire worker, limits the master's management burden of monitoring workers.

C. Scaling up vs scaling out

Scaling up helps to accelerate the annotation of genomes, but scaling up is not always the best usage of resources. A common assumption is that it is best to scale up using all of the available resources immediately. However, in practice this is seldom the truth as distribution of shared data (i.e. references), connecting to multiple resources, and spamming batch systems results in a gradual increase in resources, not an immediate deluge. This more gradual availability of ready resources can cause timeouts and under utilization of provided resources. This limitation leads users to under-provision applications instead of gradually adding resources as applications stabilizes. This was not addressed in this work other than to provide runtime feedback to the users about usage, so they are better informed. Work Queue masters track capacity of an application and inform users to add resources as the master can support more.

Compared with scaling up, scaling out can better utilize those resources for concurrent analysis of genomes, allowing for the same pool of resources to be shared, workers are kept busy with tasks from different masters. Work Queue allows for workers to match to multiple masters, enabling WQ-MAKER to share workers between instances. This provides an additional level of load balancing, without relying on additional underlying systems.

V. Exposing execution feedback

A. Clean Environment Builds

The first major obstacle was providing the users with clear feedback on the creation of the MAKER environment. Using deployment services to create, update, or modify the instance can initially cause errors and warnings, but once codified offer consistent builds. These build errors were typically only encountered by developers and could be diagnosed quickly. To communicate this, VC3 and Ansible need to have clear error handling and messages to identify errors. Build errors are mitigated when using static images, but can still be relevant as users want different configurations. Additionally, applications such as MAKER, where a variable number of the subsystems may be used, cursory testing does not always reveal configuration errors. As such new errors can occur when using different sets functionality, and must be clearly differentiated from runtime errors.

When using static images, like machine or container images, build errors are often self-explanatory, such as network errors, insufficient resources, or just bad luck. Jetstream's provided trouble-shooting gives possible solutions for users to attempt.

B. Deploying Workers

When deploying workers, users must log into each worker machine and manually start the worker process. This requires users to manage multiple ssh sessions and any changes to connection information (i.e. IP address or project name) must be reflected to all workers. We use an Ansible-playbook that allows the user to launch and manage workers from the host machine. For the user, this is as easy as creating an ssh-key

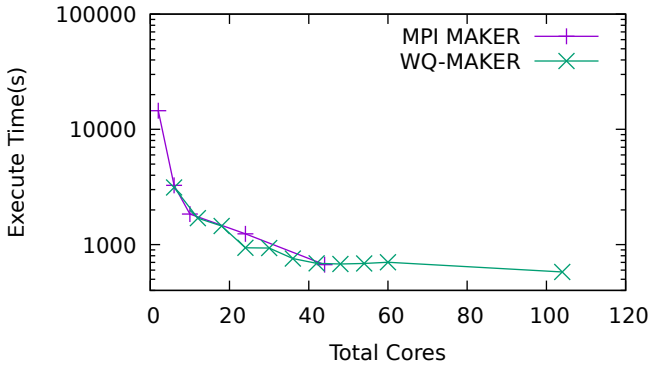

Fig. 2: Comparison on Fungal(41MB) dataset
The Fungal dataset contains 231 contigs. With similarly runtimes, we can see there was little or no overhead when using WQ-MAKER, though little gained beyond 34 cores.

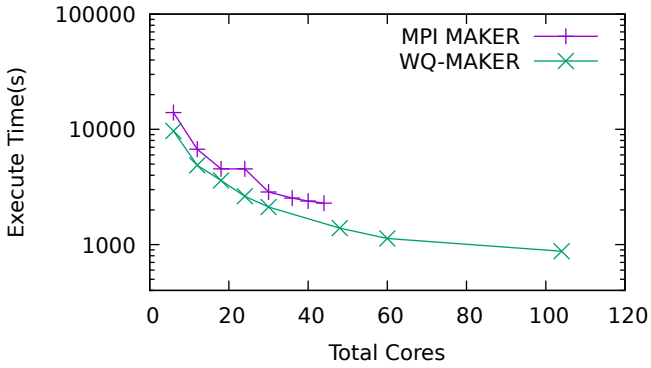

Fig. 3: Comparison on partial Hummingbird(900MB) dataset
This subset of Hummingbird contains 5000 contigs. In this image we can see improvement of WQ-MAKER over the MPI run, likely as a result of reduced contention for resources.

and saving it in Jetstream, allowing the master machine to propagate commands using Ansible. On other systems, such as Condor and SGE, Work Queue maintains a worker factory that can submit workers resources become available.

Part of deploying workers is monitoring how many are actively being used by the Work Queue master. This is done using a status program that queries masters for active workers. As previously mentioned, masters also track capacity and can allow the factory to submit workers as masters are able to support more, as a result of workers being initialized or varying task execution time.

C. Evaluate Performance

Following a successful run, WQ-MAKER verifies successful runs to ensure proper execution. This is done by rectifying the final output files against the input data to ensure that all contigs were analyzed. The produced statistics are examined by WQ-MAKER to understand the behavior on this run's data. Work Queue provides a suite of graphing scripts for more in depth analysis. These graphs help understand the task execution, worker utilization, and file transfer speeds. All Work Queue graphs used in this paper were created using these tools.

D. Diagnosing Errors

Unfortunately, WQ-MAKER does not always run perfectly and it is important to help users diagnose errors and where they originated. After a run completes, WQ-MAKER prints the successes and failure of contigs. WQ-MAKER will retry failed contigs to ensure that it was not intermittent, possibly the result of network issues, software bugs, or resource contention. On repeated failure tasks are logged, reported to the user, and abandoned to avoid wasted effort retrying them further. The output of failed tasks is stored by WQ-MAKER, allowing for users diagnose the issue later.

Work Queue provides a debugging log that can be turned on to diagnose network errors, firewall issues, or file transfer failures. If an error happens while running WQ-MAKER, the Work Queue framework will print the error along with additional information in the debug log.

VI. Evaluation

Though performance is important, this paper did not directly measure the differences in start time between machines images, container images, and deployment services. Using deployment services we provide consistent builds, but leveraged machines images were available in Jetstream. The time needed to build the software is relevant when redeploying using deployment services. VC3 allows for multi-threaded deployment, which reduces a 1 hour build to roughly 10 minutes using between 16 and 24 cores. VC3 reuses existing builds allowing us to re-enter an existing build consistently in under a minute. The consistent file structure of VC3 allows us to build once and distributed to workers to install if needed.

WQ-MAKER was evaluated using several datasets. The MPI executions were done using differently sized Jetstream instances, up to the largest of 44 cores. The WQ-MAKER executions were done using a master and a variable number of workers on medium instances, 6 cores. The fungal data set consists of 231 contigs. This data set is executed in roughly 4 hours using 2 cores locally. With increased cores, WQ-MAKER performance scales with MPI. Considering that fungal dataset is small, there is limited improvement after 40 cores, with a slight increase at 104 cores, as seen in Figure 2.

The hummingbird genome consisting of 5000 contigs, a medium sized genome sample. The results of running this genome through MPI and WQ-MAKER can be seen in Figure 3. For this larger sample we were able see improved performance over the standard MPI deployment as a result of lessening the memory burden on each partition using several workers. We were also able to see consistent reduction of execution time as we increase resources.

The saguaro cactus dataset consists of 573771 contigs This dataset took 57 hours to run using MPI MAKER on 24 cores.

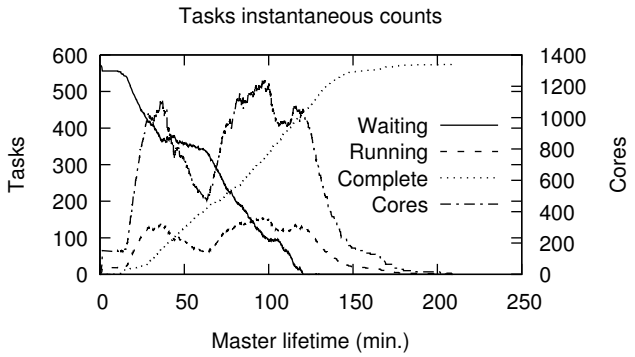

Fig. 4: Saguaro Cactus(1.6GB) with MAKER on Condor
The two lines of note are the running tasks and cores. The running tasks indicate the number of actively running MAKER tasks. The cores indicate the cores utilized by WQ-MAKER. Condor's volatility as a job scavenging system causes the variability in available resources.

	Execution Time	Cores	Total CPU Hours	Speedup
MAKER*	52 days	1	—	—
MPI MAKER	57 hrs	24	1368	—
WQ-MAKER	3.6 hrs	80-1344	1725.5	15.8

Fig. 5: Overall Performance Improvement
**MAKER's execution is estimated using the CPU time of MPI MAKER. The results from MPI MAKER and WQ-MAKER are actual runs. WQ-MAKER performed ∼16x faster than MPI MAKER using 26% more CPU hours.*

The raw CPU time spent during this job was 52 days of computation. Using WQ-MAKER, we ran this same dataset on our Condor cluster. Using Work Queue factory a mix of workers were launched. Each task was partitioned into 100 sequence, 8 core jobs to allow them to fit in the Condor instances. Figure 4 shows the number of tasks running over time as the workflow executed. The final execution time was 3 hours and 36 minutes, running 168 tasks concurrently at its peak, which equates to 1344 cores. As workers were dynamically added and removed the total CPU hours was only 1725.5, a result of WQ-MAKER's dynamic nature.

In total this project has been used by a number of users for annotation. We are actively developing and improving WQ-MAKER and working with users to better understand their needs. Table I shows a subset of the genomes that have been annotated using WQ-MAKER.

VII. Related Work

A common consideration when looking for a way to build a reproducible environment is to use container technologies such as Docker [10] and Singularity [11]. Both Docker and Singularity provide a means of creating reproducible execution environments. These containers can differ from the execution platform even at the OS level. Unfortunately, super-user privilege are required for the underlying daemon for Docker and image creation in Singularity. Similarly there are several automation systems [12], [13] that rely on system configuration to provide consistent environments. VC3 [2], does not replicate the original OS but builds the tools on the current system, allowing VC3 to operate at the user-level. The goal of VC3 is not to provide the exact execution environment as with containers, modules, etc. but to provide research tools without privileged operations.

This work builds on the work originally developed in [9]. There are several major difference between these works. The original work replaced MAKER start program, and actively used the internal data structure of MAKER. Versions changes caused frequent errors, and site migration became impossible. Other genomic work using Work Queue also includes SAND [14] which looks at genome assembly. Though Work Queue was utilized for this paper, there are several other scalable solutions, such as Pegasus [15], Swift [16], and Spark [17].

Genome	Sequences	Workers	Runtime (hrs)
Sporobolus A	11,789 contigs	22-40	144
Sporobolus B	6,615 contigs	21-35	108
Brassica rapa	44,000 scaffolds	10	4
Zea mays W22	10 chromosomes	10	1440
Zea mays nc350	6460 scaffolds	22	72
Culex tarsalis	7478 scaffolds	40	120
MP29	5003 scaffolds	40	24
Pigweed	4126 scaffolds	20	120
Sclerotiana homeocarpa iso 10	231 contigs	10	6
Sclerotiana homeocarpa iso 11	257 contigs	10	6
Calypte anna	265 super-scaffolds	10	8
Kochia scoparia	19,671 scaffolds	21	72

TABLE I: Genomes sequenced on Jetstream: This is non-exhaustive list of genomes annoted using WQ-MAKER on Jestream.

VIII. Conclusion

Deploying MAKER on the Jetstream has helped us to better understand reliable deployment, scalability, and what is useful feedback. We generated a consistent build using machine images built from VC3. We provided dynamic scalability by using Work Queue to distribute partitions, and running existing MPI parallelism at the worker. WQ-MAKER users leveraged runtime data to understand performance, improve scaling, and diagnose errors. WQ-MAKER has been used for over 30 genomes and can accelerate annotation time using a variety of resources.

IX. Acknowledgments and Availability

WQ-MAKER, Work Queue, and several other project are available from the Cooperative Computing Lab and deployed as CCTools, at `http://ccl.cse.nd.edu/`. `https://github.com/cooperative-computing-lab/cctools`

REFERENCES

[1] M. S. Campbell, C. Holt, B. Moore, and M. Yandell, "Genome Annotation and Curation Using MAKER and MAKER-P," *Curr Protoc Bioinformatics*, vol. 48, pp. 1–39, Dec 2014.

[2] B. Tovar, N. Hazekamp, N. Kremer-Herman, and D. Thain, "Automatic dependency management for scientific applications on clusters," in *International Conference on Cloud Engineering (IC2E)*, 2018.

[3] C. A. Stewart, T. M. Cockerill, I. Foster, D. Hancock, N. Merchant, E. Skidmore, D. Stanzione, J. Taylor, S. Tuecke, G. Turner, M. Vaughn, and N. I. Gaffney, "Jetstream: A self-provisioned, scalable science and engineering cloud environment," in *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, ser. XSEDE '15. New York, NY, USA: ACM, 2015, pp. 29:1–29:8. [Online]. Available: http://doi.acm.org/10.1145/2792745.2792774

[4] 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, "Xsede: Accelerating scientific discovery," *Computing in Science Engineering*, vol. 16, no. 5, pp. 62–74, Sept 2014.

[5] 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.

[6] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PLOS ONE*, vol. 12, no. 5, pp. 1–20, 05 2017. [Online]. Available: https://doi.org/10.1371/journal.pone.0177459

[7] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in hpc," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 36:1–36:10. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126925

[8] E. Dolstra, A. LÖh, and N. Pierron, "Nixos: A purely functional linux distribution," *J. Funct. Program.*, vol. 20, no. 5-6, pp. 577–615, Nov. 2010. [Online]. Available: http://dx.doi.org/10.1017/S0956796810000195

[9] A. Thrasher, Z. Musgrave, B. Kachmark, D. Thain, and S. Emrich, "Scaling Up Genome Annotation with MAKER and Work Queue," *International Journal of Bioinformatics Research and Applications*, vol. 10, no. 4-5, pp. 447–460, 2014.

[10] 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

[11] G. M. Kurtzer, "Singularity 2.1.2 - Linux application and environment containers for science," Aug. 2016. [Online]. Available: https://doi.org/10.5281/zenodo.60736

[12] I. Redhat. (2012) Ansible. [Online]. Available: https://www.ansible.com/

[13] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The spack package manager: Bringing order to hpc software chaos," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 40:1–40:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807623

[14] C. Moretti, A. Thrasher, L. Yu, M. Olson, S. Emrich, and D. Thain, "A Framework for Scalable Genome Assembly on Clusters, Clouds, and Grids," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, 2012.

[15] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman, "Pegasus: Planning for execution in grids," GriPhyN, Tech. Rep. Technical Report 2002-20, 2002. [Online]. Available: http://pegasus.isi.edu/publications/ewa/pegasus_overview.pdf

[16] 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.

[17] 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