Distributed Task Scheduling on Cloud Infrastructure for Bioinformatics Workflows and Analysis

Rick Morrow

A thesis

submitted in partial fulfillment of the

requirements for the

degree of

Master of Science

University of Washington
2023

Committee:
Ling-Hong Hung, Chair
Ka Yee Yeung

Program Authorized to Offer Degree:
Computer Science and Systems

©Copyright 2023

Rick Morrow

University of Washington Tacoma

Abstract

Efficient Distributed Task Scheduling for Bioinformatics Workflows and Analysis

Rick Morrow

Chair of the Supervisory Committee
Professor Ling-Hong Hung, School of Engineering and Technology

The datasets analyzed in bioinformatics are large and numerous, requiring complex analysis. The size and complexity of bioinformatics data often makes it impractical for researchers to run analytical workflows on their personal laptops or PCs. Bioinformatics jobs, however, can benefit from greatly reduced compute times when parallelized across multiple CPUs or cores. Historically, researchers would run analytical workflows on local or static HPC clusters using batch schedulers like Slurm. Running jobs on HPC clusters can be complicated, as jobs must be created and using scheduler specific scripts. HPC clusters are also typically a shared resource with fixed scalability and the main purpose of schedulers was to queue requests and provide resources when available . Cloud computing presents a cost-effective, scalable, reproducible and on-demand resource for researchers to extend the computing resources available to them without the overhead of acquiring and maintaining on-site infrastructure. Although the cloud provides elastic and scalable resources, there remains the challenge of effectively utilizing cloud computing resources through efficient task scheduling. Specifically, there needs to be some method of queueing and scheduling tasks, and assigning those tasks to available workers. In this project we build a task scheduler that handles bioinformatics workflows, which are split into atomic tasks that can be run in parallel, distributed across an arbitrary mix of computing resources from local machines to cloud resources of arbitrary size. The tasks themselves will be processed by containerized workers that implement a standardized and reproducible execution environment. We assess the performance of our scheduler using a real-world bioinformatics task which aligns a set of short sequences (reads) to a human reference sequence using the Burrows-Wheeler Aligner (BWA). We benchmark and compare our scheduling methodology against sequential processing of bioinformatics data using a Dockerized Burrows-Wheeler Aligner (BWA) and against a script that processes this data in parallel without containerization.

Acknowledgements

I would like to thank Dr. Ling-Hong Hung and Dr. Ka Yee Yeung for giving me opportunities to work with Biodepot's workflow builder both as an intern, and with this project. I was, and am thrilled that this work might live beyond this paper and contribute something useful to bioinformatics research. I would especially like to thank Dr. Hung for insightful advice, and incredible patience and encouragement throughout this project.

I would also like to thank Amy Senesac and Quinn Morrow for their for putting up with my regular absence while I made career transitions and worked through UW-Tacoma's MSCCS program. I could not have been successful without their understanding and support.

Contents

1 Introduction **3**
1.1 Motivation 4
1.2 Biodepot's Workflow Builder 6
2 Related Work **7**
2.2 Existing Bwb Parallel Processing 9
2.3 Problem Description 10
3 Implementation **11**
3.1 Technology Stack 11
3.2 Scheduler Architecture 13
3.3 Using the Scheduler 15
4 Benchmarking **18**
4.1 Dataset 18
4.2 Set up 18
4.3 Scheduler Performance 19
4.3.1 Comparing the scheduler to shell scripts on a single instance 20
4.3.2 Scheduler performance on multiple instances 24
4.4 Cost Estimates 27
5 Future Work **31**
6 Conclusions **33**
References **35**

1 Introduction

Analytical workflows used in scientific research are often complex sets of multi-stage processing steps where data needs to be shared by multiple processors and results need to be aggregated. Scientists have typically either executed workflows on their own machines or have used scientific schedulers on local computer clusters or supercomputers to execute workflows with parallel processing. With the increasing ease of individual and institutional access to cloud computing resources and the standardization of execution environments provided by containerization, it's becoming more common for researchers to implement distributed, containerized workflows in the cloud. Using cloud based execution environments has brought infrastructure and deployment more under the control of individual researchers, data scientists, and programmers. With containerized and serverless functions researchers can create reproducible, scalable, and modular analytical workflow execution environments without having to rely on hardware engineers, sysadmins (or sometimes computer science students) to compile software and create scheduling scripts to deploy analytical workflows on local or managed HPC clusters. The flexibility and reproducibility (and shareability) of containerized and serverless execution environments on cloud computing resources means that a researcher with a budget and an account can deploy workflows on their own time with resources appropriate to their budget – a big shift from needing to arrange time for access to HPC clusters and possibly assistance from sysadmins or other experts.

We will focus on implementing and benchmarking a containerized gene sequencing workflow using a centralized task broker and distributed parallel workers for Biodepot's Workflow Builder (Bwb). Bwb is a software suite that enables biomedical researchers to create workflows using an interactive, graphical interface and execute them with containerized analytical steps built by Biodepot [1]. Bwb's containerized architecture makes it easy to run workflows with parallelized processing whether the execution is local, cloud-based or on computer clusters.

1.1 Motivation

The bioinformatics analysis workflows and datasets that biomedical researchers use often require more storage and processing power than is available on their personal laptops and PCs. Researchers need access to infrastructure that provides large numbers of CPUs, large amounts of RAM and high capacity storage. Cloud based solutions are beginning to overtake HPC clusters and supercomputers as an infrastructure of choice for biomedical researchers looking to parallelize their research over large numbers of processes. Our research aims to provide another solution in this space that differentiates itself by being infrastructure agnostic in that it can be deployed across a mix of local and cloud resources, modular enough to be decoupled from any specific front-end implementation and easily extensible to meet evolving workflows and cloud resources. Additionally, the architecture will be modeled after asynchronous execution with microservices rather than based on HPC batch scheduling.

Biomedical researchers face increasing challenges running bioinformatics analysis workflows as datasets become larger and require more processing power to complete. DNA and RNA sequencing datasets are often in the mid-gigabyte range in size and proteomics and imaging datasets can be many terabytes in size. Experiments usually require multiple datasets to account for different treatment conditions and to provide statistically significant results. Because biomedical workflows are data intensive and processor intensive, it is less feasible to run them on individual PC's – there are simply not enough processors available on a laptop or workstation to run these analyses in reasonable time frames. Workflows can still be run on large servers, clusters or other static resources, but this is inefficient unless the resources are shared between enough researchers to be cost efficient. With Moore's law no longer holding true [2] – chip densities are still increasing, but no longer double every year or two years – cloud computing will be increasingly relied upon as a resource with on-demand scalability. Accessible methods of executing workflows in the cloud with resource provisioning that matches the constraints of research budgets will be the solutions that meet most researcher's needs.

Bioinformatics analysis is further complicated by the necessity of creating complex multi-stage workflows. Some of these analytical steps, such as the optimal alignment of a sequence to the human

genome or machine learning AI approaches to image processing can require access to robust computational resources over long execution times. As datasets get larger, workflows more complex and machine learning techniques become more advanced it becomes much less feasible for researchers to run analytical workflows on their personal machines where execution times could range from many hours to many days.

Biomedical researchers may desire more direct control over the infrastructure they execute analytical workflows on than they would typically get using HPC clusters or supercomputers. These infrastructures require scheduler specific job scripts and often cluster specific compilation of executables, which makes implementation of workflows not easily portable. HPC/supercomputer infrastructure is also usually fixed making it less flexible and scalable than cloud resources. An emerging approach to build containerized solutions that execute workflows on the cloud. Containerized solutions allow researchers to have tight control over execution environments, and enable portable solutions. Containers also facilitate a reproducible implementation of the different executables and scripts with cloud-provided on-demand computational resources. Various software platforms that execute bioinformatics workflows inside software containers on public cloud providers like Amazon Web Services, Google Cloud, and Microsoft Azure. Most of these current platforms, however, are modeled after batch execution on dedicated high-performance clusters.

These platforms can be difficult for biomedical researchers, who are used to graphical and interactive tools, to use. The technical challenges these platforms create for researchers means there is space for the development of more user-friendly bioinformatics analysis tools. One such platform is BioDepot's workflow builder (Bwb), which makes the installation and launching of genomics and image analysis workflows easier for researchers without a background in computer science.

1.2 Biodepot's Workflow Builder

Biodepot's Workflow Builder (Bwb) provides a self-installing, interactive graphical front-end [1]. Researchers can download analytical workflows from Biodepot's GitHub repositories, which execute container images pre-configured to run workflow steps and which are available on Biodepot's DockerHub

repository. Bwb's front end can be run on a local machine or on the cloud to perform the same analytical workflows using an identical front-end interface. This makes Bwb easy for non-technical researchers, who are used to using personal machines, to adopt. Workflows are typically executed where the Bwb front end is executed, so when Bwb is run on a cloud instance, that instance also runs the workflows. In this way, using larger instances allows researchers to benefit from specialized hardware available in the cloud such as GPUs. However, in order to take full advantage of the elastic and scalable capabilities of the cloud, workflow execution needs to be parallelized and distributed over as many CPU cores over as many instances as the analysis requires. With a distributed job scheduler, users could split up the analyses of terabyte size files into smaller tasks that can be queued and processed in parallel. Combined with Bwb's graphical interface and easily available workflows, biomedical researchers could fully leverage the cloud to reduce the time and increase the accuracy of their data analyses.

Biodepot had an existing task scheduling prototype built using Python and bash scripting that is used as the basis for implementing the solution developed as part of this thesis. A primary goal of the implementation was to create a bioinformatics task scheduler built with commonly used and well maintained frameworks and libraries so that the scheduler will be easy to maintain and extend by future students and engineers who work on Biodepot's software suite, and also easy to use by biomedical researchers.

Our implementation is based on microservice design principles where workflow generation and infrastructure management is abstracted, assuming that Bwb will generate workflows, aggregate and process workflow results, and will also do any programmatic infrastructure management that is required to run the scheduler. Our concern will be to remain focused on implementing a scheduling solution that is modular, front-end agnostic following contemporary microservice design principles [3], [4].

2 Related Work

Scientific analytical workflows that require more computing power than individual PCs could provide have, historically, been run on High Performance Computing (HPC) clusters. These clusters would typically

be static resources, implemented on-site at an institutional level because of the high costs to acquire and maintain them. In these environments, workflows were typically implemented using single executables compiled specifically to run on the specific cluster or supercomputer available, with workflows being specified in cluster-specific scheduler scripts. These batch scheduling scripts assume a computing environment where the resources are being shared by numerous users, all running collections of tasks commonly referred to as jobs. Scheduling scripts would often require the user to specify needed resources so that jobs could be scheduled fairly, and might not execute if the resources were incorrectly specified. This environment was one in which researchers might need to enlist expert help in compiling the executables, writing their workflow scripts and scheduling time to have their jobs added to the batch schedulers.

The Sun Grid Engine [5], and its later variants, Torque/Maui [6], and Slurm [7] are popular examples. These engines have a scripting language to control the submission of queued jobs and are designed for a large static cluster, such as those found in supercomputing facilities. Amazon batch [8] is a similarly structured scheduler designed for a cluster of cloud instances and is an example of attempting to apply these strategies to cloud resources, which are typically far more elastic than on-site infrastructure.

As cloud resources became more readily available, engineers began developing distributed task scheduling, with the goal of being able to take advantage of the cloud's elasticity and scalability. Infrastructure could be optimized to workloads with resources being able to be allocated and deallocated more dynamically. Apache's Hadoop [9] was originally designed to be deployed on clusters created from networked commercial hardware and utilizes master nodes which contain task and job trackers and data nodes, and slave nodes which contain a task tracker and a data node. Data is split into blocks and distributed throughout nodes in the cluster using the Hadoop Distributed File System (HDFS) and jobs are sent to nodes as packaged code to be executed in parallel using MapReduce [10] model for redundancy using a scatter-gather algorithm and managing the execution of processes near the nodes where data is located. Later versions of Hadoop were developed for deployment on cloud resources and MapReduce was replaced by YARN, a resource manager optimized for Hadoop. In the data science world, Apache Spark [11] is a popular platform for distributing computing which has not really caught on in the scientific HPC world,

possibly due to early problems with the Hadoop file system and the relatively greater complexity of bioinformatics workflows.

The cloud's elasticity and scalability also created possibilities for rapid deployment of reproducible execution environments. Containerized deployment with Docker and container orchestration with platforms like Google's Kubernetes [12] became increasingly popular ways to take advantage of these possibilities. Kubernetes has become very popular for distributed cloud computing because of its ability to dynamically scale workers as needed and its tight integration with software containers. Kubernetes, however, is designed specifically to support containerized microservices models with container management (pods), networking, and integrations with logging and monitoring. This differs from typical HPC workload managers, which focus on optimizing high throughput and dynamically managing task access to available resources. Orchestrating Kubernetes clusters can be technically challenging and requires some specialized DevOps knowledge, which means that without a 'plug and play' configuration Kubernetes implementations may not be feasible for biomedical researchers without a computer science background. Additionally, most bioinformatics tasks are loosely coupled, if not embarrassingly parallel so the auto-repair and auto-scaling of Kubernetes aren't as big of a concern as they are with more tightly-coupled tasks where a poorly executed or failed task can affect the performance of many dependent tasks.

In the bioinformatics world several platforms (e.g. Toil and Terra) have been created that use scripting languages such as Workflow Description Language (WDL) [13] and Common Workflow Language (CWL) [14] to create and execute workflows on cloud resources. Scripting languages can be used alongside Docker-Compose for orchestrating containerized execution environments. These implementations use existing schedulers like the previously mentioned Slurm and YARN, or Kube Scheduler [15]. Cloud Service Providers are also developing their own versions of these tools like AWS's Managed Kubernetes [16] and its own implementation of Slurm [17].

One example is the Genomic Institute at the University of California, Santa Cruz's Toil [18]. Toil is an open-source platform written in Python, which uses both WDL and CWL scripts to execute bioinformatics pipelines in the cloud with a batch scheduling framework with some management of cloud resources. Notably, it is complicated to set up and lacks a GUI for creating workflows. Another example is the Broad

Institute's Terra platform [19]. Terra is based around generating and executing Workflow Description Language (WDL) scripts. Terra manages the provisioning of Google Cloud Platform [20] services and can manage containerized execution using Docker images hosted on DockerHub. Terra also uses cloud storage buckets and configurable VMs as part of its workflow execution strategies, but it does not have any built-in parallelization support.

A significant disadvantage of implementations using intermediary scripting languages and managed cloud provisioning is that they may not allow interactive control over workflow execution. This differs from Bwb which provides a GUI for users to build and interact with workflows without intermediary scripts. Bwb provides sequences of tasks directly to execution environments after they are built and modified visually using its GUI.

2.2 Existing Bwb Parallel Processing

Bwb's Workflow Builder operates by translating front end input (usually through its GUI) into Docker commands which are then executed in containers launched from images that contain the needed scripts or binaries to process bioinformatic analyses. It already has a simple scheduler written in Bash that queues Docker commands (e.g. multiple containers) to be run in parallel. The user can specify the number of processes, and the RAM required for those processes. This is limited, however, to the threads, cores, CPUs or vCPUs available on the local machine or the cloud instance that Bwb is executed on. Bwb does have an early, partial prototype distributed task scheduler that is built in Python using the Flask API micro framework [21] and OpenAPI (now Swagger) [22]. This prototype was built several years ago and has limited functionality and a minimal API. It served as a rough guide to the general structure of the scheduler we implement.

2.3 Problem Description

Our goal is to implement a task scheduler that is ultimately platform agnostic, portable, and decoupled from the Bwb application. Basic criteria for success will be that the scheduler utilizes an API to receive

standardized requests that define tasks, then publishes those tasks to a queue or broker so that locally and remotely networked workers can process those tasks. Using an API for the scheduler means that callers do not need to be any specific application or GUI, but any caller which sends an appropriately formed request. Requests should be in the form of a list of atomic tasks that can be completed by an individual worker. A task will contain an individual task ID for tracking and logging, the cloud storage locations of necessary files, local directory structures where remote files may be stored, and result files be written, a container image that will provide the execution time environment for the executable, which will perform bioinformatics analyses and a cloud storage location where results files will be transferred on completion. The scheduler should be implemented following modularized, microservice design principles using container architecture so that individual components may be hosted on cloud instances, laptops and PCs, serverless functions or even in managed clusters like Kubernetes.

The scheduler should also be implemented with technology that is maintainable and extensible, as Bwb's Workflow Builder is developed and maintained by a mixture of academics, software developers, interns, graduate and undergraduate students with both software engineering and bioinformatics backgrounds. Hence, the scheduler should be assembled from popular, maintained libraries and tech stacks that have strong open source communities, rather than relying on a fully customized solution. The open-source end-product should include the following: a lightweight API, a task scheduling library that can handle both atomic, parallel tasks and grouping of tasks where there may be the need for some sequential or synchronous processing, a means of interacting with running jobs via a GUI, and a means of centrally aggregating logs.

3 Implementation

The Biodepot Workflow Builder (Bwb) has been developed and maintained by UW Tacoma faculty including Dr. Ling-Hong Hung, and rotating computer science, data science and bio-science students and interns. Prior to this project, I had the opportunity to work as an intern on components of the Bwb, including a web-based UI and backend that allows users to select AWS instance types, estimate costs and then launch

instances from pre-configured AMI images. I learned that it was important to implement solutions in ways that make it easier for the next interns or students to work with. In that respect, one of our qualifications for success beyond performance details was that our scheduler be implemented as simply as possible and without introducing too much technical debt so that it has some longevity. For that reason, we will spend some time discussing the technologies we chose, and the architecture of our implementation.

3.1 Technology Stack

We chose Python and the primary language for our bioinformatics task scheduler because it continues to grow in popularity, and enables multiple design patterns from scripting to object oriented and functional design strategies. It also has numerous open source libraries that will support rapid implementation of projects now and in the future, which makes it a good choice to meet our maintainability and extensibility requirements. Cloud platforms like Amazon Web Services (AWS), Google Cloud Platform (GCP) and Microsoft Azure also provide well documented SDKs for interacting with cloud based resources through Python code. These SDKs are similar enough that refactoring code from one SDK to another is not too complex. Additionally, it's common for data scientists and other scientists who are not software engineers, but rely on software solutions to be familiar with Python because it has many popular data science and machine learning frameworks.

For our the API, we chose the FastAPI framework [23] because it is lighter weight than other popular frameworks like Django and Flask but comes packaged with some useful inclusions like Pydantic models [24] for data validation and OpenAPI (formerly Swagger [22] for automatic endpoint documentation and testing through a web-based UI. FastAPI enables rapid implementation of lightweight APIs, but can be extended to create complex applications as needed. As part of our modularity requirement, our implementation is packaged with a Dockerfile using the slim-buster Python 3.9.9 as the execution environment.

We chose the Celery framework [25] for our task management framework because it is designed for large scale, asynchronous task scheduling in distributed applications. It is stable enough and mature enough that it sees use in enterprise web applications, but is simple enough that it can be implemented at much

smaller scales. Celery also uses popular message brokers like Redis and RabbitMQ to implement its task queues. Our implementation will use a single queue of tasks that can be processed independently of one another. However, Celery can use multiple queues with ranked priority and has built-in primitives such as task chains and groups that will make it possible to implement workflows requiring sequential processing steps. Celery also allows the user to define parameters like the number of times a task can be retried, the type of concurrency workers will use and can even be configured for an event-based scheduling system.

For our message broker we chose Redis [26] because it is simple to implement alongside our APIs and works out of the box with Celery. In addition, Redis works as a message broker to receive log messages from workers and publish them to our logging service (Elastic Stack). Redis also allows the creation of as many in-memory databases as memory allows so it can be extended beyond task queues and logging in future development. We chose Flower [27] and the Elastic Stack (formerly ELK) [28] for task monitoring and logging. Both of these solutions can be containerized out of the box with minimal configuration to work with our scheduler. Flower is a simple web-based UI that can be used to monitor the distribution of tasks and their status across Celery workers, and the Elastic Stack combines ElasticSearch, Kibana and Logstash to create a solution for aggregating logs, data and analytics accessible by a web-based UI.

We use Python's Docker SDK to launch the containers that execute the Burrows-Wheeler Aligner. We chose to continue to use containers for the execution environment because they provide a standardized and reproducible environment and because Biodepot maintains pre-built images on its DockerHub for its workflows which will make our scheduler adaptable to other bioinformatics analytical applications and scripts in the future.

Amazon Web Services (AWS) was chosen for our cloud platform because the Biodepot Workflow Builder (Bwb) already uses AWS in many of its applications, including a tool we helped develop that lets users estimate costs and launch instances with a simple web-based UI. Additionally, there is presently no cost associated with transferring data between S3 storage buckets and instances in the same region. Data transfer between instances and buckets in the same region quickly, and will allow us to utilize storage buckets as an ad-hoc shared filesystem.

One well known limitation of Python is that its Global Interpreter Lock [29] prevents true multithreading, which influences our choice of technologies used in the stack. By default Celery uses multiprocessing rather than multithreading for concurrency, with each Celery agent (which Celery documentation confusingly refers to as a "worker")spawning a sub-process (worker) up to its concurrency limit. Typically one agent is spawned per available instance to launch workers that run on that instance. Multiprocessing is almost always a better choice for computationally intensive tasks, while multithreading has the advantage as a lightweight solution when tasks are numerous, short and asynchronous or when tasks are I/O and network heavy. This makes Celery's default concurrency ideal for our scheduler as sequencing tasks are computationally intensive Celery does support using Python's "green threads" for concurrency as well, so the option is available if needed in the future.Additionally, Python's Docker SDK can launch containers using the host machine's Docker socket, which also fits the multiprocessing paradigm.

3.2 Scheduler Architecture

Our architecture assumes that the Biodepot workflow builder (Bwb) will manage workflow generation and abstraction into atomic tasks and the aggregation of results after jobs are completed. We utilize docker orchestration so that the components of our scheduler are modular and containerized. This structure allows the components to be hosted on any infrastructure capable of launching containers whether the infrastructure is entirely cloud based or a mixture of cloud and local resources. This architecture decouples the scheduler from the Bwb application and is ultimately platform agnostic with communication between components being accomplished with standard network protocols. The modular design also allows the scheduler to be portable and extensible.

Our modular architecture also will allow components to be updated as technologies and needs change. For example, the Elastic stack logging service could be swapped for a different logging solution. Flower could be replaced by a UI that better meets real-time task monitoring with minimal effort. Even the API could be replaced by a different framework as needed as beyond the container orchestration none of the components rely on the specific implementations of the others aside from the Celery framework. Even the

Celery framework could be replaced by another scheduling solution, but that would require more refactoring than replacing other components (see Figure 1).

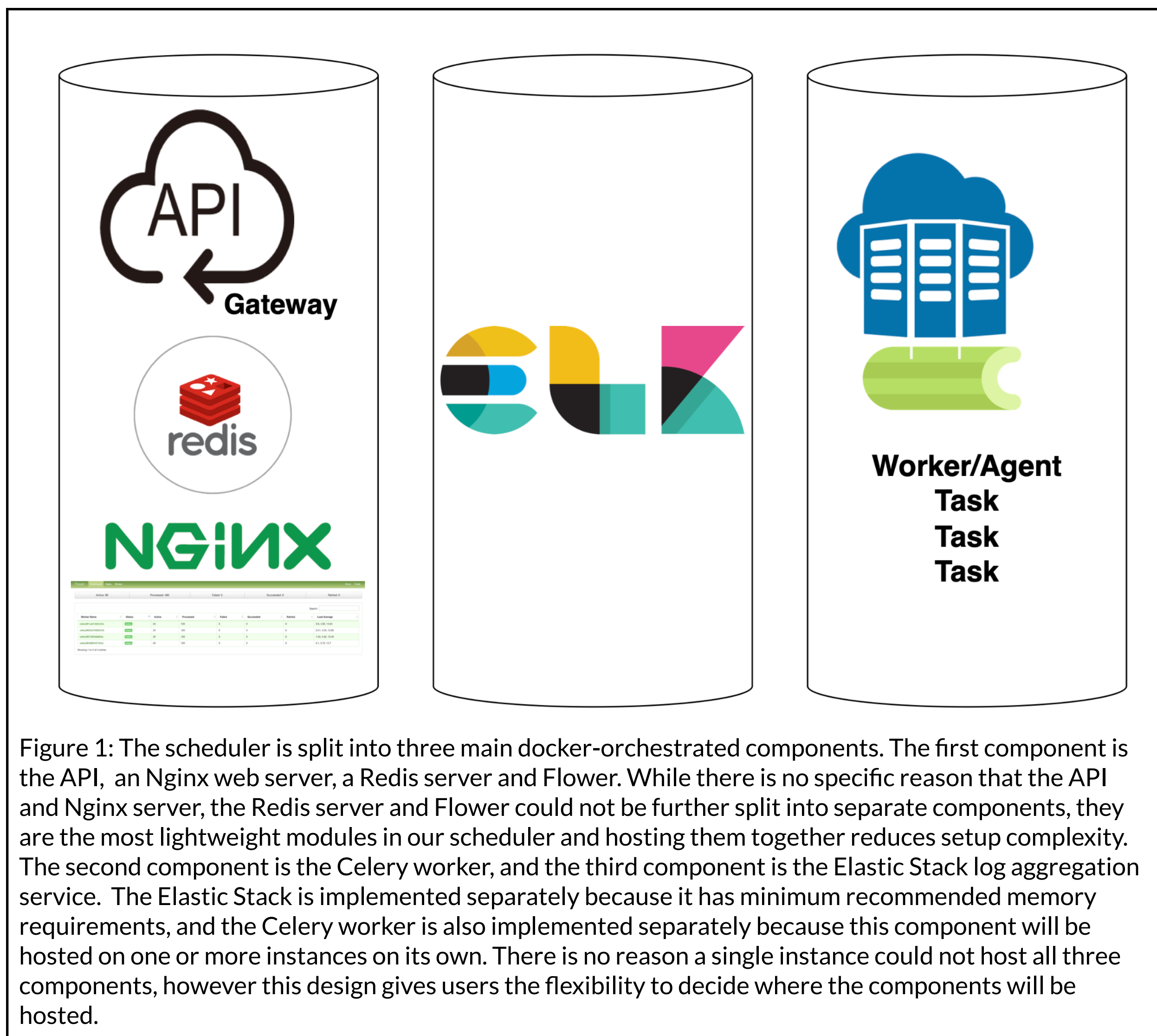

Figure 1: The scheduler is split into three main docker-orchestrated components. The first component is the API, an Nginx web server, a Redis server and Flower. While there is no specific reason that the API and Nginx server, the Redis server and Flower could not be further split into separate components, they are the most lightweight modules in our scheduler and hosting them together reduces setup complexity. The second component is the Celery worker, and the third component is the Elastic Stack log aggregation service. The Elastic Stack is implemented separately because it has minimum recommended memory requirements, and the Celery worker is also implemented separately because this component will be hosted on one or more instances on its own. There is no reason a single instance could not host all three components, however this design gives users the flexibility to decide where the components will be hosted.

3.3 Using the Scheduler

The API receives a JSON request from the caller (see Figure 2), which shows the overall architecture of our FastAPI-Celery Scheduler. In our case the caller is the Biodepot Workflow Builder (Bwb), which defines a list of tasks to process. A task is a set of key-value pairs, which include a unique ID which will allow the caller to poll for completion, an S3 key that locates the file to be processed, a list of S3 key locating necessary reference files for processing, a docker image that will be launched to execute the task, scripts or

commands that will be executed in the container, a local directory where S3 files will be stored and the where the resulting file(s) will be written to, and an S3 key that locates where the resulting file will be transferred on completion.

The API parses the request and publishes tasks to the Redis job queue. Celery workers retrieve tasks from the queue on a first in, first out (FIFO) basis and process as many tasks as their concurrency settings define. Celery workers first establish a connection to the Redis and logging server. The worker performs two types of operations. One operation is downloading the necessary support files which can be used by other workers on the same instance. The other operation is aligning the sequence in the fastq file to the reference. When a worker executes a task it first checks to see if all the support files are downloaded. If not, it will attempt to acquire a file lock on the Redis server and download the file. If all support files are present the worker will download the fastq file and run the sequence alignment as a single process in a Docker container. The Docker container runs independently of the Celery worker, but maintains a log stream (the container's stdout and stderr) to the Elastic stack, and polls the container's status. Once the container exists, the worker transfers resulting files to the S3 storage bucket and publishes a log indicating the task is finished.

Figure 2 diagrams the overall architecture of our FastAPI-Celery scheduler. While tasks are running, the Celery workers are publishing logs to the Elastic stack and reporting status back to the Redis server. Flower monitors the Redis server and provides the real-time status of workers and tasks through a web-based UI. If locally built Docker images are not being used, they can be retrieved from Biodepot's Dockerhub by the workers.

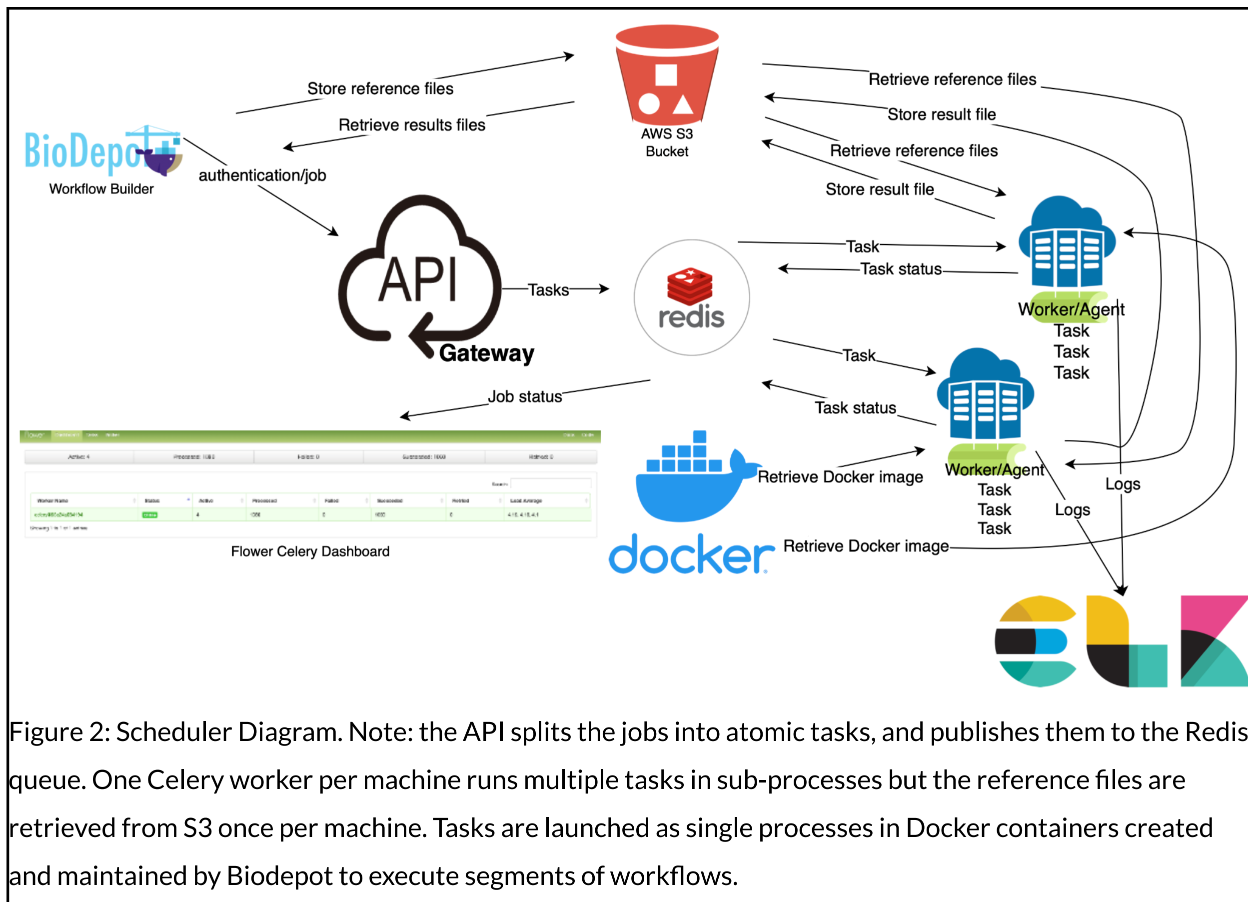

Figure 2: Scheduler Diagram. Note: the API splits the jobs into atomic tasks, and publishes them to the Redis queue. One Celery worker per machine runs multiple tasks in sub-processes but the reference files are retrieved from S3 once per machine. Tasks are launched as single processes in Docker containers created and maintained by Biodepot to execute segments of workflows.

Setting up the scheduler requires some configuration of instances because the AWS SDK, Boto3, can use the AWS credentials of the host machine as defined in docker-compose.yml files for authentication. AWS credentials can be defined programmatically, but we chose not to implement them programmatically because securely transferring credentials would require setting up SSL certificates, token based security measures, or additional security group configuration on AWS. Additionally, our implementation uses IP addresses and credentials stored in environmental variables for the Redis server and Elastic stack. With the present implementation the user only needs to install dependencies like the scheduler repository from GitHub, Docker/Docker-Compose and the AWS CLI, and configure three environmental files, and docker-compose files for each of the main components (API-Redis-Flower, Elastic Stack & Celery worker). The Celery worker only needs to be configured once and can be rapidly replicated using an instance image such as an AWS AMI. Once launched, the scheduler can process any number of discrete jobs concurrently and the caller can poll S3 locations or logs to monitor job progress and completion.

One reason we chose to store our cloud authentication and security credentials in configuration files is that implementing a robust security layer is beyond the scope of the project, but in future iterations configuration could be made easier for the user, and more flexible by implementing a SSO or token based authentication strategy. Our architecture should make integrating authentication or other services simple. In the case of adding an authentication layer, users could then pass credentials securely with requests to the API, enabling the ability to run multiple jobs accessing multiple different sets of cloud resources. This follows a contemporary web-services strategy where flexibility and interactivity are expected, which differs from batch scheduling which is primarily concerned with setting up jobs and then fairly executing them on static resources without user interaction.

4 Benchmarking

4.1 Dataset

Previous Bwb experiments have used a dataset generated by the LINCS Drug Toxicity Signature Generation Center, a NIH funded library that contains gene expression data from human heart cell lines under the influence of different drugs [30], [31]. Those experiments processed the data in a 3-step UMI RNA-seq workflow that first splits genomic (sequence) data into individual files, then aligns the sequences to a reference sequence and lastly merges the results. As the alignment step is the most computationally intense portion of the workflow and is parallelizable, our tests will use the LINCS dataset pre-split into 1,752 fastq files of sequences and align them to the same human reference used in the Bwb experiments. The results of these previous experiments served as a ground-truth for our alignments, and by using this dataset throughout our design process and initial testing we were able to verify the correctness of our implementation and the consistency of our results.

4.2 Set up

We used an AWS t2.micro [32] to run the API task scheduler, Redis server and Flower dashboard. The t2.micro instances are provisioned with a single vCPU and a gigabyte of ram. We chose a small instance to showcase the cost effectiveness of the lightweight API.. The t2.micro instances cost just $0.0116 per hour, making them very cheap to run alongside the larger instances we used for the workers. For the Elastic stack logging service, we used an AWS t2.medium instance. The t2.medium instances are provisioned with 2 vCPUs and 4gb of RAM. We chose this instance for the logging service because 4gb of RAM is a suggested minimum requirement for running the Elastic stack and the t2.medium instance was the cheapest instance available with 4GB of ram at a price of $0.0464 per hour. Both the API stack and the Elastic stack can be run on the same machine that hosts workers but for our experiments we wanted to evaluate the performance of the workers without additional overhead.

The machines used to run the workers in the experiments were AWS c5.9xlarge EC2 instances. AWS's c5-series instances are compute optimized and provision either 2nd generation Xeon Scalable Processors or 1st generation Intel Xeon Platinum 8000 Skylake-SP processors [33]. In our initial pre-testing, we had instances provisioned with different 8000 series processors, and though the difference in execution times were negligible, we decided to run the tests on Xeon Platinum 8142 3.00GHz processors because they seemed to be the most common processors provisioned on launching an instance. AWS chooses which processors to provision when an instance is created, so we relaunched instances until we had instances provisioned with the Xeon 8142 processors. The c5.9xlarge instances are provisioned with 36 vCPUs and cost $1.53 per hour to run. The c5-series instances were chosen because alignment algorithms are typically computationally intense processes, and compute optimized instances should be an appropriate choice. We also ran all experiments on 36 vCPU instances, even when running tests with a single process to reduce potential sources of variability. Potential sources of variability include AWS thread management on shared servers. AWS uses hyperthreading to manage process loads on its Intel instances, threads are represented by instances as vCPUS but are actually threads that will be executed in physical CPUs. For our c5-class instances, each physical CPU processes up to two threads represented as vCPUs. It is much more likely that

the allocation of 36 vCPUs will come from a single server (which will have 18 physical CPUs), without shared jobs, than it is for smaller vCPU counts to come from a single server. However, this manner of provisioning compute resources means that our instances can have more workers than physical cores, which can result in contention for CPU resources at high vCPU utilization. We will refer to this contention as "thread collision."

4.3 Scheduler Performance

To benchmark our scheduler, we ran tests using the Burrows-Wheeler Aligner (BWA). BWA is a software tool that is used to map short sequences against a large reference genome, such as the human genome [34]." As our research is focused on task scheduling, the specific use cases and details for this kind of genomic analysis are not discussed. BWA is a very popular alignment module that is used in a large number of bioinformatics analyses. The mapping of short sequences performed by BWA can be executed as parallel jobs as the mapping of one sequence does not affect the mapping of other sequences. Because we are testing a scheduler designed first to handle parallel tasks, BWA is a good choice for our benchmarking.

4.3.1 Comparing the scheduler to shell scripts on a single instance

We executed BWAwith our FastAPI-Celery Scheduler running with a single worker constrained to a single process on a single instance and compared it to a script that executed all of the same Docker commands sequentially, and then timed the transfer of reference and fastq files from S3 in order to evaluate the overhead of our scheduler (see Figure 3). The key difference between the scheduler and the script is that the script requires the fastq files to already be present on the instance executing the alignment, whereas our scheduler downloads the files from S3 as needed by each task, and transfers results files back to S3. We ran each test

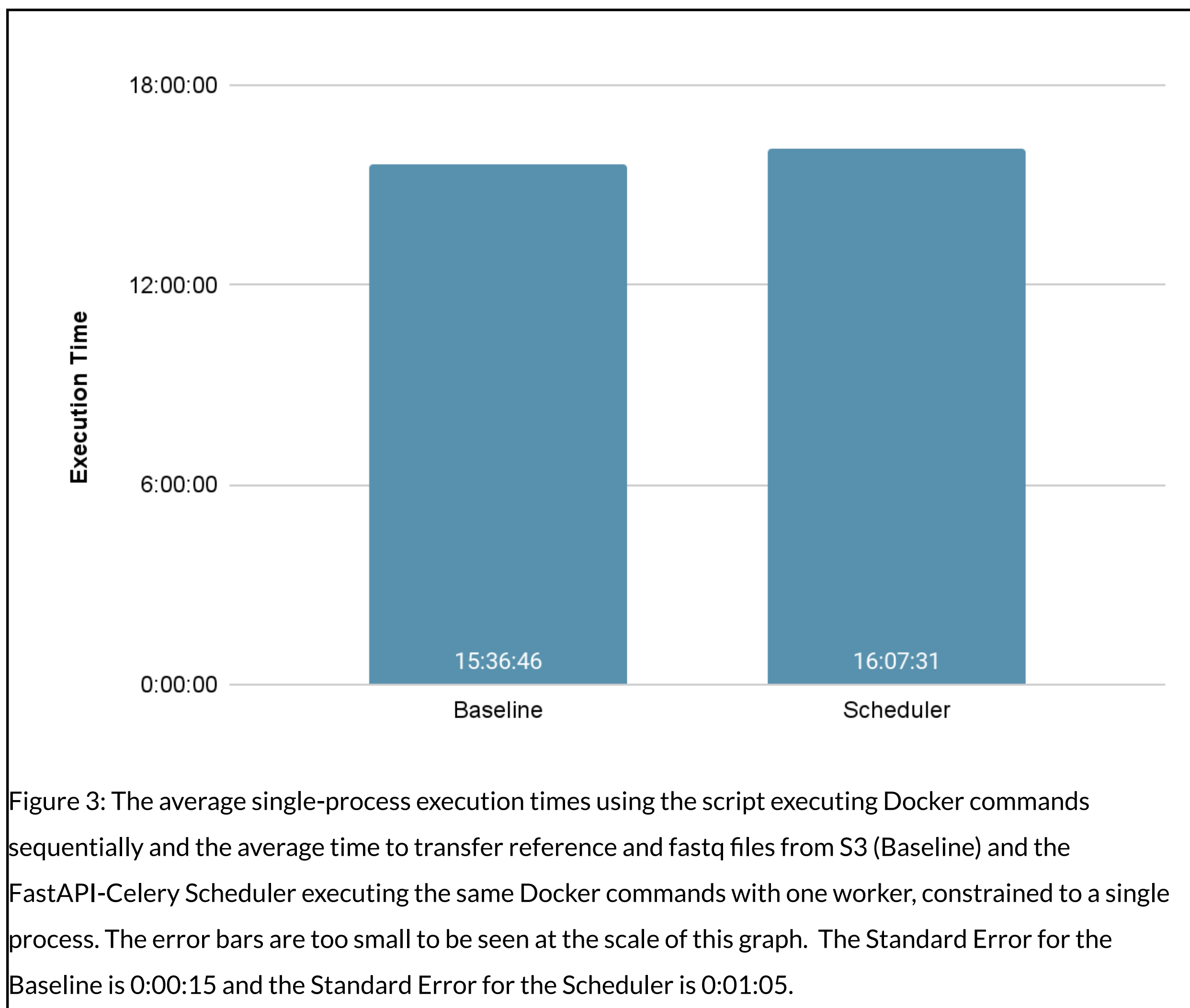

Figure 3: The average single-process execution times using the script executing Docker commands sequentially and the average time to transfer reference and fastq files from S3 (Baseline) and the FastAPI-Celery Scheduler executing the same Docker commands with one worker, constrained to a single process. The error bars are too small to be seen at the scale of this graph. The Standard Error for the Baseline is 0:00:15 and the Standard Error for the Scheduler is 0:01:05.

three times to establish the consistency of execution times, and the Standard Error is small enough that error bars are not visible on the graph. The average execution times for our FastAPI-Celery scheduler were slower than the script and file transfer by 3.18%, which suggests that the overhead of scheduling tasks is quite low. In terms of actual execution time, that 3.18% is about 30 minutes on jobs lasting over 15 hours.

For the rest of our tests we compare our FastAPI-Scheduler to an existing (Bwb Multiprocess Script) script that executes the BWA with a parallelization strategy, and we will use it to evaluate the overall performance of our scheduler at scale. The Bwb Multiprocess Script executes the parallel processing of genomic data on a folder by folder basis by concatenating individual fastq files rather than processing each file individually in order to reduce the total overhead of launching binary processes (there are 96 folders, as opposed to 1,752 individual files in our test data).

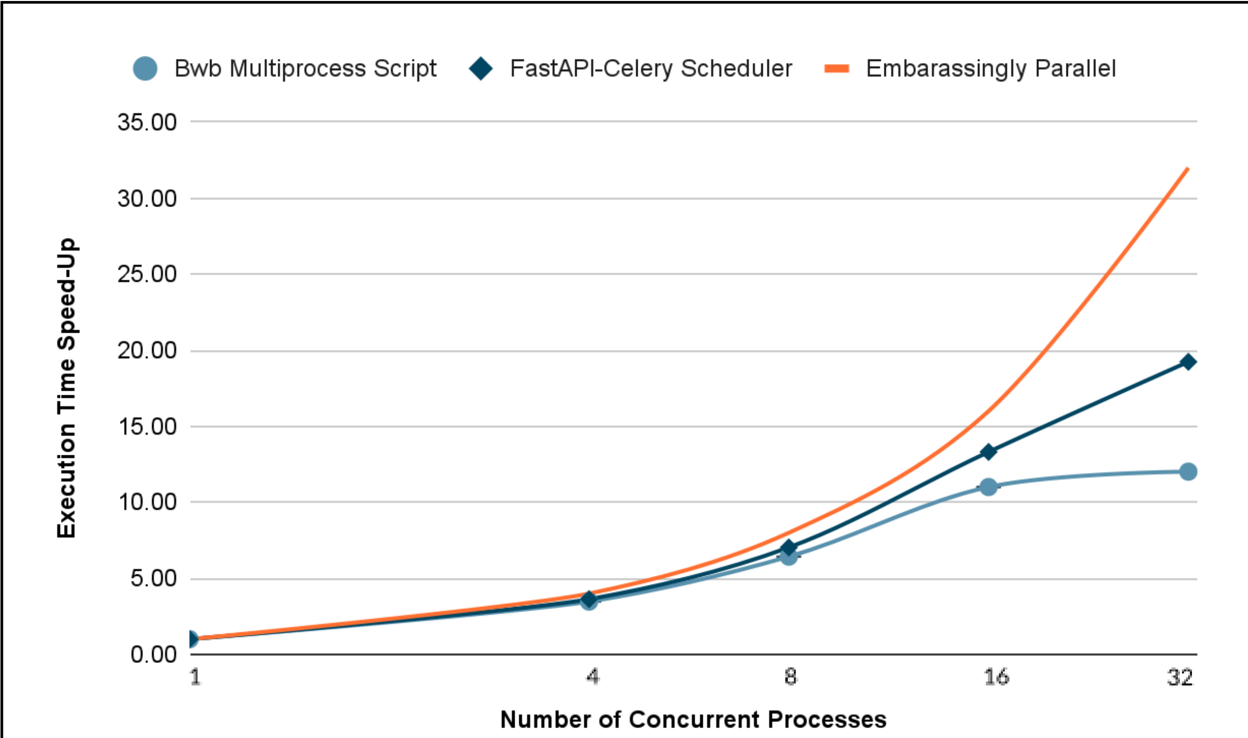

Figure 4: Comparative speed-up in execution times compared to embarrassingly parallel execution. This was calculated by dividing the single process execution time by the multiprocess execution times. As before, the Standard Error in execution times is small enough (measured in seconds and milliseconds) that the error bars are not visible.

First, we ran the Bwb Multiprocess Script, limiting it to a single process, and compared the execution time to the single process execution time for our FastAPI-Celery Scheduler. This is shown in Figure 4 We ran the test three times to evaluate potential variability in performance with the variation being too small to show in Figure 4. Next we ran the Bwb Multiprocess Script and our FastAPI-Celery Scheduler on one 36 vCPU machine with increasing concurrency: 4, 8, 16, and 32.The line plot shows the curves of both the Bwb Multiprocess Script and our FastAPI-Celery Scheduler measured against an embarrassingly parallel execution. While our scheduler performs slower than the script at lower concurrent processes, the curves show that the scheduler begins outperforming the script at about 16 concurrent processes and both begin lagging increasingly further behind embarrassingly parallel execution as the number of processes increase.

The slow down in execution times as processes increase starts becoming most pronounced at around 16 concurrent processes, suggesting that thread collision prevents the full utilization of vCPUs.

Initially, we did not expect our FastAPI-Scheduler to outperform the Bwb Multiprocess Script, however the reason is likely due to the difference in the availability of tasks. There are 96 folders in our test data, which means that the script will have 96 available concatenated fastq files to process as tasks as compared to the 1,752 individual files available as tasks to the scheduler. Each of the 96 concatenated files would have significantly longer processing times, so at the end of the job there were likely many idle processes in the script attempting and failing to acquire file locks.

We ran more tests utilizing 18, 20, 22, 24, 26, 28, and 30 concurrent processes to understand the relationship between and execution times with higher numbers of workers. In Figure 5A, we see that for both the Bwb Multiprocessing Script script and our FastAPI-Celery Scheduler the reduction in execution time flattens out starting at 16 concurrent BWA alignments, when we may start experiencing more contention for available CPU cores.

Next we graphed the reduction in execution times in our tests from 18 to 32 concurrent BWA processes to better understand the relationship between execution times and concurrent processes when thread collision is occurring. Figure 5B shows a consistent increase for the FastAPI-Celery Scheduler, but for the Bwb Multiprocessing Script the results level off and begin to decrease at higher process counts . This suggests that the scheduler will continue to scale with high numbers of processes. The overhead for the lock acquisition

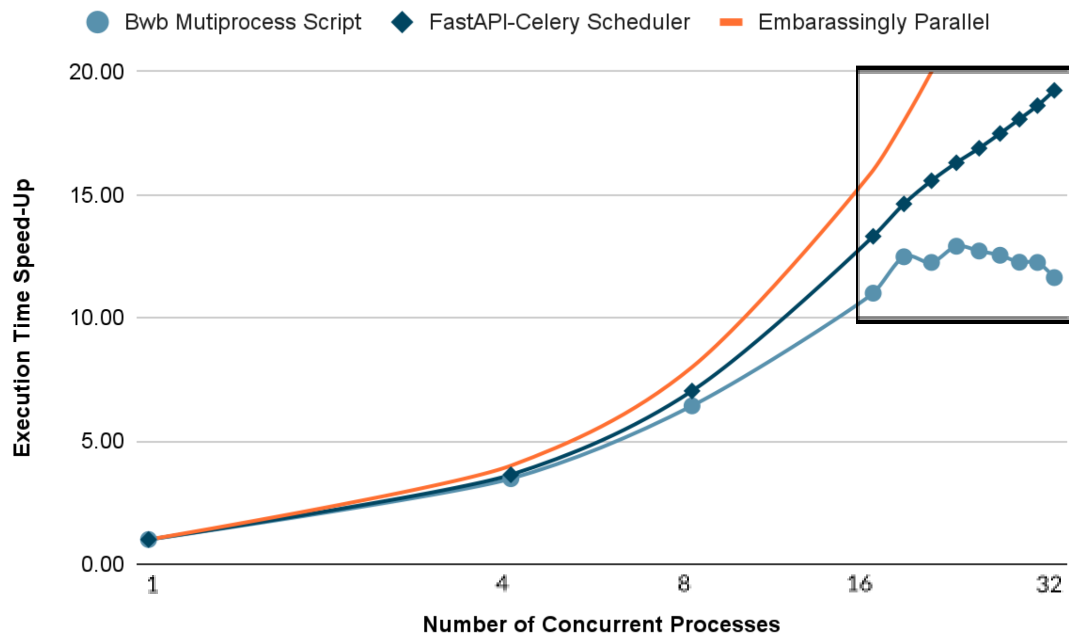

Figure5 A: Differences in execution time speed-up between 16 and 32 processes suggest that our FastAPI-Scheduler will scale much better than the Bwb Multiprocess Script.

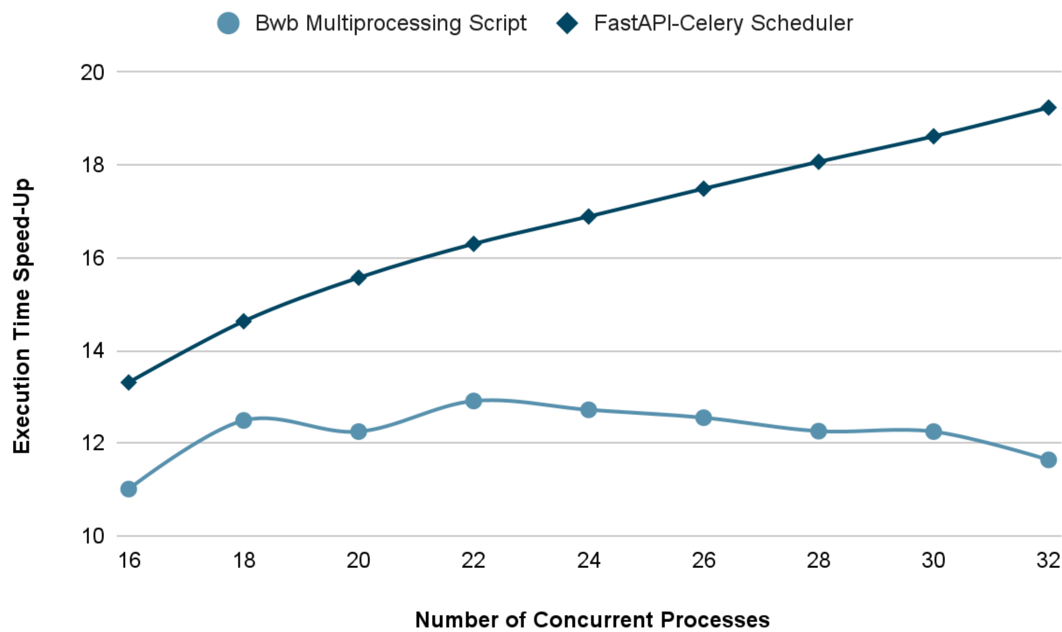

Figure 5B: Zoomed comparative execution time speed-up on 16 to 32 concurrent processes showing an ever increasing speed-up for the Celery Scheduler. In contrast the script plateaus and then shows a decrease in performance due to extra processes that do not have enough work to do but still consume resources.

Figure 5: Scaling at higher process counts

strategy for the script, however, has an increasing linear cost as the number of processes increases. When processing a smaller pool of tasks (96 vs. 1,752) of longer tasks it is likely that when the few last long-running tasks are being processed, many available processes are un-utilized. These extra processes contribute less than their share in processing jobs but add the same overhead by attempting to acquire file locks until the list of available jobs is exhausted.

4.3.2 Scheduler performance on multiple instances

One of our primary motivations in implementing our FastAPI-Celery Scheduler was to enable Bwb jobs to be executed across multiple instances in parallel, so our next step was to evaluate its performance in this context of multiple agents spawning workers deployed on multiple instances. For these tests we continued to use the same API instance and Elastic stack instances and used an AWS instance image (AMI) to replicate our Celery worker instance. On launching the replicated instances we verified that each was launched with Xeon Platinum 8142 3.00GHz processors to eliminate any noise from variation in processor speeds. We used four instances, with that host the same number of workers on each for each test. This will allow us to test the effect of increasing workers on multiple instances. Our assumption was that we would see similar reductions in execution time as process count increases, until we started encountering thread collision on the four instances.

In our experiments utilizing a single instance we saw obvious diminishing returns in speed up when our instance executed 16 concurrent Burrows-Wheeler Alignment (BWA) processes, which established 16 concurrent processes as a good starting baseline for comparison with running processes across multiple instances. Our tests with four instances start at four processes on each for a total of 16 concurrent BWA processes. We increased processes to 8 on each instance, for 32 concurrent processes, then to 16 on each instance, for 64 processes followed by 24 and 32 processes on each machine, for 96 and 128 total concurrent BWA processes.

In Figure 7, we look at the total number of processes used and compare the average speed-up in execution times of our FastAPI-Celery Scheduler running on a single instance to our scheduler running on four instances and plot them against a theoretically embarrassingly parallel execution of the job. The 4 instance curve is essentially a repeat of the 1 instance curve shifted up 4 fold and shifted to the right.

In Figure 8 we can see that there is overhead to executing tasks on multiple instances as compared to a single instance. We see that the speedup is very close to 4-fold when compared to the single instance case. The difference is 0.5% to 5% with relative performance (efficiency) becoming worse as there are more processes per instance. This could be due to per worker, scheduler overhead. However, the scheduler appears to be very efficient. The performance relative to embarrassingly parallel for 4 instances is similar to the corresponding value for 1 instance with the same number of workers per instance. The slightly worse performance is probably due to scheduler overhead.

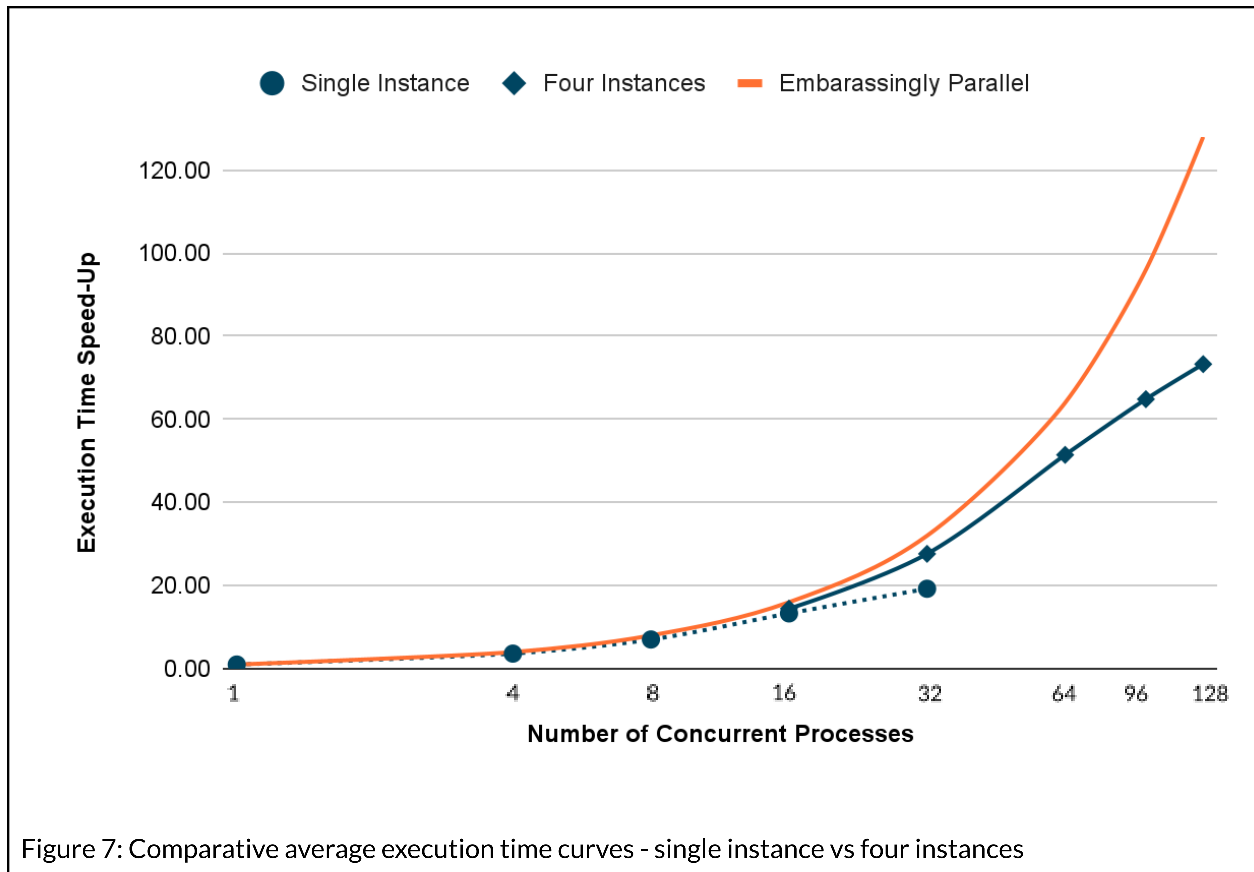

Figure 7: Comparative average execution time curves - single instance vs four instances

Number of Processes per Instance	Single Instance Speed-up	% of Embarrassingly Parallel	4 Instance Speed-up (efficiency)	% of Embarrassingly Parallel
1	1	100%	-	-
4	3.63	90.75%	14.45 (0.995)	90.31%
8	7.03	87.87%	27.64 (0.983)	86.37%
16	13.31	83.18%	51.46 (0.967)	80.41%
24	16.89	70.38%	64.86 (0.960)	67.56%
32	19.24	60.13%	73.3 (0.952)	57.27%

Figure 8: Comparing execution times to embarrassingly parallel execution on single Instance and multiple (4) instances

4.4 Cost Estimates

In this section we will discuss the costs of running the tests of our FastAPI-Celery Scheduler. We will ignore costs that are not dependent on the instance type such as storage costs for data and AMI images. These costs are insignificant over the time scale for our execution times. We also will ignore the costs for the small t-class instances needed to run the scheduler itself and the logging service. Our discussion of actual costs will be limited to c5-class instances because we executed our tests on c5.9xlarge instances.

AWS has a fixed price per vCPU hour ($0.0425) for its c5-class vCPU instances. For example, running 128 c5.large instances that have 2 vCPUs each for one hour at an hourly cost of $0.085 per instance would cost $10.88 and running 64 c5.xlarge instances that have 4 vCPUS each for one hour at an hourly cost of $0.17 per instance would also cost $10.88; increasing the size to c5.2xlarge instances that have 8 vCPUs each would also cost $10.88 to run 32 instances at a cost of $0.34/hour per instance. This pricing structure makes sense because instances are logical machines where vCPUs can be allocated to physical CPUs that exist on one or more servers and Amazon likely has fixed costs for provisioning CPU resources. If smaller instances were cheaper than larger instances, users might try to game the system by using many smaller

instances instead of a single larger instance, which would be a strategy we might try to use with our scheduler.

In a hypothetical embarrassingly parallel execution, speedup would scale perfectly with vCPUs regardless of how many vCPUs are used per instance. If the work was distributed using an ideal scheduler that has no overhead, fixed cost per vCPU hour would mean that the performance would only depend on the total number of vCPUs used and not on how the workers are distributed. In the non-ideal case we can still make some recommendations. It will be somewhat cheaper to use fewer instances at maximum vCPU utilization than it will be to use more instances at maximum vCPU utilization due to scheduler overhead. Figure 9 shows this clearly, where the total cost to run the job is slightly higher when using the same number of processes per instance on four instances as using only a single instance. What is significantly different in this data, however, is the execution time. Running a job with 32 processes on a single instance costs $1.28, whereas running a job on four instances with 32 processes each costs $1.35 (a negligible difference at the scale of our tests) but running the job with four instances takes just over 13 minutes compared to just under an hour when running the job with a single instance – the speed-up from the extra vCPUs makes the four instance execution time 3.8 times faster than the single instance execution time.

The data indicates that, in general, the best strategy for cheaper execution costs with c5-class instances will be to use fewer instances with the maximum number vCPUs available and to max out the utilization of those vCPUs. The optimal strategy for maximizing speed, based on our data, is less clear than the strategy for reducing costs; there will be a point where the speed gained from reducing thread collisions by using more workers with low vCPU utilization will be greater than the cost of scheduling overhead. Knowing where this point is on our c5-class instances would require more testing, but we suspect that reducing vCPU utilization further below 50% is unlikely to produce enough speed-up to be worth the cost of more vCPUs per worker. This can be seen in our data where we compare the execution times using 8 and 16 processes per instance. The cost is nearly the same whether you use one instance with 8 or 16 workers, or four instances with 8 or 16 workers each, but the execution time is much faster using more workers on more instances. We can also see that running the job on a single machine with 32 workers costs $1.28 but running

32 workers on four machines with low vCPU utilization costs $3.57, which means that being willing to pay 2.79 timed more to run the job returns a 3.8 times speed up in execution time.

Processes per Instance	Single Instance Execution Time	Cost of Single Instance	4 Instance Execution Time	Cost of 4 Instances
4	4:26:25	$6.63	1:06:57	$6.83
8	2:17:32	$4.33	0:35:00	$3.57
16	1:12:40	$1.86	0:18:48	$1.92
24	0:57:17	$1.46	0:14:55	$1.52
32	0:50:17	$1.28	0:13:12	$1.35

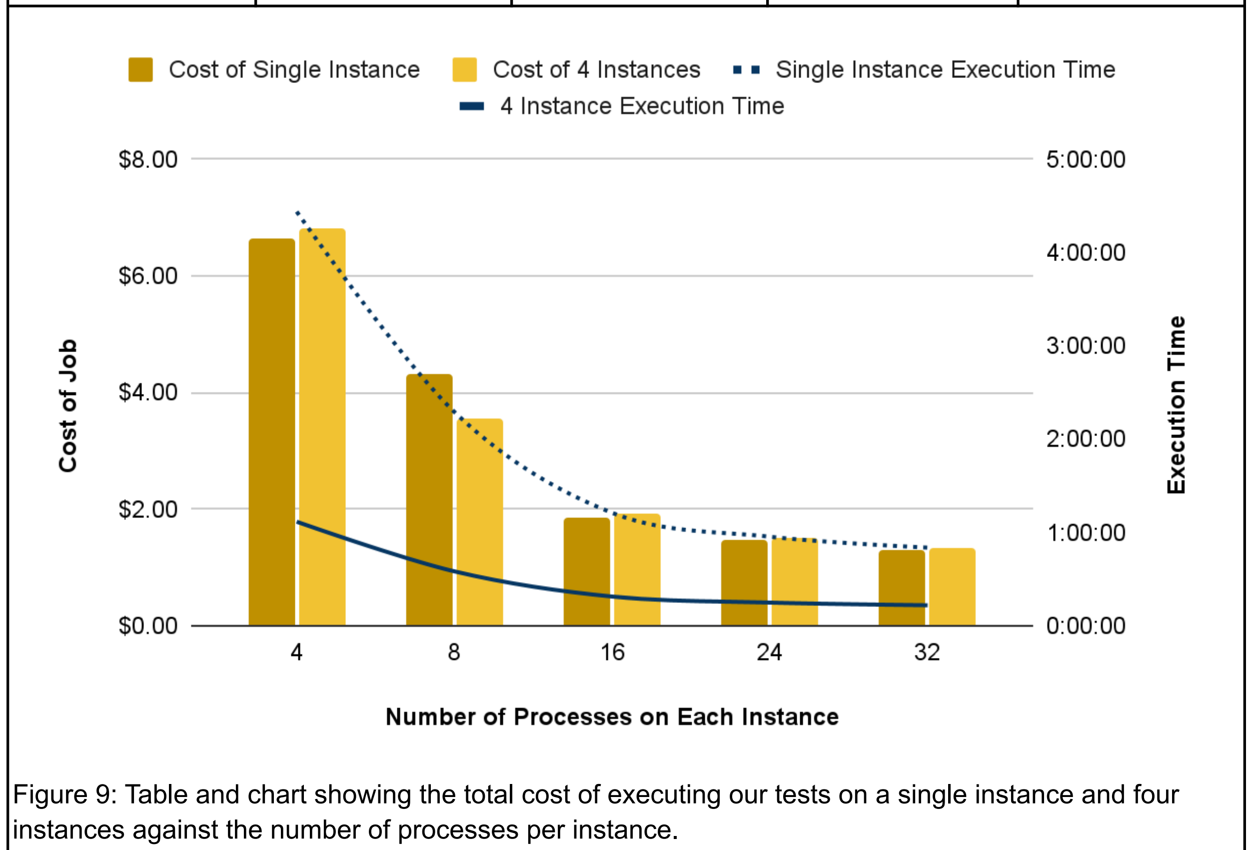

Figure 9: Table and chart showing the total cost of executing our tests on a single instance and four instances against the number of processes per instance.

We could test for trends by running more experiments using smaller 8 vCPU, 4 vCPU, and 2 vCPU instances with enough total vCPUs to match the scales of our original tests (from 16 to 128 distributed workers). If we wanted to generate data beyond the scale of original experiments we could test 1,752 processes distributed over 32 c5.9xlarge (36 vCPUs) instances, 24 c5.12xlarge (48 vCPUs) instances, 16 c5.18xlarge (72 vCPUs) instances, and 12 c5.24xlarge (96 vCPU) instances to determine whether there is any significant performance costs to maxing out vCPU usage on larger machines. We could also run experiments starting with a small number of c5.xlarge (4 vCPU) machines running one process each and successively increasing the number of instances until we begin to see diminishing returns on execution speed-up due to the scheduling overhead similarly to how saw diminishing returns due to thread collision in our experiments.

Our experiments were executed using a compute-bound workload, so our performance and costs may only be useful for other compute bound tasks. To determine other optimization strategies we would also have to run similar experiments using other instance types with both our compute-bound workload and other workloads that are memory-bound, I/O-bound, or network-bound. We may uncover 'sweet spots' where certain instance types and distribution strategies have optimal cost vs execution time points. Additionally, we would want to experiment with different Celery configurations to uncover parameters that might optimize performance further.

So far we have performed our analysis assuming that we are using on-demand instances. However, Amazon offers two kinds of EC2 instances with its AWS service: on-demand instances [33] and spot instances [34]. On-demand instances guarantee usage of the requested number of vCPUs while spot instances utilize vCPUs that are not already being utilized by on-demand instances. This means that spot instances are not always available, and can be terminated when on-demand instances need vCPU resources, which is why we ran all of our tests utilizing on-demand instances to avoid interruption of our experiments. Spot instances, however, can be requested for fractions of the cost of on-demand instances, up to a 90% discount from on-demand prices, with the user being able to set the maximum hourly price they are willing to pay to access vCPU resources. There's some element of competition here, because spot instances can be

terminated when compute resources are requested by other users at a higher price points, or other on-demand instances require those resources.

The price difference between on-demand and spot instances might suggest that the strategy of reducing costs by using large instances with maximized vCPU utilization is a good choice when using on-demand instances that have fixed pricing per vCPU hour as this where high execution costs can be incurred by underutilizing vCPU resources. Conversely, the strategy of distributing workloads over many instances with less than 50% vCPU utilization might be a good choice when using spot instances. Because spot instance are so much cheaper than on-demand instances, and it's often easier to secure smaller spot instances than it is large ones, savvy researchers might discover AWS regions and instance types that are less popular (easier to secure) and run jobs using as many instances as they can secure at a fraction the cost of using the same strategy with on-demand instances. If the tasks in a given workload are atomic, then a researcher wouldn't necessarily need to worry about instances being terminated either. Because the scheduler retrieves needed files for each task and stores the results in cloud storage, failed or missed tasks could be re-run on remaining instances or new instances simply with another API call.

An additional use case for the scheduler is when a researcher has one or more instances they are already running, where resources are not always fully utilized. Our FastAPI-Celery scheduler would enable the researcher to use the extra capacity of these instances to execute tasks (maximizing the value of existing sunk costs) by installing Celery workers with no concurrency settings,as by default a Celery worker will spawn as many sub-processes (workers) as there are CPU or vCPU resources available. This same strategy could be employed by multiple researchers who all have running instances. Our scheduler can queue multiple jobs (presently on a FIFO basis), and as long as some method of passing cloud credentials to workers was developed the scheduler would be able to retrieve files and transfer results to different cloud storage buckets. Pooling resources in this way could create a network of workers available to execute jobs on-demand.

5 Future Work

Some areas to explore in future iterations of our FastAPI-Celery scheduler might include testing rudimentary task optimization. In our tests, we processed fastq files in the order they appeared in file systems and a simple optimization strategy might be to process fastq files in order of descending file size. The BWA tends to, but not always, take longer to align larger files than it does smaller files so while processing files in descending order according to size wouldn't be a perfect optimization, it might produce some efficiency gains. We could also experiment with splitting or combining fastq files into smaller or larger individual files to discover whether there is a close-to-uniform file size where execution time is optimized with respect to the overhead of launching containers..

Celery's documentation states there may be some cases where running multiple agents (Celery workers) on the same instance, with each agent spawning multiple workers, and splitting tasks between them might be more efficient than running a single agent with the same number of workers. We did not discover any case studies or guidance regarding when to employ this strategy, it could be tested by running tests with different numbers of agents and concurrency settings. Additionally, we could experiment with different AWS instance types to discover if there are any optimal configurations of types and number of vCPUs. We used compute optimized instances in our tests because the BWA gene sequencing is compute intensive, but AWS offers many optimizations including memory, network, I/O, dedicated servers and other instance configurations. Notably, the t2 and t3 instance types feature 'burstable' performance. They guarantee a baseline performance but can boost processing speeds when needed based on a credit system where an instance earns credits when vCPUs are underutilized and spends credits when vCPUs are boosted above baseline performance [35]. As t2 and t3 instance types are typically cheaper than other instance types and boosted performance could potentially be more cost effective. AWS also now offers cheaper instances with ARM processors, so the BWA could be recompiled to run on ARM architectures which could also be more cost effective. One more variation of instance types that could be tested are P class GPU instances featuring NVIDIA and ARM-based Graviton GPUs. Running these tests might reveal configurations that optimize costs and execution times ("sweet spots"). These further experiments could also generate data that

would allow us to build a matrix of instance deployments showing worker-concurrency strategies with given cost and time constraints. Such a matrix could inform deployment choices and customizations we could make to Celery to maximize those strategies.

Another possibility for future development would be to deploy our Celery workers in cloud functions. Our containerized architecture strategy should enable this with some modifications. In this deployment scenario we could test different configurations of cloud function resources and optimize them specifically for our jobs; we could even test utilizing a mixture of instances and cloud functions assuming that some researchers might have instances available but might also want to take advantage of the inherent scalability available with cloud functions. Celery is not designed to work as a serverless application but there are a few examples of Celery workers being deployed in Lambda functions that could serve as initial guides for us to build a proof of concept.

Lastly, our implementation executes the BWA in pre-configured Docker images so immediate next steps would be extending the scheduler to work with images that execute other bioinformatics algorithms so that entire workflows could be executed, potentially including image processing with machine learning algorithms. There are also examples of FastAPI and Celery being used to train machine learning models that could serve as a basis for extending our scheduler. As our implementation was based on microservices architecture it has very few dependencies: Docker, the security layer (which is currently managed by environmental files), and HTTP for requests. It was designed with the single-responsibility principle in mind, and is front-end agnostic, which means it would be very simple to create an independent front end in a popular framework like React using SSO and tokens for security with Bwb or some other application.

6 Conclusions

We proposed to build a modular, portable and extensible task scheduler for bioinformatics workflows through the Biodepot Workflow Builder (Bwb) and test it against an existing multiprocessing strategy that is limited to a single machine or cloud instance. We designed our scheduler using Python, a language that many data scientists and researchers will be familiar with, and using established, well-maintained open-source

libraries and implemented our components with a modular, containerized strategy. Our design was intended to make it easy to update or replace components as technology and needs change because development and maintenance work on the Bwb software suite is often done by students and interns. Additionally, we implemented our scheduler behind an API that accepts standardized JSON requests from callers, making it ultimately agnostic to the front end. It can be called by Bwb, another application, scripts, manual cURL requests.

All the scheduler needs is a properly formed request, and for the files and docker images included in the request to be accessible. This is made possible by utilizing cloud storage containers (S3) as a quasi-distributed file system because data transfers within AWS regions are currently free and reliably fast. While our implementation works specifically with the Burrows-Wheeler Aligner (BWA) by launching a pre-configured Docker image for each task, our API's request structure will allow other container images to be launched with very minimal refactoring.

Our tests provide evidence that our FastAPI-Celery Scheduler runs efficiently when compared to embarrassingly parallel execution, with a limitation that high vCPU utilization on instances produces diminishing returns in execution time because tasks contend for compute resources (thread collision) . In our baseline tests we compared the performance of our FastAPI-Celery Scheduler executing a single process to a script that processes fastq files by launching Docker containers sequentially and showed that our scheduler ran only 3.18% slower than the script, which suggests that the overhead of the scheduler is acceptably low. We also tested our scheduler against an existing single-parallelization strategy (Bwb Multiprocessing Script) and found that the script outperforms our scheduler when the number of workers (processes) is less than 50% of the number of available vCPUs, but our scheduler begins outperforming the script when the number of workers is 50% or more than the number of available vCPUs. We measured diminishing returns to increasing the number of workers when the number of workers reaches approximately 44% of the available vCPUs on an instance, as the speed-up in execution times curve begins flattening with respect to the number of workers.

While we expected that there would be some cost advantages to executing our FasAPI-Celery scheduler across multiple instances, we discovered that Amazon uses a fixed cost per vCPU hour on the c5-class of

instances we used for our experiments meaning there is no cost advantage to choosing instances with fewer vCPUs. And at the scale of our experiments (maxing out at 128 workers) we saw that there is very little difference in cost (measured in pennies) between executing a job over a longer period of time with fewer workers and executing a job over a shorter period of time with more workers. Executing jobs with more workers across multiple instances is slightly more expensive at our scale, and suggests that researchers will choose to execute jobs with more workers as there is minimal cost to achieving many-fold faster execution times. This observation makes sense for our tests because our workload is compute bound and we executed jobs with nearly 100% vCPU utilization when our number of workers were maximized. But this also suggests the need for future experiments using different instance times with workloads that are memory-bound, I/O-bound, network-bound, or other scenarios where raw processing power is not the main constraining factor.

We demonstrated that our scheduler is efficient and scalable making it a good choice whether using a single instance or multiple instances, and our use of cloud storage buckets as quasi-distributed file-systems combined with the ability to rapidly deploy Celery workers from AMI images will give researchers fine-grained control over their infrastructure and its costs, which could include bidding on and securing spot instances for a fraction (up to 90% less) of the cost of on-demand instances. Researchers could also maximize the value of already sunk costs by deploying our scheduler on already running instances to take advantage of any underutilized resources. This strategy could be deployed individually or by multiple researchers in order to pool resources and create a network of available workers.

Our implementation is also relatively simple to configure with basic knowledge of container orchestration and Python, which many researchers will have. This is an advantage over more complicated solutions such as deploying workers in Kubernetes clusters. Our Celery workers can also be deployed on any resource capable of executing Docker images, making it a highly flexible solution for researchers because they can execute bioinformatics jobs on the resources they have available to them. While some minor work would still need to be done for the FastAPI-Celery scheduler to process more kinds of analysis, overall, the implementation is a success and will provide another viable option for researchers who wish to parallelize bioinformatics workflows.

Notably, Celery is designed for managing asynchronous task management in Enterprise distributed web applications and not for scientific job scheduling. But we have shown that it can work quite well for our BWA workflow with minimal overhead, and effective scaling, and has a robust set of features, which could be applied to different kinds of scientific analysis and workflows. Every framework and library we chose for our implementation is well-known, and widely used in consumer and commercial web applications, making our implementation accessible and easy to integrate with other popular frameworks and open-source tools (including GUIs). Ultimately this should make our FastAPI-Scheduler much more approachable for non-experts to work with, and by virtue of being in the open-source space, suited for contributions from scientists, open-source developers and other who are interested in bioinformatics and scientific computing than typical solutions developed specifically for scientific computing or HPC architecture.

References

[1] L.-H. Hung *et al.*, “Building Containerized Workflows Using the BioDepot-Workflow-Builder,” *Cell Syst.*, vol. 9, no. 5, pp. 508-514.e3, Nov. 2019, doi: 10.1016/j.cels.2019.08.007.

[2] “We’re not prepared for the end of Moore’s Law,” *MIT Technology Review*. https://www.technologyreview.com/2020/02/24/905789/were-not-prepared-for-the-end-of-moores-law/ (accessed Mar. 03, 2023).

[3] R. C. Martin and R. C. Martin, *Clean architecture: a craftsman’s guide to software structure and design*. London, England: Prentice Hall, 2018.

[4] B. Burns, *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services*, 1st edition. Sebastopol, CA: O’Reilly Media, 2018.

[5] “Sun Grid Engine (SGE) QuickStart — StarCluster 0.93.3 documentation.” http://star.mit.edu/cluster/docs/0.93.3/guides/sge.html (accessed Dec. 04, 2022).

[6] A. Boccia, G. Busiello, L. Milanesi, and G. Paolella, “A Fast Job Scheduling System for a Wide Range of Bioinformatic Applications,” *IEEE Trans. Nanobioscience*, vol. 6, no. 2, pp. 149–154, Jun. 2007, doi: 10.1109/TNB.2007.897474.

[7] “Slurm Workload Manager - Overview.” https://slurm.schedmd.com/overview.html (accessed Dec. 04, 2022).

[8] “Efficient Batch Computing – AWS Batch – Amazon Web Services,” *Amazon Web Services, Inc.* https://aws.amazon.com/batch/ (accessed Dec. 04, 2022).

[9] “Apache Hadoop.” https://hadoop.apache.org/ (accessed Mar. 03, 2023).

[10] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008, doi: 10.1145/1327452.1327492.

[11] “Apache Spark™ - Unified Engine for large-scale data analytics.” https://spark.apache.org/ (accessed Dec. 04, 2022).

[12] “Overview,” *Kubernetes*. https://kubernetes.io/docs/concepts/overview/ (accessed Dec. 04, 2022).

[13] “OpenWDL.” https://openwdl.org/#one (accessed Dec. 04, 2022).

[14] “Common Workflow Language User Guide — Common Workflow Language User Guide 0.1 documentation.” https://www.commonwl.org/user_guide/ (accessed Dec. 04, 2022).

[15] “kube-scheduler,” *Kubernetes*. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/ (accessed Mar. 03, 2023).

[16] “Managed Kubernetes Service – Amazon EKS – Amazon Web Services.” https://aws.amazon.com/eks/ (accessed Mar. 03, 2023).

[17] “Schedulers supported by AWS ParallelCluster - AWS ParallelCluster.” https://docs.aws.amazon.com/parallelcluster/latest/ug/schedulers.html (accessed Mar. 03, 2023).

[18] “Toil.” https://toil.ucsc-cgl.org/ (accessed Mar. 03, 2023).

[19] “Overview: Research in the cloud on Terra,” *Terra Support*. https://support.terra.bio/hc/en-us/articles/360022714931-Overview-Research-in-the-cloud-on-Terra (accessed Dec. 04, 2022).

[20] “Google Cloud documentation | Documentation.” https://cloud.google.com/docs (accessed Mar. 03, 2023).

[21] “Foreword — Flask Documentation (2.0.x).” https://flask.palletsprojects.com/en/2.0.x/foreword/#what-does-micro-mean (accessed Dec. 04, 2022).

[22] “FAQ,” *OpenAPI Initiative*. https://www.openapis.org/faq (accessed Dec. 04, 2022).

[23] “Features - FastAPI.” https://fastapi.tiangolo.com/features/ (accessed Mar. 03, 2023).

[24] “Models - Pydantic.” https://docs.pydantic.dev/usage/models/ (accessed Feb. 21, 2023).

[25] “Celery Background Tasks — Flask Documentation (2.0.x).” https://flask.palletsprojects.com/en/2.0.x/patterns/celery/ (accessed Dec. 04, 2022).

[26] “Documentation,” *Redis*. https://redis.io/docs/ (accessed Mar. 03, 2023).

[27] “Flower - Celery monitoring tool — Flower 1.0.1 documentation.” https://flower.readthedocs.io/en/latest/ (accessed Mar. 03, 2023).

[28] “Elastic Stack: Elasticsearch, Kibana, Beats & Logstash,” *Elastic*. https://www.elastic.co/elastic-stack (accessed Mar. 03, 2023).

[29] “GlobalInterpreterLock - Python Wiki.” https://wiki.python.org/moin/GlobalInterpreterLock (accessed Mar. 03, 2023).

[30] L.-H. Hung *et al.*, “Holistic optimization of an RNA-seq workflow for multi-threaded environments,” *Bioinformatics*, vol. 35, no. 20, pp. 4173–4175, Oct. 2019, doi: 10.1093/bioinformatics/btz169.

[31] L.-H. Hung, X. Niu, W. Lloyd, and K. Y. Yeung, “Accessible and interactive RNA sequencing analysis using serverless computing,” Bioinformatics, preprint, Mar. 2019. doi: 10.1101/576199.

[32] “Amazon EC2 T2 Instances – Amazon Web Services (AWS),” *Amazon Web Services, Inc.* https://aws.amazon.com/ec2/instance-types/t2/ (accessed Mar. 03, 2023).

[33] “Amazon EC2 C5 Instances — Amazon Web Services (AWS),” *Amazon Web Services, Inc.* https://aws.amazon.com/ec2/instance-types/c5/ (accessed Feb. 13, 2023).

[34] “Burrows-Wheeler Aligner.” https://bio-bwa.sourceforge.net/ (accessed Feb. 21, 2023).

[35] “Key concepts and definitions for burstable performance instances - Amazon Elastic Compute Cloud.” https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-credits-baseline-concepts.html (accessed Feb. 26, 2023).