Supporting bioinformatics analysis using a hybrid cloud and HPC architecture

Patrick McKeever

A thesis submitted
in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2025

Committee:
Ka Yee Yeung
Ling Hong Hung

Program Authorized to Offer Degree:
Computer Science and Systems

University of Washington

Abstract

Supporting bioinformatics analysis using a hybrid cloud and HPC architecture

Patrick McKeever

Chair of the Supervisory Committee:
Ka Yee Yeung
Department of Microbiology & Tacoma School of Engineering and Technology

The exponential growth of next-generation sequencing data requires novel strategies for storage, transfer, and processing of said data. We present a scheduler a based on the Temporal.io workflow framework which enables two key optimizations of bioinformatics workflows. Firstly, we enable users to transparently map workflow steps to diverse execution environments, including high-performance computing (HPC) resources managed by the SLURM resource manager. When tested on a Bulk RNA sequencing workflow, this feature allows a 26% reduction in credit consumption on the NSF Bridges 2 supercomputer by performing adapter trimming locally and all other steps on the supercomputer. Secondly, we enable asynchronous execution of workflows, a feature which guarantees that workflows will achieve reasonable resource utilization even when the scheduler cannot make use of a system's full RAM and CPU resources. When benchmarked on the same Bulk RNA sequencing workflow, this optimization facilitates a reduction in workflow makespan of between 13% and 23%, depending on the exact workflow configuration. Taken together, these features will enable reductions in the cost and time requirements of bioinformatics pipelines for researchers.

1 Introduction.

The advent of next-generation sequencing technology has resulted in significant reduction in the cost of genome sequencing, such that the cost of sequencing an entire genome has declined from roughly $95,000,000 at the turn of the century to just under $600 today [1]. Over the past two decades, genome sequencing has and continues to decline at a rate significantly faster than reductions in the price of computer storage as predicted by Moore's Law, posing significant problems for the storage and transfer of genome data. As of June 2023, the European Nucleotide Archive housed 32 petabases of raw reads, and one study forecasts that new sequencing data could be produced at a rate of 42 exabytes per year by 2025. Genomic data has attained such a volume that researchers at the Beijing Genomic Institute find it more efficient and economical to manually transport hard drives than to transmit genomic data over a network [2].

Researchers have adopted a variety of techniques to cope with the rapidly increasing scale of genomic data, mostly involving the use of cloud and/or high performance computing (HPC) for big biomedical data analyses. Cloud computing's pay-as-you-go pricing has made it attractive to researchers over competing models such as grid computing or supercomputers, which require substantial investment in order to maintain a private grid or cluster. However, supercomputers continue to exhibit benefits over cloud-based approaches for certain workloads. In particular, multi-tenancy, resource heterogeneity, and latency resulting from message passing between nodes and multi-tenancy can make supercomputers a more efficient alternative to cloud-based approaches for tightly-coupled parallel programs. Additionally, academic researchers can request credits and/or allocations from NSF-funded supercomputers, while the costs of commercial cloud platforms could pose financial obstacles to academic researchers.

Our key idea is to develop methods and tools to enable biomedical scientists to effectively leverage both commercial cloud and HPC for big data analysis. Effectively leveraging HPC and cloud computing requires significant technical knowledge. Users may wish to deploy the entirety of a bioinformatics workflow to a particular platform or may instead deploy different stages across different platforms so as to minimize cost or enhance parallelism.

This thesis presents a scientific workflow scheduler based on the Temporal.io framework [3]. This work encompasses three main contributions relative to the existing literature in workflow design. The first is the introduction of workflow language constructs to enable asynchronous execution of scientific workflows. As shown in the work below, the ability to begin executing a successor step in the workflow as soon as its predecessor has processed a single sample, rather than waiting for all samples to process, and the ability to constrain the number of samples processed at a given time allows the user to introduce optimizations that would be impossible in popular Scientific Workflow Management Systems (SWMSs) such as Nextflow [4] or Snakemake [5]. Secondly, our scheduler allows users to transparently

distribute different workflow steps to different computational environments, including both to temporal workers running on users' own servers of the cloud and remote SLURM clusters. Finally, the use of the Temporal framework and its guarantees of durable execution render workflows executed by the scheduler resilient to worker failure. We substantiate our claims of improved workflow execution by benchmarking a bulk RNA sequencing workflow, showing appreciable makespan improvements (between 13% and 22%) when using asynchronous execution constructs. Additionally, we show how cross-platform scheduling of the bulk RNA sequencing workflow across local compute resources and remote HPC clusters can reduce consumption of compute credits, allowing better scaling of data-intensive bioinformatics pipelines. In one benchmarking result, we reduce the HPC credit consumption of a Bulk RNA sequencing pipeline by 25% simply by performing adapter trimming locally.

2 Literature Review.

2.1 Asynchronous behavior in SWMSs.

Scientific Workflow Management Systems (SWMS) enable the creation and execution of standardized pipelines for data analysis. Examples of SWMSs include Nextflow [4], Snakemake [5], Cromwell [6] (with the associated Workflow Definition Language [WDL]), Taverna [7], Galaxy [8], and the BioDepot Workflow Builder [9] (BWB). Nearly all major SWMSs implicitly or explicitly represent scientific workflows as directed acyclic graphs (DAGs) in which each node represents a computational step and each edge represents data dependencies between steps. When discussing scientific workflows, it is often relevant to distinguish between an *abstract DAG* and a *physical DAG*. A node in an *abstract DAG* represents a particular type of job, whereas a node in a *physical DAG* represents a concrete workflow command that will be executed; for instance, a single node of an *abstract DAG* may correspond to several nodes in a *physical DAG*, each of which executes the same command on a different input file. An SWMS will parse iterative, conditional, and recursive structures in order to compile an *abstract DAG*, specified either through a graphical user interface (as in Galaxy, Taverna, or BWB) or through a domain-specific language (DSL, as in Nextflow, Snakemake, and Cromwell / WDL), into a physical DAG.

Popular SWMSs provide varying degrees of support for asynchronous behavior, of which we consider two types: (1) non-blocking execution and (2) dynamic workflow definitions. We define *non-blocking execution* as the ability to begin successor jobs before predecessor jobs have processed every input. For example, if step B is a successor of step A and step A produces outputs for several files, asynchronous behavior would allow for step B to begin executing once step A has produced output for the first file rather than waiting for the entire batch to complete. By *dynamic workflow definition*, we refer to the ability to execute a workflow whose physical DAG may not be known when execution begins; examples of

such behavior include branching within a DAG based on runtime conditionals or iteration over a set of files whose cardinality is not known until execution.

Most major DSL-based SWMSs allow non-blocking execution. Snakemake inherently supports such behavior through the definition of rules, each of which contains regular expressions giving the paths of input and output files, which yield a DAG; the execution of jobs are ordered based on user-specified priorities, the distance of a job from the DAG's sink node, and the size of input files in that order, and wildcard-based rules can begin executing as soon as files matching the input regular expression become available [5]. The Workflow Definition Language (WDL) likewise supports non-blocking behavior through its scatter-gather paradigm, in which scatter blocks allow iteration and parallelization over arrays of files; the inclusion of multiple tasks within a scatter block will allow for non-blocking execution of some set of tasks [10]. In Nextflow, DAG nodes can send and receive data over links in the graph, but, while sending is an asynchronous operation, receiving is asynchronous, allowing for non-blocking execution [4]. Among graphical SWMSs, support is more limited. Neither Galaxy nor Taverna nor Orange3 support non-blocking execution as described above [11].

Fewer SWMSs provide support for dynamic workflow definitions. Snakemake requires that the entire physical DAG, including input files and expected output files for each DAG node, be known before beginning execution. Nextflow allows workflows to transmit sets across channels whose cardinality is not known until runtime, but it requires that this set cardinality be known before allowing execution of subsequent nodes in the DAG, disallowing key forms of asynchronous behavior. Furthermore, Nextflow does not support the purging of temporary files during workflow execution or the resumption of workflows from temporary file sets.

2.2 Cross-Platform Execution in SWMSs.

As with asynchronous behavior, DSL-based SWMSs provide greater support for *cross-platform execution* than their graphical counterparts. By *cross-platform execution*, we mean the ability to transparently assign certain workflow steps to different execution environments; for instance, users may desire that some steps be executed locally while others be executed on the cloud or on remote SLURM clusters. This sort of distribution could be implemented in any major SWMS, but only some allow for this to be done *transparently* - that is, without requiring the user to explicitly encode logic for file transfers or resource manager interactions in their workflow definition. Nextflow natively supports cross-platform execution by allowing users to assign jobs to particular *executors*, and it provides implementations for SLURM, Kubernetes, HTCondor, and AWS Batch, among others. Depending on the executor chosen, Nextflow may handle file staging either through point-to-point transfers or by using a shared S3 bucket [12]. Snakemake, as of version 8, implements *executor plugins*

and *storage plugins*, which allow the user to transparently map Snakemake rules to different execution environments and storage services (e.g. local disk, S3, sharepoint) without altering their workflow definitions [13]. Cromwell supports a similar interface though only currently implements *backends* for Google Cloud, GA4GH TES, and a variety of HPC resource managers (including SLURM and HTCondor) [14]. By contrast, Taverna, Galaxy, and the BioDepot Workflow Builder require manual implementation of file transfer logic in order to interact with outside resource managers.

2.3 Workflow Scheduling.

Researchers have proposed a variety of algorithms to schedule DAG-structured workflows across heterogeneous compute resources. One subset of these algorithms attempts to minimize makespan based on assumptions of task duration and data transfer times; such algorithms include both heuristic-based approaches (e.g. Heterogeneous Earliest Finish Time [15]), the Approximation Algorithm of Papadimitriou and Yanakakis [16]) and search techniques (e.g. ant-colony systems [17], Tabu search [18], simulated annealing [19]). However, inferring task runtime remains difficult, and these algorithms tend to assume knowledge of the physical DAG at scheduling time. Consequently, SWMSs tend to rely on other approaches. Snakemake's default scheduler, for instance, determines a schedule through an approximate solution to the Multi-Dimensional Knapsack problem; each resource type (e.g. RAM, CPU cores) represents a dimension of the knapsack, and a greedy algorithm attempts to maximize a reward function based on user priorities, distance from the DAG sink, and input file size [5] [20].

In modern workflow management architectures, the responsibility for scheduling workflow jobs is divided between the Scientific Workflow Management System (SWMS) and the Resource Manager [21]. The former handles the parsing of workflows into Directed Acyclic Graphs and handles the overall control flow of the workflow, determining which jobs must be executed and with what parameters. All major SWMSs allow users to annotate jobs with associated hardware constraints, such as required RAM, disk space, CPU threads, and GPUs. The Resource Manager receives individual jobs and associated hardware constraints from the SWMS and schedules them on available compute resources. Examples of commonly-used Resource Managers include SLURM, Kubernetes, and HTCondor. The division of scheduling between SWMSs and Resource Managers means that Resource Managers are often unable to optimize schedules based on DAG structures. Nextflow, for example, submits workflow jobs to Resource Managers ("executors", in Nextflow terminology) as soon as all predecessor jobs have executed; the Resource Manager, in turn, determines the distribution of jobs between computational resources and the order of execution. In general, Resource Managers provide only limited support for scheduling based on overall workflow structure. The default Kubernetes executor for nextflow simply executes jobs in

a round-robin fashion. SLURM allows users to submit jobs alongside specifications of their dependencies (i.e. predecessors in the DAG), but this strategy implicitly assumes that the physical DAG is known at the time of job submission, making it unsuitable for nextflow's dynamic workflows. HTCondor DAGMan (used alongside Pegasus) allows far more granular control, such that users can submit a specification of the DAG structure and explicitly assign jobs to compute resources based on Machine, NodeType, and Rank. However, this too can only support static workflow definitions [21].

Various researchers have proposed strategies to better leverage knowledge of workflow structure during Resource Managers' scheduling. The Hi-WAY workflow scheduler supports execution of workflows from several SWMSs (DAGMan, Galaxy, and Cuneiform) via Hadoop and Yarn. During workflow execution, Hi-WAY maintains a database (Provenance Manager) of such metrics as job runtime and cross-job data transfer costs; this historical data may be leveraged to support advanced workflow-aware scheduling algorithms such as Heterogenous Earliest Finish Time (HEFT). The major shortcoming of Hi-WAY is its inability to support dynamic workflow schedules, as it assumes that the physical DAG is known once execution begins [11]. Lehmann et al (2023) propose a standardized API facilitating communication between SWMSs and resource managers which can support dynamic workflows. Using this API, an SWMS submits periodic requests to the Resource Manager specifying the preliminary form of the physical DAG, which is updated according to conditionals and iteration managed by the SWMS. A proof of concept, implemented using Nextflow and Kubernetes, showed that workflow-aware scheduling strategies based on task rank (distance of a job from the end of the critical path) outperformed non-workflow-aware scheduling strategies on six of the nine most popular nf-core workflows [21].

2.4 Temporal.

Temporal is an open-source workflow engine with an emphasis on fault tolerance and durable execution [3] [22]. Temporal workflows are defined as code in any language for which Temporal's maintainers provide an SDK. Temporal provides substantial fault tolerance guarantees by tracking a workflow instance's history on a central server; this history includes the workflow's inputs, signals and updates received from outside the workflow, activities invoked by the workflow and their results, and child workflows initiated during workflow execution. Temporal enforces the key constraint that workflows are deterministic, meaning that they produce behave identically when presented with the same event history; non-deterministic behavior must be executed within workflow activities. This allows the temporal server to serialize and reconstruct the state of a running workflow at some later time by *replaying* its event history; in a *replayed* workflow, Temporal will not re-execute completed activities, running only the workflow code needed to process their results and restore state. This allows Temporal to periodically retry failed workflows with some backoff. Likewise, users

can suspend workflow execution while awaiting signals or updates from outside the workflow, and the workflow state will be quickly restored based on the event history once the relevant signal occurs, without a worker needing to track workflow state in the interim. These features, which Temporal often describes as *durable execution*, allow for interactive and long-running workflows, in contrast to most SWMSs.

Temporal distributes workflow execution through a publish-subscribe model. A temporal server stores running and pending workflows and activities, each of which is associated with a particular queue. Temporal workers periodically poll the server for unscheduled activities and workflows and begin executing them. Furthermore, if a workflow's execution is suspended for a significant amount of time (e.g. by waiting on a long-lasting signal), a worker may cease processing a given workflow, allowing it to be rescheduled on a different worker on the same queue once the workflow's state changes. If no workers are listening to a particular queue at the time an activity is submitted to it, the workflow simply suspends execution until a worker has processed the activity. This model ensures that Temporal workflows are impervious to worker failures and relieves the server from needing to track available workers at any given time.

Despite its prevalence in industry, Temporal has not seen widespread adoption in bioinformatics, for two main reasons. The first is the lack of a SWMS-style DSL. With tools such as Nextflow or Snakemake, users only require basic knowledge of shell scripting to begin writing bioinformatics pipelines, and, even lacking this, they can select from a wide variety of open-source bioinformatics workflows in those DSLs (e.g. nf-core). By contrast, implementing workflows in temporal requires substantial knowledge of Temporal's architecture and SDKs. Second, Temporal's scheduling mechanisms are poorly-suited for computationally-intensive scientific workflows. Temporal provides only very coarse guarantees regarding scheduling. Its default behavior is to reserve a number of "slots" for each task queue to which a worker subscribes and to accept new tasks from the relevant queue until each is filled. Users may configure the maximum number of concurrent tasks and the number of slots per queue. Temporal offers no way to specify the expected memory or CPU usage of jobs within a particular task queue, nor does it allow users to specify task priorities. Furthermore, because Temporal reserves slots before polling even begins and because workers have no knowledge of a task queue's contents before polling, it is impossible to reserve system resources only after a job is ready to execute. These present substantial limitations for scientific workflows, where the memory and CPU requirements of workflow steps can vary substantially [23]. Temporal does provide an alternative option in which users can specify target RAM and CPU requirements for a worker, allowing workers to limit the acceptance of new tasks while approaching the limit; however, this too is unsuitable for scientific workflows, where a single task can consume all of a worker's available memory or CPUs.

2.5 BioDepot Workflow Builder.

The BioDepot Workflow Builder (BWB) is a graphical bioinformatics workflow manager with an emphasis on user-friendliness [9]. Users can build bioinformatics pipelines (represented as DAGs) through simple click-and-drag interaction. The BWB interface provides a variety of pre-made widgets for common bioinformatics tasks, and users may customize widgets' parameters according to their particular needs. They may also configure a widget to execute repeatedly in order to process multiple sets of inputs (iterable behavior), such as looping over a set of samples. Internally, each widget is implemented as a Docker container, enabling cross-platform portability.

One major limitation of BWB in its current state is its lack of a resource-based scheduling strategy. In contrast to other SWMSs, which allow users to specify memory and CPU constraints for particular jobs, BWB relies on a rudimentary semaphore system. A user may set a max limit on the number of instances ("workers" in BWB's terminology) of a widget which can run concurrently. This seriously hinders the portability of BWB workflows, since the number of instances of a program that may run concurrently will vary according to the memory and CPU capacities of the host machine. Furthermore, this system offers no way to constrain resource usage across multiple jobs; worker requests function on a per-widget basis. This presents a serious issue in the case where the workflow DAG contains two resource-intensive paths from source to sink, forcing users to submit very conservative worker requests.

3 Methodology.

3.1 Workflow Definitions and Configuration.

The scheduler interprets workflow definitions based on a JSON format closely modeled after BWB's workflow format. This similarity allows us to easily adapt existing BWB workflows into the scheduler's format, but the format is sufficiently general to express workflows from other SWMSs. We hope to eventually create a workflow "interchange format" based on a superset of the existing format in order to support Nextflow and Snakemake workflows as well.

A workflow definition specifies a abstract DAG in terms of nodes and links. Each node is assigned a unique identifier, a title, a docker image, estimated CPU and memory requirements for the container, a set of default parameters, a base command, and a specification of how parameters will be represented in the workflow command (e.g. as bash flags, direct arguments, environment variables, etc.); this provides the necessary information to generate a container command for a node given its parameters. Each link specifies a source node, a sink node, a source parameter name, and a sink parameter name. Once the source node of a given link completes execution, the value of the link's source node parameter generated by

that run of the source node will be transferred over the link and will overwrite the default value of the sink node's parameter. The value sent from the source node may be hardcoded within its parameters or may instead be dynamically set at runtime. (Following a convention from BWB, any file written to the container's "/tmp/output" directory is considered to represent an output parameter whose name is equivalent to the filename and whose value is equivalent to the file's contents.)

Workflow definitions can also specify iterable and asynchronous behavior within workflow nodes. An *iterable node* is one which executes multiple commands given a single set of inputs. A common use case involves executing a command once for every sample in a dataset, though the format inherited from OWS also supports more advanced use cases, such as iterating over every pair of associated paired-end FASTQ files based on a particular naming convention. Building on this, our workflow format introduces the concepts of *asynchronous* and *barrier* nodes. *Asynchronous nodes* are a subset of iterable nodes which generate a new set of outputs for each command executed. This allows them to trigger successor nodes once for every iteration of the source node such that successors are not blocked while waiting for their predecessors to process every sample in a dataset. Users may additionally identify a descendant of an asynchronous node as its "barrier"; this means that the barrier node will wait until all ancestors of itself which are descendants of the asynchronous node have completed an iteration corresponding to each iteration of the asynchronous node before the barrier node itself will generate a command and begin execution. We call these intermediate nodes which are descendants of the asynchronous node and ancestors of the barrier the *asynchronous descendants* of the asynchronous node. The workflow definition allows users to place an upper limit on the number of concurrent executions of an asynchronous node whose asynchronous descendants have not yet completed, which we call the *maximum asynchronous concurrency*; once this limit is met, the scheduler will exclusively focus on scheduling asynchronous descendants of executed asynchronous nodes rather than scheduling new asynchronous nodes. This is useful for controlling allotments of disk space which are held and released across multiple nodes within an asynchronous block.

3.1.1 Example: Bulk RNA sequencing data analysis.

To help clarify this terminology, figure 1 presents an abstract DAG for a simple Bulk RNA sequencing workflow alongside the physical DAGs of synchronous and asynchronous implementations of this workflow. Bulk RNA sequencing (RNA-seq) is a well-established high-throughput technology used to quantify gene expression levels with applications in cancer research, dignostics, drug development and clinical diagnosis. The analysis of RNA-seq data typically consists of several steps. Raw sequence data are typically stored in FASTQ format that are pre-processed by trimming of reads to remove low quality reads and adapter sequences. Next, these trimmed reads in FASTQ format are aligned to a reference genome

or transcriptome resulting in BAM files. Finally, gene or transcript expression levels are computed in the quantification step, resulting in a counts table.

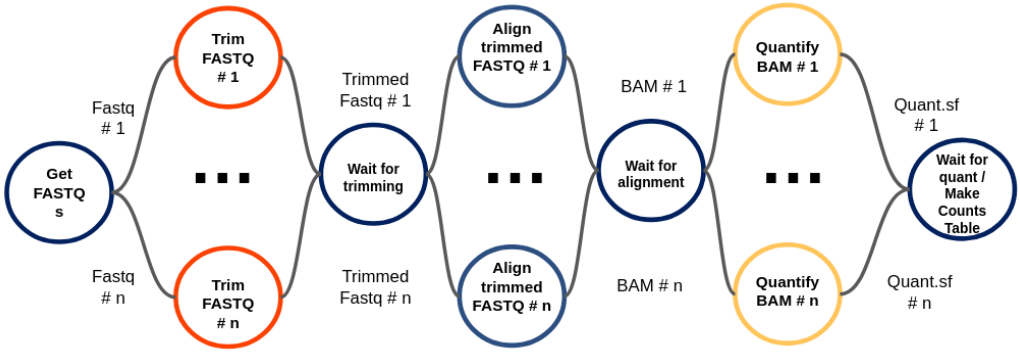
In the synchronous version, the workflow will begin trimming all reads of all samples, wait until all samples are trimmed, begin aligning each read of each trimmed sample, wait until all samples are aligned, begin quantifying each sample, wait until all samples have been quantified, and create a final counts table. In the asynchronous version, a given sample may be aligned (quantified) as soon as it has been trimmed (aligned), regardless of which steps have been completed for other pipelines. In this context, we would say that the "Trim FASTQ", "Align trimmed FASTQs", and "Quantify BAM" steps are *asynchronous descendants* of the initial asynchronous node, as each will execute a new command for each set of outputs yielded by said node. The final "Make Counts Table" node is a *barrier* node, as it will wait until all samples have been quantified before executing. Were the user to specify a *maximum asynchronous concurrency* with value m for this pipeline, the pipeline would ensure that no more than m samples are being trimmed, aligned, or quantified at a given time.

Whereas workflow definitions allow the scheduler to generate workflow commands, workflow configuration files allow users to place constraints on their scheduling and execution. Specifically, users may assign particular nodes in the graph to be executed on particular resource managers. At present, the only resource manager supported is SLURM, but the scheduler uses an abstract implementation that could easily be extended to others. By default, a node is executed on the *local executor*, meaning that the scheduling workflow will assign it to one of the subscribed Temporal workers according to a simple vector packing algorithm (see "Scheduler Implementation"); if no configuration is provided, the scheduler will execute all nodes in the graph in this manner.

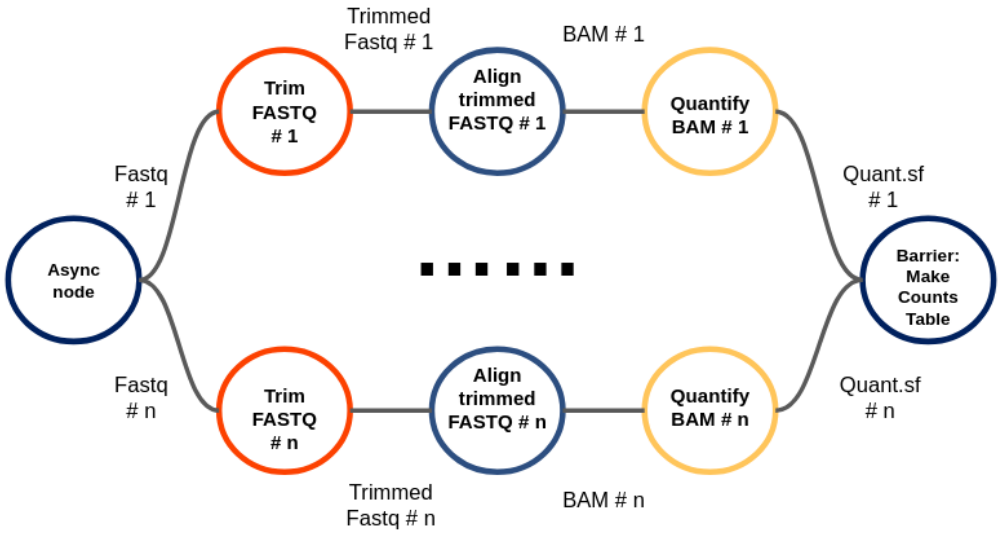
3.2 Scheduler Implementation.

We implement our workflow scheduler as a Temporal workflow which receives a workflow definition, workflow configuration, and execution mode (local or non-local) as its arguments. This workflow (which I will refer to as the "main" workflow to distinguish it from the child workflows responsible for scheduling and SLURM execution) is responsible for parsing the workflow definition, generating the relevant workflow commands, and submitting these commands to Temporal activity queues for execution on workers or on specialized resource managers. Because Temporal's default scheduling behavior is insufficient for most bioinformatics workflows, we implement a push-based scheduling scheme which tracks worker liveliness and assigns tasks to worker using first-fit decreasing bin packing.

(a) An abstract DAG for a simplified Bulk RNA sequencing workflow.



(b) A physical DAG for the same Bulk RNA sequencing workflow using iterable, synchronous behavior.



(c) A physical DAG for the same Bulk RNA sequencing workflow using asynchronous behavior. The initial "Get FASTQ files" node is *asynchronous*, because its asynchronous descendants - the nodes occurring after itself but before the corresponding *barrier* node ("Trim FASTQ", "Align trimmed FASTQ", and "Quantify BAM") - will can proceed individually through the trimming, alignment, and quantification stages without waiting until all samples have passed through a given stage to begin the next. The "Make Counts Table" node serves as a barrier, because it waits for the trimming, alignment, and quantification of all samples before executing.

Fig. 1: Synchronous and asynchronous implementations of bulk RNA sequencing.

3.2.1 Basic Architecture.

Execution of a workflow involves five components, three essential and two optional: a client, which submits a workflow request; the Temporal server, which receives this workflow request and assigns it to a worker; a set of temporal workers, who receive and process the workflow and its constituent activities; an optional S3 server (supporting any implementation of the S3 API, including open-source options such as minio) to coordinate file sharing between workers and store final results; and optional external executors such as SLURM, which may receive and execute particular workflow tasks in lieu of the temporal workers. Figure 2 provides a diagram of these components.

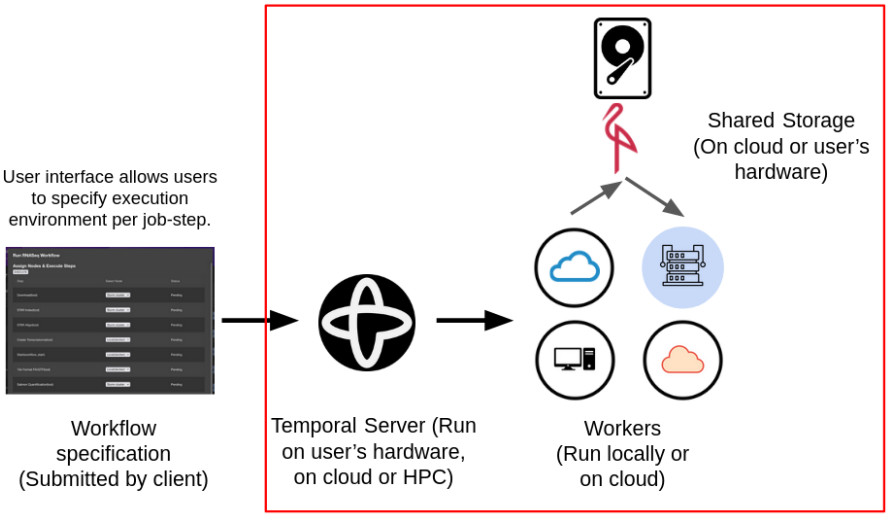


Fig. 2: Architecture diagram of the scheduler. Those items in the red box were designed as part of this thesis. The client and user interface remain works in progress and are outside the scope of my work.

Our implementation distinguishes between two types of temporal workers - *scheduler workers* and *regular workers*. A *regular worker* executes the actual commands of a workflow. It has a unique queue name and is initiated by the user with limits on RAM, CPU, and GPU usage that will be respected during task scheduling. When the scheduler assigns a command to a regular worker, activities in order to stage the command's prerequisite files, execute the command itself, and upload the results are routed to the worker's temporal queue. In contrast, *scheduling workers* execute tasks that do not require resource grants, such as the generation of workflow commands, the tracking of workflow inputs and outputs, and the assignment of workflow jobs to workers. As none of the tasks assigned to *scheduling workers* depend on file locality, they share a common queue. This division ensures that tasks responsible for a running workflow's control flow do not wait in Temporal queues

behind workflow jobs.

As discussed in the literature review, Temporal's default publish-subscribe model, which relies on slots-based rate limiting, is a poor fit for cases involving variable resource requirements between jobs. We implement a push-based job system as part of the core workflow logic while still obeying Temporal's requirement for deterministic workflow behavior. In our system, the main workflow initiates a child workflow responsible for the assignment of outstanding jobs to workers. This child workflow is submitted to the shared queue for scheduler workers. Each job, specified by a unique ID alongside a required allotment of memory, CPU cores, and GPUs, is communicated from the main workflow to the child workflow via a temporal signal, and the child workflow's responses are communicated back to its parents via signals. Likewise, regular workers announce their existence to the child workflow via temporal signals. Upon receiving a signal from a worker, including its unique queue ID, the scheduling child workflow starts a heartbeat activity on the subscribed worker's unique queue. (More specifically, each regular worker maintains a distinct queue for heartbeat activities so as to ensure that these activities are not preempted by workflow commands in the temporal slot system.) Using a temporal primitive, this heartbeat activity will continue indefinitely so long as the worker communicates its liveliness to the server every 20 seconds; failing that, a workflow exception will be triggered, and the child workflow will register the worker as dead. It will then attempt to reschedule the heartbeat activity to the worker queue using an exponential backoff. Periodically, the child scheduling workflow will execute an activity to assign outstanding job requests to workers, tracking the resources remaining on each worker.

This periodic logic motivates our decision to implement the scheduling workflow as a child workflow rather than an integral part of the main temporal workflow; temporal workflows have an upper limit on the size of their event histories, which requires them to periodically "Continue-As-New", an operation which involves terminating ongoing processes and reconstructing workflow state in a new workflow. By segregating periodic logic into child workflows, we limit the size of the main workflow's event history and circumvent the need to stop and resume ongoing workflow commands once the job history grows unmanageably large. When the scheduler child workflow's event history reaches some pre-defined size limit (by default, one-fifth of the maximum value), it cancels its ongoing heartbeat activities, serializes its state, and invokes the "Continue-As-New" operation, rescheduling all heartbeat activities within this new workflow. Because the execution of the child scheduling workflow is determined entirely by temporal signals (from both workers and its parent workflow) and the results of activities (including exceptions triggered by failed heartbeat activities), it behaves deterministically and can be replayed by the temporal server in case the parent workflow fails and is resumed.

3.2.2 Scheduling.

We treat scheduling as a vector-packing problem in which each worker has a fixed allotment of CPU cores, GPUs, and memory and where each job will occupy some subset of those resources. Our approach to scheduling consists of a simple first-fit decreasing heuristic: we loop through all available workers, then loop through all outstanding job requests ordered by their most demanding resource request as a fraction of the corresponding bin dimension, and assign each job request to the first worker capable of accommodating it. The aim of such an algorithm is to assign all available jobs to as few workers as possible, thereby maximizing resource utilization and preventing resource fragmentation. Garey et al. (1976) have shown that this algorithm achieves a competitive ratio of of $d+\frac{7}{10}$ relative to the offline version, where d is the number of dimensions, which makes it suitable for low-dimensional cases [24]. In practice, most workflow jobs will operate with $d = 2$, since only a handful of jobs require GPU grants (in which case $d = 3$).

The choice of an online algorithm is motivated by several factors. Firstly, our workflow format provides support for dynamic workflow definitions, in which the precise set and number of commands to be executed is not known at the start of workflow execution and may be updated during the workflow runtime. (This enables workflows to poll for and process arbitrarily large numbers of files as they appear.) This approach is fundamentally at odds with workflow scheduling algorithms such as HEFT, which demand comprehensive knowledge of the workflow graph before execution. Secondly, we may wish to eventually support an architecture in which workers are used by multiple users concurrently; with an online scheduling algorithm already in place, this could be accommodated almost instantly by switching the scheduling child workflow into a fully fledged Temporal workflow which may be signaled by several running workflows simultaneously rather than accepting requests exclusively from its parent workflow. Finally, heuristic-based bin-packing is a substantially simpler algorithm to implement than most offline scheduling algorithms.

3.2.3 Integration with Resource Managers and HPC.

One of the major aims of our scheduler is to allow users to easily distribute workflows across multiple execution environments, executing jobs both on their own hardware, on the cloud, and on HPC. Execution on the cloud requires no special treatment; users can provision on-demand compute instances from any major cloud provider and run a temporal worker, using the same temporal server to which the client submits workflows. By contrast, execution on multi-user HPC clusters typically requires interfacing with a distinct resource manager such as SLURM or Kubernetes. Furthermore, users on these systems often lack the relevant permissions to install the dependencies necessary to run a temporal worker; the only permissions we can generally assume are the ability to submit jobs, either through a remote shell on a login node, as in most SLURM systems, or through some API, as in

Kubernetes, HTCondor, etc. To help overcome differences in resource managers' interfaces, we introduce the abstraction of an "executor". An executor is an abstract interface which provides general functionalities to run workflow jobs, retrieve their results, transfer file dependencies, and handle resource allocations on a particular resource manager. At present, we have only implemented two executors: one for execution on local temporal workers and another for executing jobs on remote SLURM clusters. However, the interface is sufficiently general that it may be easily extended to other resource managers in the future.

As with the scheduling of jobs to activities, the submission of workflow jobs to remote resource manager relies on a child workflows. Our SLURM executor submits workflow jobs to a remote SLURM cluster via an SSH connection. The management of this SSH connection introduces some difficulties. We cannot interact with networked communication within the Temporal workflow, as this would violate determinism constraints. Likewise, we cannot open a new SSH connection within each activity, since this could cause the SLURM login node to refuse connections due to an upper bound on concurrent SSH sessions for a single user. Our solution is to delegate the management of the connection lifetime to a Temporal worker. We create a special class of worker which maintains an SSH connection with a particular SLURM cluster and whose queue name is uniquely determined by the username and endpoint associated with this SSH session; the SLURM executor then assigns temporal activities to this queue, so that activities can reuse the existing SSH connection already open on the worker. Failure of the SSH connection results in temporary failure of the worker, causing the SLURM child workflow to block until a queue corresponding to the relevant cluster becomes available. It is incumbent upon the client to ensure that a worker with a connection to the given SLURM cluster is available when a workflow requests it. To make this simple, we provide a script to users which takes a workflow definition, submits it to the temporal server, automatically establishes all workers necessary to complete said workflow's execution, and terminates these workers once the workflow has completed execution. We anticipate that our user interface will provide similar functionality.

Our executor for SLURM clusters functions primarily by sending and receiving signals to and from a child workflow. The main workflow assigns jobs to SLURM through a signal to the child workflow, and the child workflow returns results and logs through a signal to its parent. Requests to execute a command on the SLURM cluster include the job's estimated resource requirements and SLURM-specific configuration information (e.g. desired partition). Upon receiving a request from the main workflow, the child workflow submits an activity to the queue of the worker responsible for maintaining the SLURM connection, which in turn submits the command to the SLURM cluster via an sbatch script. The child workflow also periodically submits a polling job to this queue, specifying all outstanding SLURM jobs. The worker responsible for maintaining the connection with said cluster then submits an sbatch command which retrieves the state of all jobs in the workflow's activity

parameters, as well as the standard output and standard error of completed activities. It is this polling activity which necessitates the segregation of SLURM handling into a child workflow, and said child workflow periodically invokes the continue-as-new API to prevent event history overflows.

SLURM clusters may enforce particular constraints on jobs' resource requests. For instance, a cluster may require that jobs' request for memory not exceed a certain fraction of their requests for CPUs. Likewise, SLURM clusters will terminate jobs which exceed their memory allotments. To cope with these complexities, we allow users to override the memory, CPU, and GPU requests specified in their workflow definition within their workflow configurations. Should they fail to do so, we submit jobs to SLURM with CPU and GPU requests unmodified and with memory requests augmented by 20%, in order to provide a cushion for jobs which slightly surpass RAM estimates. (Snakemake does the same.)

Because Docker adopts a lax approach towards user privileges, multi-user SLURM systems generally require users to use Singularity instead. Like Docker, Singularity adheres to the Open Containers Initiative interface specification, but, unlike Docker, Singularity does not require a privileged daemon in order to execute; furthermore, permissions within the host filesystem are preserved within the container. As a result, our software allows users to execute containers both as Docker and Singularity containers. If the user specifies a Docker image when running on a system which requires Singularity, we schedule a temporal activity to the scheduler queue to convert the relevant Docker image into a Singularity container using the Singularity CLI. The only substantial difference in interface which we encountered during the conversion of several workflows was Singularity's lack of support for entrypoints, which we addressed by storing each container's entrypoint in a text file alongside the resulting Singularity image. With this modification made, our conversion approach sufficed for all but one Docker image, which explicitly relied on root privileges within the container and had to be manually rewritten.

3.2.4 File Staging.

A central design goal of our scheduler is the transparent execution of file staging. This is to say that a path referenced in one container's filesystem should appear at an identical location in other containers, regardless of the worker node on which each job executes and without any requirement for explicit file transfer logic. This entails coordination between three types of file systems: container file systems, the file systems of the hosts on which worker nodes run, and a global S3 file store which coordinates transfers between worker nodes. Users may choose to run workflows in a "local storage" mode, in which case the latter is unnecessary.

The transfer of files between workflow containers follows essentially the same scheme as in BWB. Whenever a workflow container is run, we establish a container volume mapping

the "/data" directory within the filesystem to a particular path on the host filesystem, allowing containers running on the same worker nodes to share all files within this directory. Our approach differs from BWB in one only respect; we now require users to assign a unique "storage identifier" to each instance of a particular workflow, and we assign a create a new directory on the host filesystem for each unique storage ID. It is this directory which will be mounted as "/data" in any container sharing the relevant storage ID. This achieves isolation between different workflow executions, as is essential in a multi-node and potentially multi-user system, though users can reuse the same storage identifier between workflow runs in order to preserve state.

To accommodate workflow execution in non-local storage mode, users may annotate parameters of workflow nodes as corresponding to input files and output files of particular steps. As the values of these parameters may differ between different iterations of the same node, this allows the scheduler to download only the files on which a workflow step directly depends before executing said step. Once the scheduler workflow receives a resource grant for a particular step, it will schedule an activity to the corresponding queue which will begin downloading all relevant dependencies, and, once this command has completed, an activity to upload these output files will be submitted to the same queue. Crucially, this latter file upload step begins executing after the scheduler workflow has released the resource grant corresponding to the finished command, allowing other activities to use the worker node's CPU and RAM resources while the IO-bound file upload completes. To ensure transparency of file staging, we implement a unified naming scheme. The user assigns each workflow an identifier (which may be reused between workflows) which serves as the name for a particular bucket in the workflow's data store. Every path within workflow containers with the prefix "/data" is mapped to a corresponding file in the workflow's bucket sharing the same suffix; for instance, a container path "/data/processing/genome/SA" would correspond to path "[WORKFLOW_BUCKET]/processing/genome/SA" within the data store and to the path "[SCHED_STORAGE_DIR]/processing/genome/SA" once downloaded onto the worker node. This system of transparent file transfers suffice for most simple use cases, but, should the user wish to conduct file transfers across multiple S3 buckets, they have the option of manually specifying this behavior through pre-built workflow widgets, as in BWB. Because BWB's workflow format does not manually specify the inputs and outputs of each step, it is necessary to manually annotate workflow definitions in order to support non-local execution.

The distribution of files across multiple workflow nodes raises certain issues relating to file consistency. One particular case of interest is that a workflow node's input or output file corresponds to a directory rather than a regular file. In this case, we adopt the convention that the scheduler will recursively upload any files in the directory that are present locally but not on the data store and vice versa for downloads. If a file is already present on the

data store before uploading, we first check that the local and remote versions share the same length in bytes, as this property is available from the S3 API, and upload the local version only if these lengths differ (likewise for downloads). However, this is a straightforwardly suboptimal solution, and we intend to eventually store file hashes within a scheduler database.

When transferring files to and from SLURM, the default behavior is to schedule an activity to the SLURM worker (which maintains the SSH connection with the login node) which transfers the relevant files using rsync. Each running workflow job may schedule such a task, so several rsync transfers may be active at any given time, improving throughput. If multi-node execution is enabled, files downloaded from SLURM are then uploaded to the shared file store. This too is a suboptimal approach and could be greatly improved upon through support for globus, which would allow transfers directly to and from the data store.

4 Evaluation.

In this section, we benchmark two workflows in order to show the benefits of the scheduling approaches laid out above. In all cases, the bin-packing-based scheduling heuristics outperforms the static, slot-based approach of the BWB scheduler, as would be expected. More significantly, we show that the application of novel scheduling constructs can achieve appreciable improvements in makespan by adjusting just a few workflow parameters.

4.1 Benchmarking environment.

The benchmarks discussed in the following section are conducted on two environments. The first is a local UWT server whose RAM, CPU, and disk capacities are shown in Table 1. Although the server possesses various SSDs, we relied exclusively on its 70 terabyte hard disk for benchmarking, as the other SSDs are not sufficiently large to contain the full outputs of the Bulk RNA sequencing workflow. During benchmarking, we constrain both the scheduler and BWB to consume no more than 75% of available RAM and CPUs. This models a fairly common use case on multi-user servers, where users may not wish to achieve full CPU utilization. All benchmarks were conducted on the server's hard disk. These choices are meant to faithfully reproduce the conditions during our team's usage of the Bulk RNA sequencing workflow.

In addition, some of our benchmarks are conducted on remote HPC clusters, for which we use the NSF Bridges2 supercomputer. We obtained an NSF grant of 200,000 credits on this supercomputer. Credit consumption is billed based on CPU usage, with one CPU hour corresponding to a single credit in most partitions of the SLURM cluster. (Pricing may vary in high-demand clusters, particularly those with GPUs.) Users may also expend credits to reserve space on the cluster's highly parallel Lustre filesystem, with one credit

	CPUs	RAM	Disk Capacity
Total	32	128 GB	60 TB
Used	24	96 GB	1.3TB

Tab. 1: Benchmarking statistics for UWT server. Disk usage reflects the total amount consumed rather than a hard limit.

corresponding to a single gigabyte of storage. We obtained a 10 terabyte allotment of disk space. For all benchmarks, we submitted jobs to the RM-shared partition, a multi-user partition consisting of several hundred node, whose statistics are detailed in Table 2. This partition requires that user job requests have a memory-to-CPU ratio not exceeding ten gigabytes. In cases where the RAM and CPU requirements used for local execution did not obey this constraint, we simply increased CPU requirements to the minimum value which would achieve such a ratio.

Partition	Nodes	CPUs Per Node	RAM Per Node
RM-Shared	396	128	256 GB

Tab. 2: Benchmarking statistics for RM-shared partition of NSF Bridges2 supercomputer.

4.2 Data.

For benchmarking of the Bulk RNA pipeline, we use the published GSE287843 dataset [25] that aims to study the effects of knocking out two transcriptor factors in brain and other support cells. This is a dataset originating from the MORPHIC program. This dataset consists of 28 sets of Illumina-sequenced, paired-end FASTQ files which, when compressed, occupy 181 GB of disk space. Preliminary statistics of the dataset are summarized in Table 3.

Samples	Mean Reads per Sample	Min Reads per Sample	Max Reads per Sample	Mean Read Length
28	48,317,354	36,980,962	58,775,761	151

Tab. 3: Summary statistics of GSE287843 dataset.

4.3 Bulk RNA Sequencing Workflow.

4.3.1 Workflow Description.

The MORPHIC Bulk RNA Alignment pipeline is a BWB pipeline used to align bulk RNA read data against a reference genome. The pipeline includes a number of preliminary and preprocessing steps, such as the download of genome and transcriptome annotations, the download of input FASTQ files from an S3 bucket, and the building of a genome index, all of which may be elided if the relevant files are already present on the system. The bulk of the pipeline's computational time is spent in three steps: adapter trimming using Trimgalore, alignment to the genome using STAR, and quantification using Salmon. Each of these steps are iterable, meaning that they execute once for every sample (meaning once for every single-ended FASTQ file or once for every pair of paired-end FASTQ files) in the dataset. At the end of this process, a lightweight bash script constructs a counts table that quantifies gene expression from the salmon results. The pipeline includes two additional, optional steps: Y Chromosome removal and S3 uploading. For data privacy reasons, many cell lines require that published genomic data exclude reads from or alignments to the Y chromosome, which may contain personally identifying information of the cell line donor. This workflow step waits until alignment completes, finds all reads aligned against genes in the Y chromosome for each sample, and removes corresponding alignments and reads from the alignment file and FASTQ files. The S3 upload step simply uploads all products of the workflow (reads, trimmed files, quantification, etc.).

The estimated RAM and CPU requirements of each step are given in Table 4. The RAM requirements for STAR scale with the size of the genome used for alignment, as it keeps a suffix array and index of the entire genome in memory for the duration of the pipeline. The values given below reflect the resource consumption for a full human genome.

Workflow Step	RAM Required	CPUs Required
Trim Galore	4 GB	4
STAR Alignment	32 GB	16
Y Chromosome Removal	1 GB	4
Salmon	8 GB	4

Tab. 4: RAM and CPU requirements of steps in the Bulk RNA sequencing workflow.

4.3.2 Inefficiencies in the Existing Bulk RNA Sequencing Pipeline.

The Bulk RNA Sequencing workflow contains two major sources of inefficiency. The first pertains to the Trimming and Y Chromosome Removal steps. Although these steps do little more than iterate through the reads and alignments of FASTQ and BAM files while

selectively removing certain lines, they represent a considerable portion of workflow runtime due to the large file sizes involved as well as the compression and decompression of input files. Furthermore, there is strong evidence to indicate that these tasks are disk bound. Table 5 shows how the runtime of these tasks scales as they are allocated more resources through the BWB worker system. When doubling the amount of CPU and RAM resources assigned to the trimming step, we achieve a reduction in runtime of only a quarter. The Y chromosome removal step shows similar diminishing returns relative to resource usage. (It also has a tendency to crash when assigned more than 16 threads.) Preliminary benchmarking suggests that these steps are disk-bound, a problem exacerbated both by our use of a hard disk (requiring physical seeks) and the small buffer sizes used by compression tools such as gzip and pigz.

Workflow Step	No. Workers	Execution Time	Average Time Per Sample
Trim Galore	4	02h 04m 52s	17m 50s
Trim Galore	6	01h 38m 25s	19m 41s
Trim Galore	8	01h 32m 30s	23m 08s
Y Chromosome Removal	8	03h 28m 08s	52m 02s
Y Chromosome Removal	16	02h 23m 01s	01h 11m 31s
Y Chromosome Removal	16	Crashed	N / A

Tab. 5: Scaling of Trimgalore and Y Chromosome Removal steps relative to workers. Average running times of steps from the BWB Bulk RNA sequencing workflow using workers system. The formula for average run time per sample is $\frac{t}{\lceil \frac{s}{w} \rceil}$, in which t denotes overall running time across all samples, s the number of samples (28), and w the number of workers. Note that Y Chromosome Removal does not technically use the workers system; instead, a single process is spawned which processes all samples using a number of threads given by the user. The parallelization strategy is otherwise identical, so we make no distinction in the table above.

A second inefficiency concerns the scheduling of the STAR alignment and Y Chromosome Removal steps. The default parameters of the workflow assign STAR 16 threads. This is reasonable, provided that we are willing to schedule two STAR instances concurrently, thereby maximizing utilization of all 32 CPU cores on our server; however, in our use case, where we wish to cap system CPU utilization at 75% of the total, this results in a persistent underutilization of CPUs when STAR is running.

4.3.3 Benchmark 1. Asynchronous Local Execution of Bulk RNA Sequencing Workflow.

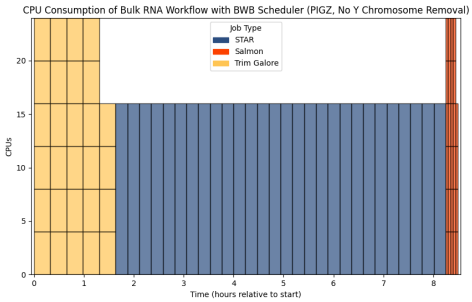
In order to address these inefficiencies in the workflow, we make use of the asynchronous execution offered by our Temporal-based scheduler. We rewrite the workflow such that trimming, alignment, and quantification may proceed in an asynchronous manner; in other words, as soon as a sample has been trimmed (aligned), the scheduler may choose to begin aligning (quantifying) it. This enables the scheduler to schedule less CPU-intensive tasks alongside alignment, achieving superior resource utilization in the middle stage of the pipeline. Additionally, we introduce a constraint that no more than two instances of Trimgalore may execute concurrently. This prevents the disk-bound inefficiencies discussed above. This latter optimization would be inefficient without asynchronous execution, as the scheduler would be unable to schedule any other workflow steps until all trimming jobs have completed, resulting in significant CPU underutilization.

To compare the approach of the Temporal-based scheduler and BWB's default worker-based system, we execute the Bulk RNA alignment pipeline in its synchronous form (using the default BWB scheduler) and in its asynchronous form (using the Temporal-based scheduler). Table 6 presents the top-line results of this benchmarking. Using asynchronous execution, we achieve a reduction in the pipeline execution time ranging between a fifth and an eighth depending on whether the Y Chromosome Removal step is included. Figure 3 shows CPU utilization over time in the synchronous and asynchronous versions of the pipeline, demonstrating how effective resource utilization in the asynchronous version achieves this decrease in runtime. Finally, Tables 7 and 8 respectively present the average runtime per step of the pipeline in versions of the pipeline run with and without the Y Chromosome removal step. (For BWB, which executes all pipeline stages synchronously, this is a meaningless distinction, but, for the Temporal-based scheduler, scheduling jobs of different types concurrently may affect average job runtime, especially as a result of disk contention.) This demonstrates that our decision to limit the concurrent executions of Trim Galore produces a noticeable decrease in that step's average runtime, ranging between 21.7% and 26.1%.

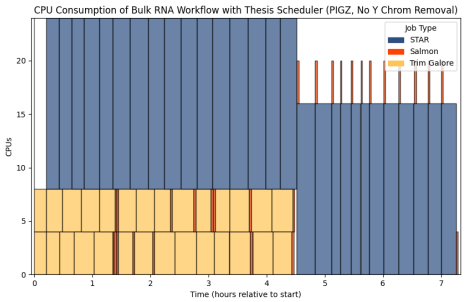
Cumulatively, these optimizations achieve runtime improvements of 12.6% and 21.78% in the versions of the workflow run with and without Y Chromosome removal.

Y Chromosome Removal	BWB Execution Time	Scheduler Execution Time	Reduction
No	08h 20m 58s	07h 17m 50s	12.60%
Yes	10h 49m 54s	08h 28m 21s	21.78%

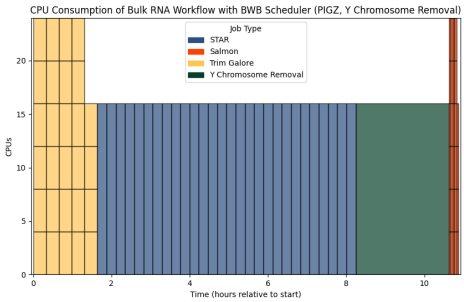
Tab. 6: Reductions in runtime due to asynchronous execution supported by our Temporal-based Scheduler (shown as "Scheduler" in the table) relative to BWB scheduler.



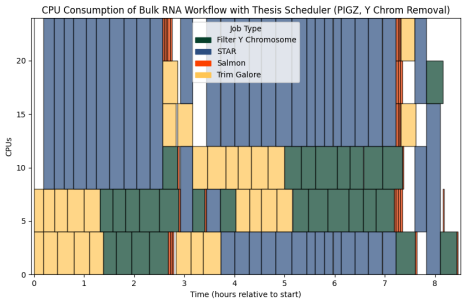
(a) CPU Utilization with BWB Scheduler, without Y Chromosome Removal



(b) CPU Utilization with Thesis Scheduler, without Y Chromosome Removal



(c) CPU Utilization with BWB Scheduler, with Y Chromosome Removal



(d) CPU Utilization with Thesis Scheduler, with Y Chromosome Removal

Fig. 3: Graphics displaying CPU utilization throughout the execution of the Bulk RNA sequencing pipeline.

Workflow Step	BWB Mean Time	Scheduler Average Time	Scheduler Maximum Time	Scheduler Minimum Time
Trim Galore	23m 08s	18m 02s	22m 33s	12m 19s
STAR Alignment	14m 09s	15m 04s	18m 58s	08m 48s
Salmon Quantification	03m 25s	01m 50s	02m 27s	01m 16s

Tab. 7: Workflow step runtime statistics from the version of the pipeline run without the Y Chromosome Removal step.

Workflow Step	BWB Mean Time	Scheduler Average Time	Scheduler Maximum Time	Scheduler Minimum Time
Trim Galore	23m 08s	17m 06s	21m 53s	11m 42s
STAR Alignment	14m 09s	14m 42s	19m 22s	09m 20s
Salmon Quantification	02m 08s	03m 03s	01m 20s	
Y Chromosome Removal	01h 11m 31s	19m 30s	25m 31s	14m 26s

Tab. 8: Workflow step runtime statistics from the version of the pipeline run with the Y Chromosome Removal step.

4.3.4 Benchmark 2. Execution of the Bulk RNA Sequencing Workflow on NSF Supercomputers.

Additionally, we benchmark the pipeline on NSF supercomputers. We test two executions, one of which executes all steps on the supercomputer and another of which executes only alignment, quantification, and generation of the counts table on the supercomputer. Figure 4 shows a breakdown of credit consumption when using the fully HPC pipeline. Table 9 shows the relative runtimes and credit consumption of these two versions of the pipeline. The simple optimization of performing trimming locally reduces credit consumption by roughly a quarter though increases the pipeline's execution time by roughly 60%, as trimming on the server presents a major bottleneck.

5 Conclusion.

Even when using such rudimentary algorithms as first fit bin-packing, allowing the user to optimize pipeline execution by exploiting domain knowledge can achieve substantial improvements in running time. In effect, our scheduler allows users a greater degree of

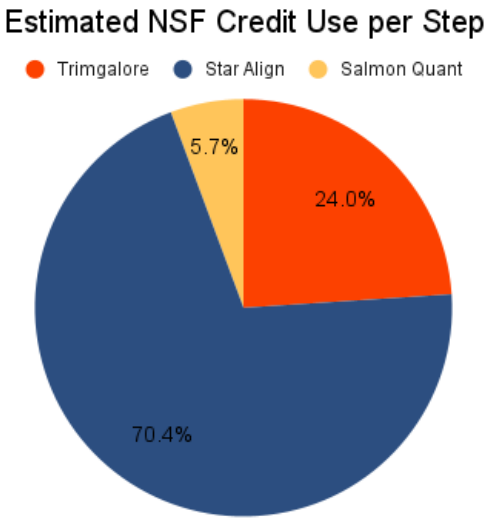


Fig. 4: Estimated credit use per step of pipeline based on conversion of SLURM job runtime to CPU hours.

Version	Running Time	Credit Usage
Full HPC	03h 01m 17s	352
Hybrid HPC	01h 53m 26s	261
Change	+ 59.81%	- 25.85%

Tab. 9: Bulk RNA sequencing workflow makespan and credit consumption. "Full HPC" refers to the version of the pipeline run entirely on an NSF SLURM cluster, while "Hybrid HPC" refers to the version in which trimming was done locally and all other steps on an NSF SLURM cluster.

control over the two key components of scheduling: job ordering and job placement. As we have demonstrated, asynchronous execution allows users to introduce optimizations on concurrent execution of workflow steps without undermining resource utilization. Similarly, our transparent handling of file transfers and resource manager interfaces allows users to easily distribute steps of the pipeline to remote HPC resources.

The use of HPC pipelines, particularly when using optimizations based on cross-platform execution, present a valuable alternative to cloud-based pipelines when scaling Bulk RNA usage. Substantially, the NSF Bridges 2 supercomputer used for benchmarking in this thesis does not charge based on data ingress or egress, eliminating a major cost of cloud execution. Assuming linear scaling of credit consumption relative to dataset size, our 200,000 credit allotment from the NSF would allow execution of 568 datasets of a similar size to GSE287843

[25] when executing all steps on HPC resources. When introducing cross-platform execution optimizations such as local trimming, this number grows to 766. In short, use of publicly-funded supercomputers will help cope with the rising size of RNA sequencing datasets while minimizing cost to individual researchers.

References

[1] Kris Wetterstrand. DNA Sequencing Costs: Data. `https://www.genome.gov/about-genomics/fact-sheets/DNA-Sequencing-Costs-Data`, 2023. [Accessed 15-03-2025].

[2] G. Reali, M. Femminella, E. Nunzi, and D. Valocchi. Genomics as a service: A joint computing and networking perspective. *Computer Networks*, 145:27–51, 2018. ISSN 1389-1286. doi: https://doi.org/10.1016/j.comnet.2018.08.005. URL `https://www.sciencedirect.com/science/article/pii/S1389128618307205`.

[3] Maxim Fateev and Samar Abbas. Temporal workflow engine. URL `temporal.io`.

[4] Paolo Di Tommaso, Maria Chatzou, Evan W Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. Nextflow enables reproducible computational workflows, April 2017. URL `http://dx.doi.org/10.1038/nbt.3820`.

[5] Johannes Köster. Parallelization, scalability, and reproducibility in next generation sequencing analysis. *Technische Universität Dortmund*, 2014. doi: 10.17877/DE290R-7242. URL `http://eldorado.tu-dortmund.de/handle/2003/33940`.

[6] Kate Voss, Geraldine Van Der Auwera, and Jeff Gentry. Full-stack genomics pipelining with gatk4 + wdl + cromwell [version 1; not peer reviewed], 2017. URL `https://f1000research.com/slides/6-1381`.

[7] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 06 2004. ISSN 1367-4803. doi: 10.1093/bioinformatics/bth361. URL `https://doi.org/10.1093/bioinformatics/bth361`.

[8] The Galaxy Community. The galaxy platform for accessible, reproducible, and collaborative data analyses: 2024 update. *Nucleic Acids Research*, 52(W1):W83–W94, 05 2024. ISSN 0305-1048. doi: 10.1093/nar/gkae410. URL `https://doi.org/10.1093/nar/gkae410`.

[9] Ling-Hong Hung, Jiaming Hu, Trevor Meiss, Alyssa Ingersoll, Wes Lloyd, Daniel Kristiyanto, Yuguang Xiong, Eric Sobie, and Ka Yee Yeung. Building containerized workflows using the BioDepot-Workflow-Builder. *Cell Syst*, 9(5):508–514.e3, September 2019.

[10] Chapter 6 Parallelization via Arrays — Developing WDL Workflows — hutchdatascience.org. https://hutchdatascience.org/Developing_WDL_Workflows/parallelization-via-arrays.html. [Accessed 05-03-2025].

[11] Marc Bux, Jörgen Brandt, Carl Witt, Jim Dowling, and Ulf Leser. Hi-way: Execution of scientific workflows on hadoop yarn. In *International Conference on Extending Database Technology*, 2017. URL https://api.semanticscholar.org/CorpusID:35175164.

[12] Overview &x2014; Nextflow documentation — nextflow.io. https://www.nextflow.io/docs/latest/overview.html. [Accessed 05-03-2025].

[13] Storage support — Snakemake 8.29.2 documentation — snakemake.readthedocs.io. https://snakemake.readthedocs.io/en/stable/snakefiles/storage.html. [Accessed 05-03-2025].

[14] Red Team at Broad Institute. Overview - Cromwell — cromwell.readthedocs.io. https://cromwell.readthedocs.io/en/stable/backends/Backends/. [Accessed 05-03-2025].

[15] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002. doi: 10.1109/71.993206.

[16] Christos H. Papadimitriou and Mihalis Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19(2):322–328, 1990. doi: 10.1137/0219021. URL https://doi.org/10.1137/0219021.

[17] Wei Chen, Jun Zhang, and Yang Yu. Workflow scheduling in grids: an ant colony optimization approach. *2007 IEEE Congress on Evolutionary Computation*, pages 3308–3315, 2007. URL https://api.semanticscholar.org/CorpusID:33694964.

[18] M. Ben-Daya and M. Al-Fawzan. A tabu search approach for the flow shop scheduling problem. *European Journal of Operational Research*, 109(1):88–95, 1998. ISSN 0377-2217. doi: https://doi.org/10.1016/S0377-2217(97)00136-7. URL https://www.sciencedirect.com/science/article/pii/S0377221797001367.

[19] Yan Gu, Feng Cheng, Lijie Yang, Junhui Xu, Xiaomin Chen, and Long Cheng. Cost-aware cloud workflow scheduling using drl and simulated annealing. *Digital Communications and Networks*, 10(6):1590–1599, 2024. ISSN 2352-8648. doi: https://doi.org/10.1016/j.dcan.2023.12.009. URL https://www.sciencedirect.com/science/article/pii/S2352864823001840.

[20] Yalçın Akçay, Haijun Li, and Susan H. Xu. Greedy algorithm for the general multidimensional knapsack problem. *Annals of Operations Research*, 150(1):17–29, Mar 2007. ISSN 1572-9338. doi: 10.1007/s10479-006-0150-4. URL https://doi.org/10.1007/s10479-006-0150-4.

[21] Fabian Lehmann, Jonathan Bader, Friedrich Tschirpke, Lauritz Thamsen, and Ulf Leser. How workflow engines should talk to resource managers: A proposal for a common workflow scheduling interface, May 2023. URL http://dx.doi.org/10.1109/CCGrid57682.2023.00025.

[22] Anas Nadeem and Muhammad Zubair Malik. A case for microservices orchestration using workflow engines. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE '22, page 6–10. ACM, May 2022. doi: 10.1145/3510455.3512777. URL http://dx.doi.org/10.1145/3510455.3512777.

[23] Worker performance — Temporal Platform Documentation — docs.temporal.io. https://docs.temporal.io/develop/worker-performance. [Accessed 05-03-2025].

[24] M.R Garey, R.L Graham, D.S Johnson, and Andrew Chi-Chih Yao. Resource constrained scheduling as generalized bin packing. *Journal of Combinatorial Theory, Series A*, 21(3):257–298, 1976. ISSN 0097-3165. doi: https://doi.org/10.1016/0097-3165(76)90001-7. URL https://www.sciencedirect.com/science/article/pii/0097316576900017.

[25] JA Diniz, B Skarnes, and P Robson. Study 4- rna-seq of male kolf2.2j hipsc-derived cortical brain organoids and extra-embryonic lineages homozygous null for six different transcription factors, 2025. URL https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE287843.