Scaling Out Bioinformatics in the Datacenter

Thèse N° 9451

Présentée le 22 juillet 2019

à la Faculté informatique et communications
Laboratoire des Systèmes de Centres de Calcul
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Sam David WHITLOCK

Acceptée sur proposition du jury

Prof. A. Argyraki, présidente du jury
Prof. E. Bugnion, directeur de thèse
Prof. I. Xenarios, rapporteur
Prof. P. Felber, rapporteur
Prof. J. Larus, rapporteur

2019

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Acknowledgements

I am tremendously grateful to have been advised by Edouard Bugnion. Ed's immense technical prowess not only aided me in finding solutions to seemingly insurmountable technical hurdles, but his broad range of expertise also taught me how to approach problems from multiple angles. By asking fundamental questions, he encouraged me to critically consider the architecture of the systems I developed during my doctoral studies as well as how I could most effectively communicate my research results. He taught me how to systematically approach technical challenges in computer science by iterating from design to development to evaluation (and possibly back to design). I am fortunate to have learned these research and career skills from Ed through his patient guidance.

I thank the members of my thesis committee (Katerina Argyraki, Pascal Felber, James Larus, and Ioannis Xenarios) for committing the time to serve in my committee and providing insightful feedback.

I am fortunate to have collaborated with many students and professors on various research projects: Katerina Argyraki, Stuart Byma, Jonas Fietz, and James Larus. These collaborations were not only necessary to the success of these difficult research endeavors, but they were also sources of camaraderie, expertise, and perspective for which I am grateful.

I thank my fellow graduate students from the DCSL and VLSC labs. Their diverse backgrounds, research interests, and hobbies made for intriguing conversation and sense of solidarity as we each pursued our own research projects.

Finally, I thank my parents for unwaveringly supporting me throughout my life. The skills and life lessons that they imparted to me enabled me to succeed in this demanding doctoral venture. Despite there being a continent and an ocean between us, my parents have remained a steadfast source of encouragement. I consider myself truly fortunate to be their son.

Lausanne, 8 May 2019 S. W.

Abstract

Whole Genome Sequencing is a process in the field of bioinformatics that transforms biological samples of DNA into an electronic dataset of genetic bases. The process consists of two sequential components. First, the laboratory process of sequencing transforms the biological DNA samples into a digital format by transcribing the sequence of bases that make up short snippets of DNA. Second, a genomic workflow uses software to transform this data into a representation that is useful for genomic analysis. Recent advancements in High-Throughput Sequencing technology enable the laboratory phase to produce data faster and at a lower cost than prior techniques were capable of. The applications and file formats used in workflows have not undergone commensurate technological advancement in order to accommodate this deluge of genomic data.

This thesis introduces a redesign of genomic workflows, their component applications, and the underlying file formats in order to scale out workflows across a data center. The design builds upon on two design components. First, a unified file format supplants the myriad existing formats in order to accommodate scale-out multi-machine I/O. The file format imposes minimal feature requirements upon the storage system, thereby enabling its use in high-performance systems for processing and cost-effective cold storage systems for long-term storage. Second, a new cloud computing framework provides an API for composing workflows in an abstract logical description and delegating the execution of the logic to a common runtime. The framework's runtime executes the logical workflow description on scale-out hardware resources while abstracting the execution details.

We combine the file format and framework to build a new set of workflows that scale out across data center resources. These scale-out workflows incorporate existing workflow applications (compartmentalized into libraries that the framework invokes) and new applications that leverage the features provided by the scale-out architecture. All workflows delegate work distribution and task concurrency to the framework's runtime and utilize a common set of subcomponents for auxiliary code (*e.g.*, I/O with various storage systems, processing different file formats). These workflows are able to scale out across a data center to the point of saturating the throughput of one or more hardware resources.

Keywords: bioinformatics, genomics, cloud computing, big data, data center, scale-out, dataflow, software as a service, software architecture, resource management

Résumé

Whole Genome Sequencing est une méthode dans le domaine de la bioinformatique qui transforme des échantillons biologiques d'ADN en un ensemble de données électroniques de bases génétiques. Ce processus consiste en deux composants séquentiels. Premièrement, le processus de séquençage en laboratoire transforme les échantillons d'ADN biologique en un format numérique à travers la transcription des séquences de bases constituant des fragments d'ADN courts. Deuxièmement, un workflow génomique utilise des logiciels pour transformer ces données en une représentation utile pour l'analyse génomique. Les progrès récents dans la technologie de séquençage à haut débit permettent à la phase en laboratoire de produire des données plus rapidement et à un coût inférieur à celui que permettaient les techniques antérieures. Les applications et les formats de fichiers utilisés dans les workflows n'ont pas connu d'avancée technologique correspondante pour prendre en charge ce déluge de données génomiques.

Cette thèse introduit une refonte des workflows génomiques, de leurs composant principaux, et des formats de fichiers sous-jacents afin de faire évoluer les workflows dans un centre de données. La conception s'appuie sur deux éléments de conception. Premièrement, un format de fichier unifié supplante la myriade de formats existants afin de prendre en charge les I/O multi-machine à déploiement horizontal. Le format de fichier impose un minimum d'exigences en terms de fonctionnalité au système de stockage, permettant ainsi son utilisation dans des systèmes de traitement à haute performance et des systèmes de stockage froids économiques pour un stockage à long terme. Deuxièmement, une nouvelle infrastructure de cloud computing fournit une API permettant de composer des workflows dans une description logique abstraite et de déléguer l'exécution de la logique à un environnement d'exécution commun. Le moteur d'exécution exécute la description logique du workflow sur les ressources matérielles déployées horizontalement tout en résumant les détails de l'exécution.

Nous combinons le format de fichier et le moteur d'exécution pour créer un nouvel ensemble de workflows capables d'être déployés horizontalement sur les ressources d'un centre de données. Ces workflows à déploiement horizontal intègrent des applications issu de workflows existantes (compartimentées dans des bibliothèques appelées par le moteur d'exécution) et de nouvelles applications exploitant les fonctionnalités fournies par l'architecture à déploiement horizontal. Tous les workflows délèguent la répartition du travail et la simultanéité des tâches au moteur d'exécution et utilisent un ensemble commun de sous-composants pour du code auxiliaire (par exemple, I/O avec divers systèmes de stockage, traitement de différents formats de fichier). Ces workflows peuvent être déployées horizontalement dans un centre de données jusqu'au point de saturation d'une ou de plusieurs ressources matérielles.

Mots clefs : bioinformatique, génomique, cloud computing, big data, centre de données, déploiement horizontal, dataflow, logiciel en tant que service, architecture logicielle, gestion de ressources

Contents

List of Figures

List of Tables

1 Introduction

Understanding the genetic code that dictates all biological processes is a crucial factor for developing next-generation medical therapies. This genetic code exists in the nucleus of each of our cells and is spelled out using an alphabet of molecules (*i.e.*, nucleic acids) chained together to form DNA. Although this alphabet is small (consisting of adenine, thymine, cytosine, and guanine in DNA, which are commonly referred to by their first letters A, T, C, and G), each of our cells interprets this alphabet by transcribing long sequences of bases into a chain of amino acids. These amino acids form the protein structures that dictate a specific cellular function and, in turn, the macroscopic functionality of an organism.

Bioinformatics is the study of genetic information in digital formats using algorithms and computer software. Molecular information is first transformed using a sequencer, which is a laboratory machine which reads short snippets of a genetic sequence to produce a set of reads in a digital format. This digital information (*i.e.*, a collection of associated reads) is then processed by a series of algorithmic transformations known as a workflow. The algorithms used in each step of the workflow are implemented as software applications, which draw upon techniques and knowledge from myriad fields, including statistics, computer science, and biology. The end result of a workflow is a dataset that researchers or medical professionals can query, *e.g.*, to determine whether a single individual contains certain a certain genetic mutation or the number of genetic variations between a large group of individuals.

Bioinformatics produces data and insights that enable personalized medicine. In contrast to the use of medical interventions based on broad population-level studies, personalized medicine is the customization of medical treatment to a particular individual. Medical professionals can use a fine-grained approach to analyze specific treatments for each individual based on the specific genetic information produced by a bioinformatics workflow. For example, the dosage of a medication may be adjusted based on genetic markers of susceptibility to side effects [110, 121].

Recent advances in sequencing technology have enabled a rapid acceleration in the production of genetic data. High-throughput sequencing (HTS) technologies produce data with a

higher throughput and lower cost than the original sequencing processes. The sequencing of the first human genome cost approximately $3 billion and required over a decade of effort from hundreds of researchers [66, 108]. New sequencing technology has reduced the cost to $1000 and is poised to reduce an order of magnitude further to $100 in a few years. These technological advancements have reduced the latency as well, from years to days, as the rate at which sequencing technology advances in speed outpaces Moore's Law [85]; a modern sequencing machine can output high-quality human genome datasets at the rate exceeding 18,000 per year, amounting to a total of over a petabyte of data per year per machine [65]. Globally, the amount of such genomic data is on track to surpass astronomical data in terms of storage volume in the near future [113].

The applications and file formats used in the bioinformatics workflows have not adequately evolved to accommodate the accelerating rate of sequencing technology [95, 108]. Bioinformatics has disproportionately focused on advances in sequencing technology while relying on legacy software applications to perform the transformations in the workflow. These legacy applications and the accompanying file formats were created when the size of each dataset was small and feasible to process on a single machine. Using these applications to process modern, high-quality datasets imposes a high latency of hours to days. It is widely accepted that one must exercise patience due to this status quo of high latency [28]. Little attention has been paid towards the creation of scalable, data center-scale applications to implement bioinformatics workflows.

This dissertation proposes a new scale-out design for bioinformatics workflows comprising three key aspects: First, a data format, the *Aggregate Genomic Data* (AGD) format, unifies the existing file formats into a single common representation that is more amenable to scale-out data processing on a wide variety of storage systems. Second, a scale-out application framework, *Pipelined TensorFlow* (PTF), provides an efficient base for composing workflows of existing bioinformatics tools into indefinitely-executing services that concurrently processing requests. Third, *Persona* combines AGD, PTF, and a set of bioinformatics workflow transformations (including existing applications encapsulated as libraries as well as custom applications) to create scale-out bioinformatics workflows. We argue that this approach is necessary due to the accelerating production of genomic data as the underlying technologies becomes cheaper and more widely available. Moreover, the file formats and the applications commonly used by bioinformatics professionals do not scale to meet this deluge of data.

1.1 Problem Definition

The sequencing process begins with the preparation of a sequencing library in a laboratory. A library is a collection of millions to billions of small fragments of DNA that serve as input to a high-throughput sequencer. The library is prepared by slicing a DNA sample into fragments within a given size range (typically hundreds to thousands of bases). Each fragment is amplified

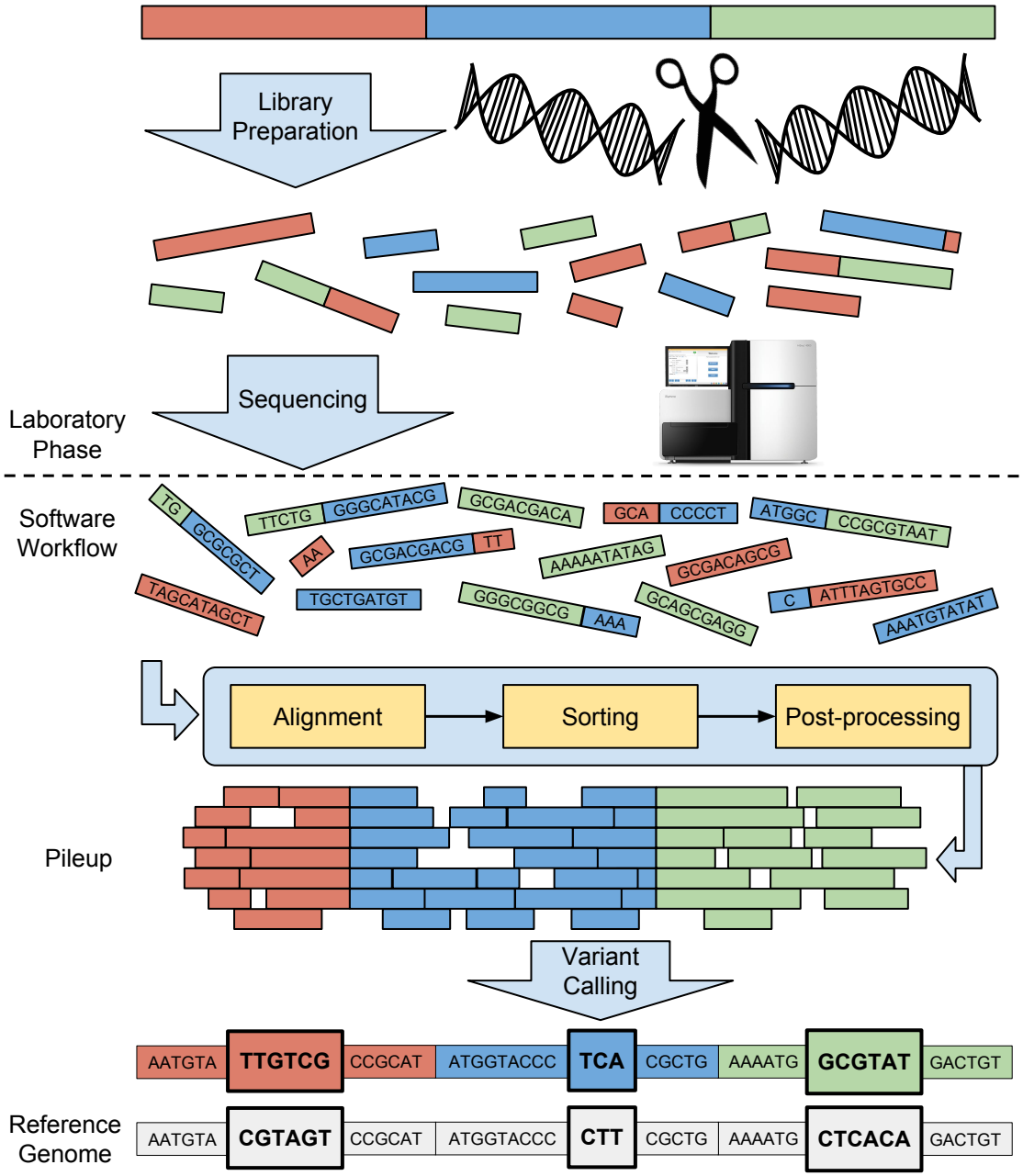


Figure 1.1 – An overview of whole-genome sequencing via the shotgun sequencing method

(*e.g.*, by polymerase chain reaction [107]) in order to reduce the error rate when reading the bases via redundancy.

The sequencer processes the library into a collection of reads. Using various HTS methods, the sequencer transcribes each fragment in the library into a read; each read consists of the sequence of bases detected by the sequencer, a numeric quality score associated with each base (the sequencer's confidence in its determination of the base, *i.e.*, A, T, C, or G), and some

associated metadata specific to each sequencer (to identify the physical subcomponent of the machine that produced the read). The output of the sequencer is a digital format, typically a single file consisting of all reads in a given library. This is the key function of the sequencer: transforming the biological information in the library into a digital dataset for downstream algorithmic transformations performed in software.

A bioinformatics workflow transforms the sequencer output into a useful result through the sequential invocation of software applications. Each application in a bioinformatics workflow encapsulates a specific algorithm that transforms an input dataset into an output dataset for downstream use. Each application typically adds fields to each read in the dataset, rearranges the order of the reads within the dataset, or updates existing fields in each record. Workflows may vary in the transformations they perform and the applications used to apply a transformation, but they typically include a few key transformations. This dissertation focuses on the following transformations:

- Alignment: undo the fragmentation of the genetic material done for the library preparation by aligning each read relative to a reference genome. Alignment uses the fact that all humans share a large portion of common DNA (approximately 99.9%). It capitalizes on this overlap by using a reference genome (a composite genome of several individuals to serve as a "standard" genome) as a guide to find the exact location of origin for each read before library preparation. The result of alignment is one or more alignment results for each read, each of which contains a location in the reference genome as well as an edit distance from the bases in the reference.
- Sorting: sorts the dataset of reads based on the ascending order of their alignment information. This I/O-intensive step is necessary to reduce the processing time of downstream workflow transformations.
- Post-processing: perform operations on the sorted dataset to adjust for errors in upstream transformations. For example, one such process, known as local realignment, uses information in overlapping (*i.e.*, neighboring) reads in the sorted dataset to find a better alignment against the reference.

The final step in typical bioinformatics workflows is variant call analysis. Variant calling uses a reference genome to determine where genetic differences exist between the reference genome and the input dataset. This is typically done by producing a pileup for each base (*i.e.*, a cross-section of the reads that overlap a given location of a base in the reference genome) to determine what the true base is in the input dataset. The output of this analysis operation is a set of polymorphisms, *i.e.*, where the input dataset differs from the reference genome. This output is much smaller in size than the input, as the number of differences between each human is small compared to the overall number of bases.

Figure 1.1 provides a visual representation of the full bioinformatics workflow, including the laboratory and workflow phases. In the laboratory phase, we show that the library preparation

cuts the input genetic material into myriad small fragments with unknown location. The sequencing process then transcribes the sequence of bases for each of these fragments. The bioinformatics workflow aligns, sorts, and then post-processes the reads into a pileup representation, which variant calling uses to identify locations where the original input genetic sequence differs from the reference genome.

1.1.1 Limitations of Contemporary Workflow Software

The focus of bioinformatics applications has been the codification of expert domain knowledge. The most important metric for each application is the quality of the results and the correct implementation of the algorithms for performing a given transformation. This is especially important for any transformation that involves approximate computation (*i.e.*, where there is no single correct output, only comparatively better or worse than other outputs) or where small parameter adjustments can have a significant effect on the algorithm's performance. The alignment transformation is a prime example encompassing both of these factors: slight perturbations in its parameters affect the quality of the alignment, and random choices must be made between alignment locations that receive an equal "score" by the application. The ability to reliably reproduce equivalent results from identical inputs is important for reproducible research in bioinformatics.

Bioinformatics applications do not typically scale up to utilize the hardware resources of a machine. Each application contains a set of core functionality, such as aligning a read, and some auxiliary code that calls into this core functionality. This auxiliary code performs I/O, handles file formatting issues, and handles multi-threading. Especially critical is the overlap between concurrent I/O and processing. These mechanisms are difficult to build in any application that seeks to scale across all hardware resources of a single machine. Each of these pieces of functionality in the auxiliary code can have a significant performance overhead if there is an error in the methodology (*e.g.*, queues between different functional elements that have high overhead). Most bioinformatics applications have at least one such performance issue, if not many, related to this auxiliary code.

Bioinformatics applications typically perform a single transformation step in the workflow. Each application is invoked on a single machine with the location of a single input dataset (*e.g.*, a path on the file system) given as a parameter. It then performs the transformation on all records in the dataset and writes the resulting dataset to a new location. This is due to both the ad-hoc evolution of the workflow (*e.g.*, new algorithms being added for further processing as needed) and the fact that separate organizations developed these algorithms and the associated software. This limits the performance of workflows in the following ways:

- No pipelining: each application must be invoked in workflow succession in order to completely process a dataset, inhibiting effective overlap of algorithmically independent calculations. Even if successive applications in a workflow are not prohibited from simultaneously processing portions of a given dataset, they lack the mechanisms to

coordinate; only external techniques, such as using a POSIX pipe to redirect I/O, can be widely used, if at all.

- Limited scale-out processing: applications are designed to process an input dataset in its entirety on a single machine. It is difficult to distribute processing across multiple machines, and the inability to pipeline means that little to no performance advantage would be garnered by doing so. In the case that it is necessary, extra workflow steps would be added to transfer the dataset between machines or an external mechanism (*e.g.*, a distributed file system) would be used.
- Excessive I/O: each workflow application must write its resulting dataset back to storage to serve as input for the next application. This creates a large I/O overhead for intermediate datasets, *i.e.*, datasets that do not serve as an initial input to, or final output of, the workflow.

Despite the data center-scale datasets these applications process, these inherent limitations in these applications and their workflows make it difficult to scale the software portion accordingly.

1.1.2 Bioinformatics File Formats

A number of different file formats are used in the different phases of bioinformatics workflows. FASTQ [33] files are used as the initial input (produced by the sequencing step); each FASTQ file contains a textual representation of the fields in each read (bases, qualities, and metadata). The Sequence Alignment Map (SAM) [80] contains the fields of FASTQ as well as alignment information; these are typically the output of the alignment and sort phases. Variant call format (VCF) [35] files serve as the final output after the variant call analysis phase. These three plaintext formats have binary counterparts, either a generic text compression (*e.g.*, FASTQ, which is typically stored using GZIP) or a custom block-compressed binary version (*e.g.*, Binary Alignment/Mapping (BAM) [80] for SAM files or binary VCF (BCF) for VCF files).

The file formats used to store genetic data are not capable of meeting the need for high-throughput workflow processing. These file formats grew out of small-scale applications for locally processing data on a single computer. This led to a number of design choices that favor easier understanding of what data is contained in the files (due to plain text formats) over efficient I/O patterns and scalable processing across a cluster of machines.

Monolithic structure: All data for a given dataset is typically contained in a single file, *e.g.*, a single FASTQ file per input or a single BAM file for alignment results. Existing formats do not contain the necessary information to describe chunks across multiple files. This monolithic structure requires either special support within the storage system or a serial work distribution step in a distributed application in order to scale out processing across independent application components. Furthermore, parallel I/O and compression may not

be possible, or may need to be coordinated via a serialized component of an application (*i.e.*, a component that cannot scale). Encapsulating an entire dataset in a single file is a basic way to provide data provenance for operations (*i.e.*, the operations of one file transform into one new file), but it limits the potential to scale out operations. For example, this would require a single `map` task in the MapReduce paradigm [36] to perform all of the I/O on a file and send out smaller chunks of work to subsequent tasks.

Record-oriented format: Within each monolithic file, records are oriented such that all fields for each record are adjacent to each other within the file's structure. This is born out of the need for humans to be able to quickly examine each record: it is much easier to examine a record if it is contiguous in the file instead of related via context across multiple regions of one file or multiple files. This means that all data for a given record is contained within one contiguous sequence of bytes (perhaps after a decompression step), requiring a single read and scan in order to consume the record by an application. This makes it easy to write simple applications and to read (especially for the plain text files), but this requires extra I/O: all fields of a record must be read by the application even if the application does not require all of them.

Non-unified formats: Data at each stage of a bioinformatics workflow is stored in different file formats even though information stored in them is often semantically equivalent. This is due to a lack of consensus about formats as technology evolved between different companies and academic organizations. It is difficult to change formats once they become widely adopted due to the ecosystems of applications and services evolve around them. Even new efforts to unify the formats tout their ability to be converted to and from existing formats [68].

1.2 Thesis Statement

Creating a cloud computing framework for bioinformatics workflows requires rethinking some of the assumptions about and reliance on existing bioinformatics applications and file formats. Naively wrapping existing applications in contemporary cloud computing frameworks leaves a lot of performance opportunities unexploited, as inefficiencies still exist in not only the bioinformatics applications and formats, but also in the current cloud computing frameworks. This thesis claims that an efficient, modular framework for expressing bioinformatics workflows and subsequently executing these workflows as a service, running indefinitely while concurrently processing an unending stream of requests, is a prerequisite for commensurately scaling the applications with the increasing throughput of HTS sequencing machines. This increased throughput enables more and higher-quality genomic datasets to be used for scientific research and medical applications.

The thesis statement is the following:

This thesis introduces an approach to scaling bioinformatics workflows across a cluster of heterogeneous hardware resources in a data center environment. The use of a cloud computing framework to unify separate workflow applications into a single domain improves resource utilization and enables the framework's runtime to pipeline requests, increasing throughput with negligible latency penalties. Delegating common functionality (*e.g.*, I/O, file formatting, compression) to a common framework removes inefficient auxiliary code from existing applications' core functionality. The framework's runtime coordinates inter- and intra-machine concurrency for all components of the workflow's logic. A new file format designed for concurrent scale-out operations on bioinformatics datasets supports the framework by reducing I/O overhead.

1.3 Thesis Contributions

In this thesis, we demonstrate how a few careful design decisions lead to an extensible and unified cloud computing framework (*i.e.*, one that applies to many different transformations) for bioinformatics workflows. By studying existing bioinformatics applications, we determine where their core functionality is and where inefficiencies are. We combine the core functionalities of these applications together into a common framework that executes workflow logic as a service that runs indefinitely to process requests.

Supporting the framework's scale-out operations is a new file format that enables parallel I/O operations on bioinformatics datasets with minimal features required from the storage system. By understanding the semantics of the myriad file formats and the ways that each workflow phase uses the information contained in the files, we develop a file format that serves all of their needs while increasing the efficiency of processing a dataset by optimizing for the access patterns.

The framework and file format together enable request concurrency and data pipelining. Request concurrency shares the same hardware resources across multiple concurrent requests. Data pipelining occurs when successive steps in the workflow logic are performed concurrently on different data items, similar to the pipelining of instructions in a CPU. The framework explicitly tracks the subcomponents of each request as successive elements of the workflow application process them. The file format enables coordination-free operations; different tasks in the application may operate on different segments of a dataset without coordinating access via an external mechanism or relying on special features in a storage system. These two elements reduce the amount of idle time by increasing the number of data items of each request that are available for processing at any point in time.

We make the following research contributions in this dissertation:

- The Aggregate Genomic Data (AGD) format, a new format for storing genomic information. This format is interoperable with existing widely-used formats in bioinformatics, but has distinct advantages for use in scale-out processing. In brief, AGD's key attributes are as follows: a) it minimizes the amount of necessary file I/O for a given application (*i.e.*, semantically unnecessary data may remain untouched during a given phase of an application, while being read by a subsequent phase), b) it enables a dataset to be split into multiple files for easier workload partitioning, c) it optimizes each data field separately, storing the data in a dense format that is variable-width and easy to scan when reading and highly compressible.

- Pipelined TensorFlow (PTF), an application framework built on top of TensorFlow (a high-performance cloud computing framework) [1] that enables users to construct bioinformatics workflows that run indefinitely and concurrently process client requests. Each application is constructed as a linear pipeline of transformations referred to as stages. Each stage is separated from up- and downstream stages by gates, which coordinates asynchronous request processing between successive stages. This pipeline may run locally on a workstation or scale out to a large cluster of heterogeneous hardware merely by scaling each stage independently; the application logic remains the same, as only a few scaling parameters must be adjusted in the configuration.

- Persona, a framework for constructing bioinformatics workflows built on PTF. A central component of Persona is its ability to compose the core components of standard, widely-used bioinformatics applications with efficient I/O operations to read and write AGD datasets. By leveraging the pipeline abstraction from PTF, Persona constructs these individual components together into a scale-out workflow, capable of executing on heterogeneous hardware resources; the same workflow application can run on a laptop or a large cluster with only minimal alterations to its configuration.

We evaluate AGD, PTF, and Persona in their effectiveness as replacements for traditional bioinformatics workflows. We demonstrate that AGD's attributes enable it be used as a drop-in replacement for traditional bioinformatics formats (*e.g.*, FASTQ, SAM) at various stages in a workflow; we also demonstrate that it can serve as a long-term storage option for genomic data, as it compresses as efficiently as contemporary formats (*e.g.*, BAM). Persona outputs identical data as traditional bioinformatics workflows, but due to PTF's scale-out design and use of AGD, it can scale a computation across a cluster of machines, effectively decreasing latency and computational overhead simultaneously.

1.4 Thesis Roadmap

This thesis is organized as follows:

- Chapter 2 discusses the background knowledge and prior work of both bioinformatics workflows and cloud computing frameworks. It includes an overview of bioinformatics and a popular workflow known as whole genome sequencing. It provides a background of the application semantics of each stage of a typical workflow and an overview of the file formats that are used as input and output to each stage. This chapter also discusses the various types of cloud computing frameworks and what aspects make them the programming paradigm of choice for scale-out applications. We then discuss TensorFlow, a recent addition to the collection of cloud computing frameworks targeted towards machine learning applications, and what it provides that is useful beyond the domain of machine learning.

- Chapter 3 provides an overview of the AGD format. It sets out the goals for an efficient, unified format for bioinformatics data in §3.1. We then demonstrate how AGD achieves those goals based on its design principles in §3.2.

- Chapter 4 describes the architecture of PTF. We extend the standard TensorFlow framework with a few simple additions of code that enable it to run application pipelines consisting of a series of transformations. PTF runs separate phases of a pipeline in parallel, enabling applications to scale both across the hardware resources of a single machine as well as of a cluster of machines. It enables the pipelining of concurrent requests on the same application, a feature not supported by standard TensorFlow, in order to increase hardware utilization.

- Chapter 5 describes the design of Persona. In this chapter, we describe how Persona decomposes existing bioinformatics workflows and applications into the PTF (and, by extension, TensorFlow) architecture. Once encapsulated into PTF pipelines, Persona composes applications into bioinformatics workflows that are able to read and write a variety of datasets (primarily AGD) as well as interact with various I/O systems using a library of common components. We describe the scale-out architecture for a two-phase workflow of genome alignment and subsequent sorting, and how we structured Persona to use the features of PTF.

- In chapter 6, we evaluate Persona on two aspects. We begin by evaluating a single Persona application on one machine to show how PTF enables it to scale across the resources a single machine (*e.g.*, CPU, memory, or NIC bandwidth). Specifically, we use the SNAP aligner [132] to align a dataset, the first operation of a typical workflow when transforming a dataset. We encapsulate the core components of SNAP as a library in Persona and benchmark it against the standard SNAP application to demonstrate how Persona benefits even single-machine applications through its efficient runtime; this evaluation shows that Persona's use of PTF enables it to efficiently scale across the resources of a single machine, specifically benefiting from the overlap of computation and I/O of separate stages of the pipeline. We demonstrate additional speedups through a similar mechanism for several other applications, such as dataset sorting and duplicate marking. After establishing Persona's benefits for single-machine bioinformatics

applications, we scale Persona pipelines across a cluster of multiple machines, all sharing a Ceph [127] storage system to transmit data, to demonstrate Persona's operating efficiency across a cluster of machines as well as AGD's support for scale-out operations. We first evaluate a scale-out version of the SNAP-based alignment application discussed in chapter 5; we scale it across the cluster to demonstrate its linear scale-out capabilities. We then evaluate the scale-out performance of the two-phase pipeline of alignment and sorting to demonstrate its scale-out capabilities. This chapter includes a discussion how pipelines can be tuned to each cluster, based on the configuration and available hardware resources.

- Chapter 7 discusses related work for the work presented in this dissertation.
- Chapter 8 concludes this thesis with a discussion of lessons learned while developing PTF and Persona.

1.4.1 Bibliographic Notes

Portions of this thesis are based on work I have previously published with a colleague, Stuart Byma, in the work ***Persona**: A High-Performance Bioinformatics Framework* [26] published at the USENIX Annual Technical Conference (2017). Stuart contributed the concept of using TensorFlow for this project, and the architectural design and implementation of porting existing bioinformatics applications into Persona. My contribution is the AGD file format, the scale-out architecture of Persona (PTF and the prior iteration in [26]), and the mechanisms to efficiently use custom data structures within Persona (resource pooling).

Not discussed in this thesis is work that I have also undertaken during my doctoral studies on network function virtualization [49].

2 Background

This chapter provides background on whole genome sequencing (WGS), a popular technique for assembling a linear sequence of nucleotides (bases) from a genetic sample, and on cloud computing frameworks. We focus on a WGS method of genome resequencing that involves two phases: a "wet" phase where the biological genetic material is prepared and sequenced into a file and a "dry" phase that processes this file using a sequencing of algorithmic transformations, *i.e.*, bioinformatics workflows. The wet phase has increased in speed, outpacing Moore's law and consequently the throughput traditional applications and workflows used for the dry phase. Cloud computing frameworks are used to speed up applications by separating the application logic, specified abstractly in a description, from the execution of that logic, enabling frameworks' runtimes to scale the application across heterogeneous hardware on a cluster of machines. These frameworks provide a solution for scaling the I/O and computational demands of bioinformatics workflows.

We discuss an overview of the technology involved in the genome resequencing for both the biological and digital phases in §2.1 and §2.2, respectively. We then discuss cloud computing frameworks in §2.3 and delve into detail on a particular framework, TensorFlow, in §2.4 due to its use as a foundation for the research contribution discussed in subsequent chapters of this dissertation.

2.1 Constructing a Genome

The advancements in the production of genomic data is due to both innovation in the production of data from biological samples and the computational techniques to subsequently transform this data into a useful format for medical practitioners and researchers. Various forms of high-throughput sequencing (HTS) technology are capable of producing input data at a sufficiently high throughput and low cost such that whole-genome sequencing is on the cusp of being a ubiquitous medical service, with applications for treatment of both an individual and for populations [112]. This section discusses the HTS technology known as shotgun sequencing, wherein the genetic sample is broken up into small sequences of DNA

and reassembled by a genome analysis software workflow. Understanding how this data is produced provides background on the computations involved in the computation and reassembly.

The process begins with the preparation of a library, *i.e.*, a prepared genetic input for the sequencing machine. The library is prepared by cutting the genetic material at known sites (breaking up long contiguous strands of DNA), filtering the selected based on a size range, and then amplified using the polymerase chain reaction (PCR), which uses repeated thermal cycling to replicate the genetic material. This preparation is not an error-free process; transformations within the algorithmic component of this process take this error rate into account and are necessary to prepare the data for further analysis.

The library is loaded into sequencer, which analyzes the genetic material to output a file, which will serve as the first electronic input for the genome analysis software workflow. This stage is where recent advances have dramatically increased throughput in recent years. Although there are several technologies that transform a library to a file, we focus on the prevalent technology from Illumina. These sequencers operate on libraries composed of short-length sequences (*e.g.*, the Illumina platinum genomes [44], a high-quality reference dataset produced on Illumina sequencers, has a read length of 101), which the library preparation must filter for before the sequencing process. Other contemporary technologies produce long reads, such as that of Nanopore [32] and Pacific Biosciences [46].

Illumina sequencing first generates clusters of each segment in a given region of a flow cell (a hardware component to which library samples bind) using the clonal amplification process. Each segment of DNA prepared for an Illumina sequencer has an adapter attached to each end, a process referred to as "tagmentation" or adapter ligation. These segments are then inserted into the flow cell, a part of the machine with complementary adapters attached to a fixed surface. Clonal amplification is applied to create clusters of identical reads within a region of the flow cell. This process involves alternatively attaching each adapter to the flow cell, denaturing, and regenerating the complementary strands. This process is also referred to as "bridge amplification", due to the way the strand adapters are repeatedly attached to the flow cell.

The Illumina sequencing process records the bases that are present in each segment using a technique called sequencing by synthesis. With the library prepared in the flow cell, the sequencer removes the reverse DNA strand of each segment, leaving only the forward strand attached to the adapter in the flow cell. This reverse strand is reconstructed, one base at a time, with free nucleotides in the flow cell that are each tagged with a given molecule that will fluoresce a unique color when exposed to light. The sequencing process runs in cycles; in each cycle, one nucleotide is attached, its fluorescence captured, and fluorescence molecule removed. The number of cycles performed for each sequence corresponds to the length of the reads produced by the sequencer.

The result of the sequencing process is a data file containing a set of reads, each of which represents the sequence of nucleotides read by the sequencing machine in a given location of the flow cell. The Illumina machines are often configured to produce *paired* reads, which are sequences from both ends of each segment of bases (both the forward and reverse complementary DNA strands of the original sample). During the sample preparation process, one must select the appropriate size of segment length based on the configured read length (how many cycles or bases will be read from each end) as well as the gap between each end (referred to as the *insert*). If the sequencer is configured to read 100 bases from each end and segments are created with a length of 300, the gap will be 100 bases on average. These pairs are output as separate records in the sequencer output file, but contain "pointers" to each other to mark that they are related from the same original segment.

The sequencing process contains many inherent sources of noise that the software applications in the subsequent workflow must correct. There can be errors in library preparation which can be magnified by the PCR process. The sequencer itself has a probability of incorrectly reading any base; the limitation on the size of reads capable by Illumina's technology is due, in large part, to the correlation of read size with error probability. The sequencer outputs a read quality score for each base, representing its confidence that it read the correct base. These quality scores are used to help control for this error rate, but these scores are taken account when processing all reads in the genomic workflow's software.

2.2 Genomic Workflow Software and Algorithms

The software applications downstream of the sequencing process transform the data into a form that is usable by researchers and medical professionals. There are many types of analysis workflows, but we will focus on that performs variant calling. The goal of this workflow is to identify locations where two individuals have genetic differences, *i.e.*, where a variation is "called". In most applications, the individuals involved are the input genetic material (*e.g.*, a patient's genetic material) and a reference genome, a composite of several individuals that is used as a genomic standard. Variants are called where known locations contain a different sequence of bases, with the difference being one of the following: a substitution of one base for another, an insertion of bases, or a deletion of bases.

The software workflow must correct for errors introduced by processes in the upstream sequencing component. Although careful library preparation and technological innovation minimizes the probability of errors in sequencing component of WGS (*e.g.*, flow cell amplification), the rate at which errors are produced by the upstream components exceeds the probability of a true variation. The Illumina machines have an error rate per read of $\geq 0.1\%$ [56]. This may seem low, but this is not adequately low to rely upon alone, as two individuals of the same species will share a large portion of their genome. This error rate must be mitigated by the workflow so that the resulting dataset (*i.e.*, a set of genomic variants from the reference)

will contain a sufficient percentage of true variants, as opposed to variations due to errors introduced by the sequencing process.

This dissertation focuses on the applications that perform the first two stages of transformation: read alignment and the subsequent sorting of the dataset of reads. We discuss these stages and several others in order to provide adequate background about how the end result (a contiguous sequence of bases corresponding to the DNA in the sequencer library) is produced.

2.2.1 Alignment

Alignment is the process by which each read is mapped to a location in the reference genome. The reference genome contains a series of "contigs", *i.e.*, contiguous regions of genetic material (approximately a single strand of DNA). A mapping location is a two-dimensional coordinate against the reference genome: the contig number and the position within the contig where the mapping begins.

The alignment process is split into two phases by most applications: global and local alignment. Global alignment is a computationally inexpensive to identify regions within the reference genome to which a read could map. Alignment applications leverage some type of index in this stage; popular aligners include the following:

- The Burrows-Wheeler Alignment (BWA) [79] application uses an index based on an eponymous algorithm [25], which is used as a component of text compression. SOAP2 [81] and variants of this approach also use a Burrows-Wheeler-based index [3, 31, 78].
- Bowtie [76] and its successor Bowtie2 [74], both of which use an FM-index [48] (an index based on the Burrows-Wheeler transform).
- Single Nucleotide Alignment Program (SNAP) [132] and the basic local alignment search tool (BLAST) [5], which use a large hash table of exact locations as an index. The hash table's key is a k-mer (*i.e.*, a string snippet of the reference genome of length k) and the value is one or more locations that contain that exact match.

Local alignment uses an algorithm to calculate an edit distance result between the read and the potential location in the reference. This edit distance result is contains the number of substitutions, insertions, and deletions of the read compared to the reference location. Commonly used algorithms for local alignment involve a dynamic programming approach such as Needleman-Wunsch algorithm [98], its successor Smith-Waterman [111], and BLAST [5].

Alignment applications provide an approximate result: different applications and different configuration of the same application will influence the alignment results. Many factors contribute to the result, if any, that an aligner will produce for any given read. Each global alignment strategy will produce a different set of candidate locations. Local alignment algorithms using various parameters to assign a score to each alignment result. For example,

Smith-Waterman algorithms assign a penalty to each type of edit (substitution, insertion, and deletion), and setting these has an influence over which alignment result is chosen on the backtracking phase of the algorithm. Scoring is also used to choose an ideal location amongst all candidate alignments. Mate pairing information, in the case of paired reads, further influences the score, considering the distance between the locations of each read in the pair. Due to the large number of sections with repeating sequences in the genome (*e.g.*, a single nucleotide repeating for many bases), random numbers must be used to break ties in the case of equivalent scores.

Alignment has two properties that are important to note: it is computationally expensive and coordination-free, *i.e.*, "embarrassingly parallel". The local alignment algorithm is asymptotically complex; Smith-Waterman and similar algorithms are quadratic in both time and space. The use of careful parameter tuning can both reduce the number of candidates produce by the global alignment stage and enable the local alignment computation to abort early if the alignment score falls below a threshold. Despite the computational expense of aligning each read, these algorithms do not require coordination between reads; each read can be aligned in parallel without synchronizing with the results of any other alignment.

2.2.2 Sorting and Downstream Analysis

Sorting orders a dataset of aligned reads based on their position in the reference. There are many different tools in the bioinformatics field that perform this operation, such as samtools [80], Sambamba [115], and tools included in the genome analysis toolkit (GATK) [90], such as Picard [24]. Sorting differs from alignment in two key ways. First, it is an exact algorithm: regardless of the chosen sorting algorithm, any application should produce the identical output for a given input. Second, sorting requires a greater degree of coordination between reads, the scheme of which differs based on the chosen sort algorithm, but which fundamentally involves at least one sequential, linear step to produce the final output.

Sorted datasets then undergo several post-processing steps to adjust the dataset.

- Duplicate removal removes reads that have been excessively amplified by the PCR amplification process in the library preparation stage before sequencing. This deduplication prevents certain error-prone regions from biasing the subsequent variant identification process.
- Local realignment adjusts the alignment result of each read based on the adjacent reads, *i.e.*, those that overlap based on location. Each read was aligned in isolation, but examining overlapping reads can provide further insight to reduce the edit distance of a read. For example, a read with many insertions may be realigned with substitutions instead by using information in overlapping reads.
- Base quality score recalibration (BSQR) adjusts the quality scores for each read. The quality assigned by the sequencing machine is often an estimation of reading that single

base correctly (*e.g.*, whether it could assign the fluorescence captured during the cycle to a particular color bin with high probability). However, multiple factors can affect the quality score across multiple reads. BSQR looks at the qualities associated with the read and those of overlapping reads, as well as models about other covariant factors, and sums up the recalibrated factors to assign a new quality score.

All post-processing steps must be done after sorting, as they must be able to query which reads overlap the location of any given read; this would be a linear operation for each read on an unsorted input.

The final step in the genomic analysis workflow is variant calling, which produces a set of variants where the input genome differs from the reference genome. Variations typically fall into one of two types:

- Single-nucleotide variants (SNVs) occur when the reads overlapping a position in the reference genome converge on a different base than the reference genome. One way to determine SNVs is to produce a "pileup" from the reads overlapping a location, selecting from these overlapping base locations the different variations. The variations in this location are computed using the quality scores and one or more variants may be called.

- Insertion and deletion variants (INDELs) occur when the reads encode for additional or fewer bases than the reference genome. These are more complicated to resolve, as they may shift subsequent base locations in the input genome. Statistical modeling is a popular choice to identify INDELs.

Popular variant callers include freebayes [52], tools included in samtools, GATK, Google's Deep Variant [106], Avocado [100] from Big Data Genomics, and many proprietary applications. The result of this step a form useful to researchers and medical professionals: a set of variations from the reference genome. These variations identify possible locations for new genes or predispositions to health issues.

2.2.3 File Formats

Genomic analysis workflows rely on a variety of different file formats to store data. These file formats descend from informal specifications from the field's nascent beginnings and have continued to be used due to adoption momentum, *i.e.*, due to the fact they serve as inputs to and outputs from popular applications. Many of these can trace their origins back to the Human Genome Project [66], specifically the Genome Browser [72] which contains publicly-available genetic information for researchers. The design decisions (*e.g.*, the use of Phred scores [47] to encode quality scoring for each sequenced read) can be traced back to these original formats.

The original formats, many still in use today, represent the data as a textual series of records. The FASTQ [33] and FASTA [104] are used to hold unaligned records, with the latter format being the predominant choice for the distribution of reference genomes. FASTQ contains the quality scores and is the electronic output of the sequencing process, with each record containing the read information, qualities, and sequencer metadata. Aligners transform FASTQ into the Sequence Alignment/Mapping (SAM) format [80], which contains the alignment information in addition to the existing FASTQ fields. SAM files are taken as input by the post-alignment steps, with the variant call format (VCF) [35] defining the format for holding the final workflow results.

Some textual formats have been replaced with binary equivalents for performance and storage. Although FASTQ does not have a widely-used binary equivalent, it is common to find support for GZIP-compressed [42] FASTQ files. The Binary Alignment/Mapping (BAM) format has supplanted the SAM format, with BGZF providing a blocked-compressed, indexable container for BAM records such that only the block containing a record must be decompressed to access it. Ongoing research into reference-based compression (*i.e.*, storing the difference between the mapped location and the read, instead of the entire read) [51] has the potential to dramatically reduce the storage requirements for alignment records. One such format in common use is the CRAM format [21], which includes additional lossy compression techniques such as quality score binning. The Binary VCF format (BCF) is a binary version of the VCF format, employing similar techniques as BAM does to the SAM format (*e.g.*, indexable block compression).

2.2.4 Application Construction

Most genomic analysis applications operate as standalone applications for use on a single machine. The emphasis when constructing most of these applications was the codification of domain logic: the primary objective was to ensure the correct operation of the algorithm(s) employed by the application, with concerns about performance or use in a data center setting being secondary. However, some progress has been made to port existing tools to frameworks more suitable for distributed computation. GATK4, the latest version of GATK, uses Apache Spark [133] as its multithreading framework.

The file formats used in contemporary genomic analysis applications impose a performance bottleneck for scale-out computation. Both text-based and binary formats use monolithic files to store data. Although some formats may have auxiliary files (*e.g.*, an index into a chunked block-compressed file), the formats do not specify a way to correlate multiple files as a single dataset. This imposes synchronization requirements in a workflow; for example, each intermediate file (*e.g.*, the SAM file produced after alignment) must be produced in entirety by the aligner before the software may invoke the next application. Limited pipelining is possible between applications and is usually facilitated through inter-process communication mechanisms such as a Unix pipe redirecting standard output of one application to the input of a subsequent one. The genomic analysis workflow's applications are typically organized by

a workflow management system, which orchestrates invocations of existing applications as an external framework.

2.3 Cloud Computing Frameworks

Before the advent of cloud computing frameworks, distributed applications contained both the application logic and the code necessary to support the execution that logic, such as work distribution across multiple machines. Frameworks such as the Message Passing Interface (MPI) [61] provided a basic interface for exchanging data between multiple processes across a cluster of machines. Each application could use this interface to coordinate between subcomponents of the application distributed over different machines, but each application must implement a large portion of code to coordinate its operations. Common patterns to mitigate stragglers, restart failed computation, and assign work units to individual machines must be reimplemented for every such application.

Cloud computing frameworks address this by providing an API for applications to specify the logic of the application. These frameworks provide a computation model for encapsulating each subcomponent of application logic, a data model for describing the channels along which data are passed between each subcomponent, and a common runtime to deploy and execute these models. The common runtime contains the logic necessary to distribute logic, handle failures, and send data between application components, such that the applications themselves do not need to do this; in most cases the frameworks do not permit such logic to be written by an application. Each framework varies in both its compute and data models, possibly supporting multiple of each based on the framework features and capabilities. Each compute and data model trades off different aspects in order to achieve better performance, application expression, or domain-specific optimizations.

2.3.1 Framework Development

The first modern iteration of these frameworks was MapReduce [36], a framework developed by Google to deploy applications across myriad commodity servers. The data model was a simple collection of key-value pairs. The computation model was based on two phases: the map phase applies an operation to one key-value pair to produce zero or more downstream key-value pairs (*e.g.*, splitting a string into whitespace-delimited words) and the subsequent reduce phase takes all key-value pairs sharing the same key and produces zero or more resulting pairs (*e.g.*, counting all identical words from the map phase to emit a total count). The computation model requires two functions, a mapper and a reducer to perform their respective operations, and imposes a global barrier between the two phases: all input key-value pairs must be processed by the mapper function such that a “shuffle” stage can aggregate all identically-keyed pairs to send to a single reducer function. The MapReduce runtime deploys the map and reduce functions across an arbitrary number of machines and coordinates the data distribution between machines. The runtime detects and mitigates failures in this

distributed computation by persisting data to disk and restarting failed components, replaying data that was affected by the failure to continue the operation.

The simple abstraction of MapReduce combined with its generality enabled widespread adoption. Apache Hadoop [7] is a prominent open-source implementation of this model in Java. The Google File System (GFS) [54], a distributed storage system used to transfer data in MapReduce applications, was concurrently implemented in Hadoop as the Hadoop File System (HDFS) [22]. A large ecosystem of related frameworks (*e.g.*, Apache FlumeJava [29], which coordinates series of MapReduce invocations to transform data collections, and Apache YARN [124], which decouples the MapReduce model from the storage system) and higher-level frameworks emerged around this model. The Apache projects Hive [117], HBase [8], Phoenix [9], and Pig [10] all provide a new service on or interface to an underlying Hadoop or HDFS installation.

Dryad [67] extends the models and concepts from the MapReduce paradigm to a more general framework for distributed computation. Instead of only a map and reduce phase to encapsulate application logic, Dryad provides a directed acyclic graph (DAG) of vertices as the compute and data model. Each vertex contains some imperative code that responds to events (*e.g.*, a new data item arriving); each edge specifies a communication link from a source to a sink vertex. Dryad executes an application by distributing each vertex across machines and connects them with the appropriate communication channels based on location (*e.g.*, a network socket between vertices on different machines or a shared memory mechanism for vertices in the same process). Although each vertex is executed by a single thread, the concurrent scheduling of multiple vertices across the entire cluster of hardware resources enables Dryad to transparently scale out computation, *i.e.*, the logic of a single vertex does not depend on the scale of the overall application. DryadLINQ [131] provides a high-level interface from the .NET framework that encapsulates Dryad DAG descriptions into language-level constructs using the C#'s Language Integrated Query (LINQ) feature. This enables Dryad DAGs to be constructed using a SQL-like language.

2.3.2 Contemporary Frameworks

Contemporary cloud computing frameworks adopt the DAG abstraction for expressing application logic. They each build upon earlier frameworks to provide simpler abstractions and performance improvements, in some cases leveraging new machine capabilities (*e.g.*, larger amounts of main memory) or domain-specific performance improvements (*e.g.*, caching based on the data model).

Spark [133] provides the abstraction of Resilient Distributed Datasets (RDDs), in-memory persistent datasets, to applications that specify data parallel operations that sequentially transform the input data. Spark generates an execution plan based on the coarse-grained description of the operation, which the application specifies using transformations such as `map` and `reduce` on an input RDD. Spark is structured as a centralized program that drives

computation through successive invocations of these execution plans as tasks on cluster resources. The Spark ecosystem includes many libraries providing a large range of functionality on top of the fundamental abstractions of the framework, including machine learning [92], graph processing [58], SQL interfaces to Spark queries [129], and stream processing using micro-batching [134].

In contrast to Spark's centralized scheduling, stream processing frameworks set up a mostly static DAG that provides low-latency processing. MillWheel [4], Apache Storm [120] (and its successor Heron [73]), and Apache Flink [27] all fall into this category, as they deploy a directed graph of tasks across a cluster of machines and orchestrate a static flow of data between these tasks as dictated by the DAG. Compared to centralized scheduling frameworks, stream processing frameworks trade off fault recovery (they must resort to higher-latency mechanisms to restart computation) for lower latency, as no centralized scheduler must be employed to start or manage a computation. Naiad [96] structures computation using its timely dataflow abstraction, which represents many other abstractions (*e.g.*, stream processing and iterative loop styles of computations to provide online query and model training in the same application) using an event-based compute abstraction for each vertex. Each vertex responds to events, which are tagged with a logical timestamp and contain either incoming data or signal an end of an epoch (*i.e.*, no further data events for a given timestamp will be received in the future). The Naiad framework coordinates message delivery and callback execution of arbitrary code in each vertex.

2.3.3 Performance

Cloud computing frameworks impose a high overhead for compute-intensive workloads. Many of the most prominent cloud computing frameworks use the Java Virtual Machine (JVM) [84] as their managed runtime. The overhead of boxing primitive types (*e.g.*, using an `Integer` object instead of a corresponding architectural data type such as `int64`) as well as the small object allocation leads to a $1.9 - 3.7\times$ slowdown compared to an optimized application written in C++ [55, 64] for a standalone application. Large-scale analytics workloads run on the Spark framework run $16 - 43\times$ slower the equivalent workload on an optimized C++-based framework [86, 87]. Even approaches wherein a JVM-based framework uses the Java Native Interface (JNI) to call from Java bytecode into a C++ task impose an overhead due to inefficiencies in the framework (*e.g.*, due to Spark's centralized scheduling).

Recent advancements have mitigated some aspects of the performance overheads. Weld [103] converges on a single in-memory data model that frameworks interface with, emitting bytecode for Weld to operate directly on the data instead of converting to and from the format for a given framework (*e.g.*, boxing and unboxing numerical data for Spark). Recent advancements in Spark's scheduler [38, 39, 102, 125] have mitigated some of the overhead of the centralized scheduling architecture. Nimbus [87] is a de novo rewrite of a cloud computing framework,

written in C++, using execution templates to driven iterative computation without constructing the DAG each time (*i.e.*, by caching the previous scheduling decisions).

Cloud computing frameworks' data models make tradeoffs for certain features over raw performance. Specifically, several popular frameworks employ a fine-grained fault tolerance mechanism, *i.e.*, one that enables an operation to be transparently restarted when a fault is detected so that existing successfully-completed work does not have to be redone. MapReduce-based frameworks do this by persisting data to disk between the map and reduce phase. Spark does this by re-executing tasks to restart the failed subset of computation in memory from the latest checkpoint. Although these guarantees may be useful when serving analytics workloads (*i.e.*, where the latency of restarting any given operation may violate an SLA), they do impose overhead when coarse-grained recovery (*e.g.*, a simple restart from the initial input) would suffice for a given workload.

2.4 TensorFlow

TensorFlow [1] is a cloud computing framework targeting machine learning workloads. It can distribute an application, represented as a graph (*i.e.*, a DAG) of operations, over a cluster of heterogeneous resources, including CPUs, GPUs, and custom-designed ASICs such as the Tensor Processing Unit (TPU) [69]. As with other cloud computing frameworks, TensorFlow separates the application logic in the graph from the execution of that logic, enabling the same logic to scale from the scale of a laptop to a data center with minimal modification. The library and runtime are written in C++ to minimize the framework's overhead, but the framework exposes a Python API to both assemble the application logic and execute it, either within the same program or different programs. The work in this dissertation specifically builds upon TensorFlow. Therefore, we discuss its architecture and operation in detail.

TensorFlow uses dataflow to express application logic as operations on tensors. Dataflow represents application logic as a DAG: nodes represent operations and edges specify the propagation of the results. Each node consumes and emits only tensors, multi-dimensional arrays of a single elementary data type (numerical or string). TensorFlow's features and design decisions are based in its focus on the machine learning domain; most of its predefined nodes perform stateless numeric computations. The nodes that do hold state are explicitly designated for this purpose, *e.g.*, holding persistent weights in a machine learning model as it is trained on successive examples.

TensorFlow graphs process tensors using a push model of execution. A feed is a group of values that populate placeholder inputs to a graph. The client program (*i.e.*, a program that interacts with the TensorFlow runtime via its API, typically written in Python) inserts a feed into a graph and requests the value of the graph's output, *i.e.*, a downstream group of tensors. The TensorFlow runtime responds to this request by propagating the feed through the graph based on a strict set of rules [2], many of which are descendant from tagged-token dataflow

(TTDF) [18]. The result is a new feed (corresponding to the downstream tensors) and side effects, *e.g.*, updating a stateful variable held by a graph node.

A batch of feeds is a collection of related feeds that together form the input for an application, *e.g.*, a series of labeled images on which to train a machine learning model [40]. Upon startup, TensorFlow applications initialize stateful variables (*e.g.*, large tensors that hold persistent data across successive feeds) and then process a batch of feeds. The client program drives each feed through the TensorFlow runtime in succession, updating the variables each time. A single graph may process concurrent feeds from a batch, but extra information must often be added to the graph to serialize parallel updates to variables.

TensorFlow's dataflow rules restrict the ways in which a given node or graph may operate, but enable both the system and practitioners to make assumptions about the operation of a graph. Most cloud computing frameworks choose a flexible mode of operation whereby each edge in the DAG may hold an unbounded number of tokens. This is useful for operations where the number of outputs produced by any given input cannot be known at the time of construction, for example splitting a string on whitespace tokens to produce a set of word values as output. In contrast, TensorFlow enforces rules on the operation of each node that restrict the behavior based on the following set of guarantees:

- Each edge may hold at most one value at a given time.
- Each node is eligible to be executed if and only if all of its input edges contain a value, *i.e.*, they are non-empty.
- Each node consumes its input values when executed, *i.e.*, the values are not available for future executions of the node. A notable exception to this is reference values, which point to persistent in-memory values, *e.g.*, large tensors that hold the weights in the model that is currently being trained.
- Each node must produce exactly one output for each of its declared outgoing edges per execution.

These set of rules contribute to an application logic that is similar to the evaluation of a mathematical formula. This enables the TensorFlow framework to rewrite portions of the graph for higher performance, *e.g.*, removing unused portions of the graph or rewriting common portions of the graph into a single aggregate node (as is done by the XLA compiler for TensorFlow [77]).

Monolithic application graphs can be decomposed into smaller graphs separated by TensorFlow queues. Each TensorFlow queue separates successive phases of an application into distinct independent graphs; these graphs are linked by the TensorFlow queue data structure that buffers feeds between upstream and downstream graphs. This separation increases concurrency within an application. For example, an initial phase typically reads in a batch of

feeds from storage (training examples) and a subsequent phase trains a model based on the values. Each graph receives a feed from its upstream queue via a dequeue node and sends the resulting feed to its downstream queue via an enqueue node. Each graph is driven by a *queue runner*, a Python thread containing no application logic that drives feeds through a graph by requesting the value of the graph's enqueue node.

The TensorFlow framework targets applications that process a single batch of feeds per invocation. While internal concurrency between feeds is possible, TensorFlow does not natively distinguish between feeds belonging to different batches of inputs. A multi-batch TensorFlow application must rely on the client program to disambiguate between feeds from different batches. The necessary performance penalty for involving the client program is the copying of data into and out from the TensorFlow runtime. Machine-learning TensorFlow applications are not inhibited by this design decision, as they typically do not concurrently process multiple batches; their graphs contain compute-intensive coarse-grained operations that have little overhead to construct on a per-batch basis.

3 Aggregate Genomic Data Format

The file formats produced and consumed by contemporary bioinformatics applications were designed for single-machine workflows. Originally constructed when bioinformatics datasets were tractable for use in single-machine computation, these file formats used to store and transmit bioinformatics data limit the distributed processing of the larger datasets that modern high-throughput sequencing machines can generate. Their limitations are primarily based on their monolithic file structure and row-oriented data format.

This chapter introduces Aggregate Genomic Data (AGD), a new, extensible file format targeting bioinformatics workflows that enables high-throughput distributed processing of genomic datasets on both single machines and scale-out workflows. AGD takes design cues from modern file formats used in data center-scale systems and tailors itself to the access patterns used in typical bioinformatics applications. Existing formats are compatible with AGD, which can be converted to or from contemporary formats at any point in a dataset's lifecycle.

We begin this chapter by outlining the goals for AGD, namely that it should be interoperable with existing formats while providing better scale-out features than existing formats. We then provide the architectural details of AGD, emphasizing how each attribute of AGD's architecture supports its design goals.

3.1 Design Goals

The design goals of AGD consist of alleviating the restrictions of traditional formats outlined in chapter 2.2.3. These formats were designed at a time when datasets were processed on a single machine through successive invocation of applications in a bioinformatics workflow. Their emphasis on simple, plain-text formats may have been appropriate for their time and scale, but attempting to rely on them to support bioinformatics dataset processing across a cluster of machines imposes limits on throughput and scalability. AGD is designed to supplant these formats to enable bioinformatics workflows to scale using modern big-data frameworks

while remaining compatible for the sake of interoperability with the existing ecosystem of tools in the field.

Efficient I/O: AGD enables applications to perform only the necessary I/O for their given transformations. The format does not force any application to read the entire dataset, involving I/O and computationally expensive preprocessing (*e.g.*, parsing and decompression), in order to read a small subset of the fields. Applications that only append fields to each record in the AGD dataset must only write out their additional results; they do not have to rewrite the entire dataset due to differing input and output formats, thereby semantically duplicating the data (*e.g.*, FASTQ to SAM during alignment). AGD enables parallel access for read and, to a customizable extent, writing without relying on a specific storage system or cloud computing framework.

Format unification and interoperability: AGD can be used as the format for input and output of applications at all stages of a bioinformatics workflow. Instead of each stage using a unique input and output format, the latter of which must be accepted as input by the subsequent stage, AGD unifies several bioinformatics formats into a single format. This enables code reuse between different applications (*e.g.*, by providing a library) and avoids expensive and inefficient parsing code being reimplemented by each of these applications. However, AGD remains compatible with each of the supplanted file formats, with simple conversion applications able to input and output AGD datasets at each step in a bioinformatics workflow.

Long-term storage: Larger, higher-quality genomic datasets are being produced with increasing throughput, making long-term storage an important consideration for file formats. AGD uses a block compression scheme that enables different fields of each record to use a different compression scheme. At the same time, AGD supports the use of uncompressed data formats where it may confer a performance advantage, such as when producing a transient AGD dataset between two applications in a workflow (*i.e.*, one not required for the final workflow output). Reprocessing a dataset stored on a high-latency long-term storage system requires that only the necessary fields be read and decompressed due to this block compression design, reducing the I/O and computation overhead for such a process.

3.2 Architecture

We describe the details of the AGD format and how applications perform common operations on AGD datasets. In particular, we emphasize how each aspect of the format enables AGD to achieve the goals set out in §3.1. AGD can be used to store arbitrary data, but we focus on the aspects related to its application in storing bioinformatics data.

Although the AGD format is not as feature-rich as other contemporary data formats, the simplicity confers advantages in performance, operation, and on-disk storage for the domain

of bioinformatics. AGD supports the common patterns of (a) scanning a dataset while reading only a subset of the fields and (b) appending an additional field to each record in the dataset without requiring the entire dataset to be rewritten to accommodate this update. These operations may be performed in parallel in datasets without more than basic storage system support; the system does not have to support concurrent file access. We discuss these design aspects in greater detail in this section.

3.2.1 Format

An AGD dataset is a table of records, each of which contains one or more fields (*i.e.*, a relational table). All records adhere to an identical flat schema: each record contains all fields, though each field may be null for any given record. AGD does not supporting repeating fields or nested schema hierarchies; each field occurs exactly once per record and cannot be defined by a further decomposition into AGD records.

Record fields each have a fixed type and a variable size. Record types determine how each field should be interpreted by an application. Currently supported record types are character strings (*e.g.*, for metadata strings), structured data (*i.e.*, variable-type data encoded using Google's protocol buffers [123]), and a compact three-bit representation for genomic bases (with 21 bases packed into the lower 63 bits of a 64-bit integer). The size for a given field may vary between records, *e.g.*, a variable length string field. The following types are used for the genomic data fields:

- Bases: compact three-bit encoding
- Qualities and metadata: character string
- Alignment results: structured data

AGD can easily support more types as needed, but these types were sufficient for the body of work encompassed in this thesis.

AGD is structured as a column-oriented dataset: identical fields of successive records are stored adjacent to each other in the file format. In contrast to record-oriented formats commonly used in bioinformatics (*e.g.*, FASTQ, SAM, BAM, VCF), AGD stores each record in a column-oriented format. Each column is stored in a dense format; subsequent records are stored adjacent to each other within a column on disk, without additional padding. Columns are separated in the storage system using separate files.

Each column is partitioned into a group of chunk files, the fundamental unit of storage used for AGD data. Each chunk file holds a contiguous subset of records for a given column, with each column typically being split into equally-sized chunk files (based on the number of records). Different columns may choose different sized chunks, based on the size of the data

contained in each field and storage system characteristics. For the work in this thesis, we fix all columns to have equal-sized chunks.

A numeric ordinal relates different fields of each record across multiple columns. Each record is assigned a unique ordinal representing its index in the logical record table of the AGD dataset (starting from 0). Each chunk file is identified by the field and ordinal range of that column it contains. A given record can be assembled by looking up the relevant chunk files for the ordinal and selecting the record out of each chunk file.

Chunk File Format

Each chunk file contains fixed-size header and a variable-size data payload. The header describes the chunk contents and the data payload contains the dense sequence of records. The data payload may be stored as a compressed block or as uncompressed records. Each field may independently choose a compression algorithm of its choice.

Chunk files use a relative index to identify record boundaries. The data payload contains a relative index of record sizes followed by the records themselves, without padding. Most items in genomic datasets contain similar-width fields, *e.g.*, the bases and qualities are usually clipped to a fixed length and the metadata fields usually follow a similar size. This means that storing the index to each record as a relative size, instead of as a byte offset into the record data, creates a highly-repetitive block of sizes for the index, which is easily compressible.

Chunk Directory and Manifest File

All chunks are stored in a single directory and are identified by a manifest file. Each chunk file is assigned a unique name based on the dataset name, the column name (*e.g.*, bases, qualities, metadata, results), and the ordinal of the first record it contains. The manifest file is stored in the JSON format and contains a list of chunk file names and columns. Bioinformatics applications that update an AGD dataset (*e.g.*, alignment) by writing out a new column will modify the manifest file to reflect the new column. If the manifest is ever lost or corrupted, it can be regenerated by scanning the chunk files and the metadata contained in their respective headers.

Figure 3.1 shows an example of a compressed AGD dataset. This dataset, containing aligned reads, is named ‘‘`MyData`’’ and has four columns: bases, qualities, metadata, and results. The dataset is split into N chunks, each containing 1000 records (based on the chunk size of the first record) and identified by their unique name and ordinal. An application accessing this dataset will first read the metadata file, determine which chunks and columns it needs to perform its operation, and then read the corresponding chunk files into memory, decompressing the data block from each into memory to access the records.

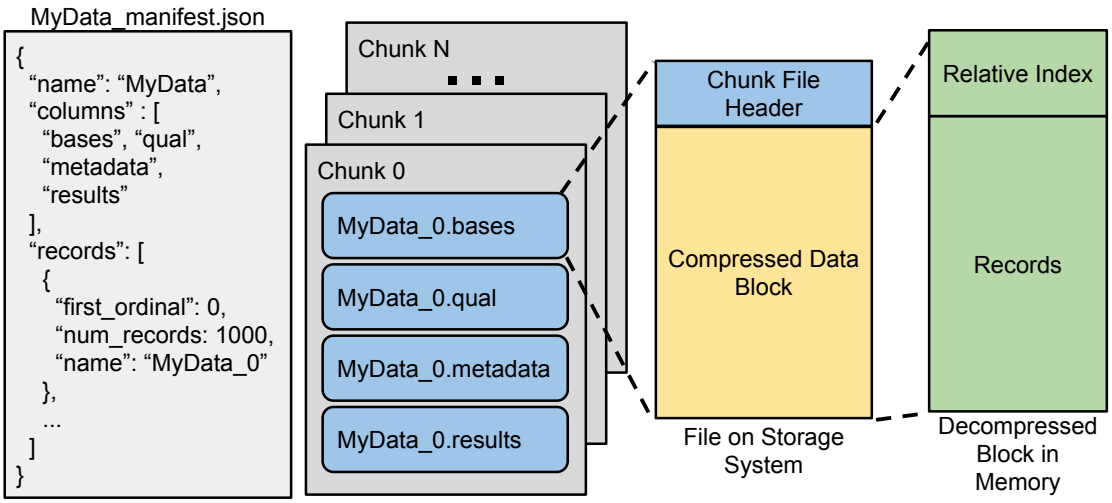


Figure 3.1 – An example dataset in the AGD format.

3.2.2 Operations

The AGD format is designed such that chunks can be distributed to independent processes without any coordination for reading or writing. Each chunk file is self-describing: a process only requires the information contained within the chunk file itself in order to access the field values for the records it contains. Unlike other file formats, no process must access any global, dataset-wide state in order to read or write a chunk. The chunk files for each chunk (one for each column) are related based on the information in each chunk file, *i.e.*, the dataset name and the ordinal contained in each header.

An application reads an AGD dataset based on the metadata file. The metadata file contains the chunk names and locations, which the application uses to access the files from the storage system. A local application can stream each chunk file through memory. A distributed application can distribute each chunk to independent workers or tasks for concurrent processing where possible, depending on application semantics. If an application requires only a subset of the fields to perform its operation on the dataset, it only reads the chunk files for the associated columns.

Applications that require fast random access to records in a chunk build an absolute index when reading the chunk file. Certain applications, such as sorting, require random access into a chunk file to assemble a sorted result. Due AGD's use of a relative index, a naive approach to this operation would require an application to sum the prefix of all record sizes in the index for every record accessed. To alleviate this linear operation, applications requiring random access scan the relative index once upon reading each chunk file to build an index of in-memory byte offsets.

AGD can update datasets in place by writing out new chunk columns. Operations that update a dataset may perform their operations on only the necessary chunk files. An update operation

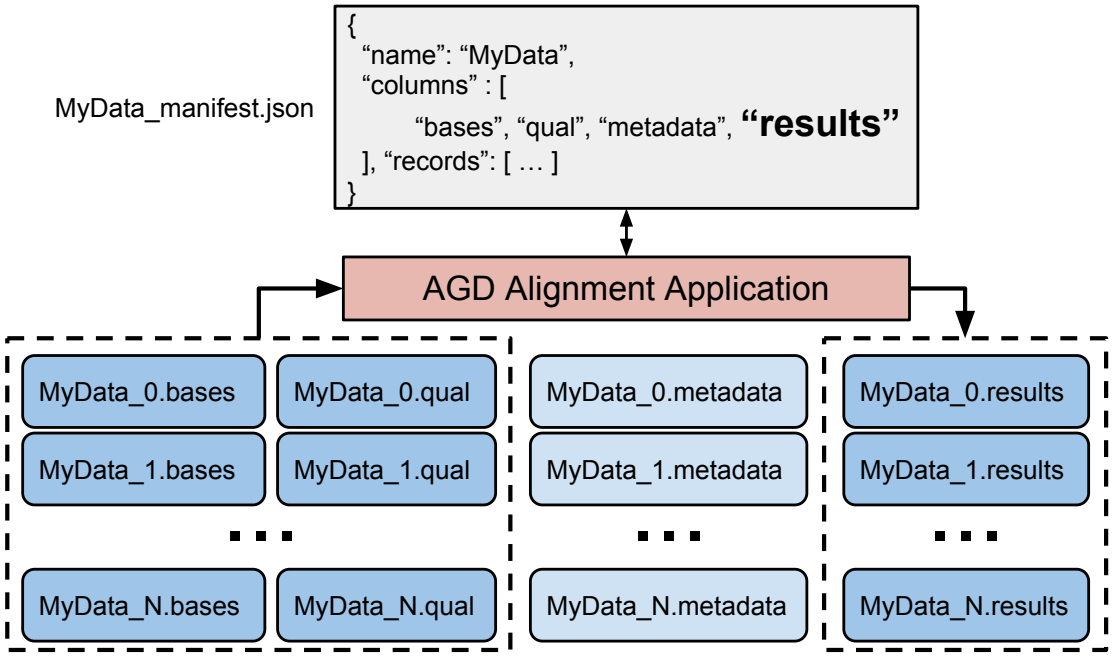


Figure 3.2 – The access patterns of an optimized alignment application for AGD datasets. The application reads in a subset of the columns (bases and qualities) and writes out a single result column (results).

to dataset may read in a subset of columns and either rewrite one or more of the columns (*e.g.*, changing the compression scheme of a block to decrease file size) or produce a new column (*e.g.*, alignment, which produces one result column and possibly additional secondary result columns).

AGD does not require storage system support for concurrent update operations. Unlike other column formats that store multiple chunks in different regions of the same file, AGD does not require the storage system to support concurrent updates to a file. This advantage of AGD is enabled by the fact that work distribution can be performed by distributing chunks to separate application elements. Each chunk is a separate entity within the storage system; I/O requests related to each chunk can be done without coordination between application elements. This enables AGD to be stored on a wide array of storage systems.

The wide variety of storage system support is an important component for AGD to serve as a format for long-term storage in object stores. The most cost-effective storage solutions available from cloud providers are cold storage solutions, such as Amazon Glacier [6] and Azure Blob Storage Archive [93], which store data for $0.004 and $0.0025, respectively, per GB/month. These cold storage solutions are blob storage systems, with a flat key-value namespace for storing unstructured text or binary data. AGD is compatible with these systems despite their dearth of features: each dataset can be segregated based on namespace with each chunk file corresponding to a unique key-value pair. AGD's compatibility with these minimally-featured

systems enables it to be used in both a high-performance (and higher-cost) storage system for when a bioinformatics application reads from and writes to it as well as a long-term storage system for the resulting data. Genomic data will likely be retained for many years, as the differences between WGS data over time can yield research and medical results.

Figure 3.2 shows an example access pattern for an alignment application. Using the manifest file, the application first determines the location of the chunk files it needs to access in the dataset. Only the bases and qualities columns are accessed in the dataset, as the alignment operation does not read or write the metadata field for any record. After each chunk is aligned, the application writes out the corresponding results column to the storage system. When all chunks have been successfully aligned, the application finally updates the manifest file to reflect the additional ‘‘`results`’’ column. If the alignment application is distributed, it may distribute chunks to different tasks from a common queue such that each task can process chunks in parallel without additional coordination.

An AGD dataset can be transformed into any contemporary bioinformatics format by a simple scan through a dataset. Recall from chapter 2.2 that bioinformatics data is stored in a table of records. Because AGD shares this data format, bioinformatics data can be transformed to or from AGD at any point in the data's life cycle provided the source and destination formats are compatible: an unaligned AGD dataset may not be convertible to a format containing alignment results (*e.g.*, SAM or BAM) if no such results are yet available.

3.3 Summary

This chapter introduces AGD, a columnar, chunked file format tailored for bioinformatics data and applications. Although it is not as feature-rich as other formats that store structured data, its simplicity lends it to serve the needs of this targeted domain well. Its dense data storage format enables applications to rapidly scan through a dataset. The column-oriented design allows unused data fields to not be read in from the storage system. AGD's column chunking provides a mechanism to mediate parallel access to a dataset, distributing chunks to different tasks to enable coordination-free processing when possible.

4 Pipelined TensorFlow

Cloud computing frameworks are an important tool for constructing scale-out applications in the data center. This is due to a) their use of an API to separate the application logic from the details of the execution and b) a distributed runtime that translates the application logic into an execution on the available hardware. However, contemporary cloud computing frameworks lack adequate performance for bioinformatics workloads due to design decisions that impose overheads for features that bioinformatics applications do not require.

This chapter introduces Pipelined TensorFlow (PTF), a cloud computing framework built on top of TensorFlow that is specifically designed to meet the needs of bioinformatics application pipelines. PTF expresses application logic as a linear sequence of transformations (*i.e.*, a pipeline). Each of these transformations is encapsulated into a PTF *stage*, which at its core is a standard TensorFlow graph. PTF *gates* coordinate the asynchronous and parallel execution of stages throughout the application pipeline by decoupling data dependencies between successive stages and buffering data between them. A PTF application pipeline executes across a cluster of machines as an indefinitely-running service that processes multiple concurrent requests. PTF builds upon TensorFlow's efficient native execution engine for low overhead, all with a minimal addition of code to the TensorFlow codebase.

The key insight of PTF is that a few careful additions of code to TensorFlow can alter its behavior to serve more effectively as a framework for indefinitely-running applications that concurrently process requests. PTF adds a small amount of code into the TensorFlow codebase and organizes the TensorFlow graph into a specific pipeline structure. This enables TensorFlow to expand beyond its purview of machine learning applications such that it can operate in a manner more akin to a tradition cloud computing framework. Due to the design of PTF, these applications executing on this new cloud computing framework can reuse all the components of TensorFlow, such as its robust distributed runtime and decoupling of application logic from execution details.

The content of this chapter proceeds as follows:

- §4.1 describes the architectural goals for PTF.
- §4.2 and §4.3 introduce the fundamental building blocks of PTF, *i.e.*, the data types used throughout the PTF framework, stages, and gates.
- §4.4 discusses the construction of PTF pipelines and mechanisms related to resource bounding during pipeline execution.
- §4.5 demonstrates PTF's architecture for adapting simple local pipelines to execute on clusters of machines.
- §4.6 discusses the compatibility of PTF with the rest of the TensorFlow ecosystem.
- §4.7 summarizes the contents of this chapter.

4.1 Introduction

A key component to designing scale-out bioinformatics workflows is a suitable framework to support such applications. Designing PTF requires an assessment of the fundamental goals for such a system because, as we have discussed in §2.3, many of the existing popular choices of frameworks on which to build such workflows impose runtime overhead that, although is acceptable for the data center applications for which they were designed, imposes a costly overhead for compute- and memory-intensive bioinformatics applications. We begin this chapter by discussing the general goals for PTF.

At a high level, we design PTF to concurrently process an indefinite stream of requests on a scale-out workflow across multiple machines. Each request contains the necessary information to transform a single dataset (*e.g.*, the filenames of an input AGD dataset to be processed by the workflow). The user submits this input data to the application as a single request and receives, as output, an aggregate response (*e.g.*, the filenames of the output AGD dataset, or the names of the columns that were written). PTF processes multiple requests concurrently on the same resources in order to share resources efficiently between concurrent requests.

PTF builds upon TensorFlow due to its proven ability to serve as a cloud computing framework, but adapts it to serve as a framework for concurrently processing multiple requests via careful additions to the code and choice of abstractions. A PTF application is constructed using the standard TensorFlow API (in Python) and executes on TensorFlow's existing high-performance runtimes. Unlike TensorFlow's standard execution model that only supports a single request per invocation, PTF overlays the semantics of a framework that supports multiple concurrent requests, more akin to contemporary cloud computing frameworks. With only a few thousand lines of code added to the source code repository of TensorFlow, PTF applications run directly on the TensorFlow runtime to perform not only the data processing functionality, but also the control plane logic as well (*e.g.*, to limit resource usage and track concurrent requests as they move through the application).

The key insight is to adopt the feed from TensorFlow as the fundamental unit of operation in a request, but to add *metadata* to each feed as well as additional data structures within the TensorFlow runtime to interpret the metadata. This approach enables PTF to reuse much of the existing functionality within TensorFlow with minimal modification; PTF does not change any components of the core TensorFlow runtime or existing library of nodes. PTF uses the metadata to capitalize on TensorFlow's dataflow semantics while interposing more flexible semantics at specific junctures in the graph.

4.1.1 Goals for PTF

In this section, we introduce a set of design goals for PTF.

Separation of logic from execution: an important component of a modern cloud computing framework is the separation of the user-defined code that encapsulates the application logic from the runtime code that executes the application. Typically, the latter is provided by the framework itself; executing a distributed application is not only a difficult task, but also one composed of myriad common elements, *e.g.*, work distribution, synchronization, load balancing, and I/O. It is critical that PTF include an abstraction for application logic that requires minimal knowledge of the execution runtime so that practitioners, who may not be experts in constructing data center applications, may create bioinformatics workflows that scale out across all available resources.

Minimal framework overhead: PTF must use a minimal amount of system resources so that the maximum available resources may be devoted to the application logic. Bioinformatics applications are capable of saturating one or more resources allocated to them; specifically, each bioinformatics application each contain a component that is bound by a hardware resource constraint. This may be CPU cores, memory for buffers or a precomputed table, or a hardware accelerator such as a GPU. In order for PTF to be a viable framework for bioinformatics workflows, it must impose a minimal framework overhead, so as to minimize the already large latency for such workflows.

Concurrent request processing: PTF must be able to manage a persistent allocation of cluster resources that can be used for concurrently processing an indefinite stream of requests. Due to the high latency of allocating and deallocating coarse-grained hardware resources in a cluster, PTF must be able to use a static set of resources allocated to it upon initialization for indefinite processing. PTF must track each request as it is processed by the application logic in order to be able to determine when the request terminates, *i.e.*, when all of the initial input data has been fully transformed by the workflow. This resource sharing is essential for increasing throughput on a fixed set of cluster resources, which in turn lowers the cost of running a bioinformatics workflow.

Isolated execution semantics: while sharing hardware resources between concurrent user requests, PTF applications must be able to present the abstraction of an *isolated* application to each request as it is processed. Specifically, the computational result of processing a single user request must not be affected by the set of other requests that may be concurrently processed by the application. Only the performance of such processing (*i.e.*, the latency of processing the request) may be affected.

Scale-up on a single machine: PTF must be able to scale out a computation across all the cores, memory, and I/O subsystems on a single machine. Furthermore, it should do this with minimal changes to the application logic, *e.g.*, configuring scaling parameters; the application logic itself must not need a large rewrite in order to adapt to each machine or hardware configuration.

Scale-out on multiple machines: a PTF application must be able to scale out computation across multiple machines. Even a powerful single machine may not be able to contain adequate resources for a bioinformatics workflow. Moreover, a single-machine solution may not be the most cost-effective manner to provision a workflow: a single machine provisioned for multiple stages of a workflow may be more expensive than multiple smaller machines, each provisioned according to the stage they perform. Therefore, PTF must be able to place different components of a bioinformatics workflow on different machines and be able to scale out each component across additional machines as they are available.

Bounded resource utilization: a PTF application must be able to bound resource usage within an application to avoid exceeding resource limits. This is critical when different subcomponents of an application exhibit different rates of throughput. For example, a compute-bound application must be able to bound an upstream read operation in order to bound the amount of preprocessed data in memory. This is crucial for memory-intensive bioinformatics applications, which must always operate in core, without swapping, for efficiency purposes.

Non-Goals

Fine-grained fault tolerance: as we have discussed in §2.3, the overhead for mechanisms related to live fault recovery (*i.e.*, recovery during a computation such that the computation can continue without restarting completely) is non-trivial. Moreover, bioinformatics workflows typically create a new dataset or append a column; overwriting the original input data is uncommon. In order to minimize the cost of these mechanisms for every request during normal operation, PTF takes the approach of a coarse-grained failure recovery mechanism: when a failure is detected, the workflow must be restarted and all requests active at the time of the detected failure must be restarted. As bioinformatics workflows are not typically run as

a response to user requests or otherwise bound by a tight latency SLA, we believe this is an appropriate tradeoff to make for an increase in performance.

4.1.2 Using TensorFlow as a Base

PTF is constructed as a framework within its namesake, TensorFlow, as a small, careful addition of code to the default TensorFlow codebase. As we discussed in §2.4, TensorFlow's execution model is not strictly amenable as a drop-in framework for PTF: its single-input style of execution (*i.e.*, one application instantiation per user request) is at odds with PTF's goal of serving as a framework for building indefinitely-executing applications that concurrently processing multiple requests. However, a few important aspects make TensorFlow a good choice for this application (*i.e.*, bioinformatics workflows):

High-performance runtime: TensorFlow was born out of Google's effort to create a high-performance machine learning application framework that scales across large clusters of machines while efficiently using the hardware resources. TensorFlow has been used and refined by thousands of users and several companies in the pursuit of even higher performance and a larger feature set [116]. By using TensorFlow as a base, PTF capitalizes on this prior work in this performance and scale-out capability that is necessary for high-performance bioinformatics workflows.

Application logic abstraction: TensorFlow's *dataflow* abstraction specifies the order in which a single feed of input may be transformed based only on data availability (of intermediate values). By construction, TensorFlow applications contain only the application logic. This frees applications written in PTF from having to express detailed execution logic by taking advantage of the TensorFlow runtime, delegating all the details to it.

Dataflow for workflows: the dataflow abstraction maps well onto bioinformatics workflows. As we discussed in §2.2, bioinformatics workflows contain a sequence of consecutive transformations. This maps well to the dataflow abstraction at this higher level, but also at a lower level: within each application of a workflow can be further broken down into a linear sequence of reading, preprocessing, computation, post-processing, and writing. At all levels of parallelism (*i.e.*, within a phase and between phases), the dataflow parallelism frees developers from having to consider the intricate details of executing a massively parallel program across multiple machines.

We choose TensorFlow as the basis for PTF's functionality over traditional cloud computing frameworks due to its native performance. The performance overheads of contemporary cloud computing frameworks, mentioned in §2.3, would hinder their use in many bioinformatics applications. Bioinformatics applications typically saturate either the memory bandwidth or, in most cases, the CPU of a given machine. Although foreign function interfaces do exist

to use functionality of a native library from a managed language's runtime (*e.g.*, the Java Native Interface [83]), these interfaces still impose overhead when compared to a purely native application [86]. We choose TensorFlow due to its native runtime in order to maximize performance of a bioinformatics workflow.

4.2 Definitions

We first discuss the high-level components of PTF as a groundwork for further sections.

4.2.1 Feeds and Batches

The fundamental unit of input to a PTF application is a TensorFlow feed (a group of tensors that populate input placeholders in a TensorFlow graph). We adopt this definition from TensorFlow because PTF shares TensorFlow's runtime. A feed contains input that represents a single item for the workflow to process, but the exact definition of the item depends on the application. A frequent feed type is a file location as well as the necessary authorization. For example, the feed type might contain the following fields to access a file in a distributed storage system:

- The name of a file
- The directory in which it is located
- A username to access the storage system
- A token to authenticate the user.

PTF groups together multiple related feeds into a batch. Each batch consists of a finite list of feeds, each of the same type, that are related to each other based on their origin, *i.e.*, the original user request. A PTF application transforms a batch asynchronously as its constituent feeds are processed through the application. The number of feeds in the batch may be altered and the feeds reordered due to parallelism in the PTF application. In many cases, PTF applications will have the following feed types:

- Input feed type: a file location on a storage system (*e.g.*, a path on the file system, or a key in a shared storage system), plus possible additional information to access the file (*e.g.*, authorization tokens to access the shared storage system). Each of these pieces of information (file path and authorization information) would typically be represented as tensors of type string.
- Output feed type: a file location of the output dataset. This location may represent either an additional column added to the original dataset (*e.g.*, in the case of alignment,

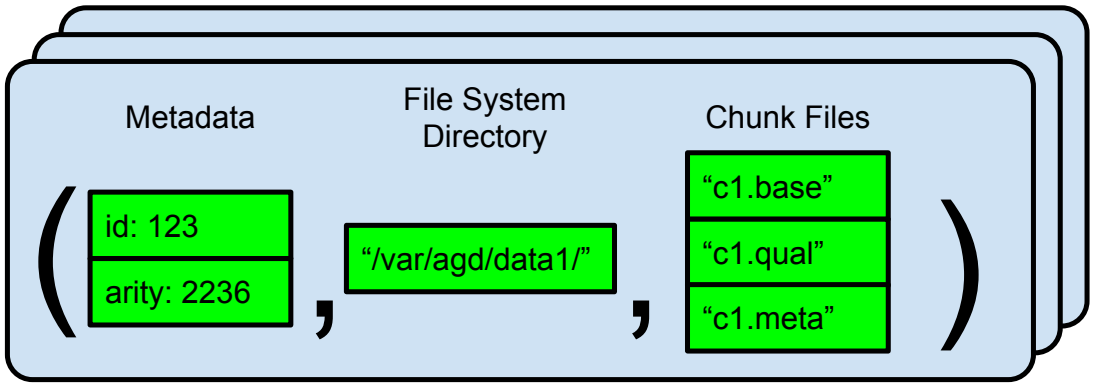


Figure 4.1 – The components of a feed type for an application that reads input in the AGD format from a file system. This includes the metadata tensor, the base directory containing all chunk files, and a vector of file names, one for each column for this AGD chunk.

which only adds one or more results columns) or the location of an entirely new dataset (*e.g.*, in the case of sorting or a dataset conversion).

In aggregate, each input batch typically represents the location of a dataset to be processed and the output batch represents either the additional columns added to the dataset (*e.g.*, results columns from the alignment process) or the location of all the chunk files of the new dataset.

A user submits a batch to a PTF application via a single API call. A PTF application processes the batch contained in each user request and aggregates the resulting batch as an output for the response returned to the user. This is an important distinction to draw from typical TensorFlow applications, discussed in §2.4, which are driven by the user on a per-feed basis. In this case, PTF applications operate in a similar manner to traditional cloud computing applications: the user has an opaque view of the inner workings of a PTF application so the user does not bottleneck batch processing.

4.2.2 Metadata

Metadata is attached to each feed consisting of two pieces of information: (1) a unique identifier associated with the originating batch (identifying the batch as it is transformed through the PTF application) and (2) an *arity* representing the number of feeds associated with each batch. By construction, the metadata for all feeds in a batch is identical. The arity may change as the PTF application transforms the batch, but the ID is unique to the batch throughout the application. Both metadata components are encoded as integers in a single TensorFlow tensor.

Figure 4.1 shows an example of a batch of many feeds, with metadata, for an application that reads an AGD dataset from a file system. Each feed corresponds to a single chunk in an AGD dataset and contains two pieces of information: the base directory where all chunk

files are stored (a string value tensor) and one tensor containing a key for each AGD chunk column to be read (as a vector-shaped tensor of type string) for each chunk to be accessed in the directory. Each feed contains all of the information necessary to access all of the chunk columns from the file system. Also shown is the metadata, a single vector-shaped tensor of length 2 containing the ID and the arity. We can determine from the arity component of the metadata that this dataset has 2236 total feeds, one for each chunk in the input dataset.

4.3 Components

PTF expresses application logic as a linear sequence of transformations. This *pipeline* abstraction is well suited to bioinformatics workflows at two different levels. At a high level, a workflow is typically a linear sequence of application invocations for each dataset (*e.g.*, alignment, then sorting, then post-processing, *etc.*). The operation performed by each application can also be decomposed into a linear sequence of steps; each intra-application pipeline typically contains a core application logic step (*e.g.*, alignment) preceded by an I/O and preprocessing step (*e.g.*, reading and decompressing an AGD chunk from a file system) and followed by a final post-processing and I/O step (*e.g.*, compressing a new AGD chunk file and writing it back to the file system).

PTF's pipelines are composed of successive *stages* synchronized via connecting *gates*. Each stage is a TensorFlow graph that statelessly transforms feeds. The gates synchronize and coordinate the concurrent execution of different feeds throughout the pipeline; they do this by interpreting metadata attached to each feed in order to provide the desired semantics. In this section, we discuss these two components, stages and gates, and how they fit into the TensorFlow ecosystem.

4.3.1 Stages

The core of each stage is a TensorFlow graph that encapsulates some subcomponent of the overall pipeline logic. This graph may be constructed with any node available in the TensorFlow library and can be of arbitrary complexity. stages provide a mechanism to apply the strict dataflow semantics of TensorFlow (as outlined in §2.4) to a subcomponent of the dataflow graph, while isolating and decoupling the execution of this graph from the rest of the application pipeline.

The graph within a stage is differentiated from regular TensorFlow graphs due to the addition of two nodes to interact with the adjacent gates in the pipeline. Instead of a user manually feeding the TensorFlow graph, a dequeue node delivers a feed from the upstream gate. After the TensorFlow runtime processes the feed on the stage's graph (according to the standard TensorFlow dataflow execution rules), the corresponding enqueue node in the graph inserts the resulting feed into the subsequent gate. Using these nodes eliminates the overhead of feeding a value into TensorFlow runtime or retrieving a resulting value, avoiding unnecessary

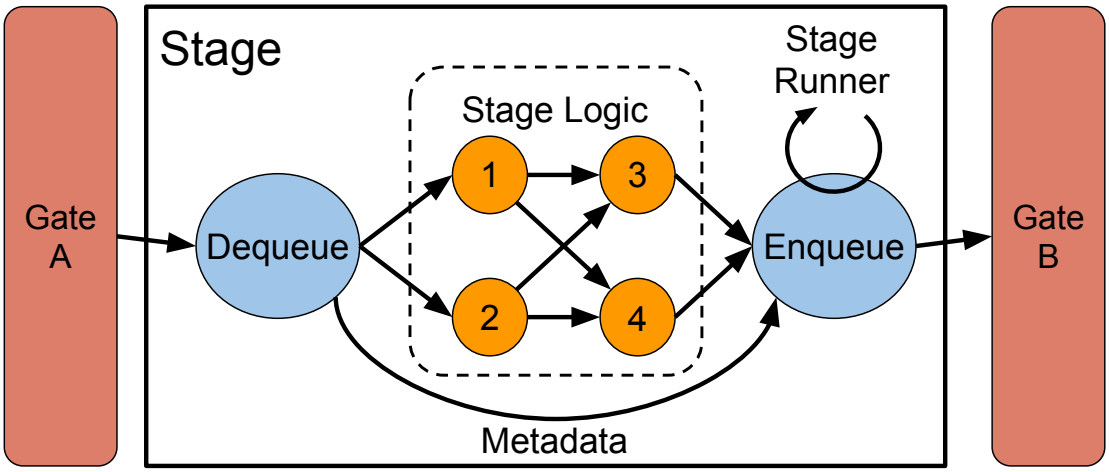


Figure 4.2 – The components of a stage: the TensorFlow graph containing the stage logic, the enqueue and dequeue nodes to communicate with the adjacent gates, and the metadata.

data copying [103] that would be required if an external framework were distributing data items (*e.g.*, via the Python API).

The last component of a stage is a *stage runner*: a Python thread that repeatedly executes the stage's graph. The TensorFlow graph contained in each stage operates the same as any other: it requires a user thread (*i.e.*, outside the runtime) to request the value of one or more tensors in the graph in order for the TensorFlow runtime to execute the corresponding nodes to produce that value. Instead of requesting a tensor value from the TensorFlow graph, the stage runner requests the result of the enqueue node. This is a null value, but the side effect of requesting this value triggers the operation we desire: the TensorFlow runtime requests a value from the upstream dequeue node, propagates the tensors throughout the graph according to the standard TensorFlow dataflow execution rules, and triggers the enqueue node to insert the value into the downstream gate. The logic for stage runners is simple (an infinite loop to execute the stage's graph until an exception occurs) and is an included component in PTF.

Stages do not read or modify the metadata as they process feeds. Although the metadata is represented as an additional value in the feed, stage's typically pass the metadata through unmodified to the enqueue node. This stateless process of feeds by each stage enables each stage to be scaled out without any additional coordination.

Figure 4.2 shows all of the components of a stage. The dequeue node gives provides the core *Stage Logic* (*i.e.*, the stage's TensorFlow graph) with a feed of 2 tensors, giving one to nodes 1 and 2. The result of this stage is the output of nodes 3 and 4, which pass their values together with the metadata to the enqueue node to pass to the next gate (Gate B). Note that the metadata bypasses the core logic in the stage's graph; this is typical of most stages, as they do not modify the metadata.

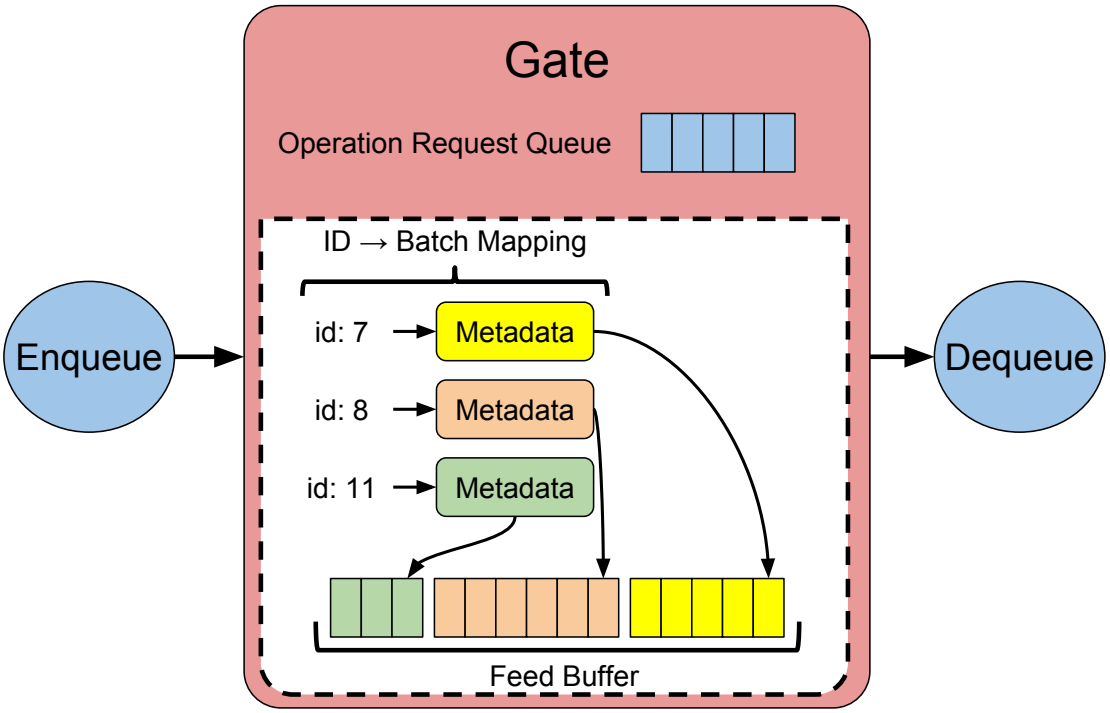


Figure 4.3 – The internal details of a gate.

4.3.2 Gates

Gates buffer feeds between successive stages so that each stage can concurrently process a different feed. In order to decouple the data dependency between adjacent stages, each gate stores feeds in an internal feed buffer between the stages. When the downstream stage is ready to dequeue a new feed, the gate selects one from its buffer and delivers it to the downstream stage. This decoupling of data dependencies between stages enables a PTF application to process multiple feeds simultaneously, in contrast to the single-feed limitation of a monolithic TensorFlow graph.

Gates interact with stages with a collection of TensorFlow nodes that, when inserted into a stage's graph, enable a stage to manipulate a gate's state. These nodes adopt the terminology of typical queue data structures:

- Dequeue nodes take a reference to a gate as input and produce a feed as output, when one is available within the gate. Dequeue nodes serve as a data *source* within each stage's graph. The operation of these nodes blocks until serviced by the gate. The result of this operation both (a) emits a feed for use by the stage's graph and (b) removes this feed from the gate.

- Enqueue nodes take, as input, a reference to a gate and a feed and insert the feed into the gate. These nodes serve as a data *sink* within each stage's graph, terminating the

dataflow propagation of values with only a side effect; these nodes do not produce any value.[1] Upon successful completion of the operation, both of following occur: (a) the feed is stored in the gate and (b) the feed is no longer available to the stage that enqueued it. The stage runner of each stage repeatedly executes this node to drive computation through the stage's TensorFlow graph.

Gates service operation requests from enqueue and dequeue nodes in a first-come, first-serve (FCFS) order. Each gate will have at least two operations (one each of enqueue and dequeue) that await processing by the gate. As each operation will modify the state of the gate's feed buffer, the gate uses a queue of pending operations and a lock to mediate access to the gate. These operations are serviced in the order in which they are enqueued into this queue. However, not all operations may be able to succeed. For example, if a gate has no feeds in its buffer, then no dequeue operation may be satisfied; in this case, the gate will still attempt any enqueue operations.

Figure 4.3 shows the details of a gate. At the core of each gate is the feed buffer and the mapping of batch ID (in this case, IDs 7, 8, and 11) to a batch metadata data structure that tracks the progress of the batch's feeds and contains a pointer to the region in the gate's feed buffer where the associated feeds are buffered. The pending queue of operations controls the order in which gate operations (*i.e.*, enqueue and dequeue operations) are processed.

Opening and Closing Batches

Each gate interprets the metadata of each feed in order to apply PTF's semantics. Specifically, as each feed is enqueued into a gate, the gate uses the metadata to track the progress of the corresponding batch as it passes through the gate. The metadata is sufficient to identify a feed by associating it to a new or existing batch and, with the help of accounting structures for each batch, to know how many feeds the gate still expects for a given batch.

A gate may *open* a new batch when it receives a feed from a new batch. If, after examining the ID component of the metadata, a gate determines that a feed is associated with a new batch, it allocates new space in its feed buffer for that batch and an accounting structure to track the batch's progress. This accounting structure is associated with the ID of each batch, *i.e.*, as a key into a mapping within the gate. It contains (a) the batch arity, (b) the number of feeds that have been dequeued by a downstream stage, and (c) the number of feeds available in the gate's buffer. These three numbers are sufficient to determine how many feeds a gate must wait for from the upstream portion of the pipeline.

A gate *closes* a batch when it determines that it has dequeued the last feed or feeds associated with a batch. The gate can determine when it has exhausted a batch (*i.e.*, there are no feeds for

[1] Nodes with no outputs in TensorFlow technically produce a null tensor value as output as a return value to client code that consumes the API (*e.g.*, through Python). In a TensorFlow graph, this output may not serve as input for further computation.

the batch in its buffer, nor will any arrive in the future) by examining the accounting structure. If the accounting structure associated with this batch shows that the number of feeds of that batch that have been dequeued from the gate is equal to the arity, the gate can be certain that it has dequeued all feeds for the batch. It deletes the accounting structure for the batch as well as the space in the feed buffer in the gate associated with this batch.

4.3.3 Batch Aggregation

PTF supports *aggregate* operations on gates that produce and consume *aggregate feeds* consisting of multiple single feeds. An aggregate feed is single feed that consists of a group of individual feeds. Individual feeds in each aggregate feed are associated with the same batch. Operations that produce an aggregate assemble the aggregate feed from a single batch, using the metadata and internal accounting structure to separate each feed in the feed buffer. Each tensor of the original (*i.e.*, individual) feed type is present in the aggregate feed, but an extra dimension is added (*e.g.*, a vector-shaped tensor is expanded to a matrix-shaped one); each aggregate feed tensor has the group of values from the set of individual feeds striped across this additional dimension.

Aggregate feed operations are used in PTF to support several use cases. A stage may use optimized nodes in its TensorFlow graph that operate on multiple feeds in a single invocation, producing an aggregate result with higher throughput. The creation of aggregate feeds can be used to serve as barrier (*e.g.*, between two phases of a pipeline), as a gate must wait for all constituent feeds in a batch to arrive at the gate before producing the aggregate feed. Aggregate feeds are also used to support PTF's scale-out architecture, discussed in §4.5.

The aggregate dequeue operation creates aggregate feeds in a PTF application pipeline. In addition to the gate reference, the aggregate dequeue node requires the size argument (S) specifying the requested aggregate size (*i.e.*, number of feeds in each aggregate feed). It waits until either (a) S feeds for a given batch are available in the gate's feed buffer to produce an aggregate feed or (b) there are fewer than S feeds available for a batch, but no more feeds will be enqueued for the batch, in which case an aggregate feed is constructed from the available feeds. This latter case occurs when the arity of a batch is not an even multiple of S. This operation transforms the arity of the associated metadata: the original arity A is modified to be $\lceil A \div S \rceil$; this arity reduction is necessary because the downstream stage may only produce one feed per aggregate feed it consumes due to the dataflow rules of TensorFlow.

Aggregate feed operations are used to support special ingress and egress gates used at the beginning and end of each PTF pipeline. These gates both support normal gate operations, but include special operations on aggregate feeds to enable PTF to avoid exposing the metadata details to users submitting requests to it.

- Ingress gates provide a batch enqueue operation, which takes an aggregate feed consisting of all feeds in a single batch as input, assigns proper metadata to the request[2],

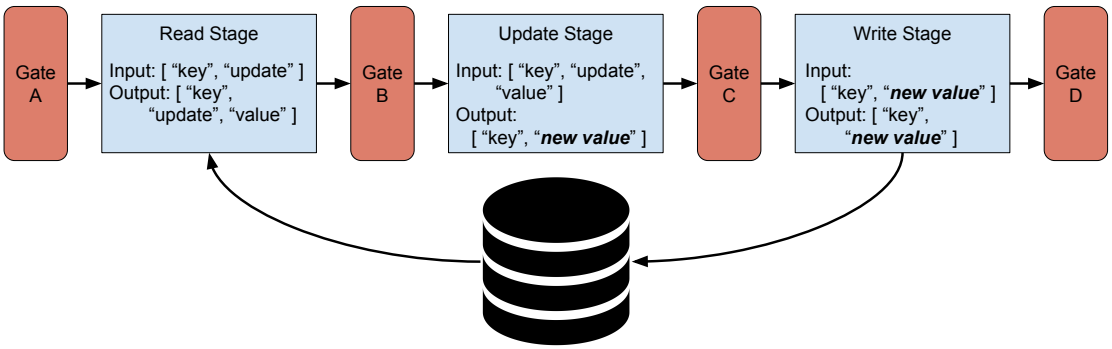


Figure 4.4 – A PTF pipeline representing an update operation.

and returns the ID of the metadata associated with the new batch. The user uses this operation to submit an entire batch of input to the pipeline at once and receive the ID as an opaque value to retrieve the resulting batch from the end of the pipeline.

- Egress gates provide a batch dequeue operation, which takes an ID (from a corresponding batch enqueue operation on the pipeline's ingress gate), waits for the batch corresponding to the ID is available in the gate, then returns all feeds of the batch as a single aggregate feed. This operation terminates the application pipeline; user code inspects and manipulates the resulting aggregate feed after it is returned from the TensorFlow API (*i.e.*, as a Python value) to form a final result for the user request.

4.4 Pipelines

Figure 4.4 shows an example of a how a simple update operation for a storage system can be composed as a PTF pipeline. This update operation takes a batch where each feed contains (a) a key to update in the storage system and (b) an update value to combine with the existing value to produce a result value. The pipeline has three operations:

- The read stage reads a single key from the storage system and enqueues the key, the update value, and the value read into the gate. the resulting feed type is comprised of the key, update type, and the value type, *e.g.*, as three string tensors in this example.
- The update stage combines the value and the update item in each feed into a *new value*; it enqueues this new value and the key into the next gate (C). The feed type now the key type and the value type (each string tensors in this example).
- Finally, the write stage writes the new value back to the storage system and enqueues the feed into the final gate (D). The feed is unmodified through this stage, as both of its members are passed as output.

[2] Each ingress gate checks the size of each aggregate feed component in dimension 0 to ensure they are all equal, uses this dimension value as the intended batch arity, and assigns a unique ID to the batch based on an incrementing integer within the gate itself.

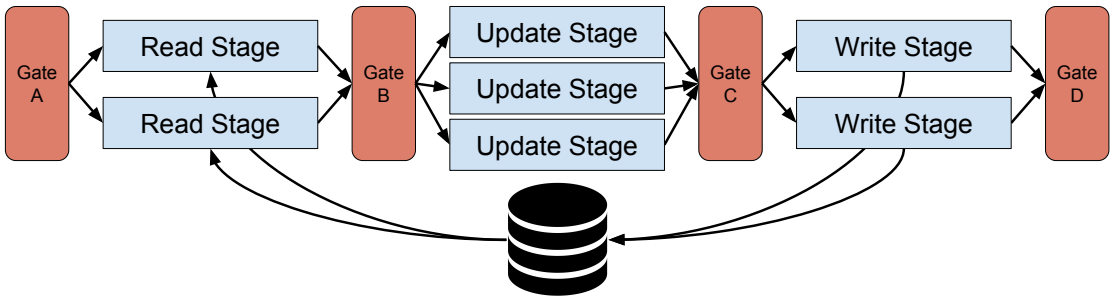


Figure 4.5 – A simple PTF pipeline with stage scaling to increase throughput.

The total output of this pipeline is a key-value pair consisting of the values written back as a result of this operation. In each stage, the metadata is passed between gates unmodified; it is not shown for the sake of clarity.

PTF pipelines enable developers to write multi-threaded applications that scale across multiple machines by composing small units of application logic to execute concurrently. The simple pipeline shown in Figure 4.4 requires (a) a library containing the TensorFlow nodes to read and write data from the storage system in addition to nodes for custom logic in the update stage and (b) a composition of these nodes into TensorFlow graphs, one for each of the three stages. After this logical composition is complete, this application pipeline benefits from the multi-threaded and scale-out runtime from PTF; the pipeline does not have to coordinate work distribution between the stages or track the progress of concurrent batches. This liberates developers from the arduous process of writing these components that are typically necessary for constructing a parallel application that scales across machines; he or she may focus on the logic of processing a single feed through a single stage, delegating the more difficult components related to pipelined execution to PTF.

4.4.1 Scaling Up Stages

Stage replication enables the throughput of any stage to increase by increasing parallelism through additional concurrency. By design, stages are stateless with regard to the feeds they process because they do not interpret the metadata. This provides an opportunity to scale up the throughput of a stage by simply duplicating the stage. Specifically, this involves (a) replicating the stage's TensorFlow graph (one per replica), including unique enqueue and dequeue nodes per stage, and (b) adding a new stage runner to execute the graph in each stage. Stages may still share resources between them, such as the executors and buffer pools described in chapter 5; they may not share the results of any dequeue operation.

Stage replication increases the degree of parallel operation available within a PTF pipeline for the TensorFlow runtime to execute, taking full advantage of hardware resources. With additional stages, the number of nodes in the overall PTF graph grows. Because stages do not share data dependencies between each other and each have their own stage runner, the

TensorFlow runtime can execute the graphs in each stage in parallel. This increased parallelism has several advantages for PTF applications:

- Increased CPU utilization: the added parallelism from scaling up a stage means that the TensorFlow runtime has a greater number of nodes it may execute at any given time. This increases CPU utilization without any additional modification to the logic of a PTF pipeline.
- Increased throughput for high-latency stages: if the latency for a stage is high, either due to a computationally expensive operation or awaiting I/O, scaling up this stage enables the throughput to be increased.
- Decreased variability: added parallelism due to scaling up a stage prevents a feed with high processing latency on that stage (*e.g.*, if the stage performs a network operation with high variability) from preventing progress of other feeds with lower latency (*i.e.*, head-of-line blocking). High-latency feeds may be surpassed by other feeds to mitigate this effect.

Figure 4.5 shows an example of the simple pipeline in Figure 4.4 with stage replication. Each stage is replicated to a different degree (two stage replicas for each of read and write, and three replicas for the update stage). Unlike the serial version of this pipeline shown in Figure 4.4, the parallel version shown in Figure 4.5 may process any feed on any subset of the stage replicas (one for each stage). Any given stage replica may process any subset of the feeds from a given batch, including none at all, and in any order; this is due to the FCFS order in which gates process requests and variability in the latency of a given stage (*e.g.*, a read stage that is delayed due to a slow read from the storage system). Due to the stateless nature of stages, this variability in ordering does not affect the outcome of the pipeline's computation for any given feed or batch.

Beyond a certain degree of stage replication, the TensorFlow runtime limits the throughput of a PTF pipeline. Recall from §2.4 that the TensorFlow runtime executes kernels (the executable constructs in the runtime corresponding to nodes in the graph) on a pool of threads (by default, one thread per CPU thread). Each of these threads executes nodes from a list of ready-to-execute nodes. Once the level of stage parallelism exceeds a saturation point such that this list is never empty (*i.e.*, all threads are always busy executing a kernel), additional scaling does not increase throughput. Beyond this point, gates determine which stages are executed based on its FCFS policy and data availability. We discuss application tuning in chapter 5.

4.4.2 Reordering

Gates emit feeds in a first-in, first-out (FIFO) order with respect to the feeds of a single batch; feeds between batches may be reordered. Note that this does not prevent reordering when

a pipeline with stage replication; if two feeds from the same batch have different processing latencies for a given parallel stage and are submitted simultaneously, the lower latency feed may surpass the other.

Gates attempt to emit feeds based on the order in which batches were opened to avoid reordering batches, where possible. When there are multiple open batches from which a gate could satisfy a dequeue operation, it chooses the operation that was opened earlier. In the case that some reordering of feeds occurred during a parallel stage (*i.e.*, reordering between feeds of different batches), this ordering preference tends to resolve reorderings earlier in the pipeline. However, it does not prohibit a complete reordering when possible, *e.g.*, such that a subsequent batch may surpass a more expensive earlier batch in the pipeline.

4.4.3 Bounding Resources

Gates require a bounding mechanism to limit resource utilization. Even in a serial pipeline, any difference in service time (*i.e.*, latency for a stage to process a feed) between successive stages in a pipeline can cause a resource explosion without a bounding mechanism. For example, if the `Read` stage in Figure 4.5 is faster than the `Update` stage, the gate between them needs a mechanism to avoid enqueuing an infinite number of feeds in a long-running version of this application.

Feed Buffer Bounding

Gates may locally limit resource utilization by bounding the size of their feed buffer when the following stage uses a dequeue operation (*i.e.*, not an aggregate dequeue). In this common case, a gate may place an upper bound on the number of feeds it will buffer. If an enqueue operation attempts to add a feed to a full gate, that enqueue operation must wait for a dequeue operation to succeed so that space becomes available.

In this locally limited case, PTF uses standard TensorFlow queues in place of gates to perform resource bounding. Without an aggregate dequeue or other such operation on a gate (*i.e.*, only standard enqueue and dequeue operations), its operation is identical to a TensorFlow queue; no differentiation between feeds based on their metadata is required to perform the requested operations. In this case, the gate is replaced with a TensorFlow queue, as TensorFlow's queues have a feed-based capacity bound that enforces the same semantics (*i.e.*, blocking until spare capacity becomes available). The feeds, including the metadata, are enqueued into and dequeued from the TensorFlow queue similar to a gate.

Credit Linking

Due to the possibility of reordering from stage parallelism, an additional resource bounding mechanism is required. In the case where stage uses an aggregate dequeue operation to

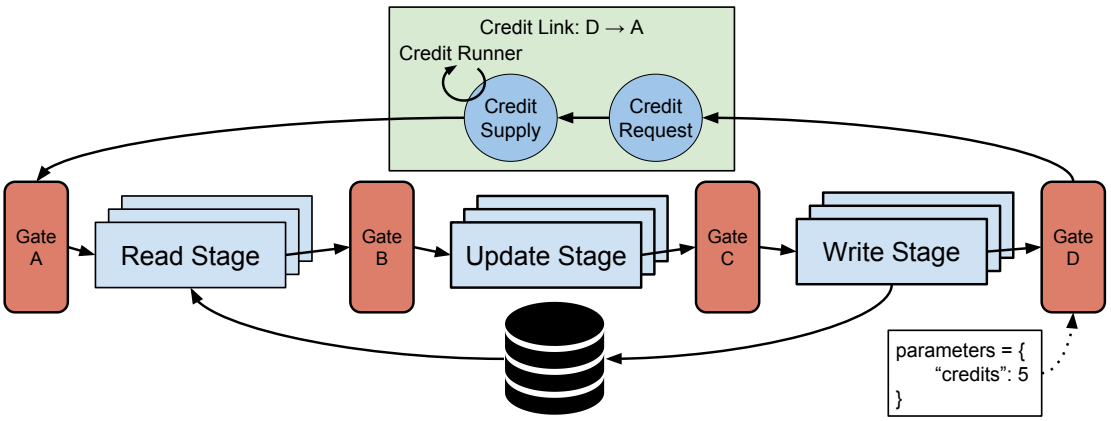


Figure 4.6 – A parallel update pipeline with a credit link between the first and last gates.

request an aggregate feed from its upstream gate, that gate may not limit the size of its feed buffer in the general case. This is because under certain configurations (*e.g.*, based on aggregate size and reorderings), it is possible for a deadlock to occur: if the aggregate dequeue operation requests an aggregate feed of B, the feed buffer is full, and all of the open batches have less than B feeds in the buffer, then neither enqueue nor dequeue operations may proceed, blocking the progress of this gate and therefore the pipeline.

PTF provides a *credit link* mechanism to bound resources based on the number of open batches that may exist between any pair of gates. A credit link originates at a downstream gate D and provides *credits* to an upstream gate U. For each credit that D sends to U, U may open one batch and begin sending feeds for that batch. The maximum number of open batches between U and D is specified based on the number of credits D may issue, a parameter set upon constructing the application.

The architecture of a credit link is similar to that of a stage; instead of processing feeds from upstream to downstream, it provides credits in the reverse direction. Each credit link is a TensorFlow graph composed of three components:

- a Credit Request node, which takes the downstream gate as a parameter and emits a number of credits (strictly greater than zero) when they become available in the gate
- a Credit Deposit node, which takes as input the upstream gate and the credits (*i.e.*, the output of the Credit Request node) and adds them to the credit count of the upstream gate
- a Credit Runner thread, which repeatedly executes the credit link graph until an exception is raised by the runtime

A gate may only be linked to at most two gates (one upstream to which it provides credits, and another downstream from which it receives credits).

Figure 4.6 shows an example of a pipeline that combines stage replication with bounded resource usage via credit linking. This pipeline is configured to allow at most 5 batches at any given time by configuring gate D (the final gate in the pipeline) to have 5 credits in its construction parameters. The credit link graph contains the two nodes of the graph necessary to transfer credits from gate D to gate A as well as the credit runner thread to drive the graph Upon startup, gate D initially issues all 5 credits to gate A. Subsequent credits are issued when gate D closes batches.

4.4.4 Lifecycle

A PTF application begins its lifecycle with a user assembling a sequence of stages and gates. The user first creates an ingress gate ($G1$) to mediate access to the pipeline and a second gate ($G2$) to receive the results of the first stage. The user then creates the first stage by doing the following:

1. Create a dequeue operation (single or aggregate) on gate $G1$ based on the stage's requirements.
2. Use nodes in the TensorFlow API to assemble the stage's TensorFlow graph.
3. Create an enqueue operation to insert the stage's result into gate $G2$.
4. Create a Stage Runner (using the PTF library, embedded within the TensorFlow runtime) to execute the enqueue operation indefinitely.

These steps are repeated for each stage replica the user creates for this stage. This process repeats for each stage in the pipeline, creating additional gates and stages in sequence.

The complete state of an assembled pipeline is represented as a large TensorFlow graph. Although the structure of a PTF pipeline is dissimilar to a typical TensorFlow graph, the PTF pipeline can still use TensorFlow's graph description and serialization capabilities. The PTF pipeline contains the following elements:

- A set of independent graphs, representing the TensorFlow graphs for each stage. These stages are connected to two gates, one upstream and one downstream, via their enqueue and dequeue operations.
- A set of gates. Each gate is identified by a unique name in the overall graph as well as a set of attributes specifying its operation, *e.g.*, number of credits to issue.
- A set of Stage runners. These are not strictly a component of the TensorFlow graph; they are threads to be run on the instantiated TensorFlow graph using the Python API. Using extensions to TensorFlow's graphs serialization capability, we can store the attributes necessary to describe each stage runner within a serialized TensorFlow graph

description. The attributes to store for each Stage runner are the name of the enqueue operation to run and the name of a `close` operation to execute if the Stage runner encounters an error (typically closes the gate and triggers the application to shut down).

- A set of credit links, consisting of the small graphs to propagate credits as well as the corresponding credit runners. Similar to stage runners, credit runners are a thread that interacts with the TensorFlow graph using the API; they are not strictly part of the TensorFlow graph. Credit runners are stored in the TensorFlow graph description in a manner similar to that of stage runners.

All of these elements may be serialized to a saved file using existing functionality in TensorFlow or executed by the TensorFlow runtime directly after assembly.

A PTF pipeline is executed using the standard TensorFlow runtime and the Python API. After assembling the PTF pipeline or loading it from a serialized description, execution begins in a Python thread with the following steps:

1. Create a TensorFlow session, which instantiates all nodes and shared resources in the TensorFlow runtime.
2. Create a new Python thread to execute each of the stage runners and credit runners.
3. Await users to connect and process their requests on the pipeline.
4. If an error occurs, cancel all pending user requests and exit.

In this scenario, the Python application that launches the application serves as a proxy for user connections. This can be accomplished via an embedded library to serve HTTP requests from Python; parallel Python threads must be used to process requests in parallel on the instantiated application.

4.5 Cluster Scale-Out

The TensorFlow runtime enables PTF to scale across multiple machines. One of the core goals for PTF, outlined in §4.1, is to be able to scale computation across a cluster of machines. Each node in the TensorFlow graph (and, by extension, each stage in the PTF pipeline) can be placed on a different logical device, which the TensorFlow runtime will instantiate on different processes (possibly on physical machines).

A naive approach to scaling a pipeline by employing stage scaling to replicate stages across multiple machines is limited by the scaling capability of gates. In order for a gate to provide the correct semantics for PTF, it must process all feeds of every batch. This is because gates use the arity in the feed metadata together with the number of feeds they each observe to apply the correct semantics, *e.g.*, for an aggregate dequeue operation or for bounding resources with a

credit link; gates must be able to know when they can close the batches based on the processing of feeds. The gates therefore impose a serial overhead for this naive approach because (a) gates must protect their in-memory data structures using a lock, serializing accesses by all operations and (b) result feeds from stages in different processes must be sent to the single machine where each gate lives, imposing a communication overhead.

This naive stage scaling approach may not be possible to use when applications use the shared object feature in the TensorFlow runtime. If an application chooses to use custom shared resource objects within the TensorFlow runtime to pass data from an upstream stage to a downstream stage, those objects would only exist in the memory of a single machine. If this naive stage scaling approach were to be used, references to these shared objects enqueued into a gate may be distributed by the gate to a different machine for the following stage; this is due to the FCFS order in which gates process operations. When the machine attempts to look up the shared object reference it receives from the gate, the operation will fail and the PTF application will shut down.

PTF scales out pipelines across multiple machines by using a two-level *nesting* hierarchy of *local* and *global* pipelines. A *local* pipeline consists of gates and stages that are placed in a single process (typically one process per machine). A *global* pipeline consists of a sequence of local pipelines separated by global gates. Local pipelines operate on *partitions* (*i.e.*, subsets of feeds from the batch, with modified metadata) that are distributed by the global gates, enabling a coarser granularity of work distribution from the global gates to the local pipelines. Each local pipeline processes its partition as an independent batch, passing the result back to the subsequent global gate to reassemble the aggregate result.

The hierarchical architecture of scale-out PTF decouples the granularity of gate operations between local and global pipelines to enable better scaling. Each gate, local or global, must process every feed from each batch that it receives in order to ensure correct operation (*e.g.*, for resource bounding and aggregate dequeue operations). The hierarchical structure of PTF for scaling out across multiple machines enables two important things:

- Local vs. global operation granularity: by using operations on partitions at the global pipeline, the total number of operations per batch is reduced based on the size of the partition. This reduces the load (and lock contention) on the global gates and the network congestion between different machines in the cluster.

- Local pipeline replication: each local pipeline operates on a partition of the global batch as an independent local batch. This decoupling of partitions from the global batch, performed by the gates in the global pipeline, enables local pipelines to be replicated to increase the throughput for the operation performed by each local pipeline. Because each local pipeline operates on partitions, the gates in a local pipeline do not need to process every feed from the global batch; they only need to process the feeds within the partition. This decoupling enables the global pipeline to distribute a subset of a batch's

feeds to each local pipeline replica while enabling gates in both the global and local pipelines to correctly apply PTF's semantics.

Using a pipeline hierarchy to distribute work across multiple machines enables PTF to support scale-out processing within the TensorFlow runtime. In contrast to other approaches that combine TensorFlow with an external framework to coordinate distributed processing [130], this hierarchy enables PTF to support scale-out applications within the TensorFlow runtime. This has practical advantages for both PTF's design and its users:

- Small addition of functionality to PTF: only a small amount of additional code is required in PTF to support this hierarchical architecture. Specifically, only gate operations that support partitions must be added; gates and stages are constructed in the same manner.
- TensorFlow distributed runtime: PTF can reuse the robust distributed runtime of TensorFlow without any modification. TensorFlow automatically inserts communication mechanisms (*e.g.*, send and receive nodes to transfer tensor values across the network) into a graph based on the graph architecture and physical topology. The TensorFlow runtime then executes graphs across multiple machines in the same manner as a local-only operation.
- Similar construction for local and distributed PTF applications: an application that constructs and executes a pipeline in a similar manner regardless of whether it is executed on a single machine or a cluster of machines. The only additional design required in order to construct a distributed PTF application is to separate sections of the logical pipeline into local pipelines, designate a logical device on which each local pipeline should be placed, and insert additional global gates between the local pipelines.
- Single serialized description of the PTF application graph: by describing the entire application within the TensorFlow framework, the description of the entire PTF application can be serialized to a single graph description similar to a local-only PTF pipeline. This enables the application to be saved and restored in support of data provenance and re-executing an application without requiring de novo construction of the application logic every time.

PTF uses constructs within the TensorFlow runtime to process data (*i.e.*, feeds) while simultaneously performing the control plane functionality for a scale-out framework.

4.5.1 Pipeline Hierarchies

The base of the pipeline hierarchy is *local* pipelines. These are the pipelines described in §4.4 except that they are constrained to disallow the naive scaling described in this section (§4.5). Specifically, PTF constrains each local pipeline in the following ways:

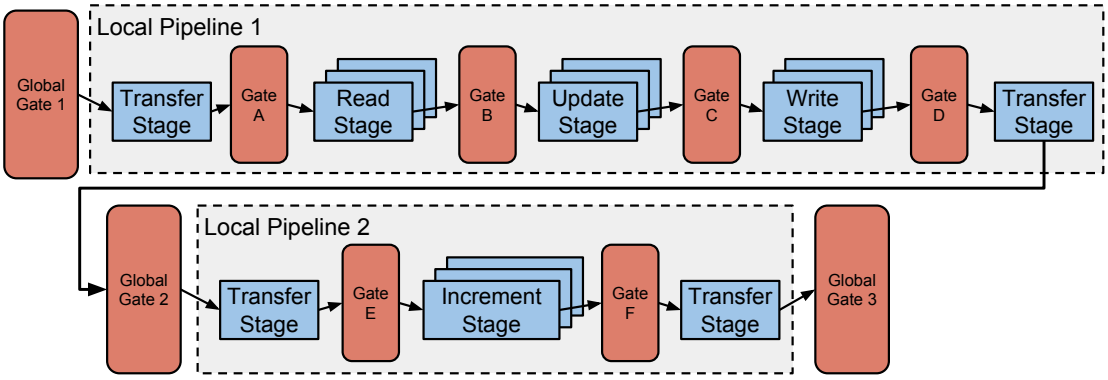


Figure 4.7 – A hierarchical pipeline consisting of two local pipelines.

- All stages and gates for the pipeline must exist in a single process (*i.e.*, a single logical device in the TensorFlow graph), typically on a single machine when provisioning a cluster.[3]

- It may not use shared resources from any other process.

These limitations avoid any of the correctness issues from using shared resources with naive scaling as well as minimize communication overhead by placing all pipeline components within a single process.

Local pipelines differ from standard pipelines due to the use of special gate operations for processing batch partitions. These gate operations, at the beginning and end of each local pipeline, enable the local pipeline to process each partition of the global batch as an independent *local-only* batch; these operations are applied at the beginning of a pipeline to enqueue the partition received from the upstream global gate and at the end of the local pipeline to deliver the partition results, as an aggregate feed, to the next global gate.

- The *partition enqueue* operation takes a partition from a global gate and inserts it into the first gate of the local pipeline as an entire batch. This operation is similar to the batch enqueue operation of ingress gates discussed in §4.3.3, except that the metadata is assigned by the global gate that created the partition; the arguments to this operation are the partition, as an aggregate feed, and the metadata from the global gate.

- The *partition dequeue* operation terminates a partition on a local pipeline. This operation is used on the last gate in the local pipeline to await the results of an entire partition and emit a single aggregate feed of all results to the next global gate. The operation is similar to a batch dequeue operation of egress gates discussed in §4.3.3, except that the metadata is output in addition to the aggregate feed.

[3] A local pipeline may take value tensor inputs from a remote machine, *e.g.*, configuration parameters.

- A pipeline may choose to forgo a partition dequeue operation when terminating a pipeline; in this case, a regular dequeue operation may be used on the last gate in the local pipeline to send individual feeds from the partition to the next global gate. A pipeline may choose this case if the desired performance is to lower the latency of individual feeds of the partition; this trades lower latency for an increase in communication overhead between the local pipeline and the downstream global gate, as each feed is sent individually instead of a single aggregate result.

Global pipelines are composed of a sequence of local pipelines separated by global gates. The global gates create partition from batches to send to local pipelines and *join* the partition results from local pipelines in order to reconstitute the resulting batch:

- Global gates create partitions of a batch by applying an aggregate dequeue operation to the original batch and modifying the metadata.
- Global gates *join* resulting partitions, sent to these gates by a local pipeline's partition dequeue operation, by interpreting the modified metadata and enqueuing the aggregate feed of results.
- In the case where the upstream local pipeline does not use a partition dequeue operation to send the entire partition at once, the global gate simply enqueues the resulting feed after interpreting the modified metadata.

Transfer stages link global and local gates to transfer partitions between them. Each transfer stage performs no calculation; the TensorFlow graph contains only two gate operations to either a) send a partition from the global gate to the first gate of a local pipeline or b) send a partition from the last gate in a local pipeline to the next gate in the global pipeline. These stages are a construct to coordinate the transfer of partitions within the TensorFlow distributed runtime itself.

Global pipelines may bound resource usage with credit links. Using the same credit linking mechanism described in §4.4.3, global gates may bound the number of open batches between any two gates. Credit linking in the global pipeline prevents the overuse of global resource shared between local pipelines, such as the utilization of storage space on a shared storage system. Local pipelines may also employ the resource bounding mechanisms from §4.4.3 to bound local resources, such as memory. The only limitation is that no credit link may be used between a local and a global gate; the semantics of batch progress in the global pipeline versus partition progress in the local pipeline preclude this.

Figure 4.7 shows a global pipeline consisting of two local pipelines which interact with a storage system. The first local pipeline (Local Pipeline 1) is identical to the simple update pipeline in Figure 4.4; it reads, updates, and subsequently writes the value back to the storage system. The output of this phase is the key that was updated, inserted into Global Gate 2. The

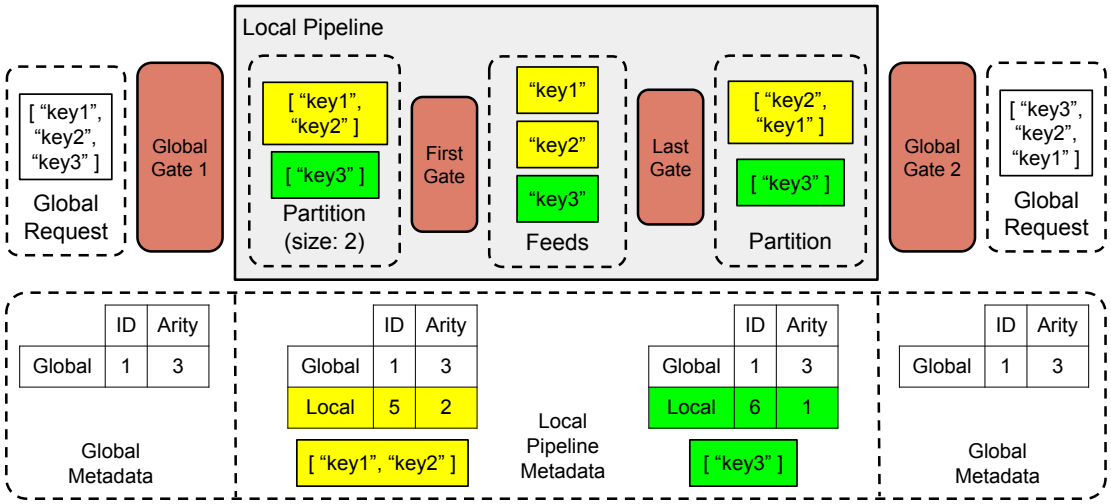


Figure 4.8 – Details of the partition metadata and how the batch is partitioned by the global pipeline for processing in the local pipelines.

second local pipeline (Local Pipeline 2) takes this key and performs an additional operation with it: incrementing a counter in the storage system for the number of updates performed to that key. Stages within each local pipeline are replicated to tune the stage to the hardware resources of each local pipeline. The first and last stage in each local pipeline is a transfer stage, which move a partition from the global pipeline to a local pipeline and vice versa. These will be elided in future diagrams for clarity.

4.5.2 Batch Partitioning

PTF creates metadata for each partition by *extending* the metadata of the original batch: the extended metadata contains the original metadata for the batch (*i.e.*, at the global level) as well as the metadata specific to each batch partition. Each local pipeline interprets the partition metadata to correctly track the progress of the partition. The global pipeline assigns the metadata for each partition when each global gate creates and sends a partition to a local pipeline. The subsequent global gate interprets the original metadata to track the batch's progress across all partitions.

The partition metadata contains a numerical ID unique to the partition and an arity based on the number of elements in the partition. Each of the partition dequeue operations on a global gate specifies a desired partition size (P), *i.e.*, the number of feeds from the global batch to create a partition. The global gate creates an aggregate feed of at most P feeds assigns a new arity based on the size of the aggregate feed.[4]This arity is combined with a unique numerical ID assigned by the global gate and appended to the existing metadata for the global batch to create the extended metadata. Both of the arity fields of the metadata must be modified if a

gate local pipeline uses an aggregate dequeue operation; both arity fields are modified based on the calculation described in §4.3.2.

Extended metadata is represented by a single numerical tensor in the TensorFlow graph, but with additional dimension. Whereas typical batch metadata is can be represented by a two-element vector-shaped tensor of integers (one element for the arity and one for the ID), extended metadata is represented by a 2×2 integer tensor. Slicing the tensor along dimension 0 yields two metadata vectors; the first of these vectors (index 0) is used for the global batch's metadata and the second is used for the partition metadata. In order to maintain coherent operations between gates in the global pipeline and those in local pipelines, all gates process metadata that is $N \times 2$ in dimension; N is 2 for a local pipeline and is 1 for the global pipeline. All gates interpret the highest-index (*i.e.*, *innermost*) slice along dimension 0 to track the progress of the batch or partition.

Figure 4.8 shows an example of how partition metadata is created and used through a local pipeline. The local pipeline used in this example is the simple update pipeline from Figure 4.7 (Local Pipeline 1) with the structural details elided. The example global batch has an arity of 3. Global Gate 1 partitions the global batch into 2 partitions of sizes 2 and 1. The First Gate in the Local Pipeline splits up each partition into its constituent feeds; the Last Gate reassembles these partitions before transferring them to Global Gate 2, which reassembles the partitions into the original batch. This figure shows the metadata tensors used by the global and local pipelines. The Global Gate 1 creates the metadata for each partition. Each gate in the local pipeline attaches this partition metadata to each feed as it moves through the pipeline, using only the local metadata to perform PTF functionality. The Last Gate in the local pipeline attaches the partition metadata to each resulting partition as it sends it to Global Gate 2, which strips the local metadata when reassembling the original batch.

4.5.3 Scaling Out Local Pipelines

PTF enables increased throughput by replicating local pipelines across multiple machines. Similar to the manner in which stages scale up via replication to increase resource utilization of a single machine running a local pipeline, local pipelines can be replicated across multiple machines to increase the resource utilization of a cluster of machines. Global gates mediate concurrent gate operations from replicated local pipelines on an FCFS basis. Each replica of a local pipeline operates independent from other replicas[5], based on the availability of batch partitions from the upstream global gate.

Figure 4.9 shows a scaled-out version of the pipeline from Figure 4.7. Local Pipeline 1 is replicated 3× and Local Pipeline 2 is replicated 2×. Each local pipeline operates independently on partitions as coordinated by the global gates. Each local pipeline uses a credit link to control

[4]The total number of feeds present in the aggregate feed may be less than P if P is not a divisor of the arity of the global batch. This follows the same semantics as an aggregate dequeue operation on a gate.

[5]Local pipelines may take constant tensor values as input from shared components of the PTF graph.

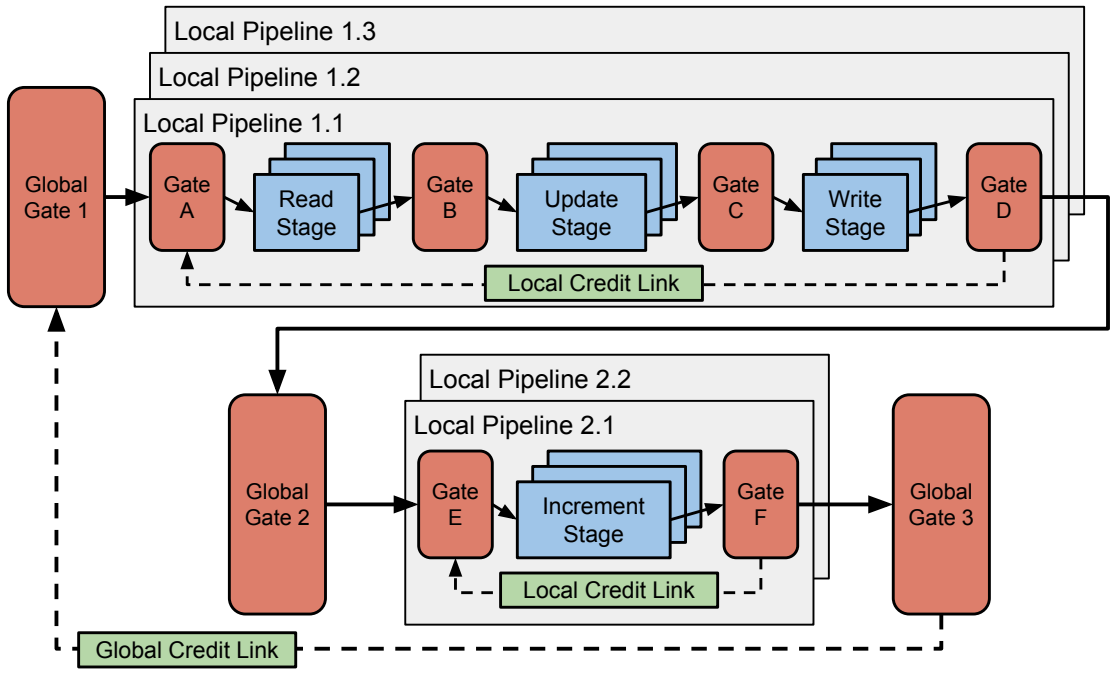


Figure 4.9 – A hierarchical pipeline with scaled-out local pipelines.

resource utilization within the machine on which they run; the global pipeline uses a credit link to control global resource usage (*e.g.*, the number of outstanding batches to the storage system).

4.5.4 Lifecycle

After assembling the aggregate TensorFlow graph containing the application logic, a worker process instantiates, hosts, and executes the elements in the graph associated with a given local pipeline. The construction and execution of a distributed PTF application pipeline builds upon the single-machine process outlined in §4.4.4.

Construction

A distributed PTF application begins its lifecycle with the construction of the application logic. A user's application begins by programmatically constructing the pipeline logic using the TensorFlow API, typically using the Python interface. The construction process starts with a global ingress gate ($G1$) to mediate global access to the distributed pipeline and a second global gate ($G2$) to receive the results of the first local pipeline. The construction process continues by doing the following:

1. Create a partition dequeue operation on gate $G1$ based on the partition size (a parameter to the construction process).

2. Assign a unique logical device (*D*) for the local pipeline.

3. Pass the result of this operation to the first gate of the local pipeline using a transfer stage.

4. Assemble a local pipeline entirely on device *D* using the construction method described in §4.4.4.

5. Pass the result of the local pipeline to an enqueue operation on gate *G*2 using a transfer stage.

These steps are repeated for each local pipeline replica the user creates for this section of the application pipeline, with a new logical device assigned for each replica. The application continues the construction process by repeating these steps in sequence for each local pipeline in the application's logic; gates and local pipeline replicas are constructed in sequence until the pipeline is terminated with a final global egress gate.

Gates in the global pipeline may be placed on any logical device, but are typically colocated on a single dedicated device. Placing global pipeline gates on a logical device corresponding a local pipeline is feasible, but the overhead of servicing the large number of gate operations converging on a global gate may have adverse effects on the performance of the local pipeline. The practical solution is to place all global gates on a single, isolated device that serves gate operations to and from local pipelines. If the size of the global pipeline grows beyond the scale of a single machine, additional devices may be added to different global gates. In practice, we have found that a single machine is sufficient to host all global gates and process user requests.

The aggregate result of constructing a distributed PTF application is a single TensorFlow graph. Even though the construction process of a hierarchical scale-out PTF application pipeline is more complicated than local-only process outlined in §4.4.4, the end result contains the same components: a set of independent TensorFlow graphs, gates, stage runners, and credit runners. The only additional information contained in the scale-out application graph is a logical device annotation for each node in the graph. Stage runners, both those that run stages in local pipelines as well as those run transfer stages, are explicitly annotated during the construction process with a device in which they should execute, as this construct is an extension to the default information in a TensorFlow graph. This deliberate placement ensures that each stage runner executes on the same logical device as the stage it is responsible for running, minimizing overhead between logical devices (*i.e.*, machines).

Execution

Execution begins with the distribution of the application graph to the worker processes. Each worker process is a Python application; each process will typically map to a single machine and requires the following information to successfully start:

- The entire serialized graph description
- The *device mapping*, which maps the logical device names (*i.e.*, those that annotate the nodes and resources in the TensorFlow graph) to worker processes. Each worker process is identified by a unique location based on the network address and port. This information is required by the TensorFlow distributed runtime to identify the network endpoint corresponding to each worker in the cluster.
- The logical device that is to be run by this particular worker

This information may be distributed to the worker processes based on one of many different mechanisms, such as command line arguments upon starting up the worker or a separate out-of-band control plane system.

Each worker process begins execution by creating a TensorFlow graph session and starting the stage runners that are assigned to it. Creation of a TensorFlow graph session involves reading the graph description and instantiating the nodes corresponding to the worker process's device in memory. After session creation, the worker process scans through the collection of stage runners and credit runners and starts a Python thread to execute each runner that is mapped to the worker's logical device. Each worker process executes its local pipeline until it receives an exception, after which it shuts down and propagates the exception, if necessary.

User requests may be processed by a distributed application in one of various mechanisms based on the scaling requirements of the application. For the cluster sizes used in our evaluation of PTF, we found that a single machine was sufficient for both serving user requests and hosting the global gates. For larger cluster sizes, user requests may be served by several machines or may ingest a stream of requests from a task queue, enqueuing the results into a downstream task queue upon completion.

4.6 TensorFlow Compatibility

PTF leverages TensorFlow's runtime and library of existing functionality. In this section, we discuss in which ways TensorFlow is directly compatible with TensorFlow functionality and in which ways it differs.

4.6.1 TensorFlow Queues

TensorFlow's queues served at the basis for gates in PTF. In the first iteration of this architecture [26], we used queues instead of gates due to their existence in TensorFlow. Both gates and queues serve a similar purpose: decoupling independent subgraphs of the overall application to coordinate concurrent processing of feeds based on feed availability. Much of the architecture of gates was inspired by TensorFlow queues; the structuring of the code for gates uses the asynchronous type of operations used for the analogous TensorFlow queue operations.

TensorFlow queues do not support concurrent request processing and differentiation. As we discussed in §2.4, TensorFlow is designed to process a single request at a time. As a result, TensorFlow queues are designed with this constraint in mind. The prior version of this system [26] had a similar structure to PTF (*i.e.*, stages operating as independent subcomponents of the overall application logic), but used TensorFlow queues to decouple adjacent stages because it operated within the constraints of TensorFlow: a single invocation could process only one operation request. PTF was the next iteration of this work, as it transformed the system in this earlier version into one that could support concurrent requests.

4.6.2 Machine Learning Workloads

The main components of a TensorFlow machine learning graph are incompatible with the stateless nature of stages. Specifically, the following aspects require features that stages disallow: (1) variables holding a model's state must persist through the execution of a training program and be uniquely named. They may be reset through custom logic in the user's code, but this is beyond the simple logic in Gate Runners. (2) custom logic driven by the API client (*e.g.*, the Python operation driving the computation) to execute the model based on desired parameters (*e.g.*, error minimization or number of training examples on which to train). This also requires a custom logic in place of the simple code for Gate Runners. (3) the gradients computed on the backward pass of training may not traverse a gate or a queue. This limits the training graph to one single graph. In summary, the training of machine learning models on TensorFlow uses large, monolithic, stateful graphs that are cheap to construct (due to the fact that most nodes are math operations) and requires custom logic in the user code (*i.e.*, not in the TensorFlow runtime) to drive training; although the runtime and nodes in TensorFlow may be shared with PTF, the applications that each framework targets are fundamentally different in nature.

4.7 Summary

This chapter introduced PTF, a framework built as an extension to TensorFlow that serves as a foundation for bioinformatics workflows. PTF leverages TensorFlow's native runtime and dataflow abstraction to construct applications as a pipeline architecture. Applications encode their logic as sequences of stages (small TensorFlow graphs that perform a single subcomponent of the aggregate logic) separated by gates. PTF scales this logic across multiple machines and heterogeneous hardware configurations automatically; the application encodes only its logic, not its execution strategy, delegating the latter to the PTF runtime. By reusing the notion of a TensorFlow feed as input (grouping related feeds of a given request into a single batch) and tagging each feed with metadata, PTF coordinates concurrent request processing without the overhead of a centralized scheduler.

5 Persona

Bioinformatics pipelines, and their constituent applications, perform operations that are critical in transforming the raw output data produced by sequencing machines into a result useful for bioinformaticians and medical professionals. Due to their origins as single-machine applications, these pipelines do not naturally scale out across multiple machines. Limitations in application structure and file format have imposed a barrier for adapting these applications to modern data centers.

This chapter introduces Persona, a framework for constructing bioinformatics built on PTF and the AGD format to scale computation out across multiple machines. PTF enables Persona to construct scale-out application pipelines across multiple machines while taking full advantage of the underlying hardware resources. AGD enables work distribution between various components (*i.e.*, PTF local pipelines) without any coordination based on its chunked columnar design. We port several existing applications into Persona and build new ones.

This chapter is organized into the following sections:

- §5.1 introduces Persona by outlining its goals and high-level architecture.
- §5.2 discusses the techniques that Persona uses to implement its functionality within the PTF framework.
- §5.3 demonstrates how common Persona elements can be assembled as a series of its common subcomponents (*e.g.*, to read and write files) combined with application logic.
- §5.4 discusses how Persona incorporates application code from existing bioinformatics applications and libraries. It includes a discussion of how Persona uses the executor pattern to decouple coarse- and fine-grained execution for porting the compute-intensive alignment applications.
- §5.5 introduces Persona's alignment application, a single-pipeline application that scales out the SNAP aligner to align datasets in the AGD format across multiple machines.

- §5.6 discusses the sort application included in Persona. This application is written specifically for Persona's architecture (*i.e.*, it is not ported from existing sort applications) and scales out its two-phase application across multiple pipelines. This section introduces the align-sort application, a three-pipeline application combining the alignment and sorting applications in sequence.
- §5.7 demonstrates a reorganization of the three-pipeline align-sort application to eliminate a local pipeline by fusing the align and sort functionality into a single pipeline.
- §5.8 summarizes the chapter.

5.1 Introduction

Persona is both a framework for constructing bioinformatics workflows and a suite of PTF pipelines implemented using this framework. Building upon its library of modular components of functionality (*e.g.*, file I/O, compression, genome alignment, *etc.*), Persona provides several bioinformatics workflows. These workflows cover a broad range of functionality, including genome alignment, sorting, post-processing, file conversion, and combinations of these functions.

Persona uses PTF as its underlying framework for the construction and execution of bioinformatics workflows. As discussed in chapter 4, PTF is purpose built for bioinformatics workflows. Persona adds the following components on top of the PTF framework:

- a library of TensorFlow nodes to implement the core bioinformatics operations
- PTF stages that use this additional library to implement components of an application's functionality
- PTF pipelines that contain a series of these stages to create full bioinformatics workflows.

Persona's use of PTF enables nearly identical pipelines to be executed on a wide variety of hardware configurations. With minimal modification to a pipeline (*e.g.*, the degree of parallelism based on cluster size), the same logical sequence of transformations in the pipeline can be executed on a practitioner's laptop as well as in a data center. This is due to PTF's decoupling between the specification of logic and execution. This flexibility enables practitioners to iterate on a small local input when developing a pipeline and then execute it on a cluster of machines with the confidence that the transformation from request to result will be identical.

Persona uses AGD as its primary format for reading and writing genomic data. As discussed in §2.2.3, contemporary bioinformatics file formats (*e.g.*, FASTQ and SAM) are not amenable to either efficient I/O or distributed computation. Although Persona supports these formats for compatibility with existing bioinformatics tools and applications, Persona uses AGD for its

highest-throughput applications. The design of AGD (*i.e.*, chunked and column-oriented) fits well PTF's constructs for parallelism (*i.e.*, stage and local pipeline scaling); each chunk of an AGD dataset serves as a unit of distributed work, enabling concurrent entities in a Persona pipeline (*i.e.*, stage or pipeline replicas) to operate on these work units without any additional coordination.

Persona supports heterogeneous pipelines consisting of both existing bioinformatics applications and new applications tailored to Persona's scale-out architecture. Existing applications are included in Persona by decoupling the core logic of each application (*e.g.*, removing I/O and preprocessing components required for the standalone application) and adding it within the TensorFlow library of nodes that comprise Persona; the decomposed applications typically consist of (a) a shared resource containing persistent components of the original application (*e.g.*, global options, a large shared data structure, or a thread pool) and (b) a set of TensorFlow nodes that interface with the existing application's code, as a library. Persona includes specific new versions of applications when existing applications do not provide sufficient scaling ability, *e.g.*, the scale-out sorting application in Persona.

Standard interfaces between different *phases* of a Persona application pipeline enable interoperability between different operations. Each Persona application is constructed as a PTF pipeline of sequential phases, where each phase is small section of the pipeline (typically one or two stages) that perform a logical operation (*e.g.*, read a file, align a sequence of snippets, compress a buffer). Each of these phases standardizes the input and output *feed types* so they are comparable with similar phases, *e.g.*, all alignment phases share a similar interface. This interface standardization enables a single phase to be substituted with a similar operation without affecting the logic in the rest of the pipeline. importing an external application, *e.g.*, a contemporary bioinformatics application, becomes a more tractable endeavor: the core of the application's logic is encapsulated into a phase by isolating the application code and adding a small additional layer of code to conform it to the phase interface.

5.1.1 Goals for Persona

We specify the following goals for Persona and demonstrate how Persona's design achieves these goals in this chapter.

Support existing applications: The wealth of domain knowledge coalesced into many existing bioinformatics applications precludes *de novo* construction of new replacement applications (*e.g.*, a new alignment application). This is due to (a) the amount of developer time spent implementing the functionality in these applications and (b) the desire by bioinformaticians to continue using similar tools because they understand the existing behavior well. Persona supports these existing applications to be used within a larger bioinformatics workflow by packaging them as a library with interoperability with other components included in Persona.

Enable novel applications: Certain applications in bioinformatics contain little domain-specific knowledge and/or are written in a way that cannot be scaled out due to their structure. For example, dataset sorting applications contain a simple operation at their core: sort an input dataset (*e.g.*, a SAM or BAM file) using a pairwise ordering function (*e.g.*, based on the results of reference alignment, or on the alphabetical based on the metadata field). There are many approaches to this operation while yielding an identical result; an approach that may be optimal for processing a dataset on a single machine may be intractable for use across a cluster of machines. Persona supports the creation of new applications in order to meet this need for scale-out application architecture.

Pipeline modularity: The same components of an application must be capable of usage in both a local and distributed setting; maintaining separate applications and codebases for different scaling requirements is not a feasible option nor one espousing good design. Using PTF's pipeline abstractions, all Persona applications are composed of pluggable, reusable components (*i.e.*, stages). PTF's pipelines can be composed of these same components regardless if the target platform is a laptop or a data center.

Pluggable common operations: Existing bioinformatics applications often write their own unique versions of code to perform common operations; file I/O, compression, and code to scan existing formats is often not sourced from a widely-available library. Persona includes (a) a common library of operations performed by these applications, (b) versions of each library operation to support the new AGD format, and (c) an abstract interface to each resource (*e.g.*, an input file) such that downstream consumers of these resources (*e.g.*, align and sort operations) can consume the data regardless of the original format or source.

Low overhead: Persona must impose a low operational overhead compared, delegating the majority of resources to executing application code. This is a critical goal because bioinformatics applications can be limited in throughput by a hardware resource. Persona achieves this by building on PTF, which builds on TensorFlow's efficient runtime, to achieve this goal.

5.2 Components

Creating a heterogeneous Persona pipeline requires careful architectural design decisions to minimize overhead. Such pipelines may be composed of existing bioinformatics imported into Persona in addition to new applications written for Persona. Each pipeline component must implement a standard interface such that it may be replaced by an equivalent component (*e.g.*, a new operation to support I/O to a new storage system) without forcing downstream components to adapt to the change. This flexibility provides Persona users the ability to incrementally change a pipeline in order to adapt it to their needs. The performance penalty that may be imposed by the conformance on standard interfaces must be minimal.

This section discusses Persona's approach to this issue of constructing heterogeneous pipelines while minimizing operational overhead. Persona pipelines consist of multiple *phases*; each phase consists of one to a few stages. Each phase that performs an operation (*e.g.*, reads a file from the local file system) maintains a similar interface to other phases that perform similar operations (*e.g.*, reading a file from a key-value store); this allows pipelines to easily swap different phases (*e.g.*, to adapt a pipeline to a new storage system or change the aligner on an alignment pipeline) or extend existing ones.

5.2.1 Shared Resources

TensorFlow restricts each node's interface to operate with tensors. Recall from §2.4 that the sole interface for each node in a TensorFlow graph, in terms of values it can consume and emit, is a tensor. For TensorFlow's target domain of machine learning this is an appropriate interface; most workloads in this domain involve the update and manipulation of large numerical tensors (*e.g.*, representing weights in a model). Once data has been imported into this tensor domain, TensorFlow includes mechanisms to efficiently load and store these values to disk, *e.g.*, to store a model after training and subsequently loading it to perform an inference operation.

Adapting bioinformatics information into a tensor format imposes an unacceptably large performance overhead for Persona. Bioinformatics formats must be converted to and from a tensor value, as the on-disk formats (*e.g.*, AGD, FASTQ, SAM, *etc.*) are not capable of using TensorFlow's file operation mechanisms to parse their formats.[1]Although certain bioinformatics data may be represented by one of TensorFlow's numerical tensor types, much of the data is in string format. Creating and populating a large string tensor with information from a bioinformatics file involves a large overhead: each element in a string tensor involves both a heap allocation and a data copy from the source because (a) string tensor elements are capable of storing dynamically-sized strings and (b) tensor lifetimes are managed by the TensorFlow runtime, necessitating a copy from the source file in order to decouple the lifetimes. A single high-quality genomic dataset for a human may contain hundreds of millions of records, which would necessitate potentially billions of copies and small allocations per dataset.

Persona bypasses this limitation by using TensorFlow's shared resource facility to store runtime resources (*i.e.*, any C++ data type) containing genetic information. Nodes pass handles to these resources through a Persona pipeline to pass data from one node to the next. For example, a node that reads a read-only file will memory map the file, store a custom data structure containing a pointer to and the size of this mapped region, and emit a handle to this structure.[2]A downstream node that receives this handle uses it to retrieve a reference to the shared resource in the TensorFlow runtime. The node may then use the resource and either pass the reference downstream as output or destroy the reference.

[1]The incompatibility is due to the TensorFlow file reading operations only supporting a single batch per invocation. They must be reset by the client logic or renewed on a subsequent invocation of the application.

[2]This handle is a string tensor containing a string key, typically assigned by the TensorFlow runtime itself, to the resource. The runtime stores these resources in a C++ map, templated on the resource type.

5.2.2 Common Interfaces

Each Persona *phase* implements a common, reusable piece of functionality within the pipeline. Each phase represents a small component of the overall computation, *e.g.*, reading a file, decompressing a file, aligning. Each phase is encapsulated (typically) one PTF stage, or at most a few stages. In order to be compatible with each other, similar phases implement similar interfaces such that a pipeline may alternate between each phase if needed in order to adapt the pipeline to a different setting, *e.g.*, adding support for a new storage system.

Each phase implements a standard interface, based feed type, in order to remain compatible with other phases. This interface enables a given phase to be exchanged for a different phase in a given pipeline to adapt its functionality in a modular fashion. For example, all read phases supporting a variety of storage systems should have compatible interfaces so a developer must only exchange these phases to adapt the application to a new storage system. [3] The feed type specifies a list of required tensors, each representing a value required or produced by the phase. The shared resource type specifies a runtime type of each shared resource in the feed.[4]Interface types include *bypassed* feed elements, *i.e.*, tensors in the feed that the phase passes through to downstream phases unmodified.

Figure 5.1 shows an example interface for a phase that reads an AGD dataset chunk-by-chunk from a Ceph storage system. The phase contains a single stage and takes a chunk name and a Ceph namespace for the dataset (two strings, serving as its input feed type), which is enqueued into the first gate. The stage extends the chunk name with the two column suffixes ("base" and "qual", constants in the stage's graph) to create keys. These keys are used in conjunction with the namespace to read the chunk files in from Ceph. Finally, these chunk files are combined into a chunk iterator, which a downstream phase will use to iterator over the records in a given chunk, and enqueued into the downstream gate. The output interface of this phase to includes the following elements:

- Feed type: the same tensors from the input (for namespace and key) as well as a resource tensor containing a reference to a buffer containing the result of the read operation.
- Shared resource type: the buffer resource's interface exposes a pointer to the heap allocation containing the result of the read value and a size of the value.

Other bypassed tensors are not shown in the diagram, but may include a string represent a new key for a downstream Ceph write phase to use for writing an updated value back to the storage system.

[3]The details required for a particular phase, such as storage system-specific authentication tokens, may differ, but the logical result of the read operation must be identical (*e.g.*, a buffer of bytes).

[4]The type of each shared resource is the C++ type (*i.e.*, `class`) for the shared resource, as it is stored in the TensorFlow runtime. This is typically an abstract data type (*i.e.*, a virtual class) so that a downstream consumer of the shared resource may be agnostic as to the source (*e.g.*, whether it is a buffer or a memory mapped file).

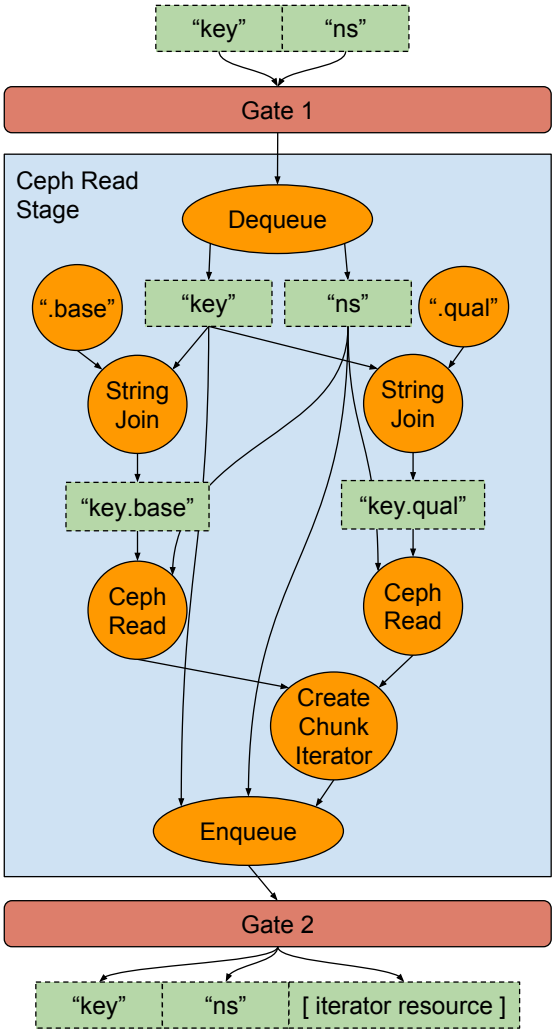


Figure 5.1 – An example Persona phase, comprised of a single PTF stage, that reads two columns of an AGD dataset chunk residing in a Ceph storage system and produces an iterator resource that a downstream phase will use to iterate over the records in the chunk.

5.2.3 Resource Pooling

Persona pipelines must manage the lifecycle of shared resources. Although TensorFlow manages the lifecycle of tensors (*i.e.*, allocating and deallocating them based on data dependencies), the lifecycle of shared resources stored in the TensorFlow runtime must be managed by Persona. The shared resource mechanism was designed to store persistent resources that persist across the entire execution of the application; in order to repurpose this mechanism to efficiently pass native resources through the pipeline, Persona must include facilities to allocate and deallocate these resources as needed (*i.e.*, based on data dependencies).

A naive approach of creating and destroying these resources during pipeline steady state operation imposes a large performance penalty. This approach entails a source phase that

creates the resource (*e.g.*, the Ceph read phase allocating a buffer to receive the operation results) and a sink phase that removes the resource from the TensorFlow runtime after use. This constant allocation and deallocation of resources imposes a large overhead on the memory manager in the process, as heap memory is used to store the resource and the resource itself may contain large heap allocations. For example, a large buffer used to hold a large AGD chunk file may be tens to hundreds of megabytes in size.

Persona addresses this challenge by managing shared resources with *resource pools*. Instead of a stage within the source phase directly creating a resource, it takes a handle to a shared resource pool for the type of resource it requests. The stage requests a resource from the pool and blocks until one is available.[5] If one is not available in the pool, the pool creates a new resource and adds it to the list of resources it tracks. The pool returns a handle to a shared resource the stage emits as a result. Downstream phases may look up this resource and use it as it would any other shared resource. The final stage within the sink phase for this resource uses a reference embedded in the shared resource to release ownership of the resource, returning it back to the pool for reuse by the source phase.[6]

This resource pooling architecture enables Persona to avoid large memory allocations in the steady state of a pipeline. During a warm-up period at the beginning of a Persona application pipeline, shared resource pools will not have any resources in their pools, triggering the creation of resources. After the warm-up period is over, resource pools will always be able to serve resource requests by the source phases from their pools; no new resources will need to be created. This tapering of allocations is particularly important for large buffer allocations: buffers grow based on their requested size, but when the sink phase releases the buffer resource back to the pool, it does not release the buffer; it only sets the size to 0. When the buffer is reused, it does not have to reallocate, as it is already allocated to the maximum size.

Persona's use of resource pooling enables pipelines to use a zero-copy architecture. Once a source phase creates a shared resource, each subsequent phase will pass the resource downstream until the final phase which releases the resource; this minimizes unnecessary copies between phases. A data copy is only triggered when there is a modification made to the data that cannot be made in place, *e.g.*, decompressing data from a buffer into another buffer. By only copying when necessary, Persona pipelines minimize unnecessary memory bandwidth overhead.

Figure 5.2 shows an example of a shared resource pool of iterators used to read data from a Ceph storage system. All resources are managed by the TensorFlow resource manager; as shown in the figure, the stages and the resource pool only manage references to these shared resources. The Ceph Read Stage, an abbreviated version of Figure 5.1, uses a resource pool of chunk iterators to pass each chunk downstream; this stage dequeues a reference to an

[5] Resource pools are not limited in size, but accesses to the pool are sequenced by a lock to protect the internal data structures.

[6] During this release phase, the resource may need to be reset. For example, memory-mapped file resources are reset by unmapping the memory region.

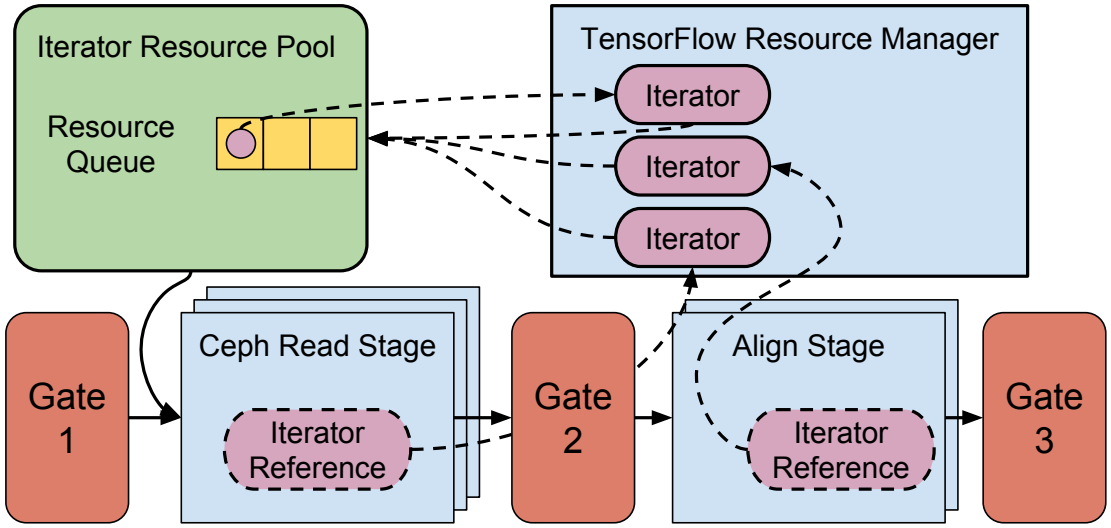


Figure 5.2 – An example of resource pooling for iterators within a Persona pipeline.

iterator from the pool (creating one if one is not yet available) and populates it using the operations discussed in §5.2.2. This reference is enqueued directly into the next gate (Gate 2) as an element of the stages output feed type. The Align Stage (the single stage of an alignment phase) consumes this iterator by iterating through each of the records and then releasing it back to the pool. Note that the Align Stage does not take it as a reference; the iterator resources themselves each contain an embedded reference back to their pool of origin. This enables the Align Stage to remain agnostic about its upstream source of iterators; it performs an identical operation regardless of the original file format or the storage system.[7]

5.3 Pipelines

Persona includes a large library of phases, and TensorFlow nodes to implement those phases, that support common operations. By delegating application logic to these common phases, Persona enables (a) the construction of new application pipelines without repeated effort to implement common functionality and (b) the porting of existing applications by supplanting their included (and often inefficient) facilities for I/O and format-specific file processing. In this section, we discuss these phases and demonstrate how they can be used to implement a simple application pipeline by combining phases from Persona's library with new custom phases that implement the desired functionality.

[7] If the align stage assumes a specific format (*e.g.*, AGD), an upstream read phase may need to perform additional transformations from the input file format. These are typically additional stages immediately downstream of the read phase for the different format (*e.g.*, FASTQ or SAM).

5.3.1 Pipeline Composition

Pipelines are composed of *core* and *peripheral* phases. The core phases implement the central computation (*e.g.*, sorting, alignment, merging, *etc.*). The peripheral phases implement the necessary pre- and post-processing to adapt the core phase to different file formats and storage systems.

This structure decouples the ancillary application components from the core functionality. By interacting with the peripheral phases via shared resources (*e.g.*, to iterate through an existing file or build up a new file), each core phase can perform its functionality independent of the rest of the pipeline. Peripheral phases can be exchanged for others to add new functionality without requiring major changes to the core phases.

5.3.2 Common Components

Persona provides a library of common operations for use in application pipelines. These operations are bundled as peripheral phases and are used implement the auxiliary functions in a traditional application. These phases are typically used as peripheral phases before and after a core phase in a pipeline; they ensure that new functionality (*e.g.*, a new aligner phase) does not have to reimplement existing functionality (*e.g.*, file I/O). They also enable existing pipelines to be adapted to new configurations (*e.g.*, a new file format or a different storage system) without requiring significant changes to existing pipelines; only the necessary peripheral phases must be changed, leaving the core phase(s) untouched.

Storage system I/O: Persona includes phases for reading and writing to two different storage systems: the local file system and Ceph [127]. The interface to the shared resource produced by any type of read phase is a read-only pointer to the data in memory and a length. This shared resource may either be (a) a pointer to a buffer, owned by the resource itself, such as in the Ceph case as it receives data from the storage cluster or (b) a pointer to a non-owning memory region, such as a buffer provided by the storage system's API (*i.e.*, that is released back to the storage system via a library call upon release) or a memory mapped file.

Dataset scanning with iterators: once a file has been read in from storage and preprocessed, Persona encapsulates it in a generic *iterator* interface to decouple the file format from the operations performed on it. The iterator interface consists of an iteration method that returns each record in succession until the end of the sequence. For an AGD dataset, creating this iterator interface requires combining several columns worth of chunk files. In order to maintain the zero-copy design, the iterator interfaces assume ownership of their source resources (*e.g.*, buffers to AGD chunk files or memory mapped files); no copying is necessary to create an iterator resource. When the iterator resources are released back to their resource pools, the iterators, in turn, release their source resources.

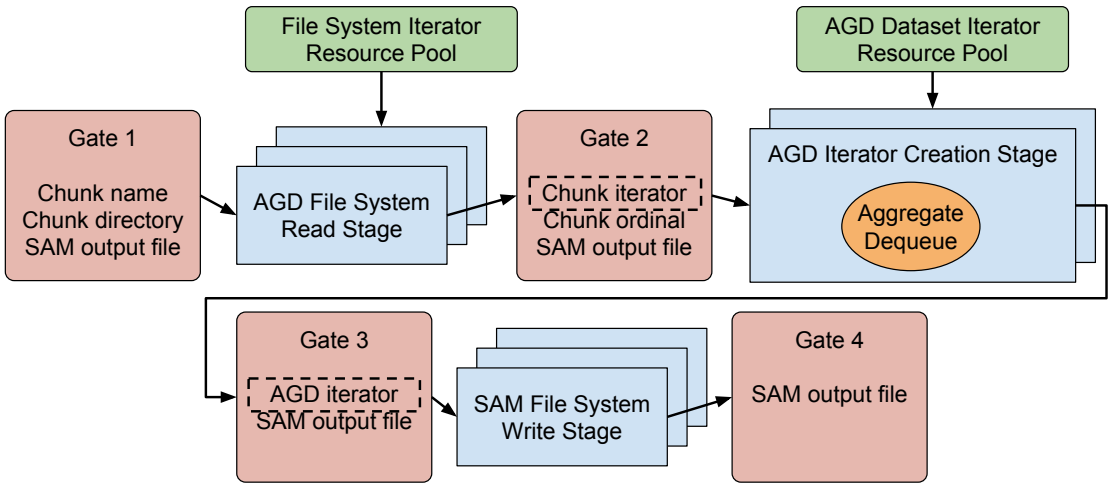


Figure 5.3 – A simple example of converting an AGD dataset into a SAM file.

Dataset assembly with constructors: *constructors* support the assembly of new datasets by repeatedly appending records. Persona supports this to construct new output datasets in various formats, as assembling an AGD dataset may differ (in terms of the records required) from assembling a BAM-formatted dataset. Underlying constructors are typically one or more buffers that expand to hold an increasing amount of data as operations append to them. For example, alignment phases use one AGD constructor per results column to output only the aligned results values; the sort phases use one AGD constructor for each column to output a completely new dataset.

Compression: the compression phases include both compression and decompression, currently only in the GZIP format. In Persona, compression is primarily included in order to support AGD's compressed chunk file format. However, the interface is sufficiently generic: the decompression interface takes as input a memory location and produces another one as its output (a buffer holding the decompressed value). In certain cases, compression is optimized with custom interfaces: the interface to assemble an AGD chunk uses two buffers, one for the records and one for the index. Persona includes specialized AGD chunk compression phases in order to avoid unnecessary data copying of these two buffers into a single buffer.

5.3.3 Example Pipeline

Figure 5.3 shows an example of how Persona applications use the included library of common phases to construct a simple pipeline. This application converts an AGD dataset into a SAM file and uses two included components in Persona:

- The AGD File System Read Stage reads an AGD dataset and produces a batch where each feed contains an iterator per chunk in the dataset. Various components of the metadata

in each chunk file header are produced by this stage (*e.g.*, the number of records, first ordinal of the chunk, dataset name, *etc.*), but the only one used in this application is the ordinal (to order the chunks).

- The SAM File System Write Stage takes a single iterator and writes out a monolithic SAM file for each record produced by the iterator.

The component added for this application is the AGD Iterator Creation Stage, which creates a custom iterator to iterate in dataset order across all records. This stage does this by (a) using an aggregate dequeue operation to produce an aggregate feed of all feeds in a batch, (b) ordering the chunk iterators in this aggregate feed based on their ordinal, and (c) creating an AGD Dataset Iterator, a single shared resource which produces records in dataset order (*i.e.*, in order of ascending AGD ordinal).

5.4 Porting Existing Applications

Encapsulating existing bioinformatics applications for use in a Persona pipeline is a necessary, but challenging component of Persona's architecture. Existing applications often contain tightly coupled components that are difficult to decouple in order to contain them in a single phase in Persona. In this section, we explain how existing applications are included in Persona and the techniques necessary to achieve maximum performance from the most complicated applications ported to Persona: the alignment applications.

5.4.1 Decomposition

Existing bioinformatics applications contain components that perform the *core* computation and other components that perform *peripheral* functionality. These peripheral processes perform the operations necessary to (a) convert application input into a format suitable for the core computation and (b) convert the result back into the desired output format. These peripheral components involve I/O, compression and decompression, and interacting with various file formats (to both scan through input files and construct output files).

The peripheral code components in existing applications are often tightly coupled with the core computation. These applications were typically developed with the intent of using them as standalone monolithic applications. Careful partitioning between core and peripheral components is uncommon; for both practical and efficiency reasons, both types of components interact with each other in a way that is difficult to separate.

Persona incorporates both the core and necessary peripheral components into a single phase per application. The application is imported directly into Persona's TensorFlow library and the core computation (*e.g.*, alignment) is encapsulated into a stage. Peripheral components are included in the same stage if they are inexpensive; expensive peripheral components are included in separate stages in the same phase so that the TensorFlow runtime may execute

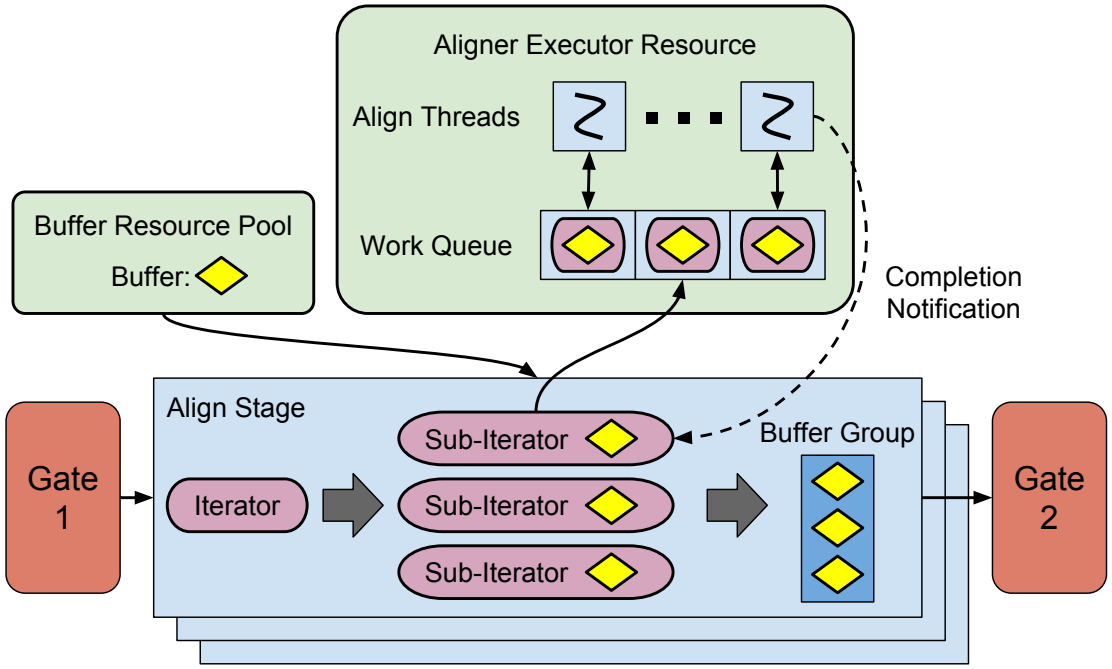


Figure 5.4 – The design of the executor model used in Persona's alignment phases.

them in parallel with the core stage. For example, certain peripheral stages may be required to transform an input file to a specific format; the BWA aligner requires a custom format for the bases, which requires an expensive conversion step that is included as an additional stage. Included applications that interpret the bases may not be able to read the AGD format for bases and alignment results; Persona includes stages that convert these inputs to more amenable formats.

Many of the peripheral components are supplanted by Persona's common phases. The common phases implement the peripheral functionality (*e.g.*, I/O and compression) more efficiently within the TensorFlow runtime; such functionality in existing applications is often tied to application-specific constructs that do not translate well, *e.g.*, a custom thread pool and work queue architecture. Common phases add additional functionality as well: no existing application supports AGD. Additional file formats and compression schemes may be easily added to Persona as new common phases without modifying the core phases of any application.

5.4.2 Executors

Effectively using all CPU cores for alignment is a critical for increasing the throughput of an alignment application. Alignment applications (*i.e.*, both BWA and the SNAP aligners included in Persona) use hundreds to thousands of CPU cycles per record to perform their operations. The amount of computation per record eclipses all other CPU operations on a local Persona pipeline performing alignment.

A naive approach of directly performing the alignment operation in a TensorFlow node inhibits performance due to the number of records in each input. Each input to such an alignment node takes, as input, a shared resource reference to an iterator (§5.3.2). The number of records in this iterator is based on the upstream phases: each iterator typically iterates over an AGD chunk which may contain hundreds of thousands to millions of records, depending on the ideal size for the characteristics of the storage system. Even if stage scaling is used to scale out this alignment computation across all cores, the latency of aligning a single chunk will still be determined by the single-thread performance of alignment and the number of records in each input. Load imbalance between each input (*e.g.*, AGD chunks with more expensive records to align) may increase latency for aligning a dataset, as aligning resources may idle while awaiting completion of straggler inputs.

Persona uses the *executor* design pattern to separate the alignment operation from direct execution in the application graph. In this pattern, the alignment operation is performed by threads in a thread pool encapsulated in an *executor* resource, a persistent resource shared by all alignment phases colocated on a single machine. Work is provided to the executor (*i.e.*, to its threads) via a common work queue; each work item is an iterator to a collection of reads. The alignment phase includes this executor as an argument to one or more alignment nodes, *i.e.*, TensorFlow nodes in the application's graph. Each of these nodes partitions each input iterator (sized in terms of the I/O input) into a sequence of smaller iterators (sub-iterators) based on a size parameter (typically 100 records per sub-iterator). The alignment nodes enqueue the sub-iterators into the executor and await the result, which is output as the result of executing the alignment node.

The executor pattern enables Persona alignment applications to decouple the I/O chunk size from the ideal computation chunk size. The computation chunk size is encoded as a parameter into the alignment nodes, as they partition the input chunk size (*i.e.*, the I/O chunk size). Regardless of the input chunk size, the computation chunk size is the same; only the number of partitions varies between chunk sizes. This bounds the maximum work imbalance between compute chunks independent of the input chunk size (based on file size on the storage system).

The Persona alignment application can independently adjust the size of both the TensorFlow runtime's thread pool (on which TensorFlow kernels are executed) and the executor's thread pool in order to achieve the highest throughput. In practice, we have found that it is sufficient to assign both thread pools to be approximately the size of the number of CPU cores; there is a high computational cost ratio of the alignment operation versus the ancillary phases in the pipeline. A key factor in this is that a single TensorFlow kernel invocation is required to both partition each input iterator and await the result from the executor; invoking a synchronous TensorFlow kernel operation per partition would impose a significant overhead and preclude some of the optimizations possible with the single-invocation model we design in Persona.

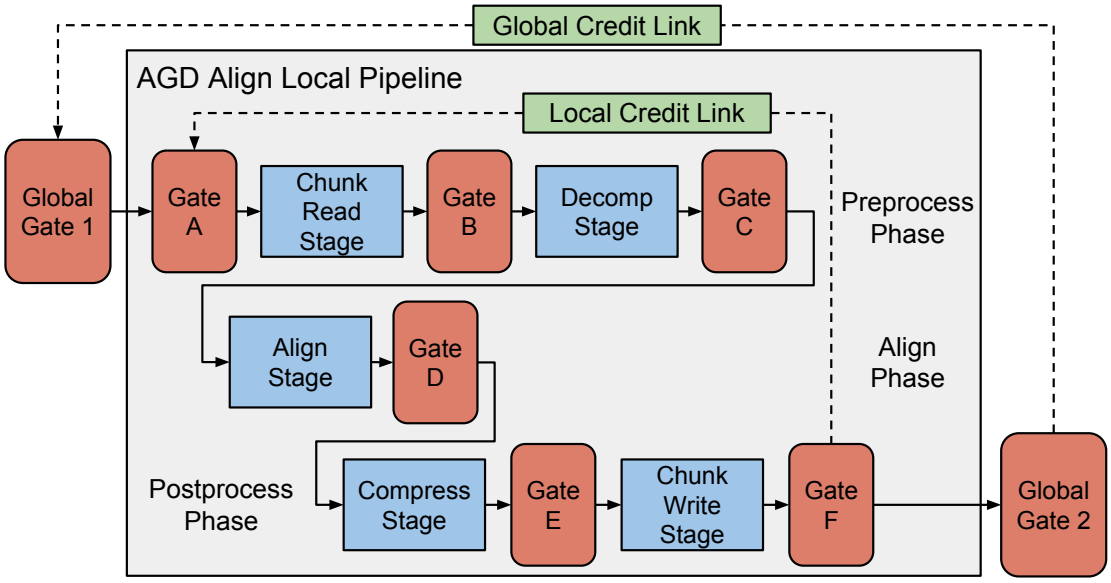


Figure 5.5 – The Persona alignment pipeline for AGD datasets showing the 3 phases of the pipeline: preprocessing, align, and post-processing.

Figure 5.4 shows the executor model used in Persona's alignment phases. The Align Stage takes an input iterator (typically representing an AGD chunk) and splits up the iterator into several sub-iterators based a sizing parameter. It combines a buffer with each sub-iterator (from a Buffer Resource Pool it takes as input) and enqueues this pair into the work queue of the aligner executor. Each Align Thread repeatedly dequeues sub-iterators from the work queue and notifies the Align Stage when it completes each task (*i.e.*, aligning records from one sub-iterator). Once the Align Stage is notified that all sub-iterators have been aligned by the executor, it combines all result buffers into a Buffer Group (a buffer-like shared resource comprised of multiple smaller buffers) and enqueues it into the downstream gate. If the Align Stage is configured to output secondary results, additional buffers are inserted into the sub-iterator for each secondary results column and aggregated into an additional Buffer Group.

5.5 Alignment Pipeline

The alignment pipeline consists of a single alignment phase and several peripheral phases to interact with the storage system, the desired file format, and to create the shared resources necessary to abstract these details to the alignment phase. We specifically focus on the SNAP alignment phase [132]; the BWA alignment phase contains a similar structure. The alignment phase follows the executor pattern discussed in §5.4.2. The SNAP executor resource takes an additional parameter: a handle to the shared k-mer location map (§2.2) that enables its fast global alignment operation.

Figure 5.5 shows the architecture of Persona's alignment application for AGD datasets. Each batch contains one feed per chunk file, with each feed containing the information to read the chunk from the storage system (*e.g.*, the file path location or the Ceph key and namespace). This application contains three phases:

- Preprocessing: reads the input AGD chunk contained in each feed and decompresses the payload to creator an iterator.
- Alignment: aligns the reads in each iterator, building up an output buffer to hold each alignment result using Persona's constructors. The shared resources associated with each input iterator are released after this phase.
- Post-processing: compresses the result buffer into an AGD chunk file and writes only the alignment results back to the storage system.

Each phase contains one or more stages, each of which can be scaled via stage replication to increase hardware resource utilization. The local pipeline may be replicated across multiple machines to scale out across hardware resources.

5.6 Align-Sort Pipeline

Persona includes sort phases for sorting datasets. Sorting is the second application in a bioinformatics workflow, following alignment, and is critical for downstream analysis applications. This section discusses Persona's implementation of dataset sorting and how the align and sort phases can be combined into a single Persona application. This is the first multi-pipeline Persona application introduced in this dissertation.

The merge pipeline included in Persona only supports AGD inputs and outputs. Sorting bioinformatics datasets incurs a high ratio of I/O to computation. The AGD format enables this I/O to be spread out across multiple machines for portions of the algorithm much more effectively than a traditional bioinformatics format. However, similar design principles discussed in this section would apply to a Persona sorting application for these legacy formats as well.

5.6.1 Sorting Architecture

The Persona sorting application is purpose-built for PTF's architecture. Unlike the process of porting existing applications into Persona phases, the sorting application creates new core phases with new code specifically for this application. This is because very little domain knowledge of bioinformatics is required to perform this operation correctly: sorting only involves rearranging an input dataset into an ascending order based on the results of the alignment operation.

Persona's sorting application is structured as a two-phase merge sort. Sorting fundamentally contains at least one serial component, but that serial component may be limited to a small, final component in the sort algorithm. This is advantageous for Persona in a scale-out context: all phases except for the final merge in a merge sort may be performed in parallel. We structure Persona's sort application in two phases, each of which is contained in a PTF local pipeline:

- Sort phase: sorts a group of N inputs into a single output file. This is the first phase of the merge sort algorithm and can be performed in parallel by many machines (via PTF local pipeline replication). Each sort phase combines groups of N AGD chunks (across all columns in the dataset) into an intermediate AGD dataset (*i.e.*, one intermediate dataset per group of N); it first sorts each of the N input chunks and then merges all of them into the intermediate dataset by creating one monolithic chunk (*i.e.*, one chunk file for each column).

- Merge phase: merges all sorted inputs into output partitions of M records each. This is the final phase and must be performed on a single PTF local pipeline per dataset in the sort application.

For large datasets, it is possible to structure the sort operation to have more than two phases. In this case, subsequent sort phases would precede the merge phase, each of which would group intermediate datasets into increasingly larger chunks. Note that only the first sort phase must sort its input chunks before merging them into a single intermediate dataset; subsequent sort phases only merge their inputs into a single output chunk. However, based on the size of the input data for a high-quality human dataset, the configuration of a large server in a data center, and the latency of a modern high-performance shared storage system (*e.g.*, Ceph), it is sufficient to structure this operation with a single sort phase.

Both the sort and merge phases make use of PTF's aggregate dequeue semantics. In the sort pipeline, the sorting stage requests an aggregate feed of size N to sort and merge. In the merge pipeline, the merging stage requests an aggregate feed of all feeds in a batch, effectively forming a barrier before merge node in the stage. In a scale-out Persona sort application, both pipelines use similar parameters when requesting partitions from the global pipeline. Due to PTF's semantics, Persona can be sure that all chunks in each aggregate feed are associated with the same batch.

5.6.2 Merge Pipeline

The merge pipeline requires the entire dataset to be read and subsequently written, possibly in a compressed output format. Overlapping the merge phase's operation with other phases that pre- and proceed it is critical to both (a) minimizing latency of the merge operation and (b) increasing utilization of hardware resources (*i.e.*, compared to a sequential application architecture). This overlapping is particularly critical for performance when (a) a shared

storage system is used in the application pipeline (*e.g.*, Ceph) or (b) the output chunks of the merge phase are compressed, requiring additional CPU resources.

In this section, we discuss relevant details of the merge pipeline's design goal of remaining PTF-compatible while overlapping large amounts of I/O with compute. First, we discuss the design of the merge node, which performs the core merging operation, specifically how it performs this computation without disrupting PTF's pipeline architecture. We then examine the merge pipeline's design for overlapping the I/O, merging, and compression phases in order to achieve hardware-bound performance.

Overcoming Architectural Limitations

Persona implements the core merge operation as a single merge node in order to circumvent TensorFlow's semantic restrictions and enable pipelined execution. Recall from §4.3.1 that a stage executes one input feed at a time to produce a corresponding output feed.[8] A single merge node contains a fundamentally serial operation (*i.e.*, merging an unknown number of intermediate files). Creating a single output for this node would require the entire merge operation to produce a full output dataset before enqueuing the resulting aggregate feed into the downstream gate, preventing the overlap of merging with downstream compression and I/O.

Persona's merge node circumvents this limitation by directly enqueuing its results in chunks (each corresponding to an AGD chunk) into the downstream gate. Based on a chunk size parameter (S) passed to the merge node during pipeline construction, the merge node repeatedly appends records in sorted order into a chunk constructor[9], directly enqueues the completed chunks into the downstream gate when S records are in the chunk or the input is exhausted, and resets the constructors (by requesting a new buffer-backed constructor from a shared pool) to repeat the chunk creation process. In effect, the merge node both (a) performs the merge operation and (b) serves as the enqueue node for its PTF stage. This design enables the merge operation to overlap with downstream compression and I/O; once the merge operation enqueues a completed chunk into the downstream gate, downstream compression and I/O can begin processing that chunk while the merge operation continues.

The merge node modifies the feed metadata. In order to enqueue into the downstream gate, the merge node requires the following inputs resulting from its preceding aggregate dequeue operation[10]:

- An aggregate feed of iterators representing the intermediate files to be merged

[8] This is due to the underlying limitations of TensorFlow, which imposes this one-to-one semantic requirement on PTF stages.

[9] The merge operation uses one chunk constructor per AGD column.

[10] The size of this dequeue node must be greater than any dataset (*i.e.*, the maximum integer size) because the aggregate must contain all feeds for a given batch.

- The metadata associated with the feed
- An arbitrary-length list of other feed tensors to be enqueued with each result into the gate. This contains fields used for downstream operations, *e.g.*, the record ID for the AGD dataset, a filesystem path to a directory where the output dataset should be written, *etc.*.

The arity of the metadata is 1, as the feed contains the entire batch. However, the arity of the output dataset is determined on the number of records in the dataset and the chunk size specified to the merge node; the merge node uses these fields to compute the proper arity for the batch[11], assigns this arity to the existing metadata (maintaining the existing IDs), and enqueues the resulting chunks into the gate.

Overlapping Compute and Storage System Input

The merge pipeline overlaps storage system input with merging using PTF's pipeline architecture. A read phase precedes the merge phase in the pipeline; this read phase is responsible for reading in all intermediate datasets and producing a dataset iterator (§5.3.2) for each dataset.[12]For datasets that are small relative to the memory available to the merge pipeline, the read phase synchronously reads in all intermediate datasets into memory and produces iterators that scan these in-memory datasets. The merge phase in the pipeline consumes these iterators by scanning them in order to produce the sorted output chunks comprising the final dataset. The read phase can run in parallel with the downstream merging of a previously-read dataset using PTF's concurrency features, as they are contained in different stages. This is similar to other Persona pipelines that read, process, and write back a result, *e.g.*, the alignment application and the sort phase.

The standard read phase is not a viable approach for large genomic datasets. The standard read phase reads the entire dataset, contained in all the intermediate files, into the memory of the merge pipeline. Although this simple approach enables fast iteration through the iterators, the memory footprint can quickly exceed a well-provisioned machine. Multiple merge stage replicas are required to saturate the throughput of a modern server (*e.g.*, using the CPU for dataset compression, after the merge phase); with the standard read phase, one dataset must be held in-memory for each replica plus additional upstream datasets awaiting a merge operation (so that the merge phase is saturated). With a high-quality dataset in uncompressed AGD format being upwards of 50 gigabytes in size, this can quickly become an untenable approach.

Persona's merge pipeline includes a lazy variant of the read phase to support large datasets. The merge operation linearly scans each of its input datasets; the merge pipeline uses this

[11]The merge node assigns this computed arity to all metadata, including both local and global in Persona's scale-out sort application.

[12]The read phase produces a dataset iterator for each AGD column's single (large) chunk file in the intermediate dataset.

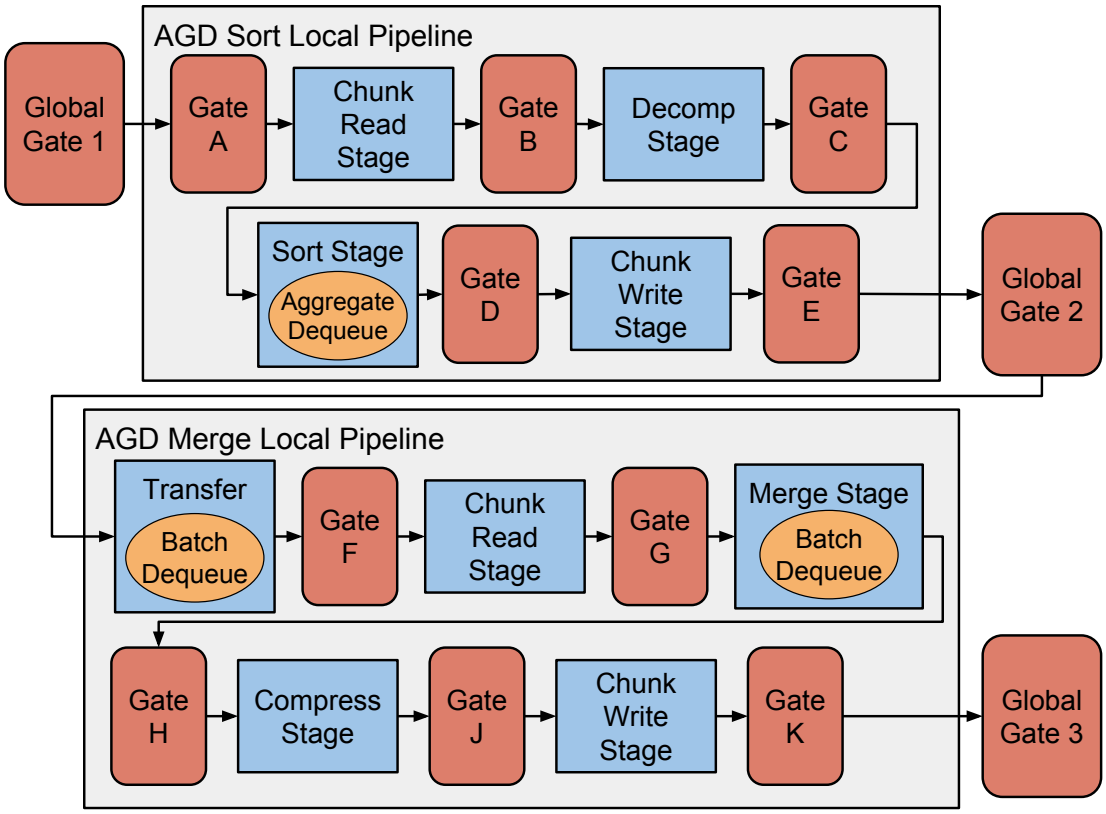


Figure 5.6 – Persona's sorting application for AGD datasets containing two local pipelines for the sort and merge phases of the merge-sort operation.

insight to support a variant of the read phase that produces lazy iterators. These lazy iterators only keep a small region of the overall input dataset in memory. This region advances using asynchronous read operations to pre-fill subsequent regions, bringing them into memory before the merge computation reaches that point in its scan through the dataset. Once the merge operation scans past part of the in-memory region of the file, the lazy iterator recycles the buffer holding that file region by issuing an asynchronous read request for further file regions. The asynchronous read operations are performed using the Ceph library's built-in asynchronous operations. The lazy read phase's iterators issue asynchronous requests into Ceph, which returns an indicator that is marked when the operation has been completed; Ceph uses an internal thread pool to manage the lifetime of the synchronous request to fulfill this operation. This operation enables lazy iterators to not block the merge operation while keeping only a small subset of each intermediate dataset in memory at any given time.

5.6.3 Merge-Sort Pipeline

Figure 5.6 shows the architecture of Persona's sorting application for sorting AGD datasets. The sort pipeline contains a similar preprocessing phase as the AGD alignment pipeline in

Figure 5.5. The sort stage of this pipeline aggregates multiple AGD chunk iterators to produce the intermediate, monolithic AGD chunks. The transfer stage, normally elided for clarity in prior PTF pipeline figures, is shown for the merge pipeline; it performs a batch dequeue, requesting a partition containing all feeds in a batch, *i.e.*, all feeds describing the location of intermediate chunks. These chunks are read in parallel by the merge pipeline's Chunk Read Stage, and aggregated again into a single aggregate feed in the Merge Stage in a similar manner. The merge pipeline shares its final post-processing phase (to compress and write the resulting sorted AGD dataset) with the alignment pipeline. Global and local credit links mediate the number of open batches in the global and local pipelines, respectively, similar to those shown in the alignment application in Figure 5.5; they are elided here for figure clarity.

Persona uses PTF's concurrency capabilities in the sort application. The first gate in the application distributes partitions across multiple sort local pipelines to process them in parallel. The batch dequeue on the second gate forms a barrier between the two pipelines: all pipelining to subsequent application components (*i.e.*, the merge pipeline) is halted by this operation, as it imposes a barrier. This logic is performed for each batch in the pipeline yields identical operation regardless of the number of ordering of concurrent batches.

5.6.4 Combining Alignment and Sorting

Persona's alignment and sorting applications can be combined to form a single *align-sort* application pipeline. Sorting a dataset based on its alignment results is a common next step in a bioinformatics pipeline, as downstream applications typically rely on their input datasets being in ascending order of alignment location. Instead of running the alignment and sorting applications separately, Persona enables the two application pipelines to be linked in sequence. This simply involves appending the sort application's two-pipeline logic (sort and merge) after the alignment application's single pipeline. This is the analog of a traditional bioinformatics workflow combining alignment and sorting.

PTF's ability to pipeline feeds of a batch across multiple local pipelines enables alignment and sorting to overlap. Unlike a traditional bioinformatics workflow which often requires sequential invocation of alignment and then sorting applications, Persona's align-sort application enables a single batch to overlap processing on the first two local pipelines: alignment and sorting. Once the global gate after the alignment local pipeline can satisfy the partition dequeue request by the subsequent sort local pipeline, it sends a partition to the latter even though other feeds of the request have yet to be aligned.

5.7 Fused Align-sort Pipeline

Combining the alignment and sorting applications by concatenating their respective pipelines creates an align-sort application with a large amount of I/O and computational overhead. This concatenation creates an aggregate align-sort application wherein each of the three

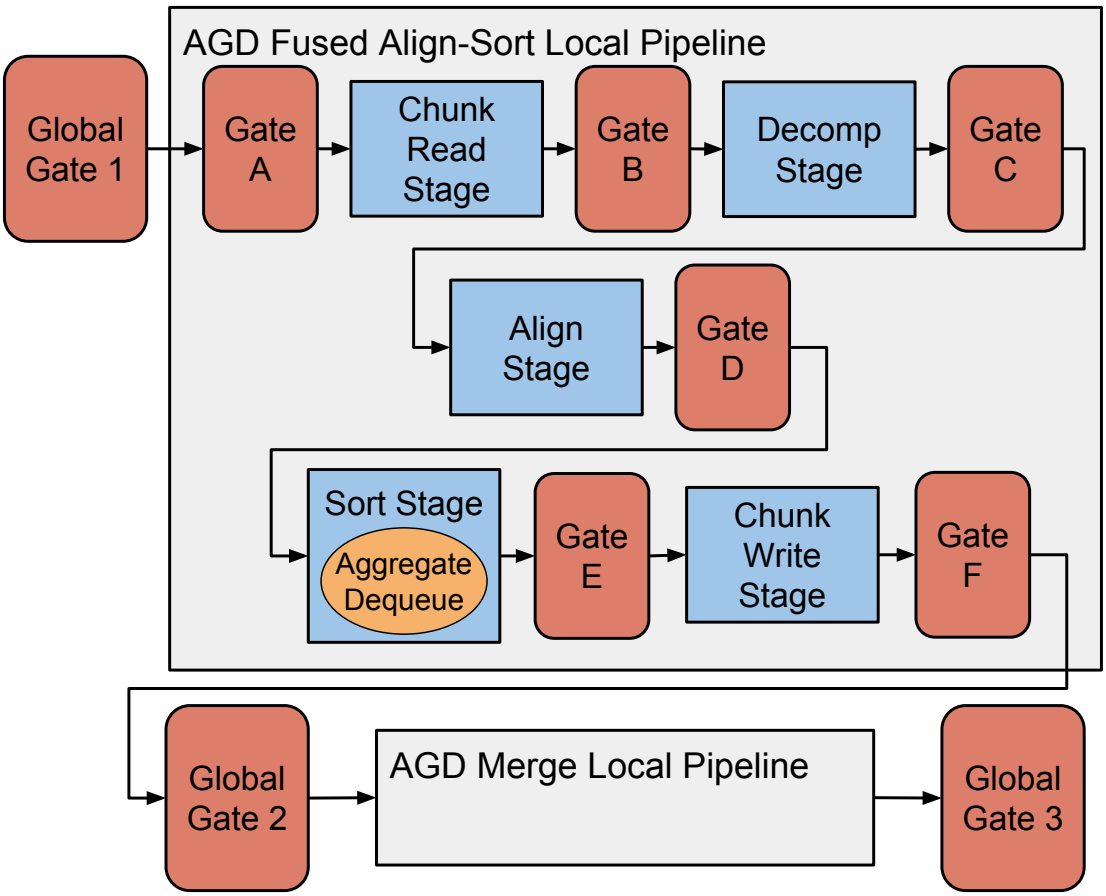


Figure 5.7 – Persona's fused align-sort application pipeline that combines alignment and sorting (the first phase of the sort pipeline) into a single local pipeline (fused align-sort pipeline).

local pipelines, in aggregate across all replicas, read and write each dataset. Compression and decompression accompany several of these I/O operations in each pipeline, consuming additional computation resources across multiple machines.

The align-sort pipeline structure can be rearranged without affecting the input or output of the application. The configuration of the storage system dictates the parameters of the application input and output datasets (*e.g.*, the filesystem path or the number of records per AGD chunk). However, the I/O patterns and pipeline structure within the application can be altered without affecting the initial input and final output configuration of the application. For example, certain stages can be placed on different local pipelines so that they are run on different machines. This enables optimizations that rearrange application phases, placing them on different pipelines or possibly eliminating some altogether.

Effectively restructuring the align-sort application begins with the examination of hardware resource utilization. Each local pipeline from an optimized Persona application is bound by a hardware resource, *e.g.*, the throughput of the CPU, memory bandwidth, or NIC. The key

insight is that phases from different pipelines may be combined without a loss in throughput if they each saturate different resources. We leverage the following observations of the performance of the standard align-sort application to serve as guidelines for restructuring application components:

- The alignment pipeline is bound by the alignment phase's CPU usage, with the upstream and downstream I/O phases underutilized.
- The output of the alignment pipeline (*i.e.*, the compressed result columns of the AGD dataset) are not needed by the overall application. These results are superseded by the sorting dataset produced at the end of the merge phase.
- The sort operation in the first phase (sorting) of the merge-sort application requires a small amount of CPU to saturate the write bandwidth.
- The sort phase must reread portions of the dataset that the alignment pipeline already reads in to perform its operation.

The *fused* align-sort application combines the alignment and sort pipelines into a single local pipeline. This pipeline fuses these two operations together by placing the sort phase immediately after alignment. The sort phase aggregates groups of N AGD chunks together and performs the sort and write-back of the intermediate dataset as on the standard sort-only pipeline. This eliminates the compression and write-back of the alignment results as well as the rereading and decompression of columns necessary for alignment by the sort pipeline.

Figure 5.7 shows the fused align-sort Persona application. The preprocessing and align phases of the fused align-sort application are similar to those in the alignment application in Figure 5.5, but instead of reading only the AGD columns used to perform the alignment operation (*i.e.*, the bases and quality scores), all input columns are read from the storage system. Unlike the alignment application's align stage, the align stage in the fused align-sort pipeline does not release the input iterator resources (*e.g.*, the decompressed input chunks in memory); this resource releasing is performed by the sort stage, as it needs all columns, including the alignment results and original input columns, to perform its operation. The merge local pipeline is identical to that shown in Figure 5.6; the details are elided for brevity.

5.8 Summary

This chapter introduced Persona, a system for constructing bioinformatics-as-a-service workflows. Building on PTF's pipeline abstraction, Persona includes a library of common functionality (*e.g.*, to read and write various file formats and storage systems) that its applications use to assemble scale-out bioinformatics workflows. PTF's runtime enables these applications to run as indefinitely-executing services, serving concurrent user requests while scaling the

computation across multiple machines. Using AGD's chunked architecture, Persona can distribute work to multiple machines with minimal coordination. Persona includes components for interacting with multiple formats (*e.g.*, FASTQ, SAM, AGD) and for performing multiple bioinformatics transformations (*e.g.*, alignment with SNAP or BWA-MEM, dataset sorting, duplicate marking).

6 Evaluation

In this chapter, we evaluate the architecture of Persona, in particular how its use of the AGD format and the PTF framework enable it to scale up across the resources of a single machine and scale out across resources of a cluster of machines. The contents of this chapter are as follows:

- §6.1 describes the hardware resources and inputs used in the experiments.
- §6.2 compares Persona to existing applications on a single machine in order to demonstrate Persona's ability to improve performance through its request pipelining capability, *i.e.*, overlapping I/O and computation more effectively than existing applications.
- §6.3 evaluates Persona's alignment application as it scales across multiple machines. In this section, we demonstrate that Persona imposes minimal overhead when distributing work (feeds) to a cluster of machines to enable linear scaling.
- §6.4 evaluates Persona's fused align-sort application as it scales across multiple machines. We analyze the more complex behavior and performance tradeoffs in this more complex Persona application and demonstrate that a properly-tuned configuration can saturate all resources.
- §6.5 concludes this chapter.

6.1 Experimental Setup

We use a cluster of 20 typical data center machines, each with two Intel Xeon E5-2680v3 CPUs at 2.5GHz, 256 gigabytes of DRAM, and a 10GbE network interface. We enable hyperthreading on all 12 cores per socket, for a total 48 logical cores per machine. All machines run Ubuntu 18.04 Linux with the distribution's default Linux kernel (4.15.0). Each machine includes 2 SSDs in RAID1 configuration for the OS, 6 SATA HDD (each drive 4TB, 7200 RPM, 1 Gb/s), a hardware RAID controller, and 10GbE network interface. For single-node (local) experiments, we store the input data on a 24 TB RAID0 disk array.

The benchmarks read and write to a genomic database of half of a paired-end whole human genome dataset from Illumina [45] (Platinum dataset, ERR174324), which consists of 223 million single-end 101-base reads. All datasets processed by Persona are stored in the AGD format with a chunk size of 100,000 (*i.e.*, the number of records in each chunk file).

For distributed (cluster) experiments, we store the input datasets, intermediate files, and output datasets on a Ceph distributed object store [127] comprised of 18 nodes, each of which has 10 disks and uses a solid-state drive to store the journal. The Ceph cluster is configured to use 3-way replication and runs the Luminous release on both client and server. All datasets are stored in the same pool in the Ceph cluster, with namespaces being used to segregate datasets from each other. Persona accesses Ceph objects via the RADOS API. The compute and storage are connected by a 40GbE-based IP fabric consisting of 8 top-of-rack switches and 3 spine switches.

The latency for each request is defined as the service time of a request once it is submitted to the pipeline. The throughput is measured as the number of bases (in millions, *i.e.*, *megabases*) processed per second.

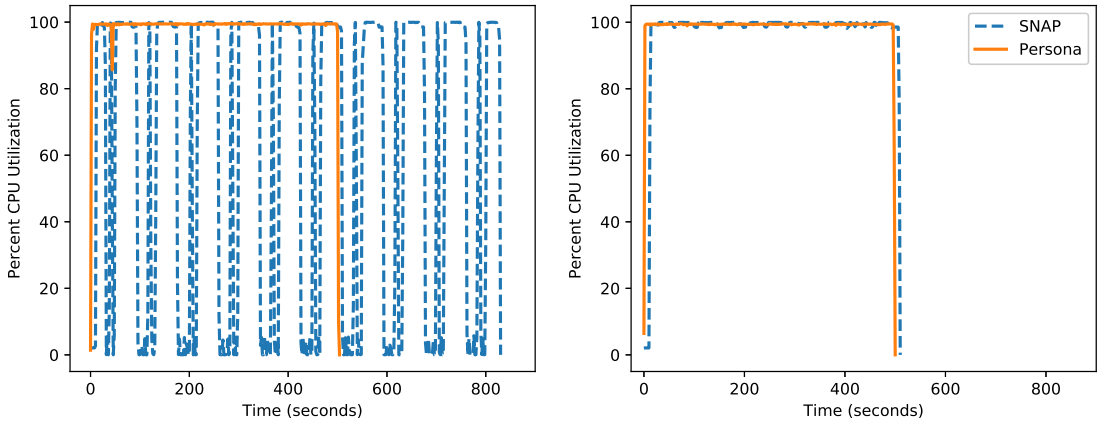
All local pipelines configure a sufficient number of stages such that a local resource is saturated (CPU, NIC, or main memory). Any additional stage replica does not increase performance, as all stages are executed by the TensorFlow runtime using a fixed-size thread pool per machine. Each additional stage replica specifies the maximum possible parallelism for a given stage, but the TensorFlow runtime decides which nodes amongst all stages to execute based on feed availability.

6.2 Single-Machine Performance

In this section, we evaluate the performance of Persona on a single machine to compare it to traditional bioinformatics applications and I/O behavior. For this purpose, we operate Persona as a single-invocation application; a single invocation of a given application is used to process one user request, which corresponds to a single batch. This provides an equitable comparison to existing single-machine applications, which operate in this manner. We demonstrate that Persona is able to scale up to use the resources of a single machine due to AGD's design and the TensorFlow runtime underlying Persona.

6.2.1 I/O Behavior of AGD

We first study the I/O behavior of Persona and AGD. I/O behavior in Persona is fundamental, since we can never assume a given patient's genome data will already be in memory (or that it even fits in memory). We perform alignment using different disk I/O configurations, using the SNAP alignment subgraph and comparing to the SNAP standalone program. We use SNAP instead of BWA because it has higher throughput and is better able to exercise the I/O



(a) Single Disk CPU

(b) RAID0 CPU

Figure 6.1 – Comparison of SNAP (GZIP-compressed FASTQ) and Persona (AGD) in CPU utilization with single disk and RAID0.

	SNAP	AGD Single Node	Speedup
Disk(Single)	817 sec	501 sec	1.63
Disk(RAID)	494 sec	499 sec	0.99
Network	760 sec	493.5 sec	1.54
Data Read	18GB	15GB	1.2
Data Written	67GB	4GB	16.75

Table 6.1 – Dataset Alignment Time, Single Server

subsystem. The single disk configuration stores the genome (and the results) on a single local disk. The RAID0 configuration uses a hardware RAID0 array of 6 disks to increase bandwidth. Both SNAP and Persona are tuned for best performance, and use 47 aligner threads.

Figure 6.1 provides a characterization of the CPU utilization using a single disk and the full RAID0 configuration. Both systems overlap I/O and decompression with alignment: SNAP uses an ad-hoc combination of threads, whereas Persona leverages dataflow execution. Figure 6.1a and Figure 6.1b show that Persona is CPU bound in both configurations, but that SNAP can only use the CPU resource fully in the RAID0 configuration.

In particular, Figure 6.1a shows a cyclical pattern with SNAP where the operating system's buffer cache write-back policy competes with the application-driven data reads; during periods of write-back, the application is unable to read input data fast enough and threads go idle. In contrast, Persona shows identical performance in both Figure 6.1a and Figure 6.1b. This is due to the efficiencies in the AGD data format (only a subset of the fields must be read) and PTF's ability to more effectively overlap I/O and computation.

Table 6.1 summarizes the difference in terms of the amount of I/O traffic required as well as the impact on execution time. While the column-orientation of AGD has a marginal benefit in terms of data input, it has a 16.75× impact on data output, and a 1.63× speedup for the single-disk configuration. When the storage subsystem provides sufficient bandwidth, as for the RAID0 configuration, the performance of SNAP and Persona are nearly identical. Persona, however, does at least the same amount of work with less hardware and eliminates the disk I/O bottleneck. As another point of comparison, we align the dataset using the popular BWA aligner [78], with 48 cores. BWA completed the alignment in 4,073 seconds (67.8 minutes, 54,891 reads per second).

The benefits of column-orientation of AGD are not limited to local disks. Table 6.1 also shows the speedup of 1.54× when the data is stored on Ceph network-attached storage. SNAP does not natively support reading from Ceph, so we use the `rados` utility to pipe the dataset in GZIP-compressed FASTQ format, and pipe the resulting SAM file into Ceph.

Finally, Table 6.1 shows that, by overlapping I/O with computation in meaningful-sized pieces, the performance of Persona is nearly identical to SNAP and CPU bound in three very different storage configurations.

6.2.2 Thread Scaling

Figure 6.2 shows the scalability of standalone SNAP compared to Persona as a function of the number of provisioned aligner threads on the 48-core server. The experiments were measured on the RAID0 configuration so that SNAP has enough I/O bandwidth. For SNAP, Figure 6.2 shows clearly: (1) a near-linear speedup for up to 24 threads, corresponding to the 24 physical processor cores of the server; (2) that, beyond 24 cores, the 2nd hyperthread increases the alignment rate of a core by 32%. At 48 threads, contention between I/O scheduling and computation causes a drop in SNAP's performance. Persona is less sensitive to operating system kernel thread scheduling decisions due to PTF's pipelining capability and overlap of computation with I/O.

As we see in our single node evaluation, the dataflow framework does have a small overhead. Every core on the server is fully occupied nearly 100% of the time. The TensorFlow framework requires few threads to execute nodes' kernels, since most of our dataflow nodes are relatively short running and are only executed when the upstream and downstream gates of a stage have an available feed and buffer space, respectively. Otherwise, these threads sleep and allow the aligner threads to utilize all cores. This shows another advantage of coarse-grain work division in the dataflow framework: little time is wasted on overhead functions. Throughput per thread used on a single node scales linearly with two different slopes (Figure 6.2); after the 24 physical cores are used, linear scaling slows as hyperthreads begin sharing core resources. SNAP scales slightly better as each thread uses slightly fewer core resources. At 48 threads, SNAP's performance drops off due to contention, as one logical core is needed to service the

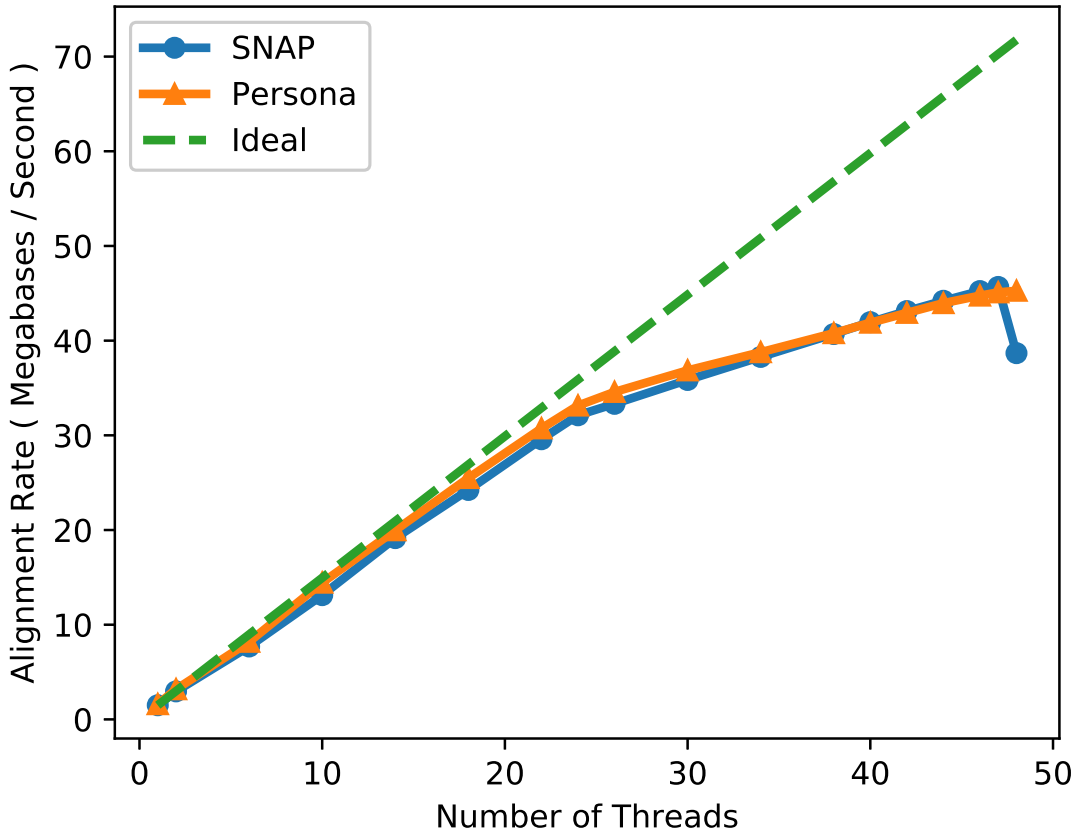


Figure 6.2 – Thread-scaling on single-node align app

I/O threads. We expect that this linear trend would continue with an increasing number of cores.

6.2.3 Sorting Performance

We compare Persona in sorting performance to Samtools [80] and Picard [24], standard utilities for sorting SAM/BAM files. Table 6.2 shows the results when configuring Samtools to use all 48 cores available. Picard does not have an option for multithreading. Samtools requires sorting input in BAM format; we include both sort and sort + conversion times. Persona can directly process aligned results in AGD, performing up to 2.32× times faster than Samtools when considering the file conversion time.

6.2.4 Conversion and Compatibility

To support existing sequencer output formats and other tools that have not yet been integrated, Persona can import FASTQ and export BAM formats at high throughput. From the local

Tool	Time	Speedup
Persona	556 sec	1.0×
Samtools	856 sec	1.54×
Samtools w/ conversion	1289 sec	2.32×
Picard	2866 sec	5.15×

Table 6.2 – Dataset Sort Time in Seconds, Single Server

filesystem, Persona can import FASTQ files into AGD datasets (with compressed data blocks in each chunk file) at 360 MB/s. The main limitation is the linear scan through the input FASTQ file in order to parse the format and the character boundaries between records and fields within each record. The 48 cores provide more than adequate capacity to compress and write each AGD chunk concurrently with the parsing process.

6.3 Scale-Out Single-Pipeline Performance

In this section, we evaluate Persona's alignment application for its scale-out performance. As discussed in §5.5, the alignment application contains all of its application logic in a single local pipeline. This is due to the independent nature of the underlying operation: aligning a single record requires no access to any other record. As a result, the application is an ideal benchmark for determining the scale-out behavior of Persona: any deviation from a linear increase in throughput defines the overhead imposed by PTF on the Persona alignment application.

Figure 6.3 shows the scaling behavior of the alignment application as the number of machines scale up. We include an ideal scaling line for comparison by multiplying the single-machine rate as the base rate for maximum scaling. This figure includes a maximum of 32 machines, additional machines beyond the cluster size of 20 machines used for the rest of this evaluation, in order to best evaluate the scaling behavior. The maximum alignment rate achieved (with 32 machines, each running one local alignment pipeline) is 1354 megabases/second.

Figure 6.3 shows that the alignment application scales linearly with the number of machines. This is due to the coarse granularity of work (*i.e.*, partition) distribution from the application's global pipeline to each local pipeline. The data distributed to each local pipeline by the first global gate is a set of chunk files to read and align; the amount of work necessary to align all records in each AGD chunk eclipses the work that Persona or PTF must do to distribute and partition work amongst the local pipelines. We anticipate that this scaling will increase until resource shared between all local pipelines becomes saturated, such as the network or the Ceph storage system.

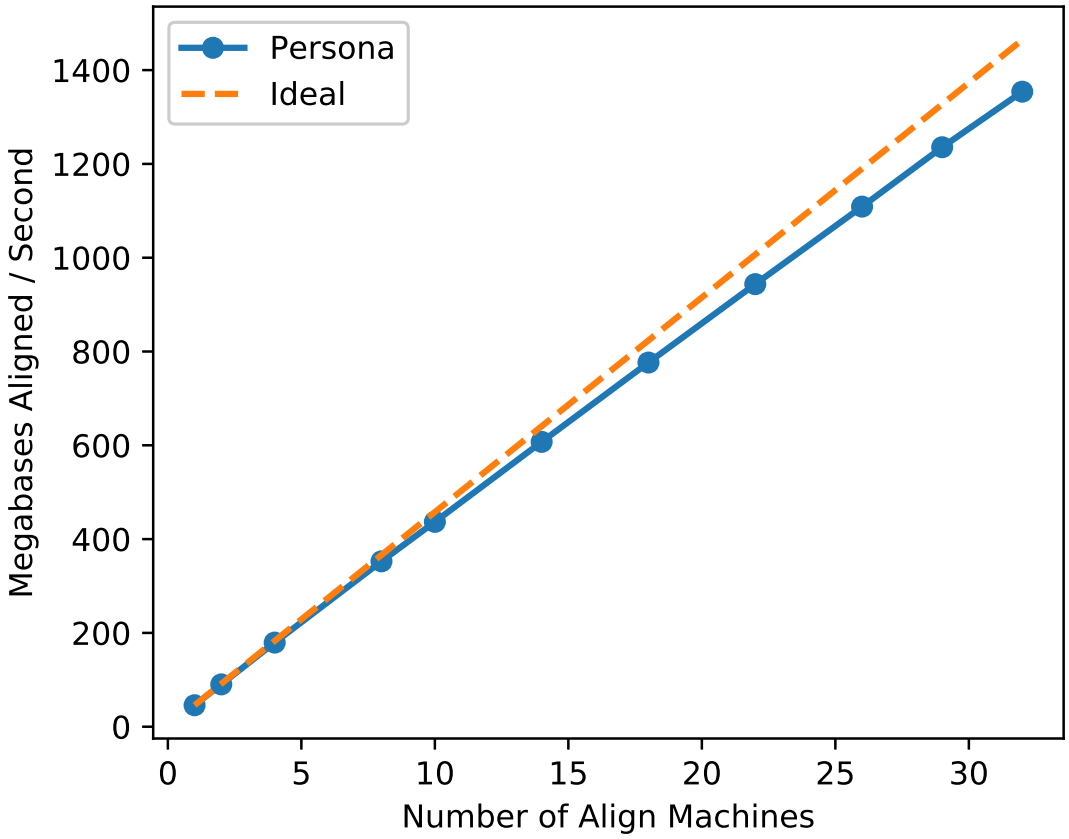


Figure 6.3 – scale-out results of only alignment

6.4 Scale-Out Multi-Pipeline Performance

In this section, we evaluate the fused align-sort pipeline on our cluster of machines. In contrast to the alignment application evaluated in §6.3, the fused align-sort application contains aggregate operations both within the local pipelines and between the two phases (align-sort and merge). This creates more complex patterns of resource usage and interaction between concurrent requests.

For this section, we use two different sets of AGD datasets as inputs. The full datasets contain a full human genome as described in §6.1. The small datasets contain a partition of 10% of the records in a full dataset (23 million single-end 101-base reads, instead of 223 million). We compare the performance of the fused align-sort application when processing each of these datasets to examine how the dataset latency affects the balance between the two phases of the application, in terms of allocated resources. This section makes additional comparisons between the application performance in both types of datasets, as measurements of application throughput and resource usage patterns are independent of the request size.

The experimental configurations for both the full and small dataset are identical except for the local merge pipeline: the small dataset configurations use the synchronous read phase whereas the full dataset configurations use the lazy read phase. Recall from §5.6.2 that the standard (*i.e.*, synchronous) read phase issues a synchronous read to the Ceph storage system and fills a buffer with the result (*i.e.*, the chunk file) before enqueuing the resulting resource into its downstream gate. In contrast, the lazy (*i.e.*, asynchronous) read phase uses the asynchronous Ceph operations to pre-fetch small regions of the requested chunk file such that only the data corresponding to the current record must be kept in memory. This lazy read is necessary due to the large footprint of the datasets: with each dataset taking up 51 GB in memory, multiple merge stage replicas needed to saturated a local merge pipeline, and at least one dataset buffered upstream of the merge phase to ensure the merge stages never have to wait for data, the overall memory footprint would exceed that of our machines were a synchronous read strategy to be used. When instrumenting the lazy iterators used in the merge phase, we never observed that the merge operation blocked while awaiting asynchronous read operation. This is due to the fact that we configured the lazy iterators to cache two regions and reissue the asynchronous read operation for the next block as soon as the current one becomes exhausted.

The experiments in this section are performed running the fused align-sort application as an indefinitely-executing service. For each experiment, the Persona application is started in its given configuration and waits until the PTF runtime establishes connectivity across all machines in the cluster; it then starts processing requests as they arrive. The experiment then starts a fixed number of clients, each of which contain a request to align and sort a dataset (full or small). Concretely, the request contains a batch with a feed type containing an AGD chunk name, which the application uses to construct the name for each column's corresponding chunk file, and a Ceph namespace, which is used to isolate datasets within the Ceph object pool. Each client submits its request (*i.e.*, batch) to the Persona application, waits for the result, and then resubmits the request indefinitely. After some warm-up period (*e.g.*, when a component of the pipeline, such as the merge pipeline's throughput, becomes fully saturated, and when resource pools in the local pipelines no longer create new resources), we take measurements for our experiments.

- Throughput is measured as a sampled rate at the end of the merge pipelines (*i.e.*, at the end of the overall computation).
- Latency is measured as the end-to-end latency of a client's request (*i.e.*, corresponding to the batch it submits to the pipeline). The steady state measurements collect data over a post-warmup time window of operation. In this case, the average latency includes only clients whose requests begin and end within the interval.
- Partition latency is measured exclusive of I/O for each pipeline. In order to measure the latency with which each local pipeline processes a partition, we exclude the read and write phases that are at the beginning and end of each pipeline, respectively. This is to accurately measure how well PTF is able to distribute work to the local pipelines

for computation and how well the core computation of each pipeline can overlap with I/O, as the focus of this pipeline is computation and not I/O. The partitions are of size 10 for the fused align-sort pipeline (*i.e.*, 10 chunks that, after sorting, form a single large intermediate chunk) and contain all chunks for use in the merge pipeline (see §5.6.1 for details on the merge operation).

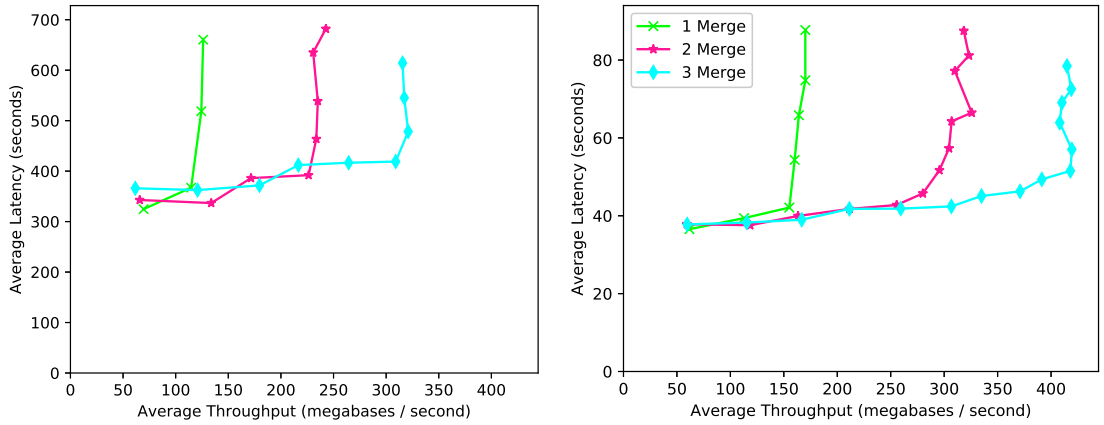
6.4.1 Benefits of Pipelining

One of Persona's main benefits as a unified framework is that it can pipeline requests across multiple stages and pipelines. Within the bounds of the application logic, different feeds from concurrent batches (each corresponding to different client requests) can be processed simultaneously to increase application throughput. Persona adds concurrent batches by increasing the number of open batches in the fused align-sort global pipeline (§5.7). When no component is saturated, increasing the number of open batches should increase the application throughput (in megabases per second) with a disproportionately minimal increase in latency, if any. When one component becomes saturated (*i.e.*, when a local pipeline becomes saturated), the throughput should be restricted by that component, with any additional open batches increasing the overall request latency.

We demonstrate the benefit of pipelining requests by varying the number of open batches on a fixed hardware configuration. Specifically, we fix the number of local pipeline replicas for both of the fused align-sort application's phases (align-sort and merge) and then measure the average throughput and request latency over a period of time. The maximum number of merge pipelines we configure ranges from 1 to 3, with 3 being the maximum we can saturate on our cluster of 20 machines. The number of local fused align-sort pipeline replicas is fixed to the maximum available on our cluster for the maximum number of local merge pipelines, in this case 17; this ensures that the merge phase (*i.e.*, the total throughput of all replicas) is the saturating component once an adequate number of open batches is configured. The variable adjusted is the number of open batches.

Figure 6.4 shows the results for this pipelining experiment. We then increment the number of open batches, starting at 1 for each series and incrementing by 1 for each successive data point. Both of these results (for small and large datasets) show that each additional open batch increases the overall throughput until a hardware component becomes saturated. After this point of saturation, additional open batches must queue in the buffer of the upstream gate to await processing.

In the case of full datasets in Figure 6.4a and 6 open batches, the configuration with 3 merge pipelines is able to achieve 321 megabases/second in its maximal configuration (*i.e.*, 17 fused align-sort pipelines), an increase of 4× over the 1 fused align-sort pipeline configuration with a 0.13× increase in request latency.



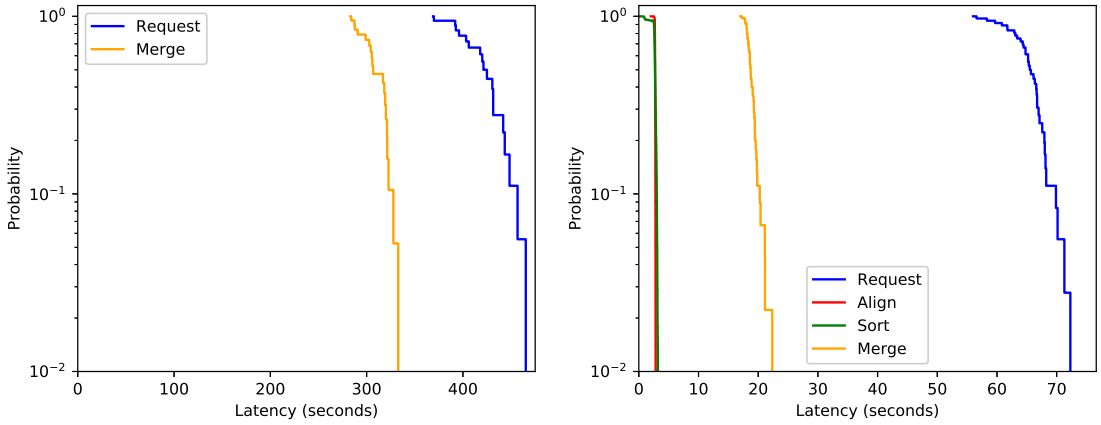
(a) Full datasets

(b) Small datasets

Figure 6.4 – Latency vs. throughput for small and full datasets

The differences between the small and full dataset results (in Figure 6.4b and Figure 6.4a, respectively) are due to the size of the datasets relative to the memory available on our cluster machines. The latency of a single request is as expected: the small dataset results are approximately 10% of the latency of the full dataset results, proportional to the difference in size. However, there is a difference between the two configurations with respect to throughput, which should be independent of the size of the request as it is measured as an aggregate rate of processing by the merge stage. The difference is due to the available memory for the machine running the local merge pipeline; with small dataset requests, the merge pipeline can be configured with 3 merge stage replicas (*i.e.*, 3 concurrent merge operations), which produces merged chunks into the downstream gate such that the subsequent compression stage saturates the CPU, while upstream stages can read several pending datasets into memory such that the merge operations never await a feed (*i.e.*, a dataset). With full dataset requests, the amount of memory used by the pipeline limits the operation to 2 merge replicas, with one pending dataset, even when using the lazy iterators to avoid reading full datasets into memory. We anticipate that the disparity between these two dataset sizes will disappear when run on machines with more memory resources.

To further evaluate the ability of PTF to overlap I/O and computation within each pipeline, we examine the latency of each feed as it is processed within the pipeline. Gates in both global and local pipelines should be constrained in Persona's architecture such that a saturated component in each pipeline bounds the throughput. PTF's mechanisms to bound resource utilization should respond to this throughput limit by controlling the feed buffering between adjacent gates and pipelines. In turn, this bounding should ensure that the variability of latency for processing any given feed is low. As mentioned in the beginning of this section, the instrumentation to measure these latencies does not include any I/O. If a feed latency measurement contains an I/O stage (read, write, or both), the begin or ending of the latency measurement is exclusive of any I/O latency component.



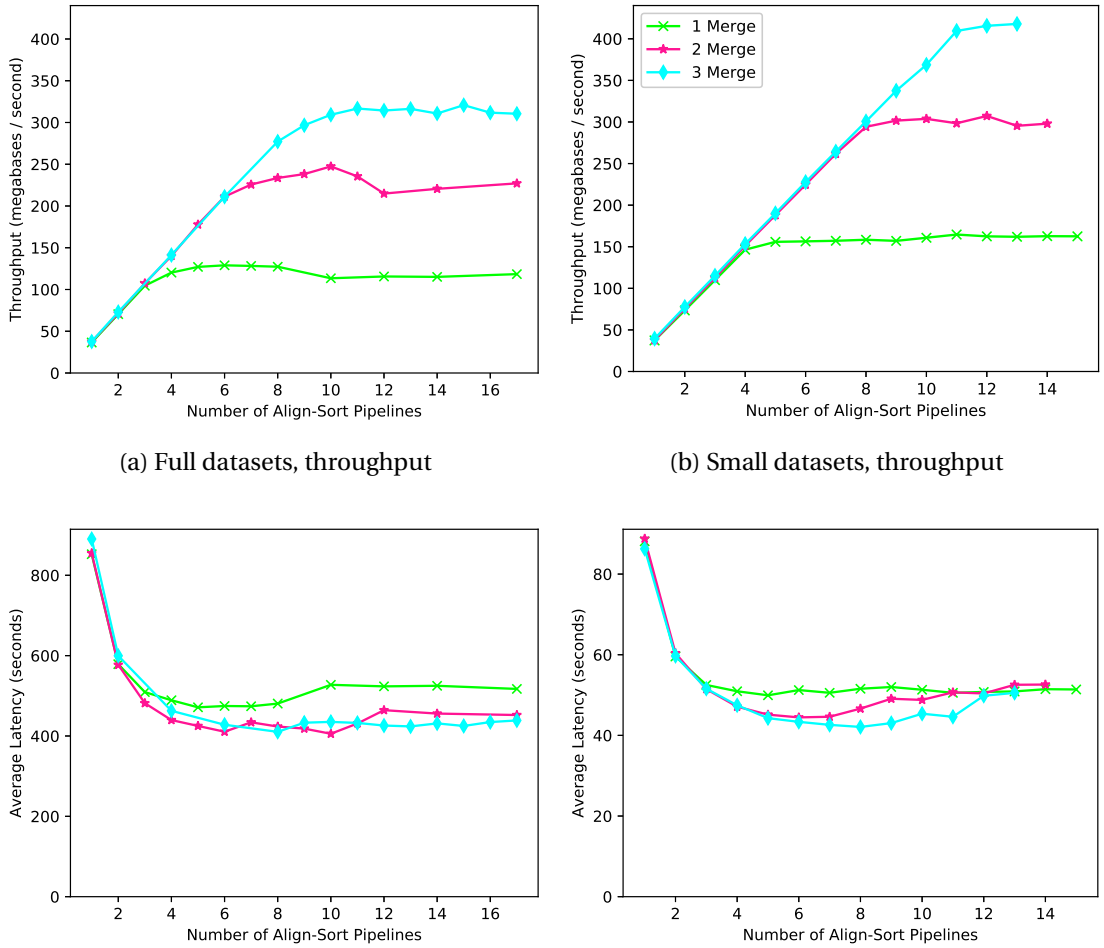
(a) Full datasets

(b) Small datasets

Figure 6.5 – Latency CCDF for full (6.5a) and small (6.5b)datasets.

Figure 6.5 shows the complementary CDF (CCDF) of feed latencies for both pipelines for both full and small datasets. Both of the configurations for large and small datasets are informed by the prior results, *i.e.*, from configurations that are as close to saturation as possible based on the number of open batches. Specifically, full datasets are configured with 7 open batches and small datasets are configured with 12 open batches. Both configurations use 3 merge pipelines and 17 fused align-sort pipelines and taken from the steady state of single executions. Figure 6.5a and Figure 6.5b show the merge and end-to-end request latencies for full and small datasets, respectively. In addition, Figure 6.5b shows the align and sort feed latencies; these latencies are similar to those on the full dataset configuration, but are omitted due to the increased latency scale of requests. Note that it is coincidental that the align and sort phases exhibit similar latency for this configuration; it is an artifact of the datasets' AGD chunk size (100,000 records per chunk for all requests) and the grouping factor for the batching dequeue preceding the sort stage in the fused align-sort pipeline. The feed latency for the merge pipeline is disproportionately larger for the full datasets compared to the small datasets; although full datasets are 10× larger than the small datasets (by number of records and chunks), the full datasets incur a slightly higher latency to merge. This is because the merge node uses a heap data structure to merge the intermediate files, and the accesses to this data structure incur fewer CPU cache hits in the larger dataset. Whereas the small dataset only merges 23 intermediate files, full datasets merge 224. As a result, merging the full dataset incurs more cache misses due to the larger heap size.

These figures confirm that (1) Persona reduces the serial latency by overlapping different phases of the application across parallel local pipelines; (2) the flow-control and scheduling mechanisms of Persona minimize tail latencies well up to the 99^{th} percentile; (3) the end-to-end request latency shows greater variability than the component latencies, and is the result of the combined effects of barrier delays and out-of-order feed delivery of concurrent



(a) Full datasets, throughput

(b) Small datasets, throughput

(c) Full datasets, latency

(d) Small datasets, latency

Figure 6.6 – Scale-out behavior (throughput and latency) for the fused align-sort application as a function of the number of fused align-sort pipeline replicas, for 1, 2, and 3 local merge pipelines. Results are shown for both full (6.6a, 6.6c) and small (6.6b, 6.6d) datasets.

requests. Nevertheless, the mean request latency for full datasets is 421.8 seconds while the 99^{th} percentile is only 463.8 seconds.

6.4.2 Scaling Behavior

Persona should enable linear throughput scaling with additional hardware resources for a saturated pipeline component. Any given configuration will have at least one component that saturates a hardware resources (*e.g.*, CPU for alignment). When additional hardware resources are dedicated to this saturated component, such as additional local pipelines on new machines, the aggregate throughput for the application should increase linearly with the additional hardware resources. This trend should continue until another component becomes

the bottleneck, at which point throughput should be dictated by the other component's maximum throughput.

Figure 6.6 demonstrates the scaling behavior of Persona for both small and full datasets in terms of application throughput and end-to-end latency. The scale-out behavior is shown for 1 to 3 local merge pipelines and have a fixed number of open batches for each series that is sufficient to saturate the application given the evaluation in §6.4.1, but are near the saturation point in the latency-vs-throughput curve in Figure 6.4.[1]This figure shows that additional hardware resources (*i.e.*, fused align-sort local pipelines) increase throughput linearly until another pipeline component becomes saturated. This be observed by scaling the number of fused align-sort pipelines for a given merge scale (1 to 3), or by observing the differences between each of the merge series for a given align-sort pipeline scale.

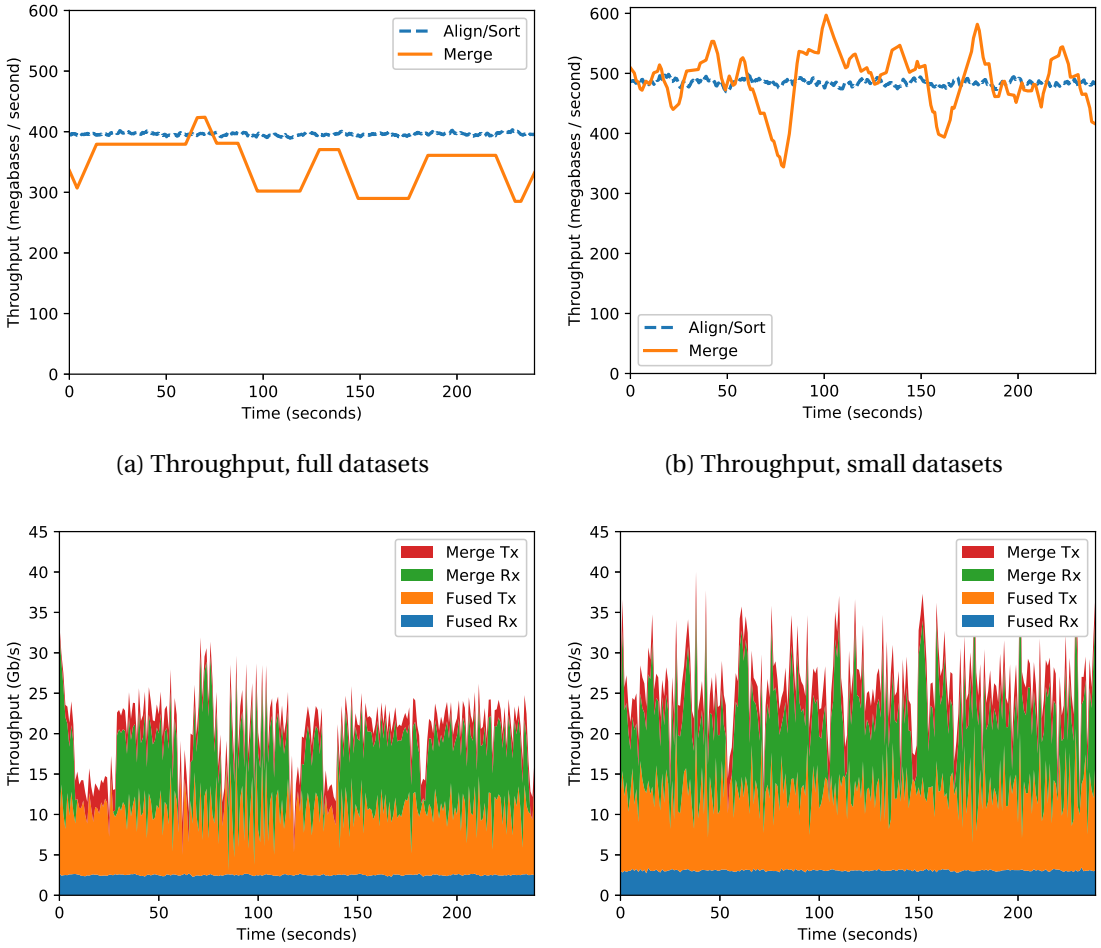
This evaluation highlights the importance of parameter tuning in Persona. The asymptotic behavior of the latency with additional machines is sensitive to many parameters in Persona, such as the number of open batches in the global fused align-sort pipeline and the number of open batches in the local merge pipeline. Additional open batches beyond the minimum sufficient number of open batches that is required to saturate the merge pipeline's aggregate throughput cause additional latency; with a high number of open batches (*i.e.*, beyond this minimum sufficient number), batches spend more time queuing in the global gate between the fused align-sort and merge local pipelines. We can see the effect of this in the 1-merge series in Figure 6.6c, where beyond 8 fused align-sort pipelines there is an increase in latency (3 open batches are configured for this series). This is a difficult tradeoff to make when merge latency is high; additional open batches decrease the probability that the merge node must wait for input from upstream stages, but at the same time it increases latency.

6.4.3 Benefits of Fusing Align and Sort

The fused align-sort application enables the user to configure fewer machines and perform less I/O compared to the baseline application. Recall from §5.6.4 that the baseline application contains three local pipelines that align, sort, and merge. Although it is possible to configure a baseline application (*i.e.*, in terms of number of machines / local pipeline replicas) to match the throughput and latency of a fused application, it requires more machines due to resource imbalance. Specifically, the fusion leads to a balanced use of each cluster node's compute and I/O resources, whereas the baseline pipeline has a mix of nodes that are either CPU-bound (the aligners) or I/O-bound (the sort nodes).

Both the fused align-sort pipeline and the baseline's align-only local pipeline are bound by the align stage. The sort stage is relatively inexpensive and takes advantage of the fact that the data is already read and decompressed into buffers to perform the alignment operation: a tuned configuration dedicates 47 aligner threads in the align-only case and 45 threads

[1]Small datasets use 4, 8, and 12 open batches for 1-, 2-, and 3-merge configurations, respectively. Full datasets use 3, 5, and 7 open batches for 1-, 2-, and 3-merge configurations, respectively.



(a) Throughput, full datasets

(b) Throughput, small datasets

(c) Ceph I/O, full datasets

(d) Ceph I/O, small datasets

Figure 6.7 – Fused runtime experiments for small (6.7b, 6.7d) and full (6.7a, 6.7c) datasets during a 4-minute period of steady-state operation, *i.e.*, not including warm-up periods for the application.

in the fused align-sort case. The minor reduction in throughput of the node is more than compensated by the reduction of 12% in aggregate I/O and the elimination of dedicated sort nodes. Furthermore, fusing aligning and sorting leads to more balanced use of CPU and NIC resources of each node of the cluster; fewer hardware resources are stranded on different machines due to the inherent architecture of the application.

Figure 6.7 shows the steady-state aggregate throughput and I/O of the fused align-sort application for small and full datasets. These figures demonstrate that PTF enables Persona to overlap I/O and compute throughout the pipeline. The I/O rate shows the aggregate I/O rate measured at the NIC; the aggregate is computed across all local pipelines of the same type. This represents all networking communication performed by each machine, but is primarily the I/O with the Ceph storage system; no other application is colocated with the

Persona application. The application throughput shows the aggregate throughput of each of the pipelines (align-sort and merge). Both the I/O and throughput rates are averaged over a 5-second window and sample at 1-second interval rates in order to reduce noise.

The throughput between the small and full dataset configurations (in Figure 6.7b and Figure 6.7a, respectively) show the difference in throughput due to the use of lazy iterators and fewer merge stage replicas within the merge local pipeline for full datasets. This highlights the need to choose hardware configuration carefully for maximum throughput; due to the inability to store an adequate number of full datasets in memory, the full dataset configuration must sacrifice throughput to ensure that the application does not get killed due to memory exhaustion.

6.5 Conclusion

This chapter has presented an evaluation of Persona and its underlying framework, PTF. We demonstrated that Persona optimizes application performance on a single machine by delegating the coordination of multiple independent subcomponents into different PTF stages such that PTF can effectively pipeline these operations; this architecture avoids inefficient and ad-hoc coordination between subcomponents, which leads to performance overheads in existing applications (such as the SNAP aligner). Persona combines this single-machine efficiency with scale-out pipelined request processing within the same framework (PTF). The same mechanisms (*i.e.*, PTF's pipeline abstraction) enable not only efficient single-machine operation, but are also used to distributed tasks (feeds) to multiple machines using Persona's hierarchical pipeline scale-out architecture. We demonstrate that Persona can scale out a single-pipeline workload (alignment) as well as a multi-pipeline workload (fused align-sort, *i.e.*, alignment followed by dataset merge sort), in the latter case saturating the hardware resources available in our cluster.

7 Related Work

This chapter discusses both the existing work from which this dissertation draws upon as well as contemporary work in bioinformatics workflows. The contributions of this thesis are inspired by existing work from several fields in computer science and existing techniques in cloud computing frameworks and file formats. Other efforts have been made prior to and concurrent with our work in the fields of bioinformatics workflows and the acceleration of individual workflow applications. We discuss both of these aspects in this chapter and how AGD, PTF, and Persona compare to these related works.

The contents of this chapter proceed as follows:

- §7.1 discusses existing and contemporary file formats to AGD, including formats that are specific to bioinformatics and more general formats that have some architectural aspects (*e.g.*, columnar design). We compare these formats to AGD to highlight some of its unique characteristics.
- §7.2 discusses contemporary works that address that, like Persona, address the performance issues in bioinformatics workflows and applications.

7.1 File Formats

The AGD format incorporates many of its design aspects from existing and contemporary formats. We discuss these formats and how they compare to AGD's features.

7.1.1 Existing Formats

The original text-based formats for storing bioinformatics discussed in §2.2.3 are the industry standard in bioinformatics. FASTQ [33] and its predecessor FASTA [104] are used to store genomic datasets as the initial workflow input; FASTQ files, typically compressed with `gzip` [42], are used to store the output of sequencing machines while FASTA files typically hold the

reference genomes, which do not require the quality scores needed by the alignment process. SAM [80] files store the output of post-alignment workflow phases, including sorting and any post-processing (§2.2.2); its binary equivalent, BAM, is a blocked-format of the SAM format with a few additional compression features, such as sequence and quality score encoding. A BAM file is often accompanied by a BAM index file, which enables random access into the BAM file.

The AGD format can supplant these existing formats due to its backward compatibility. AGD datasets contain semantically identical information to each of these existing formats; each field present in an existing format can be represented as a field in AGD. Due to this commonality between different existing formats, AGD can unify all storage formats to provide a common format.

The AGD format's improved I/O characteristics make it a much more viable option than traditional formats for scale-out processing. This is due to its chunked, columnar design, which breaks up monolithic datasets into smaller chunks for distributed processing while minimizing unnecessary I/O by read and writing only the minimal necessary columns for the operation.

7.1.2 Columnar File Formats and Data Storage Systems

Many database management systems (DBMS) are built on a column-oriented data storage architecture or have an option to do so. Beginning with the RAPID system [122], column-oriented DBMS applications have been a popular choice for users that want a data storage system that has the benefits of column-oriented data (*i.e.*, column projection) while being able to manage concurrency and metadata modification in order to provide data integrity guarantees. Such systems include IBM's SCSS [99], MonetDB [20], and C-store [114]. Google's Bigtable [30] is a popular recent iteration of this architecture, inspiring contemporary systems such as Apache Druid [13], Apache HBase [8], and Apache Kudu [14]. These are general DBMS systems that own and manage all data; by comparison, AGD provides many fewer guarantees about data integrity, relying on correct implementation within applications that write AGD datasets to maintain data integrity. AGD chooses this simple approach in order to avoid a bottleneck of a complex storage system, thereby increasing application throughput; no complex locking schemes or transaction semantics are required. Most bioinformatics workflows store the input dataset before processing for data provenance; any error can be recovered from by executing the workflow again.

Columnar file formats contain a single dataset oriented on a general storage system. These formats eschew the complexity of a DBMS to contain a single dataset and are stored on a traditional storage system, such as a file system. The Hierarchical Data Format (HDF) [50], most recently HDF5, stores data in a single file that uses metadata to describe the layout of contiguous chunked sections of columns. HDF is a common format in scientific applications such as astronomy and meteorology. Apache Parquet [16] and Apache Orc [15] are similar

formats that are widely used in cloud computing applications. Similar to HDF and Google's Dremel [91], Parquet and Orc use the columnar data format and a nested schema that is stored in columns within the file along with metadata. AGD draws upon some design concepts from these columnar formats, but it makes some design decisions that tailor it to bioinformatics. HDF5 is a complex format with a large codebase with performance issues; researchers in the field of astronomy created FITS [128] and its successor, the Advanced Scientific Data Format (ASDF) [60], in order to provide a viable alternative to meet their needs. Neither it nor Parquet are multi-file formats: concurrent reading or modification must be coordinated by a single entity (*e.g.*, a cloud computing application task to serialize updates) or support for concurrent accesses must be provided by the storage system itself. Neither format provides schema updates: adding a column would require an entirely new dataset to be created. Due to the specific needs of bioinformatics applications, in particular how they communicate data between each other in a workflow, we developed AGD to meet these specific needs.

7.1.3 Proposed Bioinformatics Formats

Several file formats have been proposed to replace the legacy file formats (§2.2.3). The ADAM framework [88, 101] proposes a new format based on Apache Parquet and Apache Avro [12] that unifies existing formats into a single schema. The MPEG-G [68] format, proposed by the Moving Picture Experts Group, is a similar commercial endeavor that includes not only a data storage format for genomic data, but also a data streaming format for network transmission and a selective encryption scheme. MPEG-G also contains encoding and compression options for various types of data, such as reference-based compression of reads and entropy-based quality score encoding. AGD, like these other formats, unifies existing formats into a single format, but has advantages due to its chunked architecture. Due to its use of Parquet, ADAM is limited by the constraints mentioned earlier in §7.1.2. MPEG-G was developed concurrently to the work in this dissertation (*i.e.*, AGD) and did not have a complete specification or reference implementation until after the publication of work contained herein.

7.2 Bioinformatics Applications and Workflow Managers

Current academic and industrial efforts seek to improve the performance of the existing ecosystem of bioinformatics applications and workflows. From single applications to workflow management systems to complete unified frameworks, several different categories of related work can be compared to the contribution outlined in this dissertation. In this section, we discuss the research efforts specific to bioinformatics that seek to improve performance in part of the broad ecosystem.

7.2.1 Distributed Bioinformatics Applications

Distributed alignment has been explored before, for example CloudBurst [109] and CrossBow [75], which use Apache Hadoop [7] to organize invocations of existing tools. They also find that the problem scales linearly and that distribution can result in significant speedups. CloudBurst reports 7 million reads aligned to one human chromosome in 500 seconds using 96 cores (5256 bases aligned per second per core), however a direct performance comparison is difficult because the alignment algorithm is different, the read size is different (36 base pairs versus our 101), and the cluster architecture and CPU were different. Cloud-Scale BWAMEM [31] is a distributed aligner that can align a genome in ~80 minutes over 25 servers, but requires different file formats for single (SAM) or distributed computation (ADAM). ParSRA [59] shows close to linear scaling using a PGAS approach, but relies on FUSE to split input files among nodes. Eoulsan [70] uses MapReduce to perform several workflow steps and supports different aligners. Pmap [63] uses MPI to scale several different aligners across servers and claims linear scaling.

More recent efforts integrate the bioinformatics applications as callable libraries in the cloud computing frameworks; in this approach, frameworks such as Hadoop and Apache Spark [133] call into existing applications using the Java Native Interface (JNI) and manage the movement of data at a finer granularity. Examples such frameworks include Halvade [37], which interfaces with GATK [90] on top of Hadoop, and the Spark-based applications SparkBWA [3] and SparkGA [97], which parallelize GATK on top of Spark.

Other efforts include SAND [94], where alignment is divided into stages for reads, candidate selection and alignment on dedicated clusters using algorithms similar to BLAST. There have also been efforts to distribute BLAST computation itself [105]. Others have shown that aligning reads to a reference genome scales linearly [62]. merAligner [53] implements a seed-and-extend algorithm that is highly parallel at all stages, but uses fine-grained parallelism more amenable to supercomputing systems rather than the clusters or data centers that Persona targets. GENALICE Map [119] reports 92 million bases aligned per second on a single machine, faster than even SNAP; however, it is a closed-source proprietary product.

In contrast to previous work, Persona and AGD provide a general high-performance framework that facilitates linear core and server scale out of not only alignment but many bioinformatics processes. Persona has negligible overhead, and does not restrict users to specific storage systems or parallel patterns. PTF's dataflow architecture can support different models of parallelism, while the Python API allows user composable pipelines. AGD provides scalable, high-bandwidth access to data. Both Persona and AGD are also extensible, making it easy to integrate new or existing tools and data schemas.

We point out that the majority of related work in scaling alignment or other genomics processing predominantly uses Hadoop or Spark frameworks over HDFS. These frameworks employ the principle of moving computation to where data resides, usually because query input data is small and the data backing the response computation is large and relatively fixed,

i.e., the ratio of reads to writes is high. In genomics however, the opposite pattern is true: input query data is large (*e.g.*, a patient's raw sequenced genome) and fixed backing data is small (*e.g.*, the reference genome). In a small-scale or research setting, the Spark/Hadoop model may well be an efficient and easy to use solution. However, we believe that in large-scale personalized health systems, where thousands of genomes must be processed and stored, this model quickly becomes too inefficient.

7.2.2 Workflow Managers

Workflow management systems coordinate the successive invocation of separate applications that constitute a bioinformatics workflow. The workflows are specified in a configuration (typically using a directed acyclic graph of data dependencies, such as in the prominent Common Workflow Language [34] (CWL)) and the workflow management system invokes applications based on the data dependencies. Workflow management systems enable researchers in many fields (*e.g.*, bioinformatics, astronomy, physics) to document the sequence of steps in the workflow in order to reliably reproduce experimental results. Galaxy [57] is an early workflow management system that remains a popular choice due to its feature set and graphical user interface, which enabled less technical users to create and execute workflows. Arvados [17], Rabix [71], Apache Airflow [11], and Toil [126] execute CWL workflows. Some systems, such as Cuneiform [23] and NextFlow [43], implement their own workflow description as a domain-specific language and seek to better-integrate existing data center systems, such as distributed storage and existing cloud computing frameworks.

Workflow management systems are limited in their ability to improve performance. In contrast with Persona, workflow management systems organize invocations, possibly in parallel, of existing bioinformatics applications; the applications invoked include the performance overheads previously discussed in §1.1.1. Without insight into the applications themselves, workflow management systems are unable to optimize the computation between successive applications. Persona's ability to decompose applications into a pipeline of invocations by encapsulating the core application logic into a pipeline or stage enables cross-application optimization, such as the fusing of alignment and the sort phases described in §5.7.

7.2.3 Cloud Computing Frameworks

ADAM: A few other research projects seek to provide a unified framework for bioinformatics. Similar to Persona's contribution of AGD, these projects provide a new file format (on disk or an in-memory representation) and a framework of composable workflow operations, which may be encapsulated existing applications (*e.g.*, an aligner) or a new application for an operation. The ADAM framework [88] from Big Data Genomics provides an integrated framework built atop Apache Spark. Featuring a custom format using Apache Parquet, this framework knits together existing applications and provides a unified format for all workflow

steps. It includes several tools built on its API, including the Avocado variant caller [100] and the Mango visualization tool [19]. The genomic programming framework (GPF) [82], also built on Spark, is similar to ADAM but lacks the extensive ecosystem of tools and applications.

Comparing these Spark-based unified frameworks to Persona presents a tradeoff. Persona eschews the JVM and Spark framework which both of these frameworks rely upon, minimizing the overhead associated with both. However, the overhead imposed by Spark could be desired for some use cases. The guarantees and abstractions provided by Spark have enabled a large number of applications to be built atop ADAM's API and file format. The ecosystem available from Spark as well as its data abstractions (*i.e.*, resilient distributed datasets) enables new applications to be built on the "narrow waist" of file formats and access mechanisms provided by ADAM. Persona sacrifices these robust but costly mechanisms in order to achieve higher throughput.

Nimbus: Nimbus [87] is a cloud computing framework written in C++ targeting analytic and scientific workloads. Optimizing the centralized scheduling architecture used by Spark, Nimbus includes the construct of execution templates to decrease scheduling overhead for repetitive tasks, such as those found in iterative algorithms in machine learning. Nimbus shows significant speed-up over Spark in comparable computations, achieving similar performance in a scale-out graphical simulation library to that of the library's hand-tuned MPI libraries. Though it does not specifically target bioinformatics workflows, it could be used to construct a scale-out workflow similar to the pipeline abstraction that PTF uses. Work on Nimbus was developed concurrently with PTF and Persona, and native cloud computing frameworks such as Nimbus may prove to be viable alternatives to TensorFlow as an underlying framework for scale-out bioinformatics applications.

8 Conclusion

This dissertation presented three contributions to computer science systems research that addressed some of the challenges of scaling out bioinformatics workflows. We first presented the Aggregate Genomic Data format (chapter 3), a new file format for storing genomic datasets. Its chunked, columnar design enables it to supplant the existing file formats due to its equivalent semantics while providing support for scale-out I/O operations. We then presented Pipelined TensorFlow (chapter 4), a framework based on TensorFlow that provides a pipeline abstraction to applications via its API and a runtime that executes these pipelines on heterogeneous hardware resources. It scales a single application from a single researcher's laptop to a large cluster of high-performance servers with only a few parameter changes (*i.e.*, placing pipeline components on different logical devices). Persona (chapter 5) ties PTF and AGD together to create scale-out bioinformatics workflows. These workflows run indefinitely as services that concurrently process requests in order to increase the utilization of the underlying hardware resources. As we demonstrated in the evaluation (chapter 6), Persona was able to leverage PTF and its request pipelining in order to saturate the resources of a cluster of machines while scaling out across additional machines. Our evaluation showed that Persona's pipelining was able to increase hardware resource utilization and application throughput with a negligible impact on latency below resource saturation.

8.1 Lessons Learned

The development of PTF and Persona required many developer-hours of effort to bring to fruition. We share some of the practical lessons we have learned during this process.

8.1.1 Using TensorFlow

TensorFlow was a necessary choice given the timeline of the work described in this dissertation. We considered existing cloud computing frameworks such as Spark [133] and Naiad [96], but were dissuaded from these frameworks when designing Persona due to overheads that we

decided were not acceptable. Although these frameworks have semantics that are much more amenable to the computation we would like to execute, they came encumbered with expensive features that we did not need and could not circumvent, such as use of the Java Virtual Machine and failure recovery mechanisms. Given these constraints, TensorFlow was the most viable candidate upon which we decided to build PTF and Persona.

Many of the design features of PTF were created in order to overcome the limited static dataflow model that underpins TensorFlow [2]. Recall from §2.4 that TensorFlow follows a set of rules that is akin to the tagged token dataflow architecture (TTDF) [18]. Each input is associated with a tag and is propagated through the dataflow graph based on the rules of static dataflow [41]: each feed processed by a TensorFlow graph will process exactly one resulting feed, with each node evaluating its inputs and propagating resulting values to downstream expressions. This is advantageous for machine learning (§2.4) because it is similar to the evaluation of a mathematical formula. However, this restricts TensorFlow's direct applicability for use as a drop-in replacement for a framework supporting analytics and "big data" workloads; these types of frameworks, such as Spark, contain semantics closer to what we would need for developing Persona.

PTF is a set of mechanisms to implement an adequate subset of features found in contemporary cloud computing frameworks without disrupting the TensorFlow runtime or library. Specifically, it implements the request tracking and aggregation semantics (*e.g.*, grouping subsets of a request's items into batches) using the metadata and gates. By using gates to interpret the metadata and apply these simple semantics, PTF does not disrupt the existing semantics of contiguous TensorFlow graphs, *i.e.*, a graph without any stateful elements like TensorFlow queues or gates. The static dataflow semantics that TensorFlow applies to each feed remain valid and enable PTF pipelines to run on the TensorFlow runtime, as each stage in a pipeline is merely a contiguous TensorFlow graph and a user thread to drive computation via the Python API. In fact, it is these static dataflow semantics that enable each gate to make a valid interpretation of each batch of feeds based on the metadata: no stage's graph may make any computation that would invalidate the metadata.

TensorFlow's runtime and API provide many important features for the development of PTF and Persona. The Python API enabled Persona to assemble complex pipelines by assembling a logical description of the application. The runtime components of Persona then start all of the processes necessary to run the application, send the logic description to each process, and start all components. As a result, no library or binary components had to be recompiled and redistributed in order to develop or troubleshoot Persona applications in most cases. This enabled fast iteration when developing Persona's applications. Given the application description, the TensorFlow runtime automatically manages the propagation of feeds between local and remote nodes in the graph, transparently adding network transfer nodes as needed. The dataflow propagation and execution of kernels by the TensorFlow runtime was foundational to Persona's operation and performance.

Future developments of cloud computing frameworks for bioinformatics may be better suited to other frameworks. Despite the benefits TensorFlow provided for the development of Persona, we believe that the semantics of TensorFlow may inhibit further development. Specifically, the static dataflow rules applied to each feed prevent TensorFlow from supporting a more general computation, *e.g.*, supporting a filtering operation that consumes a feed but does not emit resulting feed. The one operation in Persona that did not fit into the static dataflow rules was the merge stage, which took a single aggregate feed as input and output an unknown number of resulting feeds. The merge node circumvented the typical PTF and TensorFlow mechanisms by directly enqueuing feeds into the downstream gate. However, applying such custom workarounds to every operation would require significant engineering effort. Using a cloud computing framework that provides more amenable semantics as well as native performance, such as Nimbus [87] or the Timely Dataflow engine [118] (a successor to Naiad [96] written in Rust [89]). Given the availability of modern C++ frameworks for building distributed applications, future engineering effort may be better spent building a bioinformatics framework from the ground up instead of retrofitting features into TensorFlow.

8.1.2 Application Inefficiencies

It is exceedingly difficult to produce a bioinformatics application that is suited for the cloud computing ecosystem. Even large projects produced by teams of expert developers, such as TensorFlow, take months and even years to achieve reliable and predictable performance. This predicament is further exacerbated when less experienced developers attempt this feat. When examining existing bioinformatics applications, we found repeating patterns of inefficiencies:

- Runtime overheads of the Java Virtual Machine: as discussed previously in this thesis (§2.3.3), the JVM includes runtime overheads. The most common we observed was boxing and unboxing of data, where an underlying native numeric type (*e.g.*, a floating-point value) is contained as a member of a corresponding Java type (*e.g.*, a Java `Double` instance) which proxies all operations.

- Inefficient use of memory: applications written in a native language (*i.e.*, C or C++) contained both memory leaks and, more often, inefficient memory allocation patterns. For example, different frameworks would often use a data structure from the C++ Standard Template Library (STL) that allocates heap storage for each item. When these STL containers are allocated for a short-lived duration (*e.g.*, to hold some values in each iteration of a tight loop), the overhead of heap allocation and deallocation can dominate the execution time.

- Inefficient memory access patterns: native and JVM-based applications contain inefficient memory access patterns, such as random traversal of memory. This was often hidden through data structures where the algorithmic complexity was unnecessarily large or the constant factor of the complexity was large (*e.g.*, the large factor of traversing

a linked list versus an array, due to the random memory locations of each linked list node).

The SNAP aligner [132], which Persona uses as its aligner, is a notable exception; although it is written in C++, we did not observe any inefficient memory utilization or access patterns when profiling it. However, this performance comes at a cost: the code is difficult to understand, as many of the techniques used to achieve its performance create a codebase that is difficult to decipher. Large function bodies with a significant number of parameters may enable a compiler to more easily optimize the code, but it obscures the functionality to a developer.

A framework such as Persona that implements certain application components is an important step to building high-performance bioinformatics workflows. Although it is still possible to unintentionally misuse STL containers or inefficiently traverse data structures, adding a new operation only requires a handful of new nodes in Persona. Threading and overlapping computation with I/O are delegated to the TensorFlow runtime, enabling a developer to focus on the relatively smaller amount of code for his or her feature. The ADAM framework [88] is another example of this type of framework; its provided library of dataset iterators and abstractions that allow developers to focus on new features instead of reimplementing a multithreaded runtime.

8.2 Conclusion

The goal for Persona was ambitious: combine existing bioinformatics applications into a new unified framework to enable scale-out processing. We decomposed these applications into their core functionality; developed AGD, a new file format to supplant existing formats; and added support for both of these features into TensorFlow. We combined these aspects into Persona using PTF, a framework built on TensorFlow that supports the construction and indefinite execution of bioinformatics workflows that concurrently process requests. We developed several scale-out Persona applications, including alignment, alignment and sorting, and the optimized fused align-sort application. Our evaluation demonstrated that Persona can scale out applications across multiple machines to the point of resource saturation. Persona achieves this scale-out performance without any additional changes to the application logic due to PTF's architecture of stage and pipeline replication.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pages 265–283, 2016.

[2] M. Abadi, M. Isard, and D. G. Murray. A computational model for TensorFlow: an introduction. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL@PLDI)*, pages 1–7, 2017.

[3] J. M. Abuín, J. C. Pichel, T. F. Pena, and J. Amigo. Sparkbwa: speeding up the alignment of high-throughput dna sequencing data. *PloS one*, 11(5):e0155461, 2016.

[4] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *PVLDB*, 6(11):1033–1044, 2013.

[5] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403 – 410, 1990.

[6] Amazon, Inc. Amazon glacier pricing. https://aws.amazon.com/glacier/pricing/. Accessed: 16-03-2019.

[7] Apache. Hadoop. https://hadoop.apache.org/.

[8] Apache. HBase. https://hbase.apache.org/.

[9] Apache. Phoenix. https://phoenix.apache.org/.

[10] Apache. Pig. https://pig.apache.org/.

[11] Apache Airflow. http://airflow.apache.org/.

[12] Apache Avro. http://avro.apache.org/.

[13] Apache Druid, Incubating. http://druid.io.

[14] Apache Kudu. http://kudu.apache.org/.

[15] Apache Orc. http://orc.apache.org/.

[16] Apache Parquet. http://parquet.apache.org/.

[17] Arvados. Arvados, Open Source Big Data Processing and Bioinformatics. https://arvados.org/.

[18] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. Computers*, 39(3):300–318, 1990.

[19] Big Data Genomics. Mango: a scalable genome browser. https://github.com/bigdatagenomics/mango. Accessed: 25-02-2019.

[20] P. A. Boncz. *Monet: A next-generation DBMS kernel for query-intensive applications*. Universiteit van Amsterdam [Host].

[21] J. K. Bonfield and M. V. Mahoney. Compression of FASTQ and SAM format sequencing data. *PloS one*, 8(3):e59190, 2013.

[22] D. Borthakur et al. Hdfs architecture guide. *Hadoop Apache Project*, 53:1–13, 2008.

[23] J. Brandt, M. Bux, and U. Leser. Cuneiform: a Functional Language for Large Scale Scientific Data Analysis. In *EDBT/ICDT Workshops*, pages 7–16, 2015.

[24] Broad Institute. Picard. https://broadinstitute.github.io/picard/. Accessed: 08-10-2016.

[25] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. In *DIGITAL SRC RESEARCH REPORT*. Citeseer, 1994.

[26] S. Byma, S. Whitlock, L. Flueratoru, E. Tseng, C. Kozyrakis, E. Bugnion, and J. R. Larus. Persona: A High-Performance Bioinformatics Framework. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, pages 153–165, 2017.

[27] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[28] E. Cerami. SAMtools: Primer / Tutorial. 2014.

[29] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI)*, pages 363–375, 2010.

[30] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data (Awarded Best Paper!). In *Proceedings of the 7th Symposium on Operating System Design and Implementation (OSDI)*, pages 205–218, 2006.

[31] Y.-T. Che, J. Cong, J. Lei, S. Li, M. Peto, P. Spellman, P. Wei, and P. Zhou. CS-BWAMEM: A Fast and Scalable Read Aligner at the Cloud Scale for Whole Genome Sequencing (Poster). *High Throughput Sequencing Algorithms and Applications (HITSEQ)*, 2015.

[32] J. Clarke, H.-C. Wu, L. Jayasinghe, A. Patel, S. Reid, and H. Bayley. Continuous base identification for single-molecule nanopore dna sequencing. *Nature Nanotechnology*, 4:265 EP –, 02 2009.

[33] P. J. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic acids research*, 38(6):1767–1771, 2010.

[34] Common Workflow Language. Common workflow language specifications, v1.0.2. https://www.commonwl.org/v1.0/.

[35] P. Danecek, A. Auton, G. Abecasis, C. A. Albers, E. Banks, M. A. DePristo, R. E. Handsaker, G. Lunter, G. T. Marth, S. T. Sherry, et al. The variant call format and vcftools. *Bioinformatics*, 27(15):2156–2158, 2011.

[36] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[37] D. Decap, J. Reumers, C. Herzeel, P. Costanza, and J. Fostier. Halvade: scalable sequence analysis with MapReduce. *Bioinformatics*, 31(15):2482–2488, 2015.

[38] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid Datacenter Scheduling. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, pages 499–510, 2015.

[39] P. I. Delgado Borda. Hybrid, job-aware, and preemptive datacenter scheduling. page 135, 2018.

[40] L. Deng. The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]. *IEEE Signal Process. Mag.*, 29(6):141–142, 2012.

[41] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture*, ISCA '75, pages 126–132, New York, NY, USA, 1975. ACM.

[42] P. Deutsch. GZIP file format specification version 4.3. *RFC*, 1952:1–12, 1996.

[43] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame. Nextflow enables reproducible computational workflows. *Nature biotechnology*, 35(4):316, 2017.

[44] M. A. Eberle, E. Fritzilas, P. Krusche, M. Källberg, B. L. Moore, M. A. Bekritsky, Z. Iqbal, H.-Y. Chuang, S. J. Humphray, and A. L. Halpern. A reference data set of 5.4 million phased human variants validated by genetic inheritance from sequencing a three-generation 17-member pedigree. *Genome research*, 27(1):157–164, 2017.

[45] M. A. Eberle, E. Fritzilas, P. Krusche, M. Kallberg, B. L. Moore, M. A. Bekritsky, Z. Iqbal, H.-Y. Chuang, S. J. Humphray, A. L. Halpern, S. Kruglyak, E. H. Margulies, G. McVean, and D. R. Bentley. A reference dataset of 5.4 million human variants validated by genetic inheritance from sequencing a three-generation 17-member pedigree. *bioRxiv*, 2016.

[46] J. Eid, A. Fehr, J. Gray, K. Luong, J. Lyle, G. Otto, P. Peluso, D. Rank, P. Baybayan, B. Bettman, A. Bibillo, K. Bjornson, B. Chaudhuri, F. Christians, R. Cicero, S. Clark, R. Dalal, A. deWinter, J. Dixon, M. Foquet, A. Gaertner, P. Hardenbol, C. Heiner, K. Hester, D. Holden, G. Kearns, X. Kong, R. Kuse, Y. Lacroix, S. Lin, P. Lundquist, C. Ma, P. Marks, M. Maxham, D. Murphy, I. Park, T. Pham, M. Phillips, J. Roy, R. Sebra, G. Shen, J. Sorenson, A. Tomaney, K. Travers, M. Trulson, J. Vieceli, J. Wegener, D. Wu, A. Yang, D. Zaccarin, P. Zhao, F. Zhong, J. Korlach, and S. Turner. Real-time dna sequencing from single polymerase molecules. *Science*, 323(5910):133, 01 2009.

[47] B. Ewing, L. Hillier, M. C. Wendl, and P. Green. Base-calling of automated sequencer traces usingphred. *Genome research*, 8(3):175–185, 1998.

[48] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *Proceedings of the 41st IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 390–398, 2000.

[49] J. Fietz, S. Whitlock, G. Ioannidis, K. J. Argyraki, and E. Bugnion. VNToR: Network Virtualization at the Top-of-Rack Switch. In *Proceedings of the 2016 ACM Symposium on Cloud Computing (SOCC)*, pages 428–441, 2016.

[50] M. Folk, A. Cheng, and K. Yates. HDF5: A file format and i/o library for high performance computing applications. volume 99, pages 5–33.

[51] M. H.-Y. Fritz, R. Leinonen, G. Cochrane, and E. Birney. Efficient storage of high throughput dna sequencing data using reference-based compression. *Genome research*, 21(5):734–740, 2011.

[52] E. Garrison and G. Marth. Haplotype-based variant detection from short-read sequencing. *arXiv preprint arXiv:1207.3907*, 2012.

[53] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. A. Yelick. merAligner: A Fully Parallel Sequence Aligner. In *Proceedings of the 29th IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 561–570, 2015.

[54] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–43, 2003.

[55] L. Gherardi, D. Brugali, and D. Comotti. A Java vs. C++ Performance Evaluation: A 3D Modeling Benchmark. In *Proceedings of the 2012 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pages 161–172, 2012.

[56] T. C. GLENN. Field guide to next-generation dna sequencers. *Molecular Ecology Resources*, 11(5):759–769, 2019/01/21 2011.

[57] J. Goecks, A. Nekrutenko, J. Taylor, and The Galaxy Team. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. 11(8):R86.

[58] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th Symposium on Operating System Design and Implementation (OSDI)*, pages 599–613, 2014.

[59] J. González-Domínguez, C. Hundt, and B. Schmidt. parsra: A framework for the parallel execution of short read aligners on compute clusters. *Journal of Computational Science*, pages –, 2017.

[60] P. Greenfield, M. Droettboom, and E. Bray. ASDF: A new data format for astronomy. 12:240–251.

[61] W. D. Gropp, E. L. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface, 2nd Edition*. Scientific and engineering computation series. MIT Press, 1999.

[62] S. Guo and V. Phan. A distributed framework for aligning short reads to genomes. *BMC Bioinformatics*, 15(S-10):P22, 2014.

[63] HPC Lab – OSU. Parallel Sequence Mapping Tool. http://bmi.osu.edu/hpc/software/pmap/pmap.html, 2016.

[64] R. Hundt. Loop Recognition in C++/Java/Go/Scala. In *Proceedings of Scala Days 2011*, 2011.

[65] Illumina, Inc. Illumina hiseq x 10 sequencing systems. http://www.illumina.com/systems/hiseq-x-sequencing-system.html. Accessed: 08-10-2016.

[66] International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. 409(6822):860.

[67] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 EuroSys Conference*, pages 59–72, 2007.

[68] White paper on the objectives and benefits of the MPEG-G standard. Standard ISO/IEC JTC1/SC29/WG11 N17468, International Organization for Standardization, Geneva, CH, October 2018.

[69] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.

[70] L. Jourdren, M. Bernard, M.-A. Dillies, and S. Le Crom. Eoulsan: a cloud computing-based framework facilitating high throughput sequencing analyses. *Bioinformatics*, 28(11):1542, 2012.

[71] G. Kaushik, S. Ivkovic, J. Simonovic, N. Tijanic, B. Davis-Dusenbery, and D. Kural. Graph theory approaches for optimizing biomedical data analysis using reproducible workflows. In *Pac. Symp Biocomput*, 2017. http://dx.doi.org/10.1101/074708.

[72] W. J. Kent, C. W. Sugnet, T. S. Furey, K. M. Roskin, T. H. Pringle, A. M. Zahler, and D. Haussler. The human genome browser at ucsc. *Genome Research*, 12(6):996, 2002.

[73] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream Processing at Scale. In *SIGMOD Conference*, pages 239–250, 2015.

[74] B. Langmead and S. L. Salzberg. Fast gapped-read alignment with bowtie 2. *Nature Methods*, 9:357 EP –, 03 2012.

[75] B. Langmead, M. C. Schatz, J. Lin, M. Pop, and S. L. Salzberg. Searching for SNPs with cloud computing. *Genome Biology*, 10(11):R134, Nov 2009.

[76] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3):1–10, 2009.

[77] C. Leary and T. Wang. XLA: TensorFlow, compiled. *TensorFlow Dev Summit*, 2017.

[78] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[79] H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[80] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, et al. The Sequence Alignment/map Format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[81] R. Li, C. Yu, Y. Li, T. W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[82] X. Li, G. Tan, B. Wang, and N. Sun. High-performance genomic analysis framework with in-memory computing. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 317–328, 2018.

[83] S. Liang. *The Java native interface: programmer's guide and specification*. Addison-Wesley Professional.

[84] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[85] E. R. Mardis. A decade's perspective on dna sequencing technology. *Nature*, 470:198 EP –, 02 2011.

[86] O. Mashayekhi, H. Qu, C. Shah, and P. Levis. Scalable, Fast Cloud Computing with Execution Templates. *CoRR*, abs/1606.01972, 2016.

[87] O. Mashayekhi, H. Qu, C. Shah, and P. Levis. Execution Templates: Caching Control Plane Decisions for Strong Scaling of Data Analytics. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, pages 513–526, 2017.

[88] M. Massie, F. Nothaft, C. Hartl, C. Kozanitis, A. Schumacher, A. D. Joseph, and D. A. Patterson. ADAM: Genomics formats and processing patterns for cloud scale computing. *University of California, Berkeley Technical Report, No. UCB/EECS-2013*, 207, 2013.

[89] N. D. Matsakis and F. S. K. II. The rust language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*, pages 103–104, 2014.

[90] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, et al. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Research*, 20(9):1297–1303, 2010.

[91] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Commun. ACM*, 54(6):114–123, 2011.

[92] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research*, 17:34:1–34:7, 2016.

[93] Microsoft Azure. Azure storage block blob pricing. https://aws.amazon.com/glacier/pricing/. Accessed: 16-03-2019.

[94] C. Moretti, A. Thrasher, L. Yu, M. Olson, S. J. Emrich, and D. Thain. A Framework for Scalable Genome Assembly on Clusters, Clouds, and Grids. *IEEE Trans. Parallel Distrib. Syst.*, 23(12):2189–2197, 2012.

[95] P. Muir, S. Li, S. Lou, D. Wang, D. J. Spakowicz, L. Salichos, J. Zhang, G. M. Weinstock, F. Isaacs, J. Rozowsky, and M. Gerstein. The real cost of sequencing: scaling computation to keep pace with data generation. 17(1):53.

[96] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 439–455, 2013.

[97] H. Mushtaq, F. Liu, C. Costa, G. Liu, P. Hofstee, and Z. Al-Ars. SparkGA: A Spark Framework for Cost Effective, Fast and Accurate DNA Analysis at Scale. In *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology,and Health Informatics*, pages 148–157, 2017.

[98] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[99] N. H. Nie. *SCSS, a User's Guide to the SCSS Conversational System*. McGraw-Hill Companies.

[100] F. A. Nothaft. *Scalable Systems and Algorithms for Genomic Variant Analysis*. PhD thesis, University of California, Berkeley, USA, 2017.

[101] F. A. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman, M. J. Franklin, A. D. Joseph, and D. A. Patterson. Rethinking Data-Intensive Science Using Scalable Analytics Systems. In *SIGMOD Conference*, pages 631–646, 2015.

[102] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 69–84, 2013.

[103] S. Palkar, J. J. Thomas, D. Narayanan, A. Shanbhag, R. Palamuttam, H. Pirk, M. Schwarzkopf, S. P. Amarasinghe, S. Madden, and M. Zaharia. Weld: Rethinking the Interface Between Data-Intensive Applications. *CoRR*, abs/1709.06416, 2017.

[104] W. R. Pearson. [5] rapid and sensitive sequence comparison with fastp and fasta. 1990.

[105] S. Pellicer, G. Chen, K. C. Chan, and Y. Pan. Distributed sequence alignment applications for the public computing architecture. *IEEE transactions on nanobioscience*, 7(1):35–43, 2008.

[106] R. Poplin, P.-C. Chang, D. Alexander, S. Schwartz, T. Colthurst, A. Ku, D. Newburger, J. Dijamco, N. Nguyen, and P. T. Afshar. A universal snp and small-indel variant caller using deep neural networks. *Nature biotechnology*, 36(10):983, 2018.

[107] R. K. Saiki, D. H. Gelfand, S. Stoffel, S. J. Scharf, R. Higuchi, G. T. Horn, K. B. Mullis, and H. A. Erlich. Primer-directed enzymatic amplification of dna with a thermostable dna polymerase. *Science*, 239(4839):487–491, 1988.

[108] A. Sboner, X. J. Mu, D. Greenbaum, R. K. Auerbach, and M. B. Gerstein. The real cost of sequencing: higher than you think! 12.

[109] M. C. Schatz. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[110] R. L. Schilsky. Personalized medicine in oncology: the future is now. 9:363.

[111] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[112] W. W. Soon, M. Hariharan, and M. P. Snyder. High-throughput sequencing for biology and medicine. 9(1):640.

[113] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson. Big data: Astronomical or genomical? *PLOS Biology*, 13(7):1–11, 07 2015.

[114] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large DataBases (VLDB)*, pages 553–564, 2005.

[115] A. Tarasov, A. J. Vilella, E. Cuppen, I. J. Nijman, and P. Prins. Sambamba: fast processing of NGS alignment formats. *Bioinformatics*, 31(12):2032–2034, 2015.

[116] TensorFlow Case Studies. https://www.tensorflow.org/about/case-studies.

[117] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[118] Timely Dataflow. A modular implementation of timely dataflow in rust. https://github.com/TimelyDataflow/timely-dataflow. Accessed: 02-03-2019.

[119] B. Tolhuis, J. Lunenberg, and H. Karten. Ultra-fast, accurate and cost-effective ngs read alignment with significant storage footprint reduction. http://www.genalice.com/wp-content/uploads/2013/07/GENALICE-poster-HiTSeq-2013.pdf. Accessed: 08-13-2016.

[120] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy. Storm@twitter. In *SIGMOD Conference*, pages 147–156, 2014.

[121] E. Tsai, R. Shakbatyan, J. Evans, P. Rossetti, C. Graham, H. Sharma, C.-F. Lin, and M. Lebo. Bioinformatics Workflow for Clinical Whole Genome Sequencing at Partners HealthCare Personalized Medicine. *Journal of Personalized Medicine*, 6(1):12, 2016.

[122] M. J. Turner, R. Hammond, and P. Cotton. A DBMS for Large Statistical Databases. In *Proceedings of the 5th International Conference on Very Large DataBases (VLDB)*, pages 319–327, 1979.

[123] K. Varda. Protocol buffers: Google's data interchange format. *Google Open Source Blog, Available at least as early as Jul*, 72, 2008.

[124] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: yet another resource negotiator. In *Proceedings of the 2013 ACM Symposium on Cloud Computing (SOCC)*, pages 5:1–5:16, 2013.

[125] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 374–389, 2017.

[126] J. Vivian, A. A. Rao, F. A. Nothaft, C. Ketchum, J. Armstrong, A. Novak, J. Pfeil, J. Narkizian, A. D. Deran, A. Musselman-Brown, et al. Toil enables reproducible, open source, big biomedical data analyses. *Nature biotechnology*, 35(4):314, 2017.

[127] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating System Design and Implementation (OSDI)*, pages 307–320, 2006.

[128] D. C. Wells and E. W. Greisen. FITS-a flexible image transport system. page 445.

[129] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD Conference*, pages 13–24, 2013.

[130] Yahoo Inc. TensorFlowOnSpark. https://github.com/yahoo/TensorFlowOnSpark.

[131] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Úlfar Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Proceedings of the 8th Symposium on Operating System Design and Implementation (OSDI)*, pages 1–14, 2008.

[132] M. Zaharia, W. J. Bolosky, K. Curtis, A. Fox, D. A. Patterson, S. Shenker, I. Stoica, R. M. Karp, and T. Sittler. Faster and More Accurate Sequence Alignment with SNAP. *CoRR*, abs/1111.5572, 2011.

[133] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 15–28, 2012.

[134] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 423–438, 2013.

Sam D. **Whitlock**

Doctoral Student · Computer Systems

EPFL IC IINFCOM DCSL, INN 231, Station 14, CH-1015 Lausanne, Switzerland

(+41) 78 956 10 31 | sam.whitlock@epfl.ch | https://people.epfl.ch/sam.whitlock | samwhitlock | sam-whitlock

Interests

Making small code do big things: big data frameworks, datacenter software, making it easier to write simple scale-out applications.

I am interested in developing application frameworks and software that lower the cost, expertise, and man-hours required to create scale-out applications for the datacenter. I have worked on projects ranging from API design of these frameworks to the low-level execution details to troubleshooting techniques for tracking down one unruly server among thousands.

Education

École Polytechnique Fédérale de Lausanne *Lausanne, Switzerland*

Ph.D. in Computer Science *September 2013 - Present*

- Datacenter Systems Laboratory (DCSL)
- Advisor: Professor Edouard Bugnion
- Thesis: Scaling Out Bioinformatics Processing in the Data Center

University of California, Berkeley *Berkeley, California*

B.S. in Electrical Engineering and Computer Science *August 2008 - May 2012*

- Specialization in Datacenter Application Frameworks
- Member of IEEE Student Chapter on campus

Research Projects

Persona - Accelerating Bioinformatics Applications *Lausanne, CH*

Graduate Research Assistant – École Polytechnique Fédérale de Lausanne *January 2016 - Present*

- Lead developer of Persona [1], a framework based on TensorFlow that accelerates bioinformatics pipelines. Persona describes bioinformatics applications in an abstract dataflow graph and executes them identically on a single laptop or across a cluster of machines while requiring only a few parameter adjustments.
- Developed the Aggregate Genomic Data file format, which unifies bioinformatics file formats into a single format. This format enables scale-out computation on datasets without relying on a specific big data framework.
- Developed Pipelined TensorFlow (PTF) a new abstraction for TensorFlow to enable persistent, service-style applications to run indefinitely while processing concurrent user requests.

Virtual Data Planes - VNToR *Lausanne, CH*

Graduate Research Assistant – École Polytechnique Fédérale de Lausanne *September 2013 - December 2015*

- Developer of VNToR [2], a framework that uses a programmable top-of-rack switch to offload network functionality seamlessly between software and hardware based on traffic patterns
- Wrote a distributed application to benchmark the network functions in the top-of-rack switch

Automated Networking Triage *Berkeley, CA*

Staff Researcher – International Computer Science Institute *May 2012 - July 2013*

- Worked with a team from the UC Berkeley NETSYS Laboratory to develop techniques for automated troubleshooting of network applications [3, 4]
- Participated in the development of a test framework for the troubleshooting mechanism
- Developed a formal model of our troubleshooting mechanism to prove its correctness [5]

AMPlab - Mesos Project *Berkeley, CA*

Undergraduate Researcher – University of California, Berkeley *Winter 2010 - Spring 2012*

- Developed a mechanism to allow a Mesos installation's slave nodes to report each job's memory and processor usage. This mechanism can be used to monitor for symptoms of abnormal behavior, as well as schedule new tasks based on observed usage instead of allocated resources.

Skills

C/C++	Used for both professional and personal projects, familiarity with many libraries (Broadcom's SDK, TensorFlow), testing and debugging complex applications
Python	Wrote large frameworks to run experiments and collect distributed execution traces, debugging and troubleshooting custom Python extensions (TensorFlow)
Linux / Development	Familiarity with troubleshooting tools (`strace`, `tcpdump`, `perf`), revision control best practices (`git`)

Experience

École Polytechnique Fédérale de Lausanne
Lausanne, Switzerland

Teaching Assistant — *September 2014 - December 2018*

- Information, Computation, Communication (ICC) – Autumn 2014, Autumn 2016, Autumn 2018
- Practice of Object-Oriented Programming (PPS) – Spring 2015
- Software Engineering – Autumn 2015
- Programming Systems, Project – Spring 2016, Spring 2017, Spring 2018

Microsoft Research, Cambridge
Cambridge, UK

Research Intern — *June 2015 - September 2015*

- Developed code to execute network virtual functions (NFVs) on a programmable NIC. This enables user-defined code to be run on a per-packet basis without involving the CPU.

VMware
Palo Alto, California, USA

Research & Development Intern — *May 2013 - August 2013*

- Developed an improved mechanism for tracing execution events of the NVP Controller Cluster

International Computer Science Institute
Berkeley, California, USA

Staff Research Scientist — *May 2012 - May 2013*

- Researched topics pertaining to troubleshooting techniques for distributed systems [4, 3, 5]

Twitter
San Francisco, California, USA

Software Engineering Intern — *May 2011 - August 2011*

- Developed a log aggregation framework and distributed application to aid developers in troubleshooting errant behavior in their distributed systems. It collects statements logging erroneous or fatal behavior in a language-agnostic fashion and aggregates these log statements into a data store for further analysis by other applications.
- Participated in the development of distributed application frameworks

Touch of Life Technologies
Aurora, Colorado, USA

Software Engineering Intern — *May 2010 - August 2010*

- Developed or improved iOS applications that support the core product, the VH Dissector

Touch of Life Technologies
Aurora, Colorado, USA

Software Engineering Intern — *May 2009 - August 2009*

- Developed an interactive voxel-based model renderer. This application presented the user with transverse, sagittal, and coronal cross sections of the visible human model and allowed them to highlight structures manually, aiding in the identification of nerves, which were often misidentified due to their minute size.

Publications

[1] Byma, S., **Whitlock, S.**, Flueratoru, L., Tseng, E., Kozyrakis, C., Bugnion, E., Larus, J. R., "Persona: A High-Performance Bioinformatics Framework". In: *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. 2017, pp. 153–165. URL: `https://www.usenix.org/conference/atc17/technical-sessions/presentation/byma`.

[2] Fietz, J., **Whitlock, S.**, Ioannidis, G., Argyraki, K. J., Bugnion, E., "VNToR: Network Virtualization at the Top-of-Rack Switch". In: *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*. 2016, pp. 428–441. DOI: 10.1145/2987550.2987582. URL: `http://doi.acm.org/10.1145/2987550.2987582`.

[3] Scott, C., Wundsam, A., Raghavan, B., Panda, A., Or, A., Lai, J., Huang, E., Liu, Z., El-Hassany, A., **Whitlock, S.**, Acharya, H. B., Zarifis, K., Shenker, S., "Troubleshooting blackbox SDN control software with minimal causal sequences". In: *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*. 2014, pp. 395–406. DOI: 10.1145/2619239.2626304. URL: `http://doi.acm.org/10.1145/2619239.2626304`.

[4] Heller, B., Scott, C., McKeown, N., Shenker, S., Wundsam, A., Zeng, H., **Whitlock, S.**, Jeyakumar, V., Handigol, N., McCauley, J. M., Zarifis, K., Kazemian, P., "Leveraging SDN layering to systematically troubleshoot networks". In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN 2013, The Chinese University of Hong Kong, Hong Kong, China, Friday, August 16, 2013*. 2013, pp. 37–42. DOI: 10.1145/2491185.2491197. URL: `http://doi.acm.org/10.1145/2491185.2491197`.

[5] **Whitlock, S.**, Scott, C., Shenker, S., "Brief announcement: techniques for programmatically troubleshooting distributed systems". In: *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*. 2013, pp. 134–136. DOI: 10.1145/2484239.2484293. URL: `http://doi.acm.org/10.1145/2484239.2484293`.

**Ce document a été imprimé au Centre d'impression EPFL,
imprimerie climatiquement neutre, certifiée myClimate.**