EPFL

Thèse n°10 141

Parallel and Scalable Bioinformatics

Présentée le 8 mai 2020

à la Faculté informatique et communications
Laboratoire d'informatique à très grande échelle
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Stuart Anthony BYMA

Acceptée sur proposition du jury

Prof. B. Falsafi, président du jury
Prof. J. R. Larus, directeur de thèse
Prof. I. Xenarios, rapporteur
Prof. C. Dessimoz, rapporteur
Prof. E. Bugnion, rapporteur

École
polytechnique
fédérale
de Lausanne

2020

Acknowledgements

First, a deep and sincere thanks to my advisor Professor James R. Larus for his unceasing and enthusiastic support of my Ph.D. studies. Jim has been a consistent source of encouragement, wisdom, brilliant ideas, and sure guidance. I am extremely fortunate to have been able to work with him — he has been an indispensable mentor to me, and I have learned a great deal from him.

I am grateful as well to my thesis committee, Professors Edouard Bugnion, Christophe Dessimoz, Ioannis Xenarios, and Babak Falsafi, for their insightful comments and helpful discussions. A special thanks to Professors Edouard Bugnion and Christophe Dessimoz, with whom I have been privileged to collaborate with on different projects. Both of them have an infectious passion for their work, which has always inspired and encouraged me, and I am sincerely grateful for their crucial contributions to my work.

I must also acknowledge the support and encouragement from all of my colleagues, friends, and fellow graduate students over the years: Sam Whitlock, Mia Primorac, Sahand Kashani, Adrien Ghosn, Marios Kogias, Mahyar Emami, David Aksun, Jonas Fietz, Endri Bezati, Benoit Seguin, Edo Collins, Winston Haaswijk, Dana Kianfar, Adrian Altenhoff, Yuji Takabayashi, Ziga Casar. Often over untold liters of coffee and ... other beverages, our discussions, collaborations, commiserations, and celebrations have all played a special role in my personal development and success. I thank you all.

Finally, and most importantly, an everlasting thanks to my wife. Jennifer: you have always seen my potential, even when I could not, and you have always encouraged me to set ambitious goals for myself. It is thanks to your unwavering support and belief in my success that I am able to complete this endeavor. You have always been and continue to be a wellspring of positivity, cheer, love, and wisdom, the great value of which I have difficulty in finding words to describe. I am blessed to have you in my life.

Lausanne, 14 April 2020 Stuart Byma

Abstract

The field of genomics is likely to become the largest producer of data as a consequence of the large-scale application of next-generation sequencing technology for biological research and personalized medical treatments. The raw sequence data produced by these methods is limited in usefulness and requires computational analysis to unlock its potential. *Bioinformatics* is a field that combines biology, genomics, and computer science to build algorithms and software to analyze biological data. Some of the current bioinformatics tools are having difficulty keeping up with the increasing rate of data production. For example, raw sequence preprocessing, which involves aligning subsequences to a reference genome, sorting, and other operations, can take many hours. Downstream processing applications also require computational innovation — protein sequence similarity search, an important tool in protein function characterization and the study of evolution, can take weeks or months to build high-quality databases, even relatively small ones composed of just a few thousand genomes.

This thesis shows that these computational challenges can be effectively and efficiently solved by a combination of fine-grained parallelism and horizontal scaling on highly-parallel compute clusters and data centers. This is shown through three primary contributions.

First, the preprocessing of whole-genome sequencing reads is addressed with *Persona*. Persona is a high performance and scalable bioinformatics system that unifies data, tools, algorithms, and processes for alignment, sorting, duplicate marking, and other operations in a common framework that scales linearly. For example, Persona can align 220 million short reads in ~17 seconds using a 32-node cluster. Second, a new technique for measuring and analyzing heap usage is introduced, which can help bioinformatics and other programs make more efficient use of memory, leading to performance gains of up to 10%. Finally, to accelerate protein similarity search, a new clustering algorithm is introduced that exposes parallelism, which, when combined with dynamic load-balancing, allows for efficient and scalable execution, leading to speedups of over 1400× over existing methods.

Keywords: bioinformatics, frameworks, big data, data center, scale-out, profiling, parallel algorithms, clustering, protein similarity search

Résumé

Le domaine de la génomique est susceptible de devenir le plus grand producteur de données en raison de l'utilisation à grande échelle des technologies de séquençage de nouvelle génération pour la recherche biologique et les traitements médicaux personnalisés. Les séquences brutes produites par ces méthodes sont cependant d'une utilité limitée et nécessitent une traitement analytique informatique afin d'en exploiter le plein potentiel. La *bio-informatique* est un domaine qui combine la biologie, la génomique et l'informatique pour construire des algorithmes et des logiciels d'analyse de données biologiques. Certains des outils bioinformatiques actuels peinent à s'adapter à la croissance des données à traiter. Par exemple, le prétraitement des séquences brutes, qui consiste à aligner les sous-séquences sur un génome de référence, à les trier et à effectuer d'autres opérations, peut prendre de nombreuses heures. Les applications de traitement en aval nécessitent également des innovations informatiques — la recherche de similarité de séquences protéiques, un outil important dans la caractérisation de la fonction des protéines et l'étude de l'évolution, peut prendre des semaines ou des mois pour construire des bases de données de haute qualité, même lorsque celles-ci sont relativement petites et composées de seulement quelques milliers de génomes.

Cette thèse montre que ces défis informatiques peuvent être résolus de manière efficace et efficiente par une combinaison de parallélisme minutieusement orchestré et d'évolutivité horizontale sur des clusters de calcul et des centres de données hautement parallèles. Ceci est démontré par trois contributions principales.

Tout d'abord, le prétraitement de séquences de génomes entiers est abordé avec Persona. Persona est un système bioinformatique de haute performance et évolutif qui unifie les données, les outils, les algorithmes et les processus pour l'alignement, le tri, le marquage des doublons et d'autres opérations dans un cadre commun qui s'adapte linéairement aux ressources disponibles. Par exemple, Persona peut aligner 220 millions d'échantillons génétiques en 17 secondes en utilisant un cluster de 32 nœuds. Ensuite, une nouvelle technique de mesure et d'analyse de l'utilisation du tas est introduite. Celle-ci qui permet de faciliter l'analyse et l'amélioration des programmes, notamment ceux de bioinformatique, afin de les rendre plus efficaces dans leur utilisation de la mémoire, entraînant ainsi des gains de performance pouvant aller jusqu'à 10%. Enfin, pour accélérer la recherche de similarité des protéines, un nouvel algorithme de regroupement (clustering) est introduit. Cet algorithme est hautement parallélisable et, combiné à un équilibrage dynamique de la charge de travail à effectuer, permet une exécution efficace et évolutive, conduisant à des accélérations de plus de 1400× par rapport aux méthodes existantes.

Mots clefs : bioinformatics, frameworks, big data, data center, scale-out, profiling, parallel algorithms, clustering, protein similarity search

Contents

List of Figures

List of Tables

1 Introduction

Bioinformatics is an interdisciplinary field that uses algorithms and software to process and analyze biological data. However, before turning to bioinformatics and its importance, we must first gain an understanding of the underlying biological systems that are the source of this data. Therefore, we start with a discussion of genes, genomes, and DNA, before moving on to sequencing, computation, bioinformatics, and the important problems to be solved.

1.1 Genomes, Genes, and DNA

All living things on earth share at least one thing in common: they are defined by their *genome*. The genome of an organism is a complete set of genetic instructions to build the organism, allowing it to develop, grow, mature, and ultimately reproduce or replicate. These genetic "instructions" are discrete units called *genes*, which influence certain features or traits of the organism. These traits include relatively simple things such as fur color or skin pigmentation. They also include complex traits, such as susceptibility to heart disease, which could be influenced by a great many genes. The environment can also affect how genetic traits are expressed — the classic example is that a child with "tall" genes will not grow to be tall if malnourished.

Genes are units of inheritance. When organisms reproduce, they pass their genes to their offspring. Most organisms reproduce sexually, so genes from two parents are combined to form the genome of the child, with each parent contributing 50% of their genetic material. Children thus inherit traits from their parents, which is why children typically look like and take after their parents.

Genetic inheritance forms the basic mechanism of the evolution of life on Earth. Inherited traits that confer an advantage in a given environment are naturally selected for, as the organism is more likely to survive, reproduce, and pass on advantageous traits. Disadvantageous traits likewise lower the probability of survival and therefore reproduction. The combination of random mutation (which produces new beneficial traits), genetic inheritance, and time

explains how life has been able to adapt to and colonize nearly every corner of our planet, albeit over hundreds of millions of years.

Organisms often have multiple copies of their genes. Humans, for example, have two copies of each gene, one from each parent. When reproducing, each parent provides one-half of their copies, so the child will receive one copy of a gene from each parent. Copies of a gene can differ, however, and these variations of a gene are called *alleles*. Consider a gene influencing eye color, and a reproductive situation in which one parent has two alleles for brown eyes, and the other parent has two alleles for blue eyes. The child will, therefore, have one copy for brown eyes, and one for blue eyes, but which allele will be expressed? The answer is that one allele will dominate the other and is referred to as the *dominant* allele. The other is *recessive*. In humans, brown eyes are a dominant trait, so the child would have brown eyes, even though they also have a blue eye allele. Generally, the genetic pattern is called the *genotype* (here, one brown-eye allele, one blue-eye allele), while the physically observed trait is called the *phenotype* (the child has brown eyes).

This discussion has been conceptual so far, and the reader may ask: what are genes actually made of? Remarkably, all organisms on the planet, in all of their diversity, have genes composed of the same underlying physical substrate. That substrate is DNA: Deoxyribonucleic Acid, a molecular polymer contained in the nucleus of every cell in an organism.[1] DNA consists of a molecular scaffold supporting a long sequence made of four different *nucleotides*, labeled A, T, C, and G. Each nucleotide, or *base*, is bound to its complement at each point in the sequence (forming a *base pair*). The whole structure physically takes the form of two bound strands in a double helix. The double stranding of DNA has several benefits. It is more structurally and thermodynamically stable, preventing a sequence from binding to itself, and resisting physical damage. Double stranding also facilitates the copying operation that is necessary when the host cell divides. DNA sequences of complex organisms are extremely large; the human genome, for example, contains 3.2 *billion* base pairs, organized into 23 pairs of discrete DNA molecules, which are called *chromosomes*.

Certain sections of DNA correspond to genes. Specific DNA subsequences indicate the start and stop positions of a gene within a genome. For a gene to physically influence the organism and carry out its function, it must be converted into a functional molecule. This is achieved through a two-step process. First, the section of DNA is "read" and *transcribed* into a single-stranded piece of DNA-like material called RNA. This in turn is *translated* by a structure called a *ribosome* into a functional molecule called a *protein*. Proteins are also polymers but are made up of *amino acids* molecules, of which there are 20 types. Each amino acid has a 3 letter DNA/RNA encoding. After the protein amino acid chain is formed, it folds into a particular 3D shape that influences the function of the protein. Proteins are tiny molecular machines, performing a specific function within the organism. They can be simple, just having a shape

[1] Prokaryotes, or single-celled organisms such as bacteria, also have DNA, though it may not necessarily be located in a nucleus. Viruses often have only RNA. Many scientists argue that viruses are not even "alive" because they cannot self-replicate without invading other organisms.

that allows them to bind to other molecules, or they can be large, multi-protein complexes, such as those that act as valves letting materials in or out of cells.

A good example in humans is found in a subsection of the 16th chromosome. The gene found here encodes the sequence for the α-subunit of the *hemoglobin* protein — the complex responsible for carrying oxygen in our red blood cells, forming the basis of our circulatory system.

This process of transcription/translation/function is known as the central dogma of molecular biology [46] and is the fundamental basis of life on Earth. Although many of these processes, including the structure and function of DNA, were discovered more than 40 years ago, we still do not have a full understanding of biological dynamics, which are extremely complex.

Measuring, reading, and understanding the biological sequences that encode these complex dynamics is crucial to advancing our understanding of biology, genetics, and evolution. The potential benefits of understanding DNA and proteins for human well-being cannot be understated. With this knowledge, viruses and other pathogens can be identified, tracked, and combatted much more quickly and effectively, and their evolutionary origin pinpointed. Cancer-causing mutations can be understood and potentially treated on an individual basis. Medical treatments may be tailored to a person's specific genetic makeup, which we will explore in more detail in the following sections. First, however, we will see how new technology has enabled fast and cost-effective sequencing (reading) of the full genomes of living organisms.

1.2 Reading Biological Sequences

Genome sequencing is a process by which the individual base pairs of DNA are digitized, revealing the exact A, C, T, G sequence. Early technologies to achieve this were developed more than 40 years ago, the most popular method being "Sanger" sequencing [109], created by Sanger et al. in 1977. This method sequences a small piece of DNA by generating many copies of the DNA of various lengths and terminating them with a fluorescent nucleotide. By detecting this special nucleotide with a laser and light sensor, and knowing the length of the copy, this method can accurately determine the base pairs in the original piece of DNA. However, the Sanger method is quite slow, requiring many separate repeated reactions to read only one base at a time.

Modern methods, often collectively referred to as *Next-Generation Sequencing* (NGS)[2], increase sequencing throughput (i.e., base pairs sequenced per second) by massively parallelizing the whole process. Generally, modern sequencing machines operate by cutting whole-genome DNA into small fragments and reading them using electrochemical processes, similar to Sanger sequencing. The crucial difference, however, is that NGS machines can sequence mil-

[2] Also referred to as *High-Throughput Sequencing* (HTS)

lions of these fragments in parallel, leading to large increases in throughput and substantially reduced overall sequencing costs.

We will focus our discussion on shotgun sequencing and sequencing-by-synthesis (SBS) techniques, as they are the most common NGS methods and have produces most of the datasets used in this thesis. In particular, the methods described here are used in sequencing machines manufactured by Illumina, Inc. [9], one of the leading developers of sequencing technologies.

The sequencing process begins by taking raw DNA from an organism and using a wet lab chemical process to chop it into small, single-stranded fragments (recall that normal DNA is double-stranded). Fragments are then sequenced individually. Sequencing of small fragments like this is often referred to as *shotgun sequencing* — the full sequence is fragmented randomly like a shotgun blast that contains many small projectiles.

Fragments are sequenced using a process called *sequencing-by-synthesis* (SBS). First, special adapter molecules are added to the ends of the fragments. This allows them to bind to the surface of a *flow cell*. Then, a process called *bridge amplification* (or *clonal amplification*) duplicates (clones) fragments attached to the flow cell many times over so that tight clusters of many duplicates form for each fragment. Sequencing then proceeds by building a complement strand on each single-stranded fragment, one base at a time.

Bases are added to the complementary strands using special *terminator* molecules, of which there are four types, one for each nucleotide (A, T, C, G). Sequencing proceeds in cycles, where one base is added to the complement strand each cycle in the form of a terminator, with the type (A, T, C, or G) determined by the current base of the template fragment. The terminator has two functions. First, it contains a base-unique fluorescent marker that is detected by a camera, allowing the machine to determine which base was added to which strand in this cycle — the key to "reading" the sequence. Second, the terminator molecule prevents others from binding below it, so as not to add more than one base per cycle, which would confuse the camera. After a cycle completes, the terminators are chemically cleaved, leaving the new base on the complement strand, ready for the next cycle. This continues for a given number of cycles, fixing the length of the output sequences. Accuracy of this process degrades with length, so output sequences are typically 100-150 bases long. The throughput, however, is extremely high because millions of fragments are sequenced simultaneously with this process.

Software on the machine performs *base calling*, translating the fluorescent signals into a digitized sequence, called a *read*, and estimating a *quality score* for each base in the read. Quality scores Q are defined as $Q = -10log_{10}(e)$ where e is an estimated probability that the base was called incorrectly. A high score, therefore, indicates a smaller probability of error. For example, a score of 20 means an error of 1 in 100, an accuracy of 99%. The machine may be unsure which base a given signal should translate to. In this case, "N" is used to refer to the ambiguous base.

Often, a variation of this process is used, called *Paired-end Sequencing*. Paired-end sequencing is the same up to the amplification stage. However, in this case, each fragment is anchored to the flow cell at both ends of the strand. Then, the complementary strand is built and sequenced from one end, and then the other end, producing two reads (a *read pair*) for each fragment. Depending on the length of the original fragment, the paired reads may or may not overlap. The degree of overlap is called the *insert size* and is measured in base pairs. Paired-end sequencing is advantageous because it can help to produce a higher quality reassembled genome.

The output of either single or paired-end sequencing is then a large set of digitized snippets corresponding to the original DNA fragments. Again, these snippets are referred to as *reads*. In addition to being in an indeterminate order, sequenced bases are also read imperfectly, hence the reads have a relatively large number of errors (0.1-1% [88]). Reassembling these reads into full sequences and analyzing this raw data, therefore, requires algorithms and computation.

1.3 Bioinformatics

The field of *bioinformatics* is broadly concerned with the algorithms and processes underlying this reassembling of reads, as well as the myriad downstream applications of the processed data.

There is a strong belief that genomic sequencing and bioinformatics will enable powerful new medical treatments that are tailored to individuals' genetic makeup. Genetics affects our health, both directly and indirectly. We might have the misfortune to inherit a genetic mutation that causes disease directly. The hemoglobin β-subunit, for example, can be afflicted by a single DNA nucleotide mutation[3] that causes the hemoglobin protein to polymerize under certain conditions, forming structures that deform red blood cells, often into sickle-like shapes — hence the name of the disease, sickle cell anemia. Random genetic mutations may also cause a cell to grow and divide uncontrollably, causing tumors or cancer. Typically, multiple mutations are required to cause cancer, compromising both cell repair mechanisms and self-destruct mechanisms. Bioinformatics can aid in our understanding of the root causes of cancers or genetic diseases, and do so at an individual patient level.

Other times, our genetics affects our health more indirectly. For example, individuals and populations differ genetically in how fast they metabolize drugs [35]. Some people may require higher doses for a drug to take effect, even levels that may be harmful to others. A example is the anticoagulant drug *warfarin*, which has highly variable patient dose responses, due in part to genetic variations [120]. Genome sequencing is becoming an important tool in integrating genetic concerns into practical therapies [91]. Bioinformatics is a crucial component of this rapidly expanding field of *personalized medicine* or *precision medicine*, where a person's unique genetic makeup is used to decide upon and tailor treatments.

[3] Usually referred to as a Single Nucleotide Polymorphism or SNP

Unknown pathogens can also be sequenced and identified quickly by comparing them to known viruses and bacteria using bioinformatics software. By analyzing differences in the sequences, researchers can also understand the evolutionary path of pathogens and pinpoint where they arose, which may help in developing effective treatments. For example, during the SARS (Severe Acute Respiratory Syndrome, caused by a type of coronavirus) outbreak in 2002, researchers were able to sequence the entire viral genome and determine that it was not closely related to any known groups of similar viruses while providing crucial information to aid in diagnosis and the development of antiviral treatments [86].

Bioinformatics can also help us understand evolutionary relationships between species. Protein sequences can be seen as proxies of genes — the units of evolutionary inheritance. Finding similar proteins in different species can help identify evolutionary relationships among different organisms and further our understanding of how life evolved on earth. Moreover, since similar proteins often perform the same biological function, protein similarity can also help categorize newly sequenced proteins as well. As a simple example, one can recognize that humans, chimpanzees, and gorillas are all closely related via their similar hemoglobin protein sequences, which evolved previously in a common ancestor. Similar methods are used in identifying and classifying pathogens like the SARS virus, which was eventually traced to its likely origin in 2017 — a species of bats in China [69].

Personalized health and proteomics are two important examples of the many possible applications of genomic sequence data. They are most relevant to this thesis as they provide the application domain and test cases for the computer science that is the focus of this work. Although the two applications make quite different uses of sequence data, they share a need common to many bioinformatics applications: complex, expensive computations across large quantities of data.

"Large quantities" of data may be an understatement, as sequencing technology has been advancing quickly in recent years. Next-Generation Sequencing (NGS) technologies can sequence millions of DNA fragments simultaneously, vastly increasing sequencing throughput while greatly reducing costs. The Human Genome Project (1990-2003) cost over $3 billion and took over ten years to sequence a human genome [8, 14]. Today, NGS technologies can sequence entire genomes in hours, at costs in the thousands of dollars. Newer iterations of the technology have projected sequencing costs in the hundreds of dollars [17]. Each machine can output petabytes of data per year, with the global amount of biological data set to surpass that of astronomy, YouTube, and Twitter [118]. This growth has placed immense pressure on the bioinformatics field — software tools must continually process more and more data.

1.3.1 Dealing with the Data Deluge

This pressure on bioinformatics software tools is compounded by the fact that sequencing technology and data production rates have increased faster than Moore's Law of scaling for microprocessors. Moore's Law, observed by Gordon Moore in 1965 [92], states that the number

of transistors in a microprocessor doubles every two years, allowing microprocessors to improve their capabilities quickly, year after year. Moreover, in 1974, Dennard et al. predicted that the power used by a transistor would remain proportional to its size, a trend referred to as *Dennard Scaling* [47]. These two scaling laws allowed chipmakers to continually increase the clock frequency and the sophistication of their processors without consuming more power, leading to each computer generation to be significantly more powerful than the previous in terms of CPU operations per second. Unfortunately, Dennard scaling ended around 2005, mostly due to leakage current in extremely small transistors. They can be made smaller, but they will require the same amount of power. Because of this, microprocessor manufacturers can no longer increase clock frequency as they did before, and they cannot even activate all transistors on a device simultaneously. As a result, bioinformatics (and other software tools) cannot rely on increasing single-core processor performance to meet their computational demands.

New approaches have emerged to continue improving microprocessor performance. Manufacturers have started using these "extra" transistors to construct additional CPU cores. Typical modern CPU devices now consist of multiple CPU cores on a single chip, even up to tens of cores on high-end processors. Exploiting parallelism is one of the primary ways to make programs run faster. Solving large computational problems today requires programmers to develop software that can run in parallel on CPUs, as well as distribute work across separate computers. This has led to the development of computing *clusters* and *data centers* — large collections of commodity computers in central locations, tightly connected by high-speed networks. Software running on these systems must be *scalable*, that is, it must increase its performance proportionally for each additional computer it runs on. In particular, scaling software to run on distributed computers in a cluster or data center is referred to as *horizontal* scaling or *scaling out*.

Typical bioinformatics applications and formats are not ready for the era of sequence "big data" [94, 110], nor are they set to take advantage of the parallelism offered by modern processors and clusters. Many bioinformatics data file formats and application software packages came from the earlier age of sequencing, which produced far less data at a slower rate. In this world, single-threaded or brute-force methods were viable. Today with NGS, we routinely see analysis times measured in hours or even days.

Typical processing "pipelines" in bioinformatics are often just serially executed applications, each reading and writing large amounts of intermediate data to and from mass storage. The disparate file formats that these disparate applications use are monolithic and row-oriented, and thus unwieldy and unsuitable for scale-out processing. Some newer bioinformatics programs are multi-threaded, but many still only run on a single thread, and the vast majority are not engineered to scale on clusters. Bioinformatics, and software in general, can also be difficult to get right — there are many ways a program can suffer performance degradations. When a program is slow and inefficient, it is often challenging to discover the source of inefficiencies, and still more difficult to fix the problems.

The bioinformatics field *must* prioritize performance to keep pace with the rapid advance of sequencer technology. Given the current trends in computing systems, this means reorienting around performance through parallelism and horizontal scaling.

This thesis states that many important bioinformatics processes *can* be effectively parallelized, optimized, and efficiently scaled using ubiquitous commodity server clusters. First, we show that many bioinformatics applications used for preprocessing raw reads in whole-genome sequencing can be unified under a common, parallel, and scalable framework called Persona. Persona relies on the Aggregate Genomic Data format, a new file format that unifies bioinformatics data and facilitates scale-out processing. Persona can accelerate existing solutions on multicore servers, and it scales linearly on a commodity cluster. Second, we introduce a technique called detailed heap profiling for understanding how efficiently a program uses the memory heap[4], after finding that a particular bioinformatics program was slow due to inefficient heap usage patterns. A tool, Memoro, uses new techniques to expose opportunities for heap usage optimization in programs. Finally, we show that the typical brute-force or single-threaded methods for finding similar proteins can be accelerated via a new clustering approach amenable to parallelization on a commodity cluster.

1.4 Challenges

This section provides some additional detail on the challenges in bioinformatics addressed by this thesis and explains the context for the statement and contributions of this work.

1.4.1 WGS Preprocessing

NGS technologies have enabled the sequencing of entire genomes in hours. As mentioned earlier, NGS sequencers accomplish this by massively parallelizing the sequencing process, reading millions of DNA fragments at the same time.

Figure 1.1 shows an overview of an NGS sequencing and data processing, starting from raw input DNA. First, a DNA sample is cut into many small fragments in a wet lab and duplicated many times over. This mixture of fragments then fed into a sequencing machine, which produces fragment *reads* as described in Section 1.2. Each read is a string of A, T, C, G, or N (an ambiguous base). Quality scores generally decrease towards the end of a read, as the process accuracy degrades.

An NGS sequencer will output millions to billions of reads for a sample genome, all in an indeterminate order. Due to this lack of ordering, reads must be reassembled into a complete, coherent genome before further analysis can take place. This preprocessing generally involves several steps, also shown in Figure 1.1.

[4] The "heap" refers to dynamically allocated memory that a software program uses to perform its function.

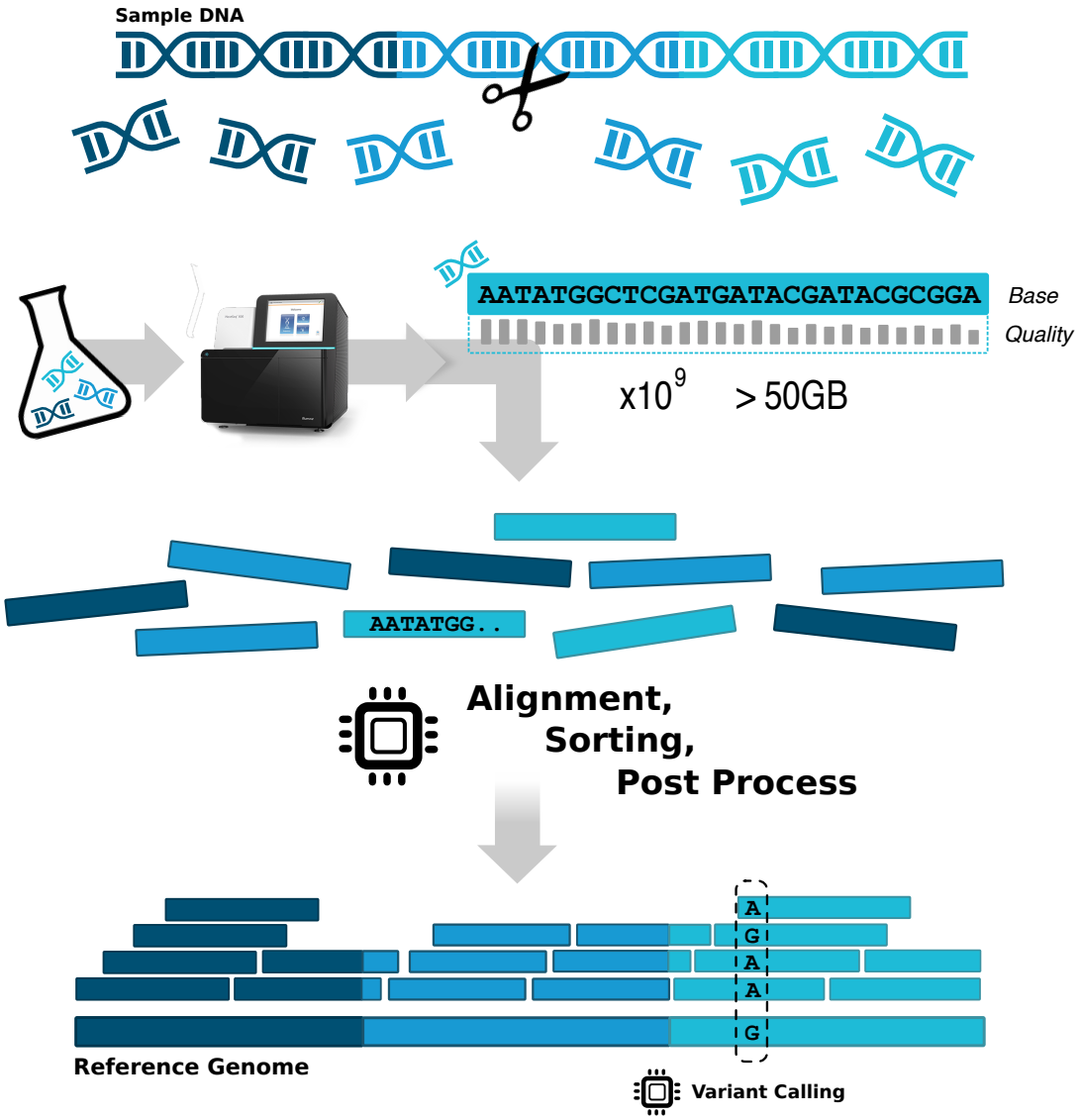


Figure 1.1 – Short-read whole genome sequencing using NGS technology.

Alignment

The initial assembly can be done via reference-guided *alignment*, where individual reads are aligned against a previously constructed reference genome, or via *de novo* assembly, where the reads are assembled without a reference. This thesis will focus on reference-guided alignment, which is more common in Whole Genome Sequencing (WGS) workflows. Since genomes of the same species are typically more than 99% the same, an algorithm can usually find a location in the reference for each read, at which the read aligns with few mismatches, inserts, or deletes. The earlier duplication of the input DNA serves to ensure that each location in the reference is covered by many reads. Alignment often involves computationally expensive

string matching comparisons, which can be done many times per read, making it an expensive step in genomic analysis.

The most common solutions for alignment use a two-step process to align reads to the reference. First, an index structure is used to efficiently map small portions of the read to exact matches in the reference, a process called *seeding*. Then, an *extension* process is applied to each seed, which uses an exact string matching algorithm to evaluate how well the remainder of the read aligns at each seed location. The best aligned extended seed is taken as the final result. Commonly used software includes BWA-MEM [78], which uses a suffix tree index, and SNAP [132], which uses a hash table index, while both use a similar extension method. Both of these tools are studied in this thesis.

Sorting

Aligned reads are sorted, usually by order of their aligned location in the reference. This step facilitates efficient algorithms and implementations for downstream analysis steps. Sorting is not computationally intense, but often requires "external" implementations that merge sorted subsets from disk because the entire dataset does not fit in memory. This makes sorting I/O intensive. Datasets are also often indexed after sorting, requiring a second traversal.

Post-Processing

Finally, many alignment post-processing steps may be performed. One example is duplicate marking, where a read is removed if there is another identical read aligned to the same location. Duplicate reads are artifacts of the DNA duplication process done before sequencing. Another post-processing step is Base Quality Score Recalibration (BQSR), which adjusts read quality scores to account for systematic technical errors due to sequencing chemistry or sequencer manufacturing flaws. These applications are typically not computationally intense, but they do traverse the entire dataset and thus generate a significant amount of disk I/O.

The aligned, sorted, and filtered dataset is then ready for further analyses. Typically, this involves *variant calling*, another expensive process that attempts to identify mutations or variations between the reference and the sample, or against other samples as well.

Challenges

There are several challenges to parallelizing and scaling out WGS preprocessing, which current applications, formats, and systems fail to address. First, there is a large number of different applications used for each step, most of which are standalone and perform only a single computation. For short read alignment alone, one source lists over 65 different tools [23]. Sub-computations are normally run one after the other, precluding overlapped execution of independent computations and forcing out-of-core and out-of-memory data transfer through

disk I/O. Although NGS technologies can produce massive datasets, bioinformatics applications are typically designed to be run in isolation on a single machine, possibly making use of multiple threads running on multiple cores, but rarely being designed with scale-out, distributed computation in mind. Making these applications scale requires completely redesigning them.

These standard bioinformatics formats are also sources of inefficiency. Raw reads, aligned reads, variant calls, sorted indexes, and other metadata all have different file formats, all of which are text-based, row-oriented formats. Some of these formats also duplicate data, such as the Sequence Alignment Map (SAM) format that unnecessarily includes all original input reads. Row-orientation requires reading an entire file even if only a subset of the data is required. Textual formats increase the amount of I/O (though some formats have analogous block compressed versions). However, the primary drawback of these formats is that they are monolithic and designed to be processed by a single application on one machine. They were created in an era before the deluge of NGS data and were not designed to support distributed computation.

1.4.2 Heap Usage Efficiency

Software is complex, and programmers need tools to understand and improve it. Modern software uses the memory heap heavily. Even small applications can make millions of allocations at many different locations in the code. Inefficient use of this memory, however, can lead to increased runtimes and increased peak memory usage.

Bioinformatics software is no exception to this. In fact, in our experience, we have found certain bioinformatics programs suffer from extremely inefficient heap usage. In one case, a widely used variant calling program spent nearly 30% of its runtime allocating and deallocating memory. As systems like this grow larger and more complex, understanding how to fix these problems becomes challenging. Libraries, frameworks, and packages hide internal allocations inside abstraction boundaries, and as a result, it is easy for inefficient heap usage patterns and performance-adverse allocations to go unnoticed. Even when a developer is looking for these issues, they can be hard to find.

Finding problems like these in bioinformatics software is crucial to improving its performance and capability of dealing with the ever-increasing amount of sequencing data. Most existing tools for heap profiling provide only a simple view of heap usage. They typically report bytes allocated/deallocated at each allocation location in the code, and record aggregated allocations up a dynamic call graph. This data is useful, but other methods may offer deeper insight into program behavior. It is valuable to understand exactly how *efficiently* a program uses its heap memory, for example:

- Was a given block of heap memory read or written (i.e. was this allocation necessary?)

- How much of an allocated memory block was accessed? Is it possible for the program to use a smaller block and use less memory?

- Was a memory block write-only or read-only? Write-only memory may be useless, while read-only heap memory may be indicative of a bug (reading uninitialized memory).

- Was the memory block accessed by multiple threads? This can aid in debugging shared memory bugs.

There are also heap usage patterns that are indicative of performance issues. Freeing an object long after its final use unnecessarily increases peak memory use. Conversely, allocating an object long before its first use has a similar effect. Allocating insufficiently large memory regions and growing them leads to unnecessary copying, which degrades performance. Similarly, repeatedly allocating and deallocating temporary memory can also affect performance. These patterns and heap usage inefficiencies can slow programs but are difficult or impossible to detect with current heap profilers, which do not record program accesses to heap memories. Finding inefficient usage patterns and eliminating their root causes can improve memory use and efficiency, leading to performance gains of up to 10%, and allowing bioinformatics and other programs to make better use of commodity hardware.

1.4.3 Protein Similarity Search

Finding similar proteins is another important problem in bioinformatics. Since protein sequences are transcribed from genes, similarities between proteins can be used as proxies to infer similarities between genes. These similarities can be used to detect *homologous* sequences or genes, which are descended from a common ancestral sequence. Homologs allow the transfer of knowledge from well-studied genes to newly sequenced ones since homologs often continue to perform the same biological function, even after accumulating differences during evolution. Figure 1.2 shows an example of this in the similarity between human and bonobo hemoglobin α-subunits. Most of today's molecular-level biological knowledge comes from studying a handful of model organisms, and then extrapolating to other organisms through homology detection. Sequence homology techniques are among the 100 most-cited papers of all time [125]. Finding similar or homologous protein sequences is a powerful way to understand newly sequenced data, or finding evolutionary relations between different organisms.

Figure 1.2 – Different proteins, same function – Alignment showing protein similarity between hemoglobin α-subunits from human and bonobo proteins. Extracted from the OMA Browser [28]

Proteins can be sequenced by finding coding regions in sequenced and preprocessed DNA reads. Proteins can also be sequenced directly using RNA-Seq [126], which uses NGS techniques to directly sequence transcribed RNA molecules in a sample. NGS techniques though still require a similar form of preprocessing as WGS, including alignment.

Current methods of finding similar proteins are computationally expensive. For a database of proteins, each is compared against all others in an exhaustive, "all-against-all" comparison, which is $O(n^2)$ in complexity. Sequences are compared using Smith-Waterman [115], a similarly expensive ($O(n^2)$) optimal string matching algorithm. This simple approach can find all similar pairs of proteins, however, it scales poorly with the number of sequences.

Large databases of proteins produced in recent years require new methods for similar pair detection. Even the *Quest for Orthologs* consortium [11], a collection of cross-species homology database projects, states "*[C]omputing orthologs between all complete proteomes has recently gone from typically a matter of CPU weeks to hundreds of CPU years, and new, faster algorithms and methods are called for*" [116]. New solutions must be parallel and scalable to take advantage of clusters and data centers, and will ideally be less than $O(n^2)$ in their computational complexity, which will allow them to scale to much larger datasets that will certainly be available in the future.

1.5 Thesis Statement

Bioinformatics must keep pace with the data production of modern sequencing technology, or innovation and further scientific and medical progress may be stifled. To provide an effective and practical means to address this problem, this thesis shows that bioinformatics computing should embrace existing, cost-effective computing hardware, such as commodity servers and, more importantly, large-scale compute clusters.

The statement of this thesis is:

> *Important computational problems in bioinformatics can be effectively parallelized and efficiently scaled out using commodity hardware clusters.*

This thesis is supported by evidence from three major research projects, which address the challenges outlined in the previous section.

First, the crucial WGS preprocessing steps of alignment, sorting, and duplicate marking are addressed using Persona, a dataflow-based, distributed framework for commodity clusters, and the Aggregate Genomic Data (AGD) file format. AGD provides a uniform representation for bioinformatics data in a single, chunked, column-oriented store. This new format is necessary to achieve the I/O bandwidth and functionality required by distributed computation. The Persona framework unifies disparate existing bioinformatics tools in a single parallel execution environment that can scale alignment linearly across a commodity cluster. Persona shows that

decoupling I/O and computation with small tasks can achieve a system that scales efficiently. Our evaluation shows that Persona accelerates several different applications (BWA-MEM alignment, sorting, duplicate marking) on commodity servers, and can scale the SNAP and BWA-MEM alignment applications linearly on a 32 node commodity cluster.

Second, we introduce a new tool that aids in the parallelization and optimization of bioinformatics software (and software in general): Memoro, a detailed heap profiler. Using the example of a bioinformatics program, we show that understanding how efficiently a program uses its heap can identify opportunities for performance optimization. This is especially relevant in the bioinformatics space, as solutions and systems are often built by bioinformaticians who are not software engineering experts. Detailed heap profiling is a technique that uses a compiler to place instrumentation at every memory access in a program, along with a runtime system to check whether accesses were inside heap memory. All program heap accesses are recorded, along with detailed statistics, and analyzed using a method that we show can effectively identify where a program is inefficiently using the heap. An evaluation of Memoro shows that the proposed method can highlight issues leading to performance gains of 10%.

Finally, we propose a new solution to the important problem of finding similar protein sequences. This proposal involves building clusters of proteins, where a cluster is defined by a representative sequence that is similar to all other cluster members. We present a new clustering algorithm called ClusterMerge that performs *precise clustering*, a type of clustering that ensures that each similar pair of proteins is placed together in at least one cluster. Similar pairs are then extracted from the clusters. ClusterMerge leverages transitivity inherent in the data to avoid comparing each sequence against all others while maintaining near-perfect accuracy (i.e. it can recover nearly as many similar pairs as a ground-truth brute-force approach). Reformulating the problem as a bottom-up *merge* of cluster sets, we show that the ClusterMerge algorithm exposes parallelism and build a parallel and scalable system to run it on commodity clusters. Our evaluation indicates that ClusterMerge can scale with up to 90% efficiency, and finds similar pairs with half as many operations as brute-force comparison.

In summary, this thesis makes the following contributions:

- Persona, a parallel distributed framework that unites various bioinformatics applications in a single execution environment that can scale on commodity clusters.
  - The AGD file format, which addresses shortcomings in existing monolithic bioinformatics formats. AGD minimizes necessary I/O, partitions datasets for distributed execution, and can support any existing bioinformatics data.
  - Distributed dataflow execution. Persona is built upon TensorFlow, which uses coarse-grain dataflow that limits framework overheads, which we augment with fine-grain task scheduling to efficiently use all computing resources.

  - I/O and compute task decoupling. Persona uses queueing to separate dataflow graphs and thereby decouple I/O and compute stages, allowing for variable granularity, leading to maximization of I/O bandwidth and even compute load balancing.
  - Linear scaling — Persona can scale SNAP and BWA-MEM alignment applications linearly over a 32-node commodity cluster. Persona also accelerates several other key WGS operations, sorting and duplicate marking.

- Detailed heap profiling: techniques and tools to help programmers understand how efficiently their programs use the heap, helping to optimize bioinformatics programs for use on commodity hardware.
  - A combined interception-instrumentation technique for tracking heap allocations, deallocations, and accesses to heap memory with an acceptable level of performance degradation.
  - A set of measures or "scores" that quantify how efficiently a program uses the heap.
  - A methodology for aggregating allocations by type, providing greater insight into allocation behavior than possible before.
  - Memoro, a system employing these techniques in 1) a compiler module and runtime for interception and instrumentation, and 2) a visualization graphical user interface (GUI) that aggregates data and scores in several forms, helping developers quickly diagnose problems.
  - A case study demonstrating that Memoro identifies impactful performance problems and that Memoro scores provide meaningful guidance.

- ClusterMerge, an algorithm and system for protein similarity search via *precise clustering* that is both accurate and highly parallel and scalable on commodity clusters.
  - A formalization of precise clustering using similarity and transitivity.
  - An algorithm, ClusterMerge, that reformulates the clustering process to expose parallelism.
  - An application of this algorithm to protein clustering for similar sequence discovery. Our system scales with up to 90% efficiency across a 32-node commodity cluster while maintaining 99.8% accuracy in similar pairs found.

1.6 Thesis Organization

This thesis is organized in roughly the same manner as the contributions have been presented. Chapter 2 presents the Persona framework, its goals, design, evaluation, and a discussion. Chapter 3 discusses detailed heap profiling, its techniques, design of both the compiler module, runtime, and visualizer, as well as the case study on real-world programs. Chapter 4 presents ClusterMerge, the algorithm, application to proteins, and evaluation. Chapter 5 provides some additional discussion and concludes the thesis.

1.7 Bibliographic Notes

The Persona project was joint work previously published with a colleague, Sam Whitlock, in the paper *Persona: A High-Performance Bioinformatics Framework* [41]. Sam contributed the AGD file format and the scale-out architecture, while I contributed the core dataflow architecture, integration of existing bioinformatics applications, and micro-architectural analyses.

2 Scaling Bioinformatics with Persona

Next-Generation Sequencing (NGS) machines produce vastly more data than previous technology such as Sanger sequencing. Because their output is unusable in its raw form, bioinformatics applications that use genomic sequence data rely on a few crucial computational steps to preprocess this raw data. These include aligning sequenced DNA fragments (reads) to a reference genome (alignment), sorting aligned reads, and various other steps. As sequencing becomes part of advanced clinical techniques, low latency of this processing pipeline becomes critical. Diagnoses and treatments must be decided on quickly, but many bioinformatics pipelines can take hours or days to complete.

The best way to solve this problem is to scale these computations efficiently across a commodity cluster. However, there are several challenges to overcome. Many separate software tools exist for each processing step in a bioinformatics pipeline, and while some are parallel on a single machine, very few are designed to scale across a cluster. Pipelines are typically constructed in an ad-hoc manner, sacrificing performance for flexibility. Existing file formats used to store data also tend to be monolithic, which makes partitioning required for parallel distributed execution expensive and non-trivial, while the formats' row-orientation precludes selective field access. Sequencer output and processing also tend to use different file formats, which duplicates data, consuming even more disk space and memory resources.

This chapter discusses our solution to these challenges: Persona, a framework that integrates and unifies existing bioinformatics applications in a common execution environment that can scale out efficiently. Persona is supported by the Aggregate Genomic Data (AGD) format, a column-oriented, partitioned design that unifies bioinformatics data under a single representation that explicitly supports distributed computation.

First, this chapter will review background material relevant to Persona and AGD, including current bioinformatics solutions for preprocessing and commonly used file formats. Next, the AGD format is introduced, followed by a discussion of the goals, design, and implementation of Persona. Finally, our evaluation of Persona and AGD shows that several WGS preprocessing

steps can be accelerated using Persona, particularly alignment, which can be scaled linearly on a cluster of 32 servers.

Bibliographic Note

This chapter is based on the paper *Persona: A High-Performance Bioinformatics Framework*, previously published and presented at the USENIX Annual Technical Conference in 2017 [41].

2.1 Background

The basics of short-read Whole Genome Sequencing (WGS) was discussed in Section 1.2. In this section, we provide a more detailed overview of the computations required to analyze the raw data produced by NGS machines, along with a discussion of the file formats that are commonly used to store raw and processed data.

2.1.1 WGS Preprocessing

Recall that modern NGS sequencing machines produce hundreds of millions to billions of reads in an indeterminate order. Each read is 100-150 base pairs long and could map to potentially any location in the sample genome, which is 3.2 billion base pairs long for humans. Reads are also produced with non-negligible error rates and vary in their quality. Computational preprocessing is therefore required to reassemble these raw reads into a complete, coherent genome upon which further analyses can be performed. Preprocessing is a critical step in most bioinformatics pipelines because few analyses can take place before a sequenced genome is reconstructed from the output of the sequencer. In particular, the important technique of variant calling, which looks for mutations or polymorphisms among DNA samples or a reference genome, requires this step. As we will see, current preprocessing solutions can be very time-consuming — parallelizing and scaling out preprocessing is therefore crucial to accelerating many bioinformatics pipelines.

There are generally two ways to reassemble a genome from raw reads. First, *de novo assembly* attempts to order the reads by comparing them against each other. Greedy de novo assemblers operate by aligning reads to one another, clustering highly overlapping reads, assembling these into larger contiguous regions, and repeating. An early example is CAP [70]. Other de novo assemblers use graph techniques on read k-mers[1] to reach a more optimal ordering of reads. Examples include SPAdes [33] and Velvet [134]. More popular, however, is *reference-guided* assembly, where reads are aligned to a preexisting reference genome. This process of alignment is one of the most computationally intense and time-consuming steps of WGS preprocessing, so we will explore it in more depth.

[1] A k-mer is a subsequence of length k. For example, `ATGG` has two 3-mers: `TGG` and `ATG`.

Alignment

Although reads from a sample can differ slightly from a reference genome, it is usually possible to find a very closely matching site for a given read because genomes of the same species are highly similar. Humans, for example, vary by up to 0.6% [45], implying that our genomes are typically over 99.4% identical. The process of *alignment* leverages this fact to locate each read in the sequencer output to its most likely position in the reference genome, in order to reassemble the reads into a complete genome. Other literature refers to alignment as *mapping* or uses the more general term *assembly*.

Due to the duplication that takes place in the sequencing process, each base in the genome will have multiple reads "covering" it. The average degree of *coverage* or *depth* is often used as a metric of sequencing quality. For example, if the coverage of a dataset is 50×, this means that each base in the sample has on average has 50 reads that include it. Higher coverage is critical to reducing overall sequencing errors because a larger consensus of reads covering a given base reduces the chance the base will be incorrectly determined. If one read contains an incorrect base, then statistically a large fraction of the other reads covering that base will have the correct base, which allows correct identification based on simple consensus or other statistical methods.

Alignment is a challenging problem. Each short read of one hundred or so base pairs must be mapped to its best matching location in a several billion character reference genome (in the case of human genomes, 3.2 billion base pairs — other species can be much smaller, e.g. bacteria at 1-10 million base pairs, or much larger, e.g. the axolotl at 32 billion base pairs). Finding the best location for a read using a globally optimal technique such as Needleman-Wunsch alignment [95] is $O(n^2)$ in time and space, making it far too expensive in practice. Practical solutions employ heuristics to find the likely best locations for a read, and then verify and rank these potential mappings using an exact *local* alignment (typically Smith-Waterman [115]).

There are two primary heuristic techniques that aligners use to find likely match locations for a read. Both involve building an index of the reference genome, which can then be used to quickly find all locations where a particular k-mer is present in the reference. The general algorithm proceeds by iterating through the k-mers present in the read, using the index to find out where these k-mers exist in the reference. The read is then locally aligned at each potential location identified by the heuristic using Smith-Waterman (S-W) [115] or similar, which generates an optimal alignment *score* that takes into account gaps, inserts and deletes. Inserts and deletes capture situations where the sample genome is missing a base or has an additional base, relative to the reference, going beyond the basic edit distance that only models mismatching bases. Gaps model multi-base insertions or deletions. Penalties to the alignment score introduced by inserts, deletes, and gaps are parameters that can be changed by the user.

Alignment scores output by S-W are used to rank potential locations, and the read is mapped to the best location found. For performance, algorithms will often terminate once they find a "good enough" match. Reads can also go unmapped if the aligner finds no well-matching location. This technique is commonly called *seed and extend* because a potential alignment is seeded using the heuristic and then extended using local alignment.

The first heuristic technique commonly used builds an FM-index [54] of the reference. An FM-index is a substring index based on the Burrows-Wheeler Transform [38], allowing the lookup of substrings in sub-linear time. The Burrows-Wheeler Aligner (BWA) [79] is a popular tool that uses this approach. Other examples include Bowtie [75], Bowtie2 [74], and SOAP2 [82].

The second common heuristic used in short read aligners is to build an index using hashing. The k-mers in the reference are hashed and inserted into a table. Read k-mers can then be looked up in constant time to find potential alignment locations (seeds) in the reference. Hash-based seeding can be faster, however, it uses significantly more memory. Examples of hash-based short read aligners include SNAP [132] and mrFAST [131].

As they use heuristics, these aligners do not always find optimal or even the same alignments. They also have many parameters that can be adjusted, for example, to find better mappings for different read lengths or trade accuracy for execution speed. However, most aligners share in common an inability to quickly process massive datasets. While seed generation tends to be fairly efficient in most tools, the extension phase that uses local alignment (an $O(n^2)$ algorithm) is very expensive and time-consuming. We have found, for example, that SNAP can spend up to 60% of its runtime doing local alignment extensions. With very large datasets, these tools can take many hours to complete the alignment of a whole dataset, even when running on a parallel machine, as many of these do.

Sorting and Post Processing

Reads are typically written to mass storage by existing tools as they are aligned. As a result, the output file contains the aligned reads still in the unspecified order produced by the sequencer. Many downstream processes are only efficient when operating on data ordered by aligned location, so sorting is a crucial step in WGS preprocessing. Randomly accessing certain locations is often required as well, so sorted data is usually indexed afterward, requiring another traversal. There are many common tools used to sort aligned reads, including samtools [81], Sambamba [121], and Picard [3], which is part of the Genome Analysis Toolkit [6].

An important processing step is to filter alignments that are likely PCR duplicates, which are caused by the duplication processes in the sequencing machine. Essentially, any two reads of the same length aligned to the same location are considered duplicates and one is removed. Some definitions of "duplicate" may also require that the read sequences themselves be identical as well, depending on the specific tool or implementation. Duplicate marking software includes GATK Picard [3], SamBlaster [53].

Base Quality Score Recalibration (BQSR) is another process that cleans the data of artifacts generated by the sequencing machine. These artifacts are caused by non-random errors in equipment, such as manufacturing flaws or sequencing chemical reactions. BQSR uses an empirically derived model to adjust quality scores to reflect these non-random errors.

File Formats

The many separate applications used in the WGS preprocessing steps use several different file formats used to store data. Initially, new file formats were created by researchers as they were needed. Some are quite old and were never intended to support highly parallel, distributed processing of the data they store. For example, FASTA [103], which was first released in 1985, is a plain text format used to store nucleotide or amino acid sequences. FASTA uses a delimiter character (“>”) to denote metadata, which is followed by the sequence itself on the next line. Often, lines are limited to 80 characters. This is so that the files were more readable on the original development terminals that could only display 80 characters in larger font sizes.

FASTQ [44] is a direct evolution of FASTA, adding a field for quality scores encoded as ASCII characters. FASTQ is the de facto standard for raw reads and is still the output format for most sequencing machines. Paired-end sequencing produces two FASTQ files, one for each mate of the pairs. For modern NGS machines, these files are tens to hundreds of gigabytes.

Aligned reads are typically stored using the Sequence Alignment Map (SAM) format [81]. SAM duplicates the data in FASTQ and includes numerous other fields specifying alignment information. It is also text-based and row-oriented. WGS post-processing (e.g. sorting) tools will typically read in a SAM file and produce a transformed copy. Most subsequent analyses of WGS data also use SAM as input, usually generating another format as output. For example, variant calling produces a Variant Call Format (VCF), another row-oriented text file similar to SAM. The Browser Extensible Data (BED) format from the bedtools suite [106] specifies genomic features as ranges along with other metadata.

Due to increasing dataset sizes, many formats now have compressed counterparts. FASTQ files are often gzip-compressed to save space. BAM is an indexed, block-compressed counterpart of SAM. CRAM [4] is a newer format that leverages reference-based compression, in which reads are stored as their differences relative to a reference genome, leading to vastly reduce storage but increased computation to extract reads.

WGS Preprocessing Solutions

Many existing systems for WGS preprocessing are built in an ad-hoc manner using shell scripts. Different applications are invoked at each stage of the pipeline, depending on the particular data or even the personal preference of a user. For example, alignment may be done with BWA-MEM, sorting done with samtools, duplicates removed with samblaster, and variants identified with GATK HaplotypeCaller. Many of these applications are open-source and rely

on de facto standard file formats described above to interface between pipeline stages. While flexible and supportive of innovation, this fundamentally limits their data transfers to text data, at a rate determined by the OS pipe system. These transfers are not necessarily slow at the OS level, however, the constant marshalling and unmarshalling of text data can impose a considerable overhead. This method also provides no solution to transfer data to remote servers over a network. Because applications are isolated, there is little to no opportunity for cross-stage optimizations, such as keeping data in memory between stages.

The Genome Analysis Toolkit (GATK) [6] is a well-known framework for genomics processing. GATK is written in Java and uses Spark [133] to distribute processing. However, GATK does not include all necessary WGS preprocessing steps, notably lacking integration of an alignment tool, a crucial step in WGS preprocessing. GATK also uses only de-facto standard file formats and relies on an underlying distributed file system (HDFS [114]) if running on a cluster. While GATK can supposedly run on a cluster using Spark, this feature still appears largely in development. It is likely the vast majority of users still use it only on a single machine as part of shell-scripted pipelines.

Another popular system is BCBio (Blue Collar Bio) [22], which allows users to describe their pipeline in a simple configuration file, and then run across multiple servers. BCBio relies on network file systems (NFS) to share large files between processes and performs fairly large-granularity partitioning upfront. This limits data movement speeds to the NFS maximum, which may not be sufficient, while the partitioning granularity may lead to load balancing issues as not all partitions will have the same processing times. While BCBio is a step in the right direction, at its core, it is still a pipeline of unmodified existing applications and does not solve the fundamental problems.

We summarize these fundamental problems as follows:

- Existing file formats are monolithic and row-oriented, which
  - prevents easy parallel access and distribution,
  - precludes efficient use of read and write I/O bandwidth,
  - prevents selective field access,
  - is inflexible, requiring new standards or formats for new application data types
- Existing WGS applications
  - are myriad and isolated, preventing intra-pipeline optimizations,
  - rarely have any built-in support for scale-out execution,
  - are not always parallel and do not always scale efficiently across cores.

The following sections describe our solutions to these problems. First, the Aggregate Genomic Data (AGD) format solves the issues related to existing file formats. Persona, a parallel and

scalable bioinformatics framework, then leverages AGD and existing application code to build high-performance pipelines that can scale efficiently across commodity clusters.

2.2 Aggregate Genomic Data Format

The Aggregate Genomic Data (AGD)[2] format [129] is a new file format for genomics data, and has several primary design goals. First, AGD seeks to unify the many different bioinformatics formats used in WGS into a single, common, extensible format that can accommodate all types of data. Unification can simplify pipelines and applications by requiring only one code library for I/O and parsing, as AGD can be used at all stages of a WGS pipeline. Format unification can also eliminate data duplication, such as that in FASTQ and SAM files, and make for more organized and consistent WGS data storage.

The second goal of AGD is to alleviate many of the performance drawbacks of current file formats and facilitate high-throughput read and write I/O. As applications scale out to more cores and servers, the underlying format must be able to sustain the I/O required by the applications, both for input and output. The row orientation of existing formats results in much more I/O than necessary when the application does not require access to all record fields, which is often the case in bioinformatics processes. Row orientation forces I/O and parsing of all data located between two records in the same column, particularly since many fields are of variable length. This can lead to unnecessary I/O and cause a bottleneck for some processes. Writing data back to storage also requires all intervening data between records to be rewritten. AGD provides selective field access so that applications can access just the fields that are required. On top of this, AGD uses data compression to reduce the disk space required for datasets.

Finally, AGD has the goal of explicitly supporting scale-out processing. Existing formats complicate scale-out processing for two reasons. First, they are monolithic, single files that must be partitioned either in advance or at runtime, which complicates the system and adds overhead. Second, their row-orientation and variable record size makes partitioning non-trivial, as the entire file must be read to find record and partition boundaries. Commonly used file compression would increase these overheads as well. These complications are further compounded by the fact that partitioning must be done centrally on a single server because the operation is sequential. The total system throughput would be limited to that of a single network interface, crippling scalability. Each different format used would also need different code written to perform the partitioning of its data. AGD is architected specifically to circumvent these issues in a simple but effective manner.

As genomic datasets continue to grow larger, the use of compression is crucial. AGD aims to support variable levels of compression as an additional objective. For example, after a dataset

[2] As explained in Section 1.7, AGD was primarily conceived of and developed by Sam Whitlock, with some contributions from myself. It is included here because it is an essential and integral component of Persona.

has been processed, the user may wish to put it in *cold storage*, i.e. cheap, durable storage that has a high read/write cost. A compression algorithm with a high compression ratio would be more suitable in this case. If a dataset is likely to need processing soon, a better solution would be to use a cheaper (computationally) compression algorithm that trades compression ratio for computing efficiency, so that data can be accessed more quickly. Ephemeral data passed between processing steps might even be stored without compression.

Format Architecture

In most areas of its design, AGD trades complexity for performance. AGD does not have a great number of features, however, this affords flexibility and performance. AGD also does not assume any dependencies — it does not require a specific filesystem or platform, only the ability to store binary objects.

At its core, AGD is a table of records. Each record is a variable-length byte array, with several structured encodings defined. AGD is a strictly flat schema, in which each record contains all fields, and it does not support array-like fields or nested records.

There are currently several encodings or *types* that AGD defines for genomic data. The first is plain text data. This is simply an ASCII encoded character array, but can also be used to store arbitrary data structures defined by the user. Persona uses this type to store quality score strings and metadata strings for sequencer reads.

The second type is a special encoding for sequencer reads. Recall that reads are made up of an alphabet of five characters: A, T, C, G, and N, an ambiguous base. This *bases* encoding uses a three-bit representation for each character, and can pack 21 bases into a single 64-bit field. All read bases are stored in this format in AGD.

The final data type currently supported by AGD is *structured* data. This is essentially a serialized in-memory data structure. While there are many structure serialization solutions, AGD currently uses Protocol Buffers [10]. Structured data fields are used to encode alignment results in Persona, which indicate in which chromosome (contig[3]) and at what position a read has been aligned, along with other mapping quality information. AGD's simplicity allows adding additional data types quite easily, however, these three types were sufficient for the WGS preprocessing implemented in Persona.

To reduce I/O and enable selective field access, we designed AGD with a column orientation. This means that records are stored by column, instead of by row. Each record field of the same column is stored physically adjacent to the next record field of the same column. This way, for example, a program can iterate over the reads in a dataset without traversing the other fields

[3] "Contig" more generally refers to a set of overlapping DNA sequences that represent a consensus region. While chromosomes in a reference genome are usually single *contigs*, there are other contigs in the reference that do not necessarily represent chromosomes.

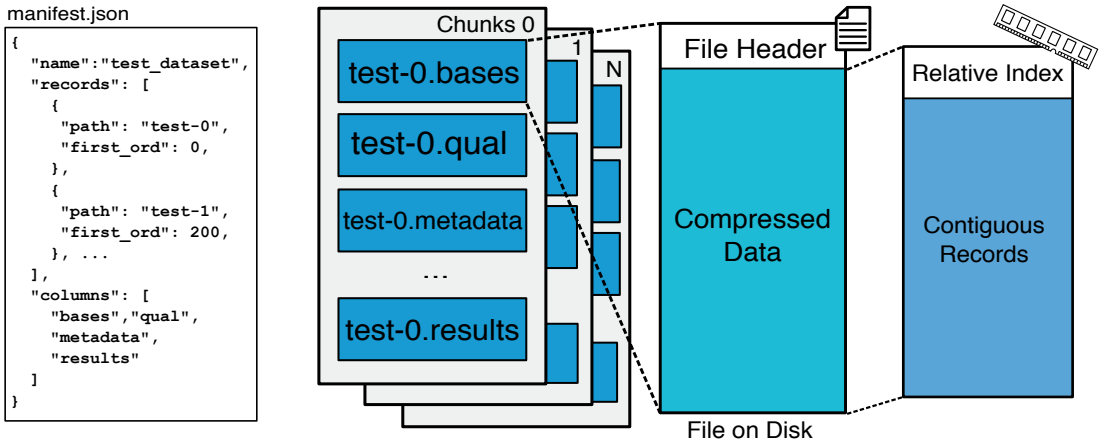


Figure 2.1 – Aggregate Genomic Data (AGD) format architecture

(metadata, quality scores), vastly reducing required I/O. Generally, AGD stores each record field in a separate column, with a numeric identifier connecting records across columns.

To better support scale-out processing, AGD partitions columns into fixed-length *chunks*. Fixed-length refers to a fixed number of records; chunks will not necessarily have the same byte size. While columns can have different chunk lengths, Persona uses uniform chunk lengths across all columns. Chunks form the basic unit of storage for an AGD dataset.

Figure 2.1 shows an overview of the AGD format architecture and on-disk representation for a dataset of aligned reads. The columns consist of bases (encoded with the *bases* type), quality scores (plain text), metadata (plain text), and alignment results (structured data). Columns, and the partitioned chunk files that they are stored in, are tracked via a dataset unique *manifest*, implemented as a simple JSON file. The manifest indicates a dataset name, which columns are present, column data type, as well as a list of chunks and how many records they contain. This example shows two chunks explicitly in the manifest — the `first_ord` field in the manifest indicates the global index of the first record in a chunk. Chunks in this example contain 200 records.

The total number of chunk files is equal to N times the number of columns. Chunk files are stored in the underlying medium (e.g. a filesystem or an object store) in a simple, binary format that starts with a fixed-size header. The header indicates version information and describes the chunk contents, and is followed by a block of compressed or uncompressed records. The data block contains contiguous record data, preceded by a relative index. Each record in the block has a corresponding entry in the relative index, indicating its size. This allows a program to efficiently traverse the records by simply advancing a pointer by the amount shown in the corresponding index. In addition, because many record fields are of similar size in genomics (e.g. read bases and quality scores), the relative index is often highly repetitive, making it highly compressible, as opposed to an absolute index that indicates the offset of each record from the start of the block.

The design of AGD facilitates many of the computations required in WGS preprocessing, and helps to optimize them. For example, AGD reduces I/O required to perform alignment, because only bases and quality scores are required to be read from disk, and only results need to be written back to a new results column. The use of relative indexes leads to efficient sequential traversal and trivial parsing.

On occasion, some programs require random access to a dataset or a chunk. For example, when sorting, the process must randomly access unsorted records from a chunk when assembling a sorted one. This can be accomplished efficiently with AGD by building an absolute index to the file contents with a single pass of the relative index. The correct chunk can be found by simply traversing the list of chunks in the manifest, or building an index of that as well.

The chunking of AGD also allows the distribution of this processing without any expensive coordination. Each chunk can be processed in isolation by different processors, for easily parallelizable operations such as alignment or duplicate marking. Chunk sizing must be managed carefully, however. Large chunks, while having better compression ratios and lower I/O overhead, can lead to straggling[4], impacting system efficiency. Smaller chunk sizes have larger I/O and parsing overheads, and may result in processing cores standing idle.

Subsequent sections in this chapter detail Persona, a high-performance bioinformatics framework that leverages these key design points of AGD to build bioinformatics pipelines for WGS preprocessing that can scale across commodity servers.

2.3 Persona Architecture

In this section, we describe the functions of Persona and its design objectives. Persona is designed with personalized medicine in mind. This looks toward the near future when a person's genetic makeup will be used to inform medical treatments and healthcare decisions. Ideally, a doctor should be able to perform WGS analyses in the time-span of an appointment, i.e. on the order of minutes. Even though the sequencing itself can take hours or days, clinicians may want to perform analyses on already sequenced patient genomes. For example, they may wish to realign a patient's raw sequence data against a new, higher quality reference genome. It is also possible that sequencing technology will advance to the point where WGS is possible on the order of minutes.

Persona, therefore, adopts a design philosophy that prioritizes the latency of individual requests. A request, for example, could be to align a dataset against a particular reference genome, perform a WGS preprocessing pipeline on a patient's raw sequence data, or compute variants of a patient's genome relative to some population. An individual request, however,

[4] *Straggling* refers to a situation in which one processor is stuck computing a large sequential task while other processors sit idle. It is a primary cause of low scaling efficiency, and it is usually a result of tasks being uneven in size or too large.

can entail a large amount of data and computation, so throughput is still an important consideration as well. Current methods, as explained in Section 2.1, can take hours or days. The primary function of Persona is to provide a solution to this problem, allowing individual WGS preprocessing requests to take place in minutes.

Persona has several design objectives. First, while AGD unifies data, Persona aims to unify bioinformatics applications and computation under a common framework. A common framework confers many advantages. Users need only install and manage a single system, rather than many separate applications with different versions. Integrated applications can rely on common subsystems to perform common tasks such as I/O, file parsing, and decompression, allowing the development of highly optimized, shared code. A common framework can also handle parallelism and distributed computing in a common manner, depending on how the application in question is parallelizable, making it much easier for integrated applications to scale their workloads. In addition, integration within a framework can allow intra-pipeline optimizations, such as passing data using in-memory buffers, as opposed to using pipes or files that incur additional overheads.

A second design objective of Persona is to scale efficiently across local cores and distributed clusters, and to do so linearly[5] when possible. Many tasks in WGS preprocessing are embarrassingly parallel, i.e. little to no effort is required to separate the tasks into isolated work units, and the processing of one work unit does not require communication with any others. Read alignment, for example, exhibits this behavior. Each read can be processed independently from the other reads; the only shared resource is the reference genome, and that is read-only and does not require synchronized accesses. Therefore, read alignment should be able to scale linearly to a limitless number of processors. However, system-level constraints must be optimized to achieve this goal. A large number of processors require sufficient I/O capacity and computation to read, parse, and decompress input data so that all processors are kept busy. Likewise, results must be written back to storage as quickly as they are produced. Persona, in conjunction with AGD, aims to provide an efficient solution to these system-level scaling barriers.

A final design objective of Persona, crucial to its primary function, is to be high-performance and efficient, that is, the majority of time spent performing a computation is spent actually computing, and not waiting for data or communicating. Persona attempts to keep all available processors busy doing useful work at all times. For example, Persona uses a zero-copy architecture, where data is never copied unless absolutely necessary. Copying large amounts of data, such as those we see in bioinformatics, is costly and wastes CPU cycles that are better spent doing actual computation.

[5] Linear scaling is ideal scaling, where each additional processor reduces the total compute time by a proportional amount.

Architectural Design and Execution Model

In this section, we describe how the design of Persona allows it to achieve the aforementioned objectives.

Persona utilizes a coarse-grain dataflow execution model. Major functions of the system — I/O, decompression, parsing, computation — are interpreted as coarse-grain dataflow operators. *Coarse-grain* refers to time granularity, with *coarse* meaning that each operator may require a significant amount of time to complete. This helps mask the latency overhead of setup and teardown of each operator. In contrast, *fine-grained* operators would be small and require little time to complete.

Dataflow operators have strictly defined inputs and outputs, and can be formed into *dataflow graphs*, which in our system are directed, acyclic graphs (DAGs). Each operator is a node in a graph. The graph is then run by a dataflow engine, which executes the graph according to dataflow semantics. Dataflow semantics refers to the execution order of nodes in the graph. Essentially, the final output node is first indicated to the engine, which then enumerates all dependencies between nodes, up to nodes with no dependencies or those that read inputs. A node can then be executed as soon as its inputs are available. Independent nodes (those with no dependency chain between them) can be executed in parallel.

Figure 2.2 shows an example of a dataflow graph. The output of node D is being produced. It depends on node E and node C (which depends on nodes A and B). Dataflow semantics allow node E to be executed in parallel with A, B, and C. C must wait for A and B to complete, while D must wait for C and E.

Persona implements major system functions to dataflow operators. For example, there are operators for performing I/O from disk or other sources (e.g. loading AGD chunks into memory), decompressing AGD chunks, parsing decompressed chunks, performing alignment of reads, writing alignment results, and other functions. This design affords a great deal of modularity and allows rapid construction or modification of pipelines. For example, if data must be accessed from a different source, only one node in the graph (the I/O operator) needs to be swapped, a trivial change. Interfaces between nodes (the graph edges) are strictly defined, and they remain the same if a node is swapped for another.

A basic dataflow model such as this offers other advantages as well. It simplifies the design, implementation, and deployment of the system, and it allows for simple integration of new applications or processing steps. Moreover, the explicit flow between operators simplifies performance and bottleneck analysis, and it makes it easy to adjust queueing between nodes for flow control and load balancing. Dataflow also naturally exposes parallelism, as we can see in the previous example where non-dependent nodes execute in parallel.

Persona uses Google TensorFlow [25] as its underlying dataflow execution engine. Although designed for machine learning applications, TensorFlow contains a general dataflow engine,

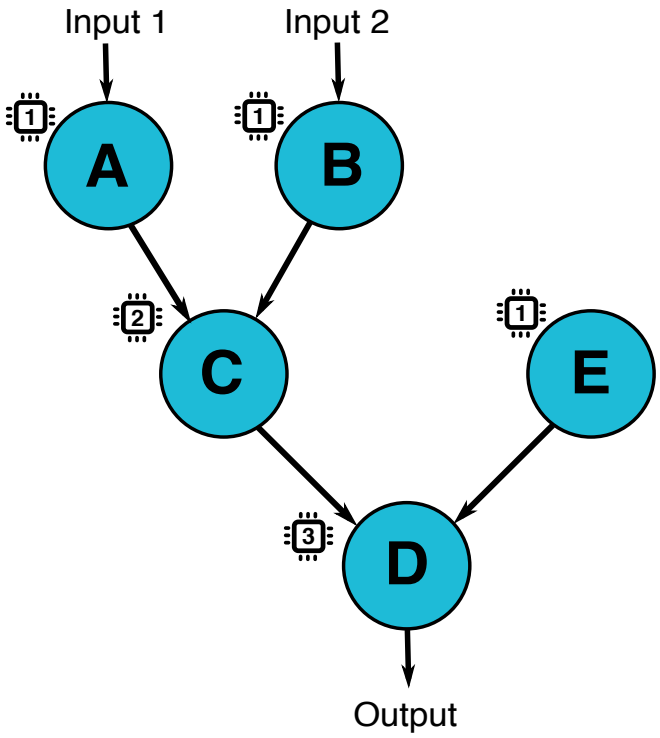


Figure 2.2 – Dataflow Graph and Execution Semantics — Output of operator D is being produced. One potential execution schedule sees E executed in parallel with A and B. C is then executed when its inputs from A and B become available. Finally, the output of D is computed. The numbered CPU icons show this ordering graphically as well.

which we repurpose to execute custom-defined operators. In TensorFlow, dataflow operators are assembled into a graph of *nodes* using a Python API. Node implementations are called *kernels*, and can be device-specific — TensorFlow allows nodes to have different kernel implementations, to map them to different hardware, most notably CPUs and GPUs. For this work, we use only CPU kernels, which are written in C++ and compiled alongside the TensorFlow runtime framework.

We demonstrate that TensorFlow can be used in this new context with minimal overhead (1%). Persona uses several techniques to achieve this, including augmenting coarse-grain system orchestration with fine-grained parallel execution within compute-intense kernels, reusing buffers to implement a zero-copy architecture, and using queueing to both limit memory use and seamlessly overlap I/O and computation.

Queuing

There is a drawback to TensorFlow's execution model in that graphs are executed in steps. Each dependent node is executed once to produce the graph output. A Persona graph would, for example, read an AGD chunk from disk, decompress and parse it, align the reads within, and write back the output AGD chunk. The problem here is that while we are performing I/O or parsing chunks, no computation is being done because the alignment operator is waiting for data. Ideally, we would like to read AGD chunks and buffer them while computation is being performed at the same time. To solve this problem, system components are further separated into *subgraphs* connected by queues, which allow the subgraphs to execute independently and seamlessly overlap their functions. Queues employ mutex locking to synchronize accesses of upstream (or producer) and downstream (consumer) subgraphs.

For example, a subgraph containing nodes that read data from disk can run in parallel with the subgraph that performs decompression of the data. Figure 2.3 shows how queues work in TensorFlow. One node (orange in Figure 2.1) provides the queue itself, a shared memory object. *Push* and *pop* nodes execute operations to push and pop data atomically into or out of the shared queue object. This organization provides a uniform interface to the queue that preserves TensorFlow execution semantics. Push and pop nodes are simply dataflow operators that can be added to a graph. In case of a full or empty queue, they block until data or space become available.

As the figure shows, it is also possible to have multiple subgraphs, upstream or downstream, accessing the queue in parallel from different thread contexts. Each context instantiates its own push/pop nodes to access the shared queue atomically. This allows Persona to seamlessly parallelize certain subgraphs if they are bottlenecks. For example, if the subgraph decompressing AGD chunks in memory cannot provide data quickly enough to the downstream subgraph aligning reads, it can be duplicated to increase its throughput. This also facilitates tuning of the overall system to maximize performance.

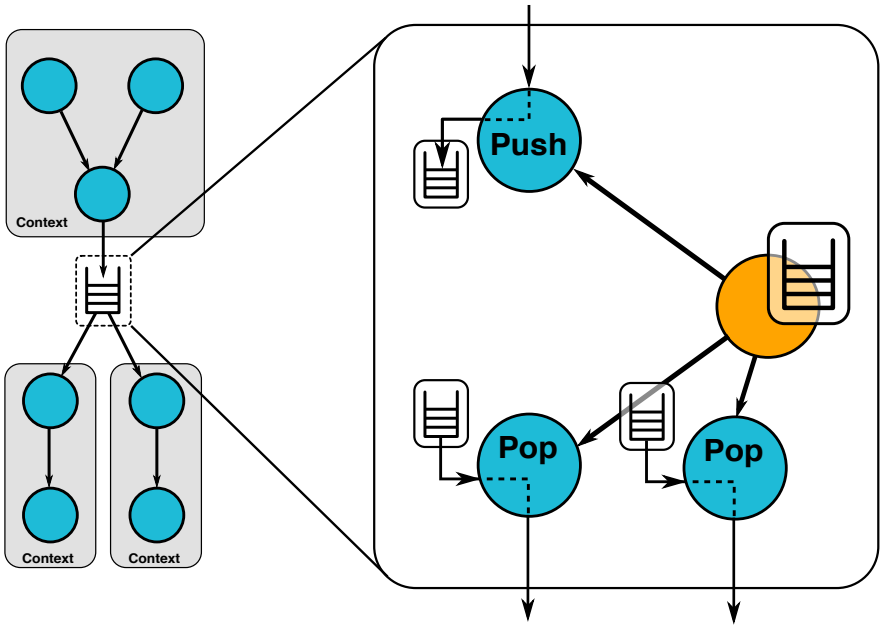


Figure 2.3 – Queue connected subgraphs as used in Persona

Persona employs *bounded* queueing. This provides an inherent mechanism to limit the memory use of the system by preventing a buildup of data in queues. Since bioinformatics can be compute-intensive, unlimited queuing could result in data loading far outpacing the rate of computation, consuming more memory than necessary to keep the processors busy. Bounded queuing limits the number of items in a queue, blocking the execution of subgraphs attempting to push more data until a downstream subgraph pops an entry and frees up space. At the same time, the queuing can ensure there is data immediately available to compute threads, so no processor ever goes idle.

Bounded queuing is also important for distributed computation. A bounded local queue on a remote server will limit the amount of work buffered in the server. Otherwise, each remote server will locally buffer large amounts of work, up to the size of the whole dataset, producing a situation equivalent to pre-partitioning and distributing all of the data. Since the computation time of each chunk of data is variable, some may take much longer to compute than others. This leads to *straggling*, where a majority of servers have finished computing their buffered data, but a final server straggles behind. Bounded queuing prevents this by limiting the amount of locally buffered work.

Data Movement and Resource Pooling

The "native" method of data movement in TensorFlow is passing *tensor* objects between nodes in the graph. This is not an efficient solution for Persona, however, because encoding string data (for reads and quality scores) in tensors leads to a large number of small memory

allocations because of the tensor object implementation. To avoid this and unnecessary copies and memory allocations, Persona moves data by passing *buffer* objects between nodes. These objects are implemented with TensorFlow's resource subsystem, which is used to share objects between nodes, the same system that manages queue objects as described above. A 2-vector tensor containing a (container, name) pair forms the reference that can be used to look up objects in the resource subsystem, which is managed by the TensorFlow runtime. The handle can be passed between nodes as a native tensor.

Buffers in Persona typically contain one AGD chunk, which forms the basic unit of I/O or computation granularity in Persona. Persona also uses the resource system to share other read-only data between nodes, such as the reference genome. In figures, blue nodes represent normal graph nodes, and orange nodes represent those that provide a shared resource.

Normally, when a graph operator needs a buffer for its output, it would simply allocate one from the OS (TensorFlow uses the OS memory allocator). Conversely, when finished with an input buffer the operator would deallocate it, again using an OS system call. This approach leads to a large number of OS-level memory allocation calls, many of which are large (tens to hundreds of megabytes in size), incurring a potentially large overhead. To avoid this, Persona uses a *resource pooling* technique. Buffers or other objects are managed by *pool* resources, which keep a list of free and in-use objects.

We will use buffers as an example. When a node needs a new buffer for its output, it will first access a buffer pool resource to obtain a free buffer. Downstream, when that buffer has served its purpose and needs to be freed, it is instead returned to the buffer pool. This technique avoids constant allocations and deallocations, as buffers are allocated up-front and reused. It also keeps the memory footprint of the entire system relatively stable. There is no limit on the number of buffers or pooled objects in circulation. However, since Persona uses bounded queuing, the system naturally reaches a steady state in which all resource object needs are provided through reuse, and no further allocations are necessary.

Constructing Graphs

Dataflow graphs in Persona, as in TensorFlow, are constructed via a high-level API in the Python language. A Python function is generated for each dataflow node when the system is compiled. These functions can then be easily used to form a graph via a chain of calls. Higher-level constructs, such as the queues that tie together the push/pop/queue nodes as shown in Figure 2.3, are allocated and connected similarly. This provides several advantages. First, it hides unnecessary detail from users of the system, such as the underlying kernel implementation code, which makes it easier to grasp what a particular graph is doing. Second, the design enforces a high degree of modularity, allowing newly constructed graphs to incorporate existing nodes and even entire subgraphs. Graphs are also easy to modify, for example, if a new storage system is used, only the nodes dealing with reading and writing data need to be changed, keep code modifications to a minimum.

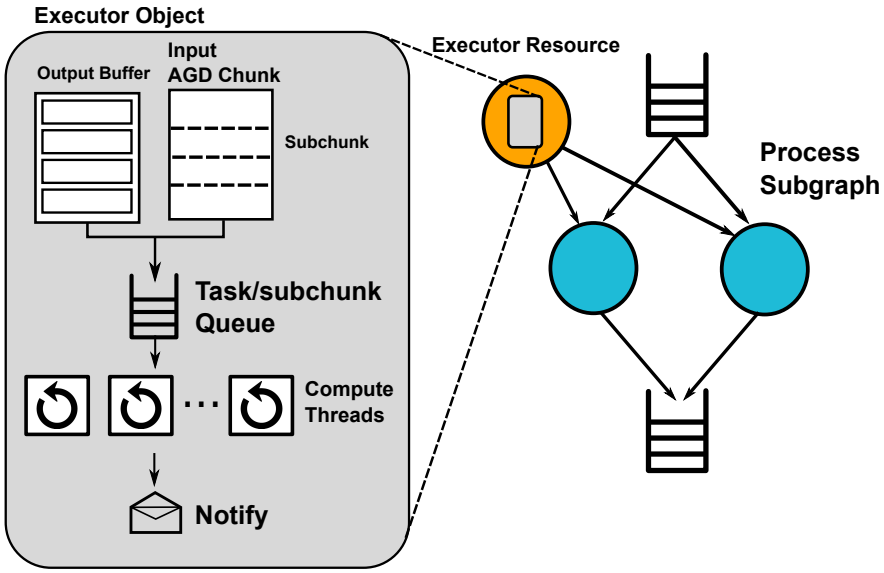


Figure 2.4 – Aligner nodes split input AGD chunks and allocated output buffers into subchunks, and delegate to a shared executor to maintain constant throughput.

Although nodes defined by Persona can be connected into arbitrary graphs, certain patterns will be more efficient than others. The following subsections describe the graphs used by Persona to build high-performance bioinformatics systems.

I/O Input Subgraph

The input subgraph is designed to keep the process subgraph fed with data while incurring minimal overhead. *Reader* nodes read AGD chunks from mass storage. Currently, Persona supports local disk access or the Ceph object store [128] — other storage systems can be supported simply by writing the interface into a new Reader dataflow node. For disk files, Reader nodes *mmap*s AGD chunk files, producing a handle to a read-only mapped file memory region. For network files, Reader nodes request the chunk files from a storage system (e.g. Ceph), putting each into a recyclable buffer resource. Once a chunk has been read, it passes via a queue to an AGD Parser node, which decompresses and parses the chunk into a useable, in-memory chunk. Chunk objects are then passed to the process subgraph via a central queue.

Processing Subgraphs

Process subgraphs implement the bioinformatics operations, starting with AGD chunk objects as inputs. We describe the implementation of several major functions that are currently implemented in Persona. Since I/O and parsing are provided by upstream and downstream graphs, integrating new or existing bioinformatics functions is usually simple.

1) SNAP Alignment — The Persona SNAP aligner node uses the SNAP short read aligner [132], an open-source tool that is highly optimized for modern servers with a large amount of memory and many cores. To attain maximum performance, each core in the system should be running the SNAP algorithm continuously on AGD chunks, however, we found the granularity of AGD chunks, being optimized for storage, is too coarse for threads and produces work imbalances that lead to stragglers. To remedy this, execution of the alignment algorithm is delegated to an *executor* resource that owns all of the threads and implements a fine-grain task queue (Figure 2.4). Multiple parallel aligner nodes feed chunks to this *executor* and wait for them to be completed. All cores in the system are thus kept running continuously doing meaningful work.

We found this model necessary since the size of AGD chunks, being optimized for storage, is too large to distribute among threads and produces work imbalance that leads to stragglers. The task queue distributes data at a subchunk granularity, which balances the load among the cores and avoids stragglers while limiting queue contention and overhead. By abstracting the CPU threads in this way, we maintain the Tensorflow dataflow abstraction while making the most efficient use of available resources. In addition, the TensorFlow dataflow model executes in steps, which would introduce pauses between alignments and noticeably reduce system throughput. We avoid this by having multiple graph-level aligner nodes running in parallel, in order to keep all CPUs occupied.

When executed, the aligner node receives chunk objects containing reads (base pairs and quality scores), a handle to a buffer pool of output objects, and a handle to the *executor* resource. The chunk object and output buffer are logically divided into subchunks and placed in the *executor* task queue as (subchunk, buffer) pairs. The aligner node uses a pool of threads, each of which dequeues subchunks from the task queue and runs the SNAP algorithm on the reads to align them to the reference genome. Once a full chunk is completed, the originating aligner node is notified, and the result buffer is placed in the subgraph output queue.

2) BWA-MEM Alignment — BWA-MEM [78] is another popular read alignment tool that uses the Burrows-Wheeler transform to efficiently find candidate alignment positions for reads. We integrate BWA-MEM in the same manner as SNAP, using the *executor* resource with a fine-grain task queue (Figure 2.4). We call BWA-MEM alignment functions directly, with only several small cosmetic code changes. For single-read alignment, this approach is straightforward, however for paired reads, BWA-MEM incorporates a single-threaded step over batches of reads to infer additional information about the data[6]. This preprocessing leads to better alignment results but separates the computationally intense multithreaded steps by a sequential computation. Therefore, the *executor* resource for BWA paired alignment divides the system threads among these tasks. We found a balance empirically, but because the computation times are data-dependent, some efficiency can be lost.

[6]Specifically, it estimates the mean (μ) and variance (σ) of the insert size (gap between aligned read pairs) distribution, and it will only fully align pair mates that hit within a window of $[\mu - 4\sigma, \mu + 4\sigma]$.

3) Sorting and Duplicate Marking — Persona also integrates full dataset sorting by various parameters, including mapped read location and read ID. The sort implementation is a simple external merge sort, where several chunks at a time are sorted and merged into temporary file "superchunks". A final merge stage merges superchunks into the final sorted dataset. Persona sort is several times faster than samtools sorting of SAM/BAM files (Section 2.4).

Duplicate finds reads that map to the same location on the reference genome. Duplicate reads can disrupt downstream statistical methods, as the duplicates are typically caused by PCR duplication in the library preparation step before sequencing. Persona duplicate marking uses a hashing technique based on the approach used by Samblaster [53]. Each read is identified uniquely by its *signature* which is comprised of the (contig, position) mapping location. A hash table efficiently finds reads that map to the same location and marks duplicates.

I/O Output Subgraph

The output subgraph mirrors the input subgraph, with Writer nodes writing AGD chunks to disk or a Ceph object store, with an optional compression stage. In general, the process subgraph is responsible for ensuring AGD chunks are properly formatted for a given AGD column, as the Writer nodes are generic and do not perform column-specific formatting.

Persona also implements an output subgraph for the common SAM/BAM format for compatibility with tools that have not been integrated or do not support AGD.

Distributed Computation

Persona uses the TensorFlow distributed runtime for execution across a cluster of servers. In TensorFlow, graph nodes can be mapped to different devices, including remote devices, such as remote server CPUs. Running a graph across a cluster first entails describing the cluster in a config file, and specifying via the Python API which nodes are to run on which server. Any time an edge connects two distributed graphs, the TensorFlow runtime system automatically places sender/receiver nodes to transfer data between the two remote nodes.

Since operations such as alignment are easily parallelized, Persona uses this approach to replicate copies of a pipeline across multiple servers. Each server can then execute alignment operations on individual AGD chunks in parallel with no coordination.

Figure 2.5 shows an overview of the entire system as it executes alignment. A control server receives a request to align a dataset, which provides a list of file paths to the AGD chunks containing the read bases and quality scores. These file paths are loaded into a central queue, which is the only point of coordination in the system, ensuring that each AGD chunk is aligned only once. Thanks to the queues, the TensorFlow runtime ensures that all pipelines on each distributed server execute in parallel. At each execution step, the graph on a remote server will pull a file path from the central queue. Then, Reader nodes access a Ceph distributed

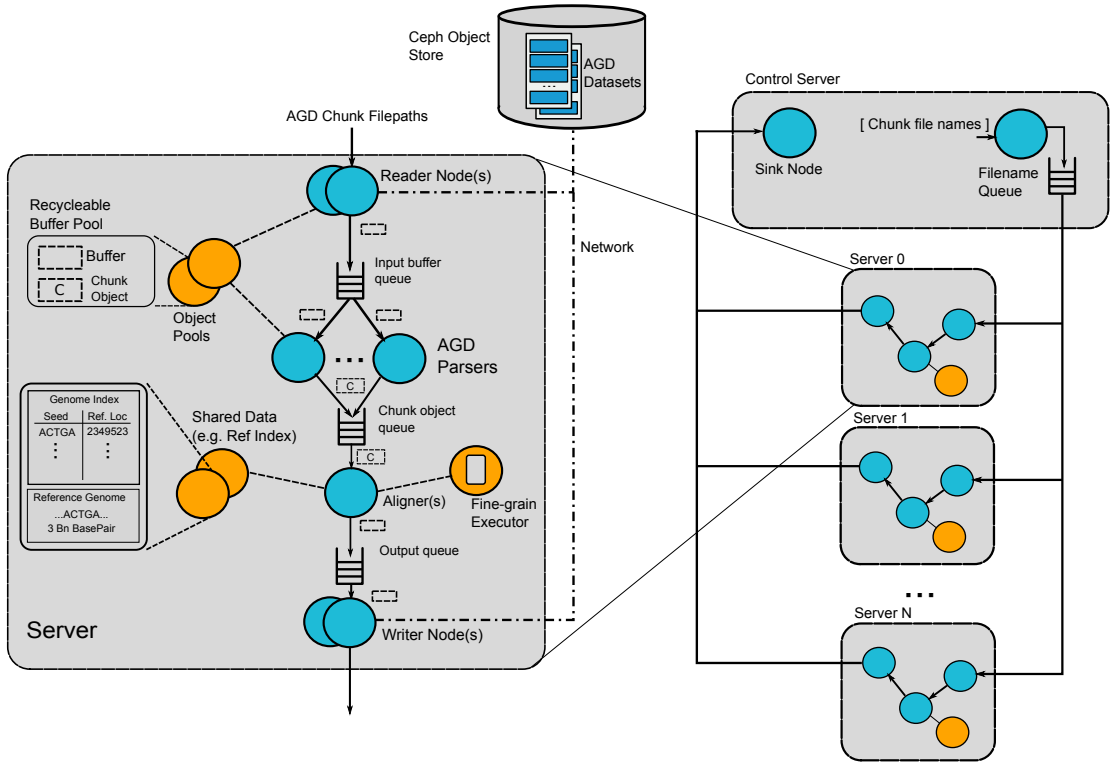


Figure 2.5 – Persona executing alignment across a distributed cluster of servers.

object store and perform the I/O to read the AGD chunk files into memory. Handles to the in-memory data are passed via queues to Parser nodes.

These Parser nodes decompress and parse the AGD files into buffer objects containing the raw reads and quality scores. These are passed via a queue to the alignment nodes, which use the shared in-memory reference genome and fine-grain executor, as described earlier in this section, to align the reads using the available CPU resources. Alignment results are written to in-memory buffer objects. Finally, Writer nodes compress the output AGD chunks and write them to the appropriate output location, in this case, a new column in the same dataset in the Ceph object store.

2.3.1 Discussion

The design of Persona allows it to achieve its design objectives, as will be shown quantitatively in our evaluation in Section 2.4. Persona and AGD both work together to unify bioinformatics data and computations under a single framework. The architecture of Persona also allows it to achieve the crucial design objective of high scalability, with the alignment process scaling linearly across a 32 server cluster. Persona is highly compute-efficient — profiling shows that Persona can dedicate nearly 100% of CPU power to the alignment computations, with the

TensorFlow framework overhead being negligible. I/O and compute overlapping ensure data is always available for computation, and consequently no compute pipeline stages are stalled.

Using TensorFlow as a general dataflow engine was a key design decision that had many benefits but also led to some challenges. Bioinformatics data is not particularly amenable to storage in tensors, the native data type of TensorFlow. Initially, we stored strings of bases, qualities, and metadata in `string` type tensors. However, this led to large amounts of small memory allocations and constant data copying since the `std::string` type owns its data. This prompted the decision to move to the recyclable buffer pooling strategy described previously. In an ideal world, the dataflow engine and runtime of TensorFlow would be separate from the Tensor data type and would allow more general types of data.

The execution semantics of TensorFlow also caused some issues when trying to maximize performance, especially in the multithreaded aligner kernels. TensorFlow executes graphs in steps, where one step executes a given graph and produces one output value. For example, if we want to process two AGD chunks, we must execute the associated graph two times, once for each chunk. However, there is a delay between one execution of a graph and the next. Therefore, parallelism must be used to ensure that threads do not sit idle between executions, which we achieved by running multiple aligner graphs in parallel. A secondary problem was the chunk granularity mismatch between I/O and compute, where chunk sizes optimal for I/O cause imbalance in computation. In the end, we solved both of these issues with a combined solution described in Section 2.3, where all threads executing a given task (e.g. alignment) are owned by a shared resource that can be fed with work by multiple graphs that get executed in parallel by TensorFlow.

Despite these difficulties, we were still pleased overall with TensorFlow. The framework provides numerous features that greatly ease development and optimization, such as node-level profiling, graph visualization, and runtime statistics including current queue states or any other variable one wishes to track. We were also pleasantly surprised at how seamlessly the implementation was able to overlap disk or network I/O with computation. We also found that the dataflow semantics, in general, enforce a high degree of code separation and modularity, which makes for seamless integration of new features e.g. different I/O subsystems.

2.4 Evaluation

In this section, we present a quantitative evaluation of Persona showing that it is a high-performance, scalable system, and can indeed effectively leverage highly parallel commodity clusters for bioinformatics preprocessing workloads.

2.4.1 Experimental Setup

We use a cluster of typical data center server machines, each with two Intel Xeon E5-2680v3 CPU chips at 2.5GHz and 256 GBytes of DRAM. With 12 cores per socket and hyper-threading enabled, each node has 48 logical cores. All machines run Ubuntu 16.04 Xenial Linux. Each machine includes two SSDs in RAID1 configuration for the OS, 6 SATA hard disks (4TB, 7200 RPM, 6 GB/s), a hardware RAID controller, and 10GbE network interface. The six disks are arranged in a 20TB RAID0 configuration, i.e. data is striped across all disks with no replication or fault tolerance, maximizing read/write bandwidth. This provides a maximum aggregate read/write bandwidth of approximately 600 MB/s. For single-node (local) experiments, we store the input data on the 20 TB RAID0 disk array. For distributed (cluster) experiments, we store the AGD dataset in a Ceph distributed object store [128] spread over 7 servers. The Ceph cluster is configured to use 3-way replication and each of its 7 nodes has 10 disks. The compute and storage are connected by a 40GbE-based IP fabric consisting of 8 top-of-rack switches and 3 spine switches.

Persona accesses Ceph objects via the Rados API [12]. Using the `rados bench` tool, we measure the peak Ceph read throughput of our configuration at 6 GB/s, with sequential reads and evenly distributed data.

In all our experiments, we use half of a paired-end whole-genome dataset from Illumina [50] (ERR174324), which consists of 223 million single-end 101-base reads. The dataset is 18 GB in gzipped-FASTQ format and 16 GB in AGD format. The use of single-end read data is an arbitrary choice; the integrated aligners of Persona and AGD also support paired-end alignment. The reference genome to which we align the dataset is the common hg19 human genome [7]. Alignment throughput is measured in bases aligned per second. This is a more realistic reporting than reads per second because that metric does not take into account the length of the read. This makes it difficult to compare against other measures that may use read of a different lengths.

2.4.2 Persona Configuration

Table 2.1 shows the graph-level and TensorFlow configuration in those experiments. All execution uses the TensorFlow direct session, unmodified. For cluster-wide execution, Persona launches a TensorFlow instance per compute server. Within each server, the first stage in the TensorFlow graph fetches a chunk name from the manifest server; the latter is implemented as a simple message queue. Unless noted, the AGD chunk size is 100,000, grouped into 2231 chunks. At this chunk size, both the bases and the qualities are ~3.5 MB. As our performance analysis focuses mainly on alignment, we read only these two columns of each chunk, totaling ~7 MB per chunk. This also shows the advantage of a column-oriented design — we can avoid reading data that is not required by the computation being performed, such as when metadata fields are not required by the alignment computation.

Parameter	Value
Filename queue capacity (chunks)	3
Input buffer queue capacity (chunks)	2
ReadData queue capacity (chunks)	3
Output result queue capacity (chunks)	3
Reader Node parallelism	2
AGD Reader parallelism	2
Aligner Node parallelism	1
Writer Node parallelism	2

Table 2.1 – Parameters used in all experiments.

	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 2.2 – Dataset Alignment Time, Single Server

2.4.3 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 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.

Figures 2.7 and 2.6 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 2.7 and Figure 2.6 show that Persona is CPU bound in both configurations, but that SNAP can only fully use the CPU resource in the RAID0 configuration because a single disk does not provide enough write throughput.

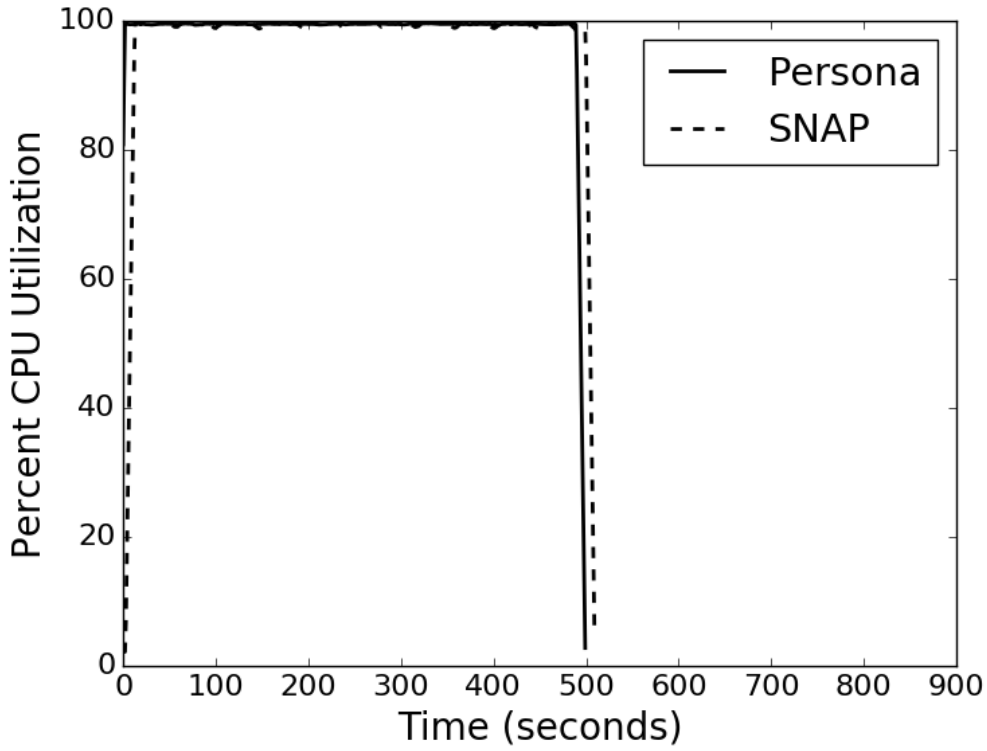


Figure 2.6 – Comparison of SNAP (GZIP'd FASTQ) and Persona (AGD) in CPU utilization with a 6-disk RAID0 array.

In particular, Figure 2.7 shows a cyclical pattern with SNAP where the operating system's buffer cache writeback policy competes with the application-driven data reads; during periods of writeback, the application is unable to read input data fast enough and threads go idle.

Table 2.2 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.

The benefits of column-orientation of AGD are not limited to local disks. Table 2.2 also shows the speedup of 1.54× when the data is stored on Ceph network-attached storage[7]. This also shows another benefit of the modularity of Persona: the Ceph data access functionality is simply implemented as another dataflow node and easily integrated into the system. To do this in SNAP or any other aligner would require significant engineering effort.

[7] SNAP does not natively support reading from Ceph, so we use the `rados` utility to pipe the dataset in gzipped FASTQ format, and feed the resulting SAM file into Ceph.

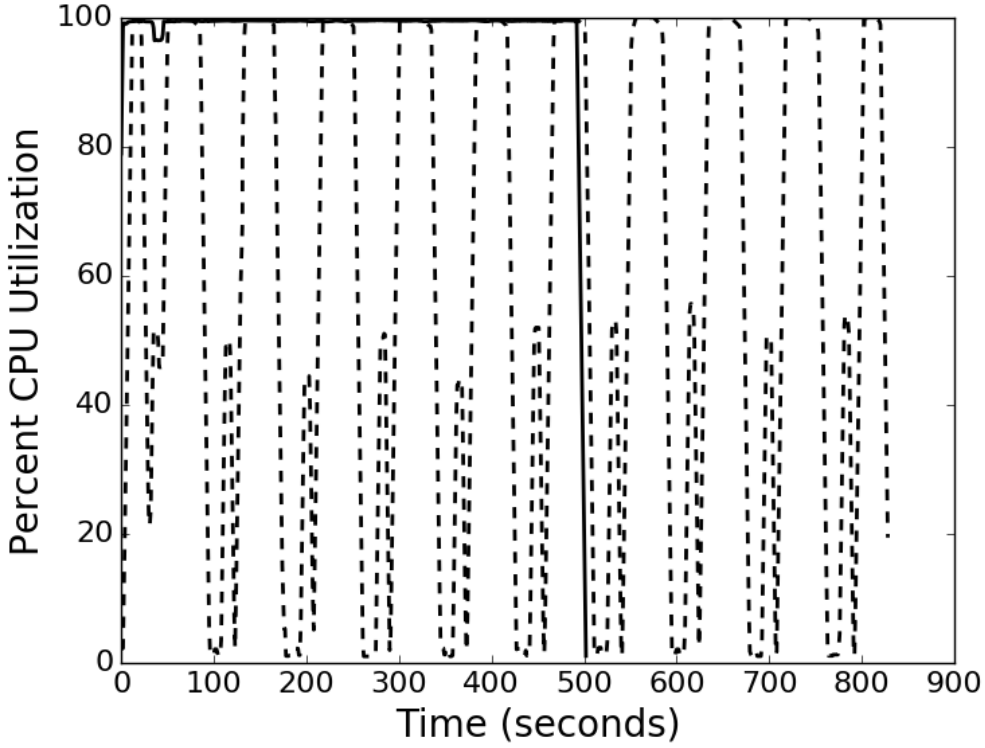


Figure 2.7 – Comparison of SNAP (GZIP'd FASTQ) and Persona (AGD) in CPU utilization with a single disk.

Finally, Table 2.2 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.

2.4.4 Single Node CPU Alignment

We characterize the thread scaling behavior for Persona with both the SNAP and BWA-MEM aligners while comparing them to their standalone baselines. These experiments show that Persona imposes negligible core-scaling overhead on the subsystems we have integrated and avoids thread and I/O saturation issues by efficient overlapping.

Figure 2.8 shows the scalability of standalone SNAP and BWA-MEM compared to Persona as a function of the number of provisioned aligner threads on a 48 core server. The experiments were measured on the RAID0 configuration so that SNAP has enough I/O bandwidth. For SNAP, Figure 2.8 shows clearly (1) a near-linear speedup up to 24 threads, corresponding to the 24 physical processor cores of the server; (2) beyond 24 cores, the 2nd hyper-thread increases the alignment rate of a core by 32%. At 48 threads, however, contention with I/O scheduling causes a drop in performance in SNAP. BWA-MEM experiences a similar drop when no threads are available to perform I/O operations to keep compute threads fed with data. Persona is

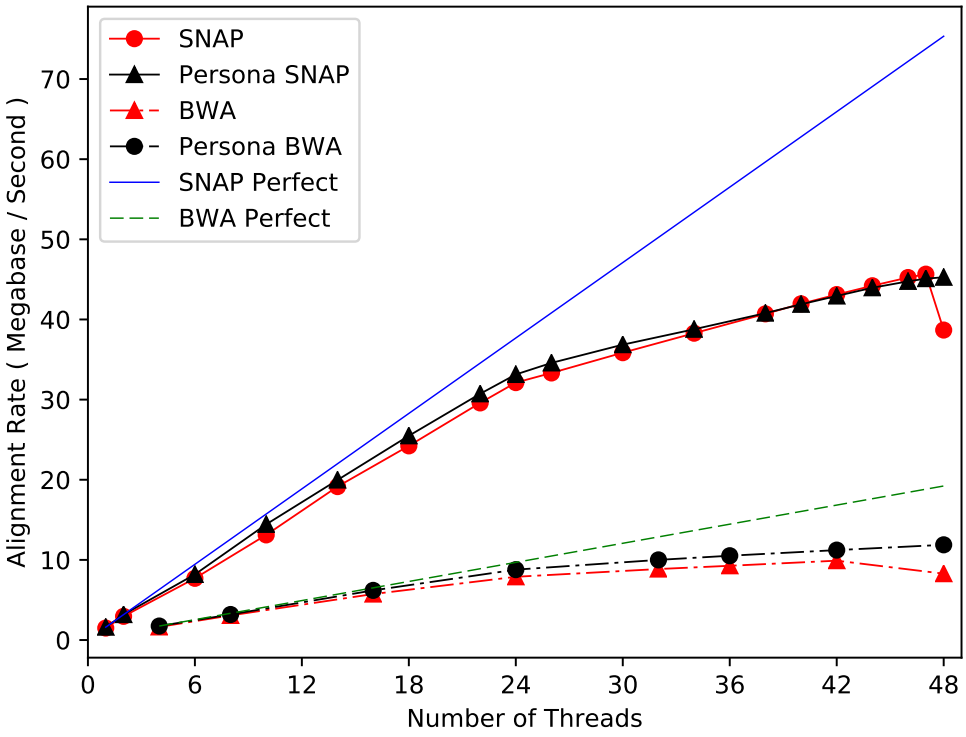


Figure 2.8 – Thread scaling of SNAP and BWA in Persona

less sensitive to operating system kernel thread scheduling decisions thanks to TensorFlow's built-in queue abstractions.

BWA scales well to 24 threads but suffers from high memory contention after hyper-threading kicks in, something that cannot be fixed without significant changes to the codebase. However, because Persona avoids setting up and tearing down threads for different steps of processing, Persona's BWA-MEM subgraph scales better than the standalone program.

2.4.5 Cluster Scalability

Figure 2.9 shows the throughput of two different systems as a function of the number of remote nodes. "Actual" represents the measured performance of Persona using the SNAP alignment node, reported in gigabases aligned per second for a single genome *i.e.*,a measurement of latency. "Simulation" is the ideal speedup line based on the maximum local server performance of ~45.45 megabases aligned per second (see Section 2.4.4).

Persona scales linearly up to the available 32 nodes by making efficient use of all compute resources, hiding all I/O latencies, and addressing the straggler problem through shallow queues. Again, we use SNAP because the higher throughput is better able to exercise the I/O subsystems. The BWA-MEM aligner throughput may be lower per node but it may scale to

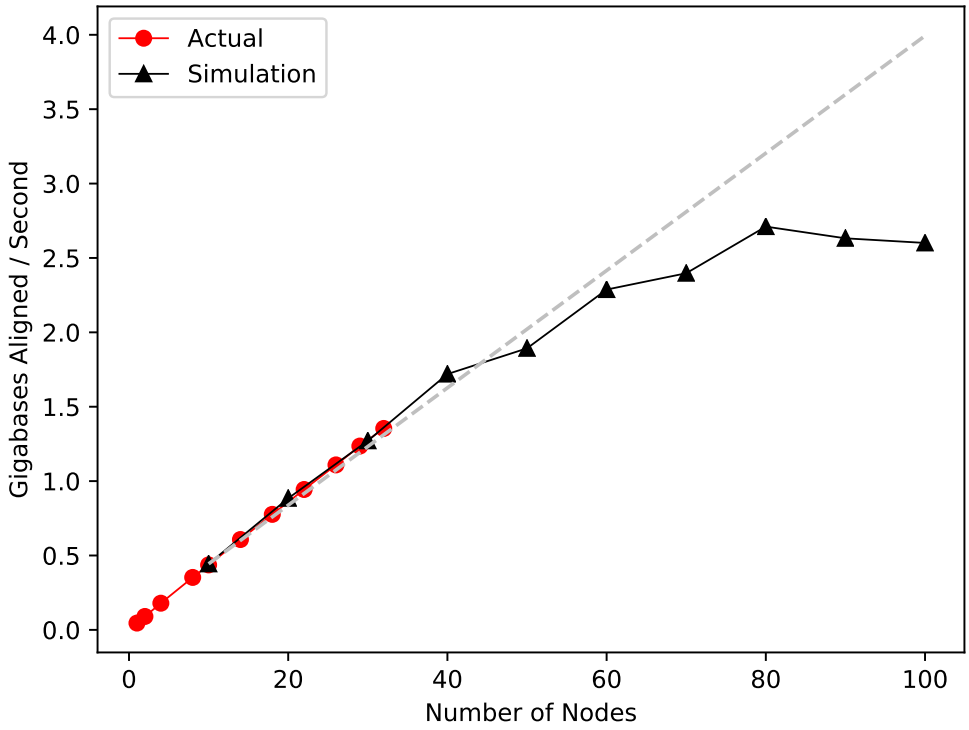


Figure 2.9 – Alignment throughput of Persona (SNAP) across a 32-server cluster. A simulation shows that the system can scale to over 60 servers until the I/O bandwidth limit of the Ceph cluster is reached.

higher numbers of servers because of proportionally lower I/O. We reiterate that our point is not to compare BWA-MEM to SNAP but to show that Persona can scale to a large number of servers by keeping process subgraphs fully supplied with data.

Using 32 servers and the SNAP process subgraph, Persona aligns the genome in 16.7 seconds, from the beginning of the request to when all results are written back to the Ceph cluster. This corresponds to 1.353 gigabases aligned per second. Given that many aligners on single nodes can take hours to align large datasets, this represents a significant gain in throughput performance and single dataset latency and is possibly the fastest whole-genome alignment to date.

We use a different methodology to test the scalability of the storage cluster. Given the alignment rate per server, we do not have enough machines in our cluster to saturate the storage cluster. In order to see where its limits are, we deployed multiple "virtual" TensorFlow sessions per server and replaced the CPU-intensive SNAP algorithm with a stub that simply suspends execution for the mean time required to align a chunk, and then outputs a representative (but incorrect) result. In this sense, the computation becomes simulated, while all I/O (disk, but the network I/O in particular) is performed as normal. Since the chunks are sized large enough

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 2.3 – Dataset Sort Time in Seconds, Single Server

to mask any long-latency outlier reads, the mean alignment time has low enough variance that the simulation model is accurate.

Figure 2.9 shows the results in the "Simulation" line. We first validate that the simulation matches the "Actual" measurements up to 32 nodes. We then observe that the Ceph cluster scales to ~60 nodes without loss of efficiency. Beyond 60 nodes, with an AGD chunk size of 100,000 reads, writing alignment results back to the Ceph storage cluster is the limiting factor. We again see that the TensorFlow framework imposes negligible overheads.

2.4.6 Sorting and Duplicate Marking

We also compare Persona in sorting performance to Samtools [81] and Picard [3], standard utilities for sorting SAM/BAM files. Table 2.3 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 plus conversion times from SAM to BAM format. The conversion time is included because it is a real overhead that bioinformatics pipelines need to deal with. Persona can directly process aligned results in AGD, performing the sorting operation up to 2.32 times faster than Samtools including the file conversion time. The sort implementation of Persona is currently naive, using `std::sort()` across chunks. These results can probably be improved substantially.

We compare the duplicate marking performance of Persona to Samblaster [53], whose algorithm is employed in our implementation. Recall that this algorithm involves forming keys out of read alignment positions and uses hashing to detect duplicates. Samblaster can mark duplicates at 364,963 reads per second, while Persona, which uses Google's optimized dense hash table, can mark duplicates at 1.36 million reads per second. Note that Persona also uses less I/O since only the results column needs to be read/written from the AGD dataset, again showing the advantages of using AGD.

2.4.7 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. Persona can import FASTQ to AGD at 360 MB/s, while BAM format files are produced from AGD at 82 MB/s. BAM

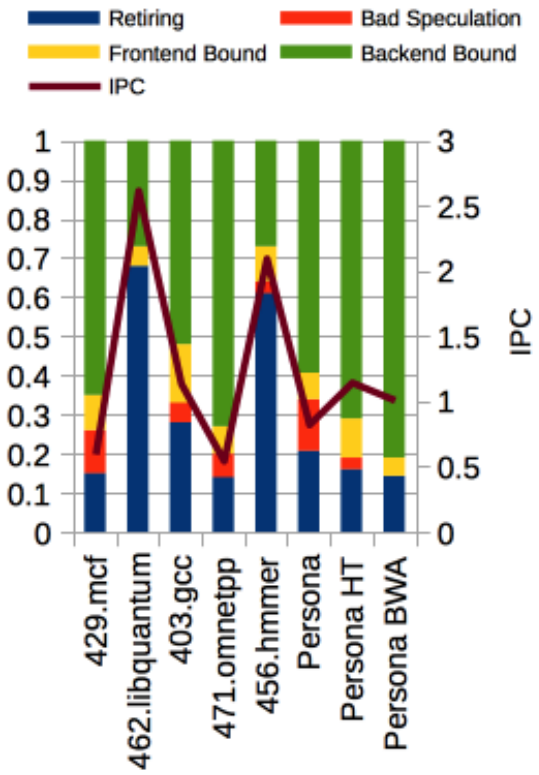


Figure 2.10 – Persona profiling analysis breakdown

export is slower because a significant amount of time is spent compressing the output as required by the format. Still, the 16GB AGD dataset can be completely exported to BAM in only 3.5 minutes. The operation is difficult to scale out because the output BAM file is a monolithic block.

2.5 Analysis & Discussion

This section presents a performance analysis of Persona running both SNAP and BWA-MEM, which we undertook to better understand system performance and possible avenues for improvement. We also discuss several attempts to accelerate the system, including using different hardware, namely the Intel Xeon Phi many-core system [71]. Lastly, we provide a Total Cost of Ownership (TCO) analysis showing that Persona/SNAP running on a commodity cluster is an extremely cost-efficient approach to executing alignment in WGS preprocessing.

2.5.1 SNAP and BWA-MEM Profiling

Our performance analysis focuses on alignment, as it is the most compute-intense step we have integrated into Persona. As this operation is the primary system bottleneck, we used

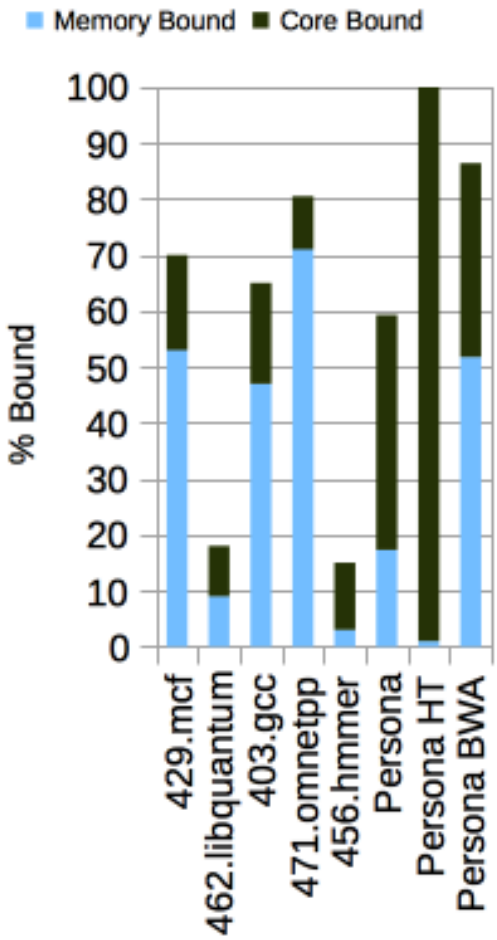


Figure 2.11 – Workload analysis (with and without Hyperthreading) compared to several SPEC benchmarks.

Intel's VTune Amplifier [107] to profile both BWA-MEM and SNAP while running in Persona, to identify any possible avenues for improvement. Figures 2.10 and 2.11 summarize the findings, while displaying several relevant SPEC benchmarks for comparison.

Both aligner profiles share some similarities, in that they are heavily CPU backend-bound *i.e.*,many cycles stalled due to lack of resources for accepting μOps into the pipeline such as D-cache misses or busy functional units. With SNAP, we see that the issue is due to the core and not memory access — primarily, this is due to short but frequent calls to a local alignment edit distance function that has a high proportion of integer arithmetic instructions causing functional unit contention, and many data-dependent instructions and branches. Figure 2.12 illustrates this from another perspective, showing memory bandwidth values (again measured using VTune) as we scale the number of threads while running SNAP. Note that this measurement is an aggregate of both CPU packages in the server. Memory bandwidth scales linearly with the number of cores, does not saturate, and is therefore not a barrier to

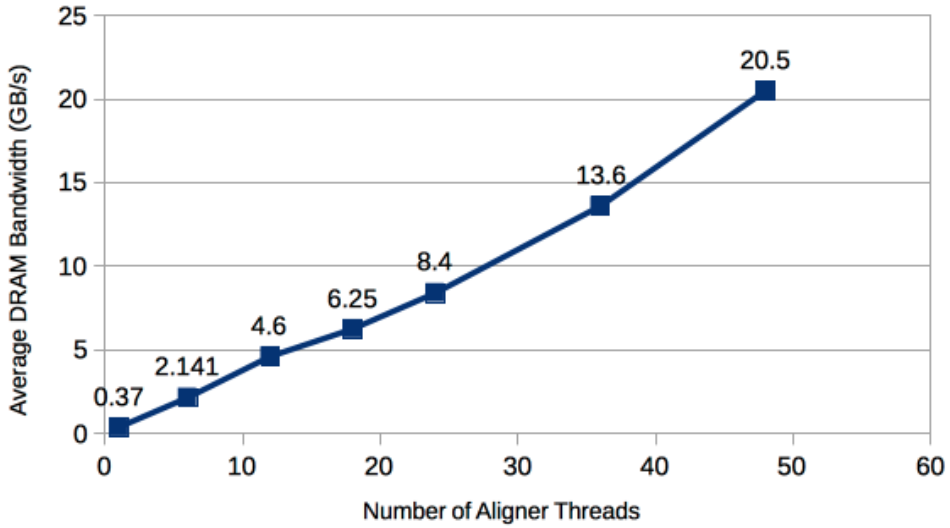


Figure 2.12 – Measured memory bandwidth of SNAP as the number of threads scale.

core scaling for the tool. Figure 2.12 shows that memory bandwidth is relatively low for the alignment workload. The per-core bandwidth required to keep the alignment threads busy is approximately 500 MB/s, and the peak bandwidth (with 48 threads performing alignment) is 20.5 GB/s. This is 15.1% of the available system bandwidth of 136 GB/s.

In BWA-MEM, the system is more memory bound. VTune reports that this is primarily due to cache and DTLB misses, and our findings corroborate previous analyses [135]. Their analysis showed that BWA is primarily limited by memory bandwidth because of irregular access patterns stemming from the suffix tree structure it uses to find potential alignment locations. The tree structure memory access patterns tend to be disjoint, as a child tree node is not likely to be contiguous with, or even near its parent. There is little locality, and caching, therefore, provides little value with a majority of accesses go all the way to DRAM.

From profiling, we found that despite its high performance, Persona/ SNAP exhibits little ILP[8], running with an IPC (Instructions per Cycle) of approximately 1. The heart of this computation is a simple routine that compares each read against potential matches in the reference genome and selects the best possibilities using an optimized linear programming technique called Landau-Vishkin [73]. Within this computation, an appreciable fraction of time is spent in short string comparisons. The architecture of modern Intel Xeon x64 servers and the Xeon Phi Knights Landing server are not well matched to this computation for a variety of reasons:

- It is difficult to express the inherent fine-grain parallelism in a linear programming algorithm. This computation has three independent sub-computations (extending the

[8] Instruction-Level Parallelism — modern processors can execute independent instructions of the same thread in parallel

three nearest, previously computed neighbors in the linear programming matrix and selecting the best), each of which involves multiple short string comparisons. A compiler cannot fuse these loops to schedule their instructions since the loop bounds differ. And, even if this was possible, existing processors do not have enough resources to run the three computations simultaneously.

- The vector (SSE/AVX) string comparison instructions, despite doubling the granularity of comparison from 64 to 128 bits, performs worse than scalar code because the distance to the first mismatch is typically small and the vector instructions have long latencies. Hyper-threading exacerbates the problem because of contention for the vector unit.

- The computation is fine-grained (average of 660 nanoseconds) and potentially accesses a large amount of data (anywhere in the 40 GB reference genome) with little locality, so the large L3 caches are of little value.

2.5.2 Other Microarchitectures

We also tested Persona on the Xeon Phi processor ("Knights Landing") [71]. At the time, we had only a pre-release version of this processor, running at a reduced clock rate of 1.3 Ghz, but its 64 cores are 4-way hyperthreaded with dual 512 bit wide SIMD units. This corresponds to a total of 256 hyperthreads. We found that thread scaling fell off after 128 threads (2 per core) due to functional unit contention, i.e. there are more instructions in the stream ready to execute than hardware functional units to execute them. Overall performance is significantly lower than the standard Xeon. VTune again showed that the workload is highly core-bound. The inner alignment loops where most time is spent have a low diversity of instructions, easily saturating available functional units. While the alignment workload scaled across cores and hyperthreads, for this application, a high clock rate is more beneficial for performance than more hyperthreads or improved vector units.

GPU architectures may appear attractive for parallel workloads, however, the alignment computations are irregular and filled with data-dependent branches. Thread divergence would likely lead to minimal SIMD performance gains.

2.5.3 TCO of Cluster Architectures

Personalized medicine is becoming practical because of dramatic decreases in the cost of genome sequencing. In light of these decreases, it is worth considering the cost contributions of the storage and computation required to enable personalized/precision medicine. We consider three cases: a single system attached to a single NGS sequencer, our balanced cluster, and a nation-wide solution. All cost figures are as of 2017. We limit the analysis to alignment, the most expensive computation we have yet integrated into Persona.

Item	Unit cost	Units	Total
Compute Server	$8,450	60	$507K
Storage server	$7,575	7	$53K
Fabric ports	$792	67	$53K
Total			$613K
TCO(5yr) [64]			$943K
Cost/Alignment (100% Utilization)			6.07¢

Table 2.4 – Cluster TCO and alignment costs. The storage cluster has 126 TB of usable capacity, corresponding to approximately 6,000 sequenced genomes.

First, Figure 2.7 shows the performance of a single server, where genomic data is stored, aligned, and processed on a local machine. A single server can, therefore, align ~144 sequenced genomes per day, if we assume that the average genome size is the same that we have used to perform our measurements. Considering the total cost of ownership (TCO) of the server over 5 years, this implies a cost of 4.1¢ per alignment, assuming full utilization. Note that this scenario has limited genome storage capacity.

Second, there are economies of scale for sequencing, and a more likely scenario would be a regional center providing sequencing, processing, and storage services. A small cluster and network storage subsystem, as we have used in our experiments, could support 5173 alignments per day. Figure 2.9 shows that our storage cluster can sustain the I/O requests of a cluster of twice this size, offering expansion capacity. Table 2.4 summarizes the cluster compute and storage costs over a 5 year lifetime. For the network fabric cost, we determine the per-port cost of the 8-TOR, 3-spine architecture deployed in our physical cluster, and multiply by the number of ports used. Table 2.4 shows that, assuming the system is fully loaded, the TCO of a genome alignment on such a regional cluster is 6.07¢, higher than above because of the larger storage subsystem needed to support the throughput.

Third, a nation-wide solution would be needed to support initiatives such as Genomics England's 100,000 Genomes [57]. For this, additional storage is required as our balanced cluster has a usable capacity of 126 TB, which can store 6,000 in AGD format (1 day worth of sequencing). One can use the 60:7 ratio of computing to storage machines as a "not to exceed" scaling guide. The TCO model of Table 2.4 can be adjusted to estimate the capacity and throughput requirements of a deployment.

Storage is the dominant cost of a cluster and therefore of WGS preprocessing as well. With our current high-throughput storage subsystem, the cost per genome for storage is $8.83, two orders of magnitude higher than the alignment cost. Genomes that are not being actively processed could be stored in a tiered storage system using slower, lower-cost storage and erasure coding [48]. Currently, using Amazon Glacier storage ($0.007 GB/month [1]), a full genome could be stored for 5 years for $6.72, only slightly less expensive than locally hosted

storage. However, a large scale storage-as-a-service solution like that of Glacier has the benefit of providing fault tolerance and a guarantee that data will not be lost, something that a local solution would find difficult or expensive to provide. Note that with higher coverage datasets, storage amounts and cost would increase.

Computation is far from the dominant contribution to the cost of sequencing a genome. Storage, while more expensive, is still far from a significant expense, but if the cost of sequencing continues to decline at its faster-than-Moore's-Law rate, storage may become the limiting factor in widespread genome sequencing. Novel compression techniques or more storage-efficient formats will likely be required. A good example is reference-based compression [55], where only the differences between reads and the reference are stored. When reads are "decompressed", they are simply reconstructed using the stored diffs and the reference bases. This can reduce storage requirements by as much as 20×.

2.6 Related Work

While we have already detailed some popular alternative solutions for building bioinformatics pipelines, this section provides a comprehensive overview of previous work related to Persona.

Because of its potential, bioinformatics and genomics have been the topic of much research. Large organizations such as the Broad Institute have established pipelines (Genome Analysis Toolkit [6]), a system similar to Persona. GATK also employs sharding for parallel data access (via an underlying distributed file system, HDFS[9]), but uses the standard SAM/BAM formats, often merging multiple input files into single files, which can limit scalability. Recently, GATK has also been ported a cloud environment, Google Genomics [2]. Microsoft also advertises cloud-based genomics capabilities [89]. However, these companies have not released details of their internal systems architectures, so it is unclear how they compare.

In terms of file formats, the recent ADAM format [87] is most similar to AGD. It also uses a column store format to achieve better compression. Data is serialized using a common framework (Avro) that supports multiple languages and is easily parsed. ADAM relies on Spark and HDFS for distributed computation, again restricting users to a single storage subsystem type. In terms of performance, ADAM claims a ~2× speedup over Picard in single node sorting, whereas Persona achieves a ~5× speedup. HDF5 [122] is a general-purpose hierarchical file format that can also support a bioinformatics schema similar to ADAM. In contrast to AGD, it restricts users to MPI for multiprocessing and is difficult to tune for high performance. TileDB [99] is a system that stores multi-dimensional array data in fixed-size data tiles, similar to HDF5 but superior in write performance and concurrency. TileDB "chunking" is similar to AGD, but it employs a more rigid data model and is generally much more complex. Parallel access is implemented using MPI as in HDF5. Furthermore, GenomicsDB [66] is built on

[9] Hadoop Distributed File System [114], a cluster data storage approach that shards files across cluster servers, to facilitate a processing model where computation is "moved" to where the data is.

TileDB to store genomic variant data in 2D arrays, columns and rows correspond to genome positions and samples, respectively.

AGD differs substantially from these formats in that it is simple and requires only a way to store keyed chunks of data. The AGD API to access chunk data can simply be layered on top of different storage or file systems, using those system's APIs for parallel access, distribution, replication, etc.

Distributed alignment has been explored before, for example, CloudBurst [111], which uses Hadoop MapReduce. 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 their alignment algorithm is different, their read size is different (36 base pairs versus our 101), and their cluster architecture and CPU is different. Cloud-Scale BWAMEM [42] 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). SparkBWA [26] is similar, scaling alignment out over a Spark cluster, but not achieving linear scaling. ParSRA [60] shows close to linear scaling using a PGAS[10] approach, but relies on FUSE to split input files among nodes. Eoulsan [72] uses MapReduce to perform several pipeline steps and supports different aligners. Pmap [68] uses MPI to scale several different aligners across servers and claims linear scaling.

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. 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 relatively 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 approach will quickly become inefficient.

Other efforts include SAND [93], 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 [104]. Others have shown that aligning reads to a reference genome scales linearly [63]. merAligner [58] 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 [123] reports 92 million bases aligned per second on a single machine, faster than even SNAP, however, it is a closed-source proprietary product.

[10] Partitioned Global Address Space, a parallel programming model that uses a global address space over physically partitioned memory.

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. The 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.

2.7 Conclusion

Current NGS technologies produce vast amounts of data that require a great deal of computation to analyze. Many current approaches to bioinformatics processing cannot keep up with the growth of data from sequencing machines. This chapter has shown that the existing state of the art algorithms and applications can, when embedded in the Persona distributed dataflow framework, scale to the required levels.

Persona can align short reads at a rate of 1.353 gigabases per second using 32 commodity servers, completing a 223 million read dataset in 16.7 seconds. When scaled up, Persona executes alignment extremely cost-efficiently, at only 6.07 cents per aligned dataset. Persona shows definitively that horizontally scaling across commodity clusters is the easiest, more compute-efficient, and most cost-efficient method of performing the large-scale bioinformatics computations of the future.

3 Optimizing Heap Performance with Memoro

The preceding chapter has shown that it is possible to scale bioinformatics computations in a compute-efficient and cost-efficient manner. While parallelism and scaling are crucial, programs must also make efficient use of the local CPU resources — well-optimized software can compound the benefits of scaling.

This is true for all software systems, but bioinformatics may, in fact, benefit disproportionately from optimization. Many commonly used tools in the domain are open-source and are developed by people who are not experienced software engineers. This can leave opportunities open for optimization.

For example, when evaluating a particular program for integration into Persona, we felt the program was underperforming, given its algorithm. We then used a profiling tool to understand where the program was spending its time and why it was so slow. What we found was surprising: the program spent over 30% of its run time allocating and deallocating memory. This left us with many questions. Why was the program making so many allocations? Was it actually using this memory? How was it using this memory? Which locations in the source code were most responsible for inefficient heap allocations? More generally, is it possible to automatically detect and quantify inefficient usage of memory of the heap?

To help answer these questions, for both bioinformatics and other software, we built a new, detailed heap profiler called Memoro. Memoro uses a combination of static instrumentation, subroutine interception, and runtime data collection to build a clear picture of exactly when and where a program performs heap allocation and, crucially, *how* it uses that memory. Memoro also introduces a new visualization application that can distill collected data into *scores* and visual cues that assist developers in quickly pinpointing and eliminating inefficient heap usage in their software. Our evaluation and experience with several applications demonstrates that Memoro can reduce heap usage and produce runtime improvements of up to 10%, allowing bioinformatics and other applications to make better use of the resources available on commodity cluster machines.

Bibliographic Note

This chapter is based on the paper *Detailed Heap Profiling*, previously published and presented at the International Symposium on Memory Management in 2018 [40].

3.1 Heap Memory Use and Profiling

Modern software relies heavily on heap memory; even small applications can perform millions of allocations at thousands of different locations within their code. Inefficient use of dynamically allocated memory can increase both peak memory usage and program run time. For example, unused memory that remains allocated wastes space, or a program in a loop might continuously allocate and deallocate memory unnecessarily, wasting time. Understanding and fixing memory allocation problems is not simple because libraries, frameworks, and packages hide internal memory allocations inside abstraction boundaries, which makes discovering and fixing problems challenging, particularly in large, complex systems. As a result, it is very easy for inefficient heap usage and performance-adverse allocations to go completely unnoticed. Even when a developer is actively looking for performance problems, they can also be difficult to find. Development tools, however, can help find and fix these issues. This chapter describes a dynamic tool called Memoro that tracks and analyzes memory allocations and usage, and visualizes the resulting data, to aid in identifying and correcting memory allocation, use, and deallocation defects.

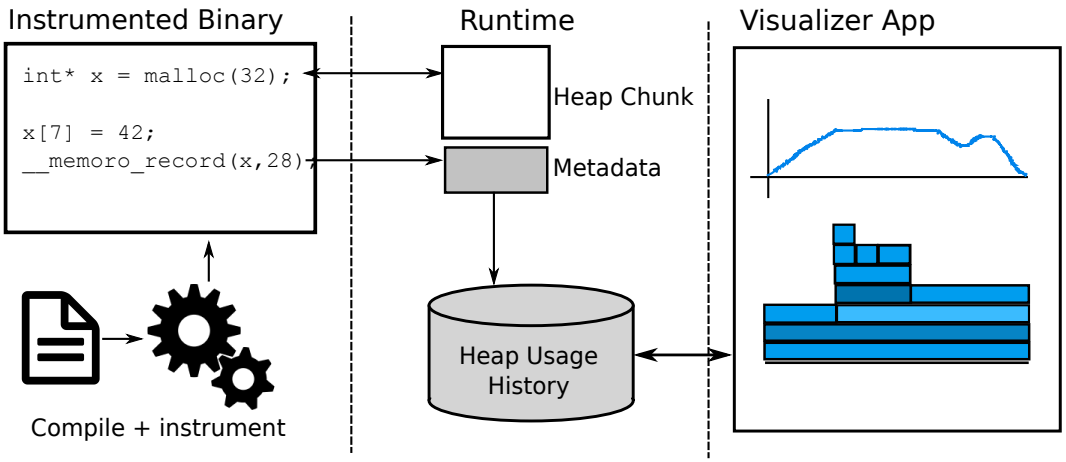


Figure 3.1 – Overview of Memoro operation — The runtime intercepts allocation calls and updates metadata at every access to a heap chunk. Accesses are detected by program instrumentation inserted by the compiler. Data is dumped to disk and visualized using a secondary *visualizer* program.

Most existing tools for profiling heap allocations provide only a simple, one-dimensional perspective on memory allocation. Typically, they report how many bytes of memory are allocated at each allocation call site and provide a mechanism for aggregating allocations up a dynamic call graph.Tools for managed languages may also report statistics related to the garbage collector. This data, while often quite informative, is lacking richness and the insight necessary to understand many memory performance problems. Other methods of data aggregation and presentations can offer deeper insights into program behavior. For example, objects of the same type may be allocated at many points in a program; and to understand the performance effects of these objects, it is often helpful to aggregate all allocations of the same type of object. It is also valuable to know how *efficiently* a program uses the words of memory that it allocates:

- How much has the program actually written and read the memory (i.e., was this allocation necessary)?
- How much of the allocated memory block was accessed by the program?
- Was the memory block write-only? Read-only?
- What was the ratio of reads to writes?
- Was the memory block accessed by multiple threads?

Finally, there are dynamic patterns of memory usage that strongly hint at program performance problems. For example, allocating an insufficiently large region of memory and repeatedly growing it leads to unnecessary data copying. Or, freeing an object long after its final access prolongs its memory block's lifetime and increases memory usage.

Figure 3.2 shows a visual representation of some of these patterns. Chunks 1, 2, and 3 show a typical buffer growth reallocation pattern. Chunk 1 lives only for a short time until it is copied into newly allocated Chunk 2 and then deleted. Eventually, Chunk 2 is filled and is copied into newly allocated and larger Chunk 3, and so on, until the maximum required size is reached. This growth pattern is typical of common constructs like `std::vector`, which keep their contents in a contiguous buffer and reallocate and copy when the current maximum size is reached.

Chunk 5 in Figure 3.2 shows a pattern of continuous allocations and deallocations, where chunks are allocated and then deallocated very quickly — they have short lifetimes. This pattern can be observed when an allocation executes inside a loop. In this case, it would be better to allocate one chunk outside the loop and reuse it, rather than waste time in the allocator every loop iteration.

Chunk 4 illustrates other inefficient uses of a single chunk in time and space. It is read and written for only a small part of its actual lifetime. It may be more efficient for the program to

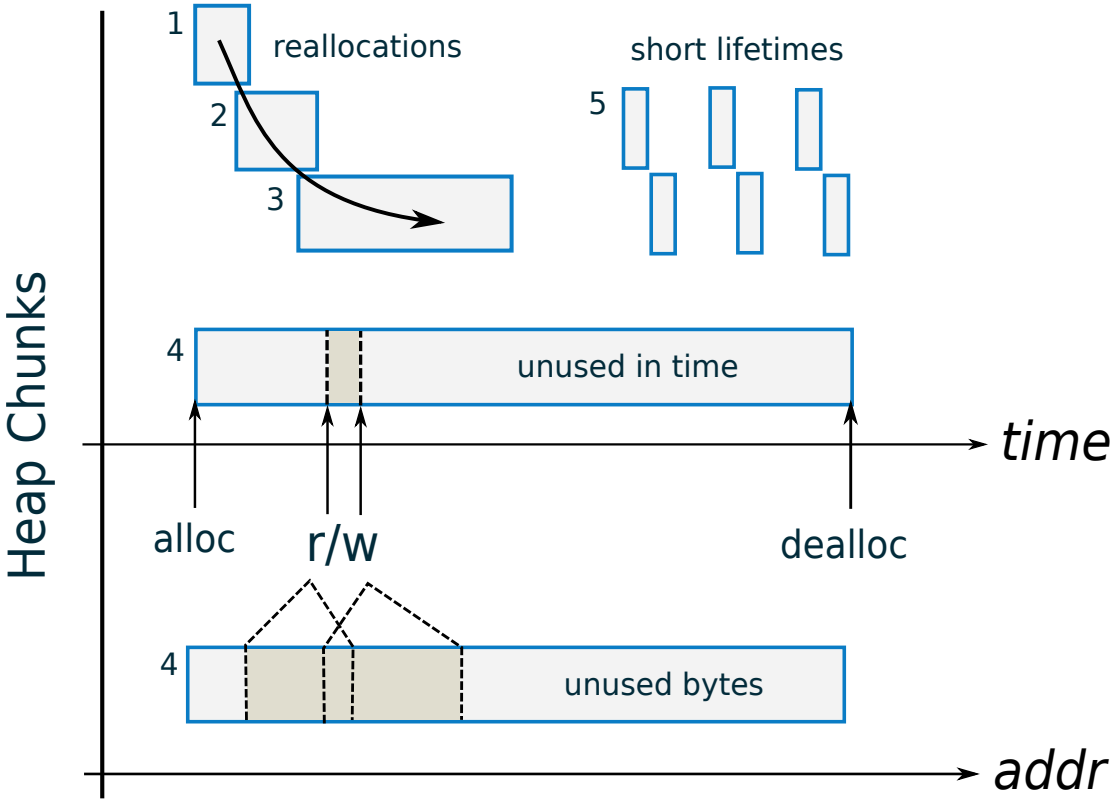


Figure 3.2 – Chunks of allocated heap memory visualized in time and space.

allocate it later or deallocate it sooner, to reduce aggregate maximum memory usage. Likewise, in space, these reads and writes access only a small portion of the allocated bytes and many bytes go unused. It may be possible for the program to allocate fewer bytes and reduce total memory use.

While memory allocation in managed (garbage collected) languages has been heavily studied, fewer tools are available for unmanaged languages, despite the increasing use of implicit memory allocation in C++ and other native libraries. This chapter focuses on memory performance problems in languages such as C and C++, with explicit memory allocation and deallocation. Most recent research on explicit allocation has focused on correctness concerns such as premature deallocation or data races. This chapter, by contrast, focuses on allocation and deallocation performance problems, and other performance problems that stem from the inefficient use of allocated heap memory.

Memoro is a new heap profiling system that provides a developer with a meaningful, quantitative analysis of a program's heap usage *and* indications of how efficiently it uses the heap. Figure 3.1 shows an overview of Memoro. It uses a combination of function call interception, static compiler instrumentation, and runtime data collection and analysis to capture detailed information about heap allocations and the use of allocated memory. This detailed information is then distilled into a set of *scores* that measure heap usage efficiency in several categories.

A cross-platform Memoro visualizer presents both summary and detailed information in a concise and effective format using graphs and other visual elements, allowing developers to quickly pinpoint potential problems. The visualizer also works in conjunction with the runtime to attach associated allocation object types to allocation points, allowing the user to aggregate heap usage by object type, a feature missing in other heap profilers for native code.

While Memoro collects a large amount of data, careful static instrumentation keeps runtime overheads at an acceptable level, similar to existing, less informative heap profiling solutions. The Memoro implementation is based on the LLVM/Clang AddressSanitizer framework [112] and is thus portable to any system that the sanitizer framework supports. Since the instrumentation and runtime are in the compiler back-end, other language front-ends should be able to use Memoro as well.

All of Memoro (the compiler instrumentation and runtime as well as the visualizer software) is open source and available [24].

3.2 Related Work

Memoro extends the functionality of existing profilers, as heap profiling is not a new concept. This section will review existing systems and literature in the area of heap profiling and analysis.

One of the best-known systems for profiling program behavior is Valgrind [96]. Valgrind is a framework for the dynamic analysis of binaries. It translates a binary at runtime into an intermediate representation and allows other tools to insert instrumentation to analyze and perform measurements on the executed instruction stream. Massif [19] is a tool that uses Valgrind for heap profiling. It tracks how many bytes of memory each source code line has allocated at specific points in time (snapshots). Third-party tools exist to visualize Massif data, but since it is snapshot-based, high-frequency events can be missed if they happen between snapshots. Memoro, on the other hand, continuously monitors all heap events, retaining maximum data fidelity. Any aggregation or windowing of data is done interactively after profiling a program. Another Valgrind tool called Dynamic Heap Analysis Tool (DHAT) [21] attempts to analyze heap usage efficiency and collects data similar to Memoro. DHAT only produces text output that is often difficult to understand, and it does not allow aggregation by data type. In addition, for large programs with many allocations, DHAT provides no mechanism to help programmers identify the most inefficient allocation points. Another drawback of Valgrind-based solutions is their high overhead, which can be upwards of 50×.

Other heap profilers intercept and redefine allocation routines in a runtime library that can be used with any executable. Google Perftools [15] uses this approach; as does HeapTrack [16], a Linux heap profiler; and MTuner [124], a Windows heap profiler. These tools have higher resolution than Massif as they do not take snapshots, and have much lower runtime overhead because they only intercept allocation functions. Unlike Memoro, they do not collect any in-

formation as to how a program used its heap memory, as they do not add any instrumentation capable of gathering and recording memory references.

In managed languages, Shaham et al. profiled Java heap allocations to perform more timely garbage collection and reduce memory consumption [113]. Chis et al. use a *ContainerOr-Contained* relation to detect high-impact patterns within Java heaps [43], some of which are similar to the patterns that Memoro detects. *Blended Program Analysis* [49] also makes use of static analysis in conjunction with a dynamic representation of the program call structure to better understand Java application performance.

Several tools use static instrumentation to analyze programs to find memory access bugs like overflows and out-of-bounds. Of particular note is the AddressSanitizer framework [112], which we have used to build Memoro. AddressSanitizer uses static instrumentation and a specialized allocator to detect memory errors such as out-of-bounds accesses and use-after-free. Other tools built on AddressSanitizer include a leak detector and a race condition detector.

DINAMITE [90] is a system that also uses compile-time instrumentation to trace memory accesses, allocations, and function calls, to analyze memory performance bottlenecks. It does not focus particularly on heap memory, and suffers from high overhead as well (36× to 537×, depending on the level of analysis).

GCSpy [105] is a tool that provides visualizations of the heap, focusing on garbage collected languages. A server API implementation is required for languages other than Java, and it does not appear to analyze how efficiently a program uses its heap objects.

Overall, Memoro goes beyond existing solutions by providing detailed profiling *and* heap usage efficiency analysis in a low-overhead package. Most heap profilers for native languages use allocation routine interception to build a time-based log of heap allocations and deallocations. While this approach can track heap usage through time, it is typically the only data collected and displayed to a developer. The developer thus lacks insight into how their program actually uses the bytes that it allocated. Gleaning this information from source code, by trying to guess what objects were accessed where and when, can be time consuming and error-prone, and it is often impractical on large projects in which developers may be unfamiliar with a majority of the code.

In addition to all of these concerns, existing profilers lack a means of mapping heap allocations to object types. Since objects of the same type can be allocated at different places in a program, without this information, there is no easy way to view their heap usage holistically.

Finally, there are few if any heap profilers (or associated data visualization tools) that are truly cross-platform.

3.3 Memoro Profiler

The Memoro profiler system provides developers with data, analyses, and visualizations that allow quick identification and diagnosis of inefficient heap usage and poor allocation strategies.

Inefficient heap usage can occur in many ways. For example, a program may allocate memory that it never writes or reads; it may be possible to eliminate these allocations. Similarly, a program may allocate memory that it only writes and never reads. A program may allocate a chunk of memory but only use a small fraction of it, effectively wasting the rest. Objects may also live on the heap long after their last reference, taking up space that could have been reused. Alternatively, objects may be allocated long before their first usage, wasting memory in another manner.

Poor allocation strategies include repeatedly allocating and deallocating an object and successively growing a buffer to larger sizes. All of these dysfunctional allocation strategies can have an impact on performance, both directly because of the time spent in allocation routines and indirectly by fragmenting the heap and making subsequent allocations more expensive.

Any program transformations to boost heap usage efficiency described in this chapter are still ultimately at the discretion of a developer. A program transformation is correct only if it does not change program behavior on all possible execution paths. Since dynamic analysis tools capture program behavior only along a subset of these paths, we assume that the developer will examine the program and find a correct transformation. It may be possible to build a tool to analyze and implement these restructurings, but that is beyond the scope of this thesis.

In this section, we will describe the data that is captured by Memoro and how it is collected. Analysis and visualization of the data is described in Section 3.4. We provide the following terminology and definitions to avoid confusion:

- **Allocation Point**: a location in a program that explicitly allocates memory (e.g. `malloc`, `new`), fully identified by its dynamic call stack trace. The stack trace disambiguates different call sequences leading to an allocation point [30] and provides the basis for aggregating chunks with the same call context. It is unique up to intra-procedural control flow and loop iteration counts. We did not find path-sensitive information to be necessary.

- **Chunk**: A region of heap memory returned at an allocation point. An allocation point produces multiple chunks in consecutive calls.

- **(Chunk) Metadata**: The data pertaining to a specific chunk.

3.3.1 Data Collection

Similar to some other profilers, Memoro builds a time-indexed log of all heap allocations and deallocations in a program. This is done by intercepting calls on the standard allocation routines (`new, malloc`, etc). For each allocation, the stack trace is logged [30] along with the allocation size and the time since the start of program execution. Time is measured in CPU clock cycles[1] Each unique allocation point is associated with all chunks that it allocated over the execution of the program, including the chunks that were freed.

To discover inefficient heap usage and multi-thread accesses, additional information is required about how and when a program accesses heap chunks. More specifically, Memoro records the following metadata values for each chunk allocated:

- The time at which this chunk was allocated (8 bytes).
- The chunk size (8 bytes).
- A unique ID denoting the allocation point stack trace[2] (4 bytes).
- The time of the first memory access to the chunk (8 bytes).
- The time of the last memory access to the chunk (before deallocation or program termination, 8 bytes).
- The *access interval*, defined as the address range of bytes that were accessed within the chunk. This is stored as two integers representing the upper and lower index access bounds (4 + 4 bytes). This measure is a trade-off from a more full representation of "actually accessed" memory, which would require a minimum of one bit per byte to indicate whether the program accessed the byte in question. Currently, we consider this an unacceptable overhead and so use the *access interval* as a proxy.
- The number of reads and number of writes to the chunk. The maximum of this value is configurable, but larger maximums can increase the metadata size overhead as more bits are required to store larger numbers (currently 1 + 1 bytes). Memoro will record up to the maximum representable number of the field (e.g. up to 256 for an 8-bit field), at which point it saturates and does not "roll back" to zero.
- Whether the chunk was accessed by multiple threads (1 byte). This requires accessing the ID of the current thread by the runtime, which acquires a global lock, affecting performance. The collection of this data is off by default.
- The thread ID of the thread that created this chunk (4 bytes).

[1] In recent multicore architectures, the timestamp counter (accessed via `rdtsc` on Intel processors) is synchronized across cores, making this approach time-distortion free. Although the execution of the timestamp instruction can be reordered by an out-of-order processor [18], introducing uncertainty, the events that we log are relatively infrequent and unlikely to be affected by this.

[2] The actual stack traces are stored in a separate hash table structure.

One other field is required by the allocator to accomplish its function:

- Whether the chunk is currently allocated or not (1 byte). Not strictly required for memory profiling, but required for the correct operation of the allocator.

The total byte size of one metadata record is current 52 bytes. This is already a large amount of metadata, however, the actual memory overhead will vary from program to program, depending on the actual allocations it makes. The memory overhead of allocating a 1-byte chunk will be extremely high, however, the overhead of a contiguous 1 GB array would be extremely small, since both allocations have the same amount of metadata.

3.3.2 Instrumentation

To collect this information, memory accesses in a program are instrumented. This means that some mechanism is used to run additional, specialized code at instrumentation points in the program to extract information about the program state. In this case, an instrumentation point is any load or store operation in the program, because we are trying to record information about heap accesses.

There are two ways to implement this instrumentation mechanism. *Dynamic instrumentation* reads the instruction stream of the program as it is executed, inserting instrumentation at runtime under defined conditions. *Static instrumentation* uses a modified compiler to insert instrumentation code at predefined points in the program binary itself. Both methods have advantages and disadvantages. Dynamic instrumentation (a good example being the tool Valgrind, discussed earlier in Section 3.2) is generally more costly in terms of runtime overhead, but it has the advantage that program source is not necessary and it need not be recompiled to be instrumented. Static instrumentation requires modifying a compiler to insert the instrumentation code and therefore requires access to the program source. However, its overhead is generally much smaller because there is no layer between the hardware/OS and the instruction stream. Both methods can easily access and record the required program state for a tool like Memoro.

Given these two options, we believe that users profiling program performance will most likely have the ability to compile the source. Even with Valgrind, debugging information is necessary to interpret the recorded data (such as stack traces). Low overhead also makes for faster debug/test cycles for users and allows the instrumented program to run in more contexts. Therefore, we chose to use static instrumentation to implement Memoro.

A modified compiler inserts instrumentation code at each memory access in the user's program. In effect, every memory access is transformed from:

```
*address = ...;  // or: ... = *address;
```

to:

```
if (IsHeapChunk(address)) {
  MemoroRecord(address, kAccessSize, kIsWrite);
}
*address = ...;  // or: ... = *address;
```

Essentially, the compiler inserts a runtime function call before a memory-accessing instruction. First, a predicate provided by the runtime system determines if an address points into heap memory. The details of this predicate evaluation are provided in Section 3.3.3, but briefly, it is possible because the runtime system implements the memory allocator and is aware of all of the heap address spaces. If the predicate evaluates true, the runtime records the access to a chunk in the chunk's metadata. The memory access executes as normal in either case, and the actual program semantics are unchanged.

In the compiler, the instrumentation pass runs after the optimization passes, to avoid instrumenting memory accesses that will be optimized away. This approach to instrumentation is similar to other tools in the sanitizer framework [112], and in fact, the Memoro instrumentation compiler pass and runtime is built on the sanitizer framework, which itself is integrated into the LLVM compiler system [76].

Because we are only interested in heap memory accesses, not *all* memory accesses need to be instrumented. There are several regions of memory in a program: global, stack, and heap. Ideally, we would only instrument the heap accesses [3]. However, in the LLVM IR, global, stack and heap memory are all accessed via the same instructions (loads and stores), so it is necessary to examine the pointer operand to determine which type of memory is being accessed. In many cases, it is easy to deduce that a reference points to a local stack object produced by `alloca`, the stack allocation instruction; in this case, we omit instrumentation because we are guaranteed that the memory is not in the heap.

Other cases are more complex. For example, an object may be allocated on the stack, and a pointer to the object passed to another function. It is difficult to tell at compile time what type of memory a pointer points to without inter-procedural analysis similar to escape analysis [36]. Without this analysis, accesses must be instrumented conservatively and pointers checked at runtime, or we risk missing potential heap accesses.

The current instrumentation pass logic follows a single link in the static single assignment (SSA) of the LLVM IR to determine if an access is to stack-allocated memory. By examining whether the pointer operand came directly from a stack allocation (`alloca`) instruction, we can identify references to stack-allocated variables and avoid instrumenting a large number of loads and stores, reducing the instrumentation and runtime overhead. More complex

[3] This is not to say that stack objects are not interesting, but they are beyond the scope of this thesis. Future work could integrate stack memory analysis into Memoro.

computations that produce pointers to the stack are currently not fully analyzed and their memory access is instrumented. In the future, a more thorough propagation analysis might eliminate more unnecessary instrumentation.

Runtime checking of pointers is one of the main causes of Memoro overhead, as we will see in more detail in our evaluation in Section 3.5.

Type Extraction

We have found that it is useful to know the data type of a chunk when visualizing and analyzing collected data. For example, objects of a class such as a tree node may be allocated at several different points in a program, and it is often convenient to aggregate these allocations and treat them as if they occurred at one call site. This was shown by our motivating example bioinformatics program — it was allocating the same object type in many different places, usually in call stacks deep in Standard Template Library (STL) containers. It was extremely difficult to see the aggregate effect of this particular object type on heap usage using standard profilers. To this end, the Memoro instrumentation pass in the compiler associates type information with allocation points.

Many allocations correspond semantically to allocating typed objects or arrays of typed objects, even though the allocate routines return `void*` (e.g. `new` or `malloc`). In the LLVM IR, an allocation function call is *usually* followed by a cast instruction. The type for a particular allocation point can be inferred from this cast. In the absence of a cast, a byte type *e.g.*,`char*` is assumed. In the rare case of multiple cast instructions, the first type that is not a byte type is preferred.

The compiler pass writes (code location, type) pairs to a file, which the visualizer uses to associate types with allocation points in its display of Memoro data. Occasionally, in templated code or at allocation code in header files, we see multiple different types mapped to one allocation point. This occurs because the compiler scope is limited to a single compilation unit *i.e.*,a source file, and an allocation point in a header can, therefore, appear in multiple places. Rather than make complex modifications to the compiler, the instrumentation pass emits all types encountered for an allocation point. The visualizer, having access to complete stack traces from the runtime system, will display the first type that is compatible with the stack trace. Although this technique is a heuristic, we find it is correct in the vast majority of cases.

3.3.3 Runtime System

The runtime system for Memoro provides two types of functionality: 1) a memory allocator to keep track of which address ranges are heap-allocated, and 2) mechanisms to log and store metadata corresponding to individual heap chunks. Metadata for a chunk includes the chunk size, where it was allocated (stack trace), timestamps recording the times of the first

and last reads/writes to the chunk, the byte access interval, the number of reads and writes, and whether the chunk was accessed by multiple threads (Section 3.3.1). This data must be updated every time a chunk is accessed.

As described above in Section 3.3.2, memory references in the program are conservatively instrumented, and not all instrumented references will be to heap memory. The runtime must first determine whether the pointer in question is pointing to memory owned by the heap, and *then* update the associated chunk metadata. To keep runtime overhead low, both the pointer ownership check and metadata lookup and update must be done *quickly*, as memory accesses occur frequently in programs.

The amount of metadata per allocated heap chunk, currently 52 bytes, is large because timestamps need to be 64 bits. 64 bits are required since time is measured in CPU cycles; even unsigned 32-bit counters would overflow before most programs finish executing. Chunk metadata can be stored in memory, which is fast, but sometimes the region for the stored metadata of freed chunks needs to be expanded, which involves copying and is expensive. We believe a better idea would be to log data to disk at this point, but care must be taken to ensure the runtime does not block.

Our runtime system borrows from the AddressSanitizer framework, whose runtime implements both the allocator and the allocation routine interception. The allocator consists of two components: a thread-local *primary* allocator, and a shared *secondary* allocator. The primary can allocate chunks efficiently *without* locking, but is limited in the alignments/sizes of chunks it can allocate. The secondary is a fallback that can allocate any size/alignment, but must be locked to ensure safety when accessing shared data structures used to allocate/deallocate chunks, and when determining pointer ownership.

The primary allocator maps up-front a large portion of address space from the OS via `mmap`. Regions, divided into freelists of chunks of varying size, utilize this space. Metadata for each chunk resides at the upper end of the address space of a region. The metadata for a chunk is thus accessed at a fixed offset, which is fast and does not require conditional branches. Because the primary allocator is aware of its address space, pointer ownership can be determined in constant time by a simple boundary comparison. Figure 3.3 shows a diagram of the primary allocator memory layout.

The secondary allocator is meant to service large and rare allocations, and will `mmap` chunks and metadata directly, with the metadata being placed in extra space at a page boundary. This requires locking for thread safety. The chunk metadata, however, can still be accessed in constant time because it is at a known offset. However, because the set of chunks is now disjoint, the secondary allocator must traverse the array of chunks when determining whether it owns a pointer. The allocator must also be locked to do so safely. Therefore, programs that allocate large chunks accessed by multiple threads will experience higher than average

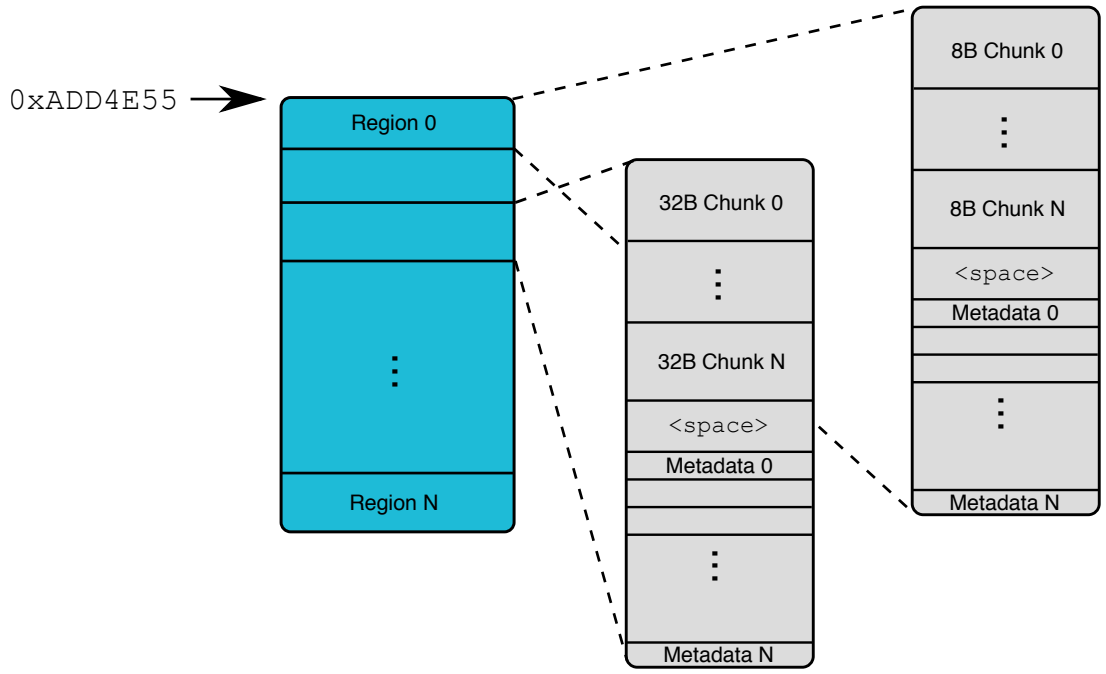


Figure 3.3 – Primary thread-local allocator memory layout. A contiguous block is divided into regions of defined chunk sizes, with metadata occupying the end of a region's address space. This allows both chunks and metadata to be looked up in constant time via offset arithmetic.

overhead[4] due to linear traversal, and lock contention in the secondary allocator. Figure 3.4 shows a diagram of the secondary allocator memory layout.

To reduce this overhead, Memoro provides the option to use a modified allocator that avoids locking while determining pointer ownership or looking up metadata. This relies on the assumption that the user has correctly synchronized accesses to shared memory, and will not deallocate shared chunks while another thread is accessing them. If this happens, the runtime may crash as it tries to access metadata for a chunk that has just been deallocated. Indeed if the program does contain such race conditions, it might crash anyway. Memoro is not designed to detect such cases, however, another sanitizer tool, the *Thread Sanitizer*, is designed for just this problem.

There is a trade-off to removing locking in the secondary allocator, however. During a pointer ownership test, unrelated chunks that are allocated/deallocated will modify the array of chunks that the allocator manages, causing the pointer ownership test traversal to *possibly* miss an owned pointer, resulting in a false negative and a missed access to a heap chunk. This can happen because of how the secondary allocator manages its array of chunks. Essentially, when a chunk is deallocated, the "hole" left in the array of chunks is plugged with the last element of the array, and the size count is decremented. If this deallocation operation happens

[4] This is simply an artifact of the current implementation; in Section 3.6 we discuss other allocator schemes and tradeoffs that may alleviate this overhead.

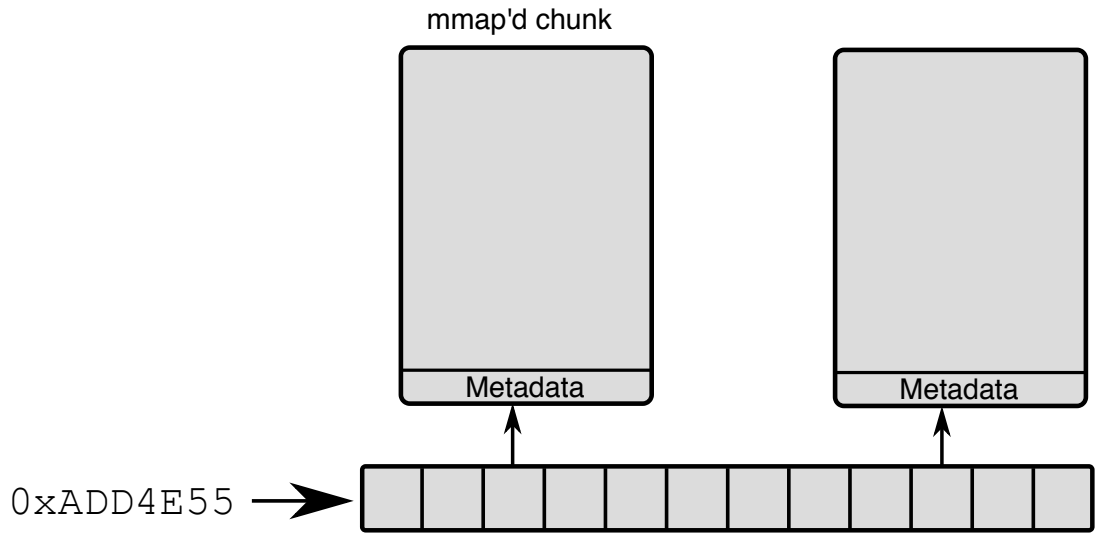


Figure 3.4 – Global secondary allocator that handles allocators larger than the primary's max chunk size. Chunks are mmap'd directly and stored in an array, leading to $O(n)$ lookup time.

while a pointer check is in progress and the check (a linear traversal) has *passed the location of the deallocated chunk*, it will likely miss the last chunk that was used to "plug the hole" in the array. If the missed last chunk did indeed contain the pointer being checked, the memory access to that chunk will not be counted.

Program semantics and safety, however, are preserved, provided the user's program has no race conditions as described above. The only side effect is the slight chance that a few read or write operations will not be counted by the runtime. We examine and quantify the overhead and benefit of the allocator modifications in more detail in Section 3.5.

While active chunk metadata is stored and managed by the allocator, freed chunk metadata is copied and stored in a separate memory-mapped array associated with the corresponding allocation point. These are stored alongside the allocation point stack traces themselves in a hash table structure. At program exit, a routine compresses and records the chunk metadata array of each allocation point into a compact binary file. The metadata itself, as well as the runtime storage of active and freed chunk metadata, will lead to slightly higher memory use compared to uninstrumented programs.

The runtime system also intercepts a variety of standard library calls that access memory (e.g. `memset`, `memcpy`), as opposed to providing an instrumented standard library. Many of these interceptions aggregate multiple accesses (individual loads and stores) into one large access, which helps reduce overhead.

The compiler modification source and binaries are available [24].

3.4 Memoro Visualizer

While the instrumentation and runtime system collect a large amount of raw data, the Memoro visualizer provides insight into program behavior using that data. As the name suggests, the visualizer makes heavy use of visual cues, produced by aggregating and analyzing data, to direct a developer to areas of a program that display inefficient usage or performance-detrimental allocation patterns. As many inefficient usage and allocation patterns can be difficult to see in raw data, Memoro employs a new technique to distill and quantify these patterns into *scores*. Scores form the primary analysis that helps prioritize the visual cues used in the visualizer. These scores run on a scale between 0 and 1, where 0 is undesirable or inefficient behavior and 1 is the most desirable, efficient behavior. Alongside data visuals, scores provide a developer with a specific indication of the problem with heap usage and the ability to quickly pinpoint the locations that are most likely responsible for poor behavior/performance. The following subsections describe the scoring algorithms, the visual displays that use the scores, and how they inform a developer.

3.4.1 Data Analysis and Scoring Algorithm

Memoro takes into account several aspects of heap usage when measuring how efficiently a program has used the heap, generating scores quantifying *lifetime, usage* and *useful lifetime*.

Lifetime

Lifetime refers to how long heap chunks are active before being freed. Memoro pays particular attention to short lifetime chunks, especially those that are grouped tightly in time. These patterns indicate regions of code in which chunks are constantly being allocated and deallocated, typically with very few reads and writes to the chunks. An example is when a developer (perhaps mistakenly) explicitly creates an object inside an inner loop. Often, these objects are not explicitly `malloced`, but rather are a stack-allocated container object whose constructor allocates heap memory. In a tight loop, repeatedly allocating memory, particularly if not heavily used, can have significant performance impacts.

Memoro builds a *lifetime score* by looking for short lifetime chunks that were allocated close together in time from the same allocation point, where “short” is a modifiable parameter. To build these *groups* of short lifetime chunks, the array of all chunks for an allocation point is sorted by time and traversed. Any chunks allocated within some threshold time of each other are added to the same group. The threshold interval is a parameter, with a default of 0.1% of total run time. Then, a score S_g for each group G in allocation point A is computed as follows:

$$S_g = \frac{\frac{\sum_{C \in G} C_l}{|C_g|}}{G_l}$$

where C_l is lifetime of chunk l, $|C_g|$ is the number of chunks in group g, and G_l is the group lifetime. The group lifetime is defined as the time difference between the earliest allocation and the latest deallocation.

The lifetime score S_l for allocation point A is then computed as:

$$S_l = \frac{\sum_{G \in A} S_g}{|G|}$$

where $|G|$ is the total number of groups.

In essence, the average chunk lifetime of each group is normalized by the group lifetime G_l and the allocation point lifetime score S_l is the average of all the group scores, resulting in a normalized value between 0 and 1, where 1 is best and 0 is worst. The group normalization avoids the situation in which a few abnormal short-lived allocations exert a strong influence on the final score.

Usage

The *usage score* for an allocation point provides a measure of how well a program makes use of the bytes of memory that it allocates. In an ideal situation, every allocated chunk will be fully written and read by a program, preferably many times (good reuse). A poor usage score can indicate that areas in a heap chunk are unused or write-only, or that only a fraction of the bytes of a heap chunk are read or written. For example, a larger than necessary buffer may contain several unused elements. Eliminating these can reduce the total heap usage and improve allocator behavior.

The usage score S_u is computed for an allocation point A as follows:

$$S_u = \frac{\sum_{C \in A} U_c \cdot C_{BytesAccessed}}{\sum_{C \in A} C_{TotalBytes}}$$

$$U_c = \begin{cases} 0 & C_{rd} = 0 \,||\, C_{wr} = 0 \\ 1 & otherwise \end{cases}$$

U_c is the *Useless* function, which returns 0 if a chunk has no reads or no writes. The usage score is thus a normalized average of the number of bytes accessed by a program relative to the total bytes allocated. To keep overheads reasonable, the profiler does not collect the full byte-level access statistics, but rather an *access interval* (lowest and highest accessed location), which is updated whenever a chunk is accessed.

Useful Lifetime

Even when chunks on the heap live for a long period, a program may not make good use of them during their entire life. Chunks may be allocated and not written or read for a long period, or vice-versa, chunks may be allocated and used but then sit idle for a long time. Heap usage over time could be reduced by allocating these chunks when they are needed and freeing them when they are no longer needed. The *useful lifetime* score quantifies this concept for an allocation point:

$$S_{ul} = \frac{\frac{\sum_{C \in A} C_{activelife}}{C_l}}{|C|}$$

where *active life* is the time difference between the first and last accesses to the chunk (recorded via instrumentation) and C_l is the total chunk lifetime. The sum is then normalized by the number of chunks to produce another score between 0 and 1.

Finally, global scores are calculated by determining the geometric mean of each score across all allocation points. Variances of global scores are also calculated to give the programmer an estimate of the score distribution across the program.

Discussion

These scores are intended to provide developers with clear, digestible measures of how efficiently their program uses the heap. They do not always convey an unequivocal truth and can be context-dependent, occasionally requiring a manual examination of a program. For example, different inputs or workloads for a given program may affect which bytes are accessed in a chunk or how long a chunk lives. In general, it is not possible to modify a program to achieve a score of 1.0. However, we have found in practice that the scores provide very helpful information to pinpoint heap usage issues. The case study in Section 3.5.2 provides some examples.

Other Inferences

Memoro will make several other metrics based on the collected data and display them for each allocation point, to help a developer prioritize issues. These include:

- Top percentile of bytes allocated: Memoro marks an allocation point if it is in the top 90th percentile of maximum heap usage.
- Top percentile of chunks allocated: Memoro marks an allocation point if it is in the top 90th percentile of total chunks allocated.
- Read- or write-only chunks *i.e.*,useless chunks.

- Runs of monotonically increasing allocation sizes, indicative that early allocations were too small.

3.4.2 Visualizer Application

Modern applications, even small ones, can perform millions of allocations at thousands of different locations. Presentation and meaningful analysis of this mountain of data is crucial to properly interpreting and quickly diagnosing problems. The visualizer application of Memoro is a separate tool that provides a visualization of the data that the instrumentation and runtime system collects. A *global view* gives a developer a bird's eye view of the entire dataset, with the option to "zoom in" to specific data points in the *detailed view*. In both views, the user can filter and aggregate by stack trace function name, time interval, or object type. Memoro scores are used throughout to give intuitive visual cues to the user, in order to guide them to areas of interest.

Global View

The global view aims to provide a developer with an overview and summary of a program's behavior as a whole. This is achieved via two visuals, a flame graph [61] (Figure 3.5) and a line graph showing the total (aggregate) heap usage across the program lifetime (Figure 3.10). A flame graph is a visualization that shows stack depth in the y axis, and stack frames in the x axis, sized proportionally according to some value, typically the number of samples in a CPU profile. The proportional size of a frame is equal to the aggregate size of all of its children plus the frame's value. For example, Figure 3.5 shows a flame graph in which the frame sizes are proportional to the number of allocations over the program lifetime. Function `main` in `contrived.cpp` contains three allocation points, one of which is contained deeper inside a `std::vector::push_back()` call. Flame graphs are useful because they give an overall summary of how the memory usage occurs at different points in a program, over its entire execution.

The Memoro flame graph can display several categories of data values, including the total number of allocations per allocation point (as seen in Figure 3.5), and total bytes per allocation point at a given point in time (selectable by the user). This is similar to memory flame graphs described in [61]. The flame graph "tips" correspond to unique allocation points, and are colored with a *severity indicator*, a visual cue that maps Memoro scores for that allocation point to a set of colors. The mapping is from the geometric average of all scores to a set of 12 colors, roughly, from blue to green to yellow to red. Global scores for lifetime, usage, and useful lifetime are also displayed along with total allocations, maximum heap usage, total allocation points, and the approximate time the program spent in allocation routines.

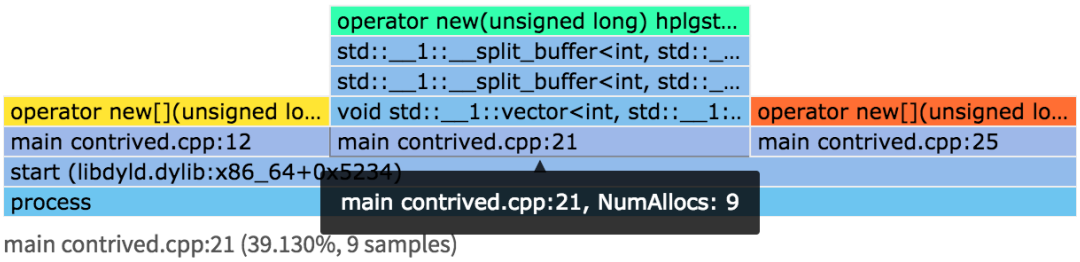


Figure 3.5 – Visualizer flame graph representation, showing a global view of all allocation points in the program. It can be organized by number of allocations (shown here), or by byte usage at any point in time. The severity indicator that color codes an allocation point score is mapped to the flame graph allocation points. Long stack traces can be viewed fully in the bottom left, or via the tooltip.

3.4.3 Detailed View

While the global view can identify problematic allocation points, a developer may need to examine these points in more detail. The detailed view allows a developer to drill down and examine individual allocation points, view their allocation patterns over time, and easily identify the cause of low scores. Figure 3.6 shows how individual allocation points are displayed, along with their aggregate heap usage – the line graph overlaid on the global aggregate graph. This makes it easy to see how individual points contribute to the total heap usage. The severity indicator in the top right corner of each allocation point again provides a visual cue that flags points with poor overall scores. Full stack traces are also displayed, with the ability to open source files and jump to the corresponding line of code. Statistics and inferences for an allocation point are displayed separately, including their scores. Allocation points can be sorted by heap usage, average score, individual score values, and the number of allocations.

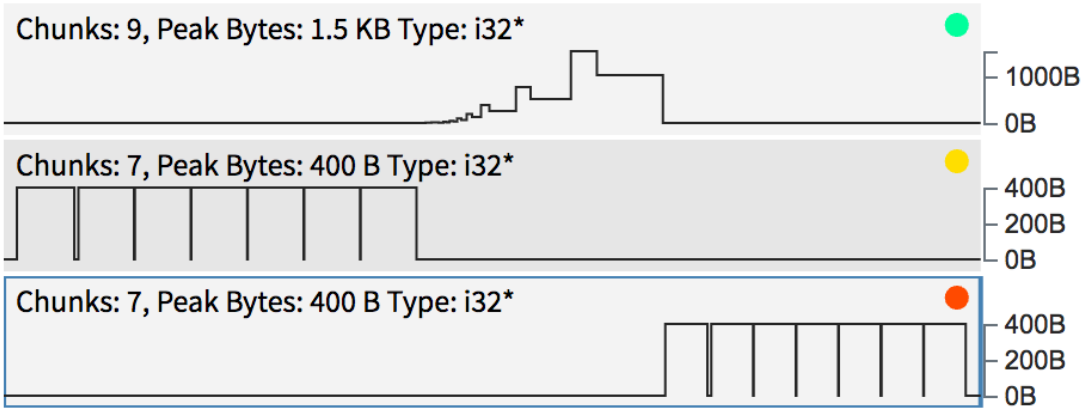


Figure 3.6 – Visualizer allocation point representation, showing number of chunks allocated, peak heap usage, object type, severity indicator, and aggregate line graph.

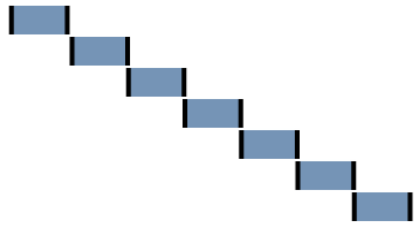


Figure 3.7 – Heap chunk allocated inside a loop, written/read, and freed. Time progresses from left to right; the length of a chunk represents its lifetime. These chunks correspond to those above in Figure 3.6 (middle, yellow indicator).

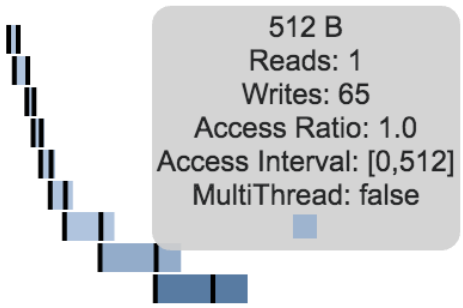


Figure 3.8 – `std::vector` reallocation pattern, with darker chunks indicate larger byte sizes, and the chunk tooltip.

Allocation patterns over time can also provide a visual indication of poor allocation strategies or inefficient usage that are not always obvious in an aggregated line graph or simple statistics like the number of chunks allocated. Because programs can make many allocations, fidelity can be lost when aggregating data. To this end, the visualizer also displays individual chunks of an allocation point as blocks, whose length corresponds to lifetime and whose color shade corresponds to their size. Black vertical lines indicate the access interval as recorded by the runtime system. Figures 3.6-3.10 show some examples of this display, including common inefficient patterns in an actual program, all of which are actual screenshots from the Memoro visualizer.

In Figure 3.7, chunks are allocated inside a loop, where they are written, read, and then freed – and then allocated again in the next iteration. Figure 3.8 shows the reallocation pattern for a `std::vector`. As integers are pushed into the vector, it allocates a larger array, copies existing data, and frees the old, smaller array. This happens several times as the vector grows exponentially. The visualizer marks larger byte size chunks with darker colors. The visualizer tooltip also shows a developer additional detailed information about individual chunks, including the number of reads, writes, size, access interval, and whether or not the chunk was accessed by multiple threads. Figure 3.9 shows chunks allocated and freed in a loop with very short access intervals, indicating low useful lifetimes. Finally, Figure 3.10 shows the combined aggregate heap usage across time.

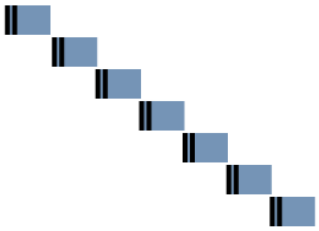


Figure 3.9 – Low useful life chunks, as indicated by extremely short access interval, indicated by the black bars.

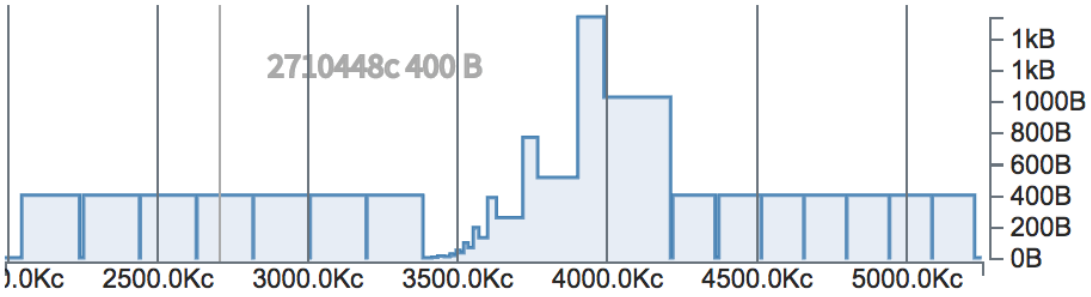


Figure 3.10 – The aggregate total heap usage graph. A cursor shows the exact cycle time and total heap usage at that time.

As can be seen in Figure 3.6, the scores for each allocation point are again translated to a clear visual cue for the user — the color-coded severity indicator in the upper right corner of each allocation point displayed. We can see that the third has the lowest score because it allocates short lifetime chunks that also have a short useful life, while accessing only four bytes out of 400, giving a low usage score as well. In contrast, the first point is much better, allocating longer lived chunks that have a long useful life and in which every byte is accessed. Note that this point could be reduced to a single allocation had we called `vector::reserve()` before pushing values. Figure 3.11 shows a full screenshot of the detailed view of the Memoro visualizer application.

The visualizer also contains a comprehensive filtering system that allows a user to filter data by keyword in stack traces, by allocation data type, or by time intervals. Aggregates, graphs, and visuals, in both global and detailed views, are updated according to the filters applied.

The visualizer GUI (Graphical User Interface) is implemented using Electron [5], a framework for building cross-platform applications using Javascript and CSS. The result is a responsive and visually pleasing product, similar to a modern web app. However, with larger programs, profiling can generate a significant amount of data — millions of data points, across many thousands of allocation points. Javascript is not particularly well suited to the efficient processing of large amounts of data, so a C++ data processing backend is used to perform all data manipulation, including file loading, aggregation, score generation, and filtering. A minimal

amount of data required for display is returned to the JS frontend, keeping the visualizer responsive and fluid.

The visualizer application, as with the rest of Memoro, is open source and available [24].

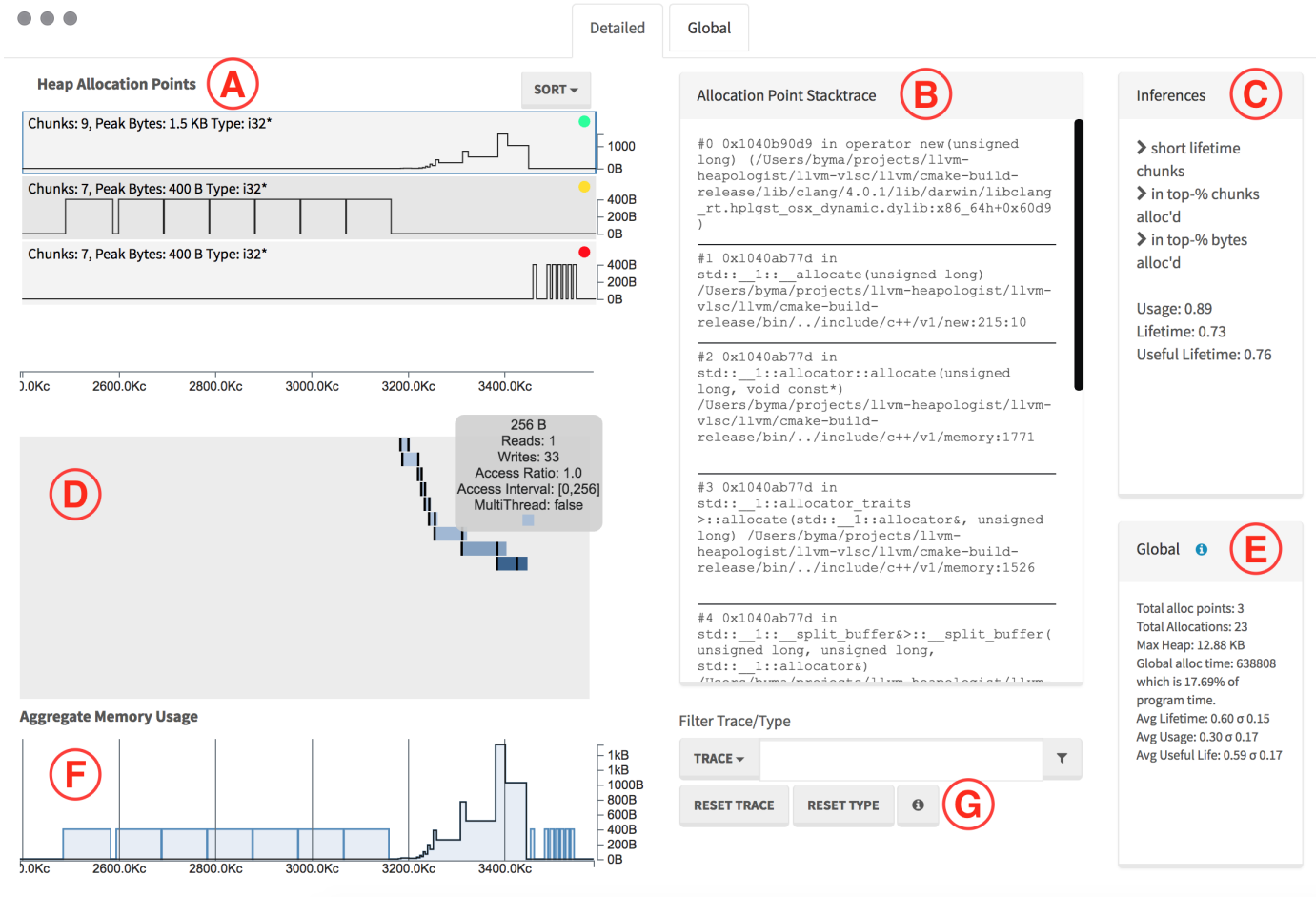


Figure 3.11 – The Memoro visualizer application, showing the detailed view. **A**: List of unique heap allocation points. **B**: Stack trace of the selected allocation point. **C**: Inferences; informative data found by the tool, and allocation point scores. **D**: Chunks of an allocation point displayed in time; the tooltip showing detailed data. **E**: Global data, geometric average scores and variances. **F**: Aggregate line graph of total heap usage over time. Dragging a selection applies a time interval filter. **G**: Filtering by trace and object type.

3.5 Evaluation and Case Study

In this section, we provide an evaluation of the runtime and instrumentation overhead of Memoro, as well as a case study showing how Memoro can lead to meaningful improvements in different programs, and in particular some bioinformatics programs. Such improvements help programs make more efficient use of available commodity cluster hardware, furthering the goal of parallel and scalable bioinformatics systems.

3.5.1 Instrumentation and Runtime Overhead

We first evaluate the effectiveness of static analysis at reducing the number of memory references that must be instrumented (Section 3.3.2). Recall that our approach follows a single link in the LLVM SSA graph to see if a load or store is accessing a stack-allocated value. If this is the case, the memory accesses can safely be left uninstrumented, as we are guaranteed they are accesses to local stack variables. On average, in all programs in our study, we found that ~66% of memory access passed to the runtime were heap accesses. This means that ~34% of accesses *could* have been left uninstrumented, thereby reducing overhead. However, as discussed in Section 3.3.2, this improvement requires more advanced, inter-procedural program analysis.

Next, we evaluate the system runtime overhead, that is, how much more time it takes to run a program instrumented with Memoro than the unmodified program. Runtime system overhead is heavily dependent on how a program uses its heap. To break down this overhead and understand which factors influence it, we use Memoro to profile two large, popular open-source programs — LevelDB [59] and Memcached [20]. LevelDB is exercised using its internal benchmark utility (`db_bench`) running the *fillseq* benchmark, which sequentially writes a series of values to the database. The read or write access ratio does not matter to Memoro, and does not influence overhead. Any access to heap memory will have the same amount of overhead generated, regardless of whether it was a read or write. What influences Memoro overhead is the *instruction mix* of the program, in particular, the ratio of read/write instructions to everything else. More reads and writes leads to more instrumentation points, which leads to more runtime evaluations and higher overhead.

For Memcahed, a load generator and measurement tool called Mutilate [77] is used to create Memcached `get` requests over 5 seconds, while Memcached itself is run with a single thread. Mutilate is run with default values. LevelDB performance is measured in MB/s while Memcached throughput is measured in Queries per Second. Experiments are run locally on a server with two Xeon E5–2680v3 CPUs at 2.5 GHz with 256GB of RAM running Ubuntu 16.04. Results are show in Table 3.1.

First, we examine the effect of the aggregate static instrumentation on performance, by replacing runtime instrumentation library with empty functions, and hence recording no data. Memcached has 1.1× lower throughput, while LevelDB has 2.1× lower throughput. We

	Memcached (Q/s)	LevelDB (MB/s)
Unmodified	52184.4	55.5
No-op Inst.	47668.3 (1.1×)	26.8 (2.1×)
Full Collection	18051.4 (2.89×)	1.8 (30.8×)
No-lock Sec.	22521.8 (2.32×)	4.1 (13.5×)

Table 3.1 – Throughput and slowdown of Memcached and LevelDB when unmodified, with no-op compiler instrumentation only, with full runtime data collection, and with the modified non-locking secondary allocator to reduce overhead.

find that this is primarily a function of the number of instrumentation points — LevelDB has significantly more loads and stores to be instrumented; at the LLVM IR level, LevelDB contains 39% loads and stores as opposed to Memcached's 19% loads and stores. As mentioned above, the program instruction mix is responsible for the difference in overhead between the two programs.

Next, we fully enable Memoro instrumentation and data collection. There is a large difference in overhead cost between the two programs, ranging from 2.89× for Memcached to 30.8× for LevelDB. To understand why, we use a standard CPU profiler (Intel VTune 2017) to see where time is spent. Profiling indicates nearly half of the time in LevelDB is spent in the secondary allocator acquiring locks[5] to safely record Memoro metadata. As it turns out, LevelDB uses an arena allocator that obtains large memory blocks via `malloc`. These allocations are served by the secondary allocator in the Memoro runtime, as the primary thread-caching allocator is unable to handle such large allocations. The remainder of the overhead (and the majority of overhead in the Memcached experiment) comes from looking up the metadata for a chunk and updating statistic counters.

This result prompted us to modify the secondary allocator to eliminate locking when determining pointer ownership, as discussed in Section 3.3.3. The final row of Table 3.1 shows that with the modified secondary allocator with lock-free ownership checking, overhead for Memcached is slightly reduced, while overhead for LevelDB is reduced by over 50% to 13.5×. Overhead for LevelDB is still higher primarily because the ownership check in the secondary allocator is linear in complexity, exacerbated by the fact that LevelDB had a high number of instrumentation points. Furthermore, of all our tests, LevelDB had the lowest proportion of heap accesses relative to all memory accesses, meaning that many accesses to non-heap data were being checked by the runtime. This results in a large number of full traversals of the secondary allocator chunk list for LevelDB, to confirm that the accesses are not in the heap.

Recall that the secondary allocator in the runtime is global, and must lock to determine pointer ownership safely. Without the lock, a thread might deallocate a chunk while another thread is attempting to determine pointer ownership — this may result in illegal memory accesses and crash the runtime. Multithreaded programs with large allocations will thus experience

[5]The benchmark by default uses two threads

higher than average overhead, due to this implicit sharing (and mutex locking) introduced by the Memoro runtime. This explains the higher overhead seen in the LevelDB experiment. Possible solutions to ameliorate this are discussed in Section 3.6.

In terms of memory use overhead, the more allocations that a program makes, the higher the overhead cost because Memoro records metadata for every allocated heap chunk. Metadata is also the same size for all chunks, meaning that small allocated chunks will have higher memory use overhead relative to large chunks. In longer running profiles, memory overhead may start to impact performance, however, we have not experienced this in any program we have profiled so far. This is mostly due to Memoro spending time to map memory from the OS to store metadata. For typical programs, however, overheads are around 3–5×, as shown in our case study below. Regardless, we are investigating static buffering techniques to eliminate this, also discussed in Section 3.6.

3.5.2 Case Study

In this section, we illustrate several uses of Memoro that lead to improvements in heap efficiency and program runtime, as well as examples that demonstrate the utility of the data analysis that Memoro performs. All tests were performed on a 2.5GHz Intel Core i7 processor with 16GB of RAM running MacOS v10.12.16 (Sierra). The Memoro instrumentation used was built into LLVM/Clang [76, 84] release version 4.0.

Protocol Buffers

Protocol buffers (protobuf) [10] is a popular framework from Google for data serialization. In protobuf, *messages* are defined using a declarative language, which the protobuf compiler compiles to classes in various languages that can serialize themselves. The protocol buffer implementation makes heavy use of the heap, especially when messages contain `repeated` fields, or arrays of data. These can be arrays of basic supported data types or arrays of sub-messages. In either case, deserializing or constructing protocol buffer messages may perform many allocations, which can incur significant performance overhead, especially in latency-sensitive systems that must process many messages per second.

Recently, the protocol buffer implementation added *arena allocation* to alleviate this issue. An arena pre-allocates a large block of memory and uses it for internal message data, to avoid repetitive OS heap allocations when constructing or deserializing messages. The point of our study is to show that Memoro will correctly identify the problem allocations that prompted this change, and present correct and meaningful analyzed data.

We construct a benchmark using the following protobuf message, which consists of a string field, a repeated field of integers (an array), and a repeated field of a sub-message that itself contains a repeated field of integers:

	Without Arena		With Arena	
Total Allocations	268117		112391	
Lifetime Score	0.40		0.60	
Usage Score	0.94		0.74	
Useful Life Score	0.72		0.37	
Runtime (ms)	72		65 (−9.7%)	
Instr. Runtime (ms)	1520 (21.1×)		1623 (24.9×)	
Memory (MB)	Uninstr.	Instr.	Uninstr.	Instr.
	2.76	26.6	3.2	17.6
		9.6×		5.5×

Table 3.2 – Protobuf benchmark with and without arena allocation. Note that the overheads are abnormally high because the benchmark does very little work relative to the number of allocations it makes.

```
message SubRecord {
    repeated int64 ids = 1;
};

message Record {
    repeated int64 numbers = 1;
    repeated string strings = 2;
    repeated SubRecord subs = 3;
};
```

The benchmark serializes an instance of a `Record` with 1000 integers, and five sub-messages each with 1000 integers. Then, the message is deserialized 1000 times in a loop, creating a new instance each time, and then destroying it. We run the benchmark with and without the arena allocation, after having compiled it with the Memoro instrumentation and runtime system. The total number of allocations and global scores generated by Memoro for each are summarized in Table 3.2.

Without arena allocation, Memoro reports a global average lifetime score of only 0.40, indicating that many allocation points in the program produce chunks that do not live long and that we may have allocations in a tight loop, as expected. Average usage is high however since every chunk allocated has been almost fully read and written by the program. The useful life score is also relatively high at 0.72 since chunks are read and written during most of their lifetime.

With arena allocation, however, we see a near reversal of values. The lifetime score has increased to 0.60, showing that the new allocation scheme has reduced the number of short lifetime allocations. While we might expect a higher value, the current implementation does not allocate strings in the arena, and the arena itself allocates several short-lived items. The loop is also simply deserializing data and not doing any real work. The total number of allocations has been cut nearly in half. The usage score has decreased, which may seem

counter-intuitive, but it is what we expect — because the arena allocates large blocks upfront, some parts of them inevitably go unused. Likewise, the useful lifetime score has decreased to 0.37, because the arena blocks live for longer than the period during which they are read and written by the program.

In effect, Memoro shows clearly the trade-off that protobuf arena allocation makes: slightly less efficient use of heap chunks in return for fewer, higher lifetime allocations, ultimately resulting in lower runtimes. In the case of this benchmark, a 9.7% improvement in execution time.

Bioinformatics

Our second case study takes us to the bioinformatics field, the central application field of this thesis. The particular toolset we analyze is the bamtools API [34], a library and associated toolkit written in C++ for manipulating BAM files [80], one of the most common file formats in bioinformatics. BAM is the binary counterpart of SAM (Sequence Alignment Map), a format used to store alignments of sequenced genomic data. Common operations include data conversion, sorting, and filtering. In this case study, we analyze two tools that use the bamtools API: sorting and filtering.

We first compile the bamtools source code using Memoro instrumentation and run the sort tool using a BAM file containing approximately 4000 aligned reads. Each read is 101 bases long, similar to the data used to evaluate Persona in Chapter 2. After visualizing the results using the Memoro visualizer, we make several observations:

- The number of allocation points is very high — there are over 5000 unique points in the code that allocate memory on the heap.
- The total number of allocations is very high — over 64000 heap allocations over the program lifetime.
- The majority of points with many allocations have extremely low lifetime scores, zero or close to zero.

A glance at the flame graph shows that the vast majority of these allocations are within the `std::sort` routine used for sorting sub-arrays of reads before merging them. Comparing the stack traces of the high-allocation points, we see that many of them involve either creating alignment data structures or copying the alignment data structures. When we filter the data based on the primary alignment data structure copy constructor, we find that it is called in over half of all allocation contexts. A quick look at the data structure reveals the issue: copying the structure is expensive because it contains several `std::string` and `std::vector` fields that also allocate memory and copy data. Because this structure is used as the template value

4K Read BAM file		
	Unmodified	Modified
Total Allocations	64022	37333
Total AllocPoints	5705	279
Lifetime	0.90	0.85
Usage Score	0.92	0.82
Useful Lifetime Score	0.60	0.55
1M Read BAM file		
Runtime (s)	31.83	28.49 (−10.5%)

Table 3.3 – Persona profiling results when sorting a 4000 read BAM file. Note how the large number of allocation points with few allocations has skewed scores of the unmodified version.

	Uninstrumented	Instrumented
Runtime (ms)	134	404 (3.01×)
Memory (KB)	7888.9	45674.5 (5.78×)

Table 3.4 – Instrumentation and runtime overhead sorting a 4000 read BAM file.

for the sort, a great number of allocations and copies are generated as the algorithm swaps values.

To reduce these costs, we identify the fields in the alignment structure that are read-only and wrap them in shared pointers. This way, the cost of copies is much reduced as only a shared reference count is incremented. The result is that the total number of allocations is reduced to ~37000, and the number of unique allocation points reduced dramatically to 279. All results of the sort analysis are display in Tables 3.3 through 3.4. While the number of allocations is still high, they have been removed from the computationally intense portion of the program (along with associated copying), resulting in an execution time improvement of 10.5% when tested with a BAM file containing one million reads. A more extensive redesign of the data structure to avoid using `string` altogether could produce further allocation reductions, but would require pervasive changes to the code base. For example, it may be possible to replace these instances of `string` with `string_view`, which does not store data but merely holds a pointer to the underlying data. Since the input file and data in memory should be read-only, the only reason to make any copies should be when writing entries to the sorted output. The runtime overhead imposed by the instrumentation was relatively low, approximately 3.0×.

We also analyze the filtering tool included with the bamtools API, profiling with the 4000 read BAM file and single filter predicate that removes any read with mapping quality below a value of 50. The visualizer quickly gives us a rundown of potential problems. There are again a large number of allocations, primarily from constructing all of the string data that comprises the alignment structure. The visualizer also flags two allocation points in particular with very low scores (and red indicators) — particularly low lifetimes and low usage, and allocations

4K Read BAM file		
	Unmodified	Modified
Total Allocations	35708	27792
Total AllocPoints	363	349
Lifetime	0.90	0.90
Usage Score	0.82	0.83
Useful Lifetime Score	0.69	0.70
1M Read BAM file		
Runtime (s)	15.85	14.34 (−9.5%)

Table 3.5 – Persona profiling results when filtering a 4000 read BAM file, pre and post modification.

	Uninstrumented	Instrumented
Runtime (ms)	71	388 (5.46×)
Memory (KB)	3739.6	16297.9 (4.36×)

Table 3.6 – Instrumentation and runtime overhead filtering a 4000 read BAM file.

proportional to the input size. The stack traces show these are caused by two data structures: `std::queue` and `std::stack`. The queue allocations are caused by copying another queue, simply to iterate over it without destroying the original since the queue interface provides no iterators. The stack is used to parse and evaluate boolean filter expressions, but since it is allocated on the program stack, it is constantly created and destroyed (thus allocating and freeing internal memory). The underlying `std::deque` container also allocates blocks of 4KB, only a few bytes of which are used, causing the low usage score.

We make two minor fixes to reduce the number of allocations and increase overall scores. First, we make the stack object static in its function (safe since the program is not multithreaded), which preserves its memory across invocations. Second, we change the queue object to a double-ended queue (`std::deque`) to gain iterators and avoid copying. The net result is that total allocations are reduced to ~38000 and the global average usage and useful life scores increase by 1%. Tables 3.5 through 3.6 summarize the results. Total execution time when filtering a BAM file with one million reads using a single predicate is reduced by 9.5%. Thus we can see that by improving Memoro scores, we can also improve overall performance. The runtime overhead imposed by the instrumentation is 5.5×, slightly higher than the previous example.

3.5.3 Discussion

Using Memoro has helped us to design the visualization and data presentation in a way that helps pinpoint issues very quickly. In all of our case studies, Memoro made heap usage issues

obvious. The majority of our time was spent gaining familiarity with the codebases to identify and understand the changes suggested by the presentation of the data.

We have not yet found an occasion when Memoro provided misleading information or false positives in terms of lifetime, usage or useful life, or any other visualization. When a low (poor) score is appears, it is usually very easy to understand by examining the allocation point in the detailed view.

3.6 Future Work

Memoro currently works well and is easy to understand and use. Nevertheless, we have plans to improve it by reducing its runtime overhead. Currently, the compiler pass only traces the address operand from a load or store instruction across one arc in the SSA graph to see if it was produced by an `alloca` instruction. Although simple, this analysis reduces runtime overhead by roughly 10%. As we have seen with the LevelDB experiment, a large proportion of instrumented load/store instructions are not heap accesses. A more sophisticated analysis could trace more pointer operands from `alloca` instructions, and eliminate more instrumentation points and reduce overhead further. Moreover, it is possible to detect strided access patterns *e.g.*,to a string or array and track all of their access with a single runtime system call, rather than separately recording each memory reference.

Memoro currently stores the metadata for all freed chunks in arrays in memory until the program terminates. We believe the memory pressure and array resizing overhead could be reduced by fixing the amount of buffering and using a separate runtime thread to periodically write this data to disk.

As shown in Section 3.5.1, multithreaded programs with large allocations can suffer atypically large overhead due to implicit memory sharing and locking introduced by the runtime secondary allocator. It may be possible to reduce this overhead by avoiding the locking when looking up chunk metadata in the secondary allocator. Although there would likely be race conditions, the difference between this unsafe metadata modification and atomic operations would likely be statistically insignificant. Another solution could be to use purely thread-local allocators to avoid any implicit sharing. This may lead to higher memory overhead, but would likely a worthwhile trade-off to keep the runtime overhead low. Or more dramatically, we could change the memory allocator to make it possible to associate efficiently metadata with large regions of memory.

Our goal is to include Memoro in the sanitizer framework that is part of the Clang and GCC distributions. This will make this tool for detailed heap analysis readily available to users of these compilers on all platforms.

Moreover, the collection methods and data analysis presented in this chapter apply to other languages and runtime systems. Managed or dynamic language runtimes could track the

same data that Memoro collects and generate output in our compact binary format. The Memoro visualizer could then be used for these systems as well, performing the same analyses and score generation, and offering the same insight into heap and memory efficiency. As the visualizer is open source, it is also easy to add new scores or metrics for other languages, as their heap usage patterns may differ from native languages.

3.7 Conclusion

Memoro is a new, detailed heap profiler that uses a combination of static instrumentation, function interception, and runtime data collection to provide a clear view of how a program uses the heap, implemented in a low-overhead, cross-platform package. Memoro can show developers not only when and where bytes were allocated on the heap, but how and when the program used those bytes. The Memoro visualizer provides a method to distill the large amounts of data collected by the runtime into *scores*, displaying these alongside allocations. These color-coded, numeric scores help developers quickly pinpoint potential heap efficiency issues.

The result is accurate and fast diagnoses of heap efficiency problems, which in three different evaluations led to significant performance improvements. In the case of the BamTools API, an important and widely used tool in bioinformatics, our changes informed by profiling with Memoro led to a performance increase of over 10%. Memoro can help to optimize bioinformatics programs, and let them take full advantage of commodity hardware resources available in standard clusters and data centers, which, as Chapter 2 has shown, are a compute and cost-efficient way to scale and parallelize bioinformatics WGS preprocessing tasks. In the next chapter, this thesis will further demonstrate that this is also true for bioinformatics analyses downstream of genomics preprocessing.

4 Scaling Protein Similarity Search with ClusterMerge

The Persona system described in Chapter 2 showed that bioinformatics preprocessing pipelines can be effectively and efficiently scaled across commodity hardware clusters. However, many downstream applications consume this preprocessed data, which will also need to be parallelized and scaled as well. An important application is the analysis of protein sequences. Specifically, we are interested in finding similar or *homologous* proteins, which are those that share a common ancestry. Finding homologous proteins is important because it helps us identify evolutionary relationships between species, as well as to characterize newly sequenced proteins by inferring their function from known proteins.

Traditional approaches to finding similar proteins are expensive, typically involving $O(n^2)$ computations. However, recent work has shown that this can be reduced by clustering similar proteins first and then performing an intra-cluster all-against-all comparison. Unfortunately, the current implementation is sequential and does not scale.

In this chapter, we introduce a solution to this problem. First, the idea of using clustering to group similar elements together in order to find all similar pairs is generalized as *precise clustering*. Then, a new algorithm called ClusterMerge is introduced to perform precise clustering. ClusterMerge exploits the transitivity of similarity to build clusters and avoid comparisons — transitively similar sequences can be clustered together and stand in for each other in subsequent comparisons, allowing the algorithm to avoid many comparisons. Crucially, ClusterMerge exposes a large amount of parallelism, which we exploit to build highly parallel, scalable, and efficient solutions that can take advantage of commodity clusters.

This chapter will first review some background material on proteins and protein similarity search. The concept of precise clustering is then introduced, followed by a description of the new ClusterMerge algorithm. Our parallel and scalable solutions are then discussed and an evaluation is presented, showing that protein similarity search via clustering can be effectively parallelized and scaled across commodity clusters, with improvements of over 1400× compared to existing sequential solutions.

Bibliographic Note

This chapter is based on the paper *Parallel and Scalable Precise Clustering for Homologous Protein Discovery*, available on arXiv [39].

4.1 Background

This section will review the basics of proteins, their relation to genes, and current methods for finding similar proteins.

4.1.1 Proteins and Protein Sequences

A protein is a functional molecule made up of a folded amino acid chain. Proteins are coded in an organism's *genes*, where a gene is a subsequence of nucleotides (A, T, C, G) within the DNA. Three DNA letters correspond to one amino acid in the chain of the protein, and special three-letter combinations signal the stop and start of gene codings within the genome. During the process of gene expression, a protein is "read" from the coding gene. The DNA of the gene is first transcribed into a *messenger RNA* (mRNA) molecule, which is also comprised of nucleotides but is single-stranded. The RNA is subsequently translated by a ribosome, another complex macromolecule, into the amino acid chain that folds into the final functional protein. Individual amino acids in the chain are also referred to as *residues*. This process, while greatly simplified in this presentation, is commonly referred to as the central dogma of molecular biology [46] — DNA is transcribed to RNA which in turn is translated into functional proteins (Figure 4.1).

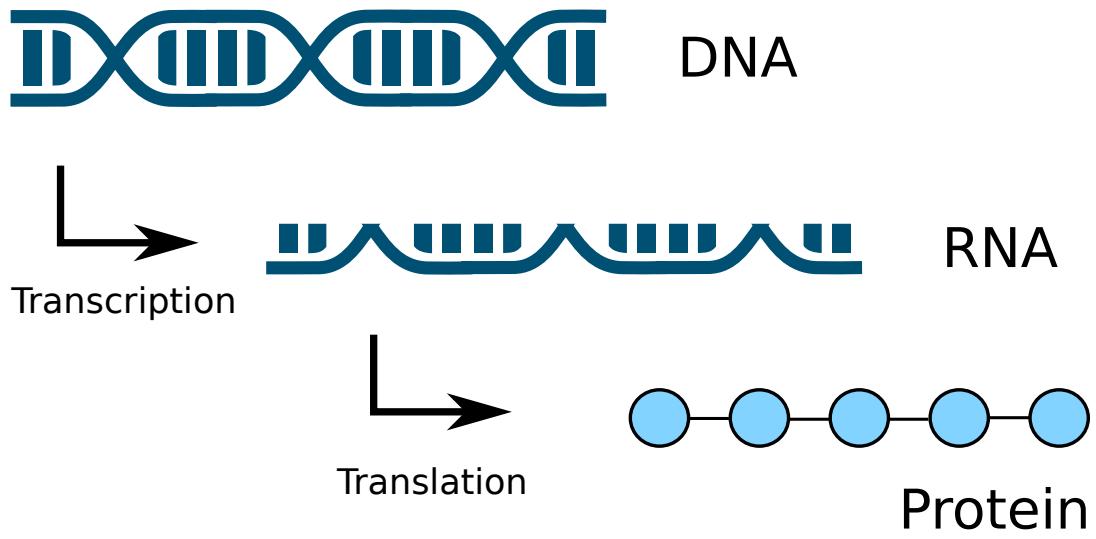


Figure 4.1 – The central dogma of molecular biology: DNA is transcribed to RNA, which is translated into proteins.

Genomics preprocessing pipelines, such as those accelerated by Persona, can be used to extract the protein sequences in a sample genome. From a whole sequenced genome, gene start and stop sequences can be detected and sequences extracted. Alternatively, techniques such as RNA-Seq [126] can sequence the RNA in an organism's cells directly (the *transcriptome*), however, this is a subset of the proteome. Other methods can sequence proteins directly, including Edman Degradation [52] and mass spectrometry methods.

Proteins perform many crucial functions in biological organisms. Recall, for example, the hemoglobin protein mentioned in Chapter 1. Hemoglobin is the protein responsible for transporting oxygen in the blood cells of almost all vertebrate species, including humans.

That hemoglobin is found in almost all vertebrates is no coincidence, it is a consequence of gene inheritance and the evolutionary process. Though small differences create variations in hemoglobin in species over time, similarities can help understand the evolutionary relationships between species. In this sense, proteins can be seen as proxies for genes, and similar proteins can allow us to infer common ancestry among genes.

Similar genes or sequences with shared ancestry are referred to as *homologs* [100], of which there are two types. *Paralogs* occur when an ancestral genome has two copies of a gene that diverge after speciation. *Orthologs* occur when genes diverge after speciation, usually retaining the same function. Because of their direct relation to speciation, orthologs are of particular interest to researchers.

Homologous genes or proteins, in general, are also important because their detection allows the transference of knowledge from well-studied genes to newly sequenced ones. Homologs, despite having accumulated substantial differences during evolution, often continue to perform the same biological function. Therefore, a newly sequenced gene or protein can be characterized by finding a highly similar protein whose function is known [29, 56]. In fact, most of today's molecular-level biological knowledge comes from the study of a handful of model organisms, which is then extrapolated to other life forms, primarily through homology detection. Several sequence homology techniques are among the 100 most-cited scientific papers of all time [125].

Because similar proteins imply an evolutionary relationship, homologs can also be used to help understand evolutionary paths between species, allowing the construction of phylogenetic trees [27].

4.1.2 Finding Similar Proteins

Finding similar proteins among a set essentially amounts to building a database, where an entry for each protein is linked to all the other similar proteins. The definition of *similarity* in this context may vary, but most sources, including this thesis, use a gapped alignment score generated by the Smith-Waterman (S-W) algorithm [115]. This algorithm can accurately model both insert and delete mutations that occur in nature, while also taking into account the fact

that some amino acids more readily replace others in proteins that have the same function but are usually found in different species. S-W uses a dynamic programming approach that always finds an optimal alignment and score, however, it is expensive, with a complexity of $O(n^2)$, where n is the length of the proteins.

Current approaches to finding similar (homologous) proteins are also computationally expensive. The baseline is to perform an exhaustive, all-against-all ($O(n^2)$, where n is the number of proteins) comparison of each sequence against all others using the S-W algorithm. This naive approach finds all similar pairs, but it scales poorly as the number of proteins grows. Several databases of similar proteins produced by this approach exist, including OMA [28] and OrthoDB [127]. Analyzing their content is costly. OMA, for example, has consumed over 10 *million* CPU hours, but includes proteins from only 2000 genomes.

The large amounts of data being produced by many laboratories require new methods for homology detection. In a report published in 2014, the *Quest for Orthologs* consortium, a collaboration of the main cross-species homology databases, reported: "*[C]omputing orthologs between all complete proteomes has recently gone from typically a matter of CPU weeks to hundreds of CPU years, and new, faster algorithms and methods are called for*" [116]. Ideally, a new algorithm with asymptotically better performance would find the same similarities as the ground truth, all-against-all comparison.

4.1.3 Saving Time with Clustering

One recent approach by Wittwer et al. [130] uses a clustering method to group sequences likely to be similar together. Then, similar proteins can be extracted by performing an all-against-all comparison within each cluster, avoiding many unnecessary comparisons between sequences in different clusters. The key to building clusters that do not miss similar pairs, while avoiding many unnecessary comparisons, lies in recognizing that protein sequence homology exhibits a transitive property. If sequence A is transitively similar to sequence B, and B is similar to sequence C, then C will be similar to A. Transitively similar homologous sequences can, therefore, be clustered together and represented by a single sequence. The represented sequences avoid all subsequent comparisons, relying on the representative to capture further similar sequences. Sections 4.3 and 4.3.1 will discuss in further detail what transitive similarity is and how it works for protein sequences.

The drawback, however, is that the algorithm implementing this approach is sequential and not parallel or scalable. Each sequence processed depends on the clustered state of all previous sequences, forming a synchronization bottleneck. The algorithm is still $O(n^2)$, at least it is while the number of clusters continues to grow.[1] A faster, preferably parallel and scalable approach, is required to deal with large-scale datasets.

[1] At some point, the growth of clusters would taper off as the clusters form a comprehensive representation of all genomes

Unfortunately, fast (sub $O(n^2)$) clustering algorithms — based on k-mer counting[2], sequence identity[3], or MinHash [37] — identify significantly fewer homologs [130], and hence are not practical for this application. Section 4.2 will review these papers in more detail. In the absence of a better algorithm, a scalable parallel implementation of an $O(n^2)$ solution would help keep pace with the production of sequence data.

This chapter presents just such a method: a new algorithm for clustering to find similar pairs in a set, which exposes parallelism that we can leverage to build highly parallel, scalable systems to execute it. First, this type of clustering is formalized as *precise clustering* — a clustering in which each similar pair is guaranteed to be found together in at least one cluster. Next, we describe a new algorithm called ClusterMerge that uses transitivity and a bottom-up cluster merging approach to build a precise clustering from a set of elements. We show how the algorithm enables the construction of highly parallel and scalable implementations, and apply this to the problem of finding homologous proteins. Our implementation finds 99.8% of ground truth homologs and runs over 1400× faster than the sequential approach, on a cluster of 32 commodity servers. This demonstrates clearly that even for downstream bioinformatics applications beyond WGS preprocessing, commodity clusters are an effective and efficient approach.

4.2 Related Work

Clustering, in general, has been the subject of considerable research, and we will review the literature in this section, with a particular focus on the clustering of bioinformatics data. Andreopoulos et al. surveyed the uses of many different clustering approaches in bioinformatics [31]. Common techniques are difficult to apply to protein clustering, however, because of the high level of precision required for this problem.

Many clustering algorithms are designed to generate *partitions*, where each element is assigned to exactly one partition. This can be a problem for homology detection via precise clustering, because some proteins are similar to many others, leading to a situation where, in order to find all homologous sequences, all proteins would all need to be in the same partition. This would defeat the purpose of the clustering in the first place. Multiple cluster membership is therefore required for proteins. Partitioning algorithms also require an equivalence relation between elements, which is stronger than the not-necessarily transitive similarity relationship in protein clustering.

Another common partitioning approach called k-means clustering requires a target number of clusters, which is unknown in advance for proteins, and also partitions the set. Hierarchical methods partition elements into a tree and preserve hierarchy among elements, but generally require a similarity matrix to exist, which is not the case for our problem, and they are expensive ($O(n^3)$). Of particular note is *agglomerative* hierarchical clustering, which also

[2] Comparing the number of matching k-mers between two sequences
[3] The number of characters that match between two sequences

uses bottom-up merge, e.g., ROCK [62]. Density-based clustering uses a local density criterion to locate subspaces in which elements are dense; however, they can miss elements in sparse regions and generally cannot guarantee a precise clustering. Density-based techniques have received attention from the parallel computing community, with the DBSCAN [108] and OPTICS [32] algorithms being parallelized by Patwary et al. [101, 102]

An additional complication of these methods is that they rely on distance metrics in normed spaces (e.g. Euclidean distance) that are usually cheap to compute. Edit distance, however, is not a norm and is expensive to compute. Although pure edit distance (i.e., Levenshtein distance) can be embedded in a normed space [98], it is not clear if the gapped *alignment* necessary for protein similarity can be as well.

The clustering of biological sequences is also the subject of considerable research. Many of these clustering algorithms employ iterative greedy approaches that construct clusters around representative sequences, a sequence at a time. If the sequence is similar to a cluster representative, it is placed in that cluster. If the sequence is not similar to any existing cluster representative, a new cluster is created with the input sequence as its representative. Some approaches use k-mer counting to approximate similarity (CD-HIT [83], kClust [65], Mash [97]), while others use sequence identity, i.e., the number of exact matching characters (UCLUST [51]). Of note is Linclust [117], an approach that operates in linear time by selecting m k-mers from each sequence and grouping sequences that share a k-mer. The longest sequence in a group is designated its *center* and other sequences are compared against it, avoiding a great deal of computation.

Unfortunately, sequence identity and k-mers are unsuitable for finding many homologs. Protein alignment substitution matrices are heterogeneous (e.g., BLOSUM62 [67]) since distinct amino acids may be closely related. Hence, protein sequences that appear different — with low sequence identity and therefore few or no shared k-mers — can often have high alignment scores. These important similar pairs will be missed by k-mer-based clustering techniques. For example, the fraction of similar sequence pairs found by kClust, UCLUST, MMSeqs2 linclust, and MMSeqs2 are 10.4%, 13.5%, 0.5%, and 36.4%, respectively. k-mer based methods are simply not capable of modeling protein sequence similarity to the degree necessary to capture all homologous sequences that we are interested in finding.

4.3 Precise Clustering

This section describes how to cluster items to efficiently find similarities and provides the basis for our parallel implementations. *Precise clustering* is a clustering that ensures each pair of similar proteins (or elements being compared, more generally) appear together in at least one cluster.[4] The similar elements can then be easily found by pairwise comparison of elements each of the resulting clusters. By clustering, many comparisons between dissimilar

[4] Since a protein sequence can be in more than one cluster, clustering is not partitioning.

elements can be avoided, because they will not be found in the same cluster. Traditional clustering techniques such as k-means, hierarchical clustering, density/spatial clustering, etc. are difficult to apply to proteins because they either partition the data and do not satisfy the precise clustering property, require a similarity matrix, or simply do not identify enough similar pairs.

For clustering elements such as proteins, the only assumption we can make is that we have a *similarity function* that produces a value when applied to two input elements. If this value is greater than a threshold, we conclude that the elements are similar. Otherwise, they are not similar. The only way to determine if two elements are similar is to compare them. We will use $f(i, j)$ to denote the similarity function — $f(i, j) > T$ therefore indicates similarity, where T is a parameter called the *threshold*. If $f(i, j) > T$, we say that i and j are a *significant pair*. We assume that the similarity function is reflexive, that is, $f(i, j) > T \iff f(j, i) > T$. We do not assume that the similarity function satisfies the triangle inequality, that is, $f(i, k) > T \wedge f(j, k) > T \not\Rightarrow f(i, j) > T$.

Clusters are defined by a single *representative* element. Every other element in the cluster is similar to the representative. However, cluster members are not necessarily similar to each other. Formally, a cluster C is a subset of elements that has a representative element r_C:

$$\forall e \in C, f(e, r_C) > T \tag{4.1}$$

The basic approach to building clusters with these minimal assumptions is to do so greedily and sequentially as with Wittwer's approach [130]. The first element forms the first cluster representative. The next element is compared against this representative, and if similar, it is added to the cluster. If not similar, it forms a new cluster with itself as representative. The process repeats until all elements have been clustered. The problem with this, however, is that when an element is added to a cluster, it is no longer available for subsequent comparisons and similar elements may be missed. This approach does not produce a precise clustering. An alternative is to form a cluster around a single element, in which case the solution devolves to an all-against-all comparison.

A stronger property than similarity, however, enables us to form a precise clustering, as well as to avoid unnecessary comparisons. That property is *transitive similarity*. If element i is *transitively similar* to an element j, the property guarantees that any subsequent element k that is similar to i will also be similar to j. If element i is a cluster representative, j can be clustered with i and all elements similar to j will be added to this cluster because of transitive similarity. No significant pairs involving j will be missed, and j does not need its own cluster. Transitive similarity is not necessarily reflexive, and we will discuss why this is the case for protein sequences in Section 4.3.1. Transitive similarity also implies similarity; if two elements are transitively similar, they are also similar.

More formally, we define a *transitivity function* $R(i,j)$ to denote transitive similarity between i and j.

$$\forall k, R(i,j) \implies f(i,k) > T \wedge f(j,k) > T$$

R is interpreted as follows: $R(i,j)$ evaluating true implies that if $f(i,k) > T$ then $f(j,k) > T$ for any k. That is, if two elements i,j are transitively similar, we know that any third element k that is similar to i will also be similar to j. Figure 4.2 shows a visual representation of transitive similarity between elements i and j.

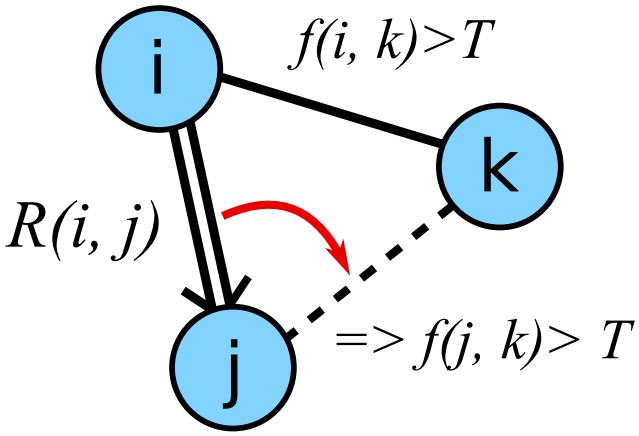


Figure 4.2 – The transitivity function $R(i,j)$ allows an inference that indicates j will be similar to k because i is similar to k.

R is required because the similarity function $f(i,j)$ is not guaranteed have a transitive property and we do not assume that it obeys the triangle inequality. Most other clustering methods use distances in normed spaces or similarity measures that can be embedded in normed spaces, which obey the triangle inequality (e.g., Euclidean distance). The clustering approach described in this chapter handles data that does not meet this requirement, which is necessary to cluster biological sequences, but could also be useful in many other contexts, such as clustering multimedia objects (e.g., images) or comparing time series (e.g., stock market, seismological, or climate data). Transitive similarity, as with similarity, is not necessarily inherent in the data. A domain expert must design $R(i,j)$ such that it satisfies the transitive similarity property.

The greedy sequential algorithm tests a new element for transitive similarity with a current cluster's representative. If an element is transitively similar to the representative, it is added to the cluster and can rely on the representative to match subsequent elements to which it is similar. If the new element is only similar to the representative, the element is added to the cluster but continues to be compared to representatives until either 1) a transitively

similar one is found or 2) a new cluster is created for the element. In this way, each element is clustered with those to which it is similar, either by becoming the representative of its own cluster or by relying on a transitively similar representative. This ensures a precise clustering while avoiding comparisons to sequences that are transitively represented.

4.3.1 Clustering Proteins

We will use proteins as an example to illustrate the concepts of similarity, transitive similarity, and clustering, as proteins are the primary motivation for this work. We use the Smith-Waterman (S-W) alignment score as the similarity function for proteins. If the S-W score is above a threshold T, two sequences are similar. Figure 4.3 shows three sequences, A, B, and C, where A is similar to B and B and similar to C. C, however, is not similar to A because it aligns with a different subsequence of B.

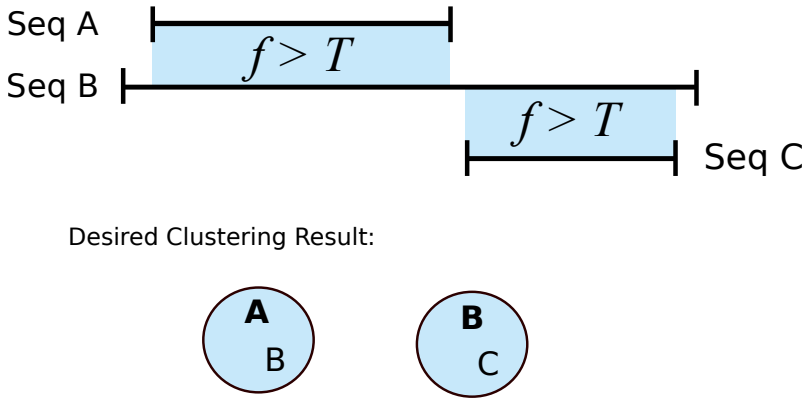
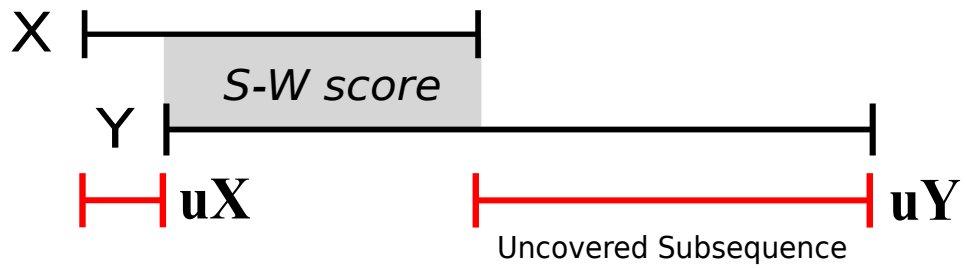


Figure 4.3 – Clustering protein sequences. Sequence A is similar to sequence B, which is similar to sequence C. C is not similar to A.

If we use only similarity to cluster, significant pairs can be missed — the order of evaluation will affect which significant pairs are found. For example, comparing $f(A,B)$ first will lead to a cluster represented by A, with B as a member. When C is then compared to the representative A, it is not similar and forms its own cluster. The (B, C) significant pair will be missed. Starting the process with sequence B would lead to a single cluster represented by B with A and C as members.

With transitive similarity, a precise clustering can be formed irrespective of comparison order. However, we first need a transitivity function for proteins. Protein sequence alignment does have a transitive property, however, S-W is a *local* alignment algorithm, meaning that it may not include ("cover") all residues (individual amino acids) in both sequences, especially when the sequences are of different lengths. If a sequence is clustered with a representative that does not completely cover it when aligned, the uncovered subsequence will be unrepresented. This may cause *subsequence homologies* to be missed, such as that between sequences B and C if B was represented by A.



$$R(X, Y) \implies score > minT,\ uY < maxU$$
$$R(Y, X) \implies score > minT,\ uX < maxU$$

Figure 4.4 – Transitivity function for protein sequences.

Therefore, subsequence homologies must be taken into account when designing a transitivity function for proteins. Figure 4.4 illustrates the transitivity function for proteins using two sequences X and Y. Depending on the size of each sequence and the alignment, there may be several uncovered residues in each sequence, shown as **uX** and **uY** in Figure 4.4. For one sequence to transitively represent the other, the alignment score between X and Y must be greater than $minT$, the transitivity threshold, a parameter typically greater than T. The number of uncovered residues in one of the sequences must be less than parameter $maxU$ (*maximum uncovered*), to ensure that homologous subsequences are not missed. In this example, Y could transitively represent X because **uX** is small, but X could not transitively represent Y because a large section of Y is uncovered. This example also illustrates that $R(i, j)$ for proteins is not reflexive.

Now we return to our example from Figure 4.3. Following the sequential clustering algorithm, $R(A, B)$ will indicate that B cannot be transitively represented by A, because the alignment does not cover a large subsequence of B. Because they are similar, however, B is added to the cluster with A and will also represent its own cluster. C can be transitively represented by B and will be placed in the cluster with B. We end up with the desired precise clustering shown in Figure 4.3. With transitive similarity, the order of comparison does not matter — any sequence not transitively represented by an existing cluster will represent its own cluster.

The sequential algorithm and transitivity function studied here is essentially that presented in [130], though they also augment the approach to use multiple representatives for a single cluster, which helps to slightly increase the percentage of significant pairs found. The drawback to this algorithm, however, is that it is sequential and difficult to parallelize because every element depends on the clustered state of all previous elements, forming a synchronization

bottleneck. In this chapter, we introduce ClusterMerge, a new algorithm for precise clustering using transitive similarity to expose parallelism, which can be exploited for highly scalable implementations that can leverage cost-efficient commodity clusters.

4.3.2 The ClusterMerge Algorithm

ClusterMerge is a new algorithm for precise clustering that exposes parallelism by structuring the clustering computation as a bottom-up merge of single-element clusters. In this section, we will explore how this is possible using transitive similarity while maintaining the precise clustering property.

Merging Clusters

The key to exposing parallelism lies in recognizing that clusters with transitively similar representatives can be *merged*. This allows us to reframe clustering as a series of *cluster merges*. Two clusters can be merged as follows. First, the representatives are compared using the similarity function f. If they are similar, the transitivity function R is applied to see if they are transitively similar. If so, the clusters can be combined into a single cluster, with one representative for all elements. Otherwise, if the representatives are only similar but not transitively similar, members of either cluster *might* be similar to the other representative. To avoid missing these similar elements, each cluster is compared against the other's representative and the similar elements are added to the other cluster. Finally, if the representatives are not similar, both clusters remain unchanged. The result is a set of one merged cluster or a set of two clusters whose representatives are not transitively similar and thus not mergeable.

Merging can also be applied to two *sets* of clusters. Algorithm 1 describes the process in detail. Each cluster in the first set (*cs1*) is compared to and possibly merged with every cluster in the second set (*cs2*). For each cluster pair, the process described above is applied. Finally, all un-mergeable clusters are returned in a new set.

Figure 4.5 illustrates the possible results of merging two clusters of proteins using this algorithm. Sequences X and Y are representatives of two clusters. Based on the result of the transitivity function (described above in Section 4.3.1) applied to X and Y, either 1) the cluster of X is merged into the cluster of Y, 2) the cluster of Y is merged into the cluster of X or 3) the clusters exchange similar members if X and Y are similar but not transitively similar. This results in a set of one fully merged cluster as in situations (1) and (2), or two clusters whose representatives are not transitively similar (situation 3).

ClusterMerge Algorithm

The ClusterMerge algorithm uses cluster merging to perform precise clustering. Each element is initially placed in its own cluster as its representative and each cluster is placed in its own

Algorithm 1 Cluster Set Merge

```
procedure MERGE(cs1, cs2)                                   ▷ merge cluster set 1, 2
    newClusterSet ← ∅
    for cluster1 in cs1 do
        for cluster2 in cs2 do
            if (cluster2.HasBeenMerged) continue
            s ← f(cluster1.rep, cluster2.rep)                ▷ Similarity
            if (s < T) continue
            if cluster2.IsTransitive(cluster1) then
                cluster2.Objs.append(cluster1.Objs)
                cluster1.HasBeenMerged ← True
                break
            else if cluster1.IsTransitive(cluster2) then
                cluster1.Objs.append(cluster2.Objs)
                cluster2.HasBeenMerged ← True
            else
                ExchangeSimilar(cluster1, cluster2)
            end if
        end for
        if !cluster1.IsMerged then
            newClusterSet.append(cluster1)
        end if
    end for
    for cluster2 in cs2 do                                   ▷ add unmerged clusters
        if !cluster2.HasBeenMerged then
            newClusterSet.append(cluster2)
        end if
    end for
    return newClusterSet
end procedure
```

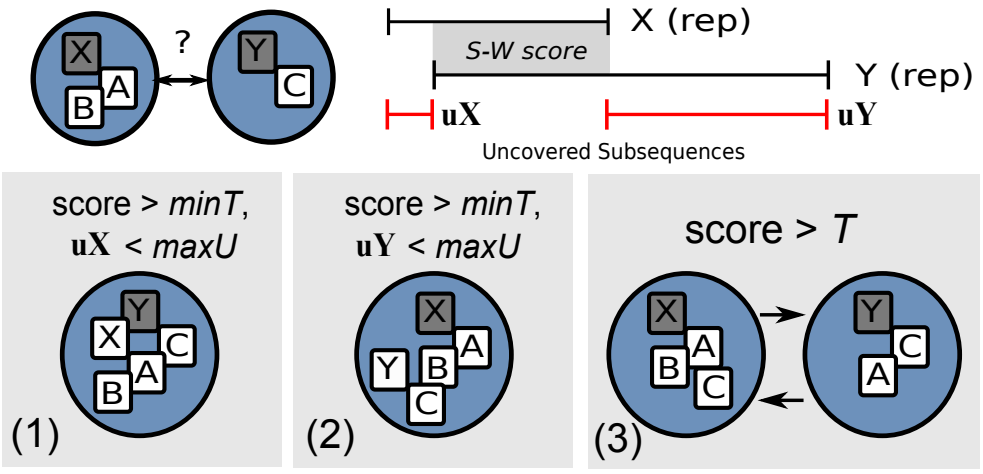


Figure 4.5 – Potential outcomes of merging two clusters of proteins.

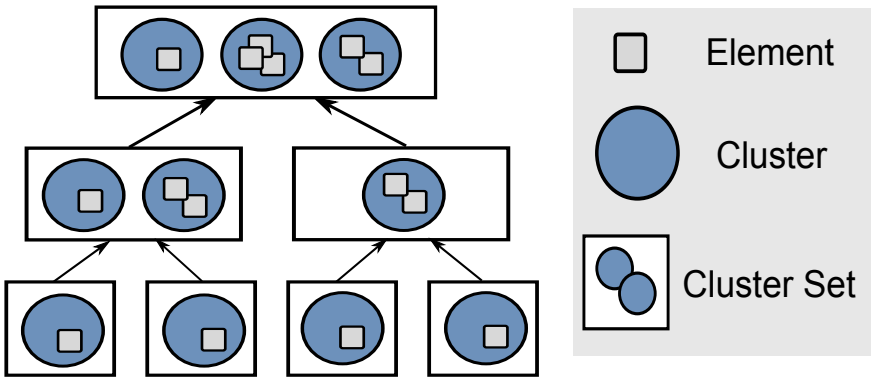


Figure 4.6 – ClusterMerge algorithm. Elements are placed in trivial clusters which are then merged until an un-mergeable set remains.

Algorithm 2 ClusterMerge

```
procedure BottomUpMerge(elements)
    setsToMerge ← Queue()
    for e in elements do
        setsToMerge.push(new ClusterSet(e))
    end for
    while setsToMerge.size() > 1 do
        cs1 ← setsToMerge.pop()
        cs2 ← setsToMerge.pop()
        csNew ← Merge(cs1, cs2)                ▷ merge sets cs1 & cs2
        setsToMerge.push(csNew)
    end while
    finalSet ← setsToMerge.pop()               ▷ final set of clusters
end procedure
```

set. Algorithm 1 is then applied to merge cluster sets in a bottom-up fashion as depicted in Figure 4.6.

Algorithm 2 describes this bottom-up merge process. To start, a new cluster set is created for each element, with a single cluster containing only that element. These cluster sets are added to a FIFO queue of sets to merge (the set merge queue). The algorithm pops two sets off the queue, merges them using Algorithm 1, and pushes the resulting cluster set onto the queue. The process terminates when only one set is left. This algorithm forms the basis of the ClusterMerge implementations further described in Section 4.4.

Discussion

With a complete[5] transitivity function, ClusterMerge will not miss any similar element pairs because all elements are implicitly compared against each other, either directly or implicitly via a transitive representative. The chosen element remains representative of its cluster until it is (possibly) fully merged with another cluster. After that, transitivity ensures that subsequent similar elements will then also be similar to the new representative. Therefore, even though cluster members are not necessarily *transitively* represented by the cluster representative, the algorithm also ensures that those non-transitively similar elements retain their own cluster.

In reality, a complete and computationally efficient transitivity function rarely exists for non-trivial elements, so an approximation is necessary, as in our motivating example of protein sequence clustering. Incompleteness in the transitivity function can lead ClusterMerge to miss some significant pairs. However, as is demonstrated in Section 4.5, even an approximate transitivity function can produce very good results. This is also why the transitivity function is applied both ways in Algorithm 1 — approximate transitivity is not necessarily reflexive.

The threshold value T is a parameter that would be chosen by an end end-user domain expert to specify the desired degree of similarity between elements. Users do not currently have influence over which elements are used as representatives, which are selected by the algorithm.

Complexity

The worst-case complexity of ClusterMerge is $O(n^2)$, however this is a fairly strict upper bound. Consider the tree structure formed by the cluster set merges, which has a depth of $log_2 n$, where n is the number of elements to be clustered. At the first layer, there are $n/2$ merges possible, each comparing two clusters of one element each. At there second layer, there are $n/4$ merges, each comparing a worst-case total of 4 clusters (if no full clusters were merged in the layer above). Generalizing this pattern we obtain

$$n/2 \times 1^2 + n/4 \times 2^2 + n/8 \times 4^2 \ldots$$

which we can reduce to

$$2n \sum_{i=0}^{log_2 n} 2^i = 2n \cdot 2^{log_2 n+1} - 1 = 2n \cdot (2n) - 1 \approx n^2$$

[5] A *complete* transitivity function correctly captures all transitive similarity in the data.

However, when clusters are fully merged, there is a reduction in work at each level, leading to sub-n^2 performance. In a more optimal case, assuming that at each step the merger of two cluster sets cuts the total number of clusters in half, complexity falls to $O(n\log n)$. Actual complexity, therefore, depends on the amount of transitivity in the data being clustered.

4.4 Parallel ClusterMerge

Now that we have a similarity and transitivity function for proteins, we can build a system to implement ClusterMerge. However, in order to handle large datasets, ClusterMerge will need to run in parallel and across multiple servers in a cluster.

There are several opportunities for parallelism inherent in ClusterMerge, which can be used to construct efficient systems for both shared-memory and distributed environments. Since the designs for shared-memory and distributed systems differ slightly, we will refer to the shared-memory design as Shared-CM and the distributed design as Dist-CM.

The obvious parallelism in ClusterMerge is that smaller sets near the bottom of the computation tree can be merged in parallel. In general, as long as there are sets of clusters to be merged in the set merge queue, a thread can pop two sets, merge them, and push the result back onto the queue. These operations are independent and can run in parallel.

However, after many merges, only a few large sets remain. The "tree-level" parallelism is no longer sufficient to keep system resources occupied, and the final merge of two sets is always sequential. Therefore, the merge of a set must also be parallelized, which is also beneficial since the sets can grow to be very large.

Shared-CM and Dist-CM both use the same technique to split large set merges into smaller work items called *partial merges*. Consider merging two cluster sets, *Set 1* and *Set 2* (Figure 4.7). A *partial merge* merges a single cluster from *Set 1* into a subset of the clusters of *Set 2*. Threads or remote workers can execute these partial merges in parallel by running the full inner loop of Algorithm 1. This allows the system to maintain a consistent work granularity by scheduling a similar number of element comparisons in each partial merge. Partial merge granularity is controlled by increasing or decreasing the size of the subset of Set 2 that the single cluster of Set 1 is merged into. The subset size is measured by counting the total number of sequences contained in the clusters, instead of counting clusters directly. This is because clusters can have extreme variation in the number of sequences they contain — five small clusters translates to much less computation than five large clusters. As a result of this partial merge technique, the load is evenly balanced, preventing stragglers and leading to good overall efficiency. Shared-CM and Dist-CM differ only in how they coordinate the synchronization of the results of partial merges.

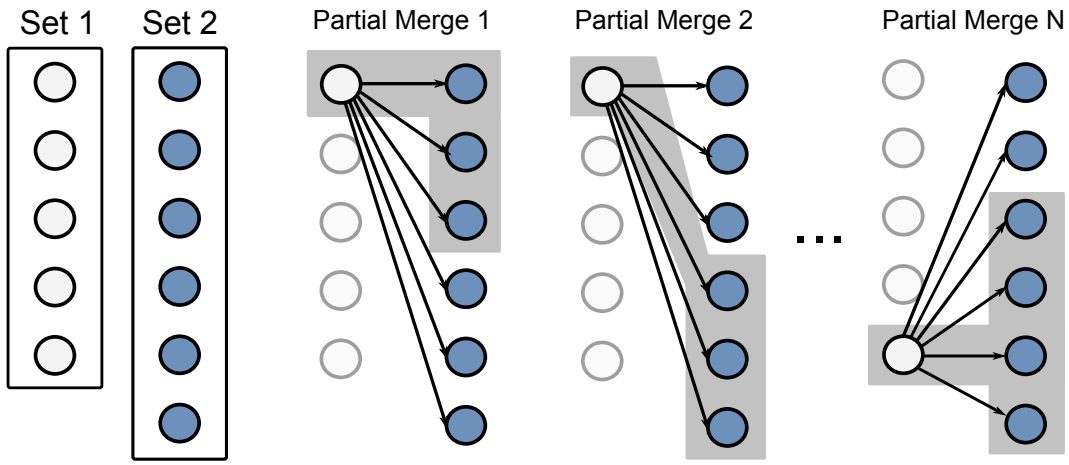


Figure 4.7 – A merge of two large cluster sets is split into partial merges. Threads (or remote workers) can then simultaneously process a merge of two sets.

4.4.1 Shared-Memory

Shared-CM is designed to be run on a multicore computer. Shared-CM splits set merges into partial merges, as described above, and allows threads to update the clusters in each set in place.

Consider a thread executing a partial merge, where a cluster from *Set 1* is being merged into some clusters from *Set 2*. While the thread has exclusive access to the cluster from *Set 1*, it has no such guarantee for the clusters in *Set 2*. Concurrent modifications, including removal of clusters and creation of new ones, can happen because of simultaneous partial merges of other items from *Set 1*.

Shared-CM uses locking to prevent races. The merging logic is the same as Algorithm 1, however, clusters of the second set are locked before being modified in-place. The final merged set is simply the clusters remaining in *Set 1* and *Set 2* that have not been fully merged. The order of merges is not fixed and the process sacrifices determinism, but the significant pair recall is as high as deterministic execution. The synchronized cluster merging algorithm is shown in Algorithm 3, where a cluster c from *Set 1* is being merged into a set cs.

Figure 4.8 illustrates the system design. A coordinating thread pops two sets off the *Set Merge* queue. The merge is split into partial merges, as described above, and they are inserted into a partial merge queue. A pool of worker threads then processes the partial merges. Once all of the partial merges in a set merge are completed, the coordinating thread collects remaining clusters from both sets into a new merged set and pushes it onto the set merge queue.

As long as there are sets to be merged, most or all processors in the machine can be kept busy. Although the design is simple, it is highly effective — Shared-CM can scale nearly linearly across all cores (Section 4.5.2).

Algorithm 3 Cluster Merge Locked

```
procedure MERGELOCKED(c, cs)                          ▷ cluster c, cluster set cs
    newClusterSet ← ∅
    for cluster in cs do
        if (cluster.HasBeenMerged) continue
        s ← f(c.rep, cluster.rep)                     ▷ Similarity
        if (s < T) continue
        cluster.Lock()
        if cluster.IsTransitive(c) then
            cluster.Objs.append(c.Objs)
            c.HasBeenMerged ← True
        else if c.IsTransitive(cluster) then
            c.Objs.append(cluster.Objs)
            cluster.HasBeenMerged ← True
        else
            ExchangeSimilar(c, cluster)
        end if
        cluster.Unlock()
        if (c.HasBeenMerged) break
    end for
end procedure
```

4.4.2 Distributed

While locking works well in a multicore computer, it would limit scalability on a distributed cluster. Dist-CM is a controller-worker distributed system. The *controller* is responsible for managing the shared state of the computation, while the majority of the computing is performed by *remote workers*. Dist-CM ensures that all processing sent to remote workers is fully independent. Workers, therefore, do not communicate with each other. They only communicate with a central controller to obtain work, resulting in a very scalable system.

Dist-CM uses several techniques to control the size of an average work item to prevent load imbalance and enable efficient scaling. First, batching is used to group small cluster sets into a single work item. This provides each remote worker with a computation that will not be dwarfed by its communication overhead. Batches are executed by a remote worker, and the resulting cluster set is returned to the controller. Batching is important for the early phase of a computation when each set is small and requires little computation.

For larger merges near the top of the tree, Dist-CM uses partial merges in much the same manner as Shared-CM to maintain a consistent work item granularity and keep all processors busy. Because there is no inter-worker communication, the controller is responsible for managing partial merge results as they are returned. Recall that each partial merge work item merges a single cluster from *Set 1* into a subset of clusters of *Set 2*. The result of a partial merge executed by a remote worker is a set containing some clusters of *Set 2*, with the single cluster

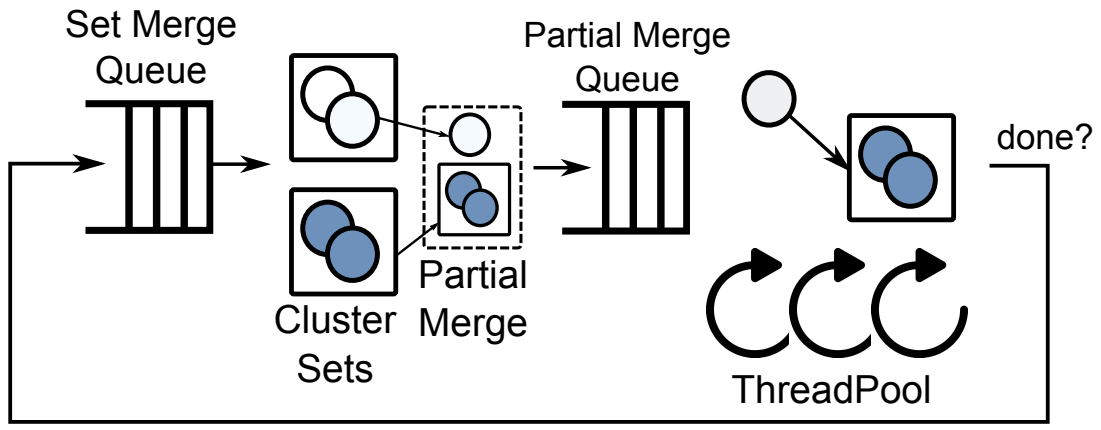


Figure 4.8 – High-level architecture of Shared-CM.

from *Set 1* possibly fully merged with one of them and/or some of its elements exchanged with the Set 2 clusters.

For each outstanding merge, the controller maintains a partially merged result, identified by an ID associated with all partial merges involved in its computation. This partially merged state begins as simply both sets of clusters. When a partial merge result is returned to the controller, it uses the ID to look up the associated partially merged state. The controller then updates the state with the returned results, adding new elements to existing clusters and marking any fully merged clusters. In the partially merged state, clusters are represented as hash tables, in order to avoid adding any duplicate sequences while retaining $O(1)$ insert time. After processing the final partial merge for a set merge, the resulting set is constructed by simply combining non-fully merged clusters from both sets.

Figure 4.9 illustrates the design of Dist-CM. Once again the set merge queue is loaded with single element cluster sets but at the central controller. A coordinating thread on the controller will pop two sets off the queue. If the sets (in terms of total clusters) are smaller than a batch size parameter, the thread will pop more sets until their size is equal to or greater than the parameter. These sets are compiled into a batch work item and pushed into a central work queue. Partial merges are then are pushed into the central work queue as individual work items. If the sets popped by the coordinating thread are large, they are split into partial merges, which are again sized based on the number of sequences in the clusters. This dynamic load balancing minimizes straggling in remote workers and is important to achieve good scaling. The central work queue feeds a set of remote worker nodes.

Results from workers are returned to the controller and either pushed onto the set merge queue if the result is a complete cluster set, or used to update the partially merged state if the result was a partial merge. If the partially merged state was completed by the result, the complete set is pushed to the set merge queue. The process finishes when the final set

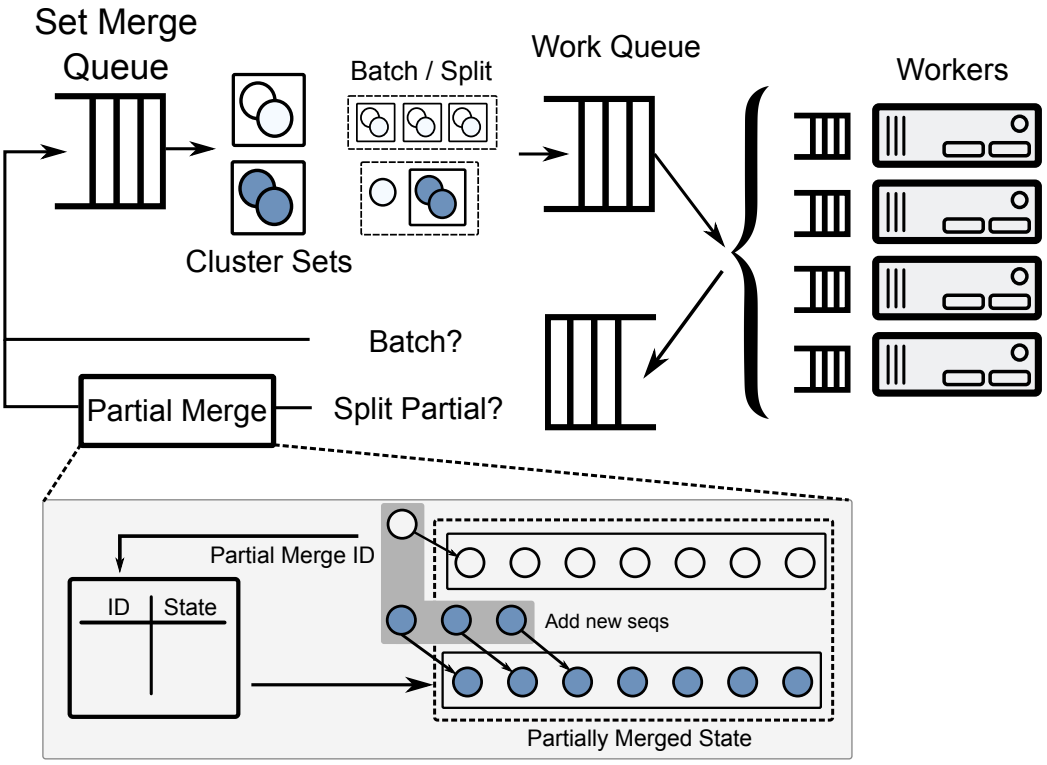


Figure 4.9 – High-level architecture of Dist-CM.

of the merge tree is complete. All messaging and communications are implemented with Zero-MQ [13], a popular lightweight distributed message queue library.

The trade-off inherent in this design is that Dist-CM does more work than necessary in exchange for no communication among workers. A cluster in a partial merge will continue to be merged into clusters in the set by Dist-CM even if these clusters were fully merged away by other workers. As a result, Dist-CM can perform slightly more work than Shared-CM and can occasionally add the same elements to the same cluster (these duplicates are removed by the controller). Because of this trade-off, Dist-CM is about 17% slower than Shared-CM when using a single remote worker.

Scalability can be adversely affected by communication overhead or the amount of work in a single work item. Very small work items will have communication overheads that may dwarf the actual computation. Very large work items can cause stragglers and load imbalance that can leave processors idle. Early versions of Dist-CM operated without dynamic sizing of partial merges; each merged a single cluster into an entire set. This led to massive load imbalance and long idle periods waiting for large merges to complete. With very large datasets, the in-memory structures representing the entire set can also grow very large and increase memory pressure in both workers and the controller. Dynamic sizing of partial merges is crucial to

ensure proper load balance and minimize stragglers, and it improved scaling efficiency by almost 3×.

Furthermore, unbalanced work distribution can cause stragglers if some workers locally queue more work than others. To avoid this, we switched from a round-robin work distribution method to a Join-Idle-Queue [85] style approach in which workers inform the controller when they need more work. A single thread in each worker is responsible for requesting work from the controller, whenever the worker's local queue has space free. This queue is kept small, balancing a trade-off between hiding latency and limiting the amount of locally queued work. This approach keeps all workers busy so long as work is available.

4.4.3 Optimizations

Several important optimizations enable efficient scaling of Dist-CM, mostly focused on memory usage and network bandwidth. In early versions, the controller sent entire sets with each partial merge request to a remote worker, which nearly saturated network bandwidth, especially when merging the last few sets with large clusters.

This communication overhead was greatly reduced through several techniques. First, each worker replicates the sequence dataset and refers to sequences by a 4-byte index. Actual sequence data is never transferred, and even large clusters with thousands of sequences only require a few kilobytes — simply an array of indexes. Still, with very large datasets, the total number of clusters and sets can grow large and memory use can become an issue, particularly in the controller that stores the entire state of the computation. The in-memory data structures therefore also use indexes to represent sequences to reduce the memory footprint.

Second, workers cache copies of sets so they are only transferred over the network once to each remote worker. To execute a partial merge, the worker can construct the appropriate subset of a set from its cached copy. A partial merge request on the wire then simply contains only one cluster from the first set and two indices from the second set, which indicate the subset into which the first cluster is to be merged.

Finally, the results of a partial merge are returned as diffs. Only newly added sequences in each cluster are sent back to the controller. This reduces network bandwidth by a considerable amount, especially when merging a small cluster into a subset of larger ones.

4.5 Evaluation

This section describes the evaluation of several aspects of ClusterMerge applied to protein sequence clustering. Both Shared-CM and Dist-CM variants are evaluated in this section. Both implementations are written in C++ and compiled with GCC 5.4.0. To compute sequence similarities, we used the Smith-Waterman library SWPS3 [119].

We use two datasets for our evaluation. One is a dataset of 13 bacterial genomes extracted from the OMA database [28], a total of 59013 protein sequences (59K dataset). This is the same dataset used by Wittwer et al., which allows comparison with their implementation. The second dataset is a large set of eight genomes from the QfO benchmark totaling 90557 sequences (90K dataset). Although these are a small fraction of the available databases, each represents billions of possible similar pairs, and require many hours to evaluate in a brute-force manner.

Our tests are performed using servers containing two Intel Xeon E5-2680v3 running at 2.5 GHz (12 physical cores in two sockets, 48 hyperthreads total), 256 GB of RAM, running Ubuntu Linux 16.04. The distributed compute cluster consists of 32 servers (768 cores), a subset of a larger, shared deployment. These are connected via 10 Gb uplinks to a 40GbE-based IP fabric with 8 top-of-rack switches and 3 spine switches. This is the same cluster setup used in the Persona evaluation in Section 2.4. The dataset is small enough such that a local copy can be stored on each server. In fact, even large protein datasets are easily stored on modern servers. For example, the complete OMA database of 14 million protein sequences fits within 10GB, a fraction of modern server memory capacity.

Our baseline for clustering comparisons is Wittwer's incremental greedy precise clustering of [130], which is the only clustering method that can achieve an equivalent level of similar pair recall.

4.5.1 Clustering and Similar Pair Recall

For consistency, our clustering threshold is the same as the incremental greedy precise clustering in Wittwer et al. [130], a Smith-Waterman score of 181. The threshold is low, but this is necessary to find distant homologs. After ClusterMerge identifies clusters, an intra-cluster, all-against-all comparison is performed, in which the sequence pairs within a cluster are aligned using Smith-Waterman. Those with a score higher than the clustering threshold are recorded as a similar pair. For our datasets, the number of actual similar pairs is small compared to the number of potential similar pairs (e.g. 1.2 million actual versus 1.74 billion potential), leading to relatively few alignments to complete this stage. Biologists may perform additional alignments to derive an optimal alignment concerning different scoring matrices, however, this is orthogonal to the concerns of this thesis.

Recall is the percentage of ground truth pairs found by our systems. The ideal recall is 100%. Both Shared-CM and Dist-CM ClusterMerge, using a minimum full merge score ($minT$) of 250 and a max uncovered residues ($maxU$) of 15, produce clusters with a recall of $99.8 \pm 0.01\%$. Recall variability is negligible and is due to the non-determinism of parallel execution. Of the pairs missed by ClusterMerge, very few were high scoring pairs. The median score of a missed pair is 191 and the average score of a missed pair is 235. These values are very close to the cluster threshold itself (in contrast to high scoring pairs, which can be greater than 1000), indicating that these are not likely biologically "important" pairs (Figure 4.10). ClusterMerge

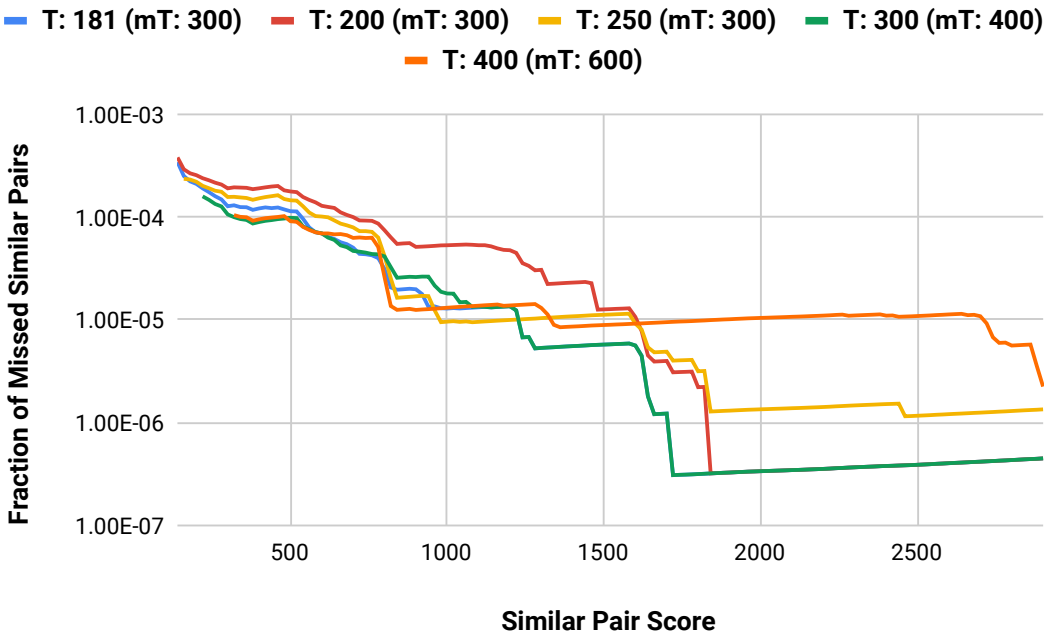


Figure 4.10 – Cumulative fraction of missed pairs reaching at least a certain similarity score, as the clustering threshold T and fully merge threshold $minT$ are varied ($maxU = 15$). ClusterMerge shows a low sensitivity to small parameter variations, while most missed similar pairs remain low-scoring ones.

misses only a handful of high scoring pairs, around one-millionth of total significant pairs, as seen in Figure 4.10.

In clustering the 59K sequence dataset, ClusterMerge performs approximately 871 million comparisons. By contrast, the full, all-against-all comparison requires approximately 1.74 billion comparisons, showing that ClusterMerge reduces comparisons by nearly 50%.

In terms of the clusters themselves, ClusterMerge generates similar clusters as incremental greedy clustering [130], with a total of 33,562 clusters. In each, the vast majority of clusters contain between 1 and 4 sequences, with a few large clusters (33% of clusters contain more than 10 sequences, 8% of clusters contain more than 100 sequences, 0.5% of clusters contain more than 1000 sequences). ClusterMerge generates slightly larger outliers, with its largest cluster containing approximately 1500 sequences, as opposed to the greedy method's largest cluster of around 1150 sequences.

Figure 4.10 shows that ClusterMerge and our transitivity function are relatively insensitive to parameter variations. Lower clustering thresholds T and lower full merge thresholds $minT$ generally lower the number of missed similar pairs, although the absolute percentage of missed pairs remains extremely low, with the majority being low-scoring pairs.

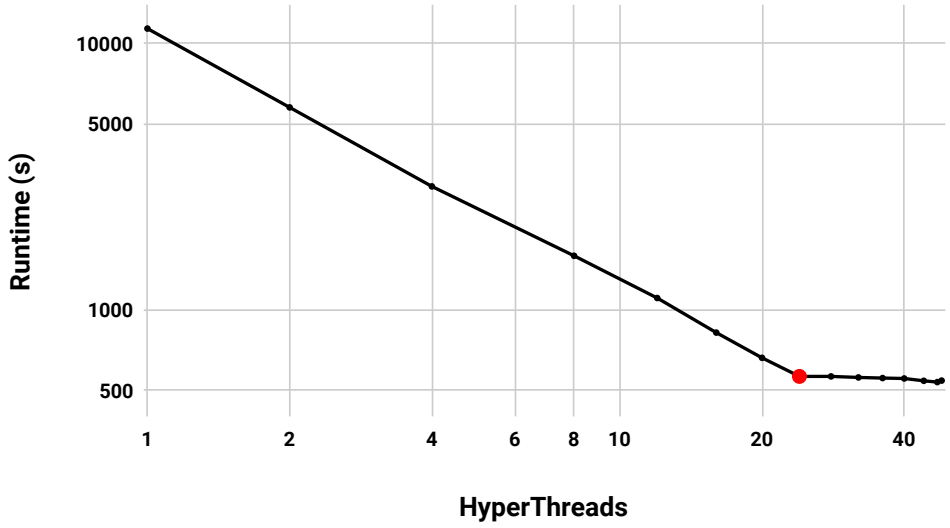


Figure 4.11 – Scaling of Shared-CM up to 48 threads. Scaling is nearly linear up to all 24 physical cores, while hyper-threading provides no benefit.

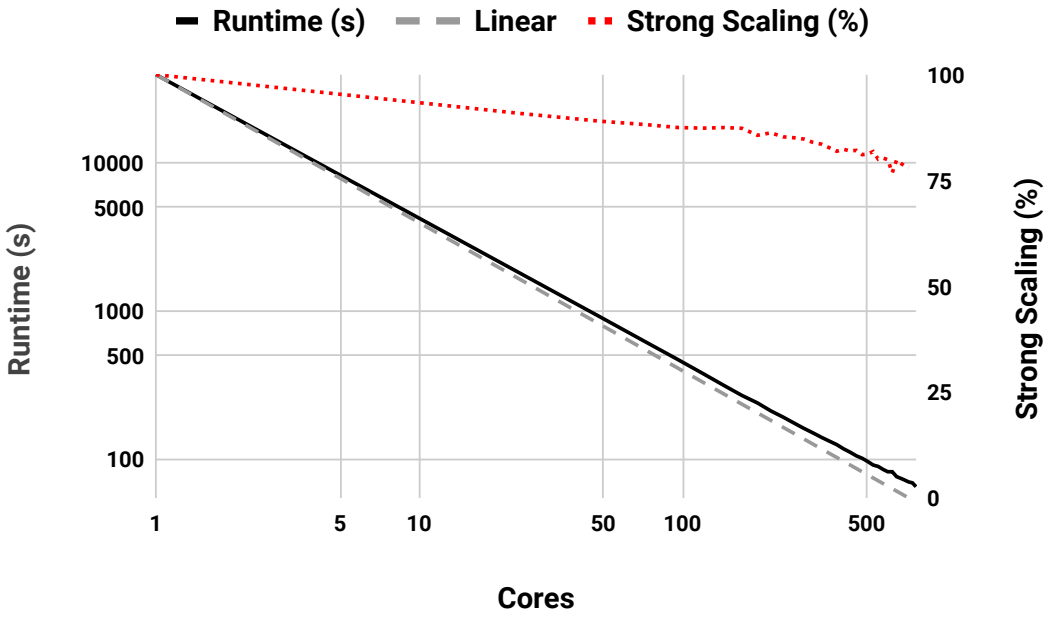
4.5.2 Multicore Shared-Memory Performance

In this section, we evaluate how well Shared-CM performs on a single multicore node. This experiment uses a reduced dataset of 28600 sequences, to lower runtimes at low thread counts. Figure 4.11 shows the total runtime decreases as we increase the number of threads. Shared-CM achieves near-linear scaling — profiling with Intel VTune indicates little or no lock contention. Memory access latency and NUMA costs have no effect as the workload is compute-bound.

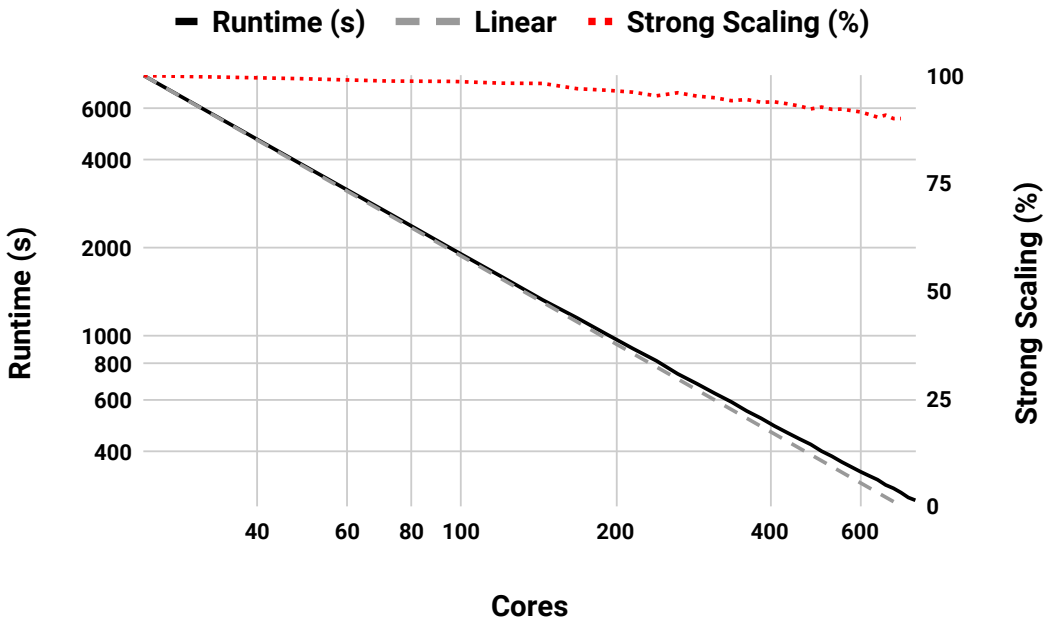
Note, however, that scaling is linear only on physical cores. The primary compute bottleneck is the process of aligning representative sequences using Smith-Waterman, which processes data that fits in the L1 cache and can saturate functional units with a single thread. Therefore, hyper-threading provides no benefit.

The only major impediment to perfect scaling is some loss of parallelism before the last and second-last merges since the second-last merge must be fully completed before work for the last merge can start to be scheduled.

Shared-CM with a single thread clusters the bacteria dataset in 31905 seconds, compared to 1486 seconds with 24 threads, a speedup of 21.5×. To compare with incremental greedy clustering, we run Wittwer's single-threaded code [130] on our machine with the same dataset, resulting in a runtime of 89486 seconds. Shared-CM is approximately 2.8× faster on a single-core and 60.2× faster using all cores.



(a) 59K dataset, 79% scaling efficiency.



(b) 90K dataset, 90% scaling efficiency.

Figure 4.12 – Scaling of Dist-CM over 32 servers (768 cores).

4.5.3 Distributed Performance

Dist-CM allows us to scale ClusterMerge beyond a single server. To evaluate the scaling of Dist-CM, we hold the dataset size constant and vary the number of servers used to process

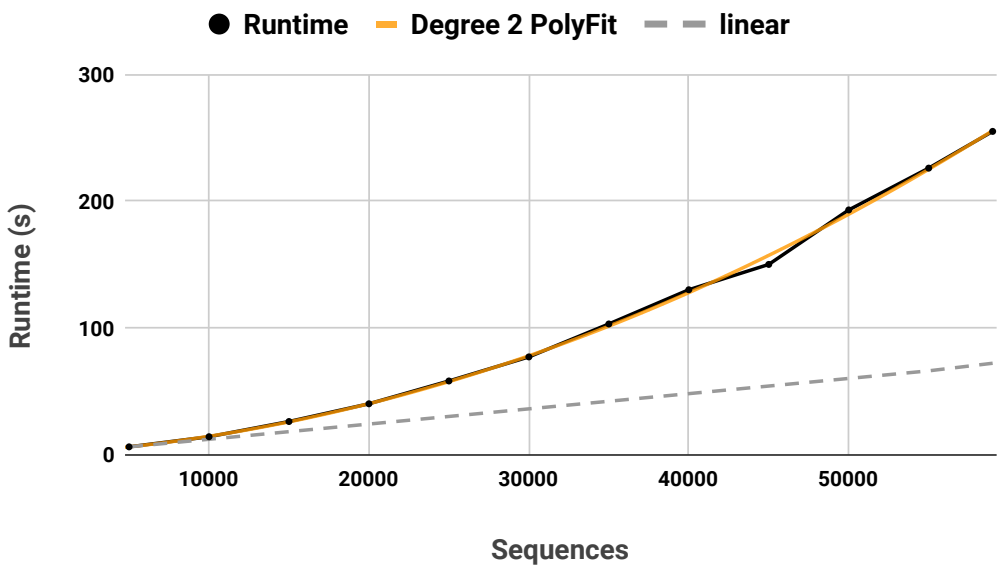


Figure 4.13 – Workload scaling of Dist-CM.

work items (batches or partial merges), otherwise known as *strong scaling*. The baseline single core runtime for Dist-CM clustering the 59K dataset is 39314 seconds. Figure 4.12a shows that on 32 nodes (768 cores), Dist-CM clusters the dataset in 65 seconds, resulting in a speedup of 604×. Strong scaling efficiency at 768 cores is 79%. Compared to single-threaded incremental greedy clustering [130], Dist-CM is 2.27× faster using a single core, and 1400× faster using the full compute cluster.

The reason for non-linear scaling is essentially the same as Shared-CM — around the last few merges of cluster sets, scheduling of work may halt as the system waits for an earlier merge to finish before it can schedule more work. There will always be some small portion of sequential execution, so perfect scaling is impossible by Amdahl's Law.

That being said, this sequential section is proportionally lower with larger datasets. Figure 4.12b shows strong scaling when clustering the larger 90K sequence dataset. The scaling is more efficient (90% at 32 nodes), with a speedup of 28.7× relative to one worker node.

In addition, we perform a weak scaling experiment in which we vary the amount of work in proportion to the number of nodes. Because our dataset is evolutionarily diverse and has relatively low levels of transitivity, ClusterMerge is closer to $O(n^2)$ in the number of sequences. The number of comparisons increases quadratically with the number of sequences. Figure 4.13 clearly shows this by varying the number of sequences that Dist-CM clusters using 10 worker nodes. The runtime curve fits almost exactly to a degree two polynomial. Therefore, for our weak scaling experiment, we increase the number of sequences at each step by a square root factor to maintain a proportional increase in workload. Figure 4.14 shows the results, again while clustering using 1 to 32 nodes. Runtime remains nearly constant throughout, indicating

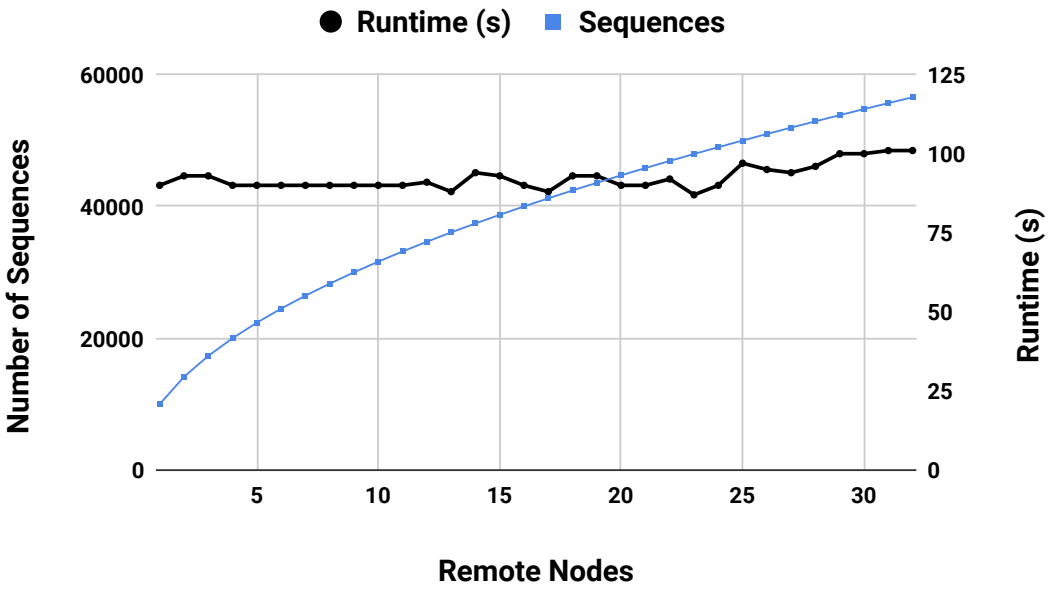


Figure 4.14 – Dist-CM weak scaling over a 32 node (768 core) cluster. Nearly 100% efficiency at 32 nodes.

a weak scaling efficiency of 95-100%. We thus expect that Dist-CM will be able to cluster much larger datasets while maintaining high scaling efficiency.

4.5.4 Effect of Dataset Composition

As noted in Section 4.3.2, the complexity, and therefore runtime depends on how many clusters can be fully merged at each level of the tree. If the transitivity function accurately represents similar elements, the number of full merges at each level is primarily affected by the number of transitively similar elements in a dataset. More transitively similar elements will result in more complete cluster merges, bringing runtime cost closer to the $O(n\log n)$ optimum.

For protein clustering, the dataset with 13 bacterial genomes has a relatively low number of transitively similar sequences since the species are, genetically speaking, very distant (more distant than humans and plants). Given a set of more closely related genomes, with more transitively similar sequences, we hypothesize that ClusterMerge will generate fewer clusters and run much faster, due to more clusters being fully merged. To test this hypothesis, we clustered a third dataset of more closely related *Streptococcus* bacteria genomes, consisting of 33 genomes of different Streptococcus strains (69648 sequences, a similar number to the other dataset).

Using Shared-CM with 48 threads, the clustering is completed in 283 seconds, producing 10500 clusters. As predicted, clustering is much faster than the 13 bacterial genome dataset (1486

seconds), as the number of transitively similar sequences leads to more full cluster merges, which are both fast and lead to less work further up the merge tree. Again, ClusterMerge produced a high recall of 99.7% of similar pairs relative to a full all-against-all. The particular genomes being clustered therefore have a marked effect on the total runtime of the algorithm.

4.6 Discussion

As there are theoretically a limited number of different protein sequences in existence, it is worth thinking about the limits of clustering proteins. It is possible that, at some point, enough proteins from different branches of the tree of life will be clustered such that any subsequent protein would be completely represented by the existing set of cluster representatives. ClusterMerge would not add any new clusters, and any subsequent protein could be classified in $O(n\log n)$ time. This would form a complete, protein-based "map" of the entire evolutionary tree of life, a powerful tool for understanding the path of evolution on earth.

However, it is not yet clear how many different genomes or proteins would be required to form such a map. It may be still some years before enough organisms have been sequenced. It is also possible that ClusterMerge will need to be augmented by other approaches that are optimized for expanding a large set of existing clusters, rather than building clusters from individual sequences. For example, ClusterMerge would be used to build an initial set of clusters from a diverse set of proteins, with a simpler parallel iterative approach taking over when the likelihood of adding a new cluster to set reaches a low enough threshold.

4.7 Conclusion

ClusterMerge is a parallel and scalable algorithm for precise clustering of elements. When applied to protein sequences, ClusterMerge produces clusters that encompass 99.8% of significant pairs found by a full all-against-all comparison, while performing 50% fewer similarity comparisons. Moreover, ClusterMerge is very parallel, and our implementations achieve speedups of 21.5× on a 24-core shared-memory machine and 604× on a cluster of 32 nodes (768 cores). The distributed implementation of ClusterMerge for protein clustering (Dist-CM) can produce clusters 1400× faster than a single-threaded greedy incremental approach, while maintaining the same level of significant pair recall. We hope that ClusterMerge will enable protein database projects such as OMA [28] to continue to build larger databases, and continue to scale as more data becomes available.

5 Conclusion

Next-generation sequencing technologies are now being applied at scale, sequencing thousands of genomes from various organisms and generating vast amounts of data. This data requires a large amount of computational analysis to unlock its secrets, which has given rise to the field of bioinformatics. This thesis shows that important computational problems in bioinformatics can be effectively parallelized and scaled out efficiently on commodity hardware clusters. Horizontal scaling and cost-effective analyses will allow bioinformatics processes to continue to meet the demands of increasing data volumes and the computational needs of businesses and researchers.

This thesis has presented three research endeavors in support of the statement. First, whole-genome sequence preprocessing is addressed using the Persona scalable bioinformatics framework. Preprocessing consists of computational stages that prepare raw sequence reads for analysis: aligning sequenced reads to a reference, sorting them, marking duplicate reads, as well as other operations. This process can take many hours with current approaches, which typically involve stringing together individual tools using scripts or other ad-hoc methods. Current file formats are also bottlenecks that prevent effective scaling on parallel computers.

Persona solves these issues by integrating different algorithms and tools into a single framework that is engineered to run efficiently on a cluster. The Aggregate Genomic Data file format unifies data under a single partitioned, column-oriented storage scheme, which supports distributed computation and reduces I/O demands. In combination with AGD, Persona leverages a combination of fine- and coarse-grain dataflow execution, load balancing, and thread and node-level parallelism to scale preprocessing across a cluster. In particular, sequencing mapping or alignment is scaled nearly linearly, aligning 223 million reads in only 17 seconds, on a cluster of 32 nodes.

Second, the *Memoro* detailed heap profiler aids developers in making more efficient use of local memory resources and improving their program's performance. For bioinformatics, in particular, many tools are open source and have been developed by non-expert software developers. This leaves many opportunities for optimization, particularly in the area of heap

usage. Inefficient heap usage can lead to serious performance degradations that are often difficult to find.

Memoro solves this problem by implementing *detailed heap profiling*. In this technique, instrumentation inserted by a compiler at every memory access record statistics whenever the program accesses heap memory. *Memoro* can determine not just when and where memory is allocated, but how the program used it, including reads, writes, and when and where those reads and writes occurred. *Memoro* distills all of this recorded data into a *score* for each allocation, with a low score indicating a higher probability of performance issues arising with data allocated at that point. Programmers can use the Memoro visualizer to easily find memory performance issues in their code.

Memoro is effective for bioinformatics programs — fixing issues found by the tool led to a 10% performance increase in the BAMTools API, a common library for building bioinformatics applications.

Finally, protein sequence similarity search is explored using the ClusterMerge system. Preprocessing in whole-genome sequencing is not the only step at which new methods are required to deal with large amounts genomic data. Downstream applications include building databases of sequenced and characterized proteins, a computationally expensive process that typically involves aligning all proteins against one another (all-against-all), an $O(n^2)$ computation. These databases are used to characterize new proteins and explore evolutionary histories and are thus very important to biological research. However, new methods are needed to future-proof their construction in the face of growing data — a lower bound of $O(n^2)$ does not scale well.

ClusterMerge uses precise clustering to build parallel solutions to this problem. Precise clustering is a clustering that ensures any two similar elements will be in at least one cluster together. Similar elements can then be easily found by comparing only elements within a cluster, reducing the total operations required. To efficiently build a precise clustering and not miss any similar pairs, ClusterMerge relies on transitivity between elements. Transitively similar elements can stand in for one another in subsequent comparison computations, allowing inference of similarity and avoidance of actual computation. The ClusterMerge algorithm then builds clusters by putting each element (protein sequence) in its own cluster and merging similar clusters.

Crucially, ClusterMerge exposes parallelism in its tree-like computation structure, which we leverage to build scalable implementations. Individual merges of clusters can also be parallelized and distributed. The resulting implementations can build precise clusterings of protein sequences 1400× faster than previous implementations.

All together, these various examples show how much the bioinformatics field can benefit from designs that carefully consider how to parallelize and run efficiently on commodity clusters. We maintain that this is the best approach to deal with the increasing amounts of bioinfor-

matics data. Commodity clusters are widely available, well understood, and much more cost-efficient than specialized supercomputing solutions. They can also easily be augmented with accelerators for specific workloads (Graphics Processing Units, Field-Programmable Gate Arrays) or integrated with different storage subsystems such as HDFS or the Ceph object store used for Persona. Given this flexibility, and the fact that bioinformatics workloads *can* be effectively parallelized, distributed, and scaled out, we believe commodity clusters are the right choice.

5.1 Discussion

Alongside the central statement of this thesis, Persona, Memoro, and ClusterMerge offer other valuable insights and lessons, while reaffirming some common themes and wisdom from computer systems design.

5.1.1 Common Themes

Persona and ClusterMerge differ in the structure of their computations. Persona, particularly the alignment workload, is embarrassingly parallel, while ClusterMerge maintains a central state that requires communication. However, once algorithmic innovations allowed ClusterMerge to parallelize and scale, we can see many similarities between the two systems.

Work imbalance is a crucial issue in both systems. In Persona/SNAP alignment, one sequence may generate significantly more work than another. A read to be aligned to the reference may generate many possible candidate locations if it has many hits in the hashed index of the reference. If there are few good candidates (i.e., locations where the read aligns well with the reference genome) among these, many will be evaluated during the search, each costing an additional edit distance computation. To show these differences in alignment times, we measure the alignment times for 50,000 random reads and show a scatterplot of alignment times shown in Figure 5.1. The alignment time of a read is roughly correlated with the number of candidates that were evaluated during its alignment. The mean alignment time is only 96 microseconds, however, a significant number of reads can take up to 16 milliseconds — several orders of magnitude more time.

ClusterMerge also experiences work imbalance from two related sources. First, individual sequences being aligned can vary in size by orders of magnitude (e.g. aligning a 400 residue protein to a 30000 residue protein). Because alignment complexity is proportional to the product of the sequence lengths, longer sequences can take a *much* longer time to align. Figure 5.2 shows alignment times for small and large matrix alignments. As the sequence length product increases into the millions, alignment time (using SWPS3) goes from tens of microseconds to multiple seconds — five orders of magnitude larger. Note that the axes use a log scale, indicating that the time increases are exponential. Second, partial merge work units, where one sequence of a cluster is merged with a subset of sequences from the other cluster,

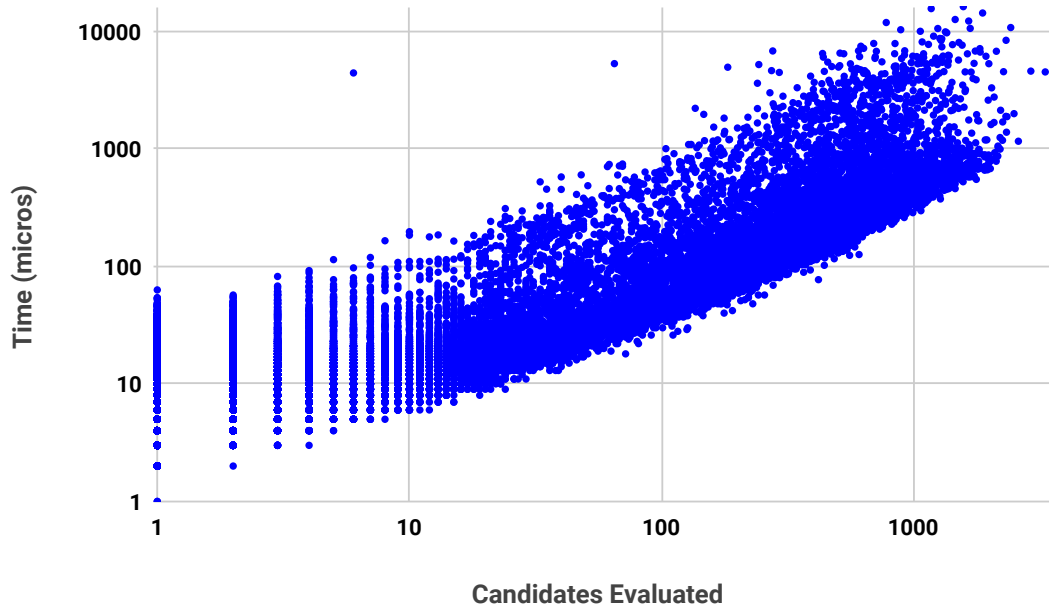


Figure 5.1 – Read alignment time in Persona/SNAP, which is roughly correlated to the number of candidate locations evaluated.

can be imbalanced again due to sequence size differences. As we can see from Figure 5.2, a very long sequence being merged into a cluster will consume much more time than a short sequence being merged into the same cluster. Though the remedies to these problems were described in the respective Persona and ClusterMerge chapters, work imbalance was generally one of the main challenges when designing these systems, especially for the scale-out design.

Work imbalance can severely reduce scaling. Initial tests of Persona that aligned entire AGD chunks using single threads experienced a large amount of straggling, where all threads save one or two finish executing, leaving most of the system idle while the remaining work completes. Likewise, ClusterMerge experienced a similar problem when partial merges consisted solely of one cluster sequence being merged into another cluster — at large dataset sizes, each partial merge in later stages of the merge tree was a large unit of work, leading to a large amount of straggling and system idleness. While this thesis is not about load balancing, the experience of building these systems has made it clear why it is an important research topic on its own, and why it is a prime consideration when designing scalable distributed systems.

Another source of load imbalance, as well as another common theme in ClusterMerge and Persona, is queuing. Queues manifest at many levels in a distributed system, often implicitly. TensorFlow queues in Persona were explicitly declared with a particular size and used to separate the functional domains of the system. ClusterMerge had some explicit queues, e.g. for distributing thread local work, but the ZeroMQ messaging layer also queues requests and responses implicitly and transparently to the system.

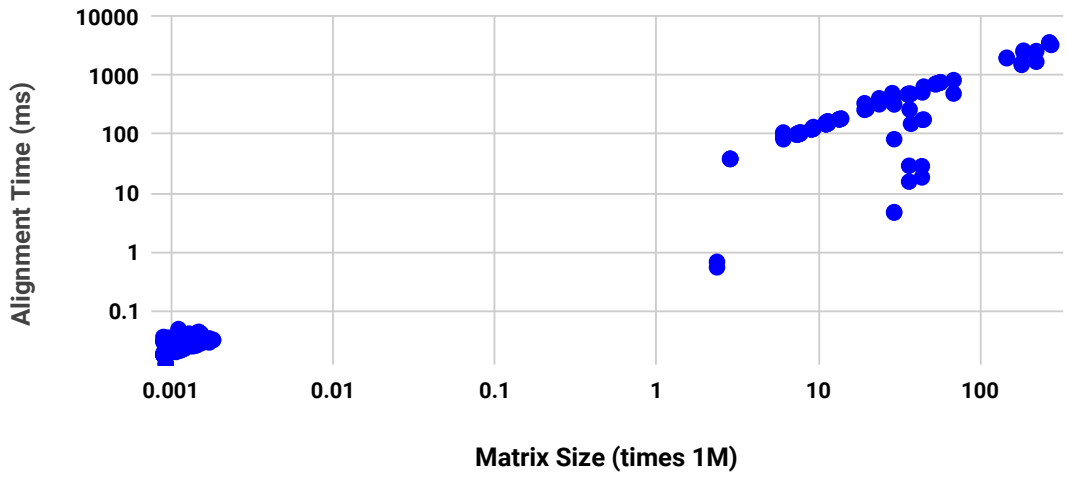


Figure 5.2 – Protein sequence alignment time versus matrix size (the produce of the sequence lengths).

The performance of both systems was affected by queuing and both had to be tuned for high performance. Persona queues were limited to relatively small sizes, otherwise, local workers in the cluster would queue work that could not be utilized by other workers when they became idle[1]. In terms of system-level work distribution, TensorFlow queuing and data movement was clear and explicit. Subgraphs on each remote worker would pull data from a central queue and locally queue a few work items, to keep all processors busy. By contrast, ClusterMerge work distribution was less clear because the underlying ZeroMQ layer hid these details. As mentioned in Chapter 4, the use of ZeroMQ was modified to achieve a Join-Idle-Queue [85] style approach similar to that used by Persona. Otherwise, the default round-robin distribution led to load imbalances. In distributed computational systems, we can see that queuing and load balancing are related, and proper handling of trade-offs is crucial to the design of efficient systems.

Memory management is also an important consideration when building high-performance computational systems. In their development, both Persona and ClusterMerge had memory usage issues that needed to be diagnosed and fixed. Tools, such as Memoro presented in Chapter 3, or profilers like Intel VTune, are extremely helpful in finding and fixing problems.

[1] It is possible that some form of work-stealing could be implemented in the TensorFlow layer, but we did not investigate this

5.1.2 Lessons

Systems for Bioinformatics

The original goal for Persona was to build a system and data format that would encompass and unify most if not all tools and computations in the bioinformatics field. With such a system, a bioinformatics workload could be easily distributed and scaled out to large clusters, with most of the complex system plumbing made transparent to users. However, going from whole-genome sequencing preprocessing to protein similarity search showed that bioinformatics spans an extremely wide array of algorithms and computational patterns. To design one system that can scale and accelerate *any* bioinformatics tool or algorithm may be an exercise in futility.

As an example, consider our first attempt at solving the protein similarity search problem, which involved engineering a solution using Persona. The general idea was to directly parallelize the sequential greedy clustering algorithm, by managing a growing set of clusters directly in remote nodes. The nodes were connected in a ring, and new sequences would pass around the ring, being compared to existing cluster representatives, and added to clusters if they were similar. At the end of a trip around the ring, after being compared to all clusters, a new cluster would be created for a sequence if necessary. The benefit was that nodes in the ring could operate in parallel, with bookkeeping information sent along with each sequence as it traversed the ring.

However, there were several issues with this design. First, there was no way to properly load balance. Clusters were assigned to nodes randomly and statically, and some nodes would end up with many large (expensive computationally) clusters, while others with smaller, cheaper clusters. A single node would always become the computation bottleneck and limit the system scalability. Second, the amount of bookkeeping was large compared to the size of each sequence. The system had to maintain state along with each sequence that would track whether or not it had been compared to all clusters created from preceding sequences in a fixed order. Otherwise, similar pairs would be missed. This bookkeeping imposed a large amount of overhead, as well as increasing complexity.

To be clear, not all downstream applications of preprocessed sequencing data requires central state like clustering. Variant calling, for example, is similar in computational structure to alignment. Variant calling is the process of analyzing "piled up" aligned reads at each location in the genome, to see if there are genetic *variants*: single nucleotide mutations (known as SNPs, single nucleotide polymorphisms), or larger variants involving multiple bases. In terms of computational structure, in most cases, small windows of the genome containing probable variants can be analyzed completely in parallel. Persona would still be a very good fit for this type of computation, although we did not yet integrate any variant calling algorithms.

The lesson here is that Persona is optimal for batched, feed-forward, throughput-oriented workloads. It was not designed to implement distributed clustering methods that rely on

mutating a centrally managed state. At the end of this exercise, it was clear that another system-level design was needed. For bioinformatics, one size definitely does not fit all.

Bioinformatics Tools and the Future

Most bioinformatics tools in common use are open source and available, which the field should be commended for. Open software and methods drive further research and development. However, this can come at a cost.

As noted near the beginning of this thesis, there are many different tools in common use for bioinformatics applications. These tools are typically the product of research-oriented developers, working at universities and research institutes, whose goal is to push forward bioinformatics methods and algorithms. As such, their concern is generally to produce software that solves problems and shows their method works and is accurate. Performance is probably often a secondary goal, and most designs and implementations are engineered to run on a single commodity processor, not a cluster. Choices in algorithm design and implementation can also severely affect performance.

Consider the BWA-MEM aligner, arguably the most popular tool for alignment in preprocessing. When aligning paired sequence reads with default settings, this tool groups reads into batches of 100,000 after aligning each pair and uses a batch to estimate the insert size (the gap between the aligned pair of reads) distribution. This presumably leads to better accuracy in the aligned reads, but introduces a sequential step in the middle of an otherwise parallel task, effectively preventing thread scaling beyond 12-15 threads. Note that this was not exposed in the evaluation of Persona in Chapter 2, because single-ended alignment was being used.

Memoro also exposed instances where design decisions led to poorer performance than necessary. In the BAMTools API for sorting, holding input file data in `std::string` objects is wasteful as it is not possible to extract substrings without copying, which is unnecessary for read-only data. It should be possible to retain input file data in *one* place in memory, and use pointers into the data to do the sorting, maintaining a zero-copy architecture.

Choices like these must be weighed more carefully, given discussions about the state of bioinformatics data in Chapter 1. Performance and scalability *must* become first-class considerations when designing new bioinformatics tools. Sequencing technologies are moving forward at a brisk pace, and computational analyses risk falling behind if bioinformatics developers do not take a high-performance, scale-out approach to building applications. As this thesis states, developers should embrace commodity clusters to ensure their systems can keep up. Performance-oriented frameworks like Persona can help developers take advantage of today's scalable computing infrastructures, hiding complexity of scaling computation and overlapping I/O, allowing developers to focus on correctness and accuracy while still being able to scale. ClusterMerge has shown that even large-scale computations with central state

can be scaled effectively on commodity clusters, if the algorithm and implementation and designed carefully.

5.1.3 Conclusion

Data pressure in bioinformatics continues to grow, with sequencing machines producing more and more data with each generation. This thesis has shown that for various bioinformatics problems involved in processing this data, adopting a distributed approach with horizontal scaling using commodity cluster hardware is an effective and efficient solution. Problems such as alignment inwhole-genomee sequencing preprocessing and protein sequence similarity search can be scaled nearly linearly across clusters of 32 servers, aligning 223 million reads in 17 seconds with the Persona system or finding all significantly similar protein pairs 1400× faster than previous methods using ClusterMerge. The Memoro detailed heap profiler can also help bioinformatics developers find heap usage inefficiencies in their code, facilitating more effective use of available hardware and memory resources. The successful performance and scaling of these systems shows that adopting commodity clusters and building applications engineered for horizontal scaling is the right approach to future-proof bioinformatics tools. This will allow researchers and clinicians to take full advantage of the genomic data revolution.

Bibliography

[1] Amazon Glacier Pricing. https://aws.amazon.com/glacier/pricing/.

[2] Broad Institute GATK on Google Genomics. https://cloud.google.com/genomics/gatk.

[3] Broad Institute Picard Tools. https://broadinstitute.github.io/picard/.

[4] CRAM toolkit. https://www.ebi.ac.uk/ena/software/cram-toolkit.

[5] Electron | Build cross platform desktop apps with JavaScript, HTML, and CSS. https://electronjs.org/.

[6] GATK | Home. https://software.broadinstitute.org/gatk/.

[7] HG19 Human Genome Download. http://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/.

[8] Human Genome Project Budget. https://web.ornl.gov/sci/techresources/Human_Genome/project/budg

[9] Illumina Inc. - Sequencing and array-based solutions for genetic research. https://www.illumina.com/.

[10] Protocol Buffers. https://developers.google.com/protocol-buffers/.

[11] Quest For Orthologs. https://questfororthologs.org/.

[12] Rados - Ceph Documentation. https://docs.ceph.com/docs/giant/man/8/rados/.

[13] ZeroMQ. https://zeromq.org/.

[14] Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, Feb. 2001.

[15] Google perftools. https://github.com/gperftools/gperftools, Oct. 2017.

[16] Heaptrack: A heap memory profiler for Linux. https://github.com/KDE/heaptrack, Oct. 2017.

[17] Illumina NovaSeq. https://www.illumina.com/systems/sequencing-platforms/novaseq.html, 2017.

[18] Intel 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf, 2017.

[19] Massif. http://valgrind.org/docs/manual/ms-manual.html, 2017.

[20] Memcached: A Distributed Memory Object Caching System. https://memcached.org, 2017.

[21] Valgrind DHAT. http://valgrind.org/docs/manual/dh-manual.html, 2017.

[22] Blue Collar Bioinformatics (BcBio). https://github.com/bcbio/bcbio-nextgen, Sept. 2019.

[23] List of sequence alignment software. https://en.wikipedia.org/w/index.php?title=List_of_sequence_alignment_s July 2019. Page Version ID: 905631542.

[24] Memoro Github Repo. https://github.com/epfl-vlsc/memoro, Feb. 2020.

[25] 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. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[26] 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.

[27] A. M. Altenhoff and C. Dessimoz. Inferring Orthology and Paralogy. In M. Anisimova, editor, *Evolutionary Genomics: Statistical and Computational Methods, Volume 1*, Methods in Molecular Biology, pages 259–279. Humana Press, Totowa, NJ, 2012.

[28] A. M. Altenhoff, N. M. Glover, C. M. Train, K. Kaleb, A. Warwick Vesztrocy, D. Dylus, T. M. de Farias, K. Zile, C. Stevenson, J. Long, H. Redestig, G. H. Gonnet, and C. Dessimoz. The OMA orthology database in 2018: Retrieving evolutionary relationships among all domains of life through richer web and programmatic interfaces. *Nucleic Acids Research*, 46(D1):D477–D485, Jan. 2018.

[29] A. M. Altenhoff, R. A. Studer, M. Robinson-Rechavi, and C. Dessimoz. Resolving the Ortholog Conjecture: Orthologs Tend to Be Weakly, but Significantly, More Similar in Function than Paralogs. *PLOS Computational Biology*, 8(5):e1002514, May 2012.

[30] G. Ammons and J. R. Larus. Improving Data-Flow Analysis with Path Profiles. In *ACM SIGPLAN Notices*, volume 33, pages 72–84. ACM, 1998.

[31] B. Andreopoulos, A. An, X. Wang, and M. Schroeder. A roadmap of clustering algorithms: Finding a match for a biomedical application. *Briefings in Bioinformatics*, 10(3):297–314, May 2009.

[32] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. OPTICS: Ordering Points to Identify the Clustering Structure. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 49–60, Philadelphia, Pennsylvania, USA, 1999. ACM.

[33] A. Bankevich, S. Nurk, D. Antipov, A. A. Gurevich, M. Dvorkin, A. S. Kulikov, V. M. Lesin, S. I. Nikolenko, S. Pham, A. D. Prjibelski, A. V. Pyshkin, A. V. Sirotkin, N. Vyahhi, G. Tesler, M. A. Alekseyev, and P. A. Pevzner. SPAdes: A New Genome Assembly Algorithm and Its Applications to Single-Cell Sequencing. *Journal of Computational Biology*, 19(5):455–477, Apr. 2012.

[34] D. W. Barnett, E. K. Garrison, A. R. Quinlan, M. P. Strömberg, and G. T. Marth. BamTools: A C++ API and Toolkit for Analyzing and Managing BAM Files. *Bioinformatics*, 27(12):1691–1692, 2011.

[35] D. J. Belle and H. Singh. Genetic Factors In Drug Metabolism. *American Family Physician*, 77(11):1553–1560, June 2008.

[36] B. Blanchet. Escape Analysis for JavaTM: Theory and Practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, Nov. 2003.

[37] A. Z. Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, pages 21–29, June 1997.

[38] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[39] S. Byma, A. Dhasade, A. Altenhoff, C. Dessimoz, and J. R. Larus. Parallel and Scalable Precise Clustering for Homologous Protein Discovery. *arXiv:1908.10574 [cs]*, Aug. 2019.

[40] S. Byma and J. R. Larus. Detailed Heap Profiling. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2018, pages 1–13, Philadelphia, PA, USA, 2018. ACM.

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

[42] 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). *HiTSeq*, 2015.

[43] A. Chis, N. Mitchell, E. Schonberg, G. Sevitsky, P. O'Sullivan, T. Parsons, and J. Murphy. Patterns of Memory Inefficiency. *ECOOP Object-Oriented Programming*, pages 383–407, 2011.

[44] 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.

[45] T. . G. P. Consortium. A global reference for human genetic variation. *Nature*, 526(7571):68–74, Oct. 2015.

[46] F. Crick. Central Dogma of Molecular Biology. *Nature*, 227(5258):561–563, Aug. 1970.

[47] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct. 1974.

[48] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. Network Coding for Distributed Storage Systems. *IEEE Transactions on Information Theory*, 56(9):4539–4551, Sept. 2010.

[49] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended Analysis for Performance Understanding of Framework-based Applications. In *International Symposium on Software Testing and Analysis*, ISSTA '07, pages 118–128, London, United Kingdom, 2007. ACM.

[50] 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.

[51] R. C. Edgar. Search and clustering orders of magnitude faster than BLAST. *Bioinformatics*, 26(19):2460–2461, Oct. 2010.

[52] P. Edman et al. Method for determination of the amino acid sequence in peptides. *Acta chem. scand*, 4(7):283–293, 1950.

[53] G. G. Faust and I. M. Hall. SAMBLASTER: Fast duplicate marking and structural variant read extraction. *Bioinformatics*, page btu314, 2014.

[54] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398, Nov. 2000.

[55] 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.

[56] P. Gaudet, M. S. Livstone, S. E. Lewis, and P. D. Thomas. Phylogenetic-based propagation of functional annotations within the Gene Ontology consortium. *Briefings in Bioinformatics*, 12(5):449–462, Sept. 2011.

[57] Genomics England (NHS). The 100,000 Genome Project. https://www.genomicsengland.co.uk, 2016.

[58] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick. merAligner: A Fully Parallel Sequence Aligner. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 561–570, May 2015.

[59] S. Ghemawat and J. Dean. LevelDB. https://github.com/google/leveldb, 2017.

[60] 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.

[61] B. Gregg. Memory flame graphs. http://www.brendangregg.com/FlameGraphs/memoryflamegraphs.htm 2017.

[62] S. Guha, R. Rastogi, and K. Shim. Rock: A robust clustering algorithm for categorical attributes. *Information Systems*, 25(5):345–366, July 2000.

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

[64] J. Hamilton. *Overall Data Center Costs*.

[65] M. Hauser, C. E. Mayer, and J. Söding. kClust: Fast and sensitive clustering of large protein sequence databases. *BMC Bioinformatics*, 14(1):248, Aug. 2013.

[66] I. Health. GenomicsDB. https://github.com/Intel-HLS/GenomicsDB/wiki.

[67] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences of the United States of America*, 89(22):10915–10919, Nov. 1992.

[68] HPC Lab – OSU. *Parallel Sequence Mapping Tool*. 2016.

[69] B. Hu, L.-P. Zeng, X.-L. Yang, X.-Y. Ge, W. Zhang, B. Li, J.-Z. Xie, X.-R. Shen, Y.-Z. Zhang, N. Wang, D.-S. Luo, X.-S. Zheng, M.-N. Wang, P. Daszak, L.-F. Wang, J. Cui, and Z.-L. Shi. Discovery of a rich gene pool of bat SARS-related coronaviruses provides new insights into the origin of SARS coronavirus. *PLoS Pathogens*, 13(11), Nov. 2017.

[70] X. Huang. A contig assembly program based on sensitive detection of fragment overlaps. *Genomics*, 14(1):18–25, Sept. 1992.

[71] I. Inc. Intel® Xeon Phi™ Core Micro-architecture. https://software.intel.com/en-us/articles/intel-xeon-phi-core-micro-architecture, 22:30:14 UTC.

[72] 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.

[73] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, June 1989.

[74] B. Langmead and S. L. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature Methods*, 9(4):357–359, Apr. 2012.

[75] 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):R25, Mar. 2009.

[76] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, page 75. IEEE Computer Society, 2004.

[77] J. Leverich. Mutilate: A High-Performance Memcached Load Generator. 2014.

[78] H. Li. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv:1303.3997 [q-bio]*, Mar. 2013.

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

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

[81] 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.

[82] 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, Aug. 2009.

[83] W. Li and A. Godzik. Cd-hit: A fast program for clustering and comparing large sets of protein or nucleotide sequences. *Bioinformatics (Oxford, England)*, 22(13):1658–1659, July 2006.

[84] LLVM. Clang: A C Language Family Frontend for LLVM. 2017.

[85] Y. Lu, Q. Xie, G. Kliot, A. Geller, J. R. Larus, and A. Greenberg. Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services. *Performance Evaluation*, 68(11):1056–1071, Nov. 2011.

[86] M. A. Marra, S. J. M. Jones, C. R. Astell, R. A. Holt, A. Brooks-Wilson, Y. S. N. Butterfield, J. Khattra, J. K. Asano, S. A. Barber, S. Y. Chan, A. Cloutier, S. M. Coughlin, D. Freeman, N. Girn, O. L. Griffith, S. R. Leach, M. Mayo, H. McDonald, S. B. Montgomery, P. K. Pandoh, A. S. Petrescu, A. G. Robertson, J. E. Schein, A. Siddiqui, D. E. Smailus, J. M.

Stott, G. S. Yang, F. Plummer, A. Andonov, H. Artsob, N. Bastien, K. Bernard, T. F. Booth, D. Bowness, M. Czub, M. Drebot, L. Fernando, R. Flick, M. Garbutt, M. Gray, A. Grolla, S. Jones, H. Feldmann, A. Meyers, A. Kabani, Y. Li, S. Normand, U. Stroher, G. A. Tipples, S. Tyler, R. Vogrig, D. Ward, B. Watson, R. C. Brunham, M. Krajden, M. Petric, D. M. Skowronski, C. Upton, and R. L. Roper. The Genome Sequence of the SARS-Associated Coronavirus. *Science*, 300(5624):1399–1404, May 2003.

[87] 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.

[88] F. Meacham, D. Boffelli, J. Dhahbi, D. I. Martin, M. Singer, and L. Pachter. Identification and correction of systematic error in high-throughput sequence data. *BMC Bioinformatics*, 12:451, Nov. 2011.

[89] Microsoft. Microsoft Genomics. https://enterprise.microsoft.com/en-us/industries/health/genomics/.

[90] S. Miucin, C. Brady, and A. Fedorova. DINAMITE: A modern approach to memory performance profiling. *CoRR*, abs/1606.00396, 2016.

[91] S. D. Mooney. Progress Towards the Integration of Pharmacogenomics in Practice. *Human genetics*, 134(5):459–465, May 2015.

[92] G. Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan. 1998.

[93] C. Moretti, A. Thrasher, L. Yu, M. Olson, S. Emrich, and D. Thain. A Framework for Scalable Genome Assembly on Clusters, Clouds, and Grids. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2189–2197, Dec. 2012.

[94] 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. *Genome Biology*, 17(1):53, Mar. 2016.

[95] 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, Mar. 1970.

[96] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA. ACM.

[97] B. D. Ondov, T. J. Treangen, P. Melsted, A. B. Mallonee, N. H. Bergman, S. Koren, and A. M. Phillippy. Mash: Fast genome and metagenome distance estimation using MinHash. *Genome Biology*, 17(1):132, June 2016.

[98] R. Ostrovsky and Y. Rabani. Low Distortion Embeddings for Edit Distance. *Journal of the ACM*, 54(5), Oct. 2007.

[99] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson. The TileDB Array Data Storage Manager. *Proc. VLDB Endow.*, 10(4):349–360, Nov. 2016.

[100] C. Patterson. Homology in classical and molecular biology. *Molecular Biology and Evolution*, 5(6):603–625, July 1988.

[101] M. A. Patwary, D. Palsetia, A. Agrawal, W.-k. Liao, F. Manne, and A. Choudhary. A New Scalable Parallel DBSCAN Algorithm Using the Disjoint-set Data Structure. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 62:1–62:11, Salt Lake City, Utah, 2012. IEEE Computer Society Press.

[102] M. A. Patwary, D. Palsetia, A. Agrawal, W.-k. Liao, F. Manne, and A. Choudhary. Scalable Parallel OPTICS Data Clustering Using Graph Algorithmic Techniques. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 49:1–49:12, Denver, Colorado, 2013. ACM.

[103] W. R. Pearson. Rapid and sensitive sequence comparison with FASTP and FASTA. *Methods in Enzymology*, 183:63–98, 1990.

[104] 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.

[105] T. Printezis and R. Jones. *GCspy: An Adaptable Heap Visualisation Framework*, volume 37. ACM, 2002.

[106] A. R. Quinlan and I. M. Hall. BEDTools: A flexible suite of utilities for comparing genomic features. *Bioinformatics*, 26(6):841–842, Mar. 2010.

[107] J. Reinders. *VTune Performance Analyzer Essentials*. Intel Press, 2005.

[108] J. Sander, M. Ester, H.-P. Kriegel, and X. Xu. Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and Its Applications. *Data Mining and Knowledge Discovery*, 2(2):169–194, June 1998.

[109] F. Sanger, S. Nicklen, and A. R. Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences of the United States of America*, 74(12):5463–5467, Dec. 1977.

[110] A. Sboner, X. J. Mu, D. Greenbaum, R. K. Auerbach, and M. B. Gerstein. The real cost of sequencing: Higher than you think! *Genome Biology*, 12(8):125, Aug. 2011.

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

[112] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[113] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap Profiling for Space-efficient Java. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 104–113, New York, NY, USA, 2001. ACM.

[114] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, May 2010.

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

[116] E. L. L. Sonnhammer, T. Gabaldón, A. W. Sousa da Silva, M. Martin, M. Robinson-Rechavi, B. Boeckmann, P. D. Thomas, and C. Dessimoz. Big data and other challenges in the quest for orthologs. *Bioinformatics*, 30(21):2993–2998, Nov. 2014.

[117] M. Steinegger and J. Söding. Clustering huge protein sequence sets in linear time. *Nature Communications*, 9(1):2542, June 2018.

[118] 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):e1002195, July 2015.

[119] A. Szalkowski, C. Ledergerber, P. Krähenbühl, and C. Dessimoz. SWPS3 – fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and ×86/SSE2. *BMC Research Notes*, 1(1):107, Oct. 2008.

[120] M. Ta Michael Lee and T. E. Klein. Pharmacogenetics of warfarin: Challenges and opportunities. *Journal of human genetics*, 58(6):334–338, June 2013.

[121] 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, June 2015.

[122] The HDF Group. Hierarchical data format version 5. http://www.hdfgroup.org/HDF5, 2000.

[123] B. Tolhuis, J. Lunenberg, and H. Karten. *Ultra-Fast, Accurate and Cost-Effective NGS Read Alignment with Significant Storage Footprint Reduction.*

[124] M. Tosic. MTuner: A C/C++ memory profiler and memory leak finder for Windows, PlayStation 4, PlayStation 3, etc. https://github.com/milostosic/MTuner, Oct. 2017.

[125] R. Van Noorden, B. Maher, and R. Nuzzo. The top 100 papers. *Nature News*, 514(7524):550, Oct. 2014.

[126] Z. Wang, M. Gerstein, and M. Snyder. RNA-Seq: A revolutionary tool for transcriptomics. *Nature Reviews Genetics*, 10(1):57–63, Jan. 2009.

[127] R. M. Waterhouse, F. Tegenfeldt, J. Li, E. M. Zdobnov, and E. V. Kriventseva. OrthoDB: A hierarchical catalog of animal, fungal and bacterial orthologs. *Nucleic Acids Research*, 41(D1):D358–D365, Jan. 2013.

[128] 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 Systems Design and Implementation*, OSDI '06, pages 307–320, Seattle, Washington, 2006. USENIX Association.

[129] S. D. Whitlock. Scaling Out Bioinformatics in the Data Center. https://infoscience.epfl.ch/record/268461, 2019.

[130] L. D. Wittwer, I. Piližota, A. M. Altenhoff, and C. Dessimoz. Speeding up all-against-all protein comparisons while maintaining sensitivity by considering subsequence-level homology. *PeerJ*, 2:e607, Oct. 2014.

[131] H. Xin, D. Lee, F. Hormozdiari, S. Yedkar, O. Mutlu, and C. Alkan. Accelerating read mapping with FastHASH. *BMC Genomics*, 14(1):S13, Jan. 2013.

[132] M. Zaharia, W. J. Bolosky, K. Curtis, A. Fox, D. Patterson, S. Shenker, I. Stoica, R. M. Karp, and T. Sittler. Faster and More Accurate Sequence Alignment with SNAP. *arXiv:1111.5572 [cs, q-bio]*, Nov. 2011.

[133] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache Spark: A Unified Engine for Big Data Processing. *Communications of the ACM*, 59(11):56–65, Oct. 2016.

[134] D. R. Zerbino and E. Birney. Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research*, 18(5):821–829, Jan. 2008.

[135] J. Zhang, H. Lin, P. Balaji, and W. C. Feng. Optimizing Burrows-Wheeler Transform-Based Sequence Alignment on Multicore Architectures. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 377–384, 2013.

STUART BYMA

Systems Architect & Engineer ◇ IT & Big Data

Place du Tunnel 9 ◇ 1005 Lausanne, Switzerland

+41 · 78 · 895 ·1024 stuartbyma@gmail.com
stuart-byma byma.stuart
sbyma

Strengths

- *Building scalable, efficient systems*
- *Fast, clean, maintainable C++/Python*
- *Project ownership from concept to deployment*

EDUCATION

École Polytechnique Fédérale de Lausanne (EPFL) *2014 - present*
Ph.D in Computer Science

University of Toronto *2014*
M.A.Sc. in Computer Engineering

University of Toronto *2011*
B.A.Sc. in Computer Engineering with Honours

CORE EXPERIENCE

Very Large Scale Computing Lab, EPFL, Switzerland 2014 - present
Ph.D Student / Research Assistant *Lausanne, Switzerland*

· Successfully designed and built *ClusterMerge* [1], a parallel and scalable distributed system for finding similar protein pairs via clustering. 1400× faster than previous methods, using a 768 core cluster (`https://github.com/epfl-vlsc/clustermerge`)

· Lead designer of *Persona* [3], a high-performance framework that can scale and accelerate bioinformatics workloads. Successfully deployed on 32-node cluster, scaling sequence alignment workloads at 100% efficiency (`https://github.com/epfl-vlsc/persona`)

· Designed and implemented *Memoro* [2], a first-of-its-kind detailed heap profiler that uses static compiler instrumentation and a runtime (built into the LLVM framework) to gather detailed statistics on program heap usage behavior. Designed and implemented a cross-platform GUI to visualize collected data (`https://epfl-vlsc.github.io/memoro/`)

Microsoft Research (MSR) Summer 2015
Research Intern *Redmond WA, USA*

· Team member of the Catapult project at MSR NExT

· Successfully built and tested a prototype distributed resource management framework for provisioning FPGA resources in a datacenter, capable of managing thousands of devices and deploying multi-FPGA subsystems

University of Toronto 2012 - 2014
MASc Thesis *Toronto, Canada*

- Successfully prototyped an FPGA virtualization system that uses partial reconfiguration to manage isolated, "virtual" device partitions executing independent hardware tasks [5], allowing users to access isolated hardware acceleration in an OpenStack cloud computing system.
- Wrote and published research results in top academic conference

ADDITIONAL EXPERIENCE

École Polytechnique Fédérale de Lausanne (EPFL) 2014 - present
Teaching and Research Assistant *Lausanne, Switzerland*

- Taught courses in beginner C/C++ and Java programming, Software Engineering, Digital Systems, Global Issues in Communication
- Supervised and mentored intern teams and Masters students, several of which have been accepted to prestigious PhD programs
- Successsfully managed multi-year software development projects
- Multiple well-received presentations/talks at academic conferences

University of Toronto 2012 - 2014
Teaching and Research Assistant *Toronto, Canada*

- Taught courses on Software Engineering, C programming, Digital Systems, and Computer Architecture
- Presenting and speaking at academic conferences

Blammo Games 2012
Programmer *Toronto, Canada*

- Developed mobile games on Android and iOS platforms
- Built a generic subsystem for tracking game objectives or goals

Aversan Inc. 2011 - 2012
Software Engineer *Mississauga, Canada*

- Testbed design and maintenance for Airbus A350 air control systems
- Developed embedded software test cases for F35 Lightning II software systems

TECHNICAL STRENGTHS

General	distributed systems, big data, data analysis, data visualization, CI/CD, datacenter systems, cloud systems, Google Cloud Platform (GCP), Amazon Web Services (AWS), Microsoft Azure, networking, Agile/Scrum, infrastructure-as-code, high-performance computing, software profiling, software testing, performance analysis, scalable systems, parallel programming, containers, relational databases, NoSQL databases, S3, bioinformatics, FPGA-based hardware systems design, storage systems, technical documentation and communication, embedded systems, electronics

Languages / Tools	C, C++, Python, Java, JavaScript/NodeJS, C#, HTML, CSS, bash, Verilog HDL, git, make, Bazel, Cmake, gcc, clang, PowerShell, Xilinx/Altera FPGA design tools, gprof, VTune, ModelSim, WireShark, MySQL, LevelDB, Linux, Windows, MacOS
Frameworks / Libraries	Android, LLVM, OpenStack, TensorFlow, Protobuf, Electron, NPM, ZeroMQ, gRPC, abseil-cpp, unix sockets, Google Test, Bootstrap CSS, Hadoop, MapReduce
Training	Xilinx Partial Reconfiguration Training

ACHIEVEMENTS

Academic	University of Toronto Engineering Dean's List
	Best Student Presentation at ISMM'18
	Masters research featured on European Alliance for Innovation blog (link)
Other	Google HashCode 2016 Finalist

LANGUAGES

- English — Mother tongue
- French — Basic Proficiency (A2), progressing to B1

EXTRACURRICULAR ACTIVITIES

- Gourmet cooking for family and friends
- Ashtanga Yoga
- Current events: The Economist, NYT, WSJ, The Onion
- Alpine sports — skiing, hiking
- Electric Guitar – blues, rock

PERSONAL INFORMATION

- Civil Status: Married
- Nationality: Canadian
- Swiss Driving License B/BE
- Swiss Permit: Permis-B since Sept. 2014

PUBLICATIONS

[1] Stuart Byma, Akash Dhasade, Adrian Altenhoff, Christophe Dessimoz, and James R. Larus. Parallel and Scalable Precise Clustering for Homologous Protein Discovery [in submission]. *arXiv:1908.10574 [cs]*, August 2019.

[2] Stuart Byma and James Larus. Detailed Heap Profiling. In *ISMM*, 2018. **Awarded Best Student Presentation at ISMM'18**.

[3] Stuart Byma, Sam David Whitlock, Laura Flueratoru, Ethan Tseng, Christos Kozyrakis, Edouard Bugnion, and James Larus. Persona: A High-Performance Bioinformatics Framework. In *USENIX ATC*, 2017.

[4] Stuart Byma, Naif Tarafdar, Talia Xu, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. Expanding OpenFlow Capabilities with Virtualized Reconfigurable Hardware. In *ISFPGA*. ACM, 2015.

[5] Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. Fpgas in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. In *Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2014.

[6] Stuart Byma, Hadi Bannazadeh, Alberto Leon-Garcia, J Gregory Steffan, and Paul Chow. Virtualized Reconfigurable Hardware Resources in the SAVI Testbed. In *TRIDENTCOM*. Springer, 2014.

[7] Stuart Byma. Virtualizing FPGAs for Cloud Computing Applications. Master's thesis, University of Toronto (Canada), 2014.