University of Tennessee, Knoxville

TRACE: Tennessee Research and Creative Exchange

Masters Theses

Graduate School

12-2019

genben: A Framework for Benchmarking Genomic Data Analysis Methods on Scalable Systems

Eric Auel
University of Tennessee, eauel@vols.utk.edu

Follow this and additional works at: https://trace.tennessee.edu/utk_gradthes

Recommended Citation

Auel, Eric, "genben: A Framework for Benchmarking Genomic Data Analysis Methods on Scalable Systems. " Master's Thesis, University of Tennessee, 2019.
https://trace.tennessee.edu/utk_gradthes/5569

This Thesis is brought to you for free and open access by the Graduate School at TRACE: Tennessee Research and Creative Exchange. It has been accepted for inclusion in Masters Theses by an authorized administrator of TRACE: Tennessee Research and Creative Exchange. For more information, please contact trace@utk.edu.

To the Graduate Council:

I am submitting herewith a thesis written by Eric Auel entitled "genben: A Framework for Benchmarking Genomic Data Analysis Methods on Scalable Systems." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Engineering.

Gregory Peterson, Major Professor

We have read this thesis and recommend its acceptance:

Edmon Begoli, Charles Cao

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

genben: A Framework for Benchmarking Genomic Data Analysis Methods on Scalable Systems

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Eric J. Auel

December 2019

© by Eric J. Auel, 2019
All Rights Reserved.

This work is dedicated to my parents, who always pushed me to work hard and to give my all in life.

Acknowledgments

I would like to thank all of those whose support motivated me and kept me set on achieving my goals. I would also like to thank my committee and major professor Dr. Greg Peterson for their support and guidance with this thesis and with my academic goals.

"This material is based upon work supported by the National Science Foundation under Grant Number 1137097 and by the University of Tennessee through the Beacon Project. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or the University of Tennessee."

Abstract

With an ever-increasing number of human DNA sequencing efforts being conducted, the amount of genetic variation data available for research has grown substantially over the past few decades. This data provides scientists with the ability to study various traits of humans and other species. Several data analysis methods can be applied to this genetic variation data, such as allele counting and principal component analysis (PCA). Software libraries like *scikit-allel* can be used to easily explore these data sets, as it contains many functions that can be directly used on genetic variation data. However, trade-offs often exist when working with unique data sets and when performing analysis on various hardware environments. Additionally, many parameters can be tweaked when storing this genetic variation data, such as compression ratios, compression algorithms, and block sizes. Having the ability to quantify the performance impact of tweaking these parameters can be extremely useful for software developers, data scientists, and researchers. Algorithms that can be used on this data could also be improved in the future, so being able to compare system resource usage before and after these modifications could be extremely insightful in terms of quantifying overall improvements of new algorithms. This thesis presents *genben*, a flexible framework that can be used to benchmark various functionality involved with analyzing genetic variation data, and it additionally provides several benchmark experiments that demonstrate the ability to test different algorithm implementations, different configuration parameters, and different hardware configurations utilizing high-performance computing systems.

Table of Contents

1 Introduction **1**

2 Background **3**

2.1 The Python Performance Benchmark Suite 3

2.2 Genetic Variation Data 4

2.2.1 The Variant Call Format 4

2.3 Principal Component Analysis 5

2.3.1 The Covariance Matrix 6

2.3.2 Implementation Using Singular Value Decomposition 6

2.4 Computational Algorithms and Software Libraries 7

2.4.1 NumPy 7

2.4.2 Zarr and scikit-allel 8

2.4.3 Dask and Dask Distributed Framework 9

2.4.4 Dask-Jobqueue 10

2.5 Tying Everything Together 10

3 Methodology and Design **12**

3.1 Design of Benchmark Tool 12

3.1.1 Functionality and Features 13

3.1.2 Implementation 18

3.1.3 Expandability 19

3.2 Additional Software Scripts 21

3.2.1 Parameter Sweeping 21

3.2.2 Job Scheduling for High-Performance Computing Systems 22
3.3 Benchmark Usage Example 23

4 Experiments **25**
4.1 Experimental Environment 25
4.2 Experiment 1: Single-Node Performance 27
4.3 Experiment 2: Scale Number of Nodes 29
4.4 Experiment 3: Vary Zarr Data Chunk Size 30
4.5 Experiment 4: Vary Zarr Compression Level 32

5 Conclusion **34**
5.1 Limitations 34
5.2 Future Work 35

Bibliography **36**

Appendices **40**
A Example Data 41
B Genomic Benchmark Tool 43
B.1 Default User Configuration File 43
B.2 Example Parameter Sweep Configuration File 48
B.3 Benchmark Usage Example: Configuration File 49
C Experimental Results 50
C.1 Experiment 1: Single-Node Performance 50
C.2 Experiment 2: Scale Number of Nodes 55
C.3 Experiment 3: Vary Zarr Data Chunk Size 58
C.4 Experiment 4: Vary Zarr Compression Level 61

Vita **64**

List of Tables

4.1 System specifications for the Beacon cluster and individual nodes. 26
4.2 List of installed Python packages for running experiments. The Anaconda distribution was used for creating a dedicated environment and for installing the packages. 26
4.3 Matrix of four PCA implementations using SVD functions from the scikit-learn, SciPy, and Dask software libraries. 27
4.4 Details of Zarr data sets used for Experiment 3. 30
4.5 Details of Zarr data sets used for Experiment 4. 32
4.6 Resulting data set sizes based on compression level for Experiment 4. 33

List of Figures

2.1 Depiction of a Dask Array. A Dask array is composed of smaller NumPy arrays. 9

3.1 List of configurable settings for the *genben* tool. 14

3.2 Program file layout for *genben*. 20

A.1 Output received when running the Python Performance library. This output provides statistics on the several benchmarks performed. 41

A.2 Example contents of a VCF file. Data Source: The 1000 Genomes Project [28]. 42

B.1 Default user configuration file generated by *genben*. The configuration file contains several sections for configuring the benchmark tool. 43

B.2 Example parameter sweep file that can be used for testing multiple parameter combinations using *genben*. 48

B.3 Example of a user-edited configuration file that can be used with *genben*. This configuration is used to achieve the goals described in Section 3.3. 49

C.1 Average execution times for each PCA algorithm tested in Experiment 1. . . 50

C.2 Average CPU utilization for each PCA algorithm tested in Experiment 1. . . 51

C.3 CPU usage profile for each PCA algorithm tested in Experiment 1. The first benchmark iteration for each is shown. 52

C.4 Maximum memory usage for each PCA algorithm tested in Experiment 1. . 53

C.5 Memory usage profile for each PCA algorithm tested in Experiment 1. The first benchmark iteration for each is shown. 54

C.6 Average execution times for the allele counting algorithm (shown left) and Dask randomized PCA implementation (shown right) tested in Experiment 2. 55

C.7 Average CPU usage for the allele counting algorithm (shown left) and Dask randomized PCA implementation (shown right) tested in Experiment 2. . . . 56
C.8 Peak memory usage for the allele counting algorithm (shown left) and Dask randomized PCA implementation (shown right) tested in Experiment 2. . . . 57
C.9 Average execution times for the six functions tested in Experiment 3. 58
C.10 Average CPU usage for the six functions tested in Experiment 3. 59
C.11 Peak memory usage for the six functions tested in Experiment 3. 60
C.12 Average execution times for the six functions tested in Experiment 4. 61
C.13 Average CPU usage for the six functions tested in Experiment 4. 62
C.14 Peak memory usage for the six functions tested in Experiment 4. 63

Chapter 1

Introduction

Over the past few decades, an ever-increasing number of human DNA sequencing efforts have been conducted to grow the amount of genome data available for research and analysis. *The Human Genome Project* released the first entire-sequencing reference genome in 2003, with the project beginning in the early 1990s [10, 17, 22]. With the research efforts of *The 1000 Genomes Project* in 2008, a genetic variation data set of 1,092 individual humans from 14 populations around the world was created and made available to the public [29]. Since this initial release by this research group, the final data set includes genetic variation data for 2,504 individuals from 26 populations [12].

This vast amount of public genome data provides opportunity for researchers and data scientists to study various traits about humans. Furthermore, several data analysis methods can be applied to genetic variation data to find certain aspects or links among human DNA sequences. Principal component analysis (or PCA), for example, can be an invaluable tool for initially evaluating a data set and extracting important information with the highest variance [15]. However, human genetic variation data can potentially consume large amounts of storage when stored uncompressed, such as data available from *The 1000 Genomes Project* [28]. In addition to this, creating data structures to represent this data can require large amounts of system memory, especially in cases where everything must be loaded into memory at once. An example data structure for this scenario would be NumPy arrays, where the underlying data is stored in memory [33]. Because of this, several implementations of

PCA algorithms may (or may not) be useful when analyzing these large genetic variation data sets.

This thesis presents a tool named *genben*, which is capable of evaluating the performance of different implementations within the *scikit-allel* and *Zarr* software libraries. To support this thesis, Chapter 2 introduces necessary concepts and software that can be used by researchers for analyzing genetic variation data, and Chapter 3 discusses the *genben* framework, including implementation details and usage examples. Chapter 4 provides real-world benchmark experiments and measurement results by varying several controllable parameters regarding the computational environment and genetic variation data storage. Finally, Chapter 5 discusses some current limitations of the benchmark framework, as well as potential items for future work.

Chapter 2

Background

Development of the *genben* tool required the research in existing benchmark software. Additionally, to support the benchmarking of *scikit-allel* and *Zarr*, their supporting software libraries were studied to determine how each piece of software interacted with one another. This included the layout of and storage techniques for genetic variation data used in analysis. Operations on this genetic variation data were also explored, such as principal component analysis. Different implementation methods of these operations, made available in libraries such as *NumPy* [20], *Dask* [7], *SciPy* [16], and *Scikit-learn* [24], are discussed in detail, and this chapter concludes with how these software and algorithms are all tied together.

2.1 The Python Performance Benchmark Suite

The Python Performance Benchmark Suite, also known as *pyperformance*, is a collection of benchmarks of implementations of the Python interpreter [11]. It allows Python developers to test implementations of Python 2 and Python 3, for instance. Some benchmark groups provided by this test suite include high-level applications, float and integer arithmetic, regex, and templating performance. Under the hood, *pyperformance* heavily utilizes the pyperf software library, which allows for wall clock execution times to be measured. pyperf also allows for several iterations of a certain benchmark to be executed, providing statistics such as minimum and maximum execution times, as well as the ability to compare across different Python environments.

Users are able to customize benchmark options with the use of a configuration file. Within this configuration file, several parameters such as warm-up periods or CPU affinity can be tweaked. Similar to the underlying pyperf library, pyperformance provides measured execution times over several iterations, and the results data is stored in a JavaScript Object Notation (JSON) file. Once the JSON results file is obtained, it can be passed into the pyperf tool to view the results and other statistics. Example output of the results and statistics data can be found in Figure A.1, where the `float` benchmark is tested.

Since *pyperformance* is deemed to be an authoritative source of Python implementation benchmarking, it allows for additional benchmark groups and tests to be created and executed. This can be accomplished by creating additional modules that perform a given benchmark on a application or other component.

2.2 Genetic Variation Data

In the context of human genetics, genetic variation describes differences between any two individuals in DNA sequences. These variations from person to person often result in different characteristics or phenotypes for individuals, such as eye color, height, and vulnerabilities to diseases. With the availability of genetic variation data collected by *The 1000 Genomes Project*, for instance, researchers can study how genetic variations in DNA sequences affect certain phenotypes.

2.2.1 The Variant Call Format

One widely-used format for storing genetic variation data is the Variant Call Format, or VCF. This format was adopted by *The 1000 Genomes Project* for storing variation data [30]. VCF files are stored as text files which contain metadata, then a header, which is followed by the actual variation data. Since the VCF is essentially stored as plain-text files, they must be uncompressed to be read and subsequently used for analysis. A snippet of a VCF data file can be seen in Figure A.2.

Metadata fields in VCF files contain essential information for reading the data properly and include the file format version number, as well as optional `INFO`, `FILTER`, and `FORMAT`

fields [30]. Following the metadata lines, the header information is stored as a single line within the text file, and this line contains the names of the 8 mandatory columns for the variation data which follows it [30]. The header line is then followed by the genetic variation data, which continues until the end-of-file (EOF). Each data line represents a different position in the DNA sequence, and diploids for each sample are tab-delimitted. Each diploid for each sample contains two binary values, represented as 0's or 1's, which are separated by a vertical separator (|). These values specify whether the base for the sample differs from the reference genome. This can be seen on the last line in Figure A.2, on the right-most side.

2.3 Principal Component Analysis

Principal component analysis, or PCA, is a statistical method that is widely used today on large data sets consisting of observational data. The invention of PCA is credited to Karl Pearson, who published on PCA in 1901 [23]. In this article, Pearson describes the need to represent a system of points by a line or plane of best fit, which becomes problematic if treating a given variable as an independent variable and the other(s) as dependant [23]. This is because the multiple observed variables can be interrelated, and since the data is *observed*, this commonly results in potential error or deviation in the observational data.

According to Jolliffe, "the central idea of principal component analysis (PCA) is to reduce the dimensionality of a data set which consists of a large number of interrelated variables, while retaining as much as possible of the variation present in the data set" [15]. The dimensionality of an input data set, containing observations and variables, can be reduced by transforming the data set into a new set of variables. Although the original data set's variables could be interrelated, the resulting, transformed variables are uncorrelated. These transformed variables are called the principal components, or PCs. Additionally, these principal components can be reordered such that the first ones represent the most variation present in all of the variables in the original data set.

2.3.1 The Covariance Matrix

Since the principal components of the data represent the variables which provide the most variance (and therefore best represent the system), finding the covariance between variables is necessary to perform PCA on the data matrix. Suppose there is a data matrix $\mathbf{X}$ of size $m \times n$, where each row of $\mathbf{X}$ represents a single variable, and each column represents a single measurement (or sample) of all variables. The covariance matrix $\mathbf{C}$ for data matrix $\mathbf{X}$ can then be computed from:

$$\mathbf{C} = \mathbf{X}\mathbf{X}^T \tag{2.1}$$

The resulting covariance matrix $\mathbf{C}$ is of size $m \times m$ and therefore a square matrix. Due to the dot products of rows and columns being computed as part of the matrix multiplication of $\mathbf{X}$ and $\mathbf{X}^T$, the diagonal terms c_k of $\mathbf{C}$ represent the variance for each variable in $\mathbf{X}$. Additionally, the off-diagonal terms of $\mathbf{C}$ represent the *covariance* between each possible pair of variables for data matrix $\mathbf{X}$ [25]. In other words, the covariance calculated for each pair of variables represents the redundancy, or correlation, between that pair.

Finding high correlation between variables is crucial for PCA because one of its prominent uses is to *reduce* the dimensionality of the original data. By finding this redundancy, the original data matrix can be reduced to a smaller dimension, which is extremely useful in cases where the original data matrix has a large number of variables. Once the covariance matrix $\mathbf{C}$ has been found, the principal components can be found by using various methods. One of these techniques are discussed in Section 2.3.2.

2.3.2 Implementation Using Singular Value Decomposition

One common method of finding the principal components of a data matrix is by computing the singular value decomposition, or SVD. Since PCA is very similarly related to SVD, their names are often used interchangeably [25]. However, using SVD for PCA assumes that the principal components are orthogonal. Since the covariance matrix $\mathbf{C}$ is the product of the data matrix and its transpose, $\mathbf{C}$ is square and additionally diagonalizable. Applying SVD directly to the input data matrix $\mathbf{X}$ yields the factorization seen in Equation 2.2, where $\mathbf{U}$

is a unitary matrix, $\mathbf{\Sigma}$ is a rectangular matrix whose diagonal values are the singular values of $\mathbf{X}$, and $\mathbf{V}$ contains columns that are orthonormal eigenvectors [15].

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{2.2}$$

Substituting Equation 2.2 into Equation 2.1 results in the following simplification [15] as seen in Equation 2.6:

$$\begin{aligned} C &= \mathbf{X}\mathbf{X}^T && (2.3) \\ &= (\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T)(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T)^T && (2.4) \\ &= (\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T)(\mathbf{V}\mathbf{\Sigma}\mathbf{U}^T) && (2.5) \\ &= \mathbf{U}\mathbf{\Sigma}^2\mathbf{U}^T && (2.6) \end{aligned}$$

In other words, the covariance $\mathbf{C}$ of data matrix $\mathbf{X}$ can be computed by using the singular value decomposition of X, rather than computing the matrix product of $\mathbf{X}$ and $\mathbf{X}^T$. Then, the eigenvalues and eigenvectors of $\mathbf{C}$ can be used for PCA.

2.4 Computational Algorithms and Software Libraries

2.4.1 NumPy

NumPy is a widely-used scientific computation library for Python [20]. One of its most prominent features is the ability to store N-dimensional array data as an object, consisting of various underlying data types. Within the *NumPy* package, several functions and modules exist to perform tasks such as linear algebra, transformations, and other manipulations on the NumPy array. One example of this is *NumPy*'s built-in SVD algorithm, which utilizes the `_gesdd` LAPACK routine [3].

NumPy arrays are stored in memory once created. This allows for fast operations on data stored in these arrays; however, tradeoffs exist by storing everything in memory. On typical computer systems, available system memory is generally smaller than total persistent storage space from solid-state drives (SSDs) or hard disk drives (HDDs). Although main

memory access times are usually several magnitudes faster than persistent storage, limited memory presents a bottleneck in terms of maximum storable array sizes.

2.4.2 Zarr and scikit-allel

Zarr improves on the capabilities of the NumPy N-dimensional array by providing chunked, compressed N-dimensional arrays [35]. Zarr arrays can also be stored on persistent storage devices, allowing for larger-than-memory computations. This is achievable by only moving necessary chunks of data to memory as needed. Another benefit of *Zarr* is that it provides an opportunity to store larger-than-memory data that can be placed on a shared file system, such as the Network File System (NFS) or Lustre. As a result, data can now be shared among multiple nodes, allowing for parallel computation on N-dimensional data in Python. This is possible when *Zarr* is used along with a software framework such as *Dask*, which is discussed later.

To support the analysis of genetic variation data, the *scikit-allel* software library provides several utility functions that can simplify the task of working with this data [19]. *scikit-allel* can utilize Zarr arrays to load and represent commonly-large genetic variation data sets, typically spanning millions of rows when representing an entire human genome. Since genetic variation data for humans can be represented by 0's and 1's, which represent whether a particular sequence varies from a reference sequence, compression can greatly reduce the overall size of the array. This is because the genetic variation rate between any two humans is around roughly 0.1 percent [34], meaning that around 99.9 percent of genetic variation data can be represented as 0's. Consequently, long sequences of 0's can be compressed to a much smaller size, greatly reducing the amount of memory or persistent storage needed to represent very large arrays of genetic variation data.

As mentioned earlier, *scikit-allel* provides several helper functions for managing and analyzing genetic variation data. Two notable functions provided by this library are (1) simple aggregation functions and (2) principal component analysis. These two features can be particularly useful for researchers and data scientists who are initially trying to understand samples of genetic variation data, including relationships between base pairs and individual samples.

Figure 2.1: Depiction of a Dask Array. A Dask array is composed of smaller NumPy arrays.

2.4.3 Dask and Dask Distributed Framework

Dask is a Python library that enables parallel computing by offering parallel, chunked data structures [7]. The Dask Array object, for example, is similar to the NumPy array and offers a subset of functionality. Dask arrays allow for parallel operation on the data by splitting up the large array into several smaller NumPy arrays, as visualized in Figure 2.1. Block sizes for each smaller array can be consistent, or the block layout could alternatively be heterogeneous. In addition to the Dask array object, *Dask*'s `Array` API provides several blocked algorithms that can be used on Dask arrays. These algorithms include functions that are also in the *NumPy* API, as well as several other linear algebra functions. One notable function is `linalg.svd(...)`, which can compute the singular value decomposition of a matrix contained within a Dask array. Unlike *NumPy*'s implementation of SVD, however, the *Dask* implementation can utilize parallelization and even run on distributed systems with the assistance of the *Dask.distributed* library.

Dask.distributed is a software library (separate from *Dask*) that allows for Python programs to leverage distributed computing systems which contain multiple nodes [6]. Like *Dask*, *Dask.distributed* provides data sharing and execution of parallel blocked algorithms among nodes. In the distributed environment, a designated scheduler node is used to

command one or more worker nodes connected to it. Nodes can communicate and share data over the traditional TCP/IP stack, via Ethernet or Infiniband, for instance. Essentially, *Dask.distributed* provides two executables, `dask-scheduler` and `dask-worker`, which can be run on the scheduler and worker nodes, respectively.

2.4.4 Dask-Jobqueue

As mentioned earlier, *Dask* can leverage a distributed cluster, containing multiple nodes, by using the *Dask.distributed* library. However, many high-performance clusters (HPCs) require jobs to be scheduled in order to use the cluster. In this situation, the *Dask-Jobqueue* library allows for deployment of the scheduler and worker programs on HPCs [8]. Internally, *Dask-Jobqueue* works by generating batch submission scripts based on the parameters configured using its Python API. Users can request the number of worker nodes to spin up, and the tool can generate and invoke the job scheduler to spin up *Dask* worker nodes.

2.5 Tying Everything Together

Dask, *Zarr*, and *scikit-allel* together provide exciting advancements for researchers to study large-scale genetic variation data, where operations can scale from lower-powered desktops to high-performance computing systems. However, these three software libraries are relatively new and still in active development. Because of this, software bugs are still plausible to exist in their code bases, and there may be additional performance improvements that could result in more efficient operation or faster execution times.

Having a tool that could directly measure resource usage and performance could arguably be extremely beneficial to the software developers contributing to these open-source projects. For example, *scikit-allel* currently uses in-memory SVD algorithms for computing the principal components of genetic variation data, and these algorithms come from the *SciPy* and *scikit-learn* software libraries. However, both of these algorithms require data to be stored in a NumPy array, which consequently requires all data to be loaded into memory at once. Another drawback of these two algorithms is that they can only leverage a single node, and therefore do not scale well to high-performance clusters containing multiple nodes.

Dask, on the other hand, provides alternate implementations for these two SVD functions, eliminating the requirement for all data to be stored in memory at once. Additionally, the SVD functions can be performed on a distributed cluster, potentially speeding up the process of applying SVD to a large data set. Having a benchmark tool capable of testing different versions or different implementations of algorithms within *Zarr* or *scikit-allel* could be very useful for evaluating before-and-after comparisons, allowing for further development and improvements to these software libraries.

Chapter 3

Methodology and Design

This chapter introduces a benchmark tool named *genben* [4], which can be used to evaluate the performance of functionality provided by the *scikit-allel* [19] and *Zarr* [35] software libraries. Several features of *genben* are discussed, including the ability to test different implementations and algorithms within *scikit-allel* and *Zarr*. This can be extremely useful for the developers of these open-source libraries because it provides them with the ability to test and compare the performance of different implementations, both within different computational environments and different parameter configurations.

3.1 Design of Benchmark Tool

As an initial effort to explore the scaling properties of the *scikit-allel* and Zarr libraries, the *genben* tool was created. This research was conducted as a joint collaboration between Oak Ridge National Laboratory (ORNL), Oxford University, and the University of Tennessee [4]. *genben* is a benchmark software framework, written in Python version 3.6, that can be used to test the scaling properties of different functions on various hardware platforms and configurations. Additionally, different versions or implementations of algorithms can be tested to determine how they compare in terms of execution time and resource usage.

3.1.1 Functionality and Features

Out of the box, *genben* allows for different aspects of *scikit-allel* to be benchmarked, including (1) aggregation methods such as allele counting, (2) PCA algorithms, and (3) the VCF to Zarr data conversion process. The benchmark tool reads its settings from a configuration file which can be edited by the user. The configuration file is stored in an INI file format [14], and settings are broken into logical sections. This configuration file is important because it allows the user to quickly change settings when running various benchmarks, and it exposes several configuration parameters of the *scikit-allel* and *Zarr* packages. Some notable benchmarking settings which can be configured by the user can be seen in Figure 3.1, and an example of a user configuration file can be seen in Figure B.1.

FTP File Downloading

Settings related to downloading data from a FTP server can be found under the `[ftp]` section of the configuration file. If a user desires to download data from a remote server, they can enable the FTP download module by setting the `enabled` parameter to `True`. If data has already been stored locally and remote download is not necessary, this parameter can be set to `False`. If the FTP download module is enabled, it will proceed to connect to the remote FTP server specified within the `server` parameter. Although most public FTP servers allow for anonymous logon (i.e. they do not require a username or password for access), servers requiring a username or password can have these values set using the `username` and `password` fields, respectively. If these two fields are left blank, then *genben* will attempt to log in anonymously.

After login is successful, the program will proceed to navigate to the location specified in the `directory` field. The `files` parameter allows the user to specify a single file or multiple files to download within the working directory. If multiple, specific files are desired, each file name can be listed, each separated using a delimiter character (e.g. a comma ","). To avoid conflicts within multiple file names, the `delimiter` character can even be modified so that multiple file names can be separated with a different delimiter character. If the user wishes to download **all** files within a directory, the `files` parameter can instead be set to

- Genetic variation data downloading/processing from FTP server
- VCF to Zarr data conversion
  - Data chunk length and width
  - Compression algorithm
  - Compression level
- Dask.distributed scheduler connection
- Benchmark settings
  - Number of runs to perform
  - Data input (either VCF or Zarr)
  - Data set to use for benchmarks
  - Number of variants and samples to use (from the data set)
  - Enable simple aggregations benchmarks
  - Enable PCA benchmarks
  - Python data array type (Dask or NumPy array)
- Benchmark output options
  - Output to CSV file
  - Output to InfluxDB server

Figure 3.1: List of configurable settings for the *genben* tool.

an asterisk character "*". If the FTP download module sees this value, it will automatically obtain a listing of all files within the working directory and download them. This process is recursive, meaning that files within subdirectories will be downloaded as well.

VCF to Zarr Conversion

The VCF to Zarr conversion module is responsible for converting any VCF files that were either downloaded using the FTP module or that were already stored locally. This module can be configured within the `[vcf_to_zarr]` section of the user configuration file, and it can subsequently be enabled or disabled using the `enabled` parameter. Since *Zarr* stores data to the file system in chunks, the desired Zarr chunk sizes can be specified using the `chunk_length` and `chunk_width` parameters. Chunk length corresponds to the number of variants are stored per chunk, and similarly chunk width determines how many samples per chunk are stored. A value of "default" can be set for both of these, meaning that the default value from the `scikit-allel` library will be used. At the time of writing, the default values for chunk length and width are $2^{16} = 65,536$ and $2^6 = 64$, respectively.

One major benefit of storing genetic variation data in the Zarr format (versus VCF) is that the data can be compressed, greatly reducing the resulting file size. When creating the Zarr data store from a VCF file, *scikit-allel* exposes several parameters to the underlying *Zarr* function calls. Similarly, *genben* exposes these compression parameters, which can be accessed and modified by the configuration file. The default compressor used by *scikit-allel* is the Blosc compression library [1], which can utilize several compression algorithms such as *zstd* [9], *lz4* [18], *zlib* [36], and *snappy* [26]. This wide selection of algorithms allow for trade-offs among compression speed, decompression speed, and compression ratios. In the context of using *genben*, the compression algorithm can be specified using the `blosc_compression_algorithm` parameter. Acceptable values for this parameter are: "zstd", "blosclz", "lz4hc", "zlib", and "snappy". In addition to specifying the algorithm, the compression level can also be specified using the `blosc_compression_level` field. This parameter expects a non-negative integer value between 0 and 9. A value of `0` corresponds to no compression, while a value of `9` represents highest compression.

Dask Scheduler Connection

To support operation on HPCs with multiple nodes, the *Dask.distributed* library allows for programs to connect to a Dask scheduler process which is responsible for distributing work to worker nodes. Since the genben framework acts as the driver program and can leverage this functionality, a `[dask]` configuration section has been placed in the user configuration file. Within this section, connection to a `Dask` scheduler can be enabled or disabled using the `enabled` parameter. If this field is set to "True", the benchmark tool will attempt to connect to a *Dask* scheduler node, defined by `scheduler_address` and `scheduler_port`, upon initialization. If this module is disabled, however, all computations will be performed on the same node which is executing *genben*.

Benchmark Parameters

Since *genben* is indeed a benchmarking tool, the `[benchmark]` section of the configuration file is the largest because it contains configurable parameters related to the actual benchmark process. Notable parameters include how many iterations to perform a given benchmark, which data set and data type to use, and which benchmark groups should be performed.

Users can specify how many iterations of a benchmark to perform by setting the `benchmark_number_runs` field. Valid values for this field are any non-negative integer value. Setting to a value of "1" will result in each operation only being performed once, whereas setting the field to a value of $n \in \mathbb{Z}^+$ where $n > 1$ will result in each operation being repeated n times. In this case, it is important to note that each operation will not be repeated contiguously. Instead, each unique operation will be performed in order, then *genben* will cycle back to the beginning and repeat each operation again. This is because some operations depend on the result of a previous operation before it can execute.

Several parameters exist which can be used for tweaking the input genetic variation data set. Firstly, the `benchmark_dataset` parameter specifies the file name of the input file, which can either be a VCF or Zarr data set. This format should be specified in the `benchmark_data_input`, which accepts the value of either "vcf" or "zarr". If "vcf" is specified here, the input VCF data set will be converted to the Zarr format so that it can be used

directly by *scikit-allel*. This process will also be benchmarked and therefore repeated if the number of benchmark iterations is set to a value greater than 1. If the input data set is larger than desired, the number of variants and samples can be limited by setting the `benchmark_num_variants` and `benchmark_num_samples` fields, respectively. If a value of "-1" is passed to either of these parameters, then all variants and/or samples will be used from the input.

Once the genetic variation data has been loaded, *genben* calls a *scikit-allel* function to represent this data as a genotype array. This array is then passed into different benchmark operations such as allele counting and PCA. Under the hood, this genotype array can be represented by either a in-memory NumPy array, or as a chunked, lazy Dask array. This can be configured by the use of the `genotype_array_type` parameter.

Individual benchmark groups, such as simple aggregations or PCA, can be enabled or disabled using the `benchmark_aggregations` and `benchmark_pca` fields. If both of these are set to "False", then essentially the only operations that will be benchmarked are the loading of the genetic variation data sets and the VCF to Zarr conversion process, if this is enabled.

Results Data Output

genben supports the output of benchmark results to a text (comma-separated) file or to an InfluxDB server [13]. As such, `[output.csv]` and `[output.influxdb]` sections exist in the configuration file to tweak results output. Both output modules have an `enabled` parameter to enable or disable output using that module. For CSV output, the column delimiter can be modified using the `delimiter` field. The default comma delimiter could be modified to the pipe character " | ", for example, to create a pipe-separated file (PSV). As for InfluxDB output configuration, the InfluxDB server connection can be specified using the `host`, and `port` fields, and the database to write results to can be specified by the `database_name` field. If the specified database requires authentication for write permissions, then the `username` and `password` fields can be used.

3.1.2 Implementation

The benchmark tool was developed using Python, supporting Python versions 2.7, 3.5, 3.6, and 3.7. The decision to support multiple versions of Python was made so that different versions of underlying packages, such as *scikit-allel* and *Zarr*, could potentially be tested for performance differences on different versions of Python. The primary reason for using Python was because the software libraries to be benchmarked were also developed using Python. By using the same programming language, the libraries could be imported and utilized directly, without the need for additional complexity or library wrappers. The benchmark tool was also constructed as a Python package, so that it could be installed using *pip*, the package installer for Python [32]. This allows testers and software developers to easily install *genben* so that it can be called directly from the command line, using the `genben` command. Additionally, installation using *pip* allows for any missing package dependencies to be installed automatically.

The implementation of the benchmark tool was split into logical sections or files that corresponded to their respective tasks, which is visualized in Figure 3.2. The majority of functionality was implemented using the files: `cli.py`, `config.py`, `dask_utils.py`, `data_service.py`, and `core.py`. The command-line interface (CLI) for the program is implemented in `cli.py`. This file acts as the main entry-point for the program, and it handles initial steps such as argument parsing from the user, determining the appropriate mode to operate in. It also utilizes all of the other files mentioned above, based on the mode of operation and arguments or settings provided by the user. Once the benchmark tool is initialized and the specified configuration file is provided by the CLI arguments, the configuration file name is passed to `config.py`, which is responsible for parsing all user configuration parameters within the file. Essentially, this converts the text data in the file into usable variables within Python, so that the benchmark process can be tuned to fit the needs of the user. An example of a user configuration file can be seen in Figure B.1. If the user opted to connect to a Dask scheduler process (using the configuration file), then the `dask_utils.py` module will be imported and used to connect to the Dask scheduler

instance [7]. This binding process results in future computations on Dask arrays to be performed using the Dask.distributed framework, instead of running on a single node [7, 6].

The data service module, implemented in `data_service.py`, serves several purposes related to the data files used during benchmarking. Functions exist to create the file system structure necessary for storing data files, as well as deleting data files that are no longer needed. The module can also download compressed or uncompressed VCF files, as well as Zarr data stores, from a remote FTP server. Once files are either downloaded from FTP or placed in the correct folder by the user, the data service module can automatically process and sort these files, additionally converting VCF files to the Zarr format if specified by the user. Finally, the data service module is responsible for calling the necessary *scikit-allel* and *Zarr* functions to load the genetic variation data from storage so that it can be used for benchmarking. Data can be loaded from a single source, or concatenated from multiple data stores. This is extremely useful in events where variation data from multiple chromosomes is needed for a single test.

The last file, `core.py`, contains the actual benchmarking implementation: it is responsible for calling the simple aggregation and PCA functions to be benchmarked. It also is responsible for measuring the wall-clock execution time for each task, as well as storing the benchmark results for future analysis. Benchmark results include the Coordinated Universal Time (UTC) start time, the run number (when performing multiple iterations), the operation name, and the measured execution time.

3.1.3 Expandability

The *genben* tool was designed such that it could be expanded to include additional benchmarks for future work. This was accomplished by creating a portable class named `BenchmarkProfiler`, located in the `core.py` file. This class is responsible for keeping track of when a given benchmark is to begin and end, organizing the benchmark results, and recording the data. Essentially, this class exposes two critical functions: `start_benchmark(...)` and `end_benchmark()`. These two functions abstract away the details of measuring and recording benchmark information. Since these two functions (respectively) are placed before and after the function call(s) to be benchmarked by the framework, a developer can use this framework

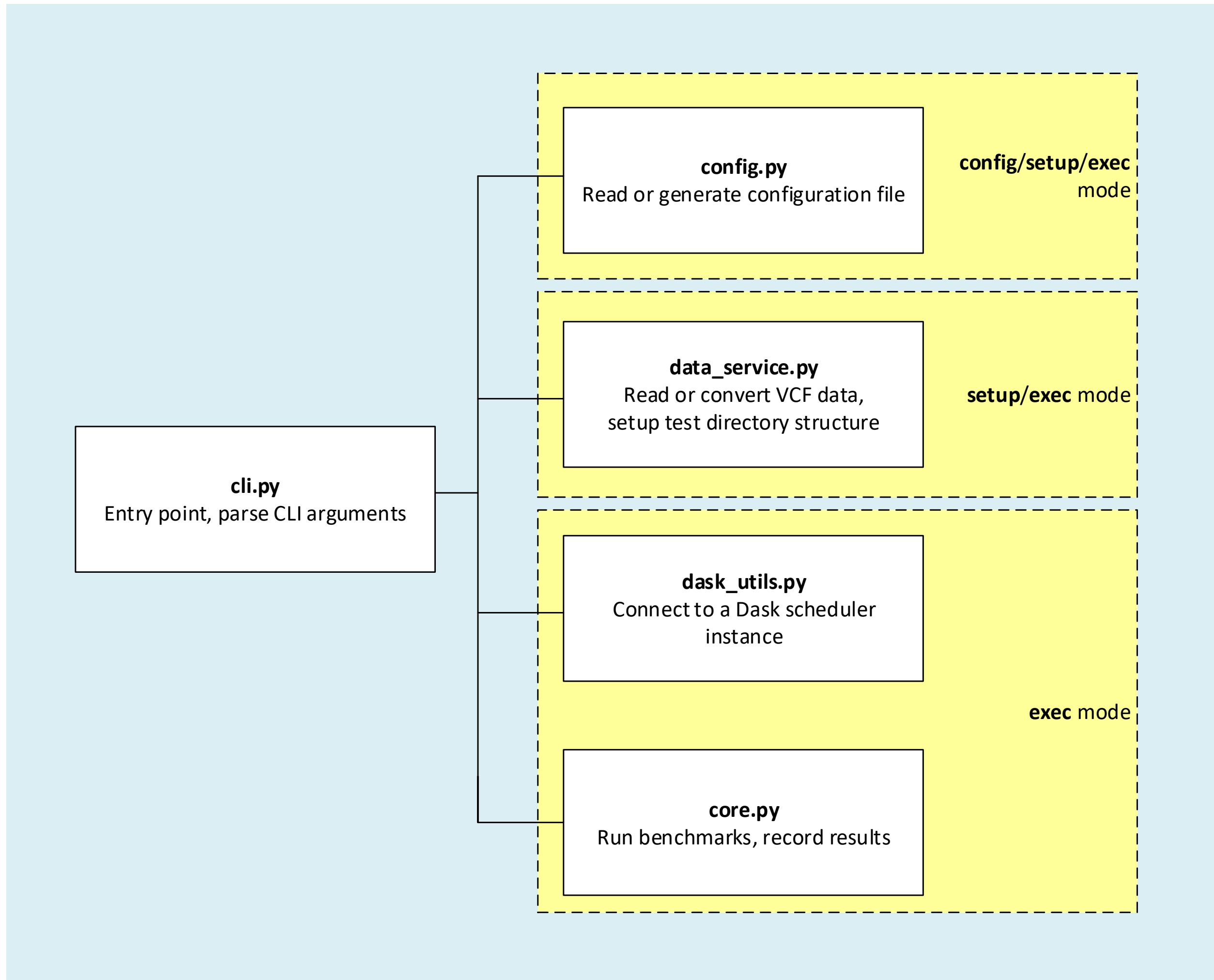

Figure 3.2: Program file layout for *genben*.

to benchmark other functionality in the *scikit-allel* or *Zarr* libraries. Additionally, any other scientific library or package could be tested by the *genben* software, if desired.

3.2 Additional Software Scripts

To assist with usage of the *genben* framework, additional scripts and tools have been created to both help obtain and visualize data from various systems and use case scenarios.

3.2.1 Parameter Sweeping

The base benchmark tool framework was designed to read from a user configuration file, and based on those settings, perform a benchmark. However, in some situations it may be desirable to test a range of values for a parameter or multiple parameters. For example, a user may wish to conduct tests on several Zarr benchmark data sets, each with differing compression ratios or chunk sizes. Or, the user may desire to perform a parameter sweep on the number of variants or samples in a data set to determine how it affects computation time for PCA.

To provide parameter sweeping functionality, an additional helper script (named `parameter-sweeper.py`) was created. Instead of passing a single user configuration file into the list of arguments to the genben program, *two* configuration files are provided: (1) a base configuration file and (2) a parameter sweep configuration file. The base configuration file is essentially the same as before. However, the parameter sweep configuration file contains all of the parameters that the user desires to sweep through. This file is mostly the same format as the base configuration file, except that parameters can be a comma-separated list of desired values to test with. Whenever two or more parameters contain multiple values, all possible combinations of parameters are tested. For example, if the user specifies two data sets (e.g., chr10 and chr11 data sets) as well as four different values for the sample size, then eight tests will be executed in total. An example parameter sweep configuration file for this scenario can be seen in Figure B.2.

3.2.2 Job Scheduling for High-Performance Computing Systems

In order to test highly-parallelized code using the benchmark tool, it is important that *genben* can be used on high performance computing (HPC) systems, such as supercomputers. However, utilizing the scheduler-worker structure of Dask's distributed framework becomes a little more difficult because many HPCs utilize a job scheduling system to share nodes and resources among its users. Although the Dask-Jobqueue project can already be used to create multiple Dask worker nodes on a HPC system, it could not immediately be used for benchmarking. This is because the intent of Dask-Jobqueue is to have worker processes spin up and shut down as resources become available. In the context of benchmarking, however, it is necessary to have all worker processes online before starting the benchmark, in order to have consistent, repeatable results. Since HPCs are generally a shared resource, it is not always possible to directly create a Dask worker process on each of the nodes before executing the benchmark tool. Because of this, a helper script was created that can be submitted to a Portable Batch System (PBS) or Moab Cluster job scheduler.

This job scheduling script was created independently from the *genben* tool because many HPC systems utilize different job scheduler systems, and some HPCs utilizing the same job scheduler even require different parameters when submitting jobs. As a result, the job submission script likely requires manual editing by the user. Instead of attempting to accommodate the use of various, unique job scheduler systems, a separate script was created that can take advantage of the Open MPI library that is commonly found on HPCs [21]. Although the Dask distributed framework does not use MPI for parallelization, the main reason for using MPI is because a benchmark job requiring multiple nodes can now be submitted as a *single* job. This is crucial for benchmarking because this ensures that all nodes will be online immediately upon the job starting, and it ensures that they will remain online for the entire duration of the benchmark, resulting in repeatable, more accurate benchmark results.

To take advantage of multiple nodes using the Open MPI library, the job scheduling script executes the `mpirun` command whenever the job is initiated. Consequently, this executes a secondary script on each of the requested nodes. This secondary script is ultimately

responsible for using the assigned rank ID (from MPI) to determine the role of that particular node. The first rank is then assigned as the Dask scheduler node, and the second rank is responsible for running the *genben* framework. Therefore, all remaining ranks are assigned to be Dask worker nodes. The secondary script can optionally create a new telegraf process, which allows for detailed metric collection for each individual node.

3.3 Benchmark Usage Example

The first step of running a benchmark using the *genben* tool is to generate and modify the user configuration file. To generate a default configuration file, the following command can be executed:

```
genben config -output_config <filename>
```

where `<filename>` is replaced with the desired file name for the newly-created configuration file. After running this command, a new configuration file will then be created by the program, similar to the one seen in Figure B.1. This file can be edited to specify where to download genetic variation data, whether to connect to a distributed Dask cluster, and various other benchmark parameters. Additionally, the simple aggregations and/or the PCA benchmarks can be enabled or disabled. For this example of using *genben*, suppose a user wants to perform the following:

1. Download genetic variation data from a public, remote FTP server
2. Convert genetic variation data to Zarr format, using default settings
3. Perform benchmarks on a pre-existing Dask cluster
4. Perform 10 runs of each benchmark
5. Run the simple aggregations and PCA benchmark groups
6. Output results to a CSV file

To achieve these goals, the user could edit the configuration file to reflect the one seen in Figure B.3. After the configuration file has been generated and customized, the setup process can be invoked by the following command:

```
genben setup -config_file <filename>
```

where `<filename>` is replaced with the file name of the configuration file created earlier. Running this command will cause VCF or Zarr data files to be downloaded from a FTP server (if enabled) and stored in the `./data/input/download` directory. Additionally, any files located in the `./data/input` directory, included any downloaded files, will be extracted and organized. Finally, any VCF-formatted files will be converted to the Zarr data format.

Once the setup process is complete, the `exec` command can be used to begin the benchmark process:

```
genben exec -config_file <filename>
```

where `<filename>` is replaced with the same value as before. *genben* will then begin reading genetic variation data from the file system and execute any enabled benchmarks. It will repeat each benchmark operation over 10 iterations, as defined in the configuration file, and store the results to a CSV file. Once this process is complete, the results data can be analyzed or combined with *Telegraf* [27] metrics data.

Chapter 4

Experiments

To support the thesis, several experiments were conducted to demonstrate potential use case scenarios of the *genben* tool. These experiments demonstrate the ability to run benchmarks in differing computational environments, such as a single node and multi-node high-performance computing systems. They also demonstrate the ability to measure how parameter changes can affect overall performance and resource usage on a system. The first experiment acts as a reference test to compare single-node performance, while the remaining experiments focus on utilizing parallelized operations that can leverage multiple nodes in a distributed system. The findings from these experiments provide insight on the amount and type of metrics that can be collected with the use of *genben* [4] and *Telegraf* [27]. Several charts are provided to visualize the data collected from each experiment, highlighting average execution times, average CPU and memory usage, and usage over time.

4.1 Experimental Environment

To conduct these experiments, the Beacon cluster at The National Institute for Computational Sciences was used [5]. Specifications on the Beacon cluster can be found in Table 4.1. Within the cluster, nodes share a Lustre file system for scratch storage, and they additionally share a NFS space for user home directories.

In order to run the genben framework, as well as the Dask scheduler and worker processes, a Python interpreter was installed by the use of the Anaconda distribution [2]. Since the

Table 4.1: System specifications for the Beacon cluster and individual nodes.

System Details	
No. Compute Nodes	48
Interconnect Type	FDR InfiniBand
Shared Storage Type	NFS, Lustre
Node Details	
No. Processors	2
Processor Type	Intel Xeon E5-2670
Memory Cap.	256 GB

Table 4.2: List of installed Python packages for running experiments. The Anaconda distribution was used for creating a dedicated environment and for installing the packages.

Software Package	Installed Version
Conda	4.6.14
Python	3.6.8
dask	1.2.2
distributed	1.25.2
genben	0.1.0
numcodecs	0.6.4
numpy	1.16.3
pandas	0.24.2
scikit-allel	1.2.1
scikit-learn	0.21.1
zarr	2.3.1

Beacon cluster is a shared resource, the Anaconda distribution and software environment was installed to the user home directory, located on the NFS share. Installing this to a shared location was critical so that each active node on the cluster could have access to the Python interpreter, as well as any additional Python software packages. After installation of the Anaconda distribution, a new Conda environment was created for package installation, and all software dependencies necessary for running the experiments were installed. For reference, software versions for installed packages can be seen in Table 4.2. The telegraf binary was also installed to the user home directory, and a configuration file was created (1) to enable collection of CPU, memory, and network activity and (2) to send results to a remote InfluxDB server.

For all four experiments, genetic variation data from *The 1000 Genomes Project* [12], chromosome number 22 (chr22) was stored in the Zarr format on the Lustre scratch file system. The compression level and block size varied for each experiment.

4.2 Experiment 1: Single-Node Performance

The first experiment was performed to demonstrate the ability to benchmark different algorithm implementations. This was done by installing the latest version of *scikit-allel* and testing this version against a modification to the library's PCA implementation. At the time of writing, *scikit-allel* by default uses in-memory implementations for PCA algorithms, calling SVD algorithms from the *SciPy* [16] and *scikit-learn* [24] libraries. Essentially, this experiment involves comparing these two currently-used SVD functions against two alternate implementations offered by *Dask* [7].

Table 4.3: Matrix of four PCA implementations using SVD functions from the scikit-learn, SciPy, and Dask software libraries.

	Full SVD	**Randomized SVD**
Single-Node Functions	scipy.linalg.svd(...)	sklearn.utils.extmath.randomized_svd(...)
Parallelizable Functions	dask.array.linalg.svd(...)	dask.array.linalg.svd_compressed(...)

In addition to demonstrating different algorithm implementations, this experiment was also intended to be a reference test in regards to single-node performance. Because of this, the Dask scheduler and worker processes were not used. Consequently, all four PCA algorithms were tested on a single node. These PCA implementations fit into two different categories: full SVD and randomized SVD. The four underlying SVD implementations for PCA were called by the *scikit-allel* library and are summarized in Table 4.3. For this test, genetic variation data from *The 1000 Genomes Project* [12], chromosome number 22, was stored in the Zarr format with no compression and a block size of 4096 variants by 4096 samples. It was decided to use no compression on the data because the genetic variation data ultimately used by the two in-memory SVD functions require the data to be stored (uncompressed) in memory, using a NumPy array. Essentially, disabling compression resulted in one less variable in this reference test, allowing all four SVD algorithms to operate on uncompressed data.

To generate the Zarr data set from the VCF file, the VCF to Zarr conversion module within *genben* was used, and the `chunk_length` and `chunk_width` parameters within the `[vcf_to_zarr]` section of the configuration file were both set with a value of "4096". Additionally, `blosc_compression_level` was set with a value of "0" to disable compression on the resulting data set. Due to memory constraints with the two in-memory SVD algorithms, the benchmark process was limited to only use the first 10,000 variants of the Zarr data set. This was done by setting the `benchmark_num_variants` parameter with a value of "10000" in the user configuration file. Additionally, each PCA implementation was repeated over 5 iterations to account for a warm-up period. For this, the `benchmark_number_runs` parameter was set with a value of "5". The final configuration parameter modified was `genotype_array_type`, which specifies how to represent the genetic variation data stored on disk. When testing the two Dask functions, this parameter was set with a value of "1", telling the benchmark tool to use a Dask array. When testing the other two SVD functions from *SciPy* and *scikit-learn*, this parameter was changed to "0" to use a NumPy array to store all data in memory.

In terms of experiment results, which can be found in Section C.1, the distributed Dask PCA algorithm takes considerably longer to execute, as seen in Figure C.1. In addition

to this, this algorithm also uses considerably more memory, as opposed to the other three algorithms, seen in Figures C.4 and C.5. However, the Dask library's randomized PCA algorithm only takes a fraction of time to execute, especially compared to the randomized PCA implementation found in the Scikit-learn library. In terms of the average CPU usage and CPU usage profile (seen in Figures C.2 and C.3, respectively), the randomized in-memory PCA algorithm appears to only use a single thread for most of its run time duration. This is because the CPU usage percentage remains around 3 to 4 percent, and a single Beacon node has 16 cores total with hyper-threading, resulting in 32 virtual cores.

4.3 Experiment 2: Scale Number of Nodes

The goal of the second experiment was to demonstrate the scaling properties of two parallelized functions available in the Dask library. The first function involves the counting of alleles, and the second function is a randomized PCA implementation. Unlike the first experiment, this one utilizes multiple nodes within a high-performance cluster. With N nodes labeled as $[0, 1, ..., N-1]$, node 0 was designated as the Dask scheduler node, node 1 was designated as the driver node, and the remaining $N-2$ nodes were assigned as Dask worker nodes who connect to the Dask scheduler. The driver node was responsible for running the genben tool, and the worker nodes were responsible for performing computations on the genetic variation data (represented as a Dask array). For this experiment, the number of worker nodes was varied for each test conducted. The number of worker nodes tested were: 1 worker, 2 workers, 4 workers, and 8 workers. The goal of this was to see how scaling the number of worker nodes in a cluster affected execution time for allele counting and PCA, as well as measure resource usage on each of the worker nodes for each scenario.

Several job submission scripts were created for this experiment, along with a single, shared benchmark configuration file. The Zarr data set created for Experiment 1 was also used for this experiment, meaning that the data set was not compressed. However, all variants and samples from the data set were used, unlike the previous experiment. This is because the Dask arrays could load the required data for computations as needed, instead of loading the entire data set into memory. To use the entire data set, the `benchmark_num_variants` and

Table 4.4: Details of Zarr data sets used for Experiment 3.

Test Number	**Chromosome**	**Chunk Length (No. Variants)**	**Chunk Width (No. Samples)**	**Compression Level [0-9]**
1	chr22	512	512	0
2		1024	1024	
3		4096	4096	
4		16384	16384	

`benchmark_num_samples` parameters were set with a value of "-1", telling *genben* to use all available data.

All results for this experiment can be found in Section C.2. Unlike the first experiment, this one focused on running two distributed algorithms and scaling the number of Dask worker nodes. As seen in Figure C.6, the execution time for allele counting and randomized PCA decreases as the number of worker nodes increase, as would likely be expected. As for CPU and memory usage in Figures C.7 and C.8, both of these decrease as the number of nodes is scaled up. This is likely because work is being spread across all nodes due to parallelization. As a result, each node spends more time reading the genetic variation data from Lustre storage.

4.4 Experiment 3: Vary Zarr Data Chunk Size

The last two experiments explore two properties that can be specified when creating a Zarr data store: the data chunk size and the compression level. This experiment focuses on the former property. In order to test how the Zarr data chunk size affects computation time and resource usage, four Zarr data stores containing identical data (from chromosome 22) were created, each with different chunk sizes. The chunk sizes used for this experiment are described in Table 4.4, and the primary goal of this experiment is to see whether computations can be performed quicker by having larger chunk sizes.

Three parallelizable operations were tested when varying the chunk size in this experiment:

1. Genotype counting

2. Allele counting

3. Randomized PCA

For genotype counting, *scikit-allel* exposes two different functions:

1. Count number of heterozyous genotypes

2. Count number of homozygous genotypes

Both of these genotype count functions allow the user to specify along which axis to perform the count, meaning that the result can be a count for each variant, or a count for each sample. For this experiment, genotype counts were performed for each variant and each sample, for both the heterozygous and homozygous count functions, resulting in four function calls being benchmarked.

All of the above-mentioned operations are able to run on multiple worker nodes with the assistance of the Dask distributed library. For these tests, a total of four nodes were used, two of which were Dask worker nodes. As with the other experiments, each test was repeated over 5 iterations. The primary configuration parameter modified for this experiment was `benchmark_dataset`, which was changed to point to each of the four data sets (with differing chunk sizes). Results from these tests can be found in Section C.3.

For all six function calls that were tested, the execution times decreased significantly as the individual block size of the genetic variation data set grew larger, as seen in Figure C.9. This is likely because this parameter represents the chunk length *and* width, meaning that increasing this parameter has a polynomial effect on the overall number of elements within each chunk. The decrease in execution time, as well as the increase in CPU utilization seen in Figure C.10, as the block size grows is likely because larger blocks result in less overhead when loading data into memory. Additionally, since each block of the data set is stored as an individual file on the file system, larger blocks can take advantage of striping across more Lustre storage target nodes [31]. Surprisingly, memory usage did not seem to correlate with the data chunk size. This can be seen in Figure C.11.

Table 4.5: Details of Zarr data sets used for Experiment 4.

Test Number	**Chromosome**	**Chunk Length (No. Variants)**	**Chunk Width (No. Samples)**	**Compression Level [0-9]**
1	chr22	4096	4096	0
2				1
3				3
4				5
5				9

4.5 Experiment 4: Vary Zarr Compression Level

This final experiment involved setting the compression level as the parameter of interest, where values ranged from 0 (meaning no compression) to 9 (maximum compression). This experiment was included because all previous experiments did not use any compression when storing the data. Since human genetic variation data is very sparse in nature, compression may largely impact performance when reading this data. Data compression could greatly affect execution time, since the data must be uncompressed before being readable. However, the positive trade-off is that overall storage space required to store this data can be decreased, potentially allowing for larger data sets to be analyzed. In addition to wall-clock execution time and resource usage such as CPU and memory usage, another goal of this experiment was to analyze the resulting Zarr data set sizes. Like Experiment 3, the same six function calls were tested, using two worker nodes, a scheduler node, and a driver node. To create each Zarr data set, the `blosc_compression_level` parameter was modified, and subsequently `benchmark_dataset` was changed before each benchmark, to point to each of the five data sets. Details on these five data sets can be seen in Table 4.5, and results from these tests can be found in Section C.4.

In terms of results, one point worth noting from this experiment was that execution times were almost identical when testing each compression level (seen in Figure C.12). Additionally, CPU usage did not vary much as the compression level increased, with the exception of the no-compression case (where compression level was set to 0). In this case, lower CPU utilization is likely because data did not need to be decompressed first before being used. Since compression and decompression of data is generally a CPU-intensive operation, the

Table 4.6: Resulting data set sizes based on compression level for Experiment 4.

Compression Level [0-9]	**Chunk Length (No. Variants)**	**Chunk Width (No. Samples)**	**Resulting Data Set Size**
0	4096	4096	5.3 GB
1			94 MB
3			78 MB
5			68 MB
9			53 MB

no-compression case was able to see lower CPU usage. CPU utilization for each operation can be seen in Figure C.13. Memory usage did not vastly change either, which can be seen in Figure C.14.

Although there were not any real performance gains or degradation due to changes in compression level, the practical takeaway from this experiment can be found in the resulting data set sizes for each compression level. As seen in Table 4.6, moving from compression level 0 (i.e. no compression) to compression level 1 (i.e. smallest level of compression) resulted in a 5.3 GB data set being reduced to only 94 MB, which is ultimately around a 98 percent decrease in data set size. The takeaway here is that data can be stored at higher compression levels to save space, even though execution times for performing operations on the data do not seem to increase.

Chapter 5

Conclusion

This thesis presented a software framework that can be used for benchmarking various functionality of different software libraries. Overall, the performed experiments demonstrated the ability for a user to tweak benchmarking parameters and hardware configurations, and these experiments additionally shed light on the scaling properties of several distributed algorithms. From the experiments performed, it appears that increasing the chunk size of the Zarr genetic variation data set resulted in the largest improvement to overall performance. In contrast, increasing the compression level on the data set resulted in little to no effect on overall performance, even though the resulting data set sizes were considerably smaller for higher compression levels. In addition to the experiments already performed as part of this thesis, *genben* is capable of being expanded for the benchmark of other data analysis packages in the future. As new implementations of algorithms for analyzing genetic variation data are tested and released in the future, or if entirely-new analysis methods are added to *scikit-allel*, the *genben* framework can be easily expanded to benchmark new functionality.

5.1 Limitations

Although the *genben* framework is flexible in that it allows for further expansion, some limitations do exist with the tool in its current state. One possible limitation is that the tool measures execution time, but it relies on an additional piece of software (i.e. Telegraf) to collect and report system metrics. Although both tools can natively report data to an

InfluxDB server, additional work is currently needed to combine the metrics data with the benchmark results information.

Another limitation is that the tool does not natively generate charts or other useful statistics; it only measures and records information. In this thesis, for instance, scripts were created to generate the charts from results and system metrics data, but these scripts are dependent on each individual use case, and they consequently must be heavily edited for each individual test.

5.2 Future Work

From the limitations above, it is apparent that *genben* could be improved to better serve its users and provide more insight on what it is measuring. The ability to automatically combine benchmark data with metrics reported by Telegraf could be invaluable. This could be done by correlating the benchmark result timestamps with the measurement timestamps reported by *Telegraf*, and *genben* could combine these to create a single, unified data set. Another potential item for future work could involve having *genben* automatically launch a *Telegraf* process on all nodes automatically, so that the user would not be required to launch these processes manually.

Additionally, the ability for a user to set various parameters that control generation of benchmarking graphs could result in a more practical and more versatile benchmarking framework. By having *genben* create rudimentary graphs, at the very least, a user such as a software developer could quickly gain insight as to how a particular algorithm is performing, based on certain configuration parameters or hardware environments.

Bibliography

[1] Francesc Alted. *What Is Blosc? | Blosc Main Blog Page*. URL: http://blosc.org/pages/blosc-in-depth/ (cit. on p. 15).

[2] Anaconda. *Anaconda Software Distribution*. 2016. URL: https://anaconda.com (cit. on p. 25).

[3] E. Anderson et al. *LAPACK Users' Guide*. Third. Society for Industrial and Applied Mathematics, 1999. ISBN: 0-89871-447-8 (cit. on p. 7).

[4] Eric Auel, Edmon Begoli, and Alistair Miles. *genben*. URL: https://github.com/ornl-oxford/genben (cit. on pp. 12, 25).

[5] R. G. Brook et al. "Beacon: Deployment and Application of Intel Xeon Phi Coprocessors for Scientific Computing". In: *Computing in Science and Engineering* 17.Mar/Apr (2015), pp. 65–72 (cit. on p. 25).

[6] Dask Development Team. *Dask Distributed*. URL: https://distributed.dask.org/ (cit. on pp. 9, 19).

[7] Dask Development Team. *Dask: Library for dynamic task scheduling*. 2016. URL: https://dask.org (cit. on pp. 3, 9, 19, 27).

[8] *Dask-Jobqueue*. URL: https://jobqueue.dask.org/ (cit. on p. 10).

[9] Facebook. *Zstandard - Real-time data compression algorithm*. URL: https://facebook.github.io/zstd/ (cit. on p. 15).

[10] "Finishing the euchromatic sequence of the human genome". In: *Nature* 431.7011 (Oct. 2004), pp. 931–945. ISSN: 0028-0836. DOI: 10.1038/nature03001 (cit. on p. 1).

[11] Alex Gaynor et al. *pyperformance: The Python Performance Benchmark Suite*. URL: https://github.com/python/pyperformance (cit. on p. 3).

[12] Richard A. Gibbs et al. "A global reference for human genetic variation". In: *Nature* 526.7571 (Oct. 2015), pp. 68–74. ISSN: 0028-0836. DOI: 10.1038/nature15393 (cit. on pp. 1, 27, 28).

[13] *InfluxDB 1.7*. URL: https://docs.influxdata.com/influxdb/v1.7/ (cit. on p. 17).

[14] *INI formats*. URL: http://www.nongnu.org/chmspec/latest/INI.html (cit. on p. 13).

[15] Ian Jolliffe. *Principal Component Analysis*. New York, NY: Springer, 1986. ISBN: 978-1-4757-1906-2. DOI: 10.1007/978-1-4757-1904-8 (cit. on pp. 1, 5, 7).

[16] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *{SciPy}: Open source scientific tools for {Python}*. URL: http://www.scipy.org/ (cit. on pp. 3, 27).

[17] Eric S. Lander et al. "Initial sequencing and analysis of the human genome". In: *Nature* 409.6822 (Feb. 2001), pp. 860–921. ISSN: 00280836. DOI: 10.1038/35057062 (cit. on p. 1).

[18] *LZ4 - Extremely fast compression*. URL: https://lz4.github.io/lz4/ (cit. on p. 15).

[19] Alistair Miles and Nick Harding. "cggh/scikit-allel: V1.1.8". In: 10.5281/zenodo.822784 (July 2017). DOI: 10.5281/zenodo.822784 (cit. on pp. 8, 12).

[20] *NumPy*. URL: https://www.numpy.org/ (cit. on pp. 3, 7).

[21] *Open MPI: Open Source High Performance Computing*. URL: https://www.open-mpi.org/ (cit. on p. 22).

[22] Georgios A Pavlopoulos et al. "Unraveling genomic variation from next generation sequencing data". In: *BioData Mining* 6.1 (Dec. 2013), p. 13. ISSN: 1756-0381. DOI: 10.1186/1756-0381-6-13 (cit. on p. 1).

[23] Karl Pearson. "On lines and planes of closest fit to systems of points in space". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2.11 (1901), pp. 559–572. ISSN: 1941-5982. DOI: 10.1080/14786440109462720 (cit. on p. 5).

[24] F Pedregosa et al. "Scikit-learn: Machine Learning in {P}ython". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on pp. 3, 27).

[25] Jonathon Shlens. "A Tutorial on Principal Component Analysis". In: (Apr. 2014) (cit. on p. 6).

[26] *snappy | A fast compressor/decompressor*. URL: http://google.github.io/snappy/ (cit. on p. 15).

[27] *Telegraf 1.10*. URL: https://docs.influxdata.com/telegraf/v1.10/ (cit. on pp. 24, 25).

[28] The 1000 Genomes Project Consortium. *The 1000 Genomes Project: FTP Download Site*. 2013. URL: ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/ (cit. on pp. 1, 42).

[29] The 1000 Genomes Project Consortium et al. "An integrated map of genetic variation from 1,092 human genomes". In: *Nature* 491.7422 (Nov. 2012), p. 56. ISSN: 1476-4687. DOI: 10.1038/NATURE11632 (cit. on p. 1).

[30] The International Genome Sample Resource. *VCF (Variant Call Format) version 4.0*. URL: http://www.internationalgenome.org/wiki/Analysis/vcf4.0/ (cit. on pp. 4, 5).

[31] The National Institute for Computational Sciences. *I/O and Lustre Usage*. URL: https://www.nics.tennessee.edu/computing-resources/file-systems/io-lustre-tips#file-striping-basics (cit. on p. 31).

[32] The Python Software Foundation. *Reference Guide - pip*. 2019. URL: https://pip.pypa.io/en/stable/reference/ (cit. on p. 18).

[33] The SciPy Community. *numpy.array - NumPy Manual*. URL: https://docs.scipy.org/doc/numpy/reference/generated/numpy.array.html (cit. on p. 1).

[34] "Understanding Human Genetic Variation". In: (2007) (cit. on p. 8).

[35] Zarr Development Team. *Zarr version 2.2.0*. URL: https://zarr.readthedocs.io/en/stable/ (cit. on pp. 8, 12).

[36] *zlib*. URL: https://zlib.net/ (cit. on p. 15).

Appendices

A Example Data

```
Total duration: 1 min 31.7 sec
Start date: 2019-07-04 13:42:43
End date: 2019-07-04 13:44:22
Raw value minimum: 800 ms
Raw value maximum: 1.39 sec

Number of calibration run: 1
Number of run with values: 20
Total number of run: 21
Number of warmup per run: 1
Number of value per run: 3
Loop iterations per value: 1
Total number of values: 60

Minimum:          800 ms
Median +- MAD:    1.09 sec +- 0.03 sec
Mean +- std dev: 1.10 sec +- 0.08 sec
Maximum:          1.39 sec

  0th percentile: 800 ms (-27% of the mean) -- minimum
  5th percentile: 1.04 sec (-5% of the mean)
 25th percentile: 1.06 sec (-3% of the mean) -- Q1
 50th percentile: 1.09 sec (-1% of the mean) -- median
 75th percentile: 1.12 sec (+2% of the mean) -- Q3
 95th percentile: 1.22 sec (+11% of the mean)
100th percentile: 1.39 sec (+27% of the mean) -- maximum
Number of outlier (out of 979 ms..1203 ms): 6
```

Figure A.1: Output received when running the Python Performance library. This output provides statistics on the several benchmarks performed.

```
1  ##fileformat=VCFv4.3
2  ##FILTER=<ID=PASS,Description="All filters passed">
3  ##fileDate=31052018_15h52m43s
4  ##source=IGSRpipeline
5  ##reference=ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/technical/reference/GRCh38_reference_genome/GRCh38_full_analysis_set_plus_decoy_hla.fa
6  ##FORMAT=<ID=GT,Number=1,Type=String,Description="Phased Genotype">
7  ##contig=<ID=chr22>
8  ##INFO=<ID=AF,Number=A,Type=Float,Description="Estimated allele frequency in the range (0,1)">
9  ##INFO=<ID=AC,Number=A,Type=Integer,Description="Total number of alternate alleles in called genotypes">
10 ##INFO=<ID=NS,Number=1,Type=Integer,Description="Number of samples with data">
11 ##INFO=<ID=AN,Number=1,Type=Integer,Description="Total number of alleles in called genotypes">
12 ##INFO=<ID=EAS_AF,Number=A,Type=Float,Description="Allele frequency in the EAS populations calculated from AC and AN, in the range (0,1)">
13 ##INFO=<ID=EUR_AF,Number=A,Type=Float,Description="Allele frequency in the EUR populations calculated from AC and AN, in the range (0,1)">
14 ##INFO=<ID=AFR_AF,Number=A,Type=Float,Description="Allele frequency in the AFR populations calculated from AC and AN, in the range (0,1)">
15 ##INFO=<ID=AMR_AF,Number=A,Type=Float,Description="Allele frequency in the AMR populations calculated from AC and AN, in the range (0,1)">
16 ##INFO=<ID=SAS_AF,Number=A,Type=Float,Description="Allele frequency in the SAS populations calculated from AC and AN, in the range (0,1)">
17 ##INFO=<ID=VT,Number=.,Type=String,Description="indicates what type of variant the line represents">
18 ##INFO=<ID=EX_TARGET,Number=0,Type=Flag,Description="indicates whether a variant is within the exon pull down target boundaries">
19 ##INFO=<ID=DP,Number=1,Type=Integer,Description="Approximate read depth; some reads may have been filtered">
20 #CHROM  POS ID  REF ALT QUAL    FILTER  INFO    FORMAT  HG00096 HG00097 HG00099 HG00100 HG00101 HG00102 HG00103 HG00104 HG00105 HG00106 HG00107 HG00108 HG00109 HG00110
21 chr22   10516173    .   A   G   .   PASS    AC=121;AN=5096;DP=8203;AF=0.02;EAS_AF=0;EUR_AF=0.02;AFR_AF=0.05;AMR_AF=0.02;SAS_AF=0;VT=SNP;NS=2548 GT  0|0 0|0 0|0 0|0 0|0
22 chr22   10522217    .   G   A   .   PASS    AC=86;AN=5096;DP=9085;AF=0.02;EAS_AF=0;EUR_AF=0;AFR_AF=0.06;AMR_AF=0;SAS_AF=0;VT=SNP;NS=2548    GT  0|0 0|0 0|0 0|0 0|0
23 chr22   10526445    .   A   G   .   PASS    AC=4929;AN=5096;DP=7978;AF=0.97;EAS_AF=0.98;EUR_AF=0.99;AFR_AF=0.92;AMR_AF=0.96;SAS_AF=1;VT=SNP;NS=2548 GT  1|1 1|1 1|1 1|1
24 chr22   10527034    .   G   T   .   PASS    AC=275;AN=5096;DP=5974;AF=0.05;EAS_AF=0.01;EUR_AF=0.02;AFR_AF=0.12;AMR_AF=0.03;SAS_AF=0.06;VT=SNP;NS=2548   GT  0|0 0|0 0|0
25 chr22   10527038    .   C   G   .   PASS    AC=208;AN=5096;DP=5947;AF=0.04;EAS_AF=0.06;EUR_AF=0.03;AFR_AF=0.01;AMR_AF=0.06;SAS_AF=0.06;VT=SNP;NS=2548   GT  0|0 0|0 0|0
```

Figure A.2: Example contents of a VCF file. Data Source: The 1000 Genomes Project [28].

B Genomic Benchmark Tool

B.1 Default User Configuration File

```
1  [ftp]
2  # Whether or not the FTP download module should be used when running benchmark
3  # tool in Setup mode.
4  enabled = False
5  # FTP Server to retrieve files from:
6  server = ftp.someserver.com
7  # Username and password to login into FTP server with.
8  # Leave both fields blank for anonymous login.
9  username =
10 password =
11 # Determines whether to use TLS or non-secure FTP
12 use_tls = False
13 # Directory on {server} to download files from:
14 directory = files
15 # File or list of files to download within {directory}.
16 # - If downloading list of files, names should be separated using
17 # delimiter specified in {file_delimiter}.
18 # - To download all files within {directory}, set {files} to "*".
19 files = sample.vcf
20 # If downloading multiple files, the delimiter that will be used for
21 # separating multiple filenames.
22 file_delimiter = |
```

Figure B.1: Default user configuration file generated by *genben*. The configuration file contains several sections for configuring the benchmark tool.

```
23 [vcf_to_zarr]
24 # Whether or not the VCF to Zarr Converter module should be used when running
25 # benchmark tool in Setup mode.
26 # Note: To run VCF to Zarr conversion as part of the benchmark process, see
27 # the [benchmark] configuration section.
28 enabled = False
29 # Alt number to assume when converting to Zarr format.
30 # If set to "auto", this will be determined during the conversion process.
31 alt_number = auto
32 # Number of variants of chunks in which data are processed.
33 # If set to "default", the default value from scikit-allel is used.
34 chunk_length = default
35 # Number of samples to use when storing chunks in output.
36 # If set to "default", the default value from scikit-allel is used.
37 chunk_width = default
38 # Type of compression to utilize when storing Zarr-formatted data.
39 # Available Compressor Types:
40 # - Blosc
41 # - LZ4 (not yet implemented)
42 # - LZMA (not yet implemented)
43 compressor = Blosc
44 # (Blosc Compressor Only)
45 # Specifies the type of compression algorithm for the Blosc compressor to use.
46 # Possible values: zstd, blosclz, lz4, lz4hc, zlib, snappy
47 blosc_compression_algorithm = zstd
48 # (Blosc Compressor Only)
49 # Specifies the compression level to use when converting to Zarr
50 # Possible values: any integer between 0 and 9
51 blosc_compression_level = 1
52 # (Blosc Compressor Only)
53 # Specifies the shuffle mode to use with the Blosc compressor.
54 # Possible Values:
55 # - NOSHUFFLE: 0
56 # - SHUFFLE: 1
57 # - BITSHUFFLE: 2
58 # - AUTOSHUFFLE: -1
59 blosc_shuffle_mode = -1
60
61 [dask]
62 # Enables/disables connection to a Dask distributed scheduler.
63 enabled = False
64 # The IP address and port of the scheduler to connect to.
65 scheduler_address = 127.0.0.1
66 scheduler_port = 8786
```

Figure B.1: Continued.

```
67  [benchmark]
68  # Specifies how many times the benchmark tool should run
69  # each available benchmark.
70  benchmark_number_runs = 5
71  # Specifies where the benchmark tool should get its data from.
72  # Possible Values:
73  # - vcf: uses datasets within the ./data/vcf/ directory. This option will
74  # result in VCF data being converted to Zarr format as part of the
75  # benchmarking process.
76  # - zarr: uses Zarr-formtted datasets within the ./data/zarr/ directory. This
77  # option will result in skipping the benchmark of the VCF to Zarr
78  # conversion process.
79  benchmark_data_input = vcf
80  # Specifies which dataset to use for the benchmarking process.
81  # If a value * is specified, the benchmark will concatenate all data in
82  # the ./data/zarr/ directory.
83  # - Note: In order to use concatenation, all data sets must have
84  # the same number of samples to align properly.
85  # - Note: Concatenation (*) can only be used when
86  # benchmark_data_input is set to zarr.
87  benchmark_dataset =
88  # Number of variants to include from the dataset input for benchmarking.
89  # If a value of -1 is passed, then all variants will be included.
90  benchmark_num_variants = -1
91  # Number of samples to include from the dataset input for benchmarking.
92  # If a value of -1 is passed, then all samples will be included.
93  benchmark_num_samples = -1
94  # Enables/disables running simple aggregations as part of
95  # the benchmarking process.
96  benchmark_aggregations = True
97  # Enables Principal Component Analysis as part of the benchmarking process.
98  benchmark_pca = True
99  # Type of data array to use when loading the genotype data for benchmarking.
100 # Possible Values:
101 # - Normal: 0
102 # - Dask: 1
103 # - Chunked: 2
104 genotype_array_type = 1
105 # [Dask] Chunk length and width to use when creating a Dask genotype array.
106 # These parameters are only used if genotype_array_type is set to use Dask.
107 # If a value of -1 is passed, it will inherit the underlying chunk size.
108 # If a value of -1 is passed for both parameters, no re-chunking will occur.
109 dask_genotype_array_chunk_variants = -1
110 dask_genotype_array_chunk_samples = -1
```

Figure B.1: Continued.

```
111 # [PCA Benchmark] Specifies the number of PCs to keep when performing PCA.
112 pca_number_components = 10
113 # [PCA Benchmark] Specifies the type of data scaling to apply to the data set.
114 # Possible Values:
115 # - Standard: 0
116 # - Patterson: 1
117 # - None: 2
118 pca_data_scaler = 1
119 # [PCA Benchmark] Sets the number of SNPs to use as a subset of the data set.
120 # If the size of the data set is smaller, that will be used instead.
121 # Additionally, a value of -1 can be passed to include everything.
122 pca_subset_size = -1
123 # [PCA Benchmark: LD] Specifies whether to enable or disable LD pruning.
124 pca_ld_enabled = False
125 # [PCA Benchmark: LD] Sets the number of iterations to perform LD pruning.
126 pca_ld_pruning_number_iterations = 2
127 # [PCA Benchmark: LD] Sets the window size to use when performing LD pruning.
128 pca_ld_pruning_size = 100
129
130 # [PCA Benchmark: LD] Number of variants to advance when performing LD pruning.
131 pca_ld_pruning_step = 20
132 # [PCA Benchmark: LD] Sets the maximum value of r^2 to include variants.
133 pca_ld_pruning_threshold = 0.01
```

Figure B.1: Continued.

```
134 [output.csv]
135 # Enables/disables benchmark results output to CSV-style file.
136 enabled = True
137 # Specifies which delimiter to use to separate fields within each row of data
138 delimiter = |
139 
140 [output.influxdb]
141 # Enables/disables benchmark results output to an InfluxDB server
142 enabled = False
143 # Host/IP address and port of the machine running InfluxDB
144 host = localhost
145 port = 8086
146 # Username and password credentials to login to InfluxDB instance
147 username = root
148 password = root
149 # Name of database to store benchmark results
150 database_name = benchmark
151 # (Optional) Tag that can be used to group multiple benchmark results
152 benchmark_group =
153 # (Optional) Specifies which machine is running the benchmark.
154 # This option can be useful for matching benchmark results when pushing
155 # data from multiple machines.
156 device_name =
```

Figure B.1: Continued.

B.2 Example Parameter Sweep Configuration File

```
1 [benchmark]
2 # Dataset to use for the benchmarking process.
3 benchmark_dataset = chr10.zarr,chr11.zarr
4
5 # Number of samples to include from the dataset input for benchmarking.
6 # If a value of -1 is passed, then all samples will be included.
7 benchmark_num_samples = 500,1000,1500,2000
```

Figure B.2: Example parameter sweep file that can be used for testing multiple parameter combinations using *genben*.

B.3 Benchmark Usage Example: Configuration File

```
1  [ftp]
2  enabled = True
3  server = ftp.1000genomes.ebi.ac.uk
4  directory = vol1/ftp/release/20130502
5  files = ALL.chr10.phase3.20130502.genotypes.vcf.gz
6
7  [vcf_to_zarr]
8  enabled = True
9
10 [dask]
11 enabled = True
12 scheduler_address = 127.0.0.1
13 scheduler_port = 8786
14
15 [benchmark]
16 benchmark_number_runs = 10
17 benchmark_data_input = zarr
18 benchmark_dataset = ALL.chr10.phase3.20130502.genotypes
19 benchmark_aggregations = True
20 benchmark_pca = True
21
22 [output.csv]
23 enabled = True
24 delimiter = ,
```

Figure B.3: Example of a user-edited configuration file that can be used with *genben*. This configuration is used to achieve the goals described in Section 3.3.

C Experimental Results

C.1 Experiment 1: Single-Node Performance

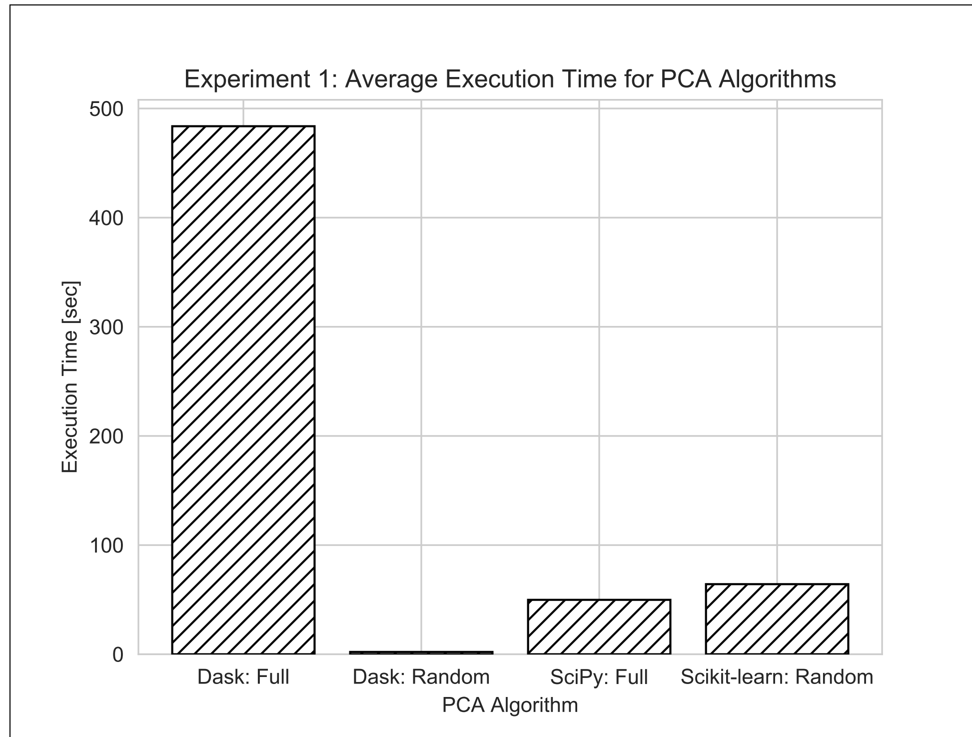

Figure C.1: Average execution times for each PCA algorithm tested in Experiment 1.

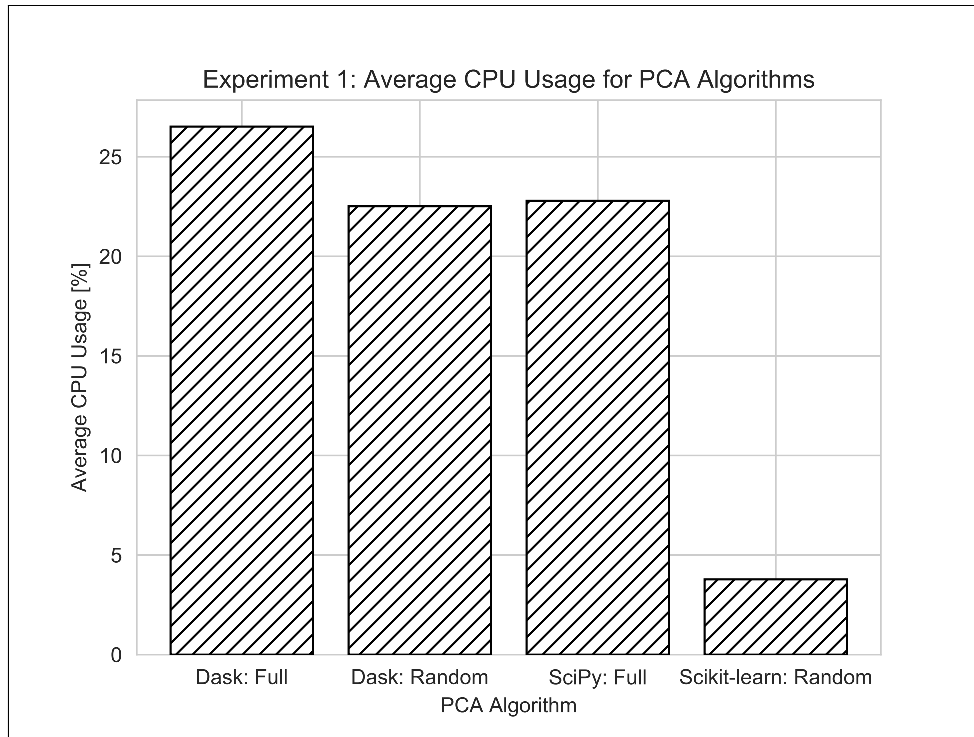

Figure C.2: Average CPU utilization for each PCA algorithm tested in Experiment 1.

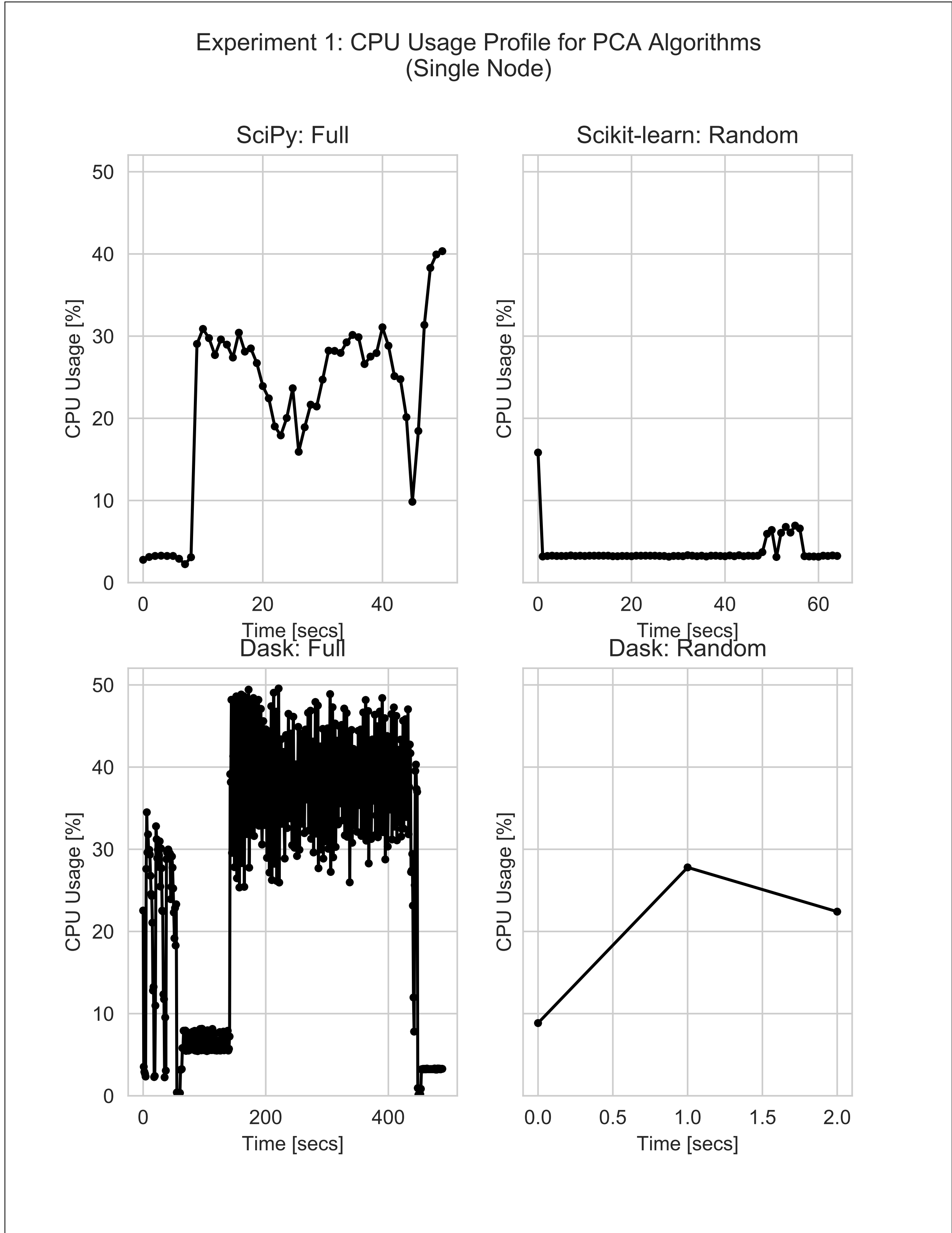

Figure C.3: CPU usage profile for each PCA algorithm tested in Experiment 1. The first benchmark iteration for each is shown.

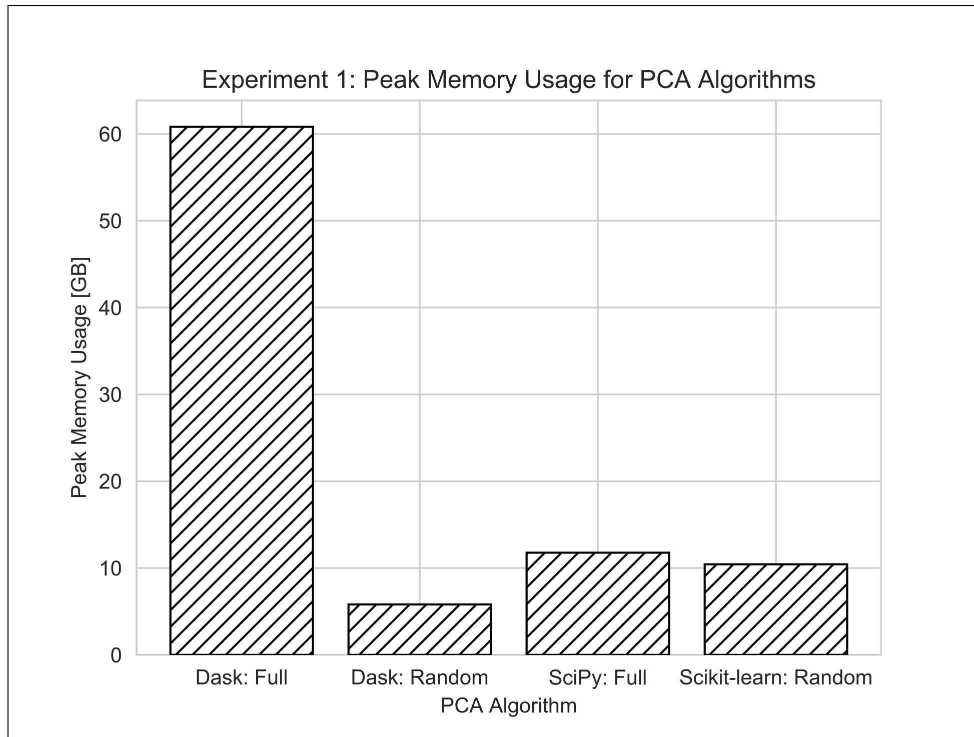

Figure C.4: Maximum memory usage for each PCA algorithm tested in Experiment 1.

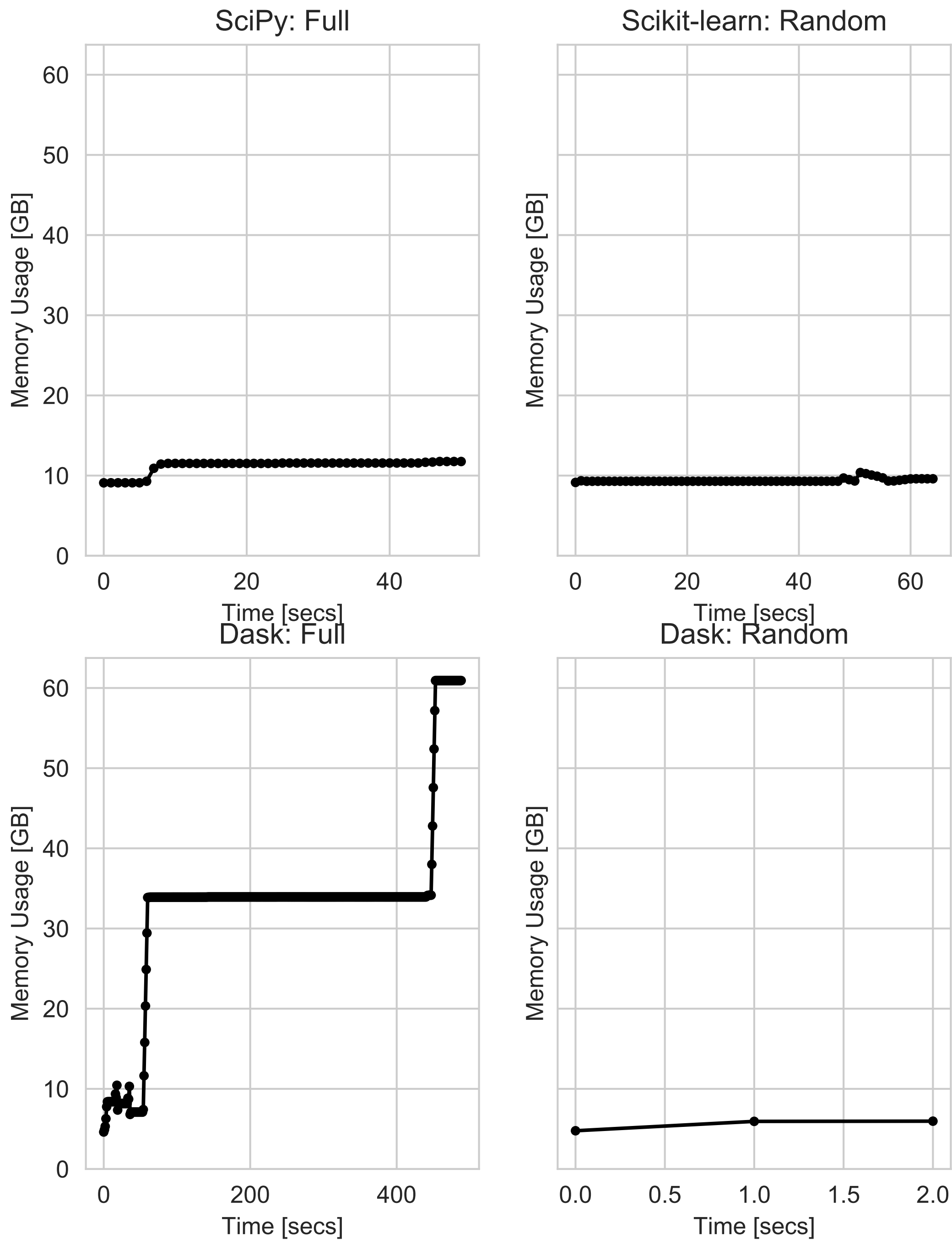

Figure C.5: Memory usage profile for each PCA algorithm tested in Experiment 1. The first benchmark iteration for each is shown.

C.2 Experiment 2: Scale Number of Nodes

Figure C.6: Average execution times for the allele counting algorithm (shown left) and Dask randomized PCA implementation (shown right) tested in Experiment 2.

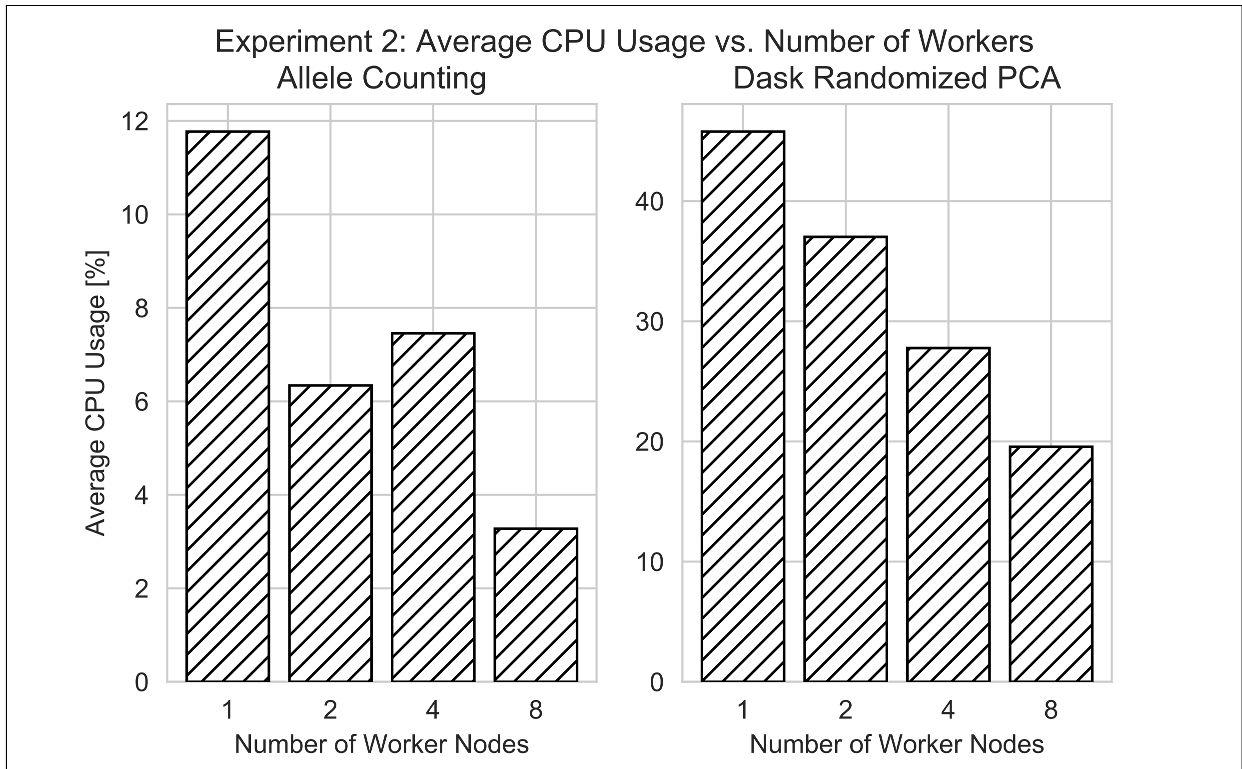

Figure C.7: Average CPU usage for the allele counting algorithm (shown left) and Dask randomized PCA implementation (shown right) tested in Experiment 2.

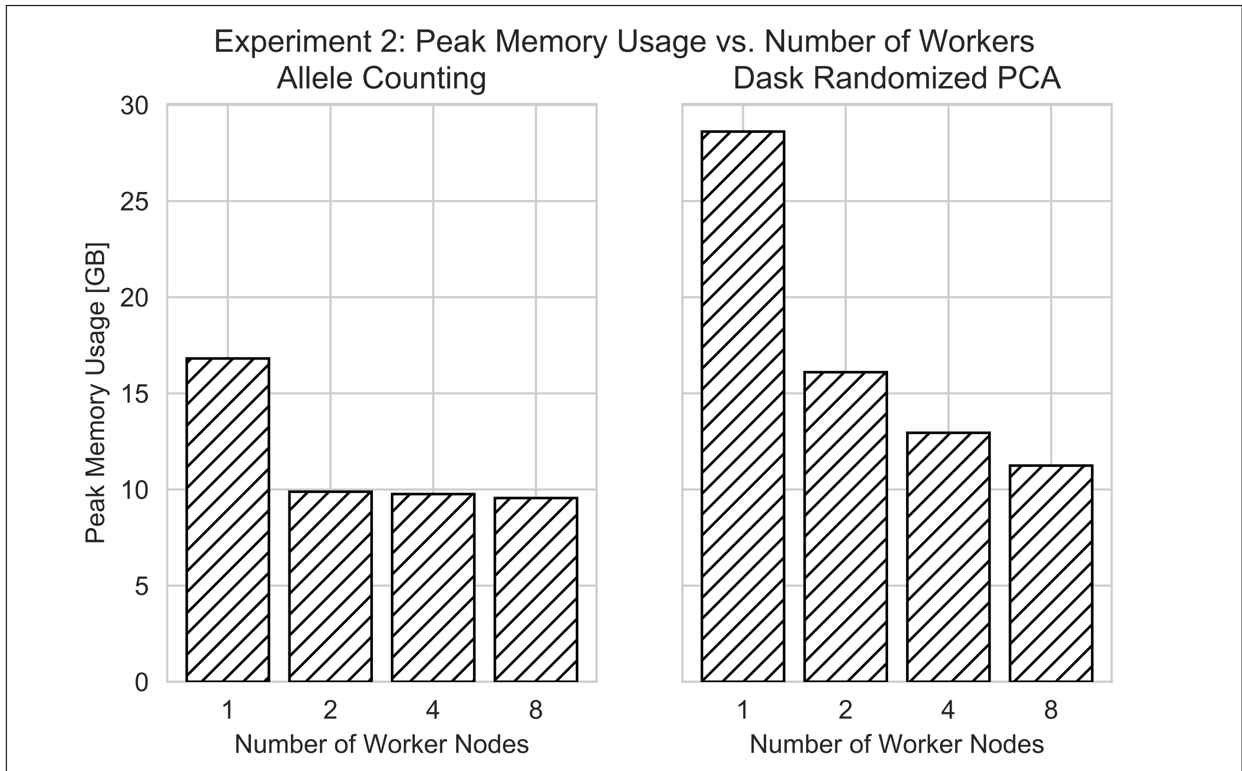

Figure C.8: Peak memory usage for the allele counting algorithm (shown left) and Dask randomized PCA implementation (shown right) tested in Experiment 2.

C.3 Experiment 3: Vary Zarr Data Chunk Size

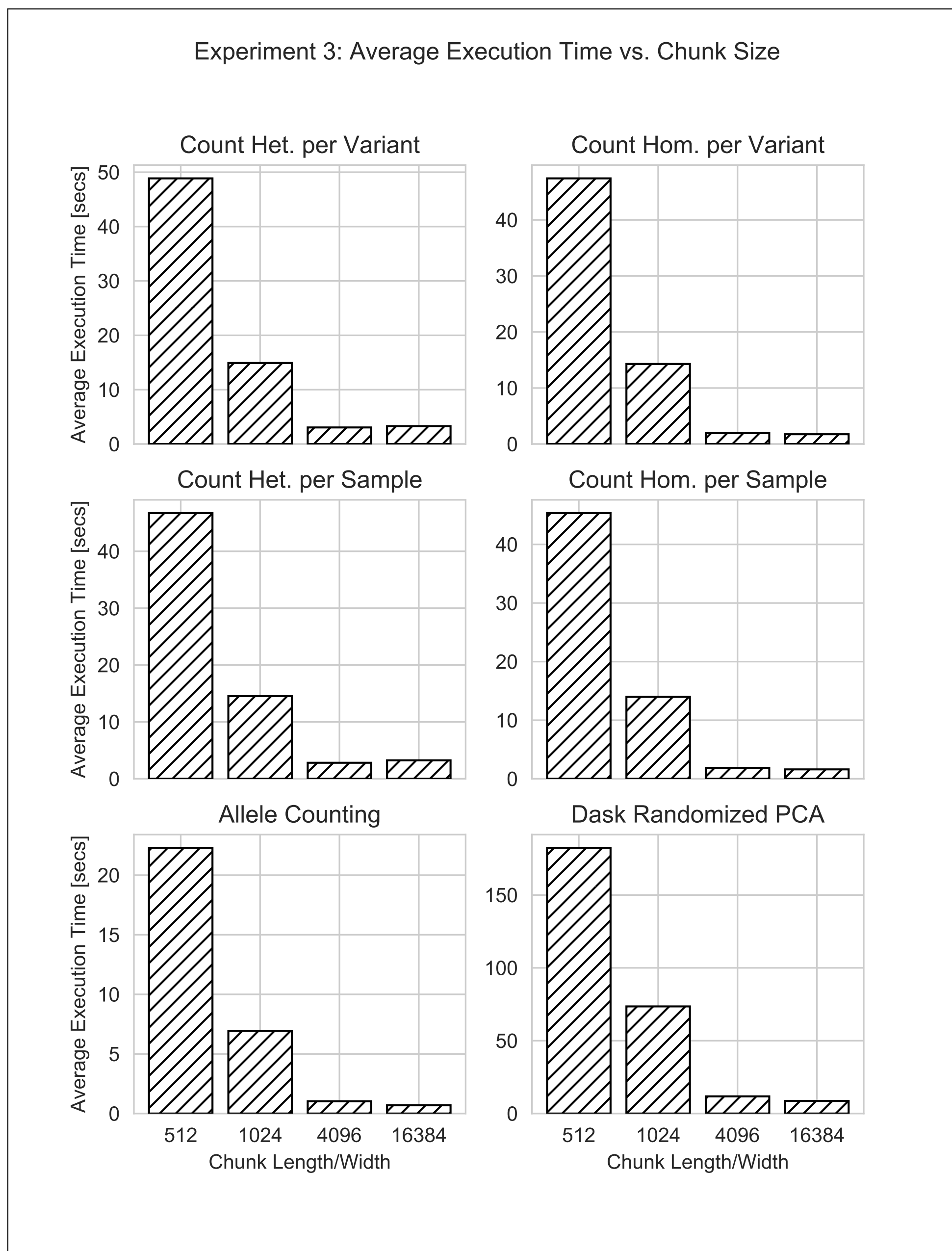

Figure C.9: Average execution times for the six functions tested in Experiment 3.

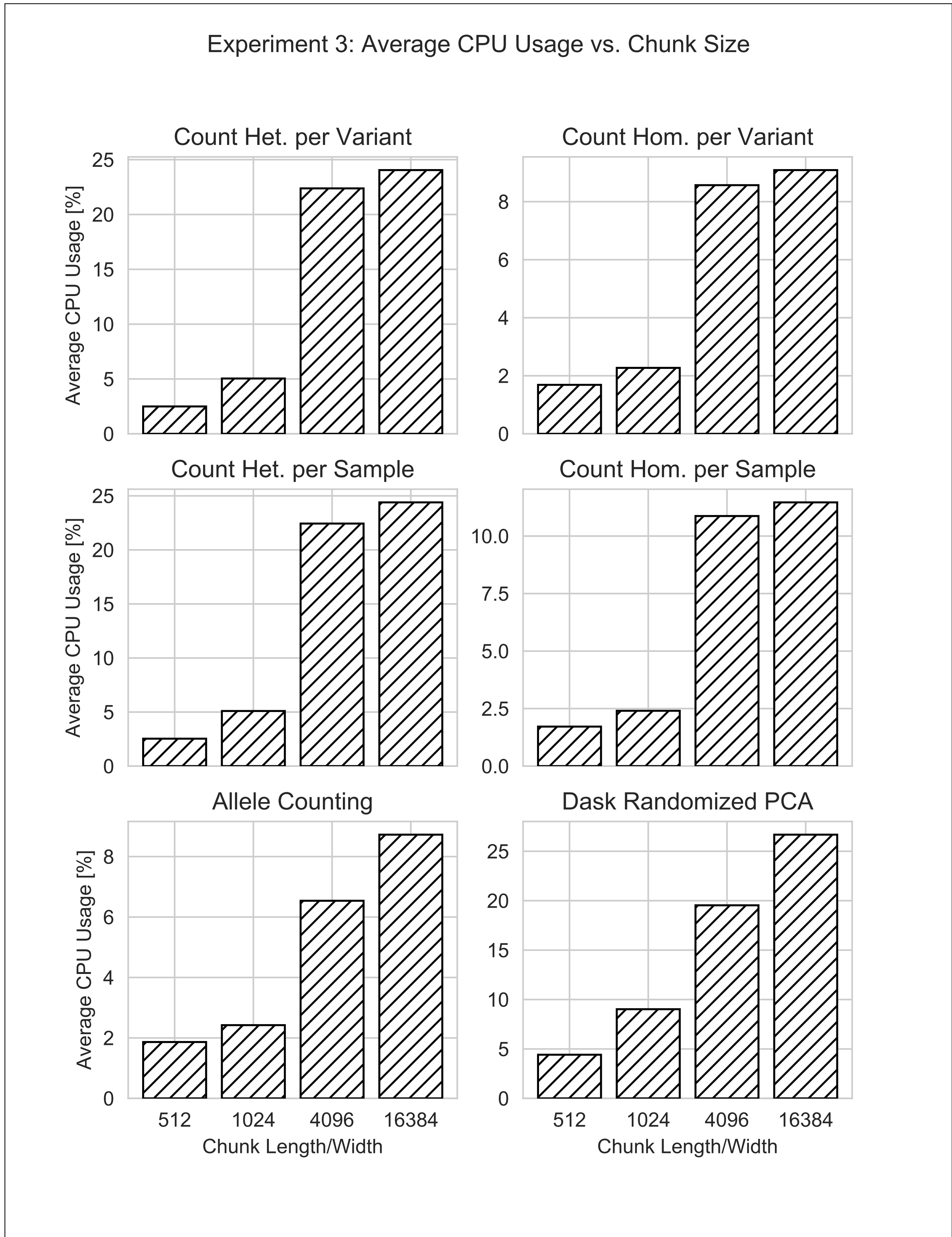

Figure C.10: Average CPU usage for the six functions tested in Experiment 3.

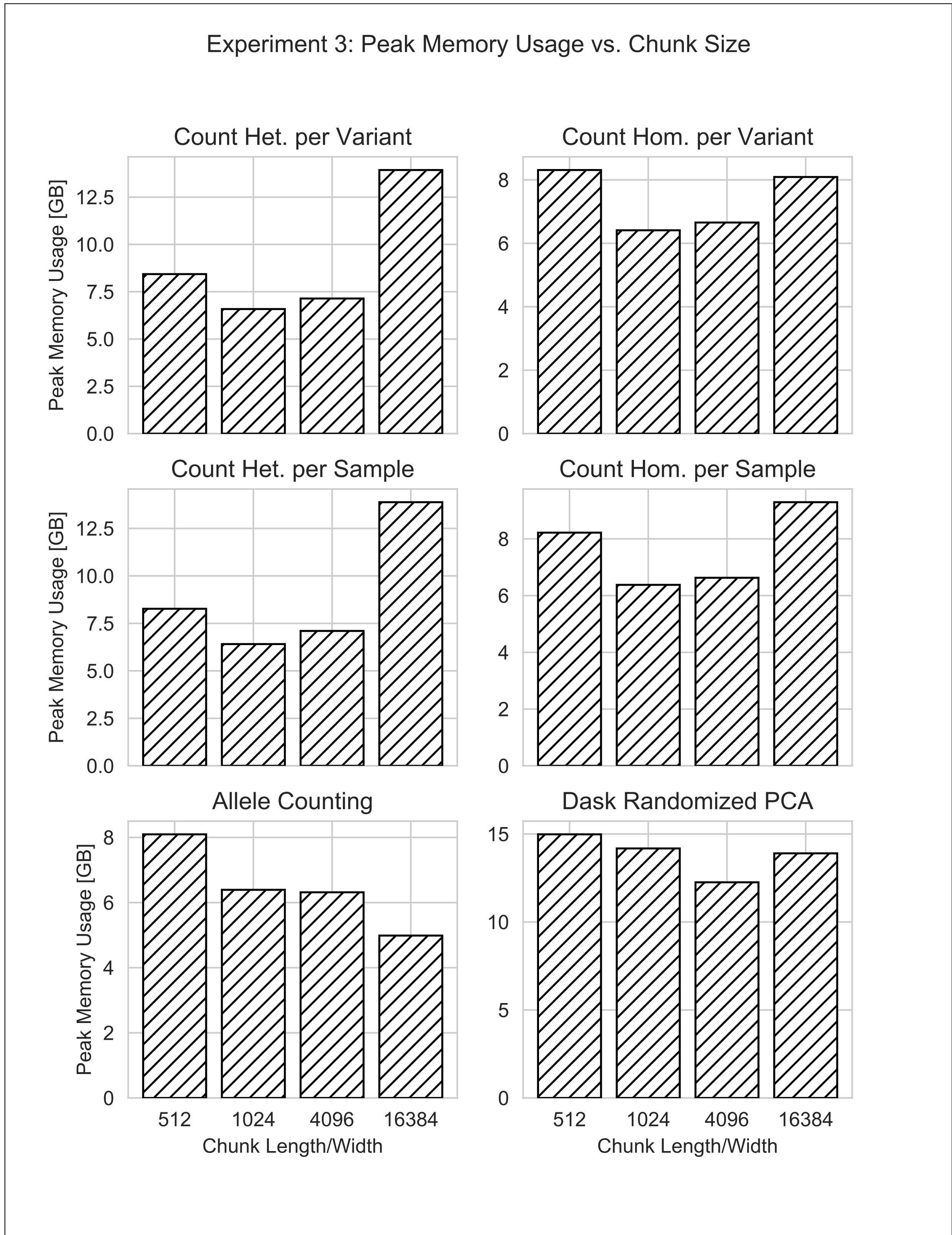

Figure C.11: Peak memory usage for the six functions tested in Experiment 3.

C.4 Experiment 4: Vary Zarr Compression Level

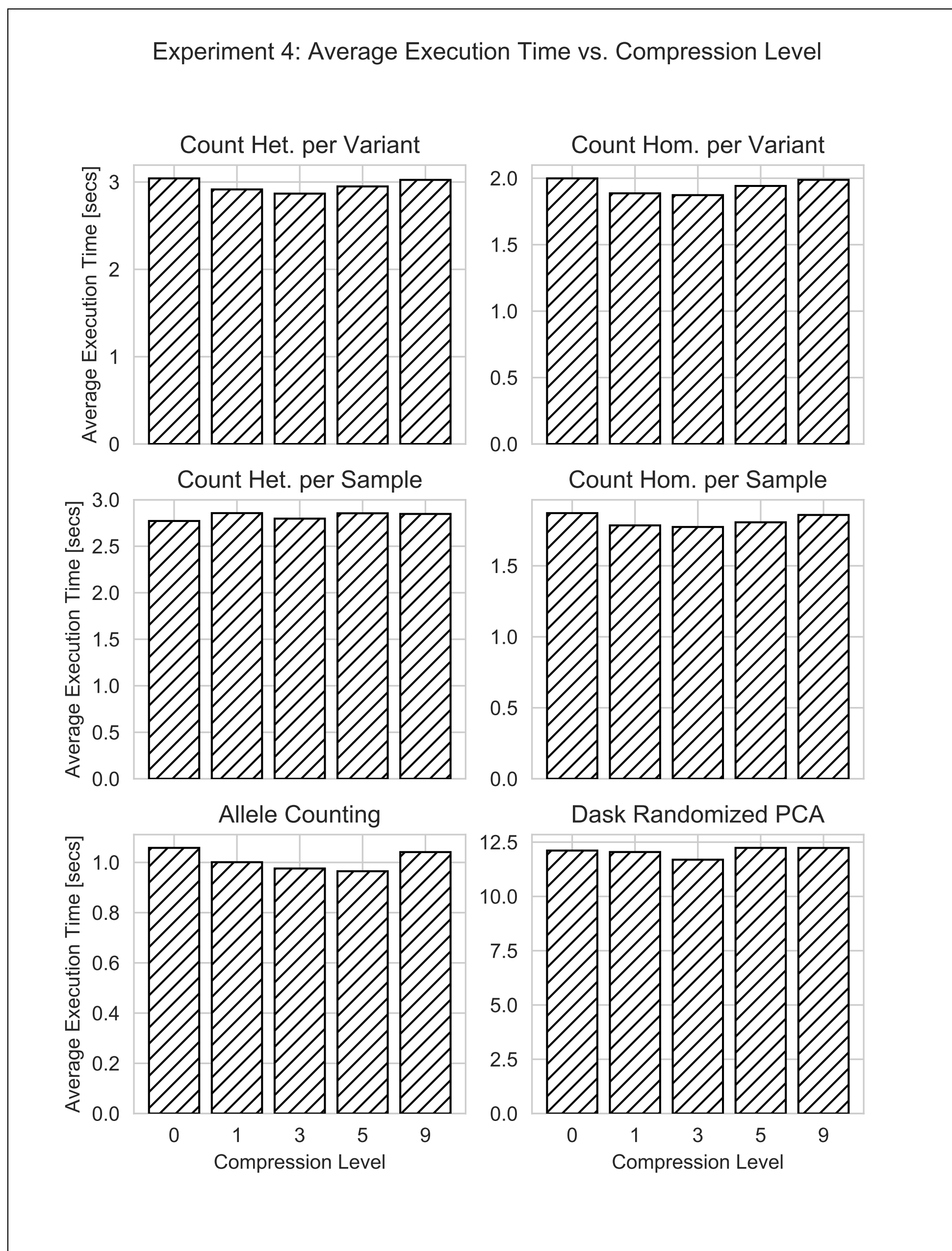

Figure C.12: Average execution times for the six functions tested in Experiment 4.

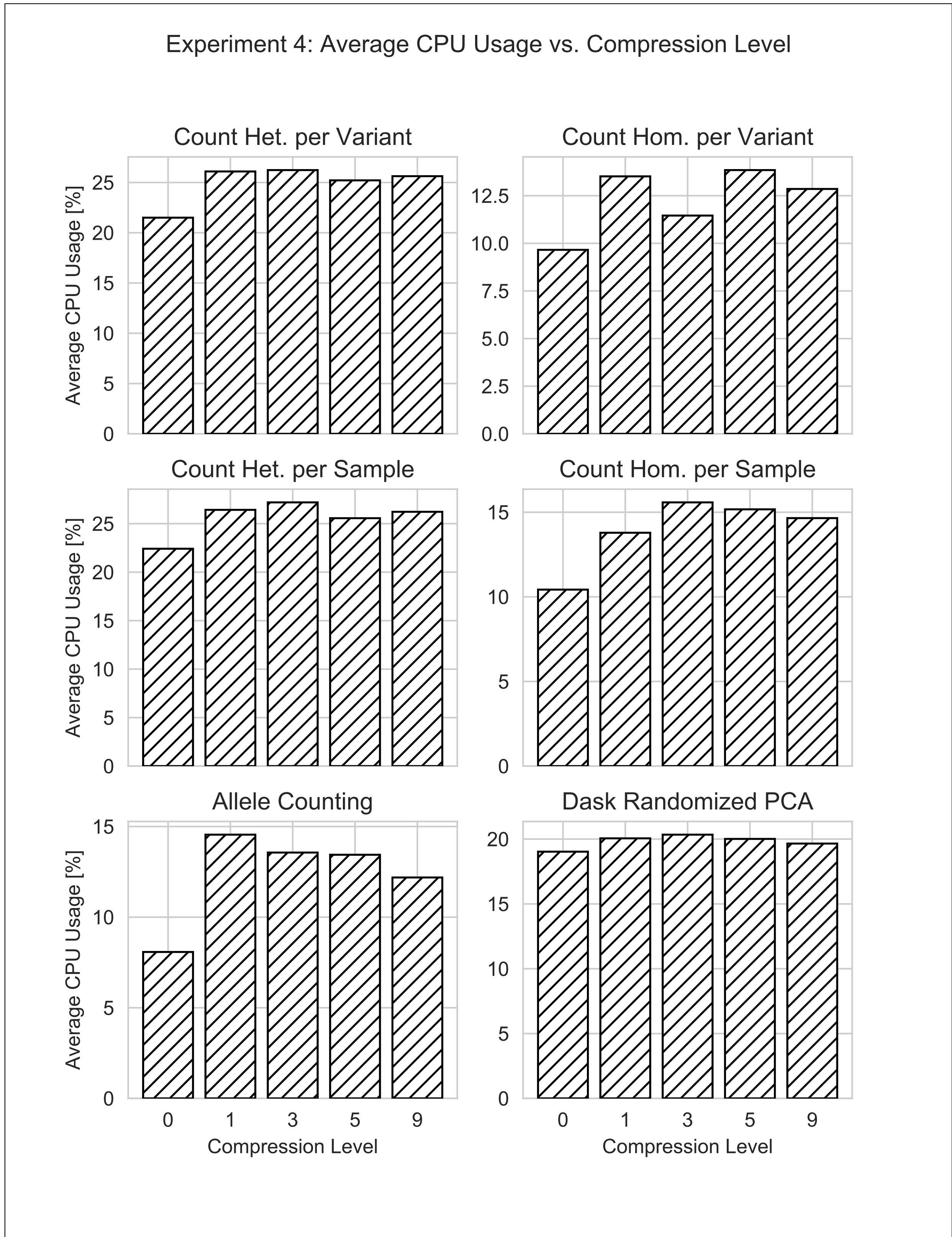

Figure C.13: Average CPU usage for the six functions tested in Experiment 4.

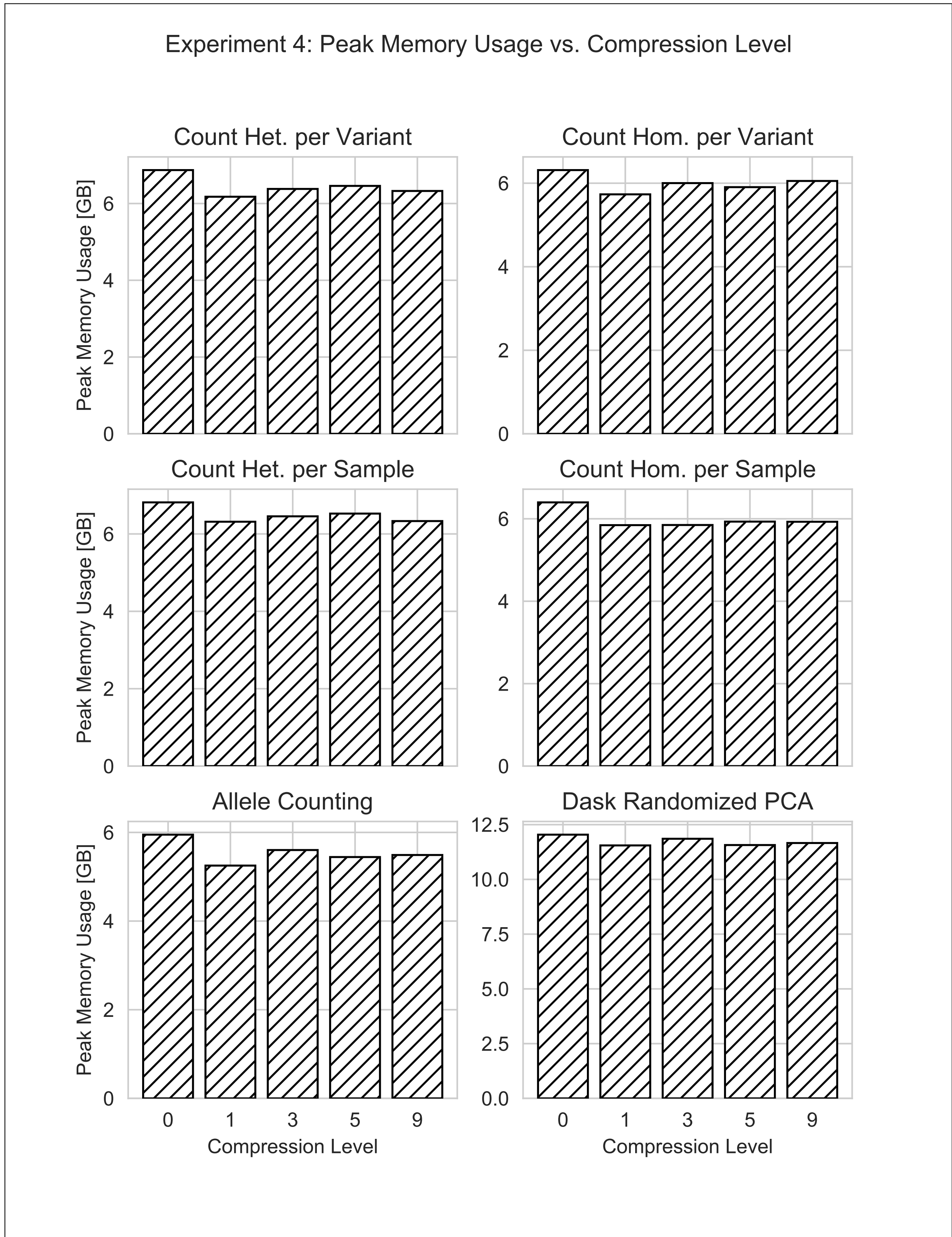

Figure C.14: Peak memory usage for the six functions tested in Experiment 4.

Vita

Eric Auel is a Masters Student at the University of Tennessee, Knoxville, and he received a Bachelor of Science Degree in Electrical Engineering in 2018. Eric grew up in Seymour, Tennessee and was a graduate of Seymour High School. Eric's professional interests include machine learning, robotics integration, and systems automation.