A Modular Parallel Pipeline Architecture for GWAS Applications in a Cluster Environment

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Faheem Abrar

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Or

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9
Canada

Abstract

A Genome Wide Association Study (GWAS) is an important bioinformatics method to associate variants with traits, identify causes of diseases and increase plant and crop production. There are several optimizations for improving GWAS performance, including running applications in parallel. However, it can be difficult for researchers to utilize different data types and workflows using existing approaches.

A potential solution for this problem is to model GWAS algorithms as a set of modular tasks. In this thesis, a modular pipeline architecture for GWAS applications is proposed that can leverage a parallel computing environment as well as store and retrieve data using a shared data cache.

To show that the proposed architecture increases performance of GWAS applications, two case studies are conducted in which the proposed architecture is implemented on a bioinformatics pipeline package called TASSEL and a GWAS application called FaST-LMM using both Apache Spark and Dask as the parallel processing framework and Redis as the shared data cache. The case studies implement parallel processing modules and shared data cache modules according to the specifications of the proposed architecture.

Based on the case studies, a number of experiments are conducted that compare the performance of the implemented architecture on a cluster environment with the original programs. The experiments reveal that the modified applications indeed perform faster than the original sequential programs. However, the modified applications do not scale with cluster resources, as the sequential part of the operations prevent the parallelization from having linear scalability.

Finally, an evaluation of the architecture was conducted based on feedback from software developers and bioinformaticians. The evaluation reveals that the domain experts find the architecture useful; the implementations have sufficient performance improvement and they are also easy to use, although a GUI based implementation would be preferable.

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor Dr. Dwight Makaroff for the continuous support of my Masters research - for his patience, motivation, sincerity and enthusiasm. His guidance helped me during the tough times of my research and writing of this thesis. I would also like to convey my heartfelt thanks to my Co-Supervisor Dr. Kevin Schneider for guiding me towards the right path for all things related to Software Engineering.

Besides my advisor, I would like to thank the rest of my thesis committee: Dr. Matthew Links, Dr. Nathaniel Osgood and Dr. Steve Robinson, for their encouragement, insightful comments and piercing questions.

I thank my fellow labmates in DISCUS Lab: Mohammed Rashid Choudhury, Habib Sabiu, Owolabi Adekoya and Tunde Olabenjo for the stimulating discussions, for the late night coffees, and for all the fun we have had in the last two and a half years. I am especially indebted to Imran Ahmed from IMG Lab for his vociferous support through and through.

Last but not the least, I would like to thank my family: my parents Jamal Uddin Ahmed and Jobeda Begum, for raising me right and guiding me towards the right path.

For the oppressed, the unheard, the forgotten. May their prayers be answered, in this life, or the next.

Contents

List of Tables

List of Figures

List of Abbreviations

GWAS	Genome Wide Association Study
API	Application programming interface
CPU	Central Processing Unit
GPU	Graphics Processing Unit
DNA	Deoxyribonucleic acid
SNP	Single nucleotide polymorphism
LMM	Linear Mixed Model
MLM	Mixed Linear Model
TASSEL	Trait Analysis by aSSociation, Evolution and Linkage
FaST-LMM	Factored Spectrally Transformed Linear Mixed Models
RF	Random Forest
GUI	Graphical User Interface
RDD	Resilient Distributed Dataset
DAG	Directed Acyclic Graph

1 Introduction

1.1 Motivation

A *Genome-Wide Association Study* (GWAS) is a method of finding genetic variations associated with particular traits by rapidly scanning polymorphisms revealed using genetic markers - DNA sequences with known locations on chromosomes - over the entire sets of DNA or genomes. The goal of these activities is to discover relationships between genetic variants and a specific attribute or trait, (e.g. a disease). There are various GWAS algorithms which can be performed over a sequence of DNA. These algorithms can be expressed as filters in a pipeline.

A *pipeline* is a connected set of operations called filters, through which data is transformed and produced as output. In a GWAS pipeline, the input is a dataset containing marker information from individuals, along with covariates and related files. Each filter represents a specific genome wide association algorithm. These filters transform the input dataset so that another algorithm can work with the transformed output. In this way, the GWAS pipeline transforms a genomic dataset to the desired output.

Several bioinformatics tools have implemented GWAS pipelines that use instances of GWAS algorithms. These tools can be broadly divided in two categories – those supporting sequential operation and those supporting parallel operation. Although researchers and developers have implemented several optimizations [15, 56, 80] to improve GWAS performance, sequential operation is still too slow for the massive volume of datasets required. In recent years, there have been numerous research works focusing on parallel solutions for GWAS. These solutions range from using multiple processors [48, 63], using GPUs [5, 25, 65], using both CPUs and GPUs [24] and using clusters of machines [26, 50]. The solutions demonstrate that it is possible to achieve speedup using parallel execution and adding more resources. For example, Luecke *et al.* [48] was able to achieve 36x speed improvment for epistasis detection using parallel MPI with Intel Xeon Phi coprocessor. Wienbrandt *et al.* [25] was able to parallellize epistasis detection using GPU and achieved four orders-of-magnitude speedup.

The variety of options can be confusing for a researcher who just wants to perform a GWAS and retrieve results at the earliest time possible. Furthermore, each tool specifies its own structure to which researchers have to adhere in order to produce results properly. For example, a genomic dataset can be represented in various formats, such as PLINK [58], Variant Call Format (VCF) [10], HapMap (HMP) [21] and FASTA [55]. Not all of these formats are supported by every tool. For these reasons, researchers resort to using multiple tools in order to produce results with which they can work.

In addition to the problem of choosing the right tool, GWAS researchers face performance, usability and flexibility problems. There are many operations in the domain of GWAS, and picking the right tool for a desired operation with the right performance can be difficult. The most appropriate tool for a GWAS operation can be lacking in performance, making it difficult for researchers to complete the operation quickly. Moreover, a GWAS tool with the right operation can be difficult to use if the interface is built poorly. Researchers often have to work with GWAS applications having confusing interfaces. Furthermore, researchers may have to resort to using multiple tools with different configurations in order to execute a batch of GWAS operations, which makes it less flexible for researchers to complete these steps of operations. This fragmentation of appropriate tools in the GWAS domain makes it quite difficult for researchers to plan and execute a GWAS workflow.

Due to these difficulties, there is a need for a generic solution for GWAS applications. This solution should be independent of structural constraints such as data format, input and output type, fixed confugurations as well as easily deployed in any environment. Furthermore, this solution can be configured to achieve the maximum performance benefit dependent on the deployment resources available.

1.2 Potential Solution

As described in the previous section, researchers have a lot of choice when it comes to conducting GWAS. Although these choices give researchers freedom to analyze DNA sequences, a specific scenario might need a specific GWAS workflow. For example, a workflow where researchers conduct epistasis detection can be different from another workflow of heritability estimation. Instead of building various complete workflows for specific scenarios, scientists can benefit from a modular architecture.

In a modular architecture, software implementations are divided into modules, and each module performs a specific set of tasks. If we model a GWAS algorithm as a set of tasks, we can transform the algorithm into a module. These modules can be independent of other modules, making them excellent candidates for use in a pipeline.

Once GWAS researchers model their tools as loosely dependent modules, they can easily pick and use whichever algorithm they want to implement on a dataset, providing that they meet the interface constraints and have equivalent pre/post conditions. Furthermore, parallelization techniques can also be modularized in this way, making it easier for researchers to pick the right parallelization tool for specific tasks. These potential solutions can be brought under an umbrella using a common architecture.

A number of potential parallel architectures such as MapReduce and MPI can be used to build modular parallel tools. MapReduce [12] is a parallel programming model that is used to process large volumes of data. It is a combination of two specific operations - *map* and *reduce*. Mapping means restructuring a dataset into different partitions and executing a common operation on each partition at the same time. The reduce operation takes the output of map operations as input and reduces the output to a combined output format

from all reducers. MPI (Message Passing Interface), on the other hand, is a standard [20] for the message passing model, the goal of which is to pass messages between processors in order to perform a task. This standard contains protocols and specifications of how parallel processing can be achieved by passing tasks as messages between processors. These architectures are just two examples that can enable parallelization and help developers increase performance as well as reduce response time for GWAS researchers.

1.3 Thesis Statement

The goal of this thesis is to show that a modular pipeline architecture for GWAS applications having a) interfaces for modular development, b) a shared data cache and c) the ability to leverage parallel computing environments such as clusters for increased performance and reduced latency can be built and maintained in a straightforward manner by application developers and used by biologists and other scientific researchers.

The proposed pipeline architecture can be built with multiple modules having limited input and output constraints, such as data format and size. Furthermore, the proposed architecture has a modular interface for parallel frameworks with which developers can build parallel modules and expose an API for the pipeline. The GWAS modules can then access the API to process data in parallel. Additionally, to ensure data integrity during various stages in this pipeline, a modular interface for a shared data cache is proposed and implemented such that developers can build modules to store and retrieve data . Together, these components will form the modular pipeline architecture for GWAS applications.

To demonstrate that the architecture works in real world situations, two case studies are conducted, where two GWAS applications are modified to implement the proposed architecture. The first case study focuses on modifying a genomic analysis pipeline called TASSEL. Details of the TASSEL package are given in a background section. A TASSEL plugin is modified which uses an implementation of modular interface for parallel frameworks. A shared data cache is also implemented using the proposed modular interface. The second case study modifies and implements the proposed architecture on Factored Spectrally Transformed Linear Mixed Models (FaST-LMM), a GWAS application. Details of FaST-LMM are also provided in the background section. Similar to the first case study, a FaST-LMM plugin is modified to use modular implementations of parallel frameworks as well as a shared data cache. Together, these implementations provide evidence that the proposed architecture increases performance for GWAS applications.

Finally, an architectural evaluation will be conducted using feedback from domain experts such as software developers and bioinformaticians. Their feedback will provide insight on the usefulness and configurability of the architecture and its implementation.

1.4 Contributions

This thesis contributes to the area of GWAS and distributed computing. The four major contributions of this thesis are as follows.

- A modular pipeline architecture for GWAS applications having interfaces for parallel processing frameworks and shared data caches. The interfaces enable GWAS applications to leverage parallel computing environments and access data from a cache, which can be faster than a traditional disk read.
- Two prototype implementations of the proposed architecture, using two popular GWAS applications: TASSEL and FaST-LMM. The prototypes used two parallel processing frameworks: Apache Spark and Dask, as well as a key-value pair based cache store cluster called Redis.
- Demonstration of performance benefits using the prototype implementations. Various experiments were conducted to provide evidence that the implementation of the proposed architecture results in increased performance.
- Demonstration of architectural quality using evaluation feedback from domain experts. The feedback from experts demonstrated that the architecture is useful and beneficial for developers and end users.

The first contribution paves the way for building modular GWAS applications which can harness the power of distributed computing using modular components. The second contribution demonstrates an example implementation of the architecture, and can be considered as a guideline for developers to build modular GWAS applications. The third and fourth contributions assert that the architecture and its implementations are indeed useful for developers building GWAS applications and biologists conducting genomic analyses.

1.5 Outline

The rest of this thesis is laid out as follows. Chapter 2 covers the background and related work for GWAS, parallel data processing frameworks and the modular pipeline pattern. Chapter 3 describes the system design of the proposed architecture. Chapter 4 describes the case studies, experimental setup of the case studies along with the design of the architectural evaluation. Chapter 5 describes the results of the experiments and evaluation. Chapter 6 concludes the thesis and outlines possible future work.

2 Background and Related Work

2.1 Genome Wide Association Study

2.1.1 Definitions

Before explaining GWAS in detail, here are some definitions of common terms that are used throughout the explanation of GWAS.

DNA Deoxyribonucleic acid [1] is a hereditary material found in all carbon-based organisms such as humans. It is a double helix assembled by base pairs and located in the cell nucleus. DNA stores information as a code of four chemical bases – A, C, G and T. The change in order of these bases determine how an organism is formed.

Genome The Genome [54] is the genetic material of an organism, consisting of a complete set of DNA and its genes. A gene is made of DNA which provide instructions to create protein. Although most genes are the same in individuals of a species, a few genes are different, which contributes to the variation of features between individuals.

Chromosome A chromosome is a DNA molecule which carries genetic information as a single unit that is transferred to the next generation. It is found in the nucleus of living cells and have a thread-like structure.

Locus A locus is a fixed physical position of a gene or a SNP on a chromosome. Several locus elements together are called a *loci*.

SNP A single nucleotide polymorphism (SNP)[38] is the occurrence of genetic variation among individuals in a single base. At a specific position in a genome, a SNP represents variation in a nucleotide, which is a single DNA building block. SNPs can potentially act as genetic markers for genes which are associated with diseases, which in turn can help researchers identify cause of those diseases.

Allele An allele is an expression of a gene or a SNP that may differ due to mutation. Diploid organisms have two copies of each chromosome - one from their father and one from their mother. The two copies are on separate but similar chromosomes. When these copies change due to mutation, they become alleles.

Heritability Estimation Heritability estimation is a statistical analysis of heritability, which is an estimation of variation in a trait due to variation in genetic factors of individuals and the variations due to the environment [75].

LMM Linear Mixed Model extends the generalized linear model to allow for random effects as well as fixed effects. A linear model describes an equation in which a continuous response variable acts as a function of a single or multiple predictor variables. In a generalized linear model, the response variable can be modelled by a function of explanatory variables, in addition to an error term. In genetics, an LMM is used for measuring genetic similarity to estimate probabilities of individuals sharing alleles which can be related to diseases or traits [46].

Genetic Marker A genetic marker [52] is a sequence of DNA which is used to identify a chromosome or locate genes on a genetic map. The arrangement of these markers on a chromosome is illustrated by genetic and physical maps. Physical maps illustrate the alignment of DNA sequences, while genetic maps illustrate the recombination frequency between molecular markers.

2.1.2 Background

The procedure of conducting a genome-wide association study is as follows. Two groups of participants are used for a study. The participants of the first group have the trait or the condition the researchers are trying to associate with a genotype. The participants of the second group do not have the trait/condition. First, the researchers acquire DNA from each participant. Next, the cells obtained from the participants are purified to extract each participant's complete set of DNA (or genome). After that, the extracted DNA samples are placed on small chips and then scanned with automated machines. The machines analyze each participant's DNA for key *markers* of genetic variations [7]. These key markers are called single nucleotide polymorphisms, or SNPs.

After the key markers of genetic variations are analyzed, the researchers compare these variations based on the frequency of the variations in the control groups. If a set of genetic variations are observed to be more frequent in the participants of control group with the trait/condition compared to the participants of group without the trait/condition, then the genetic variations are said to be correlated with the condition. From this correlation, the researchers can potentially point to the regions of the genome which contributes directly or indirectly towards the formation of the trait/condition.

GWAS has mostly been used by researchers for two reasons – to detect and identify diseases and to uncover genetic architecture. Various techniques such as an efficient mixed-model association [33], a multi-locus mixed linear model [73], a unified mixed-model [76] and a compressed mixed linear model [79] have been developed to investigate characteristics of animals and plants.

GWAS has contributed to the extensive success in detecting genetic architecture of important traits in plants and crops. Research on plants such as canola [42, 43], rice [29] and maize [44] has been carried out

that identified genes related to specific traits and the underlying genetic architecture controlling these traits. Identifying genes related to these traits can lead to increased production of crops and identifying regions for mutagenesis to generate new variations.

GWAS also has a crucial role in detecting cause of diseases in animals and plants. Researchers have been conducting GWAS to identify new genetic associations and finding genetic variations that causes common and complex plant [47, 72, 81] and human [2, 4, 11, 17, 34, 35, 36] diseases. Three independent GWAS studies in 2005 [35] discovered that a variation in the gene for complement factor H is responsible for a common form of blindness. This gene produces a protein which regulates inflammation. Researchers also achieved further success using GWAS for identifying genetic variations contributing to obesity [71], cardiovascular disease [70], Parkinson's disease [64], type 2 diabetes [18] and Crohn's disease [17].

The simplest approach in performing a genetic association study is to analyze a single trait - a test of association between a single SNP and a single trait. This approach studies the effect of single variants with respect to a trait. However, this analysis approach does not exploit information inside related traits. Furthermore, as most traits of interest are dependent on multiple factors, there might be a danger of biased results due to low frequency of association between causal loci [37].

Due to these drawbacks, researchers often conduct multivariate analysis - a joint analysis of multiple traits which may be potentially correlated. Correlation between traits can occur for two reasons [68]. First, the traits in question can be genetically correlated and the effects of SNPs can be distributed across traits. Second, the estimation error of the effects of SNPs can also be correlated across traits. If we take these correlations into account, we can expect better results compared with analysis using single traits.

2.1.3 GWAS Tools

Various tools have been developed by researchers to perform GWAS efficiently. The remainder of this section contains a summary of GWAS tools that are relevant to this work. These tools have implementations of various GWAS algorithms using optimizations to speed up the overall process. The tools and their optimizations are detailed as follows.

PLINK

PLINK [58] is an open-source GWAS tool set. The goal of PLINK is to manipulate and analyze huge datasets containing millions of genetic markers genotyped for thousands of individuals in a rapid fashion. In addition to efficient computation of basic genomic analysis, PLINK brings on an innovative solution for covering whole-genome data. PLINK provides a simple approach to handle large GWAS data sets, produce summary statistics, perform various standard association tests on huge datasets efficiently and assess rare genetic variations using common SNP panels. According to the developer of this tool, loading, filtering and performing association analysis for a data set of 100,000 SNPs for 350 individuals take around 10 seconds with PLINK [58]. PLINK is written in C and C++. A command line interface (CLI) is used to access PLINK

functionalities.

FaST-LMM

FaST-LMM (Factored Spectrally Transformed Linear Mixed Models) is a set of tools [74] derived from an algorithm for GWAS [45] which scales linearly in both runtime and memory usage according to the developers. This algorithm can perform GWAS prediction and heritability estimation on large data sets. The researchers claim that FaST-LMM reduces the computational expense of Linear Mixed Model (LMM) calculations by using the FaST-LMM algorithm. The researchers also demonstrated that FaST-LMM can work with datasets with over one million samples. It is written in Python and has Linux and Windows deployments.

Internally, FaST-LMM uses both multithreaded and MapReduce approaches to perform calculations. It uses runners, which are modules capable of running GWAS operations in multiple environments, to run distributed tasks in single core and multicore machines. However, the MapReduce approach defined in FaST-LMM can only scale to a single machine in its current form. This is because the map operations are performed using Python generators, and each generator's next step is determined by its current state, which prevents the generator operations from being truly parallel. Moreover, FaST-LMM uses nested MapReduce to distribute tasks which increases complexity for task schedulers, as the schedulers have to track and manage all the nested mappers and reducers.

TASSEL

TASSEL (Trait Analysis by aSSociation, Evolution and Linkage) [22] is a Genotyping by Sequencing (GBS) analysis pipeline written in Java. It was built with two objectives in mind - running analytics on large datasets and low memory footprint. The developers of TASSEL claim that it can run from small to extremely large studies [66, 78, 30, 69] with modest computing resources. The software consists of a number of analysis and utility packages. The analysis packages run the main analytics tasks, while the utility packages provide support for file loading and multithreading parallelism.

The architecture of TASSEL is pipeline-based, and each stage of the pipeline can run a number of processes using multithreading tools concurrently. Although TASSEL utilizes the multithreading tools provided by Java, the internal operations of plugins are mostly executed on a single thread, resulting in sequential operation. Furthermore, the analysis modules of TASSEL are entirely dependent on Java thread classes, which do not follow the modular paradigm. Consequently, it becomes hard to modify the tasks for running in a distributed environment.

2.2 Parallel Data Processing Frameworks

The most basic form of modern parallel processing involves utilizing multiple processing cores of a single machine [32]. Modern programming language implementations use threads to achieve logical parallelism,

e.g. POSIX threads in C/C++, Java Service Executor and .NET Task Parallel Library. However, this level of parallelism is limited to a single machine. Multiple execution cores in the CPU/GPU are required for physical parallelism.

In the scientific and engineering community, parallel processing is achieved with the MPI (Message Passing Interface) standard [20]. The MPI standard contains protocols and specifications of how parallel processing can be achieved by passing tasks as messages between processors. This standard is based on the message passing model in which processes communicate with other processes by sending and receiving messages. Implementations of the MPI standard, such as Open MPI [19], MPICH [27] and TMP-MPI [60] are regularly used by scientists and researchers to process data in parallel.

Although implementations of MPI work well for restricted forms of distributed computing, processing Big Data requires efficient utilization of multicore CPUs distributed across multiple independent machines. These machines form a cluster to take advantage of all processing, storage and network resources for parallel operations. Each machine in a cluster is called a *Node*. Initializing a cluster and then distributing and managing tasks in a cluster can be complicated. Parallel Data Processing Frameworks handle these complications by running task schedulers, handling input and output of parallel tasks and managing nodes in a cluster. Furthermore, these frameworks ensure that data is properly parallelized and distributed across nodes in a cluster. Examples of these frameworks and their capabilities of handling complications will be described in the following subsections.

Several parallel data processing frameworks are available in the research community and in industry, such as Spark [77], Hadoop [57] and Dask [59]. Some of these frameworks are built with the intention of handling generic big data, others are more focused on genomic data in particular. Each of these frameworks implements a distributed architecture to execute parallel tasks in multiple machines. A summary of some of the most relevant of these parallel data processing frameworks are given below.

2.2.1 Apache Hadoop

Apache Hadoop [57] is an open-source Big Data framework that handles large volumes of data efficiently using distributed computing across clusters of machines. Hadoop can be used on a large number of machines and provides fault tolerance at the application layer.

Apache Hadoop uses MapReduce programming model to process large volumes of data. A task scheduler is required to allocate data partitions and operations to various worker nodes. A job scheduler handles the scheduling of jobs and aims to utilize resources efficiently as well as reduce response time.

In addition to the MapReduce paradigm, Apache Hadoop uses a storage system called Hadoop Distributed File System (HDFS) to store large volumes of data efficiently [62]. In HDFS, files are split into large blocks and distributed across nodes of a cluster. HDFS implements data replication, which allows the same blocks to exist in multiple nodes. Due to data replication, Hadoop can use data locality (the ability to initiate computation close to the source of data) to process data in the same node as the replica of the partition

required for the task. This ensures faster execution time compared to a parallel data processing system requiring heavy use of a high speed network to transfer input data between nodes. However, data transfer is needed during shuffle and reduce phase of Hadoop execution, as *intermediate* data that is stored on the local file system of individual nodes is transferred from mappers to reducers.

The Apache Hadoop framework is written primarily in Java; in addition, there are a few modules written in C. It requires the Java Runtime Environment (JRE) to work. Although most MapReduce tasks are written in Java, Hadoop Streaming allows programming languages such as C, C++ and Python [13, 41] to stream input data to the framework as a front end interface.

2.2.2 Apache Spark

Apache Spark [77] is an open source big data framework that uses Resilient Distributed Datasets (RDDs) to persist intermediary results in node memory and a Directed Acyclic Graph (DAG) scheduler to process large volumes of data split into partitions across clusters of machines. The DAG scheduler is a stage-oriented scheduling method which transforms a logical execution plan into a physical one. An RDD is a wrapper around a collection of elements that are fault tolerant; multiple RDDs can be processed in parallel. An RDD can be persisted in-memory so that the data in the RDD as well as intermediate results can be saved in RAM, reducing disk load latency and computational overhead. Spark achieves fault tolerance of an RDD by replicating data to multiple Spark executors as well as remembering the lineage of the deterministic operations. Similar to Hadoop, Spark works on clusters of machines, provides fault tolerance and can utilize HDFS to process distributed datasets.

Although Spark provides *Datasets* as an RDD abstraction, the proposed architectural implementation uses RDD only. A Dataset is a structured collection of distributed data, which uses an optimized execution engine. The prerequisite of converting a plain dataset into a Spark Dataset is to define the structure of the dataset. However, the implementation of the proposed architecture does not provide the input dataset directly to Spark. Instead, Spark takes in a representation of the input dataset in the form of rows of objects. Therefore, RDD is used instead of Dataset to preserve the ability to work with any form of unstructured data.

Apache Spark uses the same MapReduce programming model that Apache Hadoop does. However, the difference between Hadoop and Spark is in the way Spark handles the input datasets and intermediate output datasets. Spark uses RDD to abstract away the input datasets. Furthermore, an RDD uses memory to store data, and spills to disk when memory is full. Spark also has built in RDD operations which makes it easier to operate on large datasets. Two types of operations are supported by RDD - *transformations* and *actions*. Transformations create a new dataset from an existing one, whereas actions run computations on the dataset and return a value to the driver program. One of the most common RDD operations is *map*, which transforms each element of a dataset through a function and returns a new RDD with the transformed values. Another common RDD action is *reduce*, which uses a function to aggregrate all elements of an RDD and returns a

final value to the driver program. The *map* operation can be useful for GWAS applications, as it can help transform GWAS dataset using a specified algorithm.

The core of Spark is written in Scala. This core exposes an application programming interface (API) based on the RDD abstraction, which can be accessed with programming languages such as Java,[1] Python,[2] Scala[3] and R.[4] The interface of Spark also allows to work it with other languages such as SQL.

2.2.3 Dask

Dask [59] is an open source parallel computing library written in Python. Dask implements dynamic task scheduling that accepts an interactive computational workload. It also provides collection wrappers which can work in distributed environment and run with dynamic task schedulers.

A dynamic task scheduler handles distributed computing for Dask. Depending on the runtime environment, the task scheduler can be of multiple types - a thread pool scheduler, a process pool scheduler, a synchronous scheduler or a distributed scheduler. The implemented scheduler processes spawns and manages several worker processes distributed across multiple nodes in a cluster. The parallel computations in Dask are represented by directed acyclic task graphs, similar to the Spark model of computation. The task scheduler executes the task graphs to leverage parallelism.

Dask user interfaces are divided into two levels - high and low. The high level interface contains parallel collections implemented over subset of existing Python packages such as Numpy, Pandas and Scikit-Learn. The low level interface enables delayed parallel execution of individual functions.

Although Dask can process data in parallel similar to Apache Hadoop and Apache Spark, there are a few differences. Dask is written in Python and can be seen as a component in the Python ecosystem. Furthermore, Dask is less dependent on the MapReduce paradigm than Hadoop or Spark, instead focusing on generic task scheduling.

2.2.4 Hail

Hail[5] is an open source analytic framework based on Apache Spark for genomic data analysis. It enables pipeline processing to import, process and export a variety of genomic data.

Hail is written in Python and uses Apache Spark to build and run its distributed algorithms in clusters of machines. In addition, Hail implements a specific domain language interface called *Expressions* for expressing pipeline flow. As Hail is integrated with Spark, it can leverage SQL processing and machine learning algorithms as well.

[1]Spark Java API - `https://spark.apache.org/docs/latest/api/java/index.html`
[2]Spark Python API - `https://spark.apache.org/docs/latest/api/python/index.html`
[3]Spark Scala API - `https://spark.apache.org/docs/latest/api/scala/index.html`
[4]Spark R Api - `https://spark.apache.org/docs/latest/api/R/index.html`
[5]Hail - scalable genomic data analysis. `https://github.com/hail-is/hail`.

The query functions and Expression language of Hail help users build pipelines with which they can process genomic data in parallel. Unfortunately, the users are entirely dependent on the type of input that Hail supports. Moreover, building plugins for Hail requires implementing its interface, which may not match how the user wants to process a genomic dataset. Furthermore, although Hail supports caching plugin specific data, it has no support for caching data which can be used at any stage of a pipeline.

2.2.5 VariantSpark

VariantSpark [50] is another open source toolkit for conducting GWAS based on Apache Spark. VariantSpark uses Spark as its analysis engine in a similar fashion to the mechanisms used by Hail. It is more specifically tuned however to analyze extremely wide GWAS datasets; it uses the random forest (RF) model for SNP analysis. RF is a machine learning algorithm which constructs decision trees during training and provides regression of individual trees as predictions. VariantSpark is specifically built for datasets with large samples and wide feature vector per samples.

VariantSpark uses RF because it can efficiently process extremely wide GWAS datasets by building a random forest from transposed data. Using RF, SNPs are ranked in accordance to their predictive score. RF is built from a transposed representation of a dataset, allowing efficient operation on GWAS datasets with large number of columns. The RF implementation in VariantSpark splits dataset per sample and within samples as well, while maintaining data cohesion. VariantSpark uses VCF and CSV files as input.

Although VariantSpark has fewer features than Hail, it can efficiently process big GWAS datasets. Hail supports more input formats, however, than VariantSpark does. Unfortunately, VariantSpark does not extend the pipeline functionality that Spark has, making it less likely to work with individual plugins.

2.3 Modular Pipeline Software Engineering Patterns

2.3.1 Modules and Pipelines

The concept of a module originates from the modular programming paradigm [53], which has been a widely used technique for several decades in Software Engineering. This paradigm states that the functionality of a program should be separated into loosely coupled and interchangeable components. The purpose of a module is to encapsulate and implement a particular functionality. This functionality can impose *pre-conditions* and *post-conditions*. Preconditions state the constraints that must be met before the module is performed, while postconditions describe the constraints corresponding to the output state after the operation is completed. A module should have an *interface* which expresses the services provided and consumed by the module. Furthermore, the interface should enable other modules to plug in and operate together.

The aim of modular programming is to control the complexity of a large system by dividing it into modules. In this paradigm, a system is decomposed into several modules. These modules interact with each

other to consume input and produce output. In addition, modular programming encourages loose coupling and high cohesion. Modules are loosely-coupled when they have little or no dependency on other modules. This minimal dependency of modules results in streamlined responsibilities and individual functionalities for each module, resulting in high cohesion. Finally, modules are built for reusability - they are designed in such a way that they can be interchanged to produce different outputs.

In software engineering, a pipeline is an arrangement of processing elements such as modules in such a way that the produced output of one element can be consumed by another element. An initial element begins a chain of a pipeline and takes in data as input. Processed data moves from one element to another in either stream or batch.

A pipeline can be viewed as a decomposition of a sequential process into sub-processes. These sub-processes can be implemented as modules and operate with other modules in a pipeline or in parallel. As modules are independent and interchangeable, they can be used at any stage of a pipeline, provided that they support the output of the previous module as an input.

2.3.2 Related Work on Modular Pipeline Architecture

In the recent years, a number of research projects have been conducted to build systems using modular pipeline architecture in different domains. In this section, a sample of research on modular pipeline architecture from different domains will be described first. Next, research on modular pipeline architecture belonging specifically to bioinformatics domain will be described. After that, a number of modern scientific workflows having modular and pipeline design will be introduced. Finally, an explanation will be given on how this thesis work will differ from these existing works.

The research works discussed below exploits two prevalent principles - they use a modular approach and pipeline design. Modularity is used as means to solve complex problems. For example, Torreno *et al.* [67] used modular out-of-core design to solve memory overflow problem, and Hutchison *et al.* [28] used modular workflow to develop a parallel message-passing architecture for their tool. In addition to modularity, pipelines also play an important role in solving the research problems. Utilizing a pipeline design helped render graphics [49], gather motion data [16] as well as process DNA sequence datasets [67], microarray datasets [14] and microarray images [61]. Moreover, other research [28, 40, 8] parallellized pipeline execution for improved performance. This thesis will extend the research works described below by exploiting the modular and pipeline principles.

Mobius *et al.* [49] created an open-source modular framework called *OpenFlipper* which can process and visualize complex geometric models. The software is built with plugin components which are modular in nature. The software also includes an advanced rendering pipeline which can render high-end graphics.

Flam *et al.* [16] implemented a flexible and modular architecture in a system called *OpenMoCap*, which is a real time motion capture system. A modular pipeline is used in this software to gather motion data. The software also allows new modules to be added in the pipeline for additional processing and output generation.

Another example of modular pipeline architecture implementation is the research conducted by Law *et al.* [40], in which an open-source visualization application called *ParaView* was built. The application has a modular architecture and uses parallel visualization pipeline. It was demonstrated to handle ocean temperature and salinity data. Two filters in sequence were used to analyze the dataset - geometry and filter.

Kopczynski *et al.* [9], on the other hand, built a modular framework named PEAX which can perform automatic peak extraction from raw data matrix for clustering and classification purpose. PEAX is built with a pipeline architecture in which each step performs a subtask using modules. The framework provides an interface using which new modules can be created. Using this modular pipeline architecture, the researchers claim that the software is able to automatically output peak list within a few seconds.

In addition to these modular pipeline architecture implementations, there are a few works which focused on implementing the architecture in the bioinformatics domain. Icay *et al.* [31] built an RNA workflow called SePIA that focused on solving complex and large-scale sequencing experiments by building a computational workflow. The workflow utilizes modular design and pipeline architecture. It contains modular and independent components as well as pipeline infrastructure to connect the components as a workflow. Another research on bioinformatics workflow pipelines was conducted by Hutchison *et al.* [28], in which they built a tool named *EasyProt*. This tool utilizes a parallel message-passing architecture to develop modular workflows in computational biology using cloud based resources. The system is able to parallelize modular components by using thread based parallelism to expand execution across multiple machines. Torreno *et al.* [67] focused on solving bottlenecks in processing large biological sequence datasets using modular designs. The solution was to modularize a pairwise genome comparison application named GECKO. The strategy for the solution was to use modular out-of-core strategy to use secondary storage for avoiding memory overflow. The additional improvement was to modularize GECKO process to store intermediate results to disk. The result of the modularization was reduction of computation complexity and memory consumption. Mura *et al.* [8] built a modular framework called *PaPy* which utilized parallel pipelines to flexibly execute bioinformatics workflows. The workflow utilizes modular and reusable components, and multiple workflows can be distributed in parallel.

Two additional research projects need to be mentioned that focus on modularizing microarray processing. Dondrup *et al.* [14] built a software named *EMMA*, which can store and analyze large microarray datasets efficiently. EMMA has a modular architecture as well as pipeline for data processing. The system can also be extended to support additional data sources. Another research by Samavi *et al.* [61] focused on processing DNA microarray images using a modular and scalable architecture. They proposed and implemented a modular pipeline which can process images in batches. The proposed architecture was implemented on FPGAs and was both scalable and modular for VLSI implementation.

Along with research done on implementing modular pipeline architecture itself, there are three scientific workflow applications which are relevant to this thesis because of their modular pipelined approach. The first application is called *Galaxy* [23], which is a web-based genomic workbench. The benefit of using Galaxy is

tracking and managing data provenance as well as interactive computational analysis. Next, *Kepler* [3] is a scientific workflow system which can capture scientific workflows and can run workflow design with minimal effort. Finally, *Taverna* [51] is a bioinformatics workflow tool which contains a graphical workbench tool to create and run bioinformatics experiments in the form of a workflow. The similarity between these three workflow tools is that they enable the users to create a pipeline or a workflow using modular components to conduct scientific analysis such as working with genomic data. The difference between these workflow tools and the modular pipeline implementations described earlier is that the workflows provide a higher level architecture to work with, whereas the modular pipeline implementations are more specific in nature.

All of the existing works discussed above have many of the characteristics that the proposed architecture is aiming to have. However, none of these works specifically target modularizing and implementing pipelines for GWAS operations. Furthermore, although some research did focus on parallellizing execution [28, 40, 8], none of them provide a common interface for parallel processing framworks specifically intended for GWAS operations, which can not only modularize parallel operations, but also can give end users the choice of using their preferred parallel processing frameworks.

2.3.3 Summary

In conclusion, this chapter discusses the background and research work related to GWAS parallelization. First, the definition, role and contributions of GWAS was discussed. Next, a number of relevant GWAS tools, their architecture, benefits and drawbacks were discussed. After that, parallel processing is explained and a number of relevant parallel processing frameworks were described along with their architecture, benefits and drawbacks. Next, modules and pipelines were introduced and defined. Finally, research work related to modular pipeline and its relevance to the GWAS pipeline architecture designed and evaluated in this thesis was discussed.

3 System Design and Architecture

In this chapter, the proposed architecture and its system design is laid out. First, modular pipeline and parallelization are explained in the context of software engineering with examples. Next, a few candidate applications' architectural designs are analyzed and design flaws that can be fixed are identified. Based on this analysis, design considerations are established which must be followed for the proposed architecture. Finally, the proposed architecture is presented with explanation regarding the components and the interactions between them.

3.1 Existing System Design/Architectures of GWAS applications

To build a generalized architecture for GWAS applications, existing applications must be analyzed first. The chosen applications introduced in Chapter 2 can perform multiple GWAS operations on a variety of systems. The architecture of each of these applications will be described first, then the benefits and drawbacks of the said architectures will be evaluated.

3.1.1 Modular Pipeline and Parallelization in Software Engineering

In parallel computation, large problems are divided into smaller ones and solved simultaneously. Parallel computing allows a single task to be divided and executed in parallel on the available cores in a cluster, which should enable performance improvement that scales with the number of cores [6].

Due to its modular nature, a pipeline is a good candidate for implementing *data parallelization*. At any *stage* of pipeline processing, data may be divided into small pieces and executed using modules in parallel. This ensures that all the computing resources are fully utilized, and data processing time is decreased by a factor of the amount of parallelism.

Figure 3.1 demonstrates an example of a parallel pipeline. In this pipeline, the output of module 1 is split into several partitions. Each data partition is then executed by module 2 in parallel. Finally, the output of each parallel instance is concatenated and then used as input to module 3.

This approach of parallelization, however, has a problem. It is not clear how the partition and concatenation of data is handled. Before being used by a parallel module, input data must be separated by a module. Similarly, the resultant output data from each parallel module must be correctly reduced to a single output for further use or as a terminal output. In other words, this approach closely follows the MapReduce paradigm, the map and reduce tasks. It would make sense if the map and reduce tasks are encapsulated as

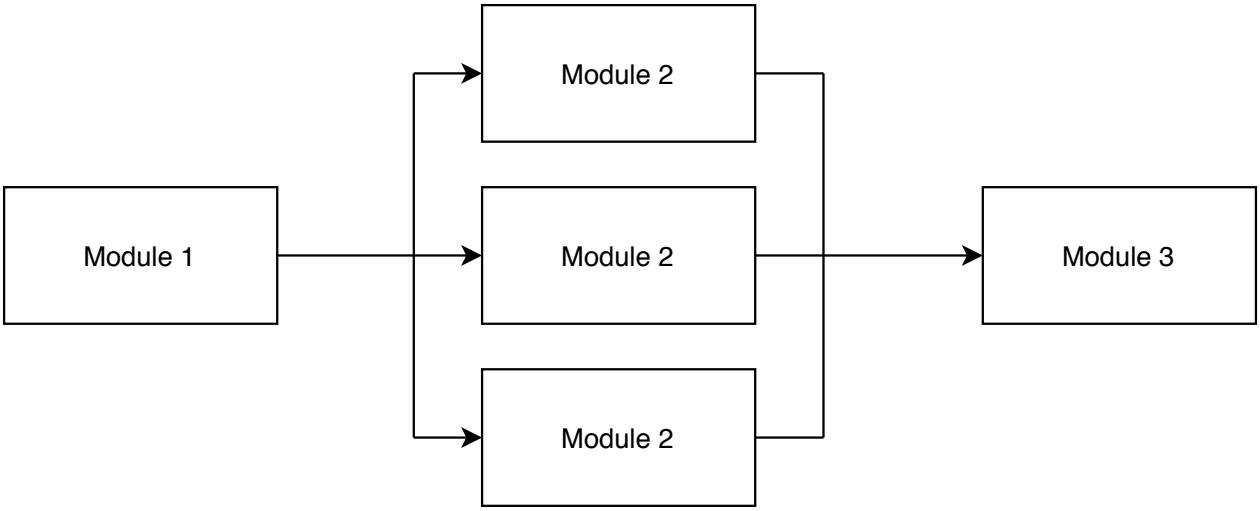


Figure 3.1: Example of a Parallel Pipeline

a module that handles the parallelization process.

Figure 3.2 demonstrates a solution to the problem. Instead of modeling data partitioning and assimilation as part of data transfer, it can be visualized as part of a module. In the pipeline displayed in Figure 3.2, Module 2 handles data partitioning, executing operations in parallel and data reduction. Other modules only require providing the input data to the parallel module which handles all the necessary operations.

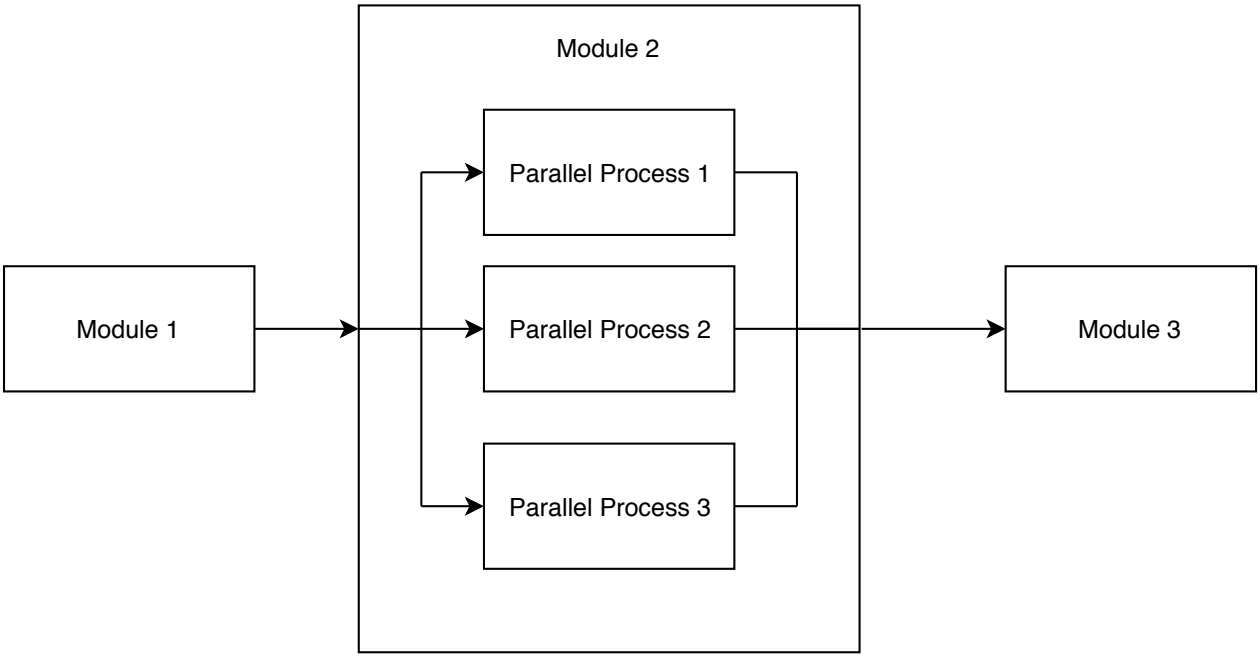


Figure 3.2: Example of a Modular Parallel Pipeline

This improved Modular Parallel Pipeline, however, still has a flaw - no separation of concerns for parallel processing. Parallel processing usually consists of operations such as data partitioning, task assignment and

result retrieval. These are the concerns that need to be separated from other tasks in a module. These operations can be executed with various tools and techniques. If we include any of these techniques as part of a module, then we need to modify the module every time we want to use a new tool or improve the technique. We need to remember that parallelization typically assumes an arbitrary process and arbitrary data - it does not matter what process is being executed in parallel. Therefore, a separate module can be constructed that will only handle parallelization and will depend on a provided processing function and data. The input of this module can be an input dataset and a function that will be applied on the input dataset in parallel. The module can optionally take configurations for tweaking parallelization as input.

The advantage of such an independent module for parallelization is that developers can utilize the module regardless of the type of data and the complexity of the parallel process. The module will only use a specific tool and technique to parallelize given data and apply the given function on each data partition. Each tool can be encapsulated in each module, and the most appropriate tool can be plugged in at any stage of a pipeline.

Figure 3.3 demonstrates a pipeline which uses such a parallel module design. Module 2 is providing the parallel module with the data to work on and the processing function which will be executed on each partition. The parallel module handles data partitioning, processing, shuffling and combining. After all the stages have been completed, the parallel module finally returns the processed output to Module 2. In this way, Module 2 does not need to know the details of parallel operations - it delegates the responsibility of parallelization to the parallel module.

3.1.2 FaST-LMM

As described in Section 2.1.3, FaST-LMM is a program which performs GWAS on a genome dataset of arbitrary size. The architecture of FaST-LMM is modular. The program contains a number of GWAS functions, such as single SNP GWAS, epistasis GWAS, heritability estimation and others abstracted behind an API. Each of these functions return statistical analysis of the GWAS function performed. Further details of the functions can be found in FaST-LMM API documentation.[1]

Each of the modular GWAS functions requires a runner to complete the operation. A runner can be a module which can run the GWAS operation in different environments. A runner module implements the *IRunner* interface, which defines a function to execute a distributable job. The runners provided with the FaST-LMM package can run the operations using a single core processor, multi-core processors or a cluster of machines.

The architecture of FaST-LMM has a few benefits such as modular reusability and system portability. The modular components of the program ensure ease of use for specific use case scenarios. The runner modules enable operation of the program in different system environments, and the program can handle arbitrary data size, depending only on the size of the worker machine being used with the runner.

[1]FaST-LMM API Documentation, `http://microsoftgenomics.github.io/FaST-LMM`

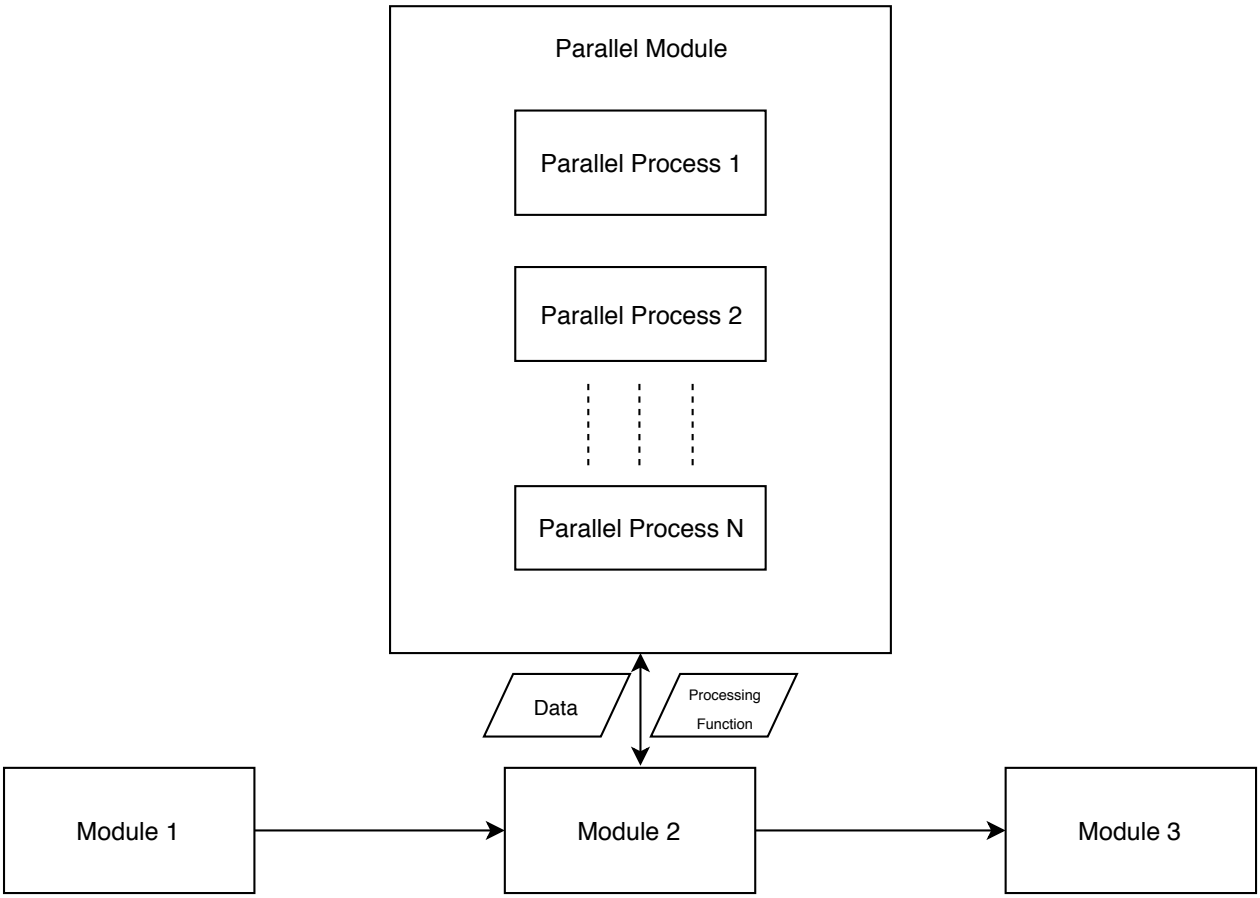


Figure 3.3: Example of a Modular Pipeline with Parallel Module

However, the architecture also has some shortcomings. Although the modules are reusable, they cannot be interconnected, as each module is meant to be a silo running one operation only. This means that the output of one module cannot be used as input to another module. This prevents ease of execution in scenarios where a genome dataset is transformed through multiple GWAS functions. Furthermore, there is no provision for caching supplementary and intermediate input files, which can considerably speed up data access.

3.1.3 TASSEL

As described in Section 2.1.3, TASSEL is a bioinformatics analysis pipeline. The main goal of TASSEL is to process raw sequence data into SNP genotypes. Like FaST-LMM, the architecture of TASSEL is also modular. The GWAS operations are encapsulated inside analysis modules, which are grouped together in an analysis package. All the analysis modules implement a common *Plugin* interface that helps developers create new modules easily. In addition to analysis modules, a set of utility modules are grouped together in a Utility package. This package includes statistical, mathematical and file loading modules.

TASSEL uses the batch sequential pipeline method as basis for its architecture. The modules can form a pipeline through which genomic data is transformed and a terminal output in a tab or comma delimited format is generated at the end of the pipeline. Usually, the file loader module loads the input dataset and supplementary file first, then passes on the loaded dataset to the next module in the pipeline, and so on. At each stage of the pipeline, TASSEL uses Java multithreading tools to run several processes concurrently, split and recombine process results, and push the output to the next stage of the pipeline.

The key benefits of the TASSEL architecture are its modular and pipeline form. TASSEL modules are reusable and replaceable in the pipeline, and the pipeline ensures simpler execution of GWAS operations based on data transformation.

Despite the use of Java multithreading tools, TASSEL is limited to using a single machine and not scalable to multiple machines in its current form. This is because the tasks are not distributable and the Java threads only work on a single machine natively. Furthermore, TASSEL lacks data caching facilities; this could speed up access during any stage of pipeline execution.

3.1.4 Hail

As described in Section 2.2.4, Hail runs on top of Apache Spark. Similar to TASSEL, Hail also has a pipeline based architecture due to its dependency on Spark. Hail uses a domain language interface called Expressions to construct the execution pipelines. This interface enables Hail users to perform complex GWAS operations.

The usage of domain specific language by Hail improves the usability of the program in complex situations. Furthermore, using distributed components of Spark, Hail can scale its operations to clusters of machines.

The architecture of Hail lacks in two aspects. First, there are no clear interfaces defined in Hail to create analysis modules, which means that developers cannot create new modules without substantial modifications to the original code. Finally, although Spark supports data caching through its use of Broadcast variables,

the caching process is too much dependent on Spark and it is not possible to use an arbitrary data cache without considerable modifications.

3.2 Design Considerations

Based on the analysis of existing GWAS applications in the previous section, the architectures of those applications are lacking in two desired features:

- Ability to scale performance in a modular way, and
- Ability to store and retrieve data with low latency in a modular way.

To mitigate these drawbacks and build a better architecture, the proposed architecture should follow some design considerations. These considerations point to the characteristics that improve the structure of the architecture and enable simpler implementation of an application. The design rules are described in the following subsections.

3.2.1 Scalability

Scalability is an attribute of a system in an environment that characterizes the ability of the system to handle increased workload when resources are added to the environment. An architecture can be called scalable if its implementation can process an increasing amount of data as resources are increased at the same rate. GWAS applications need to operate on datasets ranging from small to large to massive. A GWAS architecture should be able to scale to the demands of the application. In particular, the architecture should be able to handle a large volume of data efficiently in terms of scalable processing time. For example, if a GWAS application is executed in a cluster containing multiple machines, then the implementation of the architecture should be able to scale execution to all of the cluster machines in a linear or near-linear fashion. Furthermore, the architecture should ensure that larger programs are constructed and executed more efficiently.

3.2.2 Modifiability

As the proposed architecture is generic, a key design consideration is the ease of modifiability. Modifiability is a system attribute which defines the ease with which the system can be modified with minimum footprint on the system environment. As the modules are reusable and independent from each other, the architecture should ensure ease of modification without propagating changes to other elements. For example, if a GWAS application supports a module that utilizes parallel processing frameworks, the module should be modifiable such that changing the framework in the module does not affect any other operations or modules of the application. Furthermore, the loose coupling and high cohesion characteristics of modules can ensure that the combinations of these modules are highly interchangeable. Additionally, the developers can adapt the architecture to their needs for easier deployment.

3.2.3 Portability

Portability is a system characteristic which defines the ease with which the system can be executed on multiple environments, with minimal modification to the system. The architecture should allow for portability across different systems such as single core machines, multicore machines and clusters. Due to the variation of GWAS algorithms and datasets, different GWAS applications may have varying needs for system resources. Some applications may require more CPU power, other applications may depend on more GPU power. That is why the architecture should ensure that a GWAS algorithm is not constrained to run on a specific system. For example, the implementation of the architecture should have modules which enables a GWAS application to work with various systems such as a single machine, a cluster of machines, a GPU and so on.

3.2.4 Configurability

Configurability is a system attribute which defines the ease with which execution strategy can be changed and configured for the system. The architecture should allow the GWAS algorithms to be configured arbitrarily. Various GWAS algorithms can constitute a workflow, in which a genome dataset is transformed using GWAS algorithms. In this workflow, each GWAS algorithm can be encapsulated and parameterized so that they can be configured as required by the end user. Furthermore, the architecture should ensure that the workflow is configurable at any stage and provides output as desired.

3.2.5 Usability

Usability is an attribute of a system which defines the degree of ease with which the end users can achieve their objectives using the system.The architecture should ensure that the developers can build the components of the GWAS application easily and the end users can accomplish their desired workflow in a transparent way. For example, if the architecture supports using parallel processing frameworks to execute GWAS operations, then it should be easy for end users to pick and choose the framework to use as necessary. The system should be able to understand and learn, and it should be efficient to use.

3.3 Overall Architecture

Based on the architecture analyses and design considerations of the previous sections, the proposed architecture is presented in this section. Previous research on modular architecture [39, 53] was used as an inspiration and guideline.

The proposed architecture takes a generalized form, meaning that the architecture is less proscriptive about the implementation, so that developers can find use for it in many use cases. Thus, the intent is to allow flexibility in the use of the architecture. Two views are employed to describe the architecture - a modular decomposition view and a data flow view. The first view decomposes the architecture into modules

and displays the relation between them. The second view describes the interaction and data flow between the architectural elements.

Figure 3.4 demonstrates a modular decomposition view of the proposed architecture. At the centre of the architecture is a GWAS module. This module will encapsulate a set of operations representing a GWAS operation. The module can take in arbitrary data and produce arbitrary output. As the architecture is very generalized, it is up to the developer to extend the constraints of input and output of the GWAS module.

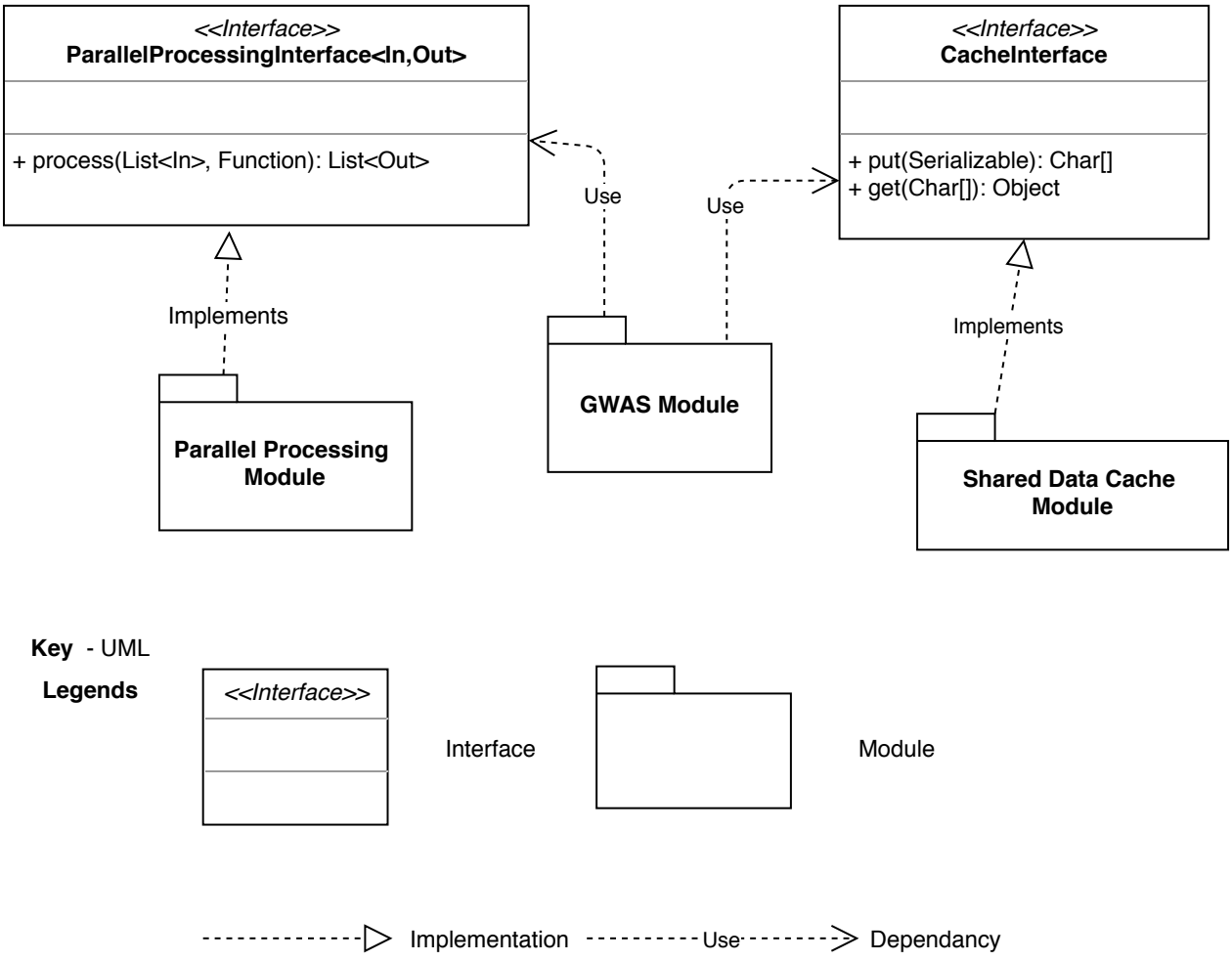


Figure 3.4: Modular Decomposition View of proposed architecture

The GWAS module is dependent on two additional modules – the Parallel Processing and the Shared Data Cache module. The Parallel Processing module processes a block of data that can be a list or a stream using a provided function. The module implements a Parallel Processing interface which defines two arbitrary data types and a process function. The process function takes in the input data in list format, along with the function which will run on the input and returns a transformed list dataset. The first data type *In*, which characterizes the input data, generalizes the type of the input in the input dataset. The second data type *Out*, which characterizes the output data, generalizes the type of the output in the output dataset.

The Shared Data Cache module stores arbitrary data in a cache store supporting key-value pairs. This module implements a Cache Interface having two methods: *get* and *put*. The *put* method takes in a serializable object and returns a key which points to the cached data in the repository, and *get* takes in the key and returns the cached object from the repository.

Although the architecture follows a very generalized approach, both the Parallel Processing and Shared Data Cache module implement specific conventions. The Parallel Processing module supports both batch and streaming input formats for passing between filters in a pipeline. The module can only process and return a dataset in a batched or streamed list format, and only the function provided to the module can be executed on the input dataset. The module is designed this way in the hope that the implementer will use the MapReduce technique to map the provided function to each partition of the input dataset. The number of partitions can be defined prior to using the module. As MapReduce is a proven technique for efficiently processing large volumes of data in a linear flow, it makes sense to design the module in a way to make it easy to implement MapReduce operations.

The Shared Data Cache module allows working with cache repositories which support key-value based caching. In a key-value cache, data is usually stored in a hash table, and the key refers to the row in which the data is stored. Data retrieval in a hash table using a key is fast and efficient. Due to this benefit, the module is designed to support any key-value based cache, so that a key can be used to retrieve stored data.

Figure 3.5 demonstrates a data flow view of the proposed architecture. The modules described in Figure 3.4 are shown here interacting with each other. The GWAS modules can be connected with each other in a pipeline. The first module in the pipeline receives input data from an arbitrary location. Subsequent modules can process and pass data through the pipeline. The terminal output from the last module of the pipeline can be forwarded to an arbitrary output location.

The GWAS modules can make use of the Parallel Processing and Shared Data Cache modules defined previously. The connection to a GWAS module from each of these modules is duplex, meaning data can be passed in any direction between each of the connected modules.

It is not necessary for a GWAS module to use the Parallel Processing and Shared Data Cache modules. However, using these modules is beneficial in certain cases. The Parallel Processing module can abstract away the distributed nature of the computation and the required communication details. A user only needs to plug in the Parallel Processing module to a GWAS module to harness the power of distributed computing and achieve scalable execution as well as a performance boost. Similarly, the Shared Data Cache can be used to cache and retrieve data at any stage of a pipeline. If a distributed cache is used as the cache repository, then the GWAS modules can access large volumes of cached data and achieve lower latency during data retrieval.

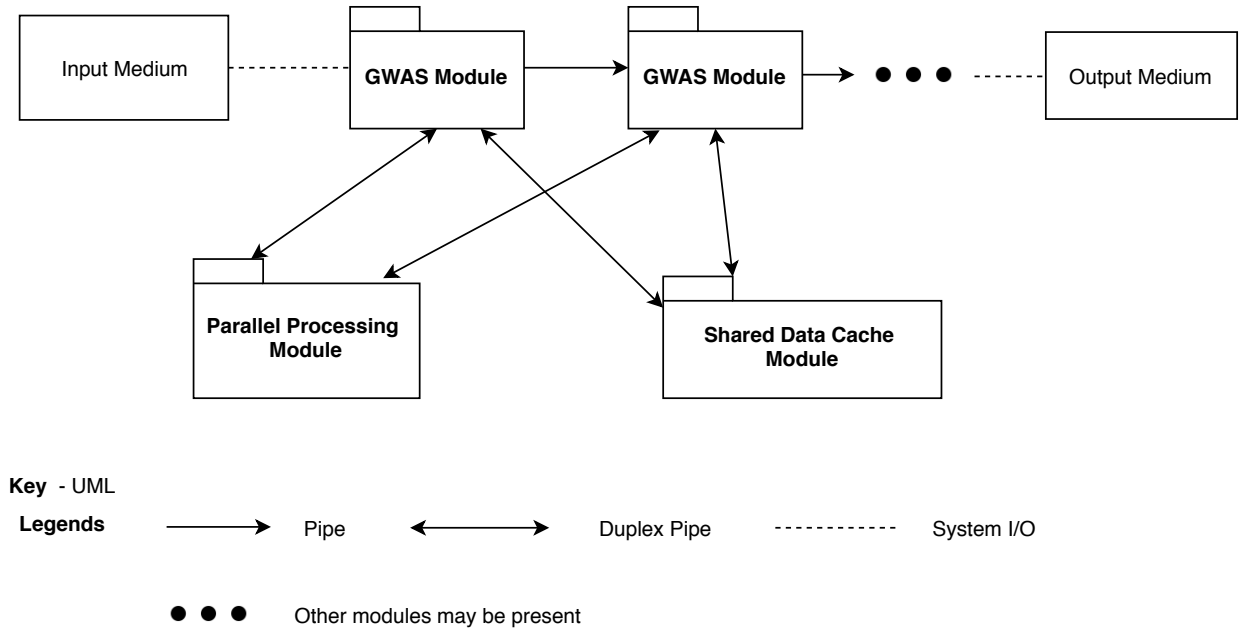


Figure 3.5: Data Flow View of proposed architecture

3.4 Architecture Quality Fulfillment

In section 3.2, a number of design considerations were described which could mitigate the shortfalls of existing GWAS applications, improve architecture structure and enable simpler application implementation. The previous section described the proposed architecture in detail. To show that the proposed architecture follows the design considerations, the relation between each of the design rules and the architecture is described. The description is given in the following subsections.

3.4.1 Scalability

The proposed architecture provides scope for scalability by introducing a parallel processing interface. This interface can handle implementations of frameworks which can work with clusters of machines. A parallel processing framework can distribute tasks to multiple processing cores. The more cores available to a framework, the more tasks can be distributed, which means that increasing resources should result in increased performance. However, the level of scalability depends not only on how the tasks are distributed, but also on sequential dependencies of tasks. In this way, the proposed architecture supports scaling GWAS applications in different environments.

3.4.2 Modifiability

The implementations of the architecture are modular in nature, which means that the implemented modules can be modified as necessary without significantly affecting operations of other modules. For example, if a GWAS application implements the proposed architecture and there are a number of modules that implement the defined interfaces, it is easy to bring changes to the modules without modifying other operations and still change how the program is executed. In this way, the proposed architecture supports modifying implementation as needed with minimal footprint.

3.4.3 Portability

The proposed architecture enables a specific form of portability by defining interfaces for parallel processing and shared data cache modules. By defining reusable modules, various parallel processing frameworks and cache stores can be used on different environments. If a specific module is needed for a specific environment, the architecture implementation will enable choosing a reusable module as needed. For example, there can be parallel processing modules implemented for both a single machine and clusters of machines, making it easy to run GWAS applications in both environments. Through these methods, portability becomes an attribute of the proposed architecture.

3.4.4 Configurability

As the interfaces described for the architecture are generic in nature, it is possible for a developer to make the implemented modules as configurable as possible. For example, by encapsulating a GWAS algorithm in a module, a developer can configure various parameters of the algorithm, set default parameters and enable execution tweaks as simply as setting command line parameters or loading a configuration file. Although it is possible to configure the parameters programmatically as part of the interface during load time, the choice should be made based on use cases. Parallel processing frameworks can follow the same course of action to be configurable. In this way, the implementation of the proposed architecture can be configured as much as needed.

3.4.5 Usability

The proposed architecture supports usability for both developers and end users. The modular nature of the architecture enables developers to encapsulate logic in blocks of modules and thoroughly test them before integrating with the main application. Consequently, the modular blocks enable end users to pick and choose operations as needed, making it easy for them to understand what is being executed and how to achieve specific objectives. A GWAS application implemented with the proposed architecture can have multiple GWAS operations encapsulated in individual modules, which can be easy to assemble in a workflow for end users. These benefits make the architecture easy to implement for developers and highly usable for end users.

3.5 Summary

In conclusion, this chapter described the system design of the proposed architecture. The architecture of candidate applications were analyzed first, and drawbacks were listed. Next, based on the analysis, design considerations for the proposed architecture were listed. Afterwards, the proposed architecture was presented and described using two architectural views. Finally, the proposed architecture was compared with the design considerations to see whether the architecture supports the design rules.

4 Case Study Details, Experimental Setup and Architectural Evaluation Details

In this chapter, the premise and setup of two case studies conducted based on the proposed architecture is described. An architectural evaluation based on survey questions is also described in this chapter.

4.1 Experimental Environment

In order to conduct the case study experiments, a number of experimental environments have been set up. These environments consist of two clusters (one physical and one virtual) and two single virtual machines. The goal of these environments is to conduct experiments in different scenarios. The environments are described as follows.

The first cluster, labeled as *Cluster 1*, is a cluster consisting of 10 physical machines. Each of the machines contain a 4 core Intel Core i7-2600 CPU with a frequency of 3.4 GHz hyper-threaded for 8 virtual cores, physical memory of 16 GB and a Gigabit network connection. The operating system is Linux Ubuntu 16.04 LTS (Xenial Xerus) for all machines. One machine is set as the master, while the rest of the machines are set as workers. The same cluster is used for configuring Redis cluster, where 3 nodes are set as master and 6 nodes are set as slaves. The master machine is not used as a Redis node.

The second cluster, labeled as *Cluster 2*, is a cluster consisting of 4 virtual machines. Each of the machines contain 7 quad-core Intel Xeon CPUs with a frequency of 2.1 GHz hyper-threaded for 56 virtual cores, physical memory of 504 GB and 492 GB of disk space. The operating system is Linux Ubuntu 18.04 LTS (Bionic Beaver) for all machines. One machine is set as the master, while the rest of the machines are set as workers.

A virtual machine called *discus-project1*, labeled as *Single Machine 1*, contains 4 quad-core Intel Core i7 CPUs with a frequency of 2.6 GHz hyper-threaded for 32 virtual cores, physical memory of 32 GB and 84 GB of disk space. The operating system is Linux Ubuntu 16.04 LTS (Xenial Xerus) for all machines.

Finally, a virtual machine labeled as *Single Machine 2* was created on a larger class server machine (*barley.usask.ca*). It contains 8 quad-core Intel Xeon Gold 6130 CPUs with a frequency of 2.1 GHz hyper-threaded for 64 virtual cores, physical memory of 1.48 TB, Gigabit ethernet connection and 137 GB of disk space. The operating system is Linux Ubuntu 18.04 LTS (Bionic Beaver) for all client virtual machines. The host runs Linux Ubuntu 18.04 LTS (Bionic Beaver).

4.2 TASSEL Case Study

The first case study is an implementation of the proposed architecture using TASSEL. As the architecture of TASSEL is already based on a pipeline, it is an ideal candidate on which to implement the architecture. The case study is laid out as follows. First, the architecture of TASSEL is described. Two architectural views for TASSEL are provided - a modular decomposition view and a data flow view. Next, the architecture is extended based on the proposed architecture, and the modified architecture design is described. A TASSEL plugin called *Weighted MLM* was chosen to be modified and extended. After that, the steps taken to implement the extended architecture is reported. A number of bottlenecks were identified and two modules from the extended architecture were inserted in place of the bottlenecks to increase performance. Finally, a number of experiments are set up to understand the performance benefits of this implementation. The experiments deal with factors such as number of cores, partition count and dataset size to verify the change in performance. Chapter 5 will report on the results of the experiments.

4.2.1 Architecture of TASSEL

Figure 4.1 demonstrates a modular decomposition view of the TASSEL architecture. In this view, TASSEL contains a set of analysis modules inside an *Analysis* package and a set of utility modules inside a *Utility* package. The analysis modules contain implementations of GWAS operations and the utility modules contain statistical, mathematical and file load operations. Both types of modules implement a *Plugin* interface, which allows the modules to be usable in a pipeline environment.

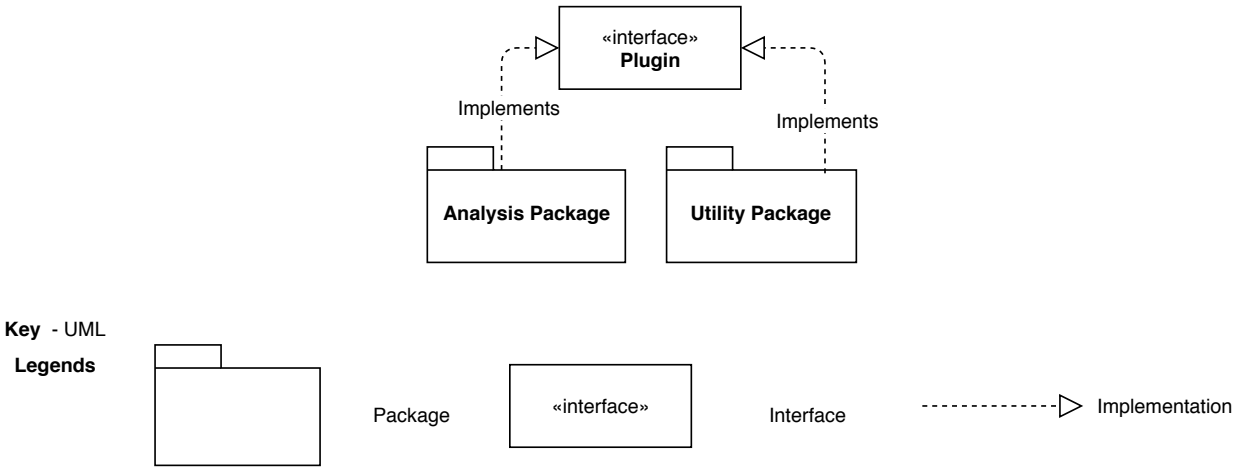


Figure 4.1: Modular Decomposition View of TASSEL architecture

Figure 4.2 demonstrates a data-flow view of the TASSEL architecture. In this view, the modules implementing the Plugin interface are connected via a pipeline. The initial module, usually a file load plugin, receives input from a data source. The plugin then prepares and loads the data into memory to be used in subsequent modules of the pipeline. The final output of the pipeline is stored in an output medium, usually a

filesystem. As TASSEL supports thread-based execution, each stage of a pipeline can execute a given plugin under multiple threads concurrently. However, the successive plugins operate sequentially.

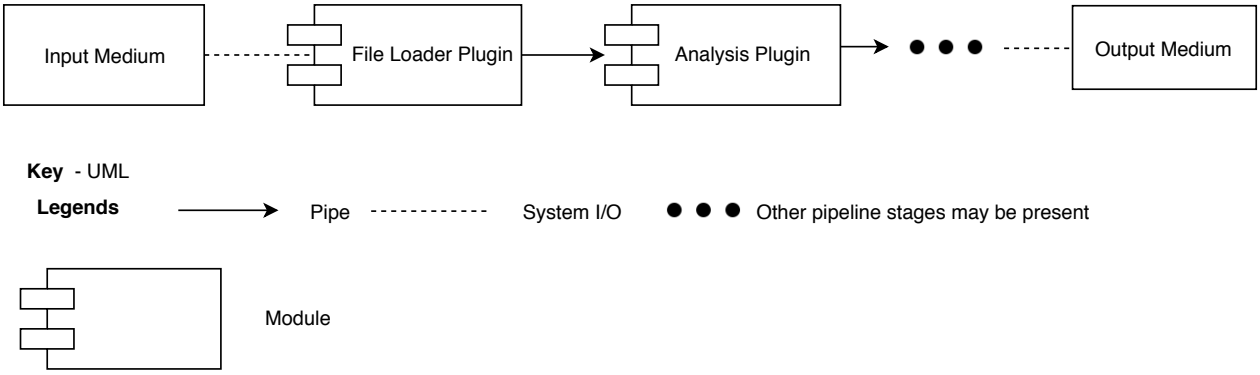


Figure 4.2: Data Flow View of TASSEL architecture

4.2.2 Modification of TASSEL Using Proposed Architecture

Figure 4.3 showcases the modular decomposition view of the modified architecture of TASSEL, using the proposed architecture. The modified architecture utilizes the Parallel Processing and Shared Data Cache modules from the proposed architecture. The modules are used by both the Analysis and Utility modules. In this way, the modules can parallelize their internal processing and store reusable data in the cache.

A data-flow view of the *modified* TASSEL architecture is shown in Figure 4.4. Here, the modules in the pipeline use both the Parallel Processing and Shared Data Cache modules. The connection between the modules in the pipeline and the helper modules are duplex, meaning that data flows both ways. It is not mandatory to use these modules; they can increase execution and data access speed.

4.2.3 Case Study Steps

The MLM module carries out association analysis using a mixed linear model (MLM). The MLM conducts association tests for each combination of markers and traits. It takes a genomic dataset as input and creates two output tables which represent model statistics and model effects. The table with the model statistics shows the test result for each trait. The second table with the model effects shows a list of estimated effects of each allele related to each marker. If compression is used with the plugin, then a third table called compression result shows the genetic and error variances, as well as the likelihood for each tested compression level.

The first step in this case study was to find where the Parallel Processing and Shared Data Cache modules could be inserted to maximize performance and decrease data access latency. To do that, bottlenecks inside the Weighted MLM plugin that can benefit from parallelization needed to be identified. Two experiments were performed with TASSEL as base to accomplish this task.

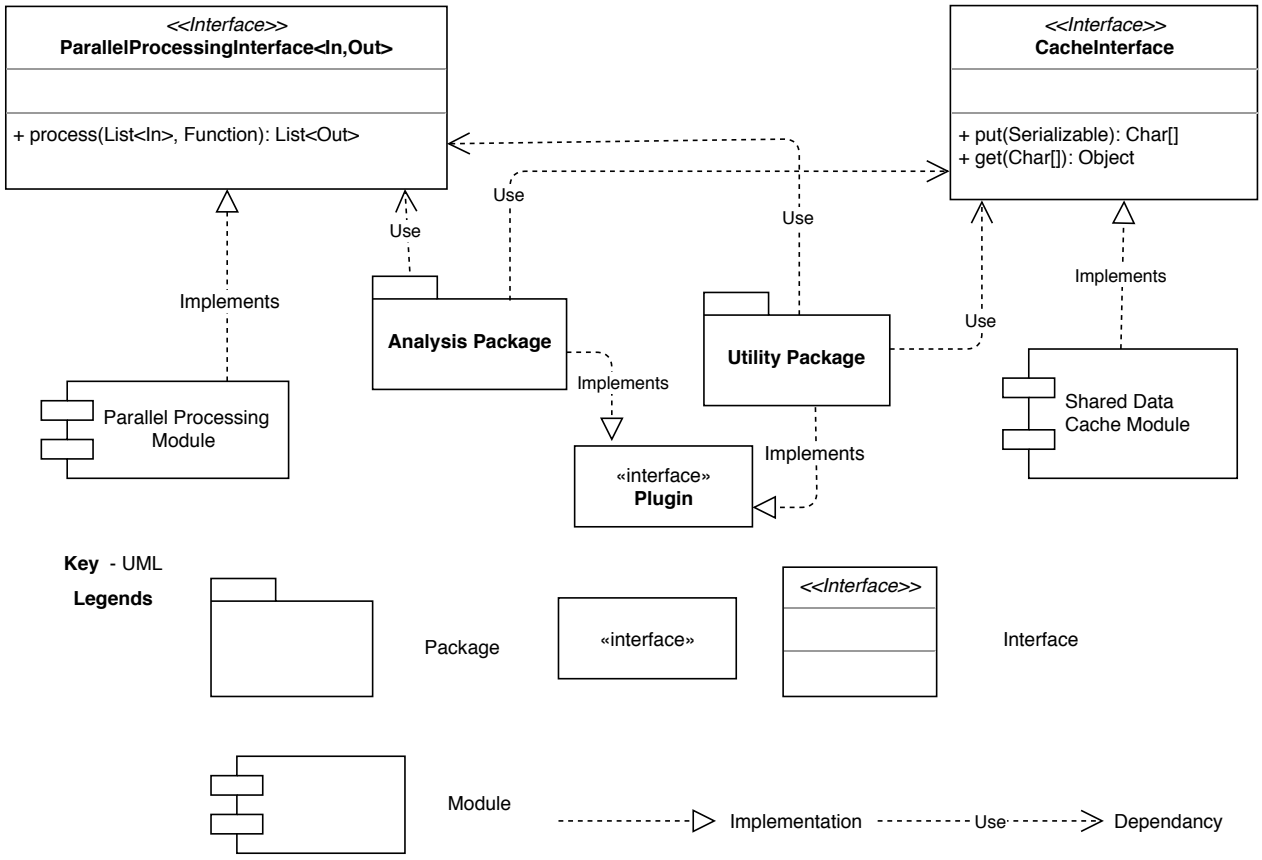


Figure 4.3: Modular decomposition view of modified TASSEL architecture extended from the proposed architecture

The experiments used a sample Arabidopsis dataset containing 214,051 SNPs as input. All of the experiments used *Single Machine 1* and had fixed parameters of 32 cores, 32 GB RAM and no GPU. From these experiments, two potential bottlenecks were found using debug breakpoints. Both bottlenecks are iterations over code blocks that take a significant portion of the overall execution time. Moreover, both bottlenecks are inside a class called *CompressedMLMusingDoubleMatrix* which is called from the weighted MLM plugin.

The first bottleneck is inside the method *computeZKZ*. The time-consuming block of code iterates through the provided genetic similarity matrix and generates a compression report. The code block for this bottleneck is described in Appendix A. The second bottleneck is inside the method *solve*. Here, the code iterates over genetic markers of the input dataset and generates output of the weighted MLM plugin. The code block for this bottleneck is described in Appendix B.

After the bottlenecks were identified, both the Parallel Processing and Shared Data Cache modules specific to TASSEL were implemented. Apache Spark was the processing framework chosen to realize the Parallel Processing module, while Redis Cluster was the cache store chosen to realize the Shared Data Cache module. The details of these implementations are as follows.

To build the Parallel Processing module, the first step at this stage was to decide what operation to

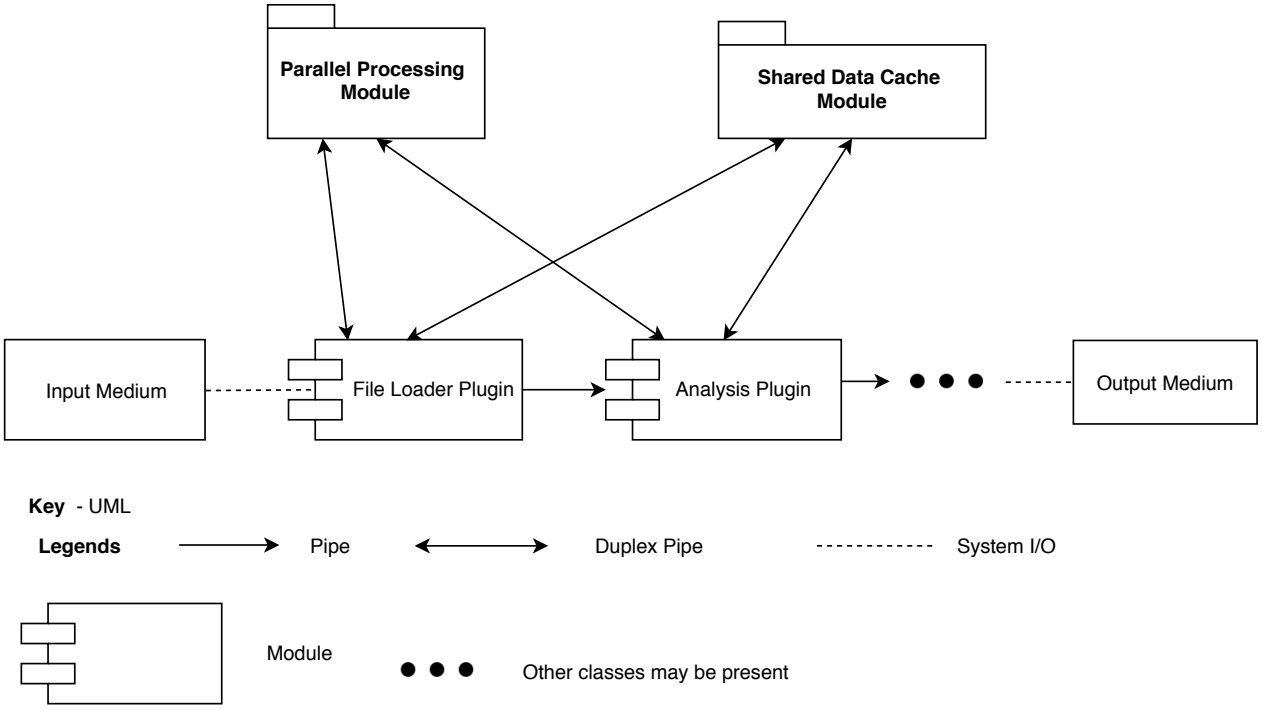


Figure 4.4: Data flow view of modified TASSEL architecture extended from the proposed architecture

run over the RDD. The RDD operation *mapPartitionsWithIndex* was the perfect candidate for this, as this operation applies the map function to each partition of the RDD and tracks the index of the original partition as well. In this way, the module will satisfy the constraints of the *ParallelProcessing* interface by representing the RDD as a collection of items. The code block of the implementation of the Parallel Processing module using Spark is provided in Appendix C.

After building the Parallel Processing module, the next step was to refactor the bottlenecks to a function which can process the iterator passed on to the module. The bottlenecks are wrapped by *for loops*, which loop over a list of genetic markers. The code block inside each for loop process each item of the list. This code block can be wrapped in a function, the parameter of which can be a single genetic marker. The code blocks demonstrated in Appendix A and B were refactored, and the resultant functions can be seen in Appendix D and E, respectively.

Redis cluster[1] was selected as the cache store for the Shared Data Cache module. Redis is a data structure store which uses a combination of memory and disk to store data in key-value pairs. In a Redis cluster, data is automatically partitioned across distributed Redis nodes. This helps in caching and replicating large data, which can be retrieved from any cluster machine with low latency. Although HDFS was another option for storing the data in a distributed fashion, data access using Redis is much faster due to its hash table structure and architectural optimization. This demonstrates that the modular architecture allows

[1]Redis Cluster Tutorial, `https://redis.io/topics/cluster-tutorial`

parts of other frameworks to be used effectively. In this case, the standard use of HDFS was replaced by an improvement. The implementation of the Shared Data Cache module using Redis is demonstrated in Appendix F.

4.2.4 Experiment Setup

Once the modules have been constructed, the implementation of the *modified* TASSEL architecture was complete. To verify the implementation and understand performance benefits, a set of experiments were conducted with the implementation using different scenarios. There are two objectives for the experiments. The first objective is to show that implementing the proposed architecture using a parallel processing framework results in increased performance. The second objective is to find out how much the performance scales as resources are increased and experiment factors are changed.

Two sets of data are used in this case study - a sample dataset of Arabidopsis thaliana and a lentil dataset. The Arabidopsis dataset is acquired from the Arabidopsis thaliana polymorphism database[2] and contains 214,051 SNPs. The provided file is in HapMap format, and the size of the file is 170 MB. All except the last set of experiments used the Arabidopsis dataset. The lentil dataset, on the other hand, is provided by the Plant Science department of the University of Saskatchewan, and contains 31,883,135 SNPs. These SNPs are divided into 8 files; 7 of those files represent SNPs from chromosome 1 to 7, and one file has unassigned SNPs. The last set of experiments will use the first chromosome, which is in VCF format. It contains 4,651,975 SNPs, and the size of the file is 33 GB.

For the first set of experiments, the number of partitions will remain the same, but the number of cores will increase. For the second set of experiments, the number of cores will remain the same, but the number of partitions will increase. For the third set of experiments, the number of partitions will increase with the amount of resources, decreasing the partition size. It is expected that this should influence the performance due to communication between the cores for intermediate and output data. In the last set of experiments, the number of SNPs in the subset of the lentil dataset will increase, while the number of partitions and the amount of resources remain fixed.

In the experiments, the number of partitions were chosen as parameters to vary instead of the size of partitions. This is due to the way TASSEL handles datasets as input. TASSEL stores each row of the input dataset as an object, then processes each row sequentially. It was therefore not feasible to partition the input dataset based on size, as a representation of the dataset is partitioned instead of the raw dataset that TASSEL uses as input directly. Partitioning the dataset itself would require substantial modification of TASSEL file loading operation, which is beyond the scope of this thesis.

In order to find out how Spark partitions the rows of objects of the Arabidopsis dataset, a test experiment was conducted with the default Spark parameters. In that experiment, Spark opted to set the number of partitions to 72, bringing the partition size to 2.36 MB. Setting 72 as the default number of partitions for the

[2] `https://github.com/Gregor-Mendel-Institute/atpolydb/tree/master/250k_snp_data`

Arabidopsis dataset, the experiments with TASSEL used a range of higher and lower number of partitions as variables, to better understand the effect of varying number of partitions.

Cluster 1 and Cluster 2 described in this chapter were used for the case study. In addition, a Redis cluster was set up in Cluster 1, having 3 master nodes and 6 slave nodes. No Redis cluster was set up on Cluster 2, as the machines in the cluster were virtual.

Baseline Experiments

Before conducting the scalability experiments, a set of preliminary experiments were conducted in which the original, unmodified TASSEL package was executed with the Arabidopsis and the lentil dataset.

For the Arabidopsis dataset, the experiments had two scenarios. In the first scenario, TASSEL used one core only for processing. In the second scenario, all cores were available for execution. Both experiments ran on *Single Machine 1*.

For the lentil dataset, subsets of the first chromosome containing 0.5 million, 1 million, 2.5 million and 4.65 million SNPs were used. All cores were used and the experiments were conducted on *Single Machine 2*.

Spark Experiments: Increasing Core Count Only

The second set of experiments kept the partition count the same, but the core count differed. All of the experiments used 8 GB memory for the Spark driver and 13 GB memory per executor. Each experiment was replicated 4 times. The partition count for all of the experiments was fixed at 252, and the partition size was 675 KB. Recall that the actual input size is much smaller than the size of the object (number of rows) that are used as input by TASSEL. 9 executors were used for the experiments. The experimental details are given in Table 4.1.

Table 4.1: Spark Experiments: Increasing Core Count Only

Exp. No.	Cores/Executor	Total Cores
7	1	9
8	2	18
9	4	36
10	7	63

Spark Experiments: Increasing Number of Partitions Only

The next set of experiments kept the core count the same, but the number of partitions differed. All of the experiments used 8 GB memory for the Spark driver and 13 GB memory per executor. Each experiment was replicated twice and the average value is reported. The executor count was fixed at 9 and the total core count was fixed at 63 for all of the experiments. The experimental details are given in Table 4.2.

Table 4.2: Spark Experiments: Increasing Number of Partitions Only

Exp. No.	Partitions	Partition Size
11	36	4.72 MB
12	72	2.36 MB
13	144	1.18 MB
14	252	675 KB

Spark Experiments: Scaling Core Count with Partition Count

The fourth set of experiments varied the number of tasks with total core counts. All of the experiments used 8 GB memory for the Spark driver and 13 GB memory per executor. Each experiment was replicated 4 times and 9 executors were used. The experimental details are given in Table 4.3.

Table 4.3: Spark Experiments: Scaling Core Count with Partition Count

Exp. No.	Cores/Executor	Total Cores	Partitions	Partition Size
15	1	9	36	4.72 MB
16	2	18	72	2.36 MB
17	4	36	144	1.18 MB
18	7	63	252	675 KB

Spark Experiments: Using lentil dataset in Cluster 1

The first chromosome of the lentil dataset used for the experiments contains around 4.65 million SNPs. For these experiments, only the dataset containing SNPs of the first chromosome has been selected. The objective of these experiments is to see how the performance scales in the Spark Cluster while the number of SNPs in the dataset changes. To accomplish this, the number of partitions and the amount of resources remains the same, while the number of SNPs changes.

Two types of experiments were conducted for this environment. For the first experiment type, executor count was fixed at 9, cores per executor was 7 and executor memory was 13 GB. For the second experiment type, executor count was fixed at 63, cores per executor was 1 and executor memory was 2 GB. The experiments use 8 GB memory for the Spark driver. For all of the experiments, the core count was fixed at 63 and the partition count was fixed at 630, making the partition size 52 MB. The experimental details are given in Table 4.4.

Spark Experiments: Using lentil dataset in Cluster 2

Similar to the previous experiment, these experiments seek to evaluate how the performance scales in the Spark Cluster while the number of SNPs in the dataset changes (total data size volume). Compared to

Table 4.4: Spark Experiments: Using lentil dataset in Cluster 1

Exp. No.	No. of SNPs	Executors	Cores/Executor	RAM/Executor
19	0.5 million	9	7	13
20	1 million	9	7	13
21	2.5 million	9	7	13
19	0.5 million	63	1	2
20	1 million	63	1	2
21	2.5 million	63	1	2

Cluster 1, *Cluster 2* has more cores (168 compared to 72), more RAM (1.5 TB compared to 144GB), faster network speed (25 Gbps compared to 1 Gbps) and the processors are slower (2.1 GHz i7-2600 CPU compared to 3.4 GHz Xeon CPU). The experiments use 300 GB memory for the Spark driver and 300 GB memory per executor. The core count for all of the experiments was fixed at 156, 3 executors were used and the partition count was fixed at 630, making the partition size 52 MB. The experimental details are given in Table 4.5.

Table 4.5: Spark Experiments: Using lentil dataset in Cluster 2

Exp. No.	No. of SNPs
22	0.5 million
23	1 million
24	2.5 million
25	4.65 million

4.3 FaST-LMM Case Study

The second case study is an implementation of the proposed architecture using FaST-LMM, another GWAS application. The case study is laid out as follows. First, the architecture of FaST-LMM is described. Next, the architecture is extended based on the proposed architecture and the modified architecture design is described. After that, the steps taken to implement the extended architecture is reported. Finally, a set of experiments are set up to understand the performance benefits of this implementation. The next chapter will report on the results of the experiments.

4.3.1 Architecture of FaST-LMM

Figure 4.5 demonstrates a modular decomposition view of the FaST-LMM architecture. FaST-LMM primarily contains three packages - *Association*, *Inference* and *Utility*. The *Association* package contains all the GWAS operations of the program. The *Inference* package contains predictors for linear mixed models and linear

regression. The *Utility* package contains helper modules needed by the other packages such as runners. As previously described, runners are modules that can run GWAS operations in different environments. While the modules inside the *Association* package run the major portion of an operation, they also use the modules of *Inference* and *Utility* packages to perform GWAS operations.

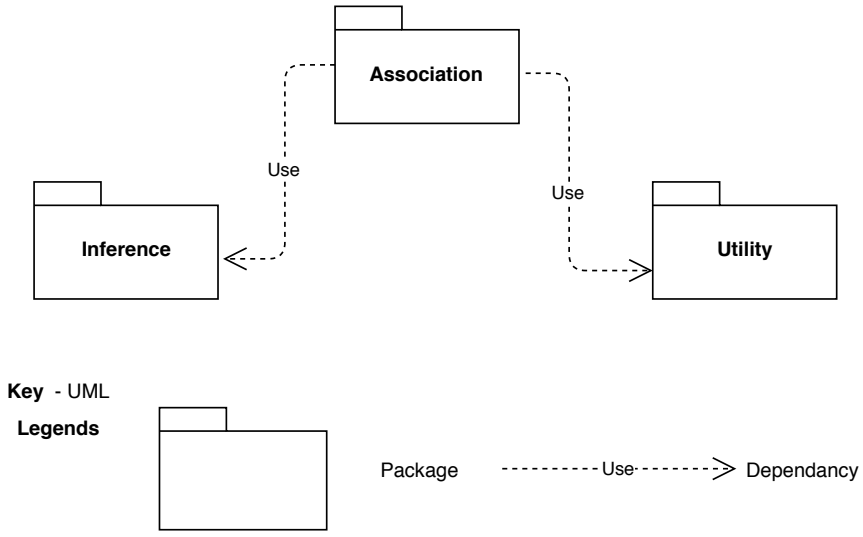


Figure 4.5: Modular Decomposition View of FaST-LMM architecture

Figure 4.6 demonstrates a data-flow view of the FaST-LMM architecture. The modules of the *Association* package can consume input, process it and produce an output. There are no multiple modules connected together with a pipeline - a single association module computes and returns output. During computation, the association module can call module operations from both *Inference* and *Utility* packages. Typically, an association module calls a runner from the *Utility* package to determine in which environment to run the operations, and a predictor from the *Inference* package to predict and score associations.

4.3.2 Modification of FaST-LMM using proposed architecture

Figure 4.7 showcases the modular decomposition view of the modified architecture of FaST-LMM, using the proposed architecture. The modified architecture borrows the Parallel Processing and Shared Data Cache modules from the proposed architecture. The borrowed modules can be used by the modules of the *Association*, *Inference* and *Utility* packages. In this way, the modules can parallelize their internal logic and store reusable data in the cache.

A data flow view of the modified FaST-LMM architecture is shown in Figure 4.8. Here, the modules can use both the Parallel Processing and Shared Data Cache modules at any stage of operation. Both of these modules can send data to and receive data from the main modules of FaST-LMM. As these modules can increase execution and data access speed, it is recommended for implementors to use these modules if there is a substantial performance benefit, although usage of these modules are optional.

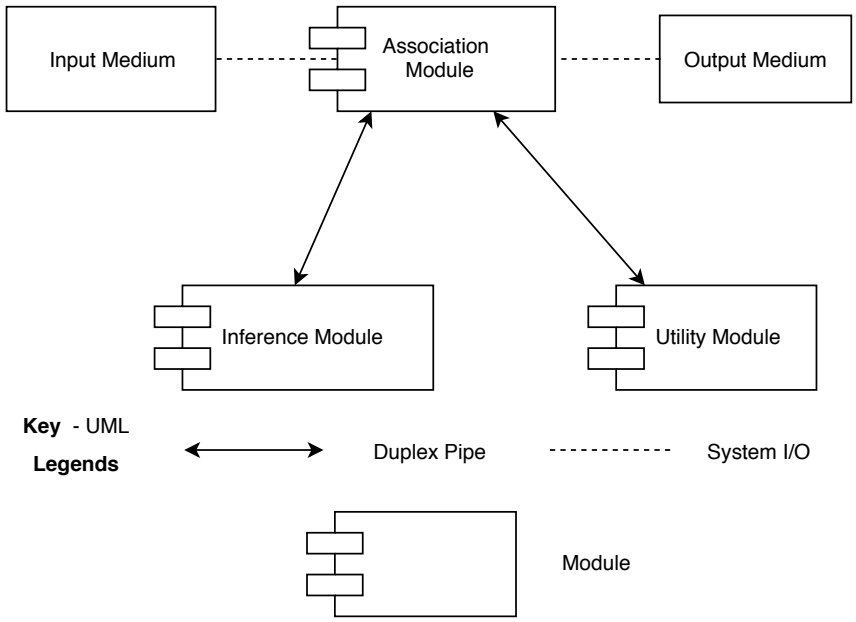


Figure 4.6: Data Flow View of FaST-LMM architecture

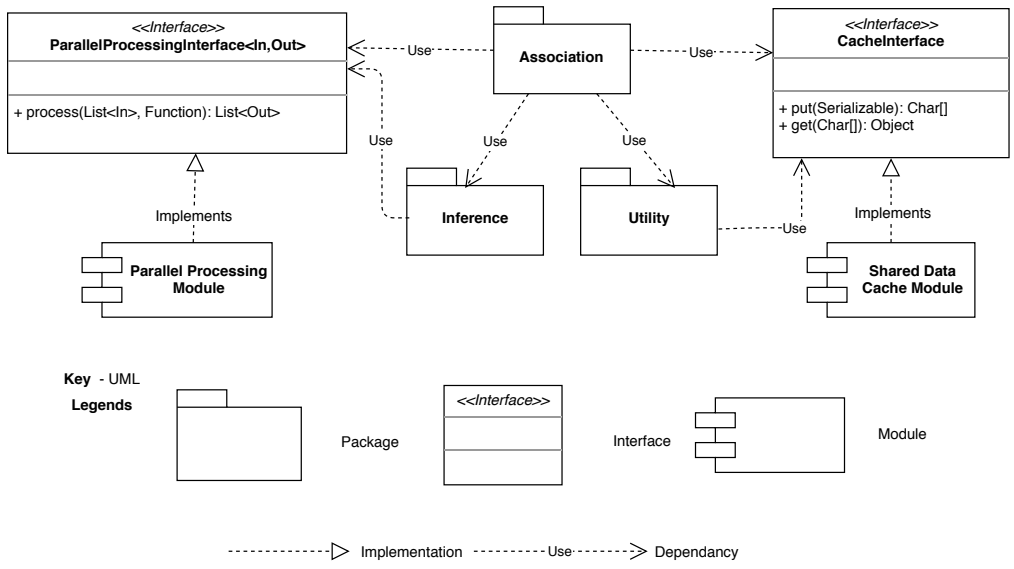


Figure 4.7: Modular decomposition view of modified FaST-LMM architecture extended from the proposed architecture

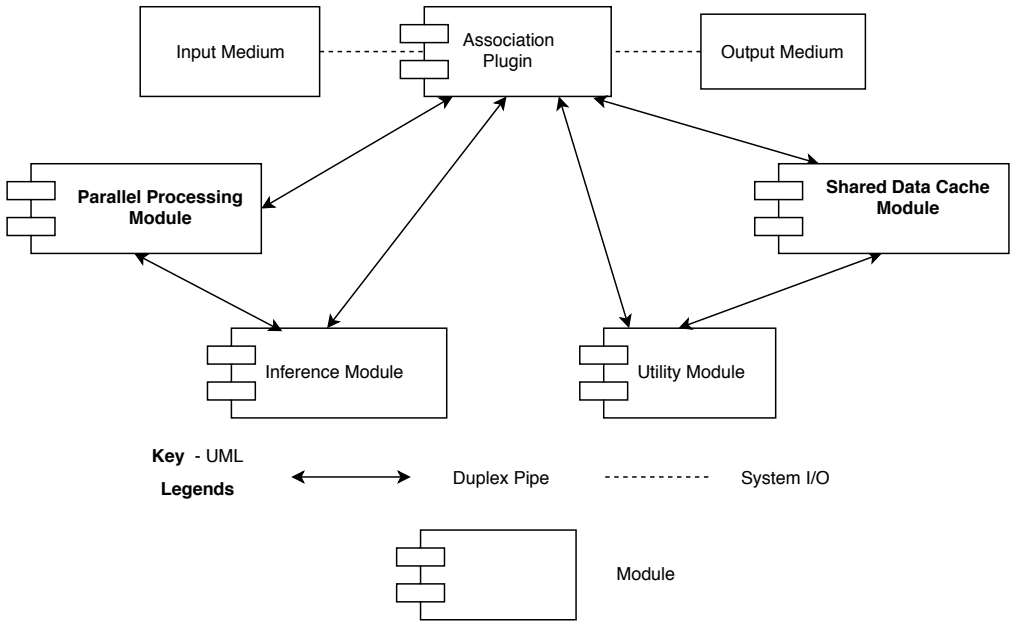


Figure 4.8: Data flow view of modified FaST-LMM architecture extended from the proposed architecture

4.3.3 Case Study Steps

In this case study, the *single_snp_all_plus_select* module of the FaST-LMM package was selected for performance improvement. This association module performs single SNP GWAS based on two kernels; one is based on all SNPs and the other is a similarity matrix based on top SNPs. It takes a SNP dataset and a single phenotype as input, and returns a dataframe with statistical values of the test SNPs.

The first step in this case study was to find out where the Parallel Processing and Shared Data Cache modules could be inserted to maximize performance and decrease data access latency. To do that, bottlenecks inside the *single_snp_all_plus_select* module, which can benefit from parallelization, needed to be identified. Fortunately, the module uses a modified implementation of MapReduce. By investigating and modifying this implementation, the Parallel Processing and Shared Data Cache modules can be integrated inside the *single_snp_all_plus_select* module.

There are three functions in the *single_snp_all_plus_select* module which are used in the map operation - *mapper_find_best_given_chrom*, *mapper_gather_lots* and *mapper_single_snp_2K_given_chrom*. The *mapper_gather_lots* function is a map operation nested inside the *mapper_find_best_given_chrom* function. In order to allow FaST-LMM to work with generic MapReduce implementations, the nested MapReduce operation needed to be flattened so that a single MapReduce job can perform the whole operation.

To flatten the nested MapReduce implementations, both the parent and child operations were modified in such a way that only the *mapper_gather_lots* function was sufficient to perform required operations in parallel. There was no need for the *mapper_find_best_given_chrom* function anymore, so it was removed. This left the *single_snp_all_plus_select* module with two MapReduce jobs to perform.

After the *single_snp_all_plus_select* module was modified, both the Parallel Processing and Shared Data Cache modules were implemented. Two processing frameworks were chosen to instantiate the Parallel Processing module - Apache Spark and Dask. Redis Cluster was the cache store chosen to instantiate the Shared Data Cache module. The implementation of the Redis Cluster as the Shared Data Cache module follows the description provided in Section 4.2.3 and demonstrated in Appendix I. The details of the implementations of the Parallel Processing modules are as follows.

To build the Parallel Processing module using Apache Spark, the first step at this stage was to transform the MapReduce operations of the *single_snp_all_plus_select* module into RDD map operations. These operations allow the module to satisfy the constraints of the *ParallelProcessing* interface by representing the RDD as a collection of items. The code block of the implementation of the Parallel Processing module using Spark is provided in Appendix G.

Similar to the implementation of Apache Spark, the first step of implementing Dask for the Parallel Processing module is to transform the MapReduce operations of the *single_snp_all_plus_select* module into distributed Dask operations. Dask provides a map function to distribute parallel tasks to workers, which allows the module to satisfy the constraints of the *ParallelProcessing* interface. The code block of the implementation of the Parallel Processing module using Dask is provided in Appendix H.

4.3.4 Experiment Setup

To verify the implementation and understand performance benefits, a set of experiments were conducted with the implementation using different scenarios. Similar to the first case study with TASSEL, there are two targets of these experiments - to show that the implementation of the proposed architecture increases performance, and to find out performance scalability in terms of increasing resources and varying experiment factors.

Two parallel processing frameworks called Dask and Spark are used for these experiments. Three sets of experiments will be conducted with each of these frameworks. For the first experiment in the set, the number of partitions remain the same while the number of cores increase. For the second experiment in the set, the number of cores remain the same the while the number of partitions increase. For the third experiment in the set, the number of partitions increases with the number of cores.

Similar to how TASSEL stores input dataset, as explained in Section 5.1.1, FaST-LMM stores each row of the input dataset as objects, then processes each row in sequence or parallel, depending on the type of runner used. In order to find out how Spark and Dask partitions the rows of objects of the Arabidopsis dataset, a test experiment was conducted using Spark and Dask with their default configurations. In the experiments, both Spark and Dask opted to set the number of partitions to 880, bringing the partition size to 193 KB. Setting 880 as the default number of partitions for the Arabidopsis dataset, the experiments with FaST-LMM used a range of higher and lower number of partitions as variables, to better understand the effect of varying number of partitions.

Cluster 1 and *Cluster 2* were used for the experiments. In addition, a Redis cluster was set up on *Cluster 1*, having 3 master nodes and 6 slave nodes. All of the experiments used the Arabidopsis thaliana dataset.

Baseline Experiments

A set of preliminary experiments were conducted in which the original, unmodified FaST-LMM package was executed in three scenarios. In the first scenario, FaST-LMM used a local runner which uses only one core for processing. In the second scenario, FaST-LMM used a multiprocessor runner which used 8 cores for processing. The last experiment in the set used the same multiprocessor runner to run with 64 cores. All of the experiments ran on *Single Machine 2*.

Dask and Spark Experiments: Increasing Core Count Only

The next set of experiments will keep the number of partitions the same, but the core count differed. All of the experiments used 8 GB memory of master machine for Dask scheduler, 8 GB memory per Dask worker, 8 GB memory for the Spark driver and 13 GB memory per Spark executor. Each experiment used 9 Spark executors, 9 Dask workers and was replicated 4 times. The partition count for all of the experiments were fixed at 880, making the partition size 193 KB. The experimental details are given in Table 4.6.

Table 4.6: Dask and Spark Experiments: Increasing Core Count Only

Exp. No.	Cores/Worker	Total Cores
7	1	9
8	2	18
9	4	36
10	7	63

Dask and Spark Experiments: Increasing Number of Partitions Only

The third set of experiments will keep the core count the same, but the number of partitions differed. All of the experiments used 8 GB memory of master machine for Dask scheduler, 8 GB memory per Dask worker, 8 GB memory for the Spark driver and 13 GB memory per Spark executor. Each experiment was replicated twice and the average value was counted. The core counts for all of the experiments were fixed at 63. 9 Dask workers and 9 Spark executors were used. The experimental details are given in Table 4.7.

Dask and Spark Experiments: Scaling Core Count with Partition Count

The fourth set of experiments varied the number of tasks with total core counts. These experiments used 8 GB memory of master machine for Dask scheduler, 8 GB memory per Dask worker, 8 GB memory for

Table 4.7: Dask and Spark Experiments: Increasing Number of Partitions Only

Exp. No.	Partitions	Partition Size
11	440	386 KB
12	880	193 KB
13	1760	96 KB
14	3520	48 KB

the Spark driver and 13 GB memory per Spark executor. Each experiment was conducted 4 times. 9 Dask workers and 9 Spark executors were used. The experimental details are given in Table 5.4.

Table 4.8: Dask and Spark Experiments: Scaling Core Count with Partition Count

Exp. No.	Cores/Worker	Cores	Partitions	Partition Size
3	1	9	440	386 KB
4	2	18	880	193 KB
5	4	36	1760	96 KB
6	7	63	3520	48 KB

4.4 Evaluation of Architecture using Feedback from Experts

The final part of this chapter is a description of an architectural evaluation using a questionnaire and asking feedbacks from domain experts. The evaluation details will be laid out, including domain expert choice and question list. The next chapter will have the responses and evaluation of the questionnaire.

In addition to performance evaluation, there is a need for the architecture and its implementation to be evaluated by domain experts and end users. In this section, a review of the proposed architecture and its implementation was conducted by preparing a questionnaire and asking for feedback from domain experts. The target of the questionnaire was domain experts in Software Engineering, Bioinformatics and Distributed Programming.

To conduct the evaluation, two sets of questionnaires were prepared. The first set of questions, demonstrated in Table 4.9, are closed-ended, meaning that the interviewee can answer the questions by saying *yes*, *no* or *no comment*. The second set of questions, demonstrated in Table 4.10, are open ended, which means that the interviewee can provide a thorough answer on those questions. The purpose of these questions is to investigate whether the architecture and its implementation are useful to domain experts and end users.

There are three objectives of the questionnaires. The first objective is to find out whether the performance benefit of the architecture makes is useful for the end users. The second objective is to find out whether the reusability of the modules makes it easy for both developers and end users to use the architecture and its

Table 4.9: Closed-Ended questions

Questions
Q1. If an architecture is proposed where multiple machines are used for executing a GWAS application faster, would you find it valuable?
Q2. Does the architecture clearly explain how reusable modules can be used to gain performance improvement?
Q3. Is the module reusability for performance gain in the architecture useful for you?
Q4. Is the performance benefit compared to the original implementation in the experiments useful for you?
Q5. Is the implementation of the proposed architecture easy enough for you to use?
Q6. Do you believe the proposed architecture is extendible towards other GWAS applications?

Table 4.10: Open-Ended questions

Questions
Q1. What benefits do you see from this implementation?
Q2. What challenges/disadvantages of the implementation do you see?

implementation. The third objective is to find out the opinion of the domain experts regarding the feasibility of the architecture to be used in additional GWAS applications.

The domain experts for the evaluation were chosen from three domains . Three experts from each domain were chosen for the evaluation. These nine experts include graduate and post graduate students as well as research assistants. These experts were chosen for their knowledge in their respective domain and their involvement in GWAS studies.

4.4.1 Summary

In conclusion, this chapter lays out the details of two case studies being conducted using the proposed architecture. The objectives of the case studies are to demonstrate prototypes of the architecture implementation and verify performance benefits of the implementations. Additionally, a plan for an architectural evaluation based on feedbacks from domain experts has also been laid out. The feedback will serve as evaluation for the architecture and the prototype implementation.

5 Case Study Performance Experiment and Architectural Evaluation Results

In this chapter, the results of the experiments performed to understand the performance and usability benefits of the case studies of the architecture described in Chapter 4 are reported and discussed. First, the experiment results of the case study in TASSEL are described. Next, the experiment results of the case study in FaST-LMM are described. Finally, the results of the architecture evaluation using feedback from experts is also reported.

5.1 Experiment Results for TASSEL Case Study

In this section, the baseline experiment results will be described first. Next, three sets of experiment results for the arabidopsis dataset follow - experiments where only the number of cores changed, experiments where only the number of partitions changed and experiments where the number of cores were scaled with number of partitions. These results indicate that using the prototype implementation results in better performance compared to original implementation, but the performance also demonstrates sub-linear scalability. After that, the experiment results using the lentil dataset are presented. The experiment results with lentil dataset indicate that although the prototype implementation performs much better than the original implementation, it can process data set sizes in a linear scalable manner to a certain threshold before encountering a memory error. Finally, discussions regarding common patterns in the results and corresponding insight is provided.

5.1.1 Baseline Experiments

Table 5.1 describes the results for the baseline experiments with the Arabidopsis dataset. Each experiment was repeated twice and the average value is reported. There is not much performance difference between the restricted and unrestricted execution of TASSEL. Using 32 cores in unrestricted environment results in 1.2% increase of execution time. This is not a significant increase in execution time, which means the performance remains almost the same for both restricted and unrestricted execution. A potential reason for the degraded performance can be the overhead in switching between cores during multicore execution. During its execution, TASSEL uses only one core for most of the operational period, except when the input files are being loaded. Therefore, allowing the application to run using multiple cores does not result in significant performance differences.

Table 5.1: Baseline experiments with original, unmodified TASSEL and Arabidopsis dataset

Exp. No.	Cores	Exec. Time
1	1	3697 seconds
2	32	3741 seconds

Table 5.2 describes the results for the baseline experiments with the lentil dataset. Each experiment was repeated twice and the average value is reported. Working on 0.5 million SNPs takes around 128 minutes, while working on 4.65 million SNPs takes around 1076 minutes. 4.65 million SNPs is 9.3 times bigger than 0.5 million SNPs but the increase in execution time is only 8.4. This suggests that the rate of computation is super linear for sequential execution as dataset size is increased. A potential reason for this is that each row in the input dataset takes roughly the same time to be processed, allowing the program to maintain linear scalability as much as possible. Furthermore, overlapping during calculation on bigger dataset can also result in more than linear speedup.

Table 5.2: Baseline experiments with original, unmodified TASSEL and lentil dataset

Exp. No.	SNPs	Exec. Time
3	0.5m	7730 seconds
4	1m	15258 seconds
5	2.5m	38205 seconds
6	4.65m	64579 seconds

5.1.2 Spark Experiments: Increasing Core Count Only

Figure 5.1 shows the performance comparison between sequential and cluster operations while scaling the number of cores only. The sequential run is the experiment done with 32 cores, while the restricted sequential run was restricted to one core only. Each experiment was repeated four times and the average value is reported. The coefficient of variation between the replications is 0.05.

Using Spark with 1 core per executor can boost performance up to 7 times compared to the sequential operation. Even though there are 9 worker machines, this shows some sequential computation that cannot be parallellized. Compared to the performance of 1 core per executor, using 7 times the cores per executor only results in 2.2 times speed improvement. The implications of these experimental results will be discussed further in the discussion section.

5.1.3 Spark Experiments: Increasing Partition Count Only

Figure 5.2 demonstrates how fixed core count and increasing partition count (decreasing partition size) affects Spark performance. There is very little difference between the performance of experiments with 36, 72 and

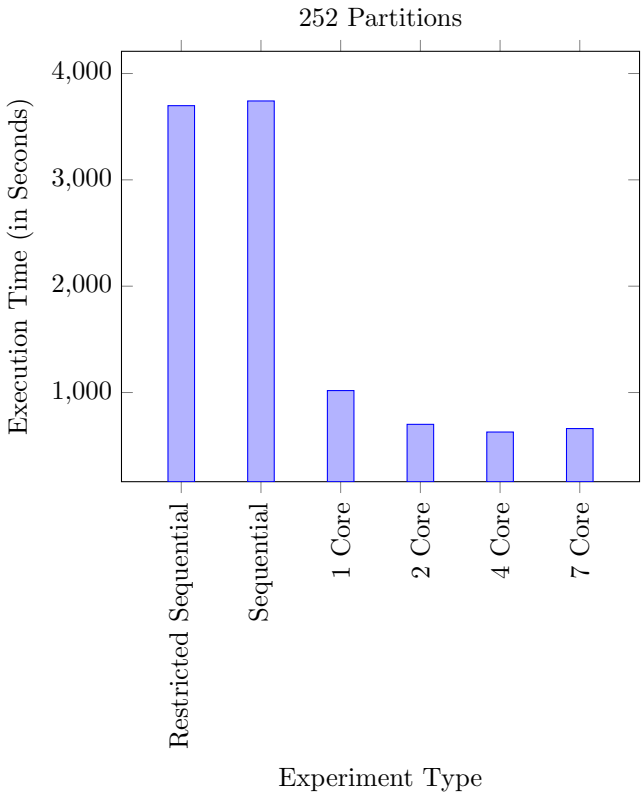


Figure 5.1: Sequential vs. Spark operation: scaling core count only

144 partitions. However, the execution time of the experiment with 252 partitions is 439 seconds, which is around 22% better than the average execution time of the previous three experiments. This suggests that there is an optimal partition count related to a core count which will bring about the best performance for a parallel operation for the specific dataset and cluster being used.

5.1.4 Spark Experiments: Scaling Core Count with Partition Count

Figure 5.3 demonstrates how increasing core count and partition count together affects Spark performance. For the experiments, the coefficient of variation between the replications is 0.07 and the experiments were repeated 4 times. Using Spark with 1 core per executor can boost performance up to 7 times compared to the sequential operation. However, the execution time does not scale further in a linear fashion as core count increases. Compared to the performance of 1 core per executor, using 7 times the cores per executor only results in 2.5 times speed improvement.

The key difference between this and the previous set of experiments is that while the previous set of experiments are changing either core count or partition count, this set of experiments is changing both factors. However, the experiment results do not differ by much compared to the experiments where only core count is increased. This, along with the result of experiment sets where only partition count is changed, suggests that the effect of dividing up the partitions into small parts may be offset by the performance

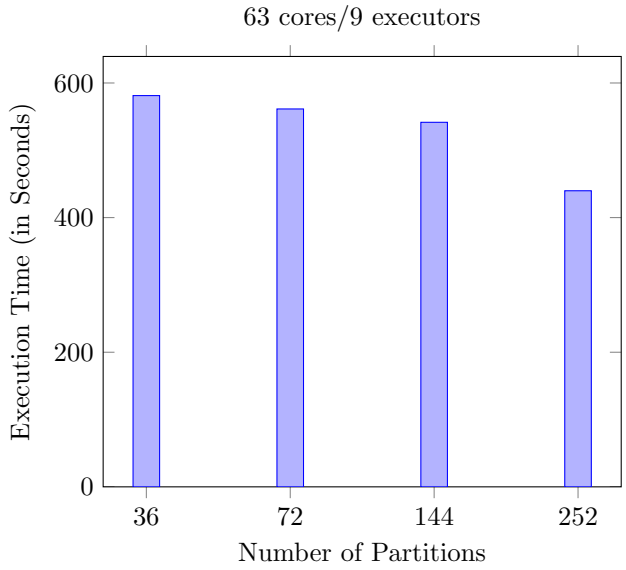


Figure 5.2: Spark operation: scaling partition count only

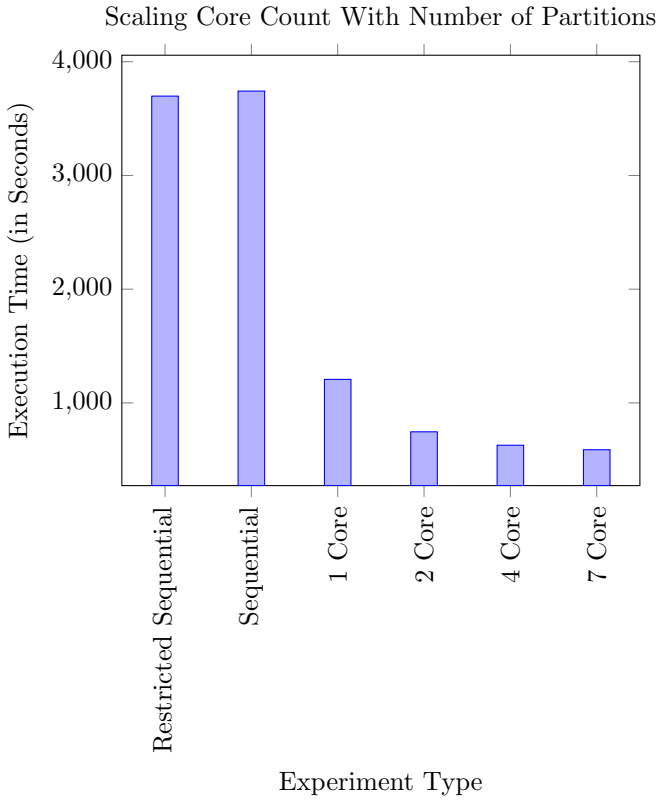


Figure 5.3: Sequential vs. Spark operation: scaling core count with number of partitions

increase due to increasing core count.

5.1.5 Spark Experiments: Lentil Dataset in Cluster 1

Figure 5.4 displays the results of the scalability experiments using the first chromosome of the lentil dataset to demonstrate how varying dataset size affects the performance and scalability of Spark in *Cluster 1*. Two replications were conducted and the coefficient of variation between the replications is 0.005.

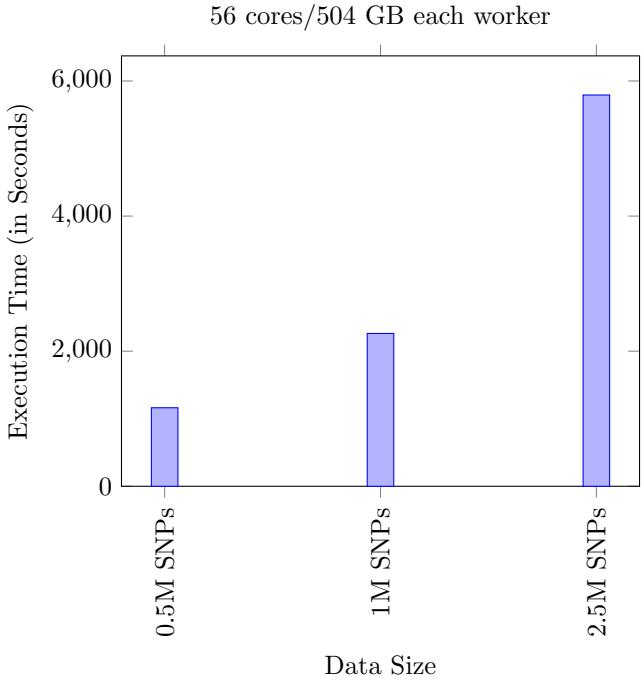


Figure 5.4: Spark experiment: Cluster 1 with lentil dataset

Unfortunately, experiment no. 19, 20 and 21 could not be executed due to Java Heap Memory errors. These experiments used 63 executors and 2 GB per memory, as described in Chapter 4. Although the actual size of the dataset is 4.65 million, the maximum amount of SNPs used in these experiments is 2.5 million. When the input dataset is being chunked and broadcasted to worker machines, the broadcasted data is initially saved in memory. As each machine in Cluster 1 had only 16 GB of RAM available, it was not possible to assign more than 2 GB RAM to each executor, so Spark threw a memory error whenever its memory limit was exceeded. The target for this set of experiments was to use more executors with less resources to see if more parallelism could be achieved. However, as the experiments were not successful, a future work would be to assign more RAM to the executors and find out if it makes any difference to the performance.

The initial experiment used 0.5 million SNPs from lentil dataset, resulting in an execution time of 1162 seconds. The next experiment used 1 million SNPs, which increased execution 1.94 times and resulted in an execution time of 2262 seconds. Finally, using 2.5 million SNPs results in an execution time of 5792 seconds,

which is a 2.56x increase in execution time compared to using 1 million SNPs. These results indicate that the rate of computation scales linearly as dataset size is increased, and Spark is able to scale performance with increasing dataset size.

5.1.6 Spark Experiments: Lentil Dataset in Cluster 2

Figure 5.5 displays the results of the experiments using the first chromosome of the lentil dataset to demonstrate how varying dataset length affects the performance and scalability of Spark in *Cluster 2*. Cluster 2 has more cores, more RAM and slower processors. Two replications were conducted and the coefficient of variation between the replications is 0.02.

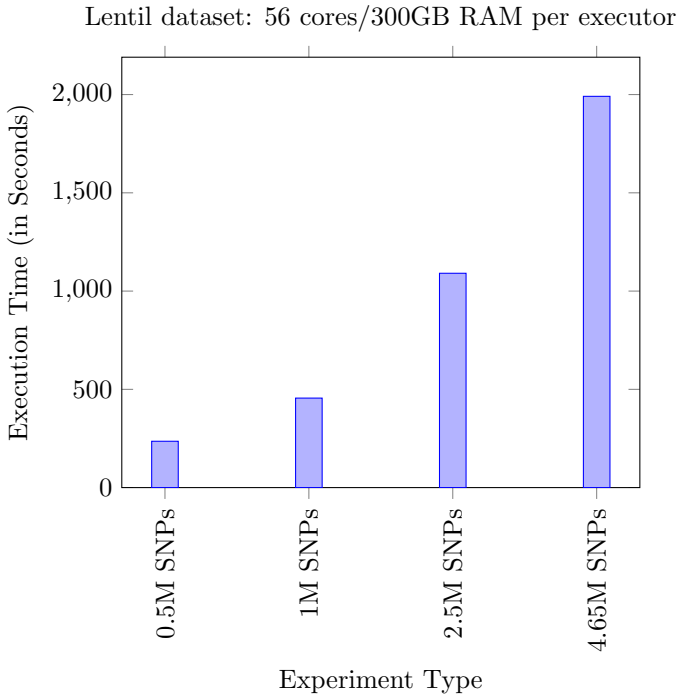


Figure 5.5: Spark experiment: Cluster 2 with lentil dataset

An attempt was made to run both chromosome 1 and 2 of the lentil dataset together - which amounted to 10 million SNPs - in Cluster 2. However, the execution encountered the same Java Heap Memory error that plagued Cluster 1 experiments. Further experiments with 1 TB and larger amount of RAM per executor are part of future work on the scalability.

The execution time for these experiments are much faster than the experiments with Cluster 1. On average, the execution time is around 5 times faster for the results of Cluster 2 when comparing with the results of Cluster 1, in spite of using only 2.5 times more cores. The rate of computation does not degrade for these experiments either.

Comparing the experiment results of Cluster 1 and Cluster 2 using lentil dataset, it can be seen that using

Cluster 2 resulted in an average of 5 times speedup compared to Cluster 1. Cluster 1 had 63 cores and 117 GB of RAM in total available to executors, whereas Cluster 2 had 156 cores and 900 GB of RAM in total available to executors. However, the processing cores of Cluster 1 had higher frequency compared to Cluster 2 (3.4 GHz compared to 2.1 GHz). As the cores of Cluster 2 are slower than Cluster 1, and both clusters use the same type of disks, a combination of memory and network speed may be responsible for the speedup. The network of Cluster 1 is capped at 1 Gbps, whereas the virtual machines in Cluster 2 share a 25 Gbps network. This indicates that network speed might be a significant reason for the speedup. To verify the effect of network speed improvement on the execution speedup, an attempt was made to run Spark experiment with lentil dataset using 2 GB RAM per executor. However, the execution encountered Java Heap Memory error, which prevented quantifying the performance difference.

5.1.7 Discussion

From the experiment results described above, it can be seen that although using Spark increases performance of TASSEL, the performance is not increasing linearly relative to resources. On average, using 7 cores per machine brings about only 2.1 times speedup compared to using 1 core per machine. This anomaly can be attributed to the way calculations occur inside map operations of the modified program. Some operations inside the map tasks for the *WeightedMLM* plugin are repeated and sequential in nature, meaning that the map tasks scale in a sub-linear fashion.

The first three experiments use the Arabidopsis dataset to measure how varying resources with fixed dataset size impacts performance. However, the experiments with lentil dataset measure how varying dataset size with fixed resources impact performance. As the lentil dataset is much bigger than the arabidopsis dataset, subsets of the data file were used initially to see if the performance scales with increasing amount of data. Using 2.5 million SNPs resulted in 5 times execution increase compared to using 0.5 million SNPs, which means the rate of computation does not degrade. This indicates that the scalability problem is not affected by the dataset size; Spark is able to scale performance with increasing dataset size.

Although Cluster 1 failed to run 4.65 million SNPs of the first chromosome of lentil dataset, Cluster 2 had no problem running it. On the other hand, Cluster 2 failed to run 10 million SNPs of the combined first and second chromosome of lentil dataset. Cluster 1 had 14 GB of memory available to each executor, whereas Cluster 2 had 300 GB memory per executor. Cluster 2 succeeded to run 4.65 million SNPs because Spark was able to store the required amount of data in memory, but the amount of memory was not enough for 10 million SNPs. A supplementary experiment was conducted on *Single Machine 2* to verify if the implemented prototype can handle larger input if more memory is provided. A standalone Spark cluster was initiated on the machine, acting as both master and worker and having 64 cores and 1.5 TB of RAM. The Spark instance had 8 executors, and each executor had 8 cores - all 64 cores were used for the experiment. Each executor was given 50 GB of memory, and the Spark driver was given 50 GB of memory as well. This TASSEL-Spark configuration was able to successfully run the complete chromosome 1 of the lentil dataset in

3025 seconds, without throwing any memory errors. This result indicates that increasing the memory of each executor can result in the implemented prototype handling more data. More work is needed to understand the relationship between dataset size and amount of memory required. The physical memory of Cluster 1 cannot be expanded, but configurations available in Cluster 2 will permit this scalability evaluation.

In the TASSEL experiments, the number of cores used in experiments of Cluster 1 were 1, 2, 4 and 7, while the number of partitions used in the same cluster were 36, 72, 144 and 252. Trying out all the combinations between the core count and the partition count would lead to a full factorial experiment. However, the experiments were not completed due to their running length. Performing the full factorial experiment can be an important future work, which may lead to greater insight on whether the core count only matters at some partition sizes.

From the discussions above and the insights from the experiment results, three conclusions can be drawn. The first conclusion is that the performance of the prototype implementation does not scale with executor size in terms of CPU cores. Although the experiments exhibit increased performance compared to sequential implementations, the sequential code blocks inside the implementation still hampers the scalability of the implementation, resulting in sub-linear scalability. Fixing this problem requires further optimization of the application, which will involve finding more sequential code blocks and making them work with parallel execution. The second conclusion is that changing the number of partitions does not affect performance substantially. This means that it is best to let the parallel processing frameworks handle the partition amount and optimize the operation in the best way possible. The third conclusion is that increasing the size of dataset results in scalable performance - execution time decreases at the same scale of increasing dataset size when resources are fixed. This means that the size of the dataset does not affect scalability of performance. The fourth conclusion is that even in cluster operations, not having enough memory in worker machines can hamper working with large datasets. It is therefore imperative to have enough memory in worker machines so that parallel operations do not face memory problems. Furthermore, discovering what level of resources are required for each experimental scenario does not seem to be trivial.

5.2 Experiment Results for FaST-LMM Case Study

In this section, the baseline experiment results are described first. Then the results of Dask and Spark experiments respectively, using the Arabidopsis dataset are described. For both sets of experiments, the results of experiments where the number of cores are increased only are provided first. Next, the results of experiments where the number of partitions are increased only will be described. Afterwards, the results of experiments where the number of cores is scaled with the number of partitions are described. These results indicate that similar to TASSEL experiments, using the prototype implementation results in better performance compared to original implementation, but the performance does not scale linearly. Finally, a comparison is conducted between the results of experiments done using Dask and Spark. The experimental

results demonstrate that there is little difference between the results of these two frameworks.

5.2.1 Baseline Experiments

Table 5.3 describes the results for the baseline experiments conducted on *Single Machine 2*. As described in the previous chapter, the experiment with local runner uses one core, while the experiments with multiproc runner use multiple cores. The local runner with one core has the worst performance. However, the performance of multiprocessor runner with 8 cores is better than that of 64 cores. This result is similar to the TASSEL baseline experiments detailed in Section 5.1.1, where using more cores resulted in degraded performance. As explained, a potential reason for the degraded performance can be the overhead in switching between cores during multicore execution.

Table 5.3: Baseline experiments with original, unmodified FaST-LMM

Exp. No.	No. of Cores	Runner type	Exec. Time (seconds)
1	1	Local	10513
2	8	Multiproc	7695
3	64	Multiproc	9050

5.2.2 Dask and Spark Experiments: Increasing Core Count Only

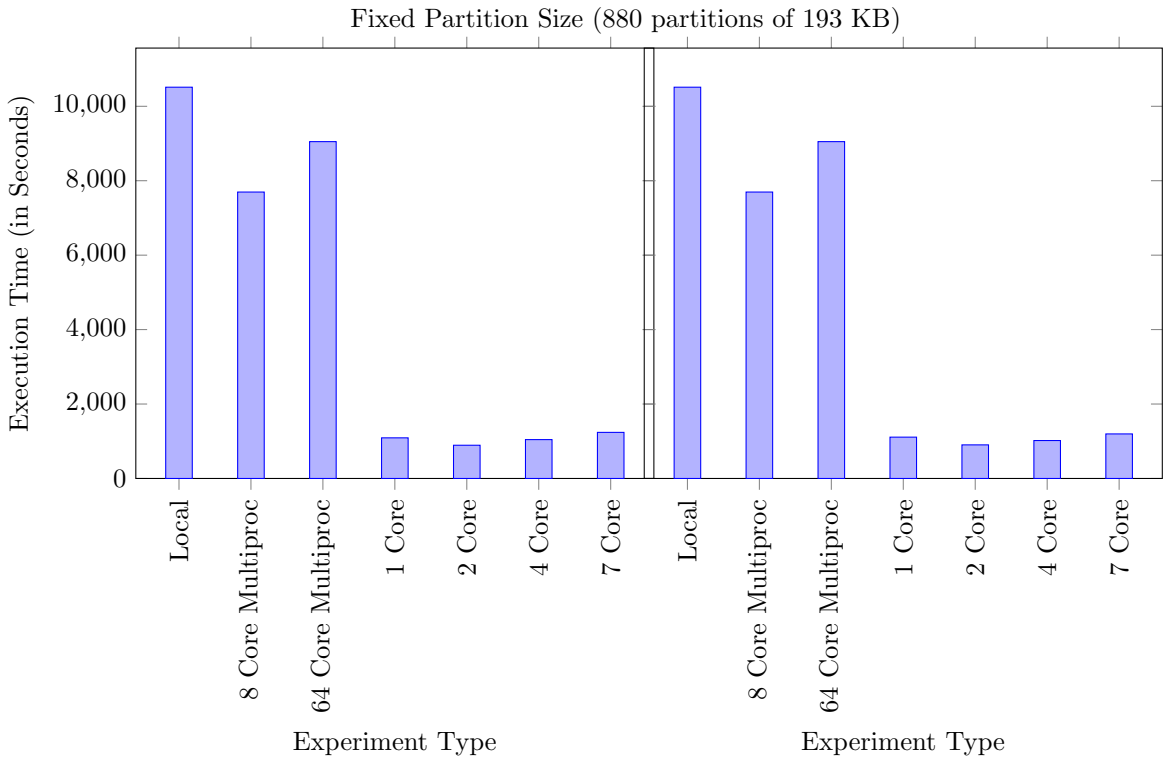
From Figure 5.6, using Dask and Spark with 1 core per worker can boost speed up to 12 times compared to local execution, and up to 8.5 times compared to multicore execution. However, the execution time increases very slightly as core count increases upwards from 2 cores per machine. The execution time scales down to a minimum from 1 core to 2 cores per machine, then gradually increases until the core count is 7 per machine. Both the Dask and Spark experiments were replicated 3 times. The coefficient of variation between the replications for Spark experiments is 0.51, while for Dask experiments it is 1.31.

5.2.3 Dask and Spark Experiments: Increasing Partition Count Only

From Figure 5.7, there seems to be very little difference between the execution time of the experiments. The target of the experiments was to determine the difference between the results of each core configuration of Dask and Spark. Table 5.4 displays the average, standard deviation and coefficient of variation between

Table 5.4: Statistics for Experiment Results of Figure 5.7

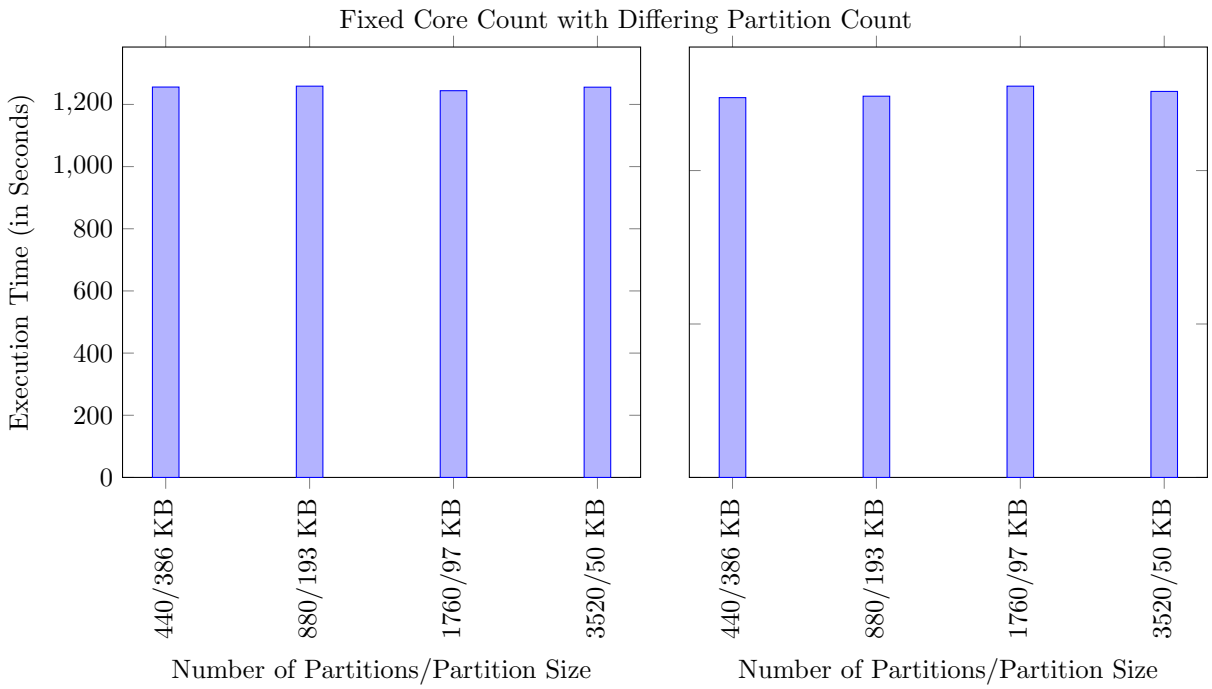
Framework	Average	Std. Dev.	COV
Dask	1253.59	5.57	0.004
Spark	1253.21	14.64	0.012



(a) Dask: increasing core count only

(b) Spark: increasing core count only

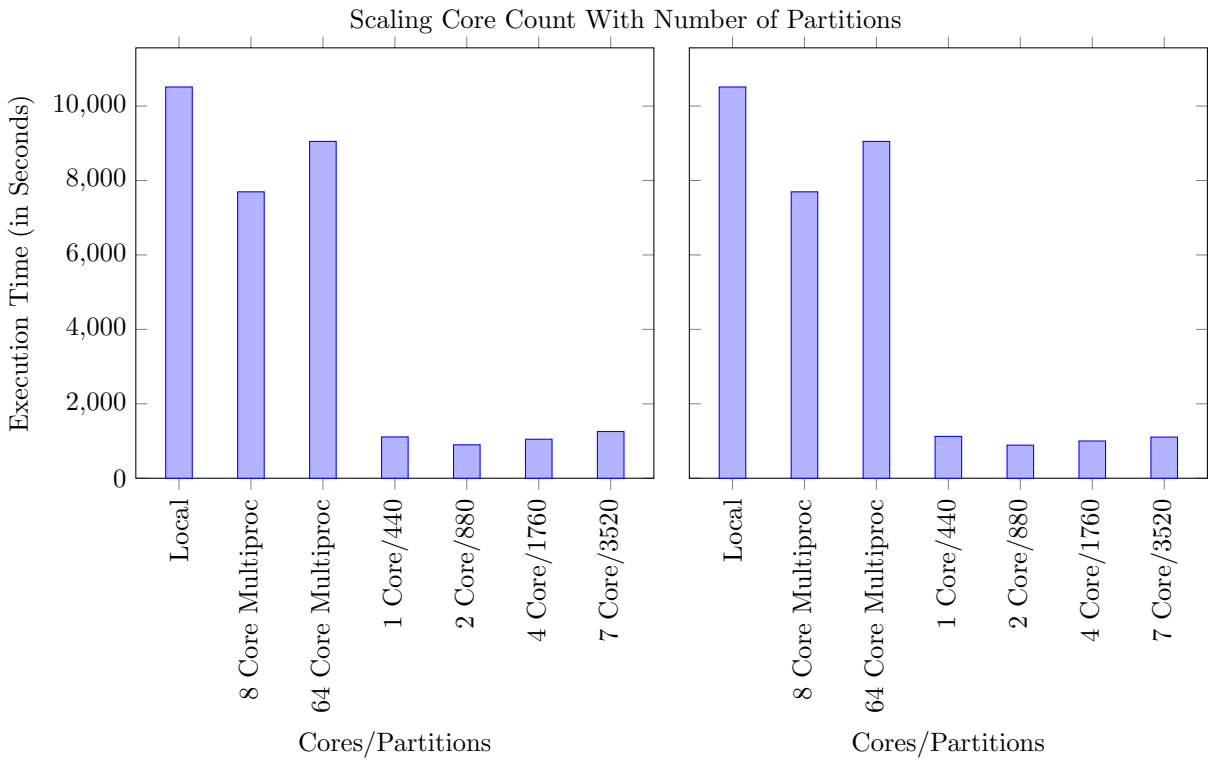
Figure 5.6: Sequential vs. Multicore vs. Dask/Spark operation: increasing core count only



(a) Dask: increasing partition count only

(b) Spark: increasing partition count only

Figure 5.7: Sequential vs. Multicore vs. Dask/Spark operation: increasing partition count only



(a) Dask: scaling core count and number of partitions **(b)** Spark: scaling core count and number of partitions

Figure 5.8: Sequential vs. Multicore vs. Dask/Spark operation: scaling core count and number of partitions

results for each core, categorized by the frameworks being used. The coefficient of variations for both Dask and Spark are relatively low, less than 0.02. This suggests that for FaST-LMM under both Dask and Spark operations and for the specific dataset and cluster being used, partition count has little effect on execution time; dividing the dataset more has no effect. However, the experiments have smaller partition size compared to the traditional ones used by Dask and Spark. The size of the Arabidopsis dataset is 170 MB, which means that 440 partitions will result in partition size of 386 KB and 3520 partitions will result in partition size of 50 KB. In comparison, the default partition size in Spark configuration is 128 MB. Therefore, it cannot be claimed with certainty that partition count has little effect on execution time when compared with Spark defaults.

5.2.4 Dask and Spark Experiments: Scaling Core Count with Partition Count

From Figure 5.8, using Dask and Spark with 1 core per worker can boost speed up to 12 times compared to sequential execution, and up to 8.5 times compared to multicore execution. However, instead of gradual reduction of execution time with increasing core count, the execution time increases as core count increases upwards from 2 cores per machine. The execution time scales down to a minimum from 1 core to 2 cores per machine, then gradually increases until the core count is 7 per machine.

Comparing the results of the previous sets of experiments which focuses on increasing core count and partition count individually, the result for this set of experiments is similar to the results of experiments with increasing core count only, and execution time remains similar for the experiments with increasing partition count.

The results of the experiments with FaST-LMM and Spark mirror the results of the experiments with FaST-LMM and Dask closely. In both sets of experiments, the execution time decreases for 2 cores and then increases for 7 cores. This suggests that both Dask and Spark perform similarly when given the same amount of resources. This conjecture will be validated in the next set of experiments, as the performance of Dask and Spark are compared more closely.

5.2.5 Performance Comparison of Dask and Spark

Comparison of Increasing Core Count only between Dask and Spark

Each experiment in Figure 5.9 was replicated 3 times. While Spark has slight advantage over Dask while running on 4 and 7 cores per machine, Dask regains the slight advantage while running on 1 and 2 cores per machine. The little difference of execution time shows that Dask and Spark are evenly matched for running parallel tasks when partition count remains static and core count increases.

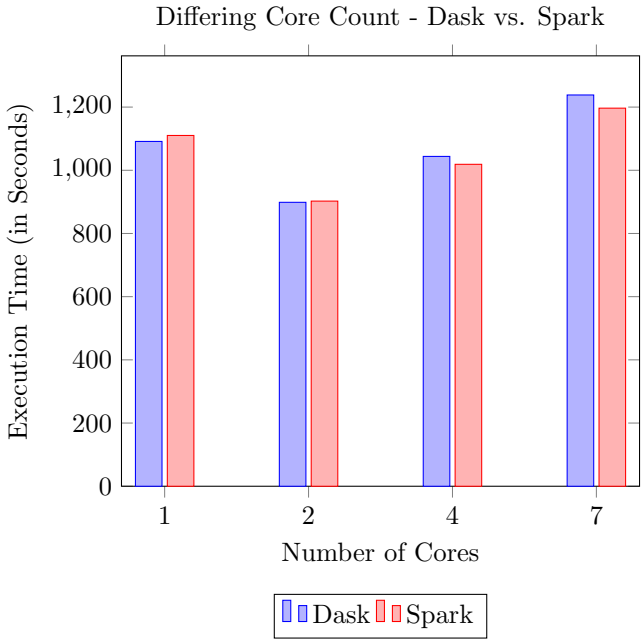


Figure 5.9: Dask vs. Spark: increasing core count only

Comparison of Increasing Partition Count only between Dask and Spark

From Figure 5.10, there is again little difference between the performance of Dask and Spark. The frameworks are evenly matched with the number of partitions as an experimental factor.

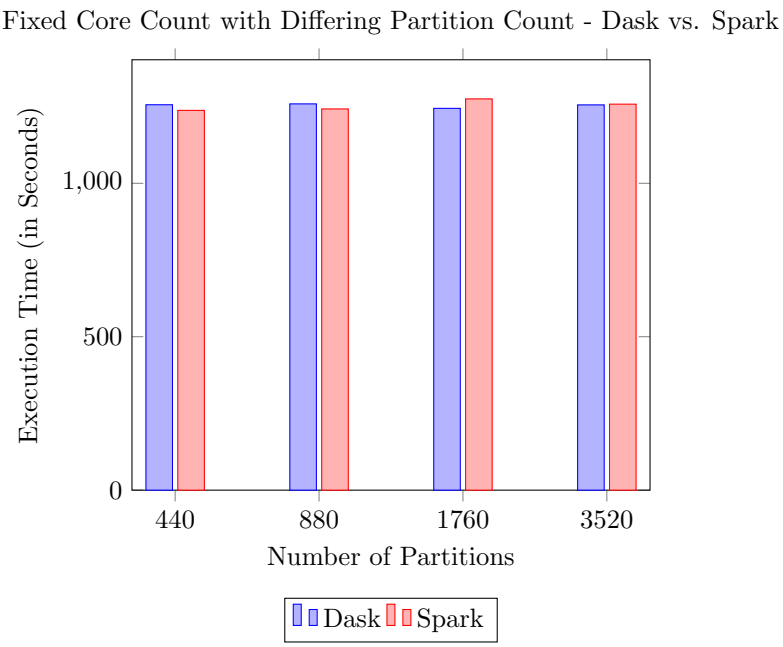


Figure 5.10: Dask vs. Spark: increasing partition count only

Comparison of Scaling Core Count with Partition Count between Spark and Dask

Figure 5.11 provides the zoomed in comparison between Dask and Spark. The little difference of execution time shows that Dask and Spark are almost evenly matched for running parallel tasks when scaling core count with partition count, with Spark having slight advantage on some cases.

5.2.6 Discussion

Compared to the experiments done with TASSEL and Spark, there are a number of differences in the experiments with FaST-LMM on Dask and Spark. The first difference is how the performance scales as core count is increased. In the experiments with TASSEL and Spark, execution time decreased as core count increased, but the scalability was sub-linear. In the experiments with FaST-LMM, Dask and Spark, not only the scalability was sub-linear, but the execution time actually increased as more cores were used. On average, using 7 cores per machine resulted in a 1.08x execution time increase compared to using 1 core per machine. The reason for the suboptimal result of the experiments is the redundant operation problem which was prevalent in the Spark experiments as well. There are sequential and repeated code blocks inside the map tasks of FaST-LMM which prevent the tasks from scaling with core count increase. Using more cores means more time spent in repeated operations, possibly resulting in increased execution time compared to using lower number of cores.

Unfortunately, it was not possible to use the lentil dataset with FaST-LMM, as FaST-LMM was not able to read the dataset because of data formatting issues that have not been resolved yet. Despite FaST-LMM not being able to use the lentil dataset, the experiments demonstrated enough evidence that the implementation

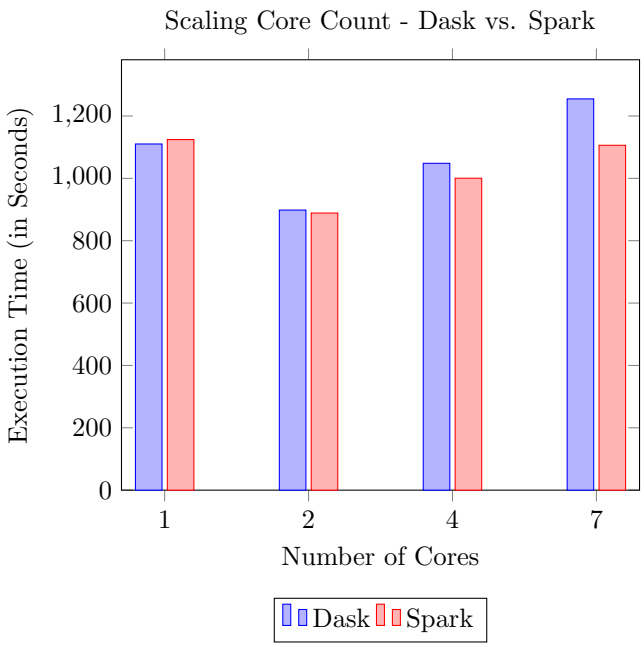


Figure 5.11: Dask vs. Spark: scaling core count per executor and partition count

of the proposed architecture is able to handle large datasets for GWAS operations. Moreover, the slowdown of operations observed in FaST-LMM can be an indication that using TASSEL may have more benefit in terms of scalability and speedup, compared to FaST-LMM. Future work can compare similar operations between TASSEL and FaST-LMM to determine which application performs better in diverse scenarios. This will require modification to the input module and/or data format.

From the above discussion, two conclusions can be drawn. The first conclusion is that similar to the experiments with TASSEL and Spark: the performance of the prototype implementation does not scale with the amount of resources. Although the experiments exhibit increased performance compared to sequential implementations, increasing the number of cores does not result in increased performance. Fixing this problem requires further optimization of how FaST-LMM operates, which may involve finding sequential code blocks and making them work with parallel execution. Comparing the performance of TASSEL and FaST-LMM, the optimizations selected for TASSEL provides more scalability than the corresponding optimization in FaST-LMM. The second conclusion is that changing the number of partitions does not affect performance, which is also similar to the conclusion drawn for TASSEL and Spark.

5.3 Evaluation results using feedback from experts

Table 5.5 lists the closed-ended questions and the responses from the domain experts. The first three questions ask about the architecture in general. For the first question, all experts agreed that it would be valuable for them to make a GWAS application faster using an architecture that utilizes multiple machines. Although most experts are clear on the topic asked on the second qeustion, two experts stated that they are unclear

Table 5.5: Closed-Ended questions

Questions	Answers		
	Yes	No	No Comment
Q1. If an architecture is proposed where multiple machines are used for executing a GWAS application faster, would you find it valuable?	**9**	**0**	**0**
Q2. Does the architecture clearly explain how reusable modules can be used to gain performance improvement?	**7**	**2**	**0**
Q3. Is the module reusability for performance gain in the architecture useful for you?	**9**	**0**	**0**
Q4. Is the performance benefit compared to the original implementation in the experiments useful for you?	**9**	**0**	**0**
Q5. Is the implementation of the proposed architecture easy enough for you to use?	**8**	**1**	**0**
Q6. Do you believe the proposed architecture is extendible towards other GWAS applications?	**5**	**0**	**3**

about the explanation due to its generalized manner, and would prefer more detailed explanation. For the third question, all experts agreed that the module reusability would be quite useful for them.

The next two questions ask about the implementation of the architecture. All experts acknowledged in Question 4 that the performance gain of the implementation prototype is indeed useful for them. All but one experts found the implementation easy enough to use, when answering Question 5.

To answer the last question, 5 experts believed that the proposed architecture can be extended towards other GWAS applications; 3 experts had no comment on the question, as they lacked enough information to answer the question.

The benefit most commonly acknowledged by the experts for the first open-ended question is the fast computation of the implementation. The experts believed that the speedup of the implementation would reduce time to conduct important genome studies. Some experts also stated that the implementation would be easier to use and configure for end users without sufficient background in computer science. Others talked about the benefit for developers - easier implementation, separation of concerns, adaptability to layer architecture and extendibility to other tools. A few experts also noted that the use of shared data cache module can result in lower latency and fast retrieval of cached data.

The drawback most commonly mentioned by the experts for the second open-ended question is the lack of a GUI system. The prototype implementation of the architecture is operated from terminal and uses command line arguments to switch modules. Although the experts found the implementation easy enough to use, they also expressed the opinion that using a GUI system would make it much easier to run GWAS operations and switch modules as needed. Some experts also commented on the generalization of the architecture and stated

that the tradeoff of generalization might be a burden for developers during implementation. The developers also have a big role to play for the usability and configurability of the implementation, and some experts are of the opinion that the implementation may depend too much on how good the developer is. Finally, a bioinformatics expert commented that there are different ways of doing GWAS studies and further work need to be done to ensure that the proposed architecture is applicable in most cases.

5.4 Chapter Discussion

A few more general observations can be made from the case studies conducted using TASSEL and FaST-LMM. The first observation is the lack of linear scaling in the experiments. The prototype implementations for both TASSEL and FaST-LMM had sub-linear scalability. The given explanation was that there are sequential and repeated code blocks inside the parallel implementations of TASSEL and FaST-LMM, which prevented the parallel tasks from scaling with core count increase. Solving this problem requires much more optimization than plugging in a parallel processing module. The best performing parallel tasks are those which are embarrassingly parallel - tasks which can be parallellized without any sequential dependancy. Sufficient effort to optimize sub-linear section of operations in TASSEL and FaST-LMM may result in increased performance and scalability, but it will be difficult to make these tasks completely independent. Therefore, it is wiser to expect sub-linear scalability in GWAS operations and set the target to optimize the operations and achieve as much scalability as possible. This optimization can come in the form of caching frequently accessed data, buffering data in memory and efficient algorithms.

In addition to the sub-linear scalability problem, the experiments also suffered from memory overflow errors. When handling the lentil dataset, TASSEL was not able to run more than 4.65 million SNPs even with 300 GB of RAM provided per executor. The complete lentil dataset has around 32 million SNPs. Optimizing the parallel tasks as mentioned previously would go a long way towards fitting more SNPs using less RAM. However, fixing the way the input dataset is loaded into the application is paramount towards loading big data efficiently. Both TASSEL and FaST-LMM loads the input dataset directly into memory. Additionally, they encapsulate each row of data into an object, which takes even more space in memory than the raw dataset does. This makes it difficult to load a big dataset without having substantial amount of memory in a machine. An intuitive solution to this problem can be using a buffer to store and operate on input dataset. The buffer can detect the amount of memory available to the application and operate on the input dataset in batches. This will prevent the memory overflow problem. However, in the perspective of the architecture proposed in this thesis, a better solution could be implementing a file operation module specifically designed to work with traditional distributed file systems such as HDFS. Together with a buffer, this will not only prevent memory overflow, but also enable storing and operating on big data scale datasets.

Furthermore, the API for the file operation module can be made interoperable with the parallel processing modules, which means that parallel operations can be directly run on the input dataset without loading all

of it in memory first. This can be an important future work.

The case studies used the number of partitions as an experiment factors. This resulted in the experiments having smaller partition size than the traditional ones. For example, in one of the experiments, 3520 partitions were used for the arabidopsis dataset, the size of which is 170 MB. This means that the size of each partition was 50 KB on average. Compared to this, the default partition size in Spark configuration is 128 MB. In a distributed filesystem such as HDFS, data is stored in fixed partition size, which usually can be in the range of multiples of megabytes. Although the size of the arabidopsis dataset is relatively small compared to a larger dataset like the lentil dataset, the decision to use partition count instead of partition size was a design choice. As explained in Section 5.1.1, both TASSEL and FaST-LMM stores each row of the input dataset as an object, so a representation of the dataset is partitioned instead of the dataset. The experiment results demonstrated that varying the number of partitions do not influence performance by much. However, the conclusion is not solid due to two reasons - the arabidopsis dataset was much too small, and the experiments were not replicated enough. A future work can focus on varying the number of partitions in different scenarios and observe how it affects performance.

The final observation can be made on how much the end users can benefit from the architecture implementation with regards to the evaluation feedback. The feedback demonstrated a need for fast GWAS operation with easy to use interface. Although the experiments did demonstrate that the implementations performed much faster than the original applications, the implementations were not able to run datasets having more than 4.7 million SNPs and to linearly scale performance. As discussed above, making the implementations run on larger datasets on a linear scale requires substantial amount of work. The implementations can run datasets in the range of gigabytes; however, running datasets in the range of terabytes require further work. Moreover, the parallellization of TASSEL and FaST-LMM was performed on a chosen GWAS plugin of each application. Running other GWAS plugins will require finding the bottlenecks of those plugins and plugging in the implemented modules on a case by case basis. It is possible to do this as evidenced by the experiments. Whether it is more beneficial to continue to optimize such implementations or use entirely modular designs from scratch is beyond the scope of this research. One of the contributions of this thesis is that it is now much easier to get a basic level of speedup from most available GWAS plugins by finding the bottlenecks and plugging in the parallel processing module. However, as mentioned previously, achieving scalable performance requires further optimization.

5.5 Summary

In conclusion, this chapter reports the results of the case study experiments as well as the architecture evaluation feedback described in Chapter 4. The case study experiments demonstrated that the implementations of the proposed architecture performed much better than the original programs. However, none of the implementations achieved linear scalability because there were sequential code blocks with repeated operations

inside every map tasks of the implementations. As core count increases, the repetition of the operations also grow in number, resulting in sub-linear scalability, that is, the performance does not scale with resource increase. The experiments with varying number of partitions and fixed resources demonstrated that the performance does not differ much when changing the amount of partitions. In addition to the Arabidopsis dataset, a lentil dataset was also used for a number of experiments. These experiments demonstrated that the performance scales down linearly with the size of the dataset. A comparison of performance between Dask and Spark using FaST-LMM indicated that there is not much performance difference between these two frameworks.

The architecture evaluation garnered mostly positive feedback from domain experts, who found the proposed architecture and its implementations useful, performance oriented and easy to use. However, the experts also noted the lack of GUI in the implementations and the tradeoff of architecture generalization the developers have to face.

6 Conclusion and Future Work

6.1 Conclusion

In this thesis, a modular pipeline architecture for GWAS applications that can improve performance by leveraging parallel processing and data cache was designed, implemented and evaluated. Although existing GWAS applications implement performance optimizations to process large volumes of data, researchers can find it difficult to adapt to the various workflows and data types these applications represent. Due to the number of GWAS application options that researchers have, a modular architecture can better suit their needs. Loosely coupled modules can represent GWAS algorithms and parallelization techniques can be encapsulated via these reusable modules.

The goal of this thesis was to build a usable, high-performing modular pipeline architecture that software developers can use to configure, deploy and maintain GWAS applications to be used in various analysis scenarios by bioinformaticians. The architecture supports interfaces for modular development, a shared data cache for lower latency access to data and the capability to leverage parallel computation in a distributed environment.

Two case studies were conducted in which the proposed architecture was implemented and evaluated to prove its usefulness and efficiency. In the first case study, a genomic analysis pipeline TASSEL was chosen to be modified using the proposed architecture. For the specific implementation, the Apache Spark parallel processing framework and the Redis cache store were modularized. For the second case study, the proposed architecture was used to implement the FaST-LMM GWAS application. Apache Spark and Dask were the chosen parallel processing frameworks to be modularized, while Redis still remained the cache store.

Based on the case studies, a number of experiments were set up to evaluate the performance benefit of the implementations. There were a few key findings of the performance experiments. The first finding was that although implementing the proposed architecture does indeed result in improved execution time, the performance does not scale as resources are increased, that is, the performance displays sub-linear scalability. Scalability of an application is usually tied to the nature of an application, and if there are sequential code blocks in parallel tasks, they will not scale with the rest of the operation. This lack of scalability on speedup can also be related to Amdahl's law, which states that non-parallel portions of a code limit the maximum speedup possible from parallelization. However, despite the limited scalability in the implementations, it has been demonstrated that it is possible to take advantage of obvious parallelization with the proposed architecture.

The second finding of the experiments was that changing the number of partitions (for the Arabidopsis dataset) or the dataset size (for the lentil dataset) has little impact on the scalability of execution. From the Arabidopsis experiments, it was demonstrated that varying the number of partitions results in almost similar execution time when the resources are fixed. Furthermore, as demonstrated by the experiments with the lentil dataset, varying dataset size in a fixed resource environment results in linear speedup; for example, if dataset size is increased by twofold, then the speed should also slowdown twofold.

The third finding of the experiments was that between TASSEL and FaST-LMM, TASSEL displayed more stability in terms of performance and scalability. Using TASSEL with Spark in the implemented prototype resulted in sub-linear scalable execution; however, using FaST-LMM with both Spark and Dask resulted in decreased performance when increasing core count. This suggests that FaST-LMM has more sequential code blocks in its GWAS implementation than TASSEL has.

The final finding of the experiments was that in a cluster environment, the memory amount in the worker machines limits the amount of data that can be operated on in parallel. As demonstrated in the experiments with the lentil dataset, Spark could only handle a specific amount of SNPs in its worker memory before it threw a memory error. This suggests that there is an opportunity to optimize GWAS applications and parallel processing frameworks so that more data can be handled, and also that even in a cluster environment, having sufficient memory in a worker machine is imperative.

In addition to performance experiments, an architectural evaluation was also conducted to receive feedback on the proposed architecture from domain experts. A questionnaire including open-ended and closed-ended question was prepared for the evaluation. In the feedback, the domain experts supported the necessity of the proposed architecture and agreed that the implementation of the architecture is useful for them, is easy enough to use and the performance benefit is noticeable. However, some experts pointed out that a GUI implementation would be easier to use, the implementation quality will largely depend on developers and the extendability of the architecture on other GWAS applications needs further work to be ensured.

6.2 Future Work

Although the architecture was built for GWAS applications, it can also be used in cases where an application with a pipeline architecture needs to increase performance with minimal change. Further work can be undertaken with these applications of different domains by implementing the proposed architecture and observing the performance benefits. The architecture can also be extended to fit in with workflow other than pipelines, as modular components can work with other paradigms as well.

As observed from the experiment results, sequential parts in the parallel tasks prevent the parallel operations from achieving linear speedup. In the future, steps can be taken to optimize such GWAS operations to reduce sequential parts and increase linear scalability. Additionally, the file load operation to read a dataset into memory by TASSEL and FaST-LMM prevented the implementations from working with a larger

dataset. Further work can be undertaken to implement a file operation module as part of the demonstrated architecture that can work with traditional distributed file systems and be interoperable with the parallel processing modules.

An advantage of implementing the proposed architecture for GWAS applications is that it can be integrated into more extensive bioinformatics workflows. Large bioinformatics projects often require orchestrating workflows where multiple pipelines are executed. These orchestrations are usually provisioned by automation engines such as Ansible,[1] Puppet[2] and Chef.[3] Further work can be done to integrate with such workflows so as to improve performance and so that the workflows can operate in multiple environments.

A number of potentially useful experiments could be conducted to gain further insights. Full factorial experiments for the core count and the partition count may lead to greater insight on whether the core count only matters at some partition sizes. In addition, it would be worthwhile to vary the number of partitions in different scenarios to observe how it affects performance. Finally, assigning more RAM to the executors to find out performance impact would also be of interest.

Although several experiments were completed with TASSEL and FaST-LMM in this thesis, there has not been an appropiate comparison of performance between these two packages. A future work can compare similar operations between TASSEL and FaST-LMM to determine which application performs better in diverse scenarios.

[1] Ansible, Simple IT Automation - `https://www.ansible.com`
[2] Puppet - `https://puppet.com`
[3] Chef - Automate IT Infrastructure - `https://www.chef.io/chef/`

References

[1] Bruce Alberts, Alexander Johnson, Julian Lewis, David Morgan, Martin Raff, Keith Roberts, Peter Walter, John Wilson, and Tim Hunt. Molecular biology of the cell. *The Quarterly Review of Biology*, 90(3):343–343, Sep 2015.

[2] Hana Lango Allen, Karol Estrada, Guillaume Lettre, Sonja I. Berndt, Michael N. Weedon, Fernando Rivadeneira, Cristen J. Willer, Anne U. Jackson, Sailaja Vedantam, Soumya Raychaudhuri, Teresa Ferreira, Andrew R. Wood, Robert J. Weyant, Ayellet V. Segrè, Elizabeth K. Speliotes, Eleanor Wheeler, Nicole Soranzo, Ju-Hyun Park, Jian Yang, Daniel Gudbjartsson, Nancy L. Heard-Costa, Joshua C. Randall, Lu Qi, Albert Vernon Smith, Reedik Mägi, Tomi Pastinen, Liming Liang, Iris M. Heid, Jian'an Luan, Gudmar Thorleifsson, Thomas W. Winkler, Michael E. Goddard, Ken Sin Lo, Cameron Palmer, Tsegaselassie Workalemahu, Yurii S. Aulchenko, Åsa Johansson, M. Carola Zillikens, Mary F. Feitosa, Tõnu Esko, Toby Johnson, Shamika Ketkar, Peter Kraft, Massimo Mangino, Inga Prokopenko, Devin Absher, Eva Albrecht, Florian Ernst, Nicole L. Glazer, Caroline Hayward, Jouke-Jan Hottenga, Kevin B. Jacobs, Joshua W. Knowles, Zoltán Kutalik, Keri L. Monda, Ozren Polasek, Michael Preuss, Nigel W. Rayner, Neil R. Robertson, Valgerdur Steinthorsdottir, Jonathan P. Tyrer, Benjamin F. Voight, Fredrik Wiklund, Jianfeng Xu, Jing Hua Zhao, Dale R. Nyholt, Niina Pellikka, Markus Perola, John R. B. Perry, Ida Surakka, Mari-Liis Tammesoo, Elizabeth L. Altmaier, Najaf Amin, Thor Aspelund, Tushar Bhangale, Gabrielle Boucher, Daniel I. Chasman, Constance Chen, Lachlan Coin, Matthew N. Cooper, Anna L. Dixon, Quince Gibson, Elin Grundberg, Ke Hao, M. Juhani Junttila, Lee M. Kaplan, Johannes Kettunen, Inke R. König, Tony Kwan, Robert W. Lawrence, Douglas F. Levinson, Mattias Lorentzon, Barbara McKnight, Andrew P. Morris, Martina Müller, Julius Suh Ngwa, Shaun Purcell, Suzanne Rafelt, Rany M. Salem, Erika Salvi, Serena Sanna, Jianxin Shi, Ulla Sovio, John R. Thompson, Michael C. Turchin, Liesbeth Vandenput, Dominique J. Verlaan, Veronique Vitart, Charles C. White, Andreas Ziegler, Peter Almgren, Anthony J. Balmforth, Harry Campbell, Lorena Citterio, Alessandro De Grandi, Anna Dominiczak, Jubao Duan, Paul Elliott, Roberto Elosua, Johan G. Eriksson, Nelson B. Freimer, Eco J. C. Geus, Nicola Glorioso, Shen Haiqing, Anna-Liisa Hartikainen, Aki S. Havulinna, Andrew A. Hicks, Jennie Hui, Wilmar Igl, Thomas Illig, Antti Jula, Eero Kajantie, Tuomas O. Kilpeläinen, Markku Koiranen, Ivana Kolcic, Seppo Koskinen, Peter Kovacs, Jaana Laitinen, Jianjun Liu, Marja-Liisa Lokki, Ana Marusic, Andrea Maschio, Thomas Meitinger, Antonella Mulas, Guillaume Paré, Alex N. Parker, John F. Peden, Astrid Petersmann, Irene Pichler, Kirsi H. Pietiläinen, Anneli Pouta, Martin Ridderstråle, Jerome I. Rotter, Jennifer G. Sambrook, Alan R. Sanders, Carsten Oliver Schmidt, Juha Sinisalo, Jan H. Smit, Heather M. Stringham, G. Bragi Walters, Elisabeth Widen, Sarah H. Wild, Gonneke Willemsen, Laura Zagato, Lina Zgaga, Paavo Zitting, Helene Alavere, Martin Farrall, Wendy L. McArdle, Mari Nelis, Marjolein J. Peters, Samuli Ripatti, Joyce B. J. van Meurs, Katja K. Aben, Kristin G. Ardlie, Jacques S. Beckmann, John P. Beilby, Richard N. Bergman, Sven Bergmann, Francis S. Collins, Daniele Cusi, Martin den Heijer, Gudny Eiriksdottir, Pablo V. Gejman, Alistair S. Hall, Anders Hamsten, Heikki V. Huikuri, Carlos Iribarren, Mika Kähönen, Jaakko Kaprio, Sekar Kathiresan, Lambertus Kiemeney, Thomas Kocher, Lenore J. Launer, Terho Lehtimäki, Olle Melander, Tom H. Mosley Jr, Arthur W. Musk, Markku S. Nieminen, Christopher J. O'Donnell, Claes Ohlsson, Ben Oostra, Lyle J. Palmer, Olli Raitakari, Paul M. Ridker, John D. Rioux, Aila Rissanen, Carlo Rivolta, Heribert Schunkert, Alan R. Shuldiner, David S. Siscovick, Michael Stumvoll, Anke Tönjes, Jaakko Tuomilehto, Gert-Jan van Ommen, Jorma Viikari, Andrew C. Heath, Nicholas G. Martin, Grant W. Montgomery, Michael A. Province, Manfred Kayser, Alice M. Arnold, Larry D. Atwood, Eric Boerwinkle, Stephen J. Chanock, Panos Deloukas, Christian Gieger, Henrik Grönberg, Per Hall, Andrew T. Hattersley, Christian Hengstenberg, Wolfgang Hoffman, G. Mark Lathrop, Veikko Salomaa, Stefan Schreiber, Manuela Uda, Dawn Waterworth, Alan F. Wright, Themistocles L. Assimes, Inês Barroso, Albert Hof-

man, Karen L. Mohlke, Dorret I. Boomsma, Mark J. Caulfield, L. Adrienne Cupples, Jeanette Erdmann, Caroline S. Fox, Vilmundur Gudnason, Ulf Gyllensten, Tamara B. Harris, Richard B. Hayes, Marjo-Riitta Jarvelin, Vincent Mooser, Patricia B. Munroe, Willem H. Ouwehand, Brenda W. Penninx, Peter P. Pramstaller, Thomas Quertermous, Igor Rudan, Nilesh J. Samani, Timothy D. Spector, Henry Völzke, Hugh Watkins, James F. Wilson, Leif C. Groop, Talin Haritunians, Frank B. Hu, Robert C. Kaplan, Andres Metspalu, Kari E. North, David Schlessinger, Nicholas J. Wareham, David J. Hunter, Jeffrey R. O'Connell, David P. Strachan, H.-Erich Wichmann, Ingrid B. Borecki, Cornelia M. van Duijn, Eric E. Schadt, Unnur Thorsteinsdottir, Leena Peltonen, André G. Uitterlinden, Peter M. Visscher, Nilanjan Chatterjee, Ruth J. F. Loos, Michael Boehnke, Mark I. McCarthy, Erik Ingelsson, Cecilia M. Lindgren, Gonçalo R. Abecasis, Kari Stefansson, Timothy M. Frayling, and Joel N. Hirschhorn. Hundreds of variants clustered in genomic loci and biological pathways affect human height. *Nature*, 467(7317):832–838, Sep 2010.

[3] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: an extensible system for design and execution of scientific workflows. In *16th International Conference on Scientific and Statistical Database Management*, pages 423–424, Santorini Island, Greece, June 2004.

[4] Carl A. Anderson, Gabrielle Boucher, Charlie W. Lees, Andre Franke, Mauro D'Amato, Kent D. Taylor, James C. Lee, Philippe Goyette, Marcin Imielinski, Anna Latiano, Caroline Lagacé, Regan Scott, Leila Amininejad, Suzannah Bumpstead, Leonard Baidoo, Robert N. Baldassano, Murray Barclay, Theodore M. Bayless, Stephan Brand, Carsten Büning, Jean-Frédéric Colombel, Lee A. Denson, Martine De Vos, Marla Dubinsky, Cathryn Edwards, David Ellinghaus, Rudolf S. N. Fehrmann, James A. B. Floyd, Timothy Florin, Denis Franchimont, Lude Franke, Michel Georges, Jürgen Glas, Nicole L Glazer, Stephen L Guthery, Talin Haritunians, Nicholas K. Hayward, Jean-Pierre Hugot, Gilles Jobin, Debby Laukens, Ian Lawrance, Marc Lémann, Arie Levine, Cecile Libioulle, Edouard Louis, Dermot P. McGovern, Monica Milla, Grant W. Montgomery, Katherine I. Morley, Craig Mowat, Aylwin Ng, William Newman, Roel A. Ophoff, Laura Papi, Orazio Palmieri, Laurent Peyrin-Biroulet, Julián Panés, Anne Phillips, Natalie J. Prescott, Deborah D. Proctor, Rebecca Roberts, Richard Russell, Paul Rutgeerts, Jeremy Sanderson, Miquel Sans, Philip Schumm, Frank Seibold, Yashoda Sharma, Lisa A. Simms, Mark Seielstad, A. Hillary Steinhart, Stephan R. Targan, Leonard H. van den Berg, Morten Vatn, Hein Verspaget, Thomas Walters, Cisca Wijmenga, David C. Wilson, Harm-Jan Westra, Ramnik J. Xavier, Zhen Z. Zhao, Cyriel Y. Ponsioen, Vibeke Andersen, Leif Torkvist, Maria Gazouli, Nicholas P. Anagnou, Tom H. Karlsen, Limas Kupcinskas, Jurgita Sventoraityte, John C. Mansfield, Subra Kugathasan, Mark S. Silverberg, Jonas Halfvarson, Jerome I. Rotter, Christopher G. Mathew, Anne M. Griffiths, Richard Gearry, Tariq Ahmad, Steven R. Brant, Mathias Chamaillard, Jack Satsangi, Judy H. Cho, Stefan Schreiber, Mark J. Daly, Jeffrey C. Barrett, Miles Parkes, Vito Annese, Hakon Hakonarson, Graham Radford-Smith, Richard H. Duerr, Séverine Vermeire, Rinse K. Weersma, and John D. Rioux. Meta-analysis identifies 29 additional ulcerative colitis risk loci, increasing the number of confirmed associations to 47. *Nature Genetics*, 43(3):246–252, Feb 2011.

[5] Lucas Beyer and Paolo Bientinesi. GWAS on GPUs: Streaming Data from HDD for Sustained Performance. In *Euro-Par 2013 Parallel Processing*, pages 788–799, Aachen, Germany, Aug 2013.

[6] Clay Breshears. *The art of concurrency: A thread monkey's guide to writing parallel applications.* O'Reilly Media, Inc., 2009.

[7] William S. Bush and Jason H. Moore. Chapter 11: Genome-Wide Association Studies. *PLOS Computational Biology*, 8(12):1–11, Dec 2012.

[8] Marcin Cieślik and Cameron Mura. A lightweight, flow-based toolkit for parallel and distributed bioinformatics pipelines. *BMC Bioinformatics*, 12(1):61, Feb 2011.

[9] Marianna D'Addario, Dominik Kopczynski, Jorg Ingo Baumbach, and Sven Rahmann. A modular computational framework for automated peak extraction from ion mobility spectra. *BMC Bioinformatics*, 15(1):25, Jan 2014.

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

[11] Patrick Danoy, Karena Pryce, Johanna Hadler, Linda A. Bradbury, Claire Farrar, Jennifer Pointon, Australo-Anglo-American Spondyloarthritis Consortium (TASC), Michael Ward, Michael Weisman, John D. Reveille, B. Paul Wordsworth, Millicent A. Stone, Spondyloarthritis Research Consortium of Canada (SPARCC), Walter P. Maksymowych, Proton Rahman, Dafna Gladman, Robert D. Inman, and Matthew A. Brown. Association of Variants at 1q32 and STAT3 with Ankylosing Spondylitis Suggests Genetic Overlap with Crohn's Disease. *PLOS Genetics*, 6(12):1–5, Dec 2010.

[12] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, Jan 2008.

[13] E. Dede, B. Sendir, P. Kuzlu, J. Weachock, M. Govindaraju, and L. Ramakrishnan. Processing cassandra datasets with hadoop-streaming based approaches. *IEEE Transactions on Services Computing*, 9(1):46–58, Jan 2016.

[14] Michael Dondrup, Alexander Goesmann, Daniela Bartels, Jörn Kalinowski, Lutz Krause, Burkhard Linke, Oliver Rupp, Alexander Sczyrba, Alfred Pühler, and Folker Meyer. EMMA: a platform for consistent storage and efficient analysis of microarray data. *Journal of Biotechnology*, 106(2-3):135–146, Dec 2003.

[15] Diego Fabregat-Traver and Paolo Bientinesi. Computing petaflops over terabytes of data: The case of genome-wide association studies. *ACM Transactions on Mathematical Software (TOMS)*, 40(4):1–27, Jun 2014.

[16] D. L. Flam, J. V. B. Gomide, and A. D. A. Araújo. Development of an Open Source Software for Real Time Optical Motion Capture. In *2016 XVIII Symposium on Virtual and Augmented Reality (SVR)*, pages 117–121, Gramado, Brazil, June 2016.

[17] Andre Franke, Dermot P. B. McGovern, Jeffrey C. Barrett, Kai Wang, Graham L. Radford-Smith, Tariq Ahmad, Charlie W. Lees, Tobias Balschun, James Lee, Rebecca Roberts, Carl A. Anderson, Joshua C. Bis, Suzanne Bumpstead, David Ellinghaus, Eleonora M. Festen, Michel Georges, Todd Green, Talin Haritunians, Luke Jostins, Anna Latiano, Christopher G. Mathew, Grant W. Montgomery, Natalie J. Prescott, Soumya Raychaudhuri, Jerome I. Rotter, Philip Schumm, Yashoda Sharma, Lisa A. Simms, Kent D. Taylor, David Whiteman, Cisca Wijmenga, Robert N. Baldassano, Murray Barclay, Theodore M. Bayless, Stephan Brand, Carsten Büning, Albert Cohen, Jean-Frederick Colombel, Mario Cottone, Laura Stronati, Ted Denson, Martine De Vos, Renata D'Inca, Marla Dubinsky, Cathryn Edwards, Tim Florin, Denis Franchimont, Richard Gearry, Jürgen Glas, Andre Van Gossum, Stephen L. Guthery, Jonas Halfvarson, Hein W. Verspaget, Jean-Pierre Hugot, Amir Karban, Debby Laukens, Ian Lawrance, Marc Lemann, Arie Levine, Cecile Libioulle, Edouard Louis, Craig Mowat, William Newman, Julián Panés, Anne Phillips, Deborah D. Proctor, Miguel Regueiro, Richard Russell, Paul Rutgeerts, Jeremy Sanderson, Miquel Sans, Frank Seibold, A. Hillary Steinhart, Pieter C. F. Stokkers, Leif Torkvist, Gerd Kullak-Ublick, David Wilson, Thomas Walters, Stephan R. Targan, Steven R. Brant, John D. Rioux, Mauro D'Amato, Rinse K. Weersma, Subra Kugathasan, Anne M. Griffiths, John C. Mansfield, Severine Vermeire, Richard H. Duerr, Mark S. Silverberg, Jack Satsangi, Stefan Schreiber, Judy H. Cho, Vito Annese, Hakon Hakonarson, Mark J. Daly, and Miles Parkes. Genome-wide meta-analysis increases to 71 the number of confirmed Crohn's disease susceptibility loci. *Nature Genetics*, 42:1118–1125, Nov 2010.

[18] Timothy M. Frayling. Genome–wide association studies provide new insights into type 2 diabetes aetiology. *Nature Reviews Genetics*, 8(9):657–662, Sep 2007.

[19] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next

Generation MPI Implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104, Berlin, Heidelberg, September 2004.

[20] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. MPI-2: Extending the message-passing interface. In *Lecture Notes in Computer Science*, pages 128–135. Springer Berlin Heidelberg, Aug 1996.

[21] Richard A. Gibbs, John W. Belmont, Paul Hardenbol, Thomas D. Willis, Fuli Yu, Houcan Zhang, Changqing Zeng, Ichiro Matsuda, Yoshimitsu Fukushima, Darryl R. Macer, Eiko Suda, Lincoln D. Stein, Fiona Cunningham, Ardavan Kanani, Gudmundur A. Thorisson, Aravinda Chakravarti, Peter E. Chen, David J. Cutler, Carl S. Kashuk, Peter Donnelly, Jonathan Marchini, Gilean A. T. McVean, Simon R. Myers, Lon R. Cardon, Gonçalo R. Abecasis, Andrew Morris, Bruce S. Weir, James C. Mullikin, Stephen T. Sherry, Michael Feolo, David Altshuler, Mark J. Daly, Stephen F. Schaffner, Renzong Qiu, Alastair Kent, Georgia M. Dunston, Kazuto Kato, Norio Niikawa, Bartha M. Knoppers, Morris W. Foster, Ellen Wright Clayton, Vivian Ota Wang, Jessica Watkin, Richard A. Gibbs, John W. Belmont, Erica Sodergren, George M. Weinstock, Richard K. Wilson, Lucinda L. Fulton, Jane Rogers, Bruce W. Birren, Hua Han, Hongguang Wang, Martin Godbout, John C. Wallenburg, Paul LArcheveque, Guy Bellemare, Kazuo Todani, Takashi Fujita, Satoshi Tanaka, Arthur L. Holden, Eric H. Lai, Francis S. Collins, Lisa D. Brooks, Jean E. McEwen, Mark S. Guyer, Elke Jordan, Jane L. Peterson, Jack Spiegel, Lawrence M. Sung, Lynn F. Zacharia, Karen Kennedy, Michael G. Dunn, Richard Seabrook, Mark Shillito, Barbara Skene, John G. Stewart, David L. Valle, Ellen Wright Clayton, Lynn B. Jorde, John W. Belmont, Aravinda Chakravarti, Mildred K. Cho, Troy Duster, Morris W. Foster, Marla Jasperse, Bartha M. Knoppers, Pui-Yan Kwok, Julio Licinio, Jeffrey C. Long, Patricia A. Marshall, Pilar N. Ossorio, Vivian Ota Wang, Charles N. Rotimi, Charmaine D. M. Royal, Patricia Spallone, Sharon F. Terry, Eric S. Lander, Eric H. Lai, Deborah A. Nickerson, Gonçalo R. Abecasis, David Altshuler, David R. Bentley, Michael Boehnke, Lon R. Cardon, Mark J. Daly, Panos Deloukas, Julie A. Douglas, Stacey B. Gabriel, Richard R. Hudson, Thomas J. Hudson, Leonid Kruglyak, Pui-Yan Kwok, Yusuke Nakamura, Robert L. Nussbaum, Charmaine D. M. Royal, Stephen F. Schaffner, Stephen T. Sherry, Lincoln D. Stein, and Toshihiro Tanaka. The International HapMap Project. *Nature*, 426(6968):789–796, Dec 2003.

[22] Jeffrey C. Glaubitz, Terry M. Casstevens, Fei Lu, James Harriman, Robert J. Elshire, Qi Sun, and Edward S. Buckler. TASSEL-GBS: A High Capacity Genotyping by Sequencing Analysis Pipeline. *PLoS ONE*, 9(2):1–11, Feb 2014.

[23] Goecks, Jeremy and Nekrutenko, Anton and Taylor, James. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology*, 11(8):R86, Aug 2010.

[24] Jorge González-Domínguez, Bertil Schmidt, Jan Christian Kässens, and Lars Wienbrandt. Hybrid CPU/GPU acceleration of detection of 2-SNP epistatic interactions in GWAS. In *Euro-Par 2014 Parallel Processing*, pages 680–691, Porto, Portugal, Aug 2014. Springer International Publishing.

[25] Jorge Gonzalez-Dominguez, Lars Wienbrandt, Jan Christian Kassens, David Ellinghaus, Manfred Schimmler, and Bertil Schmidt. Parallelizing Epistasis Detection in GWAS on FPGA and GPU-Accelerated Computing Systems. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 12(5):982–994, Sep 2015.

[26] Benjamin Goudey, Mani Abedini, John L Hopper, Michael Inouye, Enes Makalic, Daniel F Schmidt, John Wagner, Zeyu Zhou, Justin Zobel, and Matthias Reumann. High performance computing enabling exhaustive analysis of higher order single nucleotide polymorphism interaction in Genome Wide Association Studies. *Health Information Science and Systems*, 3(S1):1–11, Dec 2015.

[27] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sep 1996.

[28] Luqman Hodgkinson, Javier Rosa, and Eric A. Brewer. Parallel software architecture for experimental workflows in computational biology on clouds. In *Parallel Processing and Applied Mathematics*, pages 281–291. Springer Berlin Heidelberg, 2012.

[29] Xuehui Huang, Yan Zhao, Xinghua Wei, Canyang Li, Ahong Wang, Qiang Zhao, Wenjun Li, Yunli Guo, Liuwei Deng, Chuanrang Zhu, Danlin Fan, Yiqi Lu, Qijun Weng, Kunyan Liu, Taoying Zhou, Yufeng Jing, Lizhen Si, Guojun Dong, Tao Huang, Tingting Lu, Qi Feng, Qian Qian, Jiayang Li, and Bin Han. Genome-wide association study of flowering time and grain yield traits in a worldwide collection of rice germplasm. *Nature Genetics*, 44(1):32–39, Dec 2011.

[30] Yung-Fen Huang, Jesse A. Poland, Charlene P. Wight, Eric W. Jackson, and Nicholas A. Tinker. Using genotyping-by-sequencing (GBS) for genomic discovery in cultivated oat. *PLoS ONE*, 9(7):e102448, Jul 2014.

[31] Katherine Icay, Ping Chen, Alejandra Cervera, Ville Rantanen, Rainer Lehtonen, and Sampsa Hautaniemi. SePIA: RNA and small RNA sequence processing, integration, and analysis. *BioData Mining*, 9(1):20, May 2016.

[32] L. E. Jordan and Gita Alaghband. *Fundamentals of Parallel Processing.* Prentice Hall Professional Technical Reference, 2002.

[33] H. M. Kang, N. A. Zaitlen, C. M. Wade, A. Kirby, D. Heckerman, M. J. Daly, and E. Eskin. Efficient Control of Population Structure in Model Organism Association Mapping. *Genetics*, 178(3):1709–1723, Feb 2008.

[34] Bernard Khor, Agnès Gardet, and Ramnik J. Xavier. Genetics and pathogenesis of inflammatory bowel disease. *Nature*, 474(7351):307–317, Jun 2011.

[35] R. J. Klein. Complement Factor H Polymorphism in Age-Related Macular Degeneration. *Science*, 308(5720):385–389, Apr 2005.

[36] Jaspal S. Kooner, Danish Saleheen, Xueling Sim, Joban Sehmi, Weihua Zhang, Philippe Frossard, Latonya F. Been, Kee-Seng Chia, Antigone S. Dimas, Neelam Hassanali, Tazeen Jafar, Jeremy B. M. Jowett, Xinzhong Li, Venkatesan Radha, Simon D. Rees, Fumihiko Takeuchi, Robin Young, Tin Aung, Abdul Basit, Manickam Chidambaram, Debashish Das, Elin Grundberg, Åsa K. Hedman, Zafar I. Hydrie, Muhammed Islam, Chiea-Chuen Khor, Sudhir Kowlessur, Malene M. Kristensen, Samuel Liju, Wei-Yen Lim, David R. Matthews, Jianjun Liu, Andrew P. Morris, Alexandra C. Nica, Janani M. Pinidiyapathirage, Inga Prokopenko, Asif Rasheed, Maria Samuel, Nabi Shah, A. Samad Shera, Kerrin S. Small, Chen Suo, Ananda R. Wickremasinghe, Tien Yin Wong, Mingyu Yang, Fan Zhang, DIAGRAM, MuTHER, Goncalo R. Abecasis, Anthony H. Barnett, Mark Caulfield, Panos Deloukas, Timothy M. Frayling, Philippe Froguel, Norihiro Kato, Prasad Katulanda, M. Ann Kelly, Junbin Liang, Viswanathan Mohan, Dharambir K. Sanghera, James Scott, Mark Seielstad, Paul Z. Zimmet, Paul Elliott, Yik Ying Teo, Mark I. McCarthy, John Danesh, E. Shyong Tai, and John C. Chambers. Genome-wide association study in individuals of South Asian ancestry identifies six new type 2 diabetes susceptibility loci. *Nature Genetics*, 43(10):984–989, Aug 2011.

[37] Arthur Korte, Bjarni J Vilhjálmsson, Vincent Segura, Alexander Platt, Quan Long, and Magnus Nordborg. A mixed-model approach for genome-wide association studies of correlated traits in structured populations. *Nature Genetics*, 44(9):1066–1071, August 2012.

[38] Leonid Kruglyak and Deborah A Nickerson. Variation is the spice of life. *Nature Genetics*, 27(3):234–236, mar 2001.

[39] Michel Krämer and Ivo Senner. A modular software architecture for processing of big geospatial data in the cloud. *Computers & Graphics*, 49:69–81, Jun 2015.

[40] C. C. Law, A. Henderson, and J. Ahrens. An application architecture for large data visualization: a case study. In *Proceedings IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics*, pages 125–159, San Diego, CA, October 2001.

[41] Simone Leo, Federico Santoni, and Gianluigi Zanetti. Biodoop: Bioinformatics on hadoop. In *2009 International Conference on Parallel Processing Workshops*, pages 415–422, Sep 2009.

[42] F. Li, B. Chen, K. Xu, J. Wu, W. Song, I. Bancroft, A. L. Harper, M. Trick, S. Liu, G. Gao, N. Wang, G. Yan, J. Qiao, J. Li, H. Li, X. Xiao, T. Zhang, and X. Wu. Genome-Wide Association Study Dissects the Genetic Architecture of Seed Weight and Seed Quality in Rapeseed (Brassica napus L.). *DNA Research*, 21(4):355–367, Feb 2014.

[43] Feng Li, Biyun Chen, Kun Xu, Guizhen Gao, Guixin Yan, Jiangwei Qiao, Jun Li, Hao Li, Lixia Li, Xin Xiao, Tianyao Zhang, Takeshi Nishio, and Xiaoming Wu. A genome-wide association study of plant height and primary branch number in rapeseed (Brassica napus). *Plant Science*, 242:169–177, Jan 2016.

[44] Hui Li, Zhiyu Peng, Xiaohong Yang, Weidong Wang, Junjie Fu, Jianhua Wang, Yingjia Han, Yuchao Chai, Tingting Guo, Ning Yang, Jie Liu, Marilyn L Warburton, Yanbing Cheng, Xiaomin Hao, Pan Zhang, Jinyang Zhao, Yunjun Liu, Guoying Wang, Jiansheng Li, and Jianbing Yan. Genome-wide association study dissects the genetic architecture of oil biosynthesis in maize kernels. *Nature Genetics*, 45(1):43–50, Dec 2012.

[45] Christoph Lippert, Jennifer Listgarten, Ying Liu, Carl M Kadie, Robert I Davidson, and David Heckerman. FaST linear mixed models for genome-wide association studies. *Nature Methods*, 8(10):833–835, Sep 2011.

[46] Jennifer Listgarten, Christoph Lippert, Carl M Kadie, Robert I Davidson, Eleazar Eskin, and David Heckerman. Improved linear mixed models for genome-wide association studies. *Nature Methods*, 9(6):525–526, Jun 2012.

[47] Changlin Liu, Jianfeng Weng, Degui Zhang, Xiaocong Zhang, Xiaoyan Yang, Liyu Shi, Qingchang Meng, Jianhua Yuan, Xinping Guo, Zhuanfang Hao, Chuanxiao Xie, Mingshun Li, Xiaoke Ci, Li Bai, Xinhai Li, and Shihuang Zhang. Genome-wide association study of resistance to rough dwarf disease in maize. *European Journal of Plant Pathology*, 139(1):205–216, Jan 2014.

[48] G. R. Luecke, N. T. Weeks, B. M. Groth, M. Kraeva, L. Ma, L. M. Kramer, J. E. Koltes, and J. M. Reecy. Fast Epistasis Detection in Large-Scale GWAS for Intel Xeon Phi Clusters. In *2015 IEEE Trustcom/BigDataSE/ISPA*, pages 228–235, Helsinki, Finland, Aug 2015.

[49] J. Möbius, M. Kremer, and L. Kobbelt. OpenFlipper - A highly modular framework for processing and visualization of complex geometric models. In *2013 6th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pages 25–32, Orlando, FL, March 2013.

[50] Aidan R. O'Brien, Neil F. W. Saunders, Yi Guo, Fabian A. Buske, Rodney J. Scott, and Denis C. Bauer. VariantSpark: Population Scale Clustering of Genotype Information. *BMC Genomics*, 16(1):1052–1060, Dec 2015.

[51] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[52] Patricia G Parker, Allison A Snow, Malcolm D Schug, Gregory C Booton, and Paul A Fuerst. What molecules can tell us about populations: choosing andusing a molecular marker. *Ecology*, 79(2):361–382, Mar 1998.

[53] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec 1972.

[54] Helen Pearson. What is a gene? *Nature*, 441(7092):398–401, May 2006.

[55] William R. Pearson. [5] rapid and sensitive sequence comparison with FASTP and FASTA. In *Methods in Enzymology*, pages 63–98. Elsevier, 1990.

[56] Elmar Peise, Diego Fabregat-Traver, and Paolo Bientinesi. High performance solutions for big-data GWAS. *Parallel Computing*, 42:75–87, Feb 2015.

[57] Ivanilton Polato, Reginaldo Ré, Alfredo Goldman, and Fabio Kon. A Comprehensive View of Hadoop research-A Systematic Literature Review. *Journal of Network and Computer Applications*, 46(C):1–25, Nov 2014.

[58] Shaun Purcell, Benjamin Neale, Kathe Todd-Brown, Lori Thomas, Manuel A.R. Ferreira, David Bender, Julian Maller, Pamela Sklar, Paul I.W. de Bakker, Mark J. Daly, and Pak C. Sham. PLINK: A Tool Set for Whole-Genome Association and Population-Based Linkage Analyses. *The American Journal of Human Genetics*, 81(3):559–575, Sep 2007.

[59] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *14th Python in Science Conference*, pages 130–136, Austin, TX, Jul 2015.

[60] Manuel Saldana and Paul Chow. TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs. In *2006 International Conference on Field Programmable Logic and Applications*, pages 1–6, Madrid, Spain, Aug 2006.

[61] Shadrokh Samavi, Shahram Shirani, Nader Karimi, and M. Jamal Deen. A pipeline architecture for processing of DNA microarrays images. *The Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, 38(3):287–297, Nov 2004.

[62] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, Incline Village, NV, May 2010.

[63] Karolina Sikorska, Emmanuel Lesaffre, Patrick FJ Groenen, and Paul HC Eilers. GWAS on your notebook: fast semi-parallel linear and logistic regression for genome-wide association studies. *BMC Bioinformatics*, 14(1):166–176, May 2013.

[64] Javier Simón-Sánchez, Claudia Schulte, Jose M Bras, Manu Sharma, J Raphael Gibbs, Daniela Berg, Coro Paisan-Ruiz, Peter Lichtner, Sonja W Scholz, Dena G Hernandez, Rejko Krüger, Monica Federoff, Christine Klein, Alison Goate, Joel Perlmutter, Michael Bonin, Michael A Nalls, Thomas Illig, Christian Gieger, Henry Houlden, Michael Steffens, Michael S Okun, Brad A Racette, Mark R Cookson, Kelly D Foote, Hubert H Fernandez, Bryan J Traynor, Stefan Schreiber, Sampath Arepalli, Ryan Zonozi, Katrina Gwinn, Marcel van der Brug, Grisel Lopez, Stephen J Chanock, Arthur Schatzkin, Yikyung Park, Albert Hollenbeck, Jianjun Gao, Xuemei Huang, Nick W Wood, Delia Lorenz, Günther Deuschl, Honglei Chen, Olaf Riess, John A Hardy, Andrew B Singleton, and Thomas Gasser. Genome-wide association study reveals genetic risk underlying Parkinson's disease. *Nature Genetics*, 41:1308–1312, Nov 2009.

[65] Nicholas A Sinnott-Armstrong, Casey S Greene, Fabio Cancare, and Jason H Moore. Accelerating epistasis analysis in human genetics with consumer graphics hardware. *BMC Research Notes*, 2(1):149–154, July 2009.

[66] Jennifer Spindel, Mark Wright, Charles Chen, Joshua Cobb, Joseph Gage, Sandra Harrington, Mathias Lorieux, Nourollah Ahmadi, and Susan McCouch. Bridging the genotyping gap: using genotyping by sequencing (GBS) to add high-density SNP markers and new value to traditional bi-parental mapping and breeding populations. *Theoretical and Applied Genetics*, 126(11):2699–2716, Aug 2013.

[67] Oscar Torreno and Oswaldo Trelles. Breaking the computational barriers of pairwise genome comparison. *BMC Bioinformatics*, 16(1):250, Aug 2015.

[68] Patrick Turley, Raymond K. Walters, Omeed Maghzian, Aysu Okbay, James J. Lee, Mark Alan Fontana, Tuan Anh Nguyen-Viet, Robbee Wedow, Meghan Zacher, Nicholas A. Furlotte, Patrik Magnusson, Sven Oskarsson, Magnus Johannesson, Peter M. Visscher, David Laibson, David Cesarini, Benjamin M. Neale, and Daniel J. Benjamin and. Multi-trait analysis of genome-wide association summary statistics using MTAG. *Nature Genetics*, 50(2):229–237, Jan 2018.

[69] Subodh Verma, Shefali Gupta, Nitesh Bandhiwal, Tapan Kumar, Chellapilla Bharadwaj, and Sabhyata Bhatia. High-density linkage map construction and mapping of seed trait QTLs in chickpea (cicer arietinum l.) using genotyping-by-sequencing (GBS). *Scientific Reports*, 5(1), Dec 2015.

[70] Chris Wallace, Stephen J. Newhouse, Peter Braund, Feng Zhang, Martin Tobin, Mario Falchi, Kourosh Ahmadi, Richard J. Dobson, Ana Carolina B. Marçano, Cother Hajat, Paul Burton, Panagiotis Deloukas, Morris Brown, John M. Connell, Anna Dominiczak, G. Mark Lathrop, John Webster, Martin Farrall, Tim Spector, Nilesh J. Samani, Mark J. Caulfield, and Patricia B. Munroe. Genome-wide Association Study Identifies Genes for Biomarkers of Cardiovascular Disease: Serum Urate and Dyslipidemia. *The American Journal of Human Genetics*, 82(1):139–149, Jan 2008.

[71] J. Wang, H. Mei, W. Chen, Y. Jiang, W. Sun, F. Li, Q. Fu, and F. Jiang. Study of eight GWAS-identified common variants for association with obesity-related indices in Chinese children at puberty. *International Journal of Obesity*, 36(4):542–547, Nov 2011.

[72] Ming Wang, Jianbing Yan, Jiuran Zhao, Wei Song, Xiaobo Zhang, Yannong Xiao, and Yonglian Zheng. Genome-wide association study (GWAS) of resistance to head smut in maize. *Plant Science*, 196:125–131, Nov 2012.

[73] Shi-Bo Wang, Jian-Ying Feng, Wen-Long Ren, Bo Huang, Ling Zhou, Yang-Jun Wen, Jin Zhang, Jim M. Dunwell, Shizhong Xu, and Yuan-Ming Zhang. Improving power and accuracy of genome-wide association studies via a multi-locus mixed linear model methodology. *Scientific Reports*, 6(1):1–10, Jan 2016.

[74] Christian Widmer, Christoph Lippert, Omer Weissbrod, Nicolo Fusi, Carl Kadie, Robert Davidson, Jennifer Listgarten, and David Heckerman. Further Improvements to Linear Mixed Models for Genome-Wide Association Studies. *Scientific Reports*, 4:6874 –6886, Nov 2014.

[75] Naomi Wray and P Visscher. Estimating trait heritability. *Nature Education*, 1(1):1–29, 2008.

[76] Jianming Yu, Gael Pressoir, William H Briggs, Irie Vroh Bi, Masanori Yamasaki, John F Doebley, Michael D McMullen, Brandon S Gaut, Dahlia M Nielsen, James B Holland, Stephen Kresovich, and Edward S Buckler. A unified mixed-model method for association mapping that accounts for multiple levels of relatedness. *Nature Genetics*, 38(2):203–208, Dec 2005.

[77] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenkerx, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *2nd USENIX Conference on Hot Topics in Cloud Computing*, pages 10–16, Boston, MA, Jun 2010.

[78] X Zhang, P Pérez-Rodríguez, K Semagn, Y Beyene, R Babu, M A López-Cruz, F San Vicente, M Olsen, E Buckler, J-L Jannink, B M Prasanna, and J Crossa. Genomic prediction in biparental tropical maize populations in water-stressed and well-watered environments using low-density and GBS SNPs. *Heredity*, 114(3):291–299, Nov 2014.

[79] Zhiwu Zhang, Elhan Ersoz, Chao-Qiang Lai, Rory J Todhunter, Hemant K Tiwari, Michael A Gore, Peter J Bradbury, Jianming Yu, Donna K Arnett, Jose M Ordovas, and Edward S Buckler. Mixed linear model approach adapted for genome-wide association studies. *Nature Genetics*, 42(4):355–360, Mar 2010.

[80] Di Zhao and Shenghua Ni. Parallel Multi-Proposal and Multi-Chain MCMC for Calculating P-Value of Genome-Wide Association Study. *Parallel Processing Letters*, 23(03):1–17, Sep 2013.

[81] G. Zhou, Y. Chen, W. Yao, C. Zhang, W. Xie, J. Hua, Y. Xing, J. Xiao, and Q. Zhang. Genetic composition of yield heterosis in an elite rice hybrid. *National Academy of Sciences*, 109(39):15847–15852, Sep 2012.

Appendix A

Code block of first bottleneck extracted from Weighted MLM Plugin of TASSEL

Listing A.1: Code block extracted from computeZKZ method

```
while (g > 3) {
    cm.setNumberOfGroups(g);
    DoubleMatrix compressedZ = cm.getCompressedZ(Z);
    DoubleMatrix compressedK = cm.getCompressedMatrix(kinmethod);
    try {
        emlm = new EMMAforDoubleMatrix(data, X, compressedK, compressedZ, 0, Double.NaN);
        emlm.solve();

        //output number of groups, compression level
        //(= number of taxa / number of groups), -2L, genvar, resvar
        compressionReportBuilder.add(new Object[]{traitname, g,
                   ((double) nkin) / ((double) g),
                   -2 * emlm.getLnLikelihood(),
                   emlm.getVarRan(),
                   emlm.getVarRes()});

        if (Double.isNaN(bestlnlk) || emlm.getLnLikelihood() > bestlnlk) {
            bestlnlk = emlm.getLnLikelihood();
            bestCompression = g;
            resvar = emlm.getVarRes();
            genvar = emlm.getVarRan();
        }
    } catch (Exception e) {
        System.out.println("Compression␣failed␣for␣g␣=␣" + g);
    }

    int prev = g;
    while (g == prev) {
        exponent++;
        int prog = (int) (exponent * 100 / maxexponent);
        prog = Math.min(prog, 99);
        parentPlugin.updateProgress(prog);
        g = (int) (nkin * Math.pow(base, exponent));
    }
}
```

Appendix B

Code block of second bottleneck extracted from Weighted MLM Plugin of TASSEL

Listing B.1: Code block extracted from solve method

```
for (int m = 0; m < numberOfMarkers; m++) {
    OpenBitSet missingObsForSite = new OpenBitSet(missing);
    missingObsForSite.or(missingForSite(m));

    //only data for which missing=false are in the Z matrix
    //the block below finds the rows of Z that have no marker data.
    //Those rows/columns will need to be removed from ZKZ or from V,
    //depending on the analysis method.
    OpenBitSet missingFromZ = new OpenBitSet(nonMissingObs);

    int nonMissingCount = 0;
    for (int i = 0; i < totalObs; i++) {
        if (!missing.fastGet(i)) {
            if (missingObsForSite.fastGet(i)) {
                missingFromZ.fastSet(nonMissingCount);
            }
            nonMissingCount++;
        }
    }

    //adjust y for missing data
    DoubleMatrix ymarker = AssociationUtils.getNonMissingValues(y, missingFromZ);

    //adjust the fixed effects
    DoubleMatrix fixed2 = AssociationUtils.getNonMissingValues(fixed, missingFromZ);

    //add marker data to fixed effects
    ArrayList<Byte> markerIds = new ArrayList<>();
    int nAlleles = 0;
    int markerdf = 0;
    DoubleMatrix X;
    int[] alleleCounts = null;

    if (useGenotypeCalls) {
        byte[] genotypes = AssociationUtils.getNonMissingBytes(myGenoPheno.genotypeAllTaxa
            ↪ (m), missingObsForSite);
        FactorModelEffect markerEffect = new FactorModelEffect(ModelEffectUtils.
            ↪ getIntegerLevels(genotypes, markerIds), true);
        X = fixed2.concatenate(markerEffect.getX(), false);
        nAlleles = markerEffect.getNumberOfLevels();
        alleleCounts = markerEffect.getLevelCounts();
        markerdf = nAlleles - 1;
    } else if (useReferenceProbability) {
```

```
42          double[] genotypes = AssociationUtils.getNonMissingDoubles(myGenoPheno.
                ↪ referenceProb(m), missingObsForSite);
43          int nrows = genotypes.length;
44          X = fixed2.concatenate(DoubleMatrixFactory.DEFAULT.make(nrows, 1, genotypes),
                ↪ false);
45          nAlleles = 1;
46          alleleCounts = new int[]{nrows};
47          markerdf = 1;
48      } else {
49          X = null;
50      }
51
52      CompressedMLMResult result = new CompressedMLMResult();
53      //need to add marker information to result once Alignment is stable
54
55      if (useP3D) {
56          testMarkerUsingP3D(result, ymarker, X, Vminus.getInverse(missingFromZ,
                ↪ nonMissingObs), markerdf, markerIds);
57      } else {
58          DoubleMatrix Zsel = AssociationUtils.getNonMissingValues(zk[0], missingFromZ);
59          testMarkerUsingEMMA(result, ymarker, X, zk[1], Zsel, nAlleles, markerIds);
60          markerdf = result.modeldf - baseModeldf;
61      }
62
63      //if the results are to be filtered on pmax check for that condition
64      boolean recordTheseResults = true;
65      if (parentPlugin.isFilterOutput() && result.p > parentPlugin.getMaxp()) {
66          recordTheseResults = false;
67      }
68
69      if (recordTheseResults) {
70          //add result to main
71          //{"Trait","Marker","Chr","Pos","Locus","Site","df","F","p","errordf","MarkerR2","
                ↪ Genetic Var","Residual Var", "-2LnLikelihood"};
72          //results with additive and dominance effects
73          //{"Trait","Marker","Chr","Pos","Locus","Site","df","F","p","add_effect","add_F","
                ↪ add_p","dom_effect","dom_F","dom_p","errordf","MarkerR2","Genetic Var","
                ↪ Residual Var", "-2LnLikelihood"}
74
75          String markername = myGenotype.siteName(m);
76          String chr = "";
77          String pos = "";
78          String locus = myGenotype.chromosomeName(m);
79          String site = Integer.toString(myGenotype.chromosomalPosition(m));
80          double errordf = (double) (ymarker.numberOfRows() - result.modeldf);
81
82          tableRow = new Object[]{attr.name(),
83                  markername,
84                  locus,
85                  site,
86                  new Integer(markerdf),
87                  new Double(result.F),
88                  new Double(result.p),
89                  new Double(result.addEffect),
```

```
90              new Double(result.Fadd),
91              new Double(result.padd),
92              new Double(result.domEffect),
93              new Double(result.Fdom),
94              new Double(result.pdom),
95              new Double(errordf),
96              new Double(result.r2),
97              new Double(genvar),
98              new Double(resvar),
99              new Double(-2 * lnlk)};
100         siteReportBuilder.add(tableRow);
101
102         //add result to alleles
103         //"Trait","Marker","Chr","Pos","Allele","Effect", obs
104         int numberOfRowsKept = totalObs - (int) missingObsForSite.cardinality();
105         if (useReferenceProbability) {
106             tableRow = new Object[]{attr.name(),
107                     markername,
108                     locus,
109                     site,
110                     "",
111                     result.beta.get(result.beta.numberOfRows() - 1, 0),
112                     numberOfRowsKept
113             };
114
115             //record the results
116             alleleReportBuilder.add(tableRow);
117         } else if (nAlleles > 1) {
118             for (int a = 0; a < nAlleles; a++) {
119                 Double estimate;
120                 if (a < nAlleles - 1) {
121                     estimate = result.beta.get(result.beta.numberOfRows() - nAlleles + 1 +
                            ↪ a, 0);
122                 } else {
123                     estimate = 0.0;
124                 }
125                 tableRow = new Object[]{attr.name(),
126                         markername,
127                         locus,
128                         site,
129                         NucleotideAlignmentConstants.getNucleotideIUPAC(markerIds.get(a)),
130                         estimate,
131                         alleleCounts[a]
132                 };
133
134                 //record the results
135                 alleleReportBuilder.add(tableRow);
136             }
137         }
138
139     }
140     iterationsSofar++;
141     int progress = (int) ((double) iterationsSofar / (double) expectedIterations * 100);
142     progress = Math.min(99, progress);
```

```
143     parentPlugin.updateProgress(progress);
144 }
```

Appendix C

Implementation of Parallel Processing Module for TASSEL using Apache Spark

Listing C.1: Code block of implemented Parallel Processing Module

```
1  public class Spark<T,S> implements java.io.Serializable, ParallelModuleInterface<T,S> {
2
3      public SparkConf tconf;
4
5      public Spark(){
6          tconf = new SparkConf().setAppName("Tassel␣Application");
7      }
8
9      public List<S> process(List<T> data, Function2<Integer, Iterator<T>, Iterator<S>> f) {
10         SparkSession tspark = SparkSession.builder().config(tconf).getOrCreate();
11         JavaSparkContext tsc = JavaSparkContext.fromSparkContext(tspark.sparkContext());
12
13         int repartition_count = this.getRepartitionCount(tconf); //Calculate repartition
               ↪ count based on spark configuration
14
15         JavaRDD<T> distData = tsc.parallelize(data, repartition_count);
16         JavaRDD<S> mappedData = distData.mapPartitionsWithIndex(f, true)
17                 .persist(StorageLevel.MEMORY_ONLY_SER());
18
19         return mappedData.collect();
20     }
21 }
```

Appendix D

Refactored code block of the first bottleneck of TASSEL to a function

Listing D.1: Refactored code block

```
public Function2<Integer, Iterator<Integer>, Iterator<EmmaResult>> processGCalculation(
       CompressedDoubleMatrix cm,kinshipMethod kinmethod,String dataKey,
       String xKey, String zKey
   ){
   return new Function2<Integer, Iterator<Integer>, Iterator<EmmaResult>>(){

       @Override
       public Iterator<EmmaResult> call(Integer index, Iterator<Integer> rows){
           //resolving data from cache
           DoubleMatrix data = (DoubleMatrix) cache.get(dataKey);
           DoubleMatrix X = (DoubleMatrix) cache.get(xKey);
           DoubleMatrix Z = (DoubleMatrix) cache.get(zKey);

           List<EmmaResult> map = new ArrayList<EmmaResult>();
           while(rows.hasNext()){
               int g = (int) rows.next();
               cm.setNumberOfGroups(g);
               DoubleMatrix compressedZ = cm.getCompressedZ(Z);
               DoubleMatrix compressedK = cm.getCompressedMatrix(kinmethod);
               EMMAforDoubleMatrix emlm = new EMMAforDoubleMatrix(data, X, compressedK,
                   ↪ compressedZ, 0, Double.NaN);
               try{
                   emlm.solve();
               } catch (Exception e) {
                   System.out.println("Compression␣failed␣for␣g␣=␣" + g);
               } finally{
                   EmmaResult emmaResult = new EmmaResult();
                   emmaResult.g = g;
                   emmaResult.lnLikelihood = emlm.getLnLikelihood();
                   emmaResult.varRan = emlm.getLnLikelihood();
                   emmaResult.varRes = emlm.getVarRes();
                   map.add(emmaResult);
               }
           }
           return map.iterator();
       }
   };
}
```

Appendix E

Refactored code block of the second bottleneck of TASSEL to a function

Listing E.1: Refactored code block

```
1  public Function2<Integer, Iterator<Integer>, Iterator<ArrayList<ArrayList<Object[]>>>>
       ↪ processMarkers(
2      CompressedMLMusingDoubleMatrix c,OpenBitSet missing,int nonMissingObs, int totalObs,
           ↪ String yKey, String fixedKey,
3      boolean useGenotypeCalls, boolean useReferenceProbability, GenotypePhenotype
           ↪ myGenoPheno, boolean useP3D,
4      SymmetricMatrixInverterDM Vminus, String zkKey, int baseModeldf, GenotypeTable
           ↪ myGenotype,
5      PhenotypeAttribute attr, double genvar, double resvar, double lnlk) {
6      return new Function2<Integer, Iterator<Integer>, Iterator<ArrayList<ArrayList<Object
           ↪ []>>>>() {
7          @Override
8          public Iterator<ArrayList<ArrayList<Object[]>>> call(Integer index, Iterator<
               ↪ Integer> rows) {
9              //retrieving cache values
10             DoubleMatrix y = (DoubleMatrix) cache.get(yKey);
11             DoubleMatrix fixed = (DoubleMatrix) cache.get(fixedKey);
12             DoubleMatrix[] zk = (DoubleMatrix[]) cache.get(zkKey);
13
14             //Array List which saves total site report and allele rows
15             ArrayList<ArrayList<ArrayList<Object[]>>> totalRow = new ArrayList<ArrayList<
                   ↪ ArrayList<Object[]>>>();
16             while (rows.hasNext()) {
17                 //Array list which saves each site report and allele instances
18                 ArrayList<ArrayList<Object[]>> currentRow = new ArrayList<ArrayList<Object
                       ↪ []>>();
19                 int m = rows.next();
20                 OpenBitSet missingObsForSite = new OpenBitSet(missing);
21                 missingObsForSite.or(c.missingForSite(m));
22
23                 //only data for which missing=false are in the Z matrix
24                 //the block below finds the rows of Z that have no marker data.
25                 //Those rows/columns will need to be removed from ZKZ or from V, depending
                       ↪ on the analysis method.
26                 OpenBitSet missingFromZ = new OpenBitSet(nonMissingObs);
27
28                 int nonMissingCount = 0;
29                 for (int i = 0; i < totalObs; i++) {
30                     if (!missing.fastGet(i)) {
31                         if (missingObsForSite.fastGet(i)) {
32                             missingFromZ.fastSet(nonMissingCount);
33                         }
34                         nonMissingCount++;
```

```
35                  }
36              }
37
38              ////////
39              //adjust y for missing data
40              DoubleMatrix ymarker = AssociationUtils.getNonMissingValues(y, missingFromZ
                    ↪ );
41
42              //adjust the fixed effects
43              DoubleMatrix fixed2 = AssociationUtils.getNonMissingValues(fixed,
                    ↪ missingFromZ);
44
45              //add marker data to fixed effects
46              ArrayList<Byte> markerIds = new ArrayList<>();
47              int nAlleles = 0;
48              int markerdf = 0;
49              DoubleMatrix X;
50              int[] alleleCounts = null;
51
52              if (useGenotypeCalls) {
53                  //String[] genotypes = AssociationUtils.getNonMissingValues(myGenoPheno
                        ↪ .getStringGenotype(m), missingObsForSite);
54                  byte[] genotypes = AssociationUtils.getNonMissingBytes(myGenoPheno.
                        ↪ genotypeAllTaxa(m),
55                          missingObsForSite);
56                  FactorModelEffect markerEffect = new FactorModelEffect(
57                          ModelEffectUtils.getIntegerLevels(genotypes, markerIds), true);
58                  X = fixed2.concatenate(markerEffect.getX(), false);
59                  nAlleles = markerEffect.getNumberOfLevels();
60                  alleleCounts = markerEffect.getLevelCounts();
61                  markerdf = nAlleles - 1;
62              } else if (useReferenceProbability) {
63                  double[] genotypes = AssociationUtils.getNonMissingDoubles(myGenoPheno.
                        ↪ referenceProb(m),
64                          missingObsForSite);
65                  int nrows = genotypes.length;
66                  X = fixed2.concatenate(DoubleMatrixFactory.DEFAULT.make(nrows, 1,
                        ↪ genotypes), false);
67                  nAlleles = 1;
68                  alleleCounts = new int[] { nrows };
69                  markerdf = 1;
70              } else {
71                  X = null;
72              }
73
74              CompressedMLMusingDoubleMatrix.CompressedMLMResult result = c.new
                    ↪ CompressedMLMResult();
75              //need to add marker information to result once Alignment is stable
76
77              if (useP3D) {
78                  c.testMarkerUsingP3D(result, ymarker, X, Vminus.getInverse(missingFromZ
                        ↪ , nonMissingObs),
79                          markerdf, markerIds);
80              } else {
```

```
81                  DoubleMatrix Zsel = AssociationUtils.getNonMissingValues(zk[0],
                        ↪ missingFromZ);
82                  c.testMarkerUsingEMMA(result, ymarker, X, zk[1], Zsel, nAlleles,
                        ↪ markerIds);
83                  markerdf = result.modeldf - baseModeldf;
84              }
85
86              // //if the results are to be filtered on pmax check for that condition
87              // boolean recordTheseResults = true;
88              // if (c.parentplugin.isFilterOutput() && result.p > parentPlugin.getMaxp()
                    ↪ ) {
89              // recordTheseResults = false;
90              // }
91              boolean recordTheseResults = true;
92
93              if (recordTheseResults) {
94                  //add result to main
95                  //{"Trait","Marker","Chr","Pos","Locus","Site","df","F","p","errordf","
                        ↪ MarkerR2","Genetic Var","Residual Var", "-2LnLikelihood"};
96                  //results with additive and dominance effects
97                  //{"Trait","Marker","Chr","Pos","Locus","Site","df","F","p","add_effect
                        ↪ ","add_F","add_p","dom_effect","dom_F","dom_p","errordf","
                        ↪ MarkerR2","Genetic Var","Residual Var", "-2LnLikelihood"}
98
99                  String markername = myGenotype.siteName(m);
100                 String chr = "";
101                 String pos = "";
102                 String locus = myGenotype.chromosomeName(m);
103                 String site = Integer.toString(myGenotype.chromosomalPosition(m));
104                 double errordf = (double) (ymarker.numberOfRows() - result.modeldf);
105
106                 ArrayList<Object[]> siteReportRows = new ArrayList<Object[]>();
107                 siteReportRows.add(new Object[] { attr.name(), markername, locus, site,
                        ↪  new Integer(markerdf),
108                     new Double(result.F), new Double(result.p), new Double(result.
                            ↪ addEffect),
109                     new Double(result.Fadd), new Double(result.padd), new Double(
                            ↪ result.domEffect),
110                     new Double(result.Fdom), new Double(result.pdom), new Double(
                            ↪ errordf),
111                     new Double(result.r2), new Double(genvar), new Double(resvar),
                            ↪ new Double(-2 * lnlk) });
112
113                 //add result to alleles
114                 //"Trait","Marker","Chr","Pos","Allele","Effect", obs
115                 ArrayList<Object[]> alleleRows = new ArrayList<Object[]>();
116                 int numberOfRowsKept = totalObs - (int) missingObsForSite.cardinality()
                        ↪ ;
117                 if (useReferenceProbability) {
118                     alleleRows.add(new Object[] { attr.name(), markername, locus, site,
                            ↪ "",
119                         result.beta.get(result.beta.numberOfRows() - 1, 0),
                                ↪ numberOfRowsKept });
120                 } else if (nAlleles > 1) {
```

```
                    for (int a = 0; a < nAlleles; a++) {
                        Double estimate;
                        if (a < nAlleles - 1) {
                            estimate = result.beta.get(result.beta.numberOfRows() -
                                ↪ nAlleles + 1 + a, 0);
                        } else {
                            estimate = 0.0;
                        }
                        alleleRows.add(new Object[] { attr.name(), markername, locus,
                            ↪ site,
                                //markerIds.get(a),
                                NucleotideAlignmentConstants.getNucleotideIUPAC(markerIds
                                    ↪ .get(a)), estimate,
                                alleleCounts[a] });
                    }
                }
                //Add site report and allele rows to the current row container
                currentRow.add(siteReportRows);
                currentRow.add(alleleRows);
                totalRow.add(currentRow);
            }
            ////////
        }
        return totalRow.iterator();
    }
  };
}
```

Appendix F

Implementation of Shared Data Cache module for TASSEL using Redis

Listing F.1: Implementation of Shared Data Cache module

```
1  public class RedisCache implements CacheInterface, Serializable {
2
3      private String host;
4      private Integer port;
5
6      public RedisCache(String host, Integer port){
7          this.host = host;
8          this.port = port;
9      }
10
11     @Override
12     public String put(Serializable o) {
13
14         String key = Integer.toString(System.identityHashCode(o));
15         try(ByteArrayOutputStream b = new ByteArrayOutputStream();
16             ObjectOutput output = new ObjectOutputStream(b)){
17                 output.writeObject(o);
18                 output.flush();
19                 try(JedisCluster jedis = new JedisCluster(new HostAndPort(this.host, this.
                       ↪ port))){
20                     jedis.set(key.getBytes(), b.toByteArray());
21                     return key;
22                 }
23         }
24         catch(Exception e){
25             e.printStackTrace();
26             return null;
27         }
28
29     }
30
31     @Override
32     public Object get(String key) {
33         try (JedisCluster jedis = new JedisCluster(new HostAndPort(this.host, this.port)))
               ↪  {
34             try (ByteArrayInputStream bin = new ByteArrayInputStream(jedis.get(key.getBytes
                   ↪ ()));
35                     ObjectInput input = new ObjectInputStream(bin)) {
36                 return input.readObject();
37             }
38         }
39         catch(Exception e){
40             e.printStackTrace();
```

```
41              return null;
42          }
43      }
44
45  }
```

Appendix G

Implementation of Parallel Processing Module for FaST-LMM using Apache Spark

Listing G.1: Implementation of Parallel Processing module

```
try:
    from pyspark import SparkContext, SparkConf, StorageLevel
except:
    pass

class Spark:
    try:
        conf = SparkConf().setAppName("FaST-LMM")
    except:
        pass

    def __init__(self):
        pass

    def process(self, data, process_function):
        sc = SparkContext.getOrCreate(conf=Spark.conf)
        pdata = sc.parallelize(data,len(data))
        del data
        collected_data = pdata.map(process_function).collect()
        del pdata
        return collected_data
```

Appendix H

Implementation of Parallel Processing Module for FaST-LMM using Dask

Listing H.1: Implementation of Parallel Processing module

```
from dask.distributed import Client
from distributed.diagnostics import progress
import dask.bag as db

class DaskCluster:

    def __init__(self,host,port):
        self.host = host
        self.port = port

    def process(self, data, process_function):
        client = Client("{}:{}".format(self.host,self.port))
        scattered_data = client.scatter(data)
        map_operation = client.map(process_function,scattered_data)
        progress(map_operation)
        output = client.gather(map_operation)
        client.close()
        return output
```

Appendix I

Implementation of Shared Data Cache module for FaST-LMM using Redis

Listing I.1: Implementation of Shared Data Cache module

```
from rediscluster import StrictRedisCluster
import uuid
import cPickle

class Redis:

    def __init__(self, host, port):
        self.host = host
        self.port = port

    def put(self, obj):
        startup_nodes = [{"host": self.host, "port": self.port}]
        rc = StrictRedisCluster(
            startup_nodes=startup_nodes, decode_responses=True)
        # Convert object to pickle
        serialized_object = cPickle.dumps(obj)
        key = self.generate_random_key()
        rc.set(key,serialized_object)
        return key

    def get(self, key):
        startup_nodes = [{"host": self.host, "port": self.port}]
        rc = StrictRedisCluster(
            startup_nodes=startup_nodes, decode_responses=True)
        return cPickle.loads(rc.get(key))

    def generate_random_key(self):
        random = str(uuid.uuid4())
        random = random.upper()
        random = random.replace("-", "")
        return random[0:16]
```