UNIVERSITY OF CALIFORNIA SAN DIEGO

Efficient Systems for Advanced Data Analytics

A Thesis submitted in partial satisfaction of the
requirements for the degree Master of Science

in

Computer Science

by

Liangde Li

Committee in charge:

Professor Arun Kumar, Chair
Professor Niema Moshiri
Professor Lawrence Saul

2022

The Thesis of Liangde Li is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2022

DEDICATION

I dedicate my thesis work to my parents and grandparents. A special feeling of gratitude to my loving parents, whose encouraging words lighten up the dark night in my life.

I also dedicate this thesis to my advisor and thesis committee members. My unparalleled advisor Prof. Arun Kumar led me to research computer science. And I cherish my thesis committee members, Prof. Niema Moshiri and Prof. Lawrence Saul, for their time taken to review this thesis.

Finally, I dedicate this thesis to my senior lab peers, Dr. Supun Nakandala and Yuhao Zhang, for their help and suggestion during my work.

EPIGRAPH

Systems are to algorithms what soil to seed.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Arun Kumar for his support as the chair of my committee, my advisor. Through multiple drafts and many long nights, his guidance has proved invaluable.

Chapter 2 is currently being prepared for submission for publication of the material. Li, Liangde; Kumar, Arun. The thesis author was the co-author of this material.

Chapter 3 contains material from "Intermittent Human-in-the-Loop Model Selection Using Cerebro: A Demonstration", which appears in Proceedings of VLDB Endowment, Volume 14, Issue 12, Pages 2687-2690. Li, Liangde; Nakandala, Supun; Kumar, Arun. The thesis author was the co-author of this paper and contributed to the design and implementation of the system.

VITA

2020	Bachelor of Engineering(Hons) in Mechatronic Engineering University of New South Wales, Australia
2020	Bachelor of Science in Computer Science University of New South Wales, Australia
2022	Master of Science in Computer Science, University of California San Diego

ABSTRACT OF THE THESIS

Efficient Systems for Advanced Data Analytics

by

Liangde Li

Master of Science in Computer Science

University of California San Diego, 2022

Professor Arun Kumar, Chair

Many algorithms have evolved in the past decade. Genetic analysis and deep learning are two representative groups of them. With the progress in the big data era and decreasing cost of data collection, these algorithms are being applied to a larger volume of data, leading to the new development of efficient systems in these domain areas. We evaluate the performance of some new emerging salable genetic analysis tools to provide practitioners with some guidelines. We also propose a new paradigm that we call *intermittent human-in-the-loop model selection* to mitigate pains in deep learning model selection.

Chapter 1

Introduction

Data analytics algorithms are being applied to boosting volume of data. People observe and identify many efficiency issues during their work, which attracts the computer systems community to develop many scalable and efficient systems trying to solve them. Two representative types are scalable genetic analysis tools in the bioinformatics area and distributed model training in the machine learning community.

Genetic analysis is a popular topic in the 21st century, with many efforts have been made to develop tools for it. However, with the exploding of genetic data because of the drop in DNA sequencing price in Moore's Law, scalability issues arise. Some efforts have been made to address it, but there is a lack of unified comparative evaluation of these systems. We take a major step towards filling this gap. We introduce a suite of Genome-Wide Association Study (GWAS) specific tests based on our experience with GWAS workloads. We evaluate a few popular or emerging genetic analysis systems using our tests: PLINK, Hail, and Sgkit. Our study has revealed some sub-optimal features and bottlenecks in these systems. Our findings have already led to improvements in Sgkit.

Deep learning (DL) is revolutionizing many fields. However, there is a major bottleneck to the broad adoption of DL: the *pain of model selection*, which requires exploring a large config space of model architecture and training hyper-parameters before picking the best model. The two popular paradigms for exploring this config space pose a false dichotomy. AutoML-based

model selection explores configs with high throughput but uses human intuition minimally. Alternatively, interactive human-in-the-loop model selection entirely relies on the human instinct to explore the config space but often has extremely low throughput. To mitigate the above drawbacks, we propose a new paradigm for model selection that we call *intermittent human-in-the-loop model selection*.

Chapter 2

Benchmark of Genetic Analysis Tools

2.1 Introduction

A deep understanding of Human DNA is vital for Disease Prediction and Precision Medicine. Therefore, tools have been developed in the last two decades for genetic analysis to explore hidden information in DNA. Human Genome DNA sequence was first determined in 2003 in Human Genome Project [5], which took nearly 3 billion dollars. This cost dropped dramatically at the rate near Moore's Law to around 1000 dollars nowadays. If only some of the markers are wanted, there are services online to complete them at a price of fewer than one hundred dollars. This cost reduction makes it possible to determine a larger population's DNA sequence. Thus, genetic analysis tools must support large-scale population genome data and potential scale-out for a growing volume of data.

This situation has led to the development of *scalable distributed* Genetic Analysis systems in recent years, which provide genetics-specific built-in functions and allow users to scale their analysis pipelines to distributed computing and storage systems without manually handling data distribution and communication. These systems provide API embedded in Python as their front end and aim to scale complex statistical genetic analysis, such as the GWAS pipeline.

While the recent activity on scalable genetic analysis systems has led to many new tools and evaluation of their performance, two fundamental practical questions remain largely unanswered: *From a comparative standpoint, how effective and efficient are such systems for*

scalable genetic analysis? By effectiveness, we mean how "easy" they are to use in terms of coding amount and setup logistics. We mean how "fast" they run in different settings by efficiency. *Is it worth going distributed for a given data scale?* By worthy, we mean when these distributed systems would have an advantage against mature single-machine systems. Existing work [12] on such systems focuses on one system and explores its performance in different settings. This *lack of a unified comparative understanding* of these systems may bring extra work and cost to users, especially in the commercial, public cloud environment.

We take a step towards filling this gap by introducing a suite of performance evaluation tests for genetic analysis systems and performing an empirical comparison of several popular and emerging genetic systems using our tests. We delineate five orthogonal axes: *task complexity*, *Data Scale*, *Computational Scale* , *Data Shape* and *Task Hyperparameter*. We include five joint GWAS operations with different computation behaviors and purposes for task complexity. We focus on commodity CPU clusters for computational scale, varying the number of CPU cores in single-node mode and cluster nodes in distributed mode.

We compare the following systems: PLINK [21], Hail [25] and Sgkit [7]. They are selected due to their popularity and representativeness. PLINK is a widely used mature single node system. Hail is built on Spark's mature distributed computing engine, providing APIs and Expressions. Sgkit is built on top of the Dask engine, providing data scientists friendly syntax and environment.

Based on our empirical study, we recognize and summarize the strengths and weaknesses of each system to give a guideline for practitioners to decide which one fits their needs most. We also identify some gaps and propose some open research questions. Overall, this work makes the following contributions.

- To the best of our knowledge, this is the first work to create a unified framework for evaluating popular genetic analysis systems.
- Using our tests, we perform an extensive empirical study comparing PLINK, Hail, and

Sgkit on a single node and analyzing Hail and Sgkit on distributed clusters.

- We analyze and distill our experience into guidelines for practitioners and propose some open research questions for future work.
- Our findings have already resulted in bug fixes for Sgkit.

2.2 Background

2.2.1 GWAS and Genetic Analysis

Genome-Wide Association Study (GWAS) is an approach in the bioinformatics field, genetic analysis to find the association between disease or trait (Phenotype) and DNA (Genotype). Human DNA has different coding in specific locations of DNA sequence, where this collection of multiple versions of coding is a set of alleles. Capital letters like "A" often represent the dominant allele and little letters like "a" for the recessive allele. Genotype is the occurrence of these alleles in an individual, and there are three genotypes in this biallelic case: "AA", "Aa", and "aa". Among them, "AA" and "aa" are called homozygous, and "Aa" is called heterozygous. The meaning of symbols is shown in Table 2.1. Many people's genomes and phenotypic data are required for GWAS, which uses a series of algorithms to reveal hidden information. This information may include that environment has pressure on individuals, causing the diminishing of a specific allele. Or a phenotype has a strong correlation with some genotypes. The goal is the ability to use DNA to predict disease with statistical confidence.

Table 2.1. Notations of Genetics Symbols

Symbol	Notation
m	Number of variants in dataset
n	Number of samples in dataset
i	Index of variants
j	Index of samples
A, B	Dominant allele
a, b	Recessive allele
n_{AA}, n_{Aa}	Number of genotype call AA, Aa

Genetic analysis systems aim to provide users with a canned roster of algorithms and APIs to perform data analytics on users' data. At times, data transformation is necessary before the analysis pipelines. These systems encapsulate the most prevalent genetic analysis algorithms and abstract away low-level systems issues like parallel/distributed computation. Our goal is not to design a new genetic analysis system but to provide a systematic and quantitative comparison of popular and representative systems using our carefully designed suite of tests and, hopefully, provide a reference for practitioners to choose between these systems.

2.2.2 Overview of Compared System

PLINK [21] is an open-source command-line program written in C/C++ and focuses on the phenotype/genotype datasets. It is easy to start using, and the users can do the data analysis by just opening a terminal and running the commands with corresponding arguments. The datasets are represented by two tabular-format plain text files, PED and MAP. PED file contains the information of samples, and MAP comprises the information of Single-nucleotide polymorphism (SNP) of each Chromosome position. While PED is in plain text format, PLINK also provides BED, a binary version of PED. PLINK supports the data management operations, such as recoding, updating data, and merging the datasets. In addition, PLINK has a robust library of advanced analytic tools on genotype datasets. The users can do advanced analysis, such as association testing, population stratification, and LD estimation, by calling the PLINK command corresponding arguments. PLINK also supports parallel computation using multi-cores for some tasks, especially those heavy computation tasks. However, PLINK does not offer distributed implementation, so we skip it for the distributed settings.

Hail [25] is an open-source Python library for data analysis tools specialized in genetic datasets. The input dataset can be imported from many sources, such as CSV and JSON, and then read as a single tabular dataset called Table, like SQL table, Pandas DataFrame, or Spark DataFrame. Hail provides the basic operations over Table, such as aggregation, filtering, grouping, and joining. Moreover, a datatype called MatrixTable, a distributed extension of Table,

specializes in the genetic datasets. The Matrix Table can be imported from VCF, BGEN, and BED file formats and the data formats supported by the Table above. Since the genetic datasets are sparse, the Matrix Table is stored in "coordinate form", so the storage we need is much lower than that for the Table. MatrixTable supports some high-level functions on the genetic dataset, such as linear regression, PCA, normalization, and all the functions for the Table. Operations on MatrixTable can be distributed computing by Spark.

SgKit is an open-source Python package to offer data analysis on genetic datasets. It can read from various population genomic variants file formats, such as VCF, BGEN, and BED, and store the data in Zarr format. Sgkit represents the data in a unique structure of Xarray, a dataset that supports the data in a multi-dimensional tensor. A significant advantage is that the Sgkit implements many classical GWAS calculations on the data, such as association testing, LD pruning, and relatedness estimation, and it is easy to apply these functions to the genetic dataset. Sgkit scales the tasks in parallel by Dask [23], a popular python environment scalable computation framework. However, the scalability for some functions, such as PCA and LD estimation, is in the bottleneck.

2.3 Data Preparation

The data used in the experiment are from Phase 3 of the 1000 Genome Project, collected from 2504 people (samples). We select the data from Chromosome 1 to Chromosome 22 for most experiments and Chromosome 21 solely to perform the drilled-down tests on Sgkit and Hail in eager mode experiments to expose more details.

The 1000 Genome (1KG) Phase 3 data are publicly available online for each chromosome in compressed Variant Call Format(VCF), vcf.bgz. By using Bcftools [24], we concatenate them into 1 vcf.bgz file, which has 14 GB in size. As VCF is in plain text format, the uncompressed version would be much larger. For example, the compressed Chromosome 1 data have 1.1 GB but 61 GB after decompressing. In the uncompressed version VCF file, the first few dozen lines

are header to describe the metadata of this file. The central part of the data is separated into lines, where each line corresponds to one variant and expresses all samples' genotype calls. Thus, a VCF file can be considered a plain text matrix, where each row is a variant and each column is a sample and, of course, has metadata in the header.

Because Sgkit cannot directly create a dataset by reading from a VCF file, it always needs to read from Zarr storage. We convert these VCF data to each tool's native data format, Zarr for Sgkit, MatrixTable for Hail, and BED for PLINK, to make it a fair comparison. In addition, they are all binary formats. The 1000 Genome data have 23GB in Zarr, 15GB in MatrixTable, and 48 GB in BED. Zarr is a format to store N-dimensional arrays and can be chunked into small pieces for distributed operations on these arrays. Sgkit loads Zarr stored data into memory as Xarray's Dataset objects, as shown in figure 2.1. The central part of the data is an n-dimensional array with three standard axes, *Variant, Sample* and *Ploidy*. Ploidy is two in the case of 1000 Genome data because humans are diploid; that is, a person usually has 23 pairs of chromosomes, and each pair has two homologous copies from the father and mother. *Variant Data* is metadata for each variant-row in the figure and stored globally. Similarly, *Sample Data* for each sample-column in the figure. MatrixTable is a format introduced by Hail developers, a Matrix, like what appeared in Sgkit, with metadata for each row and column. BED is a primary format of genotype calls for PLINK and should be distinguished from the other well-known 'BED' (Browser Extensible Data) format in genetics. The BED file should be used with BIM and FAM files also outcomes when converting VCF to BED.

There are no public phenotype data for 2504 people for privacy reasons. So, we create some synthetic numeric phenotype data, a numeric number for each person, to complete the association study. These phenotype data are stored as plain text files. PLINK can read it directly. Sgkit must assign it as a field to the dataset, and Hail must read it into native Table format and then annotate the MatrixTable with this phenotype data Table.

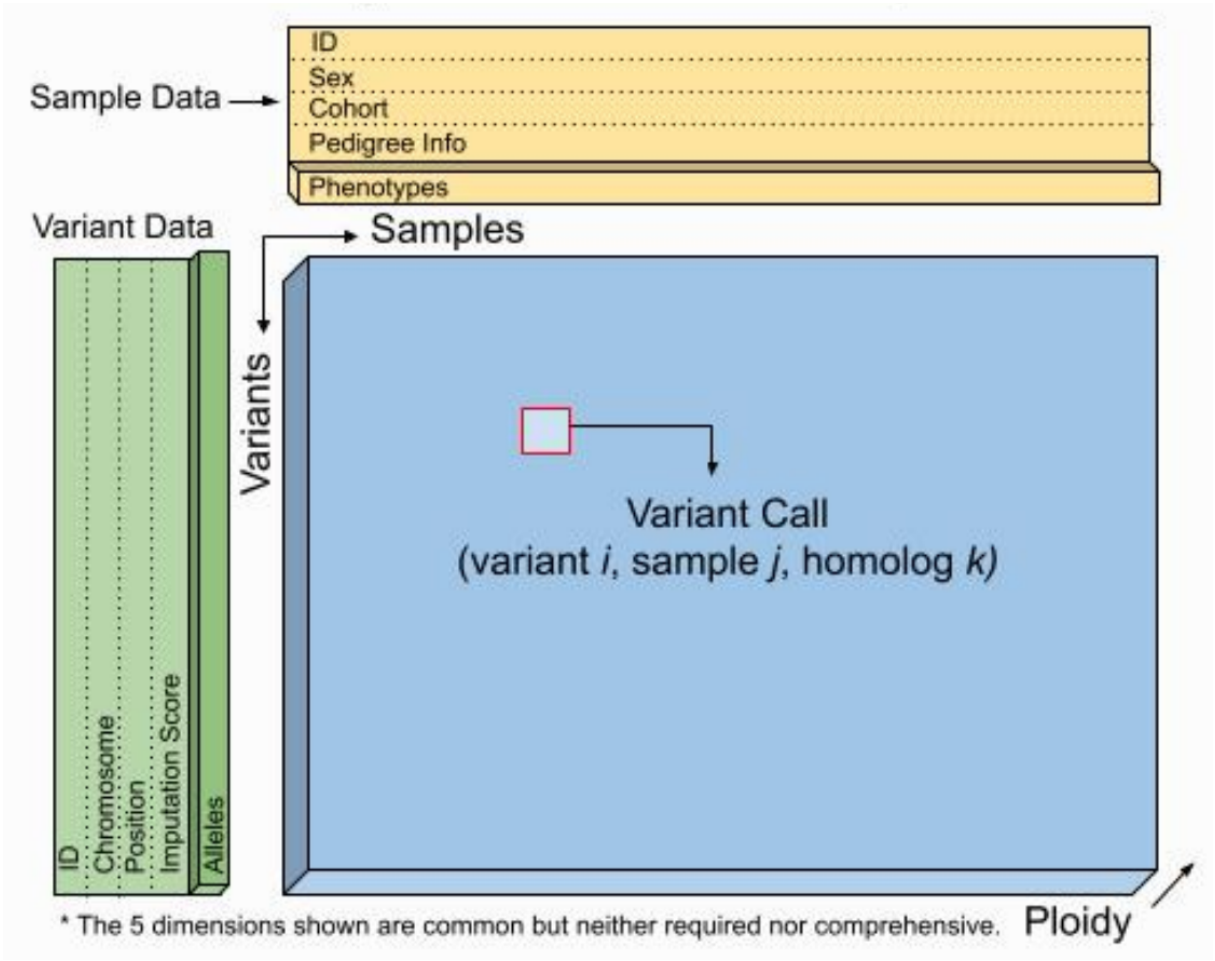

Figure 2.1. Xarray Dataset of Genetic Data in Sgkit [6]

Algorithm 1. Creating Synthetic Phenotype Data

Data Genotype Call $X(m,n,2)$

1: $XL = sum(X, axis = -1)$
2: $AL = random(size = m)$
3: $Y = XL^T AL + random(size = n)$
4: **return** Y

2.4 Description of Tests

We delineate our experiment suites along five orthogonal axes. We explain the settings and list parameters to vary to test these tools thoroughly for each axis.

Axis 1: Task Complexity. We have three groups of tasks. The first group is basic statistical methods and is usually performed for each variant. The second involves operations on the original dataset based on some computation results. The last group is Linear Algebra methods in genetics.

Axis 2: Data Scale Factors. In the single-node experiment, we use chromosome 21 and the whole genome chromosomes data, so they vary in several variants. In the distributed experiment, we use synthetic varying in the number of samples because it is more likely to collect DNA sequence data from more people instead of finding more variants in genomes.

Axis 3: Computational Scale Factors. We have two sets of experiments to address the computational scale factors. In the single-node experiment, we vary the number of cores used, and in the multi-node experiment, we vary the number of machines while using all cores.

Axis 4: Data Shape Factors. We have two types of shardings of data for Sgkit, *tile sharding* and *row sharding*. We vary the sharding types in the distributed experiments.

Axis 5: Task Hyperparameter Factors. Some tasks in our experiment contain hyperparameters, and we examine how these hyperparameters affect the runtime.

2.4.1 Task Complexity

We tested each system by the operations in three groups Task Complexity and Difficulty. The difference between each level is the complexity of the tasks. The first level is EASY. The operations will only do some statistical summary, linear, and easy to scale up at this level. The second level is MEDIUM. The procedures in this level will do some calculations across the datasets, such as HWE exact test and LD-prune. The third level is HARD. The operations have a considerable time complexity (more than quadratic), such as dimensional reduction technique

PCA, SVD, or Ordinary Least Square Linear Regression (OLS) model. These algorithms are summarized in Table 2.2.

Table 2.2. Algorithms for Axis 1 (task complexity).

Algorithm	Input	Output	Complexity	Purpose
Allele Frequency (AF)	X(m,n,2)	AF(m,2)	O(mn)	Basic Statistical Summary.
Hardy-Weinberg Equilibrium (HWE)	X(m,n,2)	p(m)	O(mn)	Finding variants that are not evolving in the population.
Linkage Disequilibrium Prune (LD-Prune)	X(m,n,2)	Depends	O(mw)	Finding a subset of uncorrelated variants.
Principal Component Analysis (PCA)	X(m,n,2)	PCs(m,nPC)	$O(m^3+m^2n)$	Finding ancestry relationship in data population.
Ordinary Least Square Linear Regression (OLS)	X(m,n,2), y(n)	Beta(m)	$O(m^3+m^2n)$	Identifying correlation between variants and traits.

Statistical Summary

Doing some basic statistical summary is essential and always the first thing to do after collecting data. We choose the Allele Frequency as our goal in this group to represent tools' performance on those statistic work. An allele is one version of a gene at a specific location(locus), for example, in genotype Aa, where A and a are both alleles. Allele frequency is to calculate A's frequency and a's frequency in the population of data collection. Allele Frequency is one of the most frequently used statistic numbers in genetics.

Algorithm 2. Allele Frequency

Data Genotype Call $X(m,n,2)$

for each variant i **do**
  Define n_{Aa} as the number of heterozygous Aa;
  $n_{Aa} \leftarrow sum(x[i,:] == [1,1])$
  Define n_{AA} and n_{aa} as the number of homozygous AA and aa;
  $n_{AA} \leftarrow sum(x[i,:] == [2,0])$
  $n_{aa} \leftarrow sum(x[i,:] == [0,2])$
  $n \leftarrow n_{AA} + n_{Aa} + n_{aa}$
  $AF[i,0] \leftarrow \frac{2*n_{AA}+n_{Aa}}{2*n}$
  $AF[i,1] \leftarrow \frac{2*n_{aa}+n_{Aa}}{2*n}$
end for
return AF

Genetic Computation and Operation

The tasks in this group are more complex than simple statistical summary as it involves some genetic calculation and can potentially do some operation on the original dataset based on

these calculated values. The two representations we choose are Hardy-Weinberg Equilibrium exact test and Linkage Disequilibrium Prune.

Hardy-Weinberg Equilibrium means, without migration, mutation, and natural selection, that a random mating large size population's genotype frequency should be able to get directly from allele frequency. For example, if allele A's frequency is p and allele a's frequency is q and we have $p+q=1$. Then genotypes AA, Aa, and aa should have frequency $p^2, 2pq$ and q^2 respectively. This was first raised in the early 20th century [8] [27]. However, there can be a deviation from the observed genotypes call, and the expected genotypes call calculated from allele frequency. So, Chi-squared test can be used to tell its statistical significance. After getting the Chi-square value and referring to the Chi-square distribution table, we can get the p-value for each variant. However, genetic analysis tools usually implement it by an improved method [28].

Algorithm 3. Hardy-Weinberg Equilibrium p-value

for each variant **do**
  Define n_{Aa} as the number of heterozygous Aa;
  Define n_{AA} and n_{aa} as the number of homozygous AA and aa;
  Define n_A as the number of allele A: $2*n_{AA}+n_{Aa}$;
  $n \leftarrow n_{AA}+n_{Aa}+n_{aa}$
  Calculate the mid point: mid $= \lfloor n_A*(2*n-n_A)/(2*n) \rfloor$;
  If mid has different odevity from n_A **Then** mid = mid $+1$;
  Set $p($ mid $)=1$;
  $x \leftarrow mid$
  while $x \geq 0$ **do**
    $p(x-2) = p(x)\frac{n_{Aa}(n_{Aa}-1)}{4(n_{AA}+1)(n_{aa}+1)}$
    $x \leftarrow x-2$
  end while
  $x \leftarrow mid$
  while $\leq n_A$ **do**
    $p(x+2) = p(x)\frac{4n_{AA}(n_{aa})}{(n_{Aa}+2)(n_{Aa}+1)}$
    $x \leftarrow x+2$
  end while
end for

Figure2.2 shows the difference between Linkage Equilibrium and Linkage Disequilibrium. Linkage Disequilibrium means the nearby alleles on the same chromosome are usually associated. For example, genotype AB has abnormal high frequency, as 49% in B) as opposed to 25% in A). So Linkage Disequilibrium Prune is usually used to get a set of uncorrelated variants. The correlation coefficient r^2 [9] is usually used to measure it, and users set a threshold for it. The pruning process is considered a window at a time. For each pair of variants, if their pairwise r^2

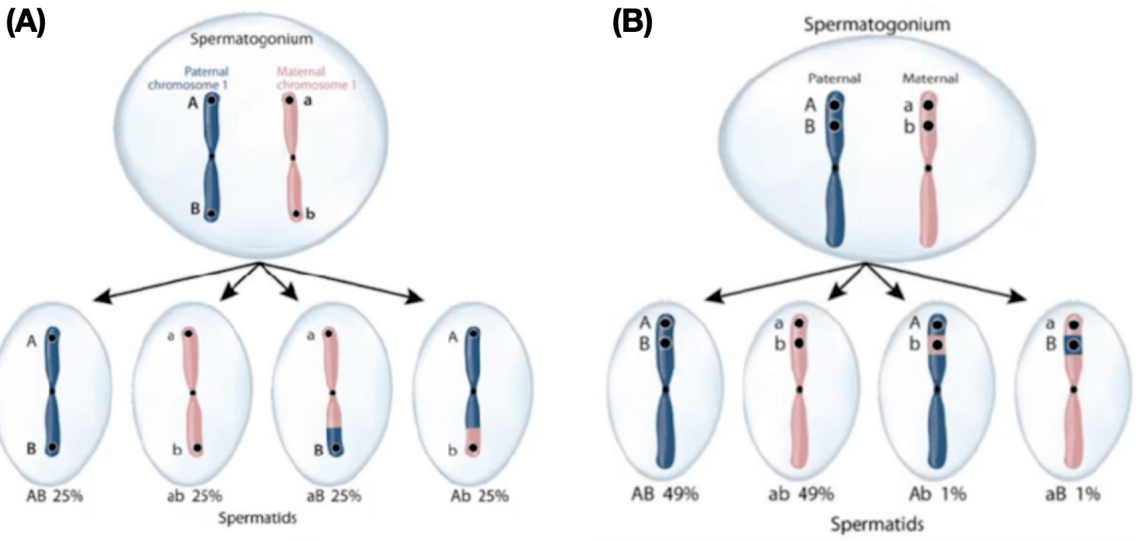

Figure 2.2. A) Linkage Equilibrium. B) Linkage Disequilibrium. [20]

is under the threshold, drop the variant with a smaller Minor Allele Frequency.

Algorithm 4. Linkage Disequilibrium prune

Input $r2, W, step$
for each variant i **do**
  for each $j \leq i+W \wedge j > i$ **do**
    Define variant i has Alleles A and a
    Define variant j has Alleles B and b
    $D = P(AB) - AF(A) * AF(B)$
    $r^2 = \frac{D^2}{AF(A)AF(a)AF(B)AF(b)}$
    if $r^2 < r2$ **then**
      Prune the variant that min($AF(A)$, $AF(a)$, $AF(B)$, $AF(b)$) belongs to
    end if
  end for
  $i \leftarrow i + step$
end for

Linear Algebra

The tasks in this group are some linear algebra algorithms applied in the genetic analysis. We choose the Principal Component Analysis (PCA) and Ordinary Least Square Linear Regression (OLS). PCA is used to eliminate the confounders in the dataset, the stratified distribution of phenotype because of relationships on ancestry. The OLS is used to find the gene most related to the traits.

Algorithm 5. Principal Component Analysis

1: Z is zero mean standard normalized of X
2: Then the covariance matrix $C = Z^T Z = U\Sigma V^T$ ▷ For eigenvector method, eigenvectors of Z are principal components. For Singular Value Decomposition method, columns of V are principal components.

Algorithm 6. Ordinary Least Square Linear Regression

1: **Define** C as the covariance matrix
2: $X_L = X - \text{Proj}_C(X)$
3: $Y_L = Y - \text{Proj}_C(Y)$
4: The coefficient of X can be calculated as:
5: $\hat{\beta} = \left(X_L^T X_L\right)^{-1} X_L^T Y_L$

2.4.2 Data Scale Factors.

Use data generators like msprime [11] to generate different scales of synthetic data, varying the number of samples because it is more likely to collect data from more people instead of finding more variants in the human genome.

2.4.3 Computational Scale Factors.

We vary the number of cores used in single node experiments and the number of nodes in the distributed experiments to show how these tools scale in different methods. In single-node mode, we run experiments with 1, 2, 4, 8, and 16 cores, respectively, and we make sure these cores are all physical cores on the chips. In multi-node mode, we run experiments with 1, 2, 4, and 8 machines, respectively, and each machine has 40 logical cores.

2.4.4 Data Shape Factors.

We vary the sharding of data in multi-node experiments for Hail and Sgkit to show how these systems perform on different shapes of data. For Hail's data, we have partitions of 960, 8128, and 20318. For Sgkit's data, we have chunks of 10000x1000, 4000x2504 and 10000x2504.

2.4.5 Task Hyperparameter Factors.

The value of parameters in the tasks above can also affect the runtime. We vary the number of Principal Components in PCA, the thresholds in LD-Prune, and the window size in

LD-Prune to experiment with how different values affect the time taken.

2.5 Experimental Comparison

Experimental Environment. We executed our experiments in CloudLab "Wisconsin" site using the c220g2 instance type[22]. Each physical node has the following hardware specifications: two Intel E5-2660 v3 10-core CPUs at 2.60 GHz, 160GB ECC Memory, One Intel DC S3500 480 GB 6G SATA SSDs, Two 1.2 TB 10K RPM 6G SAS SFF HDDs and Dual-port Intel X520 10Gb NIC (PCIe v3.0, 8 lanes) network adapter. Thus, each node has 20 physical cores and hyper-threading to 40 virtual/logical cores. We run a program reading OS information to find the actual physical cores among 40 logical cores and ensure each core used in the single node experiments corresponds to a different physical core. For example, 16 cores used in single node experiments are 16 logical cores on 16 different physical cores. The operating system we used is Ubuntu 18.04. All software is installed in this vanilla Ubuntu OS, and in distributed experiments, we use virtual environments to isolate packages used between Hail and Sgkit environments. Spark and Dask clusters run in standalone mode. Table 2.3 describes the versions of key software packages we used.

Table 2.3. Software Packages and version

Name	Version	Name	Version
Python	3.8.12	Spark	3.1.2
Sgkit	0.4.0	Hadoop	3.2.2
Hail	0.2.85	Dask	2021.12.0
PLINK	1.9		

Methodology. In the single node setting, all runs are repeated three times to report their average, and we find that each set of these three runtimes is close enough. Because every PLINK command is invoked from the command line and always starts with reading the dataset and ends with writing results to files, there is no clear boundary between IO and computation. Thus, we also include IO times of Hail and Sgkit programs compared with PLINK. As Spark and Dask do

not conduct computation unless necessary, which is called lazy execution, the computation in Hail and Sgkit is triggered by force writing results to files. To further explore the time distribution of loading, execution, and writing, we also complete eager mode experiments of Hail and Sgkit on a single node. That is persisting dataset into memory before executing each algorithm and forcing count on Spark in Hail before writing output to the file system. In Sgkit, we *persist* the results of each algorithm into memory in distributed Dask memory. We do not use Dask's *compute* because it collects data into NumPy or panda's objects in python program memory space, which involves extra unnecessary work. This eager mode single node experiments use Chromosome 21 (Chr21) data to be able to fit into memory.

In a multi-node setting, all warm cache runs are repeated ten times and discard the first five runs because, in practice, we find that runtime can keep decreasing in four contiguous runs. We also discard the maximum and minimum times run in the rest five runs and use the average of the remaining three runs. We also conduct cold cache runs for some algorithms, which clear page cache, dentries, inodes in-memory cache, and buffers. These cold cache runs are repeated five times, and we discard maximum and minimum same as in warm cache runs. In a multi-node setting, HDFS is used to store datasets distributively. Data files are *put* to HDFS through a simple command-line utility, whose time is not considered in the benchmark and, in practice, takes a few hours depending on sharding sizes of data and the number of distributed nodes.

System Configuration Tuning. Although it was mentioned in SLAB[26] that tuning configuration parameters for Spark are non-trivial, that was the case for Spark version 2.2.0. We found those configuration parameters are not killing problems for Spark version 3.1.2 anymore, at least for Hail built on top of it. The only parameter matter is the number of partitions of the dataset, and it can be easily adjusted at the data transformation stage. However, we found that the performance of Sgkit is sensitive to the configurations of the Dask Distributed module, and tuning them is non-trivial and time-consuming, especially for new users of Dask. We adopted a policy of "reasonable best effort" for tuning and made Sgkit able to complete the experiments. The following suggestions are based on our experience in our hardware

environment. *scheduler.allowed-failures* defaulted 3 is too low. *scheduler.work-stealing* defaulted True can causing the systems hanging occasionally. *worker.memory.rebalance.measure* defaulted "optimistic" may be outperformed by "process". *worker.memory.spill/pause/terminate* defaulted to 0.7/0.8/0.95 can increase a bit or turning them off to get better performance, especially for memory intense workload. *comm.timeouts.connect, comm.timeouts.tcp* defaulted 30s is too low, and a number more than 60s may sounds reasonable.

2.5.1 Results for Single Node

The runtime of single node experiments in seconds, using 16 cores running in lazy mode, are summarized in Table 2.4. Three of them use 1000 Genome data, and the other two use Chromosome 21 data because of running out of time.

Table 2.4. Single node 16 cores lazy mode results in seconds on 1KG or Chr21

Task	Sgkit	Hail	PLINK
Allele Frequency	715.6	406.5	61.33
HWE	1496.77	383.4	117.7
LD Prune (Chr21)	1662	7639	216.7
PCA (Chr21)	387.1	337.9	146.0
OLS	2171.7	1764	11937

Single-Node - Allele Frequency Allele Frequency is typically calculated as part of Quality Control process at the beginning of GWAS. Hail and Sgkit implement it in a single function call as well as other Quality Control calculations, while they are usually light weight. However, because of the lazy execution model of Spark and Dask as a nature, we can try our best to only trigger Allele Frequency computation, specifically invoking writing or persisting Allele Frequency results array alone, in lazy or eager mode, respectively. We vary the number of cores on a single node to study multicore speedup behaviors. Figure 2.3 presents the A) results of total wall time of three tools in lazy mode on 1KG dataset and B) execution(AF) wall time of two tools in eager mode on chr21 dataset. The y axis is time in seconds in log scale. Hail and Sgkit present close to linear speedup as the benefit of additional cores. We note that Hail's runtimes

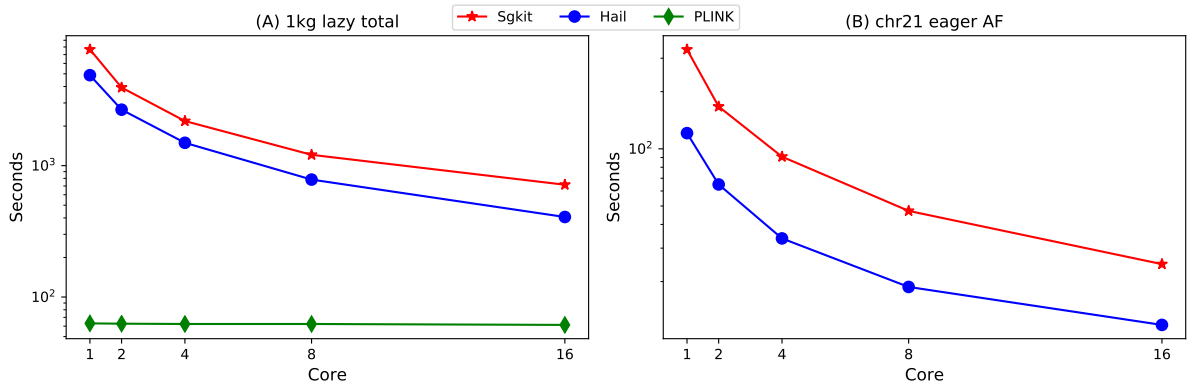

Figure 2.3. A) User perspective of allele frequency total wall time on 1000 Genome data. B) Computing time of Sgkit and Hail on allele frequency in eager loading mode on Chromosome 21 data.

are faster than Sgkit but comparable, that is no more than 100% faster in all cases. PLINK does not implement parallel computing in the Allele Frequency algorithm but presents extraordinary runtimes.

Single-Node - HWE Hardy-Weinberg Equilibrium p-value results are usually used with Allele Frequency results together to filter out variants. The naive way of calling the selection function based on AF and HWE in Sgkit caused the Dask engine two rounds of passing the genotype data to calculate AF and HWE. The optimal method provided by developers uses the method chaining[1], which creates a pipeline completing computation and selection in one pass over of data. Under the same setup of a single node as in Allele Frequency, figure 2.4 presents the A) results of total wall time of three tools in lazy mode on the 1KG dataset and B) execution(HWE+Filtering) wall time of two tools in eager mode on chr21 dataset. We note that Hail and Sgkit present almost linear speedup in both lazy modes, the whole workload on 1KG and eager mode execution workload on Chr21. PLINK is still non-parallel for HWE but outperforms significantly the other two.

Single-Node - LD Prune In single-node LD Prune experiments, because of the wide range of timeout in Sgkit and Hail on 1000 Genome data in lazy mode, we used Chromosome 21 data in lazy mode instead. Figure 2.5 presents the results. PLINK still outperforms the other

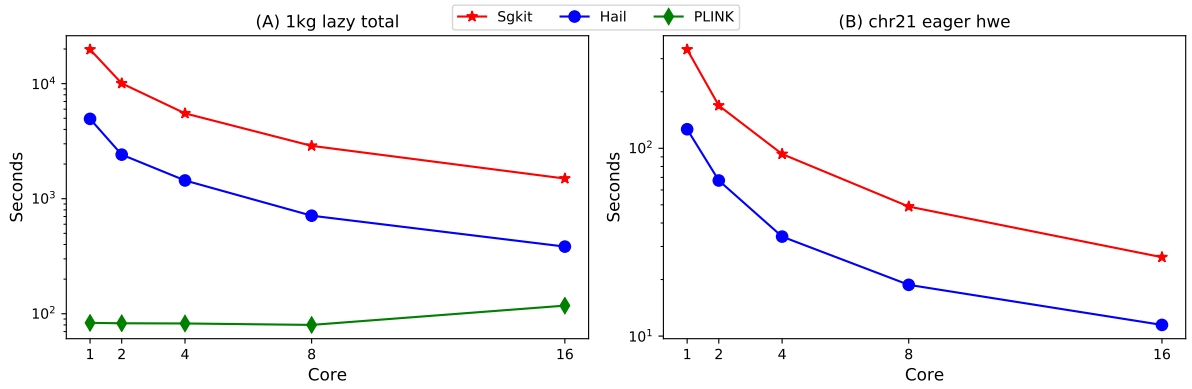

Figure 2.4. A) User perspective of HWE total wall time using 1000 Genome data. B) Computing time of Sgkit and Hail on HWE in eager loading mode using Chromosome 21 data.

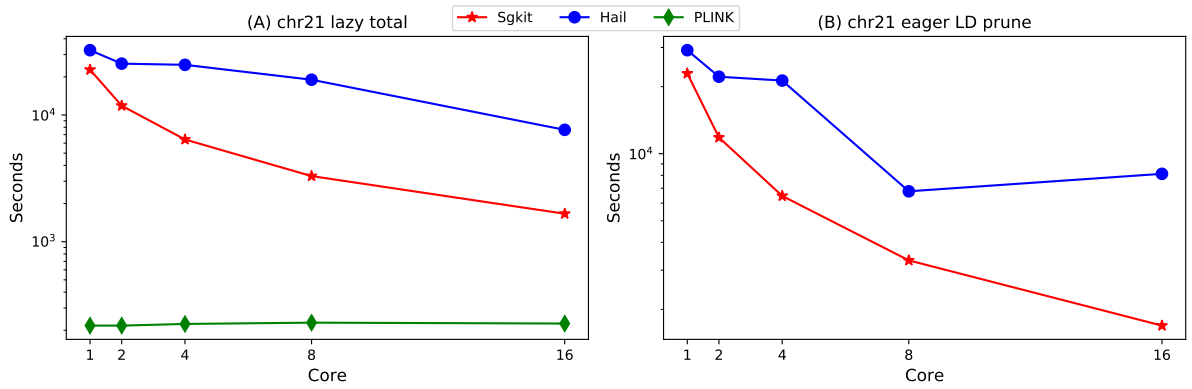

Figure 2.5. A) User perspective of LD-Prune total wall time using Chromosome 21 data. B) Computing time of Sgkit and Hail on LD Prune in eager loading mode using Chromosome 21 data.

two in all numbers of cores settings. Sgkit has linear speed up in this LD Prune task. But Hail presents badly, and its developers confirmed in their forum that LD Prune might be the most problematic method in their system.

Single-Node - PCA Same as LD Prune above, Chromosome 21 data is used for PCA in lazy mode experiment because of timeout issue on 1000 Genome data. Figure 2.6 presents the results. Hail and Sgkit achieve almost linear speedup in the range from 1 core to 4 cores but become sub-linear speedup if they add more cores. Also, Hail and Sgkit have similar whole-workload runtimes on Chromosome 21 data. PLINK implements multicore parallel computing for PCA to achieve almost linear speedup at core range from 1 to 4, but its runtimes on 16 cores

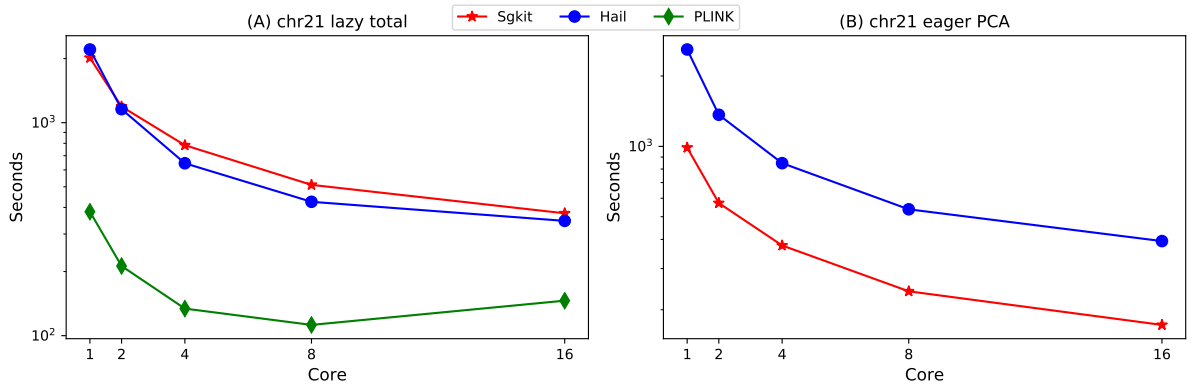

Figure 2.6. A) User perspective of PCA total wall time using Chromosome 21 data. B) Computing time of Sgkit and Hail on PCA in eager loading mode using Chromosome 21 data.

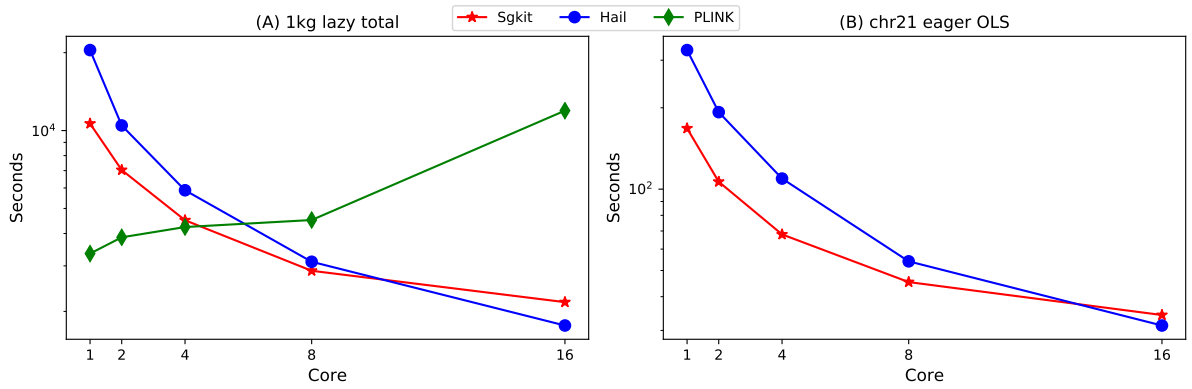

Figure 2.7. A) User perspective of OLS total wall time using 1000 Genome data. B) Computing time of Sgkit and Hail on OLS in eager loading mode using Chromosome 21 data.

are even higher than on 8 cores, and runtimes on 8 cores have little improvements on 4 cores.

Single-Node - OLS Figure 2.7 presents OLS results. PLINK implements multicore parallel computing but performs terribly. Hail has an almost linear speedup and outperforms Sgkit using 16 cores but is significantly slower using a single core.

2.5.2 Results for Multi-Node

Multi-Node - Sgkit-Dask - Vary Numbers of Workers We found that given a fixed number of cores on each node, the number of workers and the number of cores per worker affect the runtimes of Dask clusters. The default value is 1 Dask worker on each node and assigns all resources to it, including cores and memory. However, we found the program would run slow.

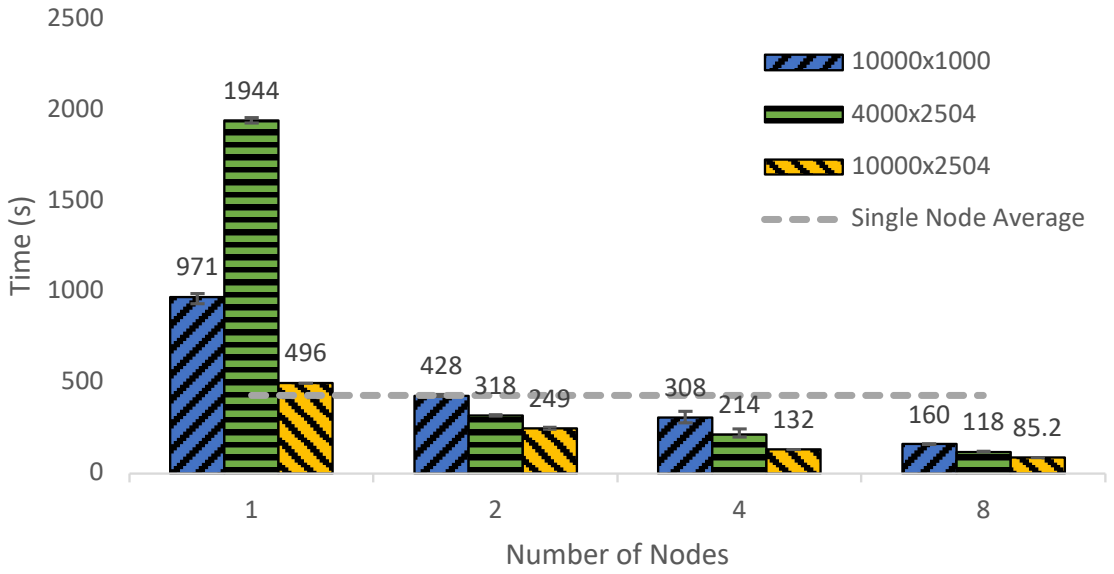

Figure 2.8. Sgkit Allele Frequency on 1000 Genome Data in Distributed Mode with 8 Workers on Each Node and Warm Cache.

Thus, we tried the "auto" value for the number of workers, which asks Dask program to decide the number of workers and cores per worker. For a node that comes with c cores, the number of workers would be found by the formula $min(factor(c)) \geq \sqrt{c}$, that is the minimum factor of c just greater than or equal to the square root of c. In our case, the square root of 40 logical cores is around 6.3, and the minimum factor of 40 that is no less than 6.3 is 8, so in "auto" mode, 8 workers would be created on each node, and each has 5 cores/threads. We also examine the 40 workers; each has one core set to compare with the auto mode. Figures 2.8, 2.9 present the results of these two settings. Generally, 40 workers set has an advantage when running with 1 or 2 nodes, but 8 workers setting has better performance using 8 nodes. This can be explained by the effect of Python Global Interpreter Lock (GIL). We found that GIL-holding functions often happen on Dask workers in our task, causing the 5 cores Dask workers only to utilize 1 core while running them. Thus 40 workers settings have advantages until using 8 nodes, which results in 320 workers leading to a significant communication load.

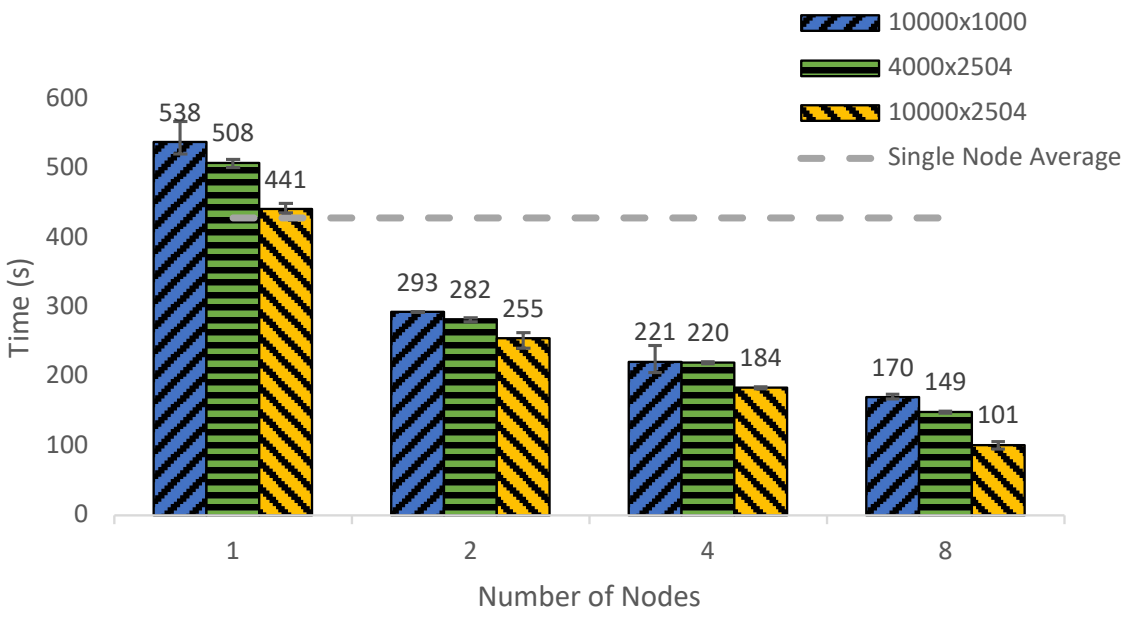

Figure 2.9. Sgkit Allele Frequency on 1000 Genome Data in Distributed Mode with 40 Workers on Each Node and Warm Cache.

Multi-Node - Sgkit-Dask - Warm/Cold Cache The differences between figures 2.8, 2.9 and figures 2.10, 2.11 present the effect of warm or cold cache in Allele Frequency method. We found this difference when we observed consistent decay of runtimes in the first a few runs of all sets of experiments, and it was especially significant when we used more nodes. The buffers, and cache of OS cache the data files, reducing file reading time significantly. Because the other algorithms usually run after Allele Frequency, we would only execute warm cache experiments for them.

Multi-Node - Vary Sharding Sgkit and its underlying dataset library Xarray provide an easy way to chunk the dataset to any size in any data dimension. Because variants and samples are two main dimensions in our dataset, we use variants x samples to represent the result of each small piece of data after chunking. The default chunking size of the 1000 Genome dataset in Sgkit is 10000x1000, which means the dataset is chunked into tiles. We propose two other prototypes of chunking, 4000x2504 and 10000x2504, which are all row sharding of data, which

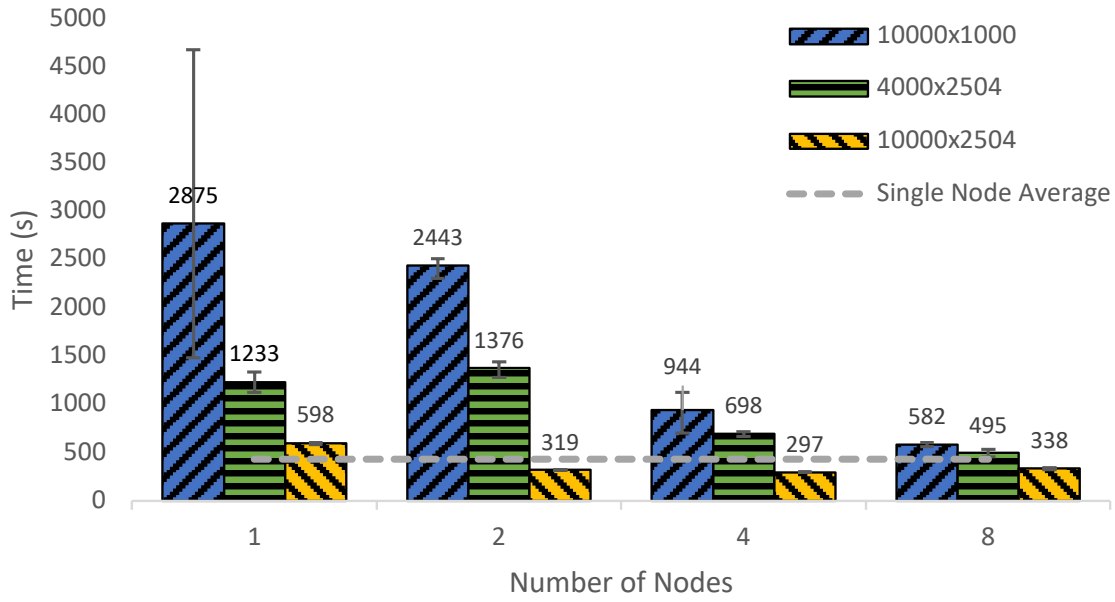

Figure 2.10. Sgkit Allele Frequency on 1000 Genome Data in Distributed Mode with 8 Workers on Each Node and Cold Cache.

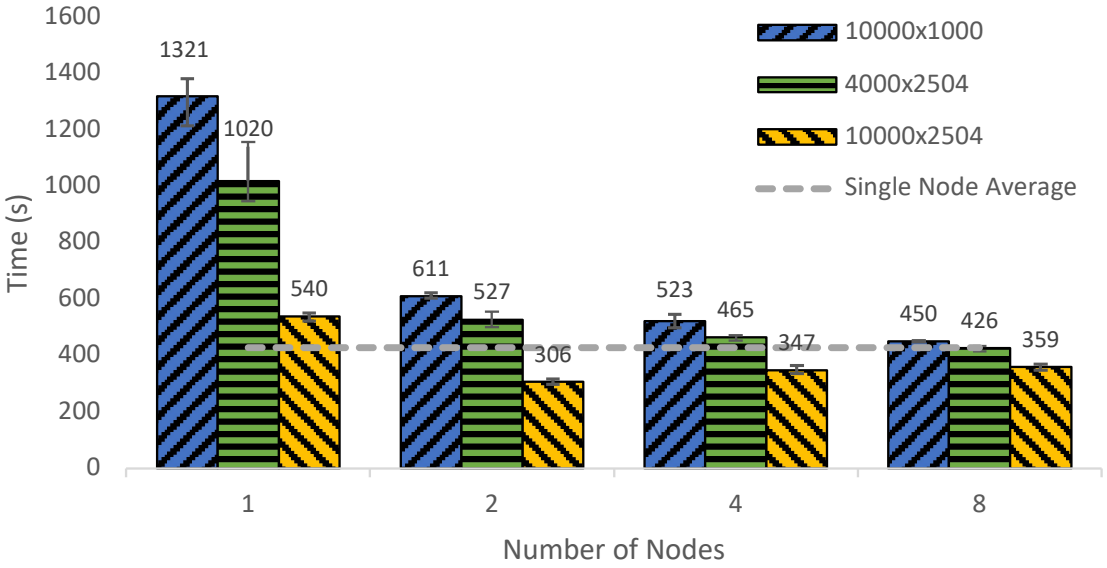

Figure 2.11. Sgkit Allele Frequency on 1000 Genome Data in Distributed Mode with 40 Workers on Each Node and Cold Cache.

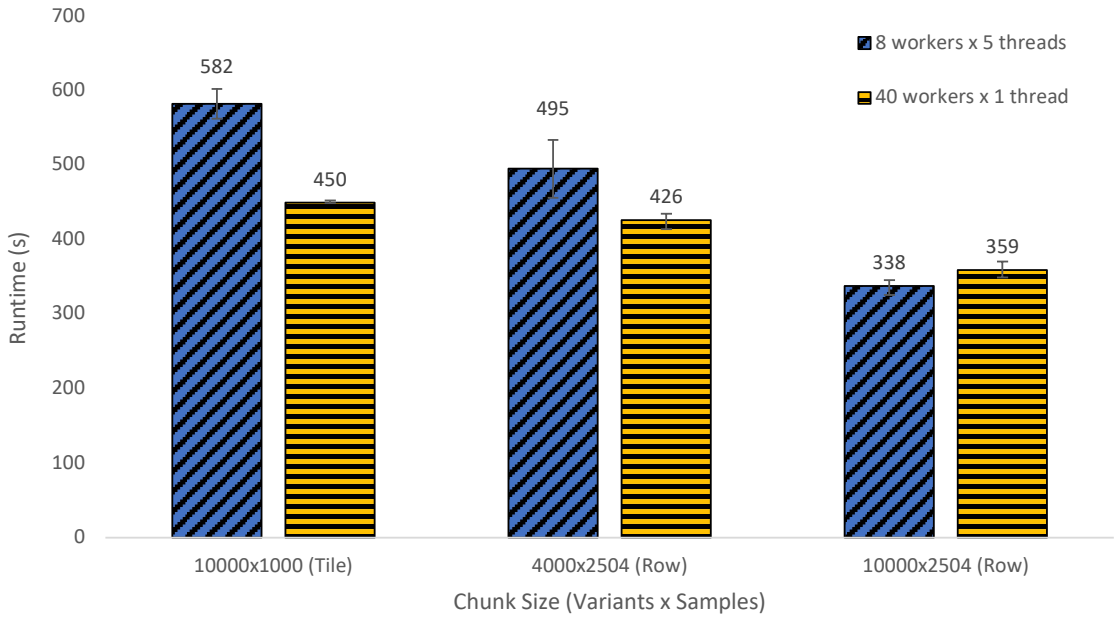

Figure 2.12. Sgkit Allele Frequency on 1000 Genome Data in Distributed Mode with 8 Workers on Each Node and Cold Cache on Different Sharding of Data.

means the complete information of any specific variant exists in only one chunk. 4000x2504 was chosen because it is roughly the same size as the default 10000x1000 chunk. 10000x2504 was chosen to explore the effect of the size of chunks on runtimes. We did not add a chunk-size smaller than 4000x2504 because it would trap into HDFS's well-known "small files issue", which causes a significant overhead when reading data from HDFS. Figures 2.12, 2.13 present cold and warm cache comparison between shardings respectively. From these Allele Frequency results, we found that 4000x2504 is better than 10000x1000, and slightly increasing the row sharding size to 10000x2504 has more improvements in our experiment environment. This conclusion is independent of the number of workers on each node and cache status. That can explain this in row sharding; each chunk has all information needed to calculate Allele Frequency. So it reduces communication and unnecessary dependency in the computation graph built by Dask. In HWE experiments, 1 node and 2 nodes experiments timeout. We observe superlinear scale-out when expanding the cluster from 4 nodes to 8 nodes in all settings, as shown in Table 2.5.

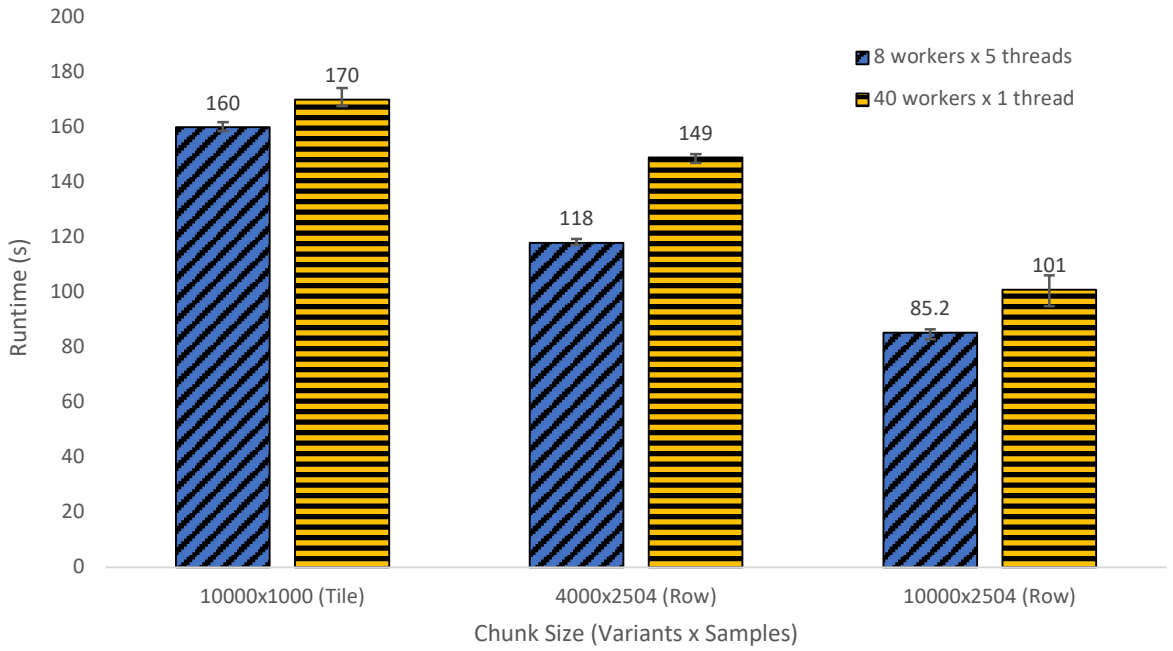

Figure 2.13. Sgkit Allele Frequency on 1000 Genome Data in Distributed Mode with 8 Workers on Each Node and Warm Cache on Different Sharding of Data.

Table 2.5. Resulting Runtimes in seconds of HWE using Distributed Sgkit with 8 Workers on Each Node.

	Warm Cache		**Cold Cache**	
	4 nodes	**8 nodes**	**4 nodes**	**8 nodes**
10000x1000	4714.1	549.6	4757.9	618.5
4000x2504	4683.3	445.9	4740.1	1172.5
10000x2504	3237.5	276.7	3791.9	347.9

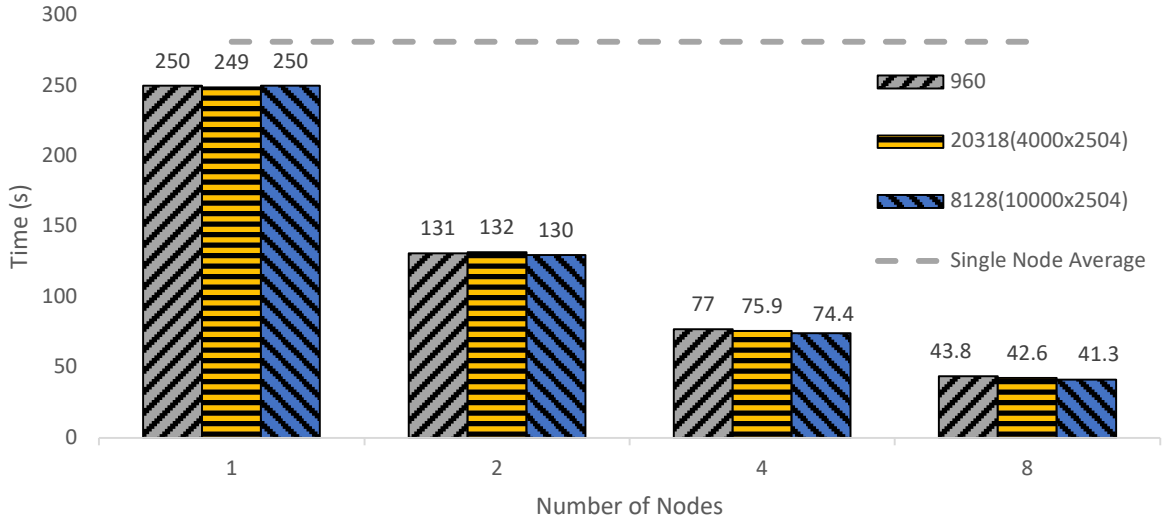

Figure 2.14. Hail Allele Frequency on 1000 Genome Data in Distributed Mode.

Hail only provides row shardings, and this number is controlled by a configurable parameter, the number of partitions. It is suggested by developers of Hail to have at least 2-4 partitions per core, so we chose number 3 and re-partitioned the dataset into 960 partitions as the suggested value because we have 320 logical cores using 8 nodes. To make a fair comparison with Sgkit, we partitioned Hail's dataset into 4000x2504 and 10000x2504 shardings, which correspond to 20318 and 8128 partitions. Figures 2.14, 2.15 present the results of distributed Hail on Allele Frequency and HWE respectively. We found Hail scales out almost linearly, independent of the number of partitions, as Hail performs the same on different partitions if this partition number is greater than the minimum requirement.

2.6 Analysis and Discussion

We first guide practitioners on the strengths and weaknesses of these systems. We then give a summary of the key takeaways from our empirical analysis. Finally, we propose some open research questions.

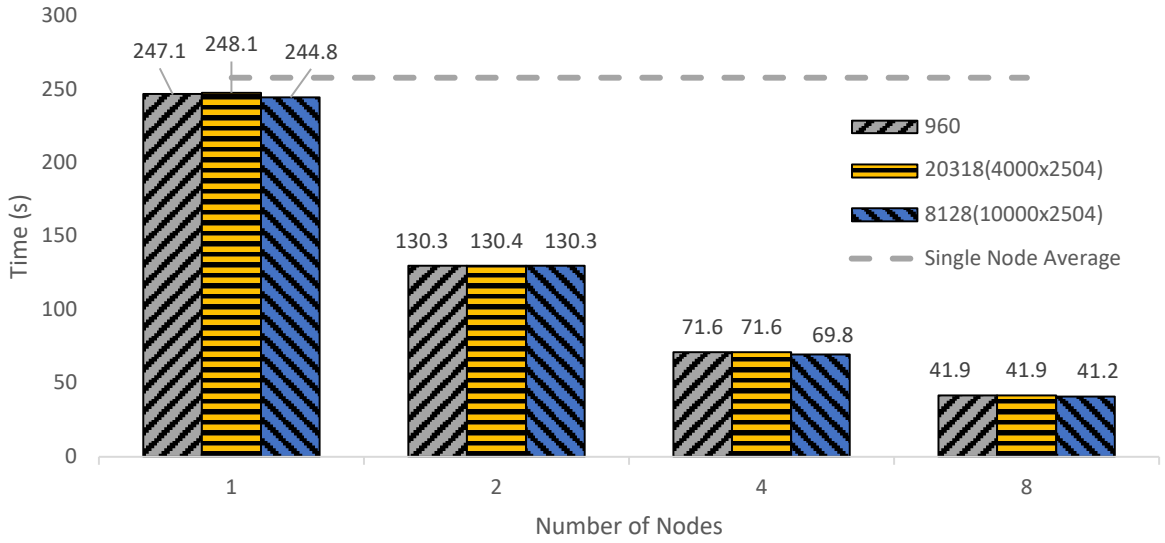

Figure 2.15. Hail HWE on 1000 Genome Data in Distributed Mode.

2.6.1 Guidelines for Practitioners

We now summarize each tool's strengths and weaknesses.

PLINK. We found PLINK to be the fastest in most of the single node experiments. This has several reasons, including its implementation language C/C++. The next is its native data format that the genotype data are stored how exists in memory, so no interpreting or converting is needed when loading it. As it is the first tool being developed among the three tools, people have well-tuned it in the past decade. One drawback is its closed ecosystem, as it only provides a roster of functionalities and lacks support for customized analysis. And users cannot analyze the data interactively as the nature of being a command-line tool. Another drawback is its lack of distributed support. So, PLINK is recommended if practitioners have a clear goal about what routine analysis is needed, and these functionalities are provided by PLINK; also, the data must be fit in one node.

Hail. Hail has excellent paralleled/distributed features and well-documented instructions and tutorials. It also provides a helpful and popular forum to help users. A lot of the time,

when we met problems, issues, or bugs using Hail, we found there were already discussions and solutions in its forum. However, Hail uses fewer open-source libraries than Sgkit, so it will benefit less from the evolution of other popular libraries in the Python ecosystem but has better control over its quality if its developers put in enough effort. And we do see its developer devote to it as they do new releases biweekly. Finally, compared to PLINK, it has a steeper learning curve at the beginning for people who are not familiar with programming.

Sgkit. We found Sgkit to be the one most friendly to practitioners familiar with the Python ecosystem, especially Numpy and Pandas. This is because it uses many popular third-party libraries, from representing the dataset to storing the data. Thus, we reckon Sgkit is more potent for performing customized analysis and operations on the data. However, the Dask cluster is hard to tune, especially customizing its configurations to users' hardware. Thus, we suggest developers of Sgkit provide more instructions on tuning the Dask config for Sgkit users or implementing auto-tuning. As Sgkit is the newest among these three and is being actively developed and fixed, more tuning of code details can potentially improve its performance.

2.6.2 Open Research Questions

We identify a few key gaps that require more research from the data systems community.

Auto-tuning Data Shape and System Configuration Parameters. The two distributed systems we examined provide the functionality to change the data shape and size of sharding, which means leaving this responsibility to users to find the optimal shape. Also, the default values they provide are far from optimal. Thus, more work is needed to find a good enough data shape for users automatically. More importantly, auto-tuning for Dask configuration parameters is requested more urgently. Extending the lessons of auto-tuning in Spark from version 2 to version 3 to Dask is another avenue for new research.

Fault Tolerance of Dask Distributed. We experienced Dask cluster hanging many times, although we finally found it is mostly related to *work-stealing*. We argue that improving the fault tolerance property of Dask Distributed would be beneficial.

Chapter 2 is currently being prepared for submission for publication of the material. Li, Liangde; Kumar, Arun. The thesis author was the co-author of this material.

Chapter 3

Intermittent Human-in-the-Loop Model Selection

3.1 Introduction

Deep learning (DL) is revolutionizing many fields. It is now being used in various domains including e-commerce, web, and even in critical applications such as in healthcare. However, there is a major bottleneck for the wide adoption of DL: the *pain of model selection*. The accuracy of a trained model heavily depends on the model architecture and hyper-parameter values used during training. Thus, practitioners often have to perform a search over the potential config space, in order to pick the best model.

Paradigms for Searching the Config Space From our conversations with DL practitioners and our own experience building large-scale DL applications we find two main paradigms: 1) AutoML and 2) interactive human-in-the-loop. In the AutoML paradigm, the user will initiate a model selection workload by specifying a config search space and a canned AutoML procedure. AutoML procedures implement a search heuristic such as Bayesian optimization (e.g., HyperOpt [2]), evolutionary search (e.g., PBT [10]), and random search (e.g., ASHA [16]). It then uses the parallelism available in a cluster (or a single machine) to explore configs with high throughput. As model selection progresses, the user will receive the results of the explored configs. Figure 3.1(A) presents an illustration of this paradigm. While there are advanced AutoML procedure implementations of the above-mentioned heuristics, recent surveys [3] have

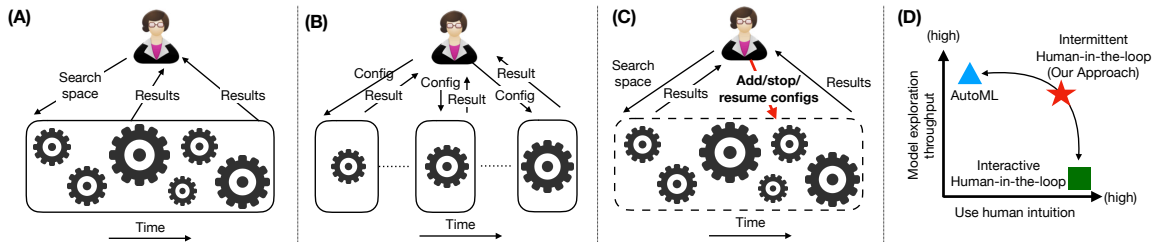

Figure 3.1. A) AutoML-based model selection. B) Interactive human-in-the-loop model selection. C) Our paradigm of *intermittent human-in-the-loop model selection*. D) Qualitative comparison of different paradigms.

shown that an overwhelming majority of ML researchers and practitioners often use simple techniques like grid (explore all configs) or random (randomly sample configs) search.

In interactive human-in-the-loop model selection [30, 4], the user retains full control over the search process. They will explicitly specify a config (or few configs) to explore and wait until it finishes. Based on the results of the explored configs and human intuition about the search space, they will specify the next config (or set of configs) to explore. Figure 3.1(B) illustrates this paradigm.

False Dichotomy of Existing Paradigms We contrast the above paradigms on two dimensions: 1) exploration throughput and 2) the ability to use human intuition. As shown in Figure 3.1(D), AutoML-based model selection explores configs with high throughput. But the only time it relies on human intuition is during the initial search space specification. Thus, it may inefficiently explore the config space and incur significant resource costs, which could have been avoided by a human intervention. The human-in-the-loop model selection primarily relies on human intuition but operates at very-low throughput levels due to the inherent limitations of human interactivity. Also, many DL configs are so long-running that false promises of "interactivity" become a prison for DL practitioners that wastes their time. Overall, we see a major gap between AutoML-based and human-in-the-loop model selection paradigms.

This Work To overcome the above-mentioned drawbacks, we propose a new paradigm we call *intermittent human-in-the-loop model selection*. It is a hybrid of both AutoML-based and interactive human-in-the-loop model selection. However, unlike the latter, human exploration

is not mandatory in our approach. As an analogy, the interactive exploration is akin to instant messaging (IM), whereas our paradigm is akin to email threads or Slack channels. Without interactivity, the former becomes not usable. But our approach is more flexible due to asynchronous, spread-out-over-time yet stateful exchanges that can still subsume full interactivity. We implement our paradigm in CEREBRO, a new platform for resource-efficient deep learning model selection [18]. We extend CEREBRO with a graphical user interface, a REST API, and change existing components to support our new paradigm. In this demonstration, we will allow the audience to use CEREBRO to perform intermittent human-in-the-loop model selection using 5 real-world DL model selection workloads. Our paradigm is an ideal fit for DL model selection workloads due to their long-running nature, and thus, we focus on DL for now. But it is readily applicable to any other ML model family too.

3.2 Technical Contributions

3.2.1 New Paradigm for Model Selection

Our intermittent human-in-the-loop paradigm breaks the false dichotomy of AutoML-based and interactive human-in-the-loop model selection. It is motivated by both observations about model selection practice [13] and our experience in training DL models for public health applications [14]. It starts similar to the AutoML-based paradigm where the user specifies the search space and picks a canned AutoML procedure like Grid, Random, or even a more advanced one like HyperOpt. However, instead of passively waiting by just consuming the results of explored configs, we enable the user to steer the model selection process. User can now *create* new individual configs or batch of configs using a refined search space, *stop* running configs, and *resume* stopped configs.

Creating new configs outside the control of the AutoML procedure enables the user to inject human intuition into the overall model selection process. New configs can also be created by first cloning an existing config along with its trained parameters and then by tweaking only

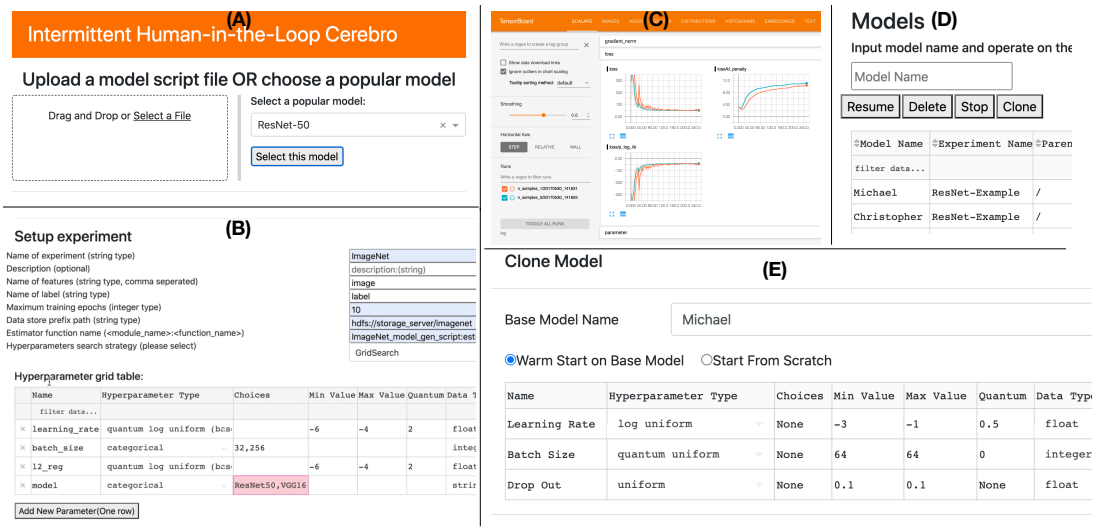

Figure 3.2. User interface for intermittent human-in-the-loop model selection. (A) UI to either pick a canned ML model (e.g., ResNet50) or upload a script file defining a custom model. (B) UI to specify experiment metadata, training data information, and config search space. (C) Visualizing the learning curves using embedded TensorBoard. (D) UI listing all configs and controls to add/stop/resume configs. (E) UI to create a drill-down model selection workload.

some of the hyper-parameters like learning rate or batch size. Users can use this feature to make the model training adaptable based on human intuition. They can also dynamically reprioritize the training of some configs over the others by using the stop and resume feature. Thus, as shown in Figure 3.1(D) our paradigm can seamlessly navigate the exploration throughput and human intuition usage tradeoff space based on the available user interaction level. In a sense our approach fulfils the desire for "dialogue with the algorithms" we have heard from many ML/DL practitioners, except neither party is forced to respond promptly.

3.2.2 UIs for Intermittent Specification

System UI provides graphical controls that enable the user to perform intermittent human-in-the-loop model selection. It is implemented using Python Dash visualization library and runs in a web browser which makes it portable. It is integrated with a backend REST API to perform the user-requested actions.

The user will start interacting by either picking a canned ML model from a roster or by

uploading a Python script defining a custom ML model using the UI shown in Figure 3.2(A). We currently support 4 popular DL models in our roster: ResNet50, MobileNet, BERT-base, and DistilBert. New models can be easily added to the roster. Also, the custom script option can support arbitrary Keras models. After picking a model, the user will be then prompted with the UI shown in Figure 3.2(B) to specify a name, description, AutoML search procedure, names of features and label columns, the path to the training data, and the maximum number of training epochs for any model. If a custom script is uploaded, the user is required to specify the entry point function name in that script. This entry point function should take a dictionary of config values as input and return a compiled Keras model as output. The user is also required to specify the search spaces for the available configs. The list of available configs is fixed for a canned model. For a custom model, it can be defined manually. After specifying these values, the user can launch the model selection workload. The user can visualize model training and validation metrics, such as loss and accuracy, through an embedded TensorBoard UI as shown in Figure 3.2(C). User can also add/stop/resume configs using the controls shown in Figure 3.2(D) or create a new drill-down workload on a refined search space using the UI shown in Figure 3.2(E).

3.2.3 Decoupled System Architecture

Our paradigm translates to two key system design decisions: (1) *decoupling* the specification of what configs to explore from scheduling their training and (2) being able to *multiplex* the training of many configs on the fly on the same cluster. Otherwise, it is simply not possible to run multiple model selection workloads at the same time or even increase the model selection throughput of a single workload without provisioning more resources. While resource provisioning has become easy with cloud computing, cloud users also often need to limit their resource usages due to cost concerns. For others like domain science users, it may be simply not possible to provision more resources such as in fixed-sized campus clusters.

We implement our paradigm in CEREBRO. CEREBRO uses a novel parallel execution

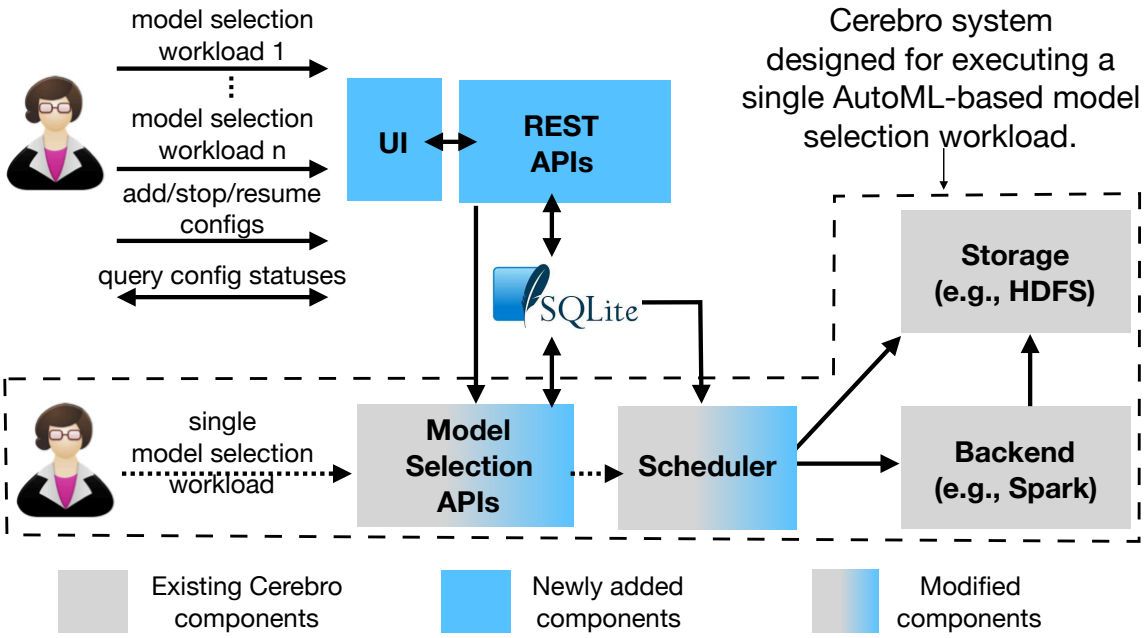

Figure 3.3. High-level system architecture diagram of CEREBRO along with the changes and additions to support our intermittent human-in-the-loop model selection paradigm.

strategy for stochastic gradient descent (SGD)-based training (e.g., like in DL) called *model hopper parallelism* (MOP) that ensures SGD properties. SGD reads training data *sequentially* and repeats it for several iterations. A single iteration is also called an *epoch* of training. MOP breaks a single epoch of training on partitioned data into multiple sub-units called *sub-epochs*; a sub-epoch operates on a single data partition. Given a set of configs, MOP schedules them using an *epoch-level* scheduling template where all configs are trained for the current epoch before training the next epoch for any config. Also, it multiplexes the training of the configs by asynchronously scheduling sub-epochs on workers. The scheduler ensures that both a config is trained sequentially and on all partitions. Overall, MOP significantly increases the model selection throughput without provisioning more resources. Originally, CEREBRO was designed to execute a single AutoML-based model selection workload at a time. Figure 3.3 presents CEREBRO system architecture. More details about the CEREBRO system can be found in our VLDB 2020 paper [18].

We leverage the epoch-level scheduling template of CEREBRO to support our new

paradigm. We also add a new graphical user interface (UI), a REST API and update CEREBRO's model selection APIs and scheduler to achieve our requirements. UI sends user requests to the model selection APIs through the REST API. We changed the model selection APIs such that they now write the configs to an SQLite database instead of directly interacting with the scheduler. User-created configs are also directly added to this database. The scheduler will read all the configs to be trained from this database and train them for one epoch. After completing training for one epoch it will update the training metrics of the config in the database. And this process will continue. Whenever the user wants to stop (resp. resume) a config, it will be marked as such in the database and will be ignored (resp. considered back) by the scheduler. Figure 3.3 presents the modified CEREBRO system architecture.

3.3 Related Work

While a few commercial software products including Sagemaker Autopilot, Azure Automated ML, and Determined AI have attempted to streamline AutoML-based model selection, unlike our paradigm, none of these products enable the user to steer a model selection process while it is running. Several other works [13, 31, 15] have also emphasized the importance of more human control in AutoML-based model selection. However, to the best of our knowledge, ours is the first system prototype that enables the user to intervene and steer the model selection process on par with the meta-heuristic while it is running. We also provide details of a system architecture to realize this new paradigm. Ideas similar to our stop and resume-based model training reprioritization approach have been explored in relational query processing settings [19, 29]. Our work was inspired in part by such ideas but ours is the first to apply them in the context of ML model selection workloads, with the main novelty here being our focus on iterative training procedures such as SGD and intervention by observing evolving learning curves.

Chapter 3 contains material from "Intermittent Human-in-the-Loop Model Selection Using Cerebro: A Demonstration", which appears in Proceedings of VLDB Endowment, Volume

14, Issue 12, Pages 2687-2690. Li, Liangde; Nakandala, Supun; Kumar, Arun. The thesis author was the co-author of this paper and contributed to the design and implementation of the system.

Chapter 4

Conclusion and Future Work

In this thesis, we complete two pieces of work for efficient systems for advanced data analytics. We take the first step to fill the gap of lacking a comparative evaluation of genetic analysis systems. We show some interesting findings from our experiment suite. In the new paradigm of model selection, we demonstrate its improvements in terms of throughput and using human intuition.

4.1 Future Work Related to Benchmark of Genetic Analysis Tools

The first piece of work to be done is to finish all distributed experiments of the remaining three algorithms, LD-Prune, PCA, and OLS. Also, experiments related to the Data Scale Factors (Axis 2) and the Task Hyperparameter Factors (Axis 5) are left to be done. Finally, the auto-tuning of Dask configurations and improving fault tolerance of Dask Distributed are two topics worth putting more effort into because they may enable Sgkit to execute more powerful operations on data reliably.

4.2 Future Work Related to Intermittent Human-in-the-Loop Model Selection

First, much work is needed to formalize this new paradigm and develop new AutoML procedures that explicitly take advantage of human input. Almost all AutoML procedures today are developed without any human interactivity in mind. Some even make assumptions that prohibit human interaction [17]. Second, it would be appealing to add elastic scaling and cloud-native scheduling support to CEREBROto reduce runtimes subject to monetary constraints. Finally, CEREBRO's decoupled architecture can be generalized to support multi-tenancy to run concurrent model selection workloads on the same infrastructure.

Bibliography

[1] Tom Augspurger. datasframe – modern pandas (part 2): Method chaining. https://tomaugspurger.github.io/method-chaining.html, April 2016. (Accessed on 05/29/2022).

[2] James Bergstra, Dan Yamins, and David D Cox. HyperOpt: A Python Library for Optimizing the Hyperparameters of Machine Learning Algorithms. In *Proceedings of the 12th Python in Science Conference*, volume 13, page 20. Citeseer, 2013.

[3] Xavier Bouthillier and Gaël Varoquaux. *Survey of Machine-Learning Experimental Methods at NeurIPS2019 and ICLR2020*. PhD thesis, Inria Saclay Ile de France, 2020.

[4] Chengliang Chai and Guoliang Li. Human-in-the-loop Techniques in Machine Learning. *Data Engineering*, page 37, 2020.

[5] Francis S Collins, Michael Morgan, and Aristides Patrinos. The human genome project: lessons from large-scale biology. *Science*, 300(5617):286–290, 2003.

[6] Sgkit developers. Sgkit: Getting started. https://pystatgen.github.io/sgkit/latest/getting_started.html, 2020. (Accessed on 05/26/2022).

[7] Sgkit developers. sgkit: Statistical genetics toolkit in python. https://pystatgen.github.io/sgkit/latest/index.html, 2020. (Accessed on 06/06/2022).

[8] Godfrey H Hardy. Mendelian proportions in a mixed population. *Science*, 28(706):49–50, 1908.

[9] WG Hill and Alan Robertson. Linkage disequilibrium in finite populations. *Theoretical and applied genetics*, 38(6):226–231, 1968.

[10] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population based training of neural networks. *CoRR*, abs/1711.09846, 2017.

[11] Jerome Kelleher and Konrad Lohse. Coalescent simulation with msprime. In *Statistical*

Population Genomics, pages 191–230. Humana, New York, NY, 2020.

[12] Ines Krissaane, Carlos De Niz, Alba Gutiérrez-Sacristán, Gabor Korodi, Nneka Ede, Ranjay Kumar, Jessica Lyons, Arjun Manrai, Chirag Patel, Isaac Kohane, and Paul. Avillach. Scalability and cost-effectiveness analysis of whole genome-wide association studies on google cloud platform and amazon web services. *Journal of the American Medical Informatics Association*, 27(9):1425–1430, 2020.

[13] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M Patel. Model Selection Management Systems: The Next Frontier of Advanced Analytics. *ACM SIGMOD Record*, 44(4):17–22, 2016.

[14] Arun Kumar, Supun Nakandala, Yuhao Zhang, Side Li, Advitya Gemawat, and Kabir Nagrecha. Cerebro: A Layered Data Platform for Scalable Deep Learning. In *CIDR*, 2021.

[15] Doris Jung-Lin Lee and Stephen Macke. A Human-in-the-loop Perspective on AutoML: Milestones and the Road Ahead. *IEEE Data Eng. Bull.*, 42(2):59–70, 2019.

[16] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Ben Recht, and Ameet Talwalkar. Massively Parallel Hyperparameter Tuning. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020*. mlsys.org, 2020.

[17] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A Novel Bandit-based Approach to Hyperparameter Optimization. *JMLR*, 18(1):6765–6816, 2017.

[18] Supun Nakandala, Yuhao Zhang, and Arun Kumar. Cerebro: A Data System for Optimized Deep Learning Model Selection. *Proc. VLDB Endow.*, 13(12):2159–2173, July 2020.

[19] Sivaramakrishnan Narayanan and Florian Waas. Dynamic Prioritization of Database Queries. In *ICDE*, pages 1232–1241. IEEE, 2011.

[20] Physeo. Usmle step 1 linkage disequilibrium - youtube. https://www.youtube.com/watch?v=DvrAuMyu4wU, Jan 2019. (Accessed on 06/03/2022).

[21] Shaun Purcell, Benjamin Neale, Kathe Todd-Brown, Lori Thomas, Manuel AR Ferreira, David Bender, Julian Maller, Pamela Sklar, Paul IW De Bakker, Mark J Daly, and Pak C. Shamd. Plink: a tool set for whole-genome association and population-based linkage analyses. *The American journal of human genetics*, 81(3):559–575, 2007.

[22] Robert Ricci, Eric Eide, and CloudLab Team. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[23] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, volume 130, page 136. Citeseer, 2015.

[24] samtools. Bcftools by samtools. https://samtools.github.io/bcftools/, Feb 2022. (Accessed on 06/06/2022).

[25] Hail Team. Hail. https://github.com/hail-is/hail/commit/edeb70bc789c, 2021.

[26] Anthony Thomas and Arun Kumar. A comparative evaluation of systems for scalable linear algebra-based analytics. *Proceedings of the VLDB Endowment*, 11(13):2168–2182, 2018.

[27] Wilhelm Weinberg. On the demonstration of heredity in man. *(1963) Papers on human genetics*, 1908.

[28] Janis E Wigginton, David J Cutler, and Gonçalo R Abecasis. A note on exact tests of hardy-weinberg equilibrium. *The American Journal of Human Genetics*, 76(5):887–893, 2005.

[29] Johannes Wust, Martin Grund, and Hasso Plattner. Dynamic Query Prioritization for In-memory Databases. In *In Memory Data Management and Analysis*, pages 56–68. Springer, 2013.

[30] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya Parameswaran. Helix: Holistic Optimization for Accelerating Iterative Machine Learning. *Proc. VLDB Endow.*, 12(4):446–460, 2018.

[31] Ce Zhang, Arun Kumar, and Christopher Ré. Materialization Optimizations for Feature Selection Workloads. *ACM Transactions on Database Systems (TODS)*, 41(1):1–32, 2016.