Accelerating Data-Intensive Applications Using MapReduce

Hwee Yong Ong

MSc in Computer Science
School of Informatics
The University of Edinburgh
2010
Abstract

Information explosion propelled by the exponential growth in digitised data is an unstoppable reality. To be able to extract relevant and useful knowledge from this voluminous data in order to make well-informed decision is a competitive advantage in the information age. However, the attempts to transform raw data into valuable knowledge face both data and computational intensive challenges. As a result, parallel and distributed computing is often strongly sought after to alleviate these challenges. While there are many distributed computing technologies being proposed over the years, MapReduce has gained an immense amount of interest in recent years due to its simplicity and superb scalability at low cost.

The challenges brought along by information explosion are real and the potential benefits offered by MapReduce are alluring. With these as the backdrop, this project studies the feasibility, applicability and performance of employing MapReduce model for a distributed data mining scenario based on a use case found in the Life Sciences domain. Deriving from a conceptual serial data mining process involving several data mining algorithms, a prototype data mining system capable of distributed automated image annotation is developed. Within the prototype, the data mining algorithms are adapted to the MapReduce model to perform distributed data mining tasks with high performance data-parallelism. The successful implementation demonstrates that the data mining algorithms can be painlessly adapted to the MapReduce model with Hadoop API.

In addition, experiments with real data are conducted on a Cloud computing platform to evaluate the distributed computation performance of the prototype in terms of speedup, scaleup and sizeup. The evaluation results show that the data mining algorithms can be adapted to MapReduce model with promising speedup, good scaleup and superb sizeup. Interestingly, the experimental process also reveals that Cloud computing could be an efficient and cost-effective approach in meeting the needs of research-based experiments.
Acknowledgement

I would like to express my special thanks of gratitude to all who has helped and contributed to the completion of this dissertation, in particular my deepest appreciation to my project supervisor, Dr. Liangxiu Han, for her tireless dedication in providing valuable advice and guidance. Last but not least, I thank my family for their support in everything I do.
Declaration

I declare that this dissertation is composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Hwee Yong Ong
## Table of Contents

1. **Introduction** .................................................................................................................. 1

2. **Background** .................................................................................................................. 3
   2.1 Data-Intensive Applications and Knowledge Discovery ................................................. 3
   2.2 Data Mining ................................................................................................................... 4
       2.2.1 Data Mining Process ............................................................................................... 4
       2.2.2 Data Mining Methods ............................................................................................. 5
   2.3 Parallel and Distributed Computing ................................................................................ 8
       2.3.1 POSIX Threads (Pthreads) ..................................................................................... 9
       2.3.2 Open Multi-Processing (OpenMP) ......................................................................... 9
       2.3.3 Compute Unified Device Architecture (CUDA) ....................................................... 10
       2.3.4 Message Passing Interface (MPI) ........................................................................... 11
       2.3.5 Unified Parallel C (UPC) ....................................................................................... 11
       2.3.6 Intel Threading Building Blocks (TBB) ................................................................... 12
       2.3.7 Fortress .................................................................................................................. 13
       2.3.8 MapReduce .......................................................................................................... 14
       2.3.9 Dryad ..................................................................................................................... 15
   2.4 EURExpress-II Case Study .............................................................................................. 16

3. **Methodology** .................................................................................................................. 17
   3.1 Data Mining Process ....................................................................................................... 17
   3.2 Data Mining Algorithms ................................................................................................ 19
       3.2.1 Median Filter .......................................................................................................... 19
       3.2.2 Wavelet Transformation ....................................................................................... 20
       3.2.3 Fisher’s Ratio ........................................................................................................ 27
       3.2.4 K-Nearest Neighbour (KNN) ............................................................................... 29
   3.3 MapReduce and Hadoop ................................................................................................ 30
       3.3.1 Hadoop Distributed File System .......................................................................... 33
       3.3.2 Hadoop MapReduce ............................................................................................. 35
       3.3.3 Other Hadoop Offerings ....................................................................................... 42

4. **Implementation** .............................................................................................................. 43
4.1.1 MRController ................................................................. 44
4.1.2 Data Packer ................................................................. 44
4.1.3 Image Pre-processing ..................................................... 45
4.1.4 Feature Generation with Wavelet Transformation ............... 46
4.1.5 Testing and Training Set Extraction for 10-Fold Cross Validations ... 48
4.1.6 Feature Selection with Fisher’s Ratio ................................ 49
4.1.7 Classification with K-Nearest Neighbour ........................... 51

5 Experiment and Evaluation .................................................. 55
5.1 Experiment Overview ....................................................... 55
5.1.1 Setup and Configuration ................................................ 55
5.1.2 Experimental Results .................................................... 56
5.2 Evaluations ..................................................................... 57
5.2.1 Speedup ....................................................................... 57
5.2.2 Scaleup ....................................................................... 61
5.2.3 Sizeup ....................................................................... 62
5.3 Cost Effectiveness of Cloud Computing ............................... 63

6 Conclusion and Future Work .................................................. 65

Bibliography ........................................................................ 67

Figure 1: CRISP-DM Process Model .................................... 5
Figure 2: Overview of Data Mining Scenario for Automatic Gene Expression Annotation ..................................................... 17
Figure 3: Example of a 3x3 Median Filter ................................. 20
Figure 4: Comparison of Time-Frequency Resolution between Fourier Transformation and Wavelet Transformation .................. 21
Figure 5: Example of Wavelets from Different Wavelet Families ... 22
Figure 6: Discrete Wavelet Transformation using Filter Banks ....... 24
Figure 7: Lifting Scheme for Discrete Wavelet Transformation ........ 24
Figure 8: Daubechies-4 Discrete Wavelet Transformation with Lifting Scheme ...... 25
Figure 9: 2-D Discrete Wavelet Transformation with Daubechies-4 Wavelets ...... 27
Figure 10: Illustration of Class Separation with Mean and Fisher's Ratio........ 28
Figure 11: Illustration of K-Nearest Neighbours Algorithms ............. 30
1 Introduction

Information explosion is a reality. In fact, the rate and dimensions of information growth will be further accelerated with increasing digitization and ubiquitous computing in all facets of our lives. On the other hand, the broad availability of data coupled with increased computing capability and decreased computing and storage cost has also created vast research and commercial opportunities in data intensive computing. Data mining, a classic topic in data intensive computing, exhibits great potential in addressing the many challenges posed by information explosion. Data mining is the process of extracting patterns from enormous data, and transforming the data into knowledge. However, the non-trivial process often requires substantial amount of computing resources and storage spaces. Hence, for both performance and scalability, distributed computing is strongly sought after for practical implementations.

Over the years, many distributed and parallel computing technologies and frameworks were proposed [1][2][3]. In particular, MapReduce has gained immense amount of interest in recent years and has rapidly become a popular platform for distributed computing due to its simplicity and scalability at low cost. MapReduce was brought to limelight by Google’s successful implementation, claiming thousands of programs being implemented and executed on thousands of distributed nodes, processing more than 20 PB of data on a daily basis [2].

The need in data-intensive computing is real and the potential benefits offered by MapReduce are alluring. This project is initiated to study the challenges of developing a scalable data-intensive application with minimum effort and cost. Through investigation of the feasibility, applicability and performance of employing the MapReduce model for a distributed data mining scenario, this project aims to achieve the following objectives:

- To understand data-parallel processing using the MapReduce model for addressing performance issues in data intensive applications;
- To investigate how to adapt algorithms used in data mining process to the MapReduce model;
To develop a prototype and conduct performance evaluation to facilitate comparisons with other frameworks that support data intensive applications. The data mining scenario to be studied is based on a use case found in the field of Life Sciences, in which an automated annotation of gene expression for mouse embryo images is proposed.

Among others, this project makes the following contributions:

- Adaptation of data mining algorithms with MapReduce model;
- Implementation of a prototype system capable of distributed data mining with MapReduce model;
- Deployment of the prototype on a Cloud computing platform and performance evaluation based on standard performance metrics, namely speedup, scaleup and sizeup;

The remaining of this dissertation is organized as follows: Section 2 provides a review on works related to the project; Section 3 describes the methods and techniques; Implementation details are presented in Section 4; Experiments and evaluations are discussed in Section 5; Section 6 concludes the dissertation and discusses possible future works.
2 Background

This section presents the background information for a better understanding of the topics involved in this project.

2.1 Data-Intensive Applications and Knowledge Discovery

Cannataro et al. provided a concise definition of data-intensive applications: “Applications that explore, query, analyse, visualise, and, in general, process very large scale data sets are known as Data Intensive Applications”[4].

In this information age, digital data in the scale of hundreds of petabyte are being generated everyday. This includes electronic transactions, publications and media managed by government establishments and commercial organisations, electronic records from numerous sensors, social contents created by casual internet users, results from huge scientific experiments such as the Large Hadron Collider (LHC) at CERN, and many more. What we are experiencing now is probably just the tip of the iceberg. It was estimated that the amount of digital information created in the year of 2011 would be approximately 1,800 exabytes, expanding by a factor of 10 every five years [5]. At such an astonishing information growth rate, it is not surprising that many organisations recognised the critical needs for data-intensive applications to process and analyse these ever-growing data in order to gain competitive advantages. In many cases, the question is not on when to build these data-intensive applications, but what are the requirements and how to build and deploy them immediately [6].

One of the key aspects in data-intensive applications is to discover useful knowledge that is hidden within the voluminous data. This knowledge would, in turn, enable organisations to make timely decisions with a greater degree of confidence. In the academic world, this is a subject well-studied in the field of Knowledge Discovery in Databases (KDD). Fundamentally, the field of KDD is the study of methods and techniques for making sense of data. From a more encompassing perspective, it is an interdisciplinary field since it also deals with issues in other related fields such as machine learning, pattern recognition, statistical analysis, database management, expert systems, data visualisation, and high-performance computing. Over the years,
researchers in the KDD field have devoted vast amount of effort in developing theories and tools to address the challenges in mapping massive low-level data into alternative representations that are more compact and useful. Many of these works are highly related to techniques in machine learning, pattern recognition and statistics.

The entire KDD process is complex and usually non-trivial. At the core of most KDD processes is the application of data-mining methods for pattern discovery and extraction. The other steps in the KDD process such as pre-processing and transformation of data, incorporation of appropriate prior knowledge, validations and interpretation of results are all essential to ensure useful knowledge is derived from the data [7].

Since its inception, KDD has found many successful applications in science (such as astronomy and bioinformatics) and business (such as marketing, financial investment, fraud detection, manufacturing, telecommunications, and internet agents) [7].

2.2 Data Mining

While KDD is commonly used to refer to the overall process of discovering useful knowledge from data, data mining refers to the particular step in this process which involves the application of specific algorithms for extracting patterns from data. Data mining is defined as “the nontrivial extraction of implicit, previously unknown, and potentially useful information from data” [8]. From a layman perspective, it is the automated or, more usually, semi-automated process of extracting patterns from enormous data, and transforming the data into useful knowledge.

2.2.1 Data Mining Process

To understand data mining, we shall start with a brief discussion on a commonly used data mining process model - Cross Industry Standard Process Model for Data Mining (CRISP-DM) [9]. CRISP-DM is a leading methodology used for describing the approaches that are commonly used in data mining. It describes a data mining process with six major phases, namely: Business Understanding, Data Understanding,
Data Preparation, Modeling, Evaluation and Deployment. Figure 1 shows the overview of CRISP-DM process model.

![CRISP-DM Process Model](source: http://www.crisp-dm.org/Process/index.htm)

Figure 1: CRISP-DM Process Model

Note that the flow of the phases in CRISP-DM process model is not strictly sequential; but often requires moving back and forth between different phases depending on the outcome of each phase, or what needs to be performed next. The arrows indicate the most important and frequent dependencies between these phases. The outer circle symbolizes the cyclic nature of data mining process in which it continues even after a solution has been deployed. The lessons learned during the process can trigger new, often more focused business questions on the underlying data. At the same time, subsequent data mining processes will benefit from the experiences of the previous ones.

### 2.2.2 Data Mining Methods

While all phases described in the CRISP-DM process model are equally important, the actual data intensive computation usually take places within the modeling phase. The modeling phase is also where a lot of data mining methods and algorithms are
applied to extract knowledge from the data. Generally, the data mining methodologies involve models fitting (or patterns recognition) based on the underlying data. The algorithms used in the models fitting are usually related to those developed in the field of pattern recognition and machine learning. The model fitting process is often complex and the fitted models are the keys to the inferred knowledge. However, the correctness of these models (as in whether they reflect useful knowledge) will often require human judgment during the evaluation phase.

Due to its potential benefits, the field of data mining has attracted immense interests from both academics and commercial tool vendors. As a result, many different data mining methods and algorithms were developed by various parties at different time. While it is not difficult to find literatures that had attempted to categorise these data mining methods [10][11][7], we will only briefly discuss some of the more commonly used data mining methods as outlined below:

**Classification**: Given a set of predefined categorical classes, the task attempts to determine which of these classes a specific data item belongs. Some commonly used algorithms in classification are Support Vector Machines (SVM), C4.5, K-Nearest Neighbours (KNN) and Naïve Bayes.

**Clustering**: Given a set of data items, the task attempts to partition the data into a set of classes such that items with similar characteristics are grouped together. Some of the most popular algorithms used in this method are K-Means clustering, Expectation-Maximization (EM) and Fuzzy C-Means algorithms.

**Regression**: Given a set of data items, the task analyses the relationship between an attribute value (the dependent variable) and the values of other attributes (the independent variables) in the same data item, and attempts to produce a model that can predict the attribute value for new data instances. When a regression model is one in which the attribute values are dependent on time, it is sometime referred to as ‘time-series forecasting’.
Vector Machines (SVM), Generalized Linear Model (GLM), Radial Basis Function (RBF) and Multivariate Adaptive Regression Splines (MARS) are some algorithms frequently used in regression model fitting.

**Associations:** Given a set of data items, the task attempts to identify relationships between attributes within the same data item or relationships between different data items. When the time dimension is considered for the relationships between data items, it is often referred to as ‘sequential pattern analysis’. The Apriori algorithm is a popular technique used in learning association rules.

**Visualization:** Given a set of data items, the task attempts to present the knowledge discovered from the data in a graphical form that is more understandable and interpretable by humans.

**Summarization:** Given a set of data items, the task attempts to find a compact description for a subset of the data items. This could either be in the form of numerical or textual summary.

**Deviation Detection:** Given a set of data items, the task attempts to discover the most significant changes in the data from previously observed values. In the centre of this method is the identification of “outliers”, many algorithms can be applied to perform this task. Support Vector Machines (SVM), regression analysis and Neural Networks are all possible approaches used in this method.

While there existed vast number of data mining methods and algorithms, a salient point is that each technique typically suits some problems better than others. The key in successful data mining lies in understanding the underlying data and asking the right question. Ultimately, the aim is to achieve an adequate level of “interestingness” in the discovered knowledge [12].
It is crucial to understand that not all patterns discovered by the data mining algorithms are necessarily valid. In order to validate that the discovered patterns existed in the general data set, a common technique called k-fold cross-validation [13] is often used for the evaluation. The technique involves splitting the available data into testing set and training set. The training set is used for training the data mining algorithms to acquire the patterns. The testing set is used to validate the learnt patterns and to provide a measurement to the desired accuracy. Further tuning of the process and algorithms are performed if the learnt patterns do not meet the desired accuracy.

2.3 Parallel and Distributed Computing

The non-trivial process of extracting knowledge from enormous data often requires substantial amount of computing resources and storage spaces. For both performance and scalability considerations, distributed and/or parallel computing is strongly sought after for practical implementations [14][15]. A list of literatures on distributed data mining can be found in [16].

The terms “parallel computing” and “distributed computing” share many similarities and the usage of these terms are often confusing. The fundamental principle is to decompose or divide a large problem into smaller ones, which are then solved concurrently. Hence, parallel/distributed computing is predominately concern about the partitioning of workload into tasks and the mapping of these tasks into workers. In general, we could broadly discuss parallel/distributed computing technology in the context of its computation model (task parallelism versus data parallelism) and communication model (message passing versus shared-memory) [17]. In task parallelism, the computation tasks are divided into distinct sub-tasks (i.e. different computation programs) which are then assigned to different computing units. In data parallelism, all computing units run the same program but operate on different data streams. In message passing model, the communications between different tasks (or computing units) are based on sending/receiving messages; while in shared-memory model, they communicate via reading/writing to a shared memory. This project examines the performance impact of a data parallelism approach.
To contrast and compare the different methods that are available, the following subsections give a brief overview on some common parallel/distributed technologies used in both scientific and commercial applications.

### 2.3.1 POSIX Threads (Pthreads)

A thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system [18]. Pthreads is a POSIX standard for threads and its implementations are available in many Unix-like POSIX systems (e.g. FreeBSD, GNU/Linux, Mac OS X and Solaris) and Microsoft Windows. Pthreads is essentially a set of C programming language types and procedure calls [19], implemented with a pthread.h header/include file and a thread library for creating and manipulating threads. Thread management in Pthreads requires programmer to explicitly create and destroy threads by making use of pthread_create and pthread_exit function. The workload partitioning and task mapping are explicitly specified by programmers as arguments to the pthread_create function. In addition, the programmers will have to take care of the data race and deadlock issues arising from multiple worker threads accessing a piece of shared data. To ease some of these programming difficulties, Pthreads provides the implementations of mutex (mutual exclusion) variables and semaphore.

### 2.3.2 Open Multi-Processing (OpenMP)

OpenMP is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran on Unix and Microsoft Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior [20]. Similar to Pthread, the unit of workers in OpenMP is threads. The worker management is implicitly performed by the programmers through the use of special compiler directives to specify code regions that are to be run in parallel. The number of threads to be created is specified via an environment variable. Hence, the programmers do not need to manage the lifetime of the executing threads. The workload partitioning and task-to-worker mapping are simplified through the use of
built-in loop scheduling schemes (static, dynamic, runtime, and guided). Programmers are relieved from the data sharing and synchronization issues among threads through pre-defined constructs known as data-sharing attribute clauses and synchronization clauses.

2.3.3 Compute Unified Device Architecture (CUDA)

CUDA is a general purpose parallel computing architecture (with a new parallel programming model and instruction set architecture) that leverages the parallel compute engine in NVIDIA GPUs (Graphics Processing Unit) to solve many complex computational problems in a more efficient way than on a CPU[21]. It is an extension of C programming language designed to support parallel processing on NVIDIA GPU. Although, CUDA comes with a software environment that allows developers to use C as a high-level programming language, other languages or application programming interfaces are supported through its peer technologies such as CUDA FORTRAN, OpenCL, and DirectCompute [22]. CUDA only works with NVIDIA GPUs from the G8X series onwards, including GeForce, Quadro and the Tesla line.

In CUDA, the parallel computing system is modelled as a construction consisting of a host device (i.e. CPU) and one or more computation resources (i.e. GPU). The computation of tasks (known as “kernel” in CUDA) are distributed and performed in parallel among a set of threads in the GPU. Unlike conventional threads in CPU, the GPU architecture for threads has a two-level hierarchy: block and grid. A block is a set of tightly coupled threads, each identified by a thread ID and executing on a separate Arithmetic Logic Unit (ALU) within the GPU. Threads within each block have access to a sharable memory. A grid is a set of loosely coupled of blocks with similar size and dimension. Like OpenMP, programmers do not need to manage the lifetime of threads in CUDA. The programmers just need to specify the dimension of the grid and block required for a computation task. Note that the dimensions of grid and block defined the workload partitioning and worker mapping in CUDA. Synchronization among threads of the same block in CUDA is supported through the syncthreads() function.
2.3.4 Message Passing Interface (MPI)

MPI is a language-independent message-passing library interface specification [23]. It is often seen as the de-facto standard for developing parallel applications on distributed memory architecture. Its implementations are available as a set of routines (or API) in C, C++, and Fortran. Among the many implementations, the most common ones are MPICH [24], LAM/MPI [25] and OpenMPI [26].

Unlike parallel computing with multi-threading, MPI defines each worker as a process. The programmer does not need to manage the creation, scheduling, or destruction of processes. Through the use of a command-line tool – “mpirun”, the programmer instructs the MPI runtime infrastructure the number of required processes and the mapping of processes to processors. However, the workload partitioning and task mapping have to be explicitly specified by the programmers via assignment of computing tasks to the processes. The key aspect of MPI is such that the communication among processes adopts the message passing paradigm where data sharing is through sending/receiving messages between processes. Two classes of communication operations are defined: point-to-point and collective. Point-to-point operations (e.g. MPI_Send and MPI_Recv) are meant to facilitate communications between processes, whereas collective operations (such as MPI_Bcast) are for communications involving more than two processes. Synchronization among multiple workers is performed through the MPI_Barrier operation, where individual process executions are blocked until all processes have entered the barrier call.

2.3.5 Unified Parallel C (UPC)

UPC is an explicit parallel extension of ANSI C and is based on the partitioned global address space programming model, also known as the distributed shared memory programming model. UPC keeps the powerful concepts and features of C and adds parallelism; global memory access with an understanding of what is remote and what is local; and the ability to read and write remote memory with simple statements [27]. Many UPC implementations are available, including offerings from
commercial vendors like HP and IBM as well as academic institutions such as Berkeley UPC Compiler.

In UPC the distributed system is viewed as a collection of threads operating in a partitioned global address space that is logically distributed among threads. Each thread has affinity with a portion of the globally shared address space. Each thread has two types of memory accesses: to its own private address space or to other threads’ address space. The thread affinity is used to improve the performance of memory accesses by optimizing memory-access between a thread and the per-thread address space where the thread has been bound to. The programmer does not have to manage the lifetime of worker threads, but simply specify the number of threads required through the command-line tool, “upcrun”. Workload partitioning and worker mapping can be either implicit or explicit. Implicit workload partitioning and task mapping are supported through an API called “upc forall” which is similar to for loop in C programming, except that the content of the iteration will be run in parallel. The explicit workload partitioning and worker mapping is similar to MPI, where the programmers have to specify the data for each computing threads. UPC uses the Partitioned Global Address Space (PGAS) paradigm to facilitate communication among threads. The access to the PGAS is performed with three types of pointers: (i) private pointer where the private pointers point to their own private address space, (ii) private pointer-to-share where the private pointers point to the shared address space, and (iii) shared pointer-to-share where the shared pointers from one address space point to the other shared address space.

UPC provides both implicit and explicit worker synchronization via these programming constructs: Barrier (for blocking synchronization similar to the one in MPI), Split phase barrier (for non-blocking synchronization), Fence (for shared data reference management), Locks (for shared data protection) and Memory consistency control (“Relaxed” or “Strict” for controlling access to shared data).

2.3.6 Intel Threading Building Blocks (TBB)
TBB is a C++ template library developed by Intel Corporation for writing software programs that take advantage of multi-core processors. It abstracts the low-level threading details of creating and managing threads. Applications written with TBB tend to be more portable since parallelism is achieved through library calls and utilization of a task manager for scheduling. The task manager analyses the system the software is running on, chooses the optimal number of threads, and performs load balancing that spreads out the work evenly across all processor cores. The library consists of data structures and algorithms that simplify parallel programming in C++ by avoiding requiring a programmer to use native threading packages such as POSIX threads or Windows threads, or even the portable Boost Threads[28][29].

The worker is computing threads created through the built-in parallelism algorithm and data structure templates such as parallel_for, parallel_reduce, parallel_scan, parallel_while, parallel_do, pipeline, parallel_sort, concurrent_queue, concurrent_vector and concurrent_hash_map. In addition, it supports scalable memory allocation and provides the necessary workers synchronization mechanisms through mutual exclusion (such as mutex, spin_mutex, queuing_mutex, recursive_mutex) and atomic operations (such as fetch_and_add, fetch_and_increment, fetch_and_decrement, compare_and_swap, fetch and_store).

TBB implements task stealing to dynamically balance parallel workload across available processor cores to increase the core utilization. A TBB program creates, synchronizes and destroys graphs of dependent tasks according to the built-in algorithms. Tasks are then executed respecting graph dependences. Unlike many other parallel programming technologies, TBB aims to decouple the programming from the particulars of the underlying machine.

2.3.7 Fortress

Fortress is a specification for a general-purpose, statically typed, component-based programming language designed for producing robust high-performance software with high programmability. It is intended to be a “growable language”, i.e., a language that can be gracefully extended and applied in new and unanticipated
contexts. It supports state-of-the-art compiler optimization techniques, scaling to unprecedented levels of parallelism and of addressable memory [30].

Like most of its counterparts, the unit of worker in Fortress is threads. Worker management, workload partitioning and worker mapping in Fortress can be implicit or explicit. In Fortress, the iterative for loops is parallel by default (i.e. implicit). For explicit thread management, the “spawn” keyword is used for the creation of a thread and the “stop()” function for stopping a thread execution. The workload partitioning and worker mapping for explicitly spawned threads are similar to CUDA: the programmer needs to specify how a workload is partitioned into tasks and how the tasks are assigned to threads. For synchronization, two constructs are used: reductions and atomic expression. The intent of reduction is to avoid the need for synchronization by performing a local computation as far as possible. The atomic expression is used to control the single atomic data reads and writes among the parallel execution threads.

2.3.8 MapReduce

MapReduce is a programming model for processing and generating large datasets [2]. It is inspired by functional languages and targeted for data-intensive computations. In MapReduce, the format of the input is application-specific, the output is a set of <key,value> pairs. The programmer expresses the desired computation as two primitive functions: Map and Reduce. The Map function is applied on the input data and produces a list of intermediate <key,value> pairs. The Reduce function is then applied to the list of intermediate values that has the same key. It typically performs some form of convergence or merging operation, and produces zero or more output <key,value> pairs.

A key benefit in MapReduce is that the programmer does not need to deal with the complicated code parallelism, and focuses on the required computation. The MapReduce runtime is responsible for parallelization and concurrency control. The runtime splits the input automatically into parts that are processed concurrently on multiple nodes. The intermediate <key,value> pairs with the same key are then
grouped together, and partitions of these intermediate pairs are created based on some hashing function on the keys. The partitions are processed in parallel by the Reduce tasks running on multiple nodes. In both Map and Reduce stages, the runtime must dynamically decide the size of the data partitions, the number of computing nodes, the assignment of data partitions to nodes, and the allocation of memory buffer space. These decisions (including the workload partitions) can either be implicit (automatically decided by the runtime based on some default settings) or explicitly specified by the programmer via APIs or configuration files.

### 2.3.9 Dryad

Dryad [31] is a general-purpose distributed execution engine for coarse-grain data-parallel applications [3]. It is developed by Microsoft and designed to scale from powerful multi-core single computers, through small clusters of computers, to data centers with thousands of computers.

In Dryad, the computation in the distributed system is modeled as a directed graph: programs are graph vertices, and the communication channels between programs are graph edges. In this structure, a Dryad job scheduler is seemed as a graph generator which can synthesize any directed acyclic graph. The job graphs can even change during execution, in response to important events in the computation. Upon execution, the Dryad runtime runs the distributed application by executing the vertices (programs) of this graph on a set of available computers, communicating as appropriate through flies, TCP pipes, and shared-memory FIFOs.

The programmers just need to write simple and sequential programs and connects them using one-way channels. These steps are performed without knowledge of the complexity in thread management and synchronization. During execution, the Dryad runtime discovers the size and placement of data at run time, and modify the graph as the computation progresses to make efficient use of the available resources. In addition, it handles the recovering from communication or computer failures, and transporting data between computing nodes.
To make development of Dryad applications easier, Microsoft provides another complimentary software tool called DryadLINQ. DryadLINQ generates the necessary Dryad computations from the LINQ (Language-Integrated Query) expressions, which is an extension to C#. The expressiveness of DryadLINQ allows easy incorporation of the MapReduce-styled computation and the traditional relational algebra used in database systems.

2.4 EURExpress-II Case Study

In order to evaluate the practicality of using MapReduce to accelerate data intensive applications, this project applies the MapReduce model to a real data mining use case in the Life Sciences domain. The objective of this use case is to employ data mining techniques to facilitate the scientific exploration in the EURExpress-II Project [32]. The EURExpress-II project is a European integrated project involving 12 partners, aimed at generating expression data for more than 20,000 mouse genes by RNA in situ hybridization (ISH) on sagittal sections from E14.5 wild type C57Bl6 embryos.

The EURExpress-II project has built a transcriptome-wide atlas for the developing mouse embryo using RNA in situ hybridization experiments. A transcriptome-wide atlas describes the gene expression patterns of the subject in study. To date the project has collected over 500,000 captured images (~4TB) from the ISH experiments. These images are then manually annotated by domain experts. The annotation consists of tagging each image with anatomical terms from a set of 1,500 anatomical components. Tagging an image with a term means that the anatomical component is exhibiting gene expression in some part of the image. With the advancement in bioinformatics technologies that is capable of creating huge amount of such images in very short time, developing efficient automatic annotation technique is of utmost important. As such, a new data mining framework for automatic annotation of gene expression patterns in images from developmental mouse embryos is proposed in [33].
3 Methodology

Through a data mining use case based on real scenarios found in the EURExpress-II project, this project investigates the feasibility and performance of adapting data mining algorithms to the MapReduce model. The use case is to automate the manual annotation of gene expression patterns found in images produced by the Life Sciences experiments.

3.1 Data Mining Process

The high-level data mining and integration process of the use case is shown in Figure 2. The method used is essentially a pattern recognition task with the aim to produce multiple classifiers; each classifier identifies whether the gene expression pattern of an anatomical component is expressed in an image. Like most machine learning implementations, in order to automatically annotate the images, three phases are required: training stage, testing stage and deployment stage. We can view every phase as a process and each of these processes can be further divided into sub-processes. For instance, for the training phase, the sub-processes include image processing, feature generation, feature selection/extraction and classifier construction. The testing phase reuses and repeats the first three sub-processes and adds an evaluation for the classifiers. The deployment phase contains only the application of the classifier which then performs the automatic image annotation.
Depending on the goal of the individual sub-process, appropriate algorithms are used to achieve the desired outcome. Specifically, in this project, the following algorithmic processes are adapted to the MapReduce model:

- **Rescaling and Median Filter** for Image Processing
  The input images are of variable size. Rescaling is performed to obtain a standard size of 300x200 pixels. Median filtering is then used to reduce the image noise. The output is a standardized and de-noised image that can be represented as a two-dimensional array.

- **Wavelet Transformation** for Feature Generation
  Features representing gene expression patterns in the image are generated with wavelet transformation. Wavelet transformation allows decomposition of the images into space and frequency domains, allowing characteristic identification of images at a lower resolution (hence reducing the computation complexity). Nevertheless, the number of features generated with wavelet transformation is still considerable large (e.g. an image with dimension of 300x200 will generate 60,000 features).

- **Fisher Ratio Analysis** for Feature Selection and Extraction
  Feature selection is a procedure used to select the most important features out of a set of features so that their number is reduced and at the same time their class discriminatory information is retained as much as possible. For computation efficiency, it is necessary to select the most discriminating features from the huge number of features generated based on the method discussed above. Fisher Ratio Analysis is used to find the separation space for discriminating features of two classes by maximizing the difference between classes and minimizing the difference within the class. This facilitates the feature selection and extraction process.

- **K-Nearest Neighbors (KNN)** for Classifier Construction
The classifier construction process attempts to produce multiple classifiers, one for each gene anatomical component. As a starting point, a simple classification algorithm, the K Nearest Neighbors is used here. The neighbors are taken from a set of images for which the correct classification is known (i.e. images containing certain gene expression). The image being processed is then classified by a majority vote of its neighbors, with the image being assigned to the class most common amongst its $k$ nearest neighbors.

The processes described above are essential steps in both the training and testing phase, with the exception of prediction evaluation replacing classifier construction during the testing phase, where the performance of the classifiers produced in the training phase are being evaluated. For evaluation, 10-fold cross validation is used in which the dataset is divided into 10 subsets and, each time one of the 10 subsets is used as the test data while the other 9 subsets are used as training set. Average error is then computed across all 10 trials. Tuning and refinement to the data mining processes are performed to achieve the desired standards. The deployment phase will see the deployment of the adequately tuned classifiers to automatically annotate the images.

### 3.2 Data Mining Algorithms

The key elements in the data mining process lie in the underlying algorithms used for the various stages in the process. It is the combination of these algorithms that provides the overall result of the process. Within the process, individual algorithm can be replaced by more suitable one if required. This is especially important for future performance improvement (be it accuracy or computation time) of the method. The algorithms described below are used in this project. Since the objective of this project is to evaluate the acceleration of data-intensive applications, the algorithms used here serve as a good starting point, and is not in any way guaranteed to be the best algorithms in achieving the data mining goals.

#### 3.2.1 Median Filter
Median filtering is a nonlinear signal processing technique that has been proven to be very useful in noise removal (or suppression in some cases) in images [34]. It is widely-used in image processing because of its properties of edge preservation and robust impulse noise attenuation; at the same time it is highly computation efficient.

In a one-dimensional signal, the median filter will consist of a sliding window encompassing an odd number of data points. The center data point in the window is replaced by the median of the data points in the window. In image processing, the image forms a two-dimensional signal input to the median filter. The sliding window, in this case, can be of more complex structure such as a “mxm box” or a “cross”. The median filter then individually considers each pixel in the image as the centre of the window, and replaced it with the median value of all the pixels found in the window. Figure 3 shows an example of a 3x3 media filter.

![Figure 3: Example of a 3x3 Median Filter](image)

Note that although the median filter is very effective in suppressing impulse noise, it has the tendency to suppress valid signals in some cases. There are many strategies suggested in overcoming such issues and many more variations have been proposed to improve the performance [35][36][37]. In general, the output of the median filter is expected to be a smoothened image with useful details retained. The above discussed the general algorithm and a practical scenario; for theoretical analysis of the median filter, please refer to [38].

### 3.2.2 Wavelet Transformation
Wavelet transformation is a well-studied subject in signal processing. Fundamentally, wavelets are mathematical functions that cut up data into different frequency components, allowing the study of each component with a resolution matched to its scale [39]. It is often viewed as a better alternative to the traditional Fourier transformation because it exhibits the property of signal decomposition into both frequency and time domains with certain tradeoffs.

The wavelet analysis procedure is based on a wavelet prototype function, commonly known as analysing wavelet or mother wavelet. The temporal (or time domain) analysis is performed with a contracted, high-frequency version of the prototype wavelet, while the frequency analysis is performed with a dilated, low-frequency version of the same wavelet. Hence, like Fourier transformation where the transform consists of basis functions that are sines and cosines, Wavelet transformation consists of more complicated basis functions called wavelets. However, unlike sine and cosine functions, wavelets functions are localized in both space and frequency. The localization feature allows analysis at different time-frequency resolution with varying basis functions. In addition, it also produces “sparseness” in the transformed representations (i.e. in the wavelet domain) which are then exploited in many applications such as data compression, features extraction and noise reduction. Figure 4 depicts a comparison of the time-frequency resolution between Fourier transformation (with a square wave) and wavelet transformation (with Daubechies wavelets).

(a) Fourier transformation with square wave window  
(b) Wavelet Transformation with Daubechies wavelets

(source: An Introduction to Wavelets [39])

Figure 4: Comparison of Time-Frequency Resolution between Fourier Transformation and Wavelet Transformation
Since wavelets are basically mathematical functions, wavelet transformation comprises of an infinite set of wavelets. Wavelets that have same basic properties are grouped together into a wavelet family. Different wavelet families make different trade-offs between how compactly the basis functions are localized in space and how smooth they are. Within each family of wavelets are wavelet subclasses distinguished by the number of coefficients and by the level of iteration. Wavelets are classified within a family most often by the number of vanishing moments, an extra set of mathematical relationships for the coefficients that must be satisfied, which is directly related to the number of coefficients. Figure 5 shows some commonly used wavelets from different wavelet families.

While a detailed discussion on continuous wavelet transform can be found in [40], we will discuss the discrete wavelet transform in greater detail as it is more relevant to this project. Discrete wavelet transform is any wavelet transform for which the wavelets are discretely sampled to represent the original wavelet such that efficient computation with digitized (sampled) signals can be performed. The general form of the basis functions for discrete wavelet transformation is given as follows:

$$\psi_{(s,l)}(x) = 2^{-s/2}\psi(2^{-s}x - l)$$
where variables $s$ and $l$ are integers that scale and dilate the mother wavelet function $\psi$ to generate wavelets. The scale index $s$ indicates the wavelet’s width and the location index $l$ gives its position. Notice that every time the mother wavelet is rescaled or dilated by a factor of 2, the bandwidth is halved. In order to make up for the “missing” components, a scaling function is introduced to represent the low-pass spectrum. The scaling function is sometimes known as the “father” wavelet function and it can be expressed using the same mother wavelet function:

$$\varphi(x) = \sum_{k=-1}^{N=2} (-1)^k c_{k+1} \psi(2x + k)$$

where $c_k$ is known as the wavelet coefficients and it must satisfy linear and quadratic constraints of the form:

$$\sum_{k=0}^{N-1} c_k = 2, \quad \sum_{k=0}^{N-1} c_k c_{k+2l} = 2 \delta_{l,0}$$

with $\delta$ being the delta function and $l$ is the location index.

With respect to the description above, wavelet transformations are implemented as a series of low-pass and high-pass filters (or filter bank) in digital signal processing [41][40]. The low-pass filters, $g$, and high-pass filter, $h$, then correspond to the scaling and wavelet functions where the wavelet coefficients play a significant role. In such arrangement, the transformation simply involves iteratively down-sampling and filtering of the signal as illustrated in Figure 6.
In 1998, Wim Sweldens developed a much simplified and computation efficient method for discrete wavelet transformation called the lifting scheme [42]. The general form of discrete wavelet transformation with lifting scheme involves three basic steps: split, predict and update. At every level of decomposition, the split step divides the signal into odd and even indexed elements. Given any decomposition level $j+1$, the predict step predicts the odd elements from the even elements produced in the previous iteration using a predict operator ($P$) and an equation of the form of: $\text{odd}_{j+1} = \text{odd}_j - P(\text{even}_j)$. After the predict step, the original value of the odd elements would have been overwritten by the difference between the odd element and its even "predictor". The update step needs to compensate for the information loss by operating on the differences that are stored in the odd elements with an update operator ($U$) with an equation: $\text{even}_{j+1} = \text{even}_j - U(\text{odd}_{j+1})$.

Figure 7 illustrate a two levels discrete wavelet transformation with lifting scheme.

Wavelet transformation has been successfully used in image processing. Among them, Duabechies wavelets were found to be very useful in image classification.
This project uses the Daubechies-4 (D4) wavelets with lifting scheme to generate the image features required in the data mining processes. The algorithm used is based on the implementation described in [45] and the java implementation found at [46]. The lifting scheme is implemented with 2 update steps (U₁ and U₂) and a normalisation step (N₁ and N₂ combined) as shown below:


**Figure 8: Daubechies-4 Discrete Wavelet Transformation with Lifting Scheme**

The detailed algorithms for each step involved for a 1-dimensional data are as follows:

**Split:**

divides the input data into even elements and odd elements; and stored the even elements in the first half of an N element array (S[0] to S[half - 1]), odd elements in the second half of the same array (S[half] to S[N - 1]).

**Update 1 (U₁):**

for n = 0 to half - 1

\[ S[n] = S[n] + \sqrt{3} S[half + n] \]

**Predict (P₁):**

\[ S[half] = S[half] - \frac{\sqrt{3}}{4} S[0] - \frac{\sqrt{3}-2}{4} S[half - 1] \]

for n = 1 to half – 1
\[ S[\text{half} + n] = S[\text{half} + n] - \frac{\sqrt{3}}{4} S[n] - \frac{\sqrt{3} - 2}{4} S[n - 1] \]

**Update 2 (U) 2:**

for \( n = 0 \) to \( \text{half} - 2 \)

\[ S[n] = S[n] - S[\text{half} + n + 1] \]

\[ S[\text{half} - 1] = S[\text{half} - 1] - S[\text{half}] \]

**Noramlize (N1 and N2 combined):**

for \( n = 0 \) to \( \text{half} - 1 \)

\[ S[n] = \frac{\sqrt{3} - 1}{2} S[n] \]

\[ S[n + \text{half}] = \frac{\sqrt{3} + 1}{2} S[n + \text{half}] \]

For a 2-dimensional data, such as a \( m \times n \) image, the 2-D discrete wavelet transformation is performed by first applying the 1-D transformation horizontally on the rows and then vertically on the columns. This is equivalent to performing a transpose on the \( m \times n \) matrix of the image, performed the 1-D discrete wavelet transformation, and transposed the \( n \times m \) matrix again. Since every level of discrete wavelet transformation is essentially a low-pass and high-pass filters bank, four subband coefficients (LL1, HL1, LH1, and HH1) can be acquired after 1-level of 2-D discrete wavelet transformation. Decomposition can be further continued by applying the same procedure on the lowest resolution sub-image (LL) to obtain another four subband coefficients (LL2, HL2, LH2, and HH2). In the 1-level decomposition, LL1 is the approximation coefficients matrix; HL1, LH1 and HH1 are the horizontal, vertical and diagonal detail coefficients matrices respectively. Figure 9 shows the results of the 2-D discrete wavelet transformation with Daubechies-4 wavelets. The mouse embryo image presented here is drawn from the dataset used in this project.
3.2.3 Fisher's Ratio

Fisher's linear discriminant [47] is a popular method used in classification. However, it can also be viewed as a dimensionality reduction technique that attempts to project a high-dimensional data onto a line. The linear projection, \( y \), of a D-dimensional vector \( x \) to a line takes the form:

\[
y = w^T x
\]

A discussion based on a two class classification problem will be presented here as it is most relevant to the work performed in this project. In a two classes classification (or dimension reduction) with Fisher’s linear discriminant, the objective is then to find a threshold, \( \omega_0 \), on \( y \) such that we could classify \( y \geq -\omega_0 \) as class \( C_1 \) and as class \( C_2 \) otherwise. The simplest measurement for class separation, when projected onto \( w \), is to take the separation between the mean of the two classes, i.e.

\[
m_2 - m_1
\]

where

\[
m_1 = \frac{1}{N_1} \sum_{n \in C_1} x_n, \quad m_2 = \frac{1}{N_2} \sum_{n \in C_2} x_n
\]

and class \( C_1 \) has \( N_1 \) data points and class \( C_2 \) has \( N_2 \) data points.

However, a major problem in projecting multi-dimension data onto one dimension is that it leads to a loss of information, and may lead to classes that are well separated
in the original D-dimensional space to become strongly overlapping in one dimension. To avoid such overlapping from arising, Fisher’s idea of maximizing a function to produce a large separation between the projected class means and at the same time provide a small variance within each class is used. The concept makes use of the Fisher’s ratio (or Fisher’s criterion) rather than the mean as the class separation criterion. The Fisher ratio is defined to be the ratio of the between-class variance to the within-class variance. For a two classes case, the function of using Fisher’s Ratio for projection onto \( w \) is defined as follows:

\[
J(w) = \frac{(m_2 - m_1)^2}{s_1^2 + s_2^2}
\]

where \( s_1^2 \) and \( s_2^2 \) are the within class variance for class \( C_1 \) and class \( C_2 \) respectively [48]. Figure 10 shows an illustration of the differences between class separation using class mean and Fisher’s ratio.

![Illustration of Class Separation with Mean and Fisher's Ratio](source: Pattern Recognition and Machine Learning [48])

From Figure 10, it is clear that the one-dimensional projection using class mean (as shown in Figure 10a) causes significant overlapping between the two classes; whereas the projection with Fisher’s ratio (as shown in Figure 10b) successfully created a clean separation between the two classes.
In this project, the Fisher Ratio is used as a form of "Signal-to-Noise" Ratio measurement to select the representative components in the feature domain. In the processing, the Fisher ratios of all generated features from the two classes (whether an image exhibit a gene expression) are computed from the training dataset. The features are then ordered in descending order using the Fisher’s Ratio and the corresponding index components with top scores are then selected to be used in the KNN classifier.

### 3.2.4 K-Nearest Neighbour (KNN)

KNN [49] is a supervised instance-based learning algorithm used for classifying objects based on closest training examples in the feature space. The KNN algorithm is relatively simple: an object is classified by a majority vote of its neighbors, with the object being assigned to the class most common amongst its k nearest neighbors. There is no optimal number for the choice of K. A good choice is usually data dependent and derived through heuristic approaches such as cross-validation. While choosing a large K could reduce the effect of noise, it could result in computation inefficiency and ‘blurring’ of class boundaries. Hence a small positive integer is typically chosen for practical implementation. In a special case where K = 1, the object is simply assigned to the class of its nearest neighbor, also known as the nearest neighbor algorithm.

The selection of K-nearest neighbours is based on the distance between two data points in the feature space. Euclidean distance is usually used as the distance metric. For every unknown instance, the distance between the unknown instance and all the instances found in the training set is calculated. The K instances in the training set that have nearest distance to the unknown instance are selected, and the unknown instance will be assigned the class of the dominating neighbours. Figure 11 illustrate the KNN algorithm for a two classes example.
The unknown instance (denoted as green circle) should be classified either to the first class of blue squares or to the second class of red triangles. If \( k = 3 \) (denoted with the inner circle), it is classified to the second class because there are 2 triangles and only 1 square. If \( k = 5 \), (marked with the outer circle) it is classified to first class since there are now 3 squares vs. 2 triangles.

### 3.3 MapReduce and Hadoop

This project adapted a data mining process to the MapReduce model making use of a popular open source implementation called Hadoop. MapReduce and Hadoop play a very important role in this project and an in-depth understanding of these works is critical.

MapReduce is a parallel/distributed programming model. It is inspired by the map and reduces primitives present in functional languages. In its pure form, various implementations of the MapReduce interface are possible, depending on the desired context. Some currently available implementations are: shared-memory multi-core system [50][51], asymmetric multi-core processors [52], graphic processors [53], and cluster of networked machines [2]. Nevertheless, the most popular implementation is probably the one introduced by Google, which utilizes large clusters of commodity PCs connected with switched Ethernet. In essence, the Google’s MapReduce framework simplifies the development and lowers the cost of large-scale distributed applications on clusters of commodity machines.
In the MapReduce model, an application is implemented as a sequence of Map-Reduce tasks, each consisting of a Map phase and a Reduce phase that process a large number of independent data items. The MapReduce system infrastructure takes care of the parallelization, computation distribution, task management and fault recovery, allowing the programmers to focus on the core application algorithms.

The computation takes a set of input key-value pairs, and produces a set of output key-value pairs. The programmer expresses the desired computation as two functions: Map and Reduce. The *Map* function, written by the programmer, takes an input pair and produces a set of intermediate key-value pairs. The MapReduce framework groups together all intermediate values associated with the same intermediate key and passes them to the Reduce function. The *Reduce* function, also written by the programmer, accepts an intermediate key and the entire set of values associated with that key. It then merges these values to form a possibly smaller set of values. Typically just zero or one output value is produced per Reduce function invocation. For memory efficiency in handling huge data set, the intermediate values from the Map function are supplied to the Reduce function via an iterator. Note that in order for the Reduce function to process all intermediate values associated with the same key, an additional shuffle/sort phase is required to correctly marshal the intermediary key-value pairs from multiple Map functions to the responsible nodes running the Reduce function. Figure 12 shows an illustration of the general MapReduce data flow.
In a nutshell, the map and reduce functions can be summarized as follows:

\[
\text{map (key1, value1) } \rightarrow \text{ list (key2, value2)}
\]

&

\[
\text{reduce (key2, list (value2)) } \rightarrow \text{ list (key3, value3)}
\]

Although the distributed computing is largely simplified with the notions of Map and Reduce primitives, the underlying infrastructure is non-trivial in order to achieve the desired performance. A key infrastructure in Google’s MapReduce is the underlying distributed file system to ensure data locality and availability. In fact, Google’s MapReduce implementation leverages and depends heavily on an in-house distributed file system known as Google File System (GFS) [54]. Combining the MapReduce programming model and an efficient distributed file system, one can easily achieve the goal of distributed computing with data parallelism over thousands of computing nodes; processing data on terabyte and petabyte scales with improved system performance and reliability.

The success of Google’s MapReduce has prompted the development of many similar implementations such as: Hadoop [55], GridGain [56], Skynet [57], etc. Among these, Hadoop is one of the pioneers and probably the most recognized and widely-used. Hadoop is a java open source project that attempts to reproduce the Google's
MapReduce implementation. In a larger context, it is a collection of software projects intended for reliable and scalable distributed computing. The two key components are the Hadoop’s Distributed File System (HDFS) and the Hadoop MapReduce framework. Yahoo!, being the largest contributor of Hadoop, has been known to be using Hadoop extensively in enabling its business [58]. In additional, many major industry’s players such as IBM, Google, Microsoft and Facebook has also rendered their support for Hadoop [59][60].

This project uses Hadoop for the implementation of the data-intensive application. Hadoop (release 0.20.2) will be discussed in greater details in the following subsections.

3.3.1 Hadoop Distributed File System

HDFS is inspired by Google File System. It is designed to provide high throughput access with reliable storage of very large files across machines in a large cluster. It is designed for high performance and thus not as general-purpose as the commonly used distributed file systems such as Network File System (NFS) and Common Internet File System (CIFS). Applications that use HDFS are assumed to perform long sequential streaming reads from files. HDFS is optimized to provide streaming read performance, sacrificing the random seek times to arbitrary positions in files. Due to the large size of files, and the sequential nature of reads, the system does not provide mechanism for local data caching because the overhead of caching warrants a re-read from the HDFS.

HDFS is a block-structured file system. Files are broken into blocks of fixed size and stored across a cluster of one or more machines. Hence, a file in HDFS is stored as a sequence of blocks, and these blocks are not necessary on the same machine. Furthermore, individual files are stored with replication for fault tolerance purposes. During file access, the target machines which hold each block are chosen randomly on a block-by-block basis. Thus access to a file may require the cooperation of multiple machines.
HDFS employs a master/slave architecture. The master server, called NameNode, splits files into blocks and distributes them across the cluster with the desired replication factor. It holds all metadata information about files stored in the distributed file system. To overcome the single point of failure of NameNode, a periodic checkpoints node called SecondaryNameNode can be defined to copy the metadata from the NameNode at regular interval. The SecondaryNameNode does not replace the functionality of the NameNode in its failure. A crashed NameNode will render a total failure of the HDFS, but could be recovered to the last checkpoint using the metadata stored in the SecondaryNameNode. The HDFS slaves, called DataNodes, are the actual store of the data blocks. They serve read/write requests from clients and propagate replication tasks as directed by the NameNode. Figure 13 depicts the HDFS architecture.

![HDFS Architecture](http://sundar5.wordpress.com/2010/03/19/hadoop-basic/)

In HDFS, new DataNode can be added to the cluster easily as the new DataNode will automatically contact the NameNode and join the cluster. Although new files will be stored on the new DataNode, for optimum usage of the aggregated storage, data should be evenly balanced across all nodes. Hadoop includes an automatic data balancer tool that will intelligently balance blocks across the nodes to achieve an even distribution of blocks within a specified threshold. Similarly, to remove or
decommission a DataNode, Hadoop provides a decommissioning feature to ensure that no blocks are entirely replicated within the to-be-retired set of DataNodes.

3.3.2 Hadoop MapReduce

The Hadoop MapReduce is a software framework for distributed processing of large data sets on computing clusters. It is essentially a java implementation of the Google’s MapReduce framework. Hadoop MapReduce runs on top of HDFS, enabling collocation of data processing and data storage. It provides the runtime infrastructure to allow distributed computation across a cluster of machines with the famous abstraction of map and reduce functions used in many list processing languages such as LISP, Scheme and ML. Conceptually, a Hadoop MapReduce program transform lists of input key-value data elements into lists of output key-value data elements with two phases, namely map and reduce.

The first phase, map, involves a Mapper class which takes a list of key-value pairs as input and triggers its map() function. The map() function transforms each key-value pair in the input list to an intermediate key-value pair. The Hadoop MapReduce runtime then shuffle these intermediate key-value pairs and grouped together all the values associated with the same key and partition these groups of key-value pairs accordingly. The second phase, reduce, involves a Reducer class which takes the groups of key-value pairs from the map() function as input and triggers its reduce() function. The reduce() function receives an iterator of values which are associated with the same key and attempts to combine or merge these values to produce a single key-value pair as its output. Note that there could be multiple instances of Mapper and Reducer running on different machines; thus the map() and reduce() functions are executed concurrently, each with a subset of the larger data to be processed. Individual map tasks do not exchange information with one another, nor are they aware of one another's existence. Similarly, different reduce tasks do not communicate with one another. The only communication takes place only during the shuffling of the intermediate key-value pairs for the purpose of grouping values associated with the same key. The programmers never explicitly marshals information from one machine to another; all data transfer is handled by the Hadoop
MapReduce runtime, guided implicitly by the different keys associated with the values. Figure 14 shows the high-level pipeline of a MapReduce job executed in a Hadoop cluster.

![Figure 14: High-level Hadoop MapReduce Pipeline](source: http://developer.yahoo.com/hadoop/tutorial/module4.html)

In Hadoop, a MapReduce program applied to a data set is referred to as a Job. A Job is made up of several tasks that are executed concurrently, as far as the underlying hardware is capable of. The runtime of Hadoop MapReduce uses a master/slave architecture that is similar to the one employed in HDFS. The master, called JobTracker, is responsible for:

(a) querying the NameNode for the locations of the data block involved in the Job,

(b) scheduling the computation tasks (with consideration of the block locations retrieved from the NameNode) on the slaves, called TaskTrackers, and

(c) monitoring the success and failures of the tasks.

With the above understanding, we can examine the Hadoop MapReduce framework with greater details. The following discussion is based on [61]. Figure 15 depicts a detailed Hadoop MapReduce processing pipeline with two nodes.
The input for a Hadoop MapReduce task is typically very large files residing in the HDFS. The format of these input files is arbitrary, it could be formatted text file, binary format or any user-defined format. The key components of the Hadoop MapReduce framework are briefly discussed below:

**InputFormat:** The InputFormat defines where and how the input files are to be read, and how they are split up for processing. It defines the InputSplits that will break a file into chunks and provide a factory for RecordReader objects that will read the file. Hadoop provides implementations of two file-based InputFormat: TextInputFormat and SequenceFileInputFormat. Both TextInputFormat and SequenceFileInputFormat are derivations of a generic file-based InputFormat called FileInputFormat. In Hadoop, the default
InputFormat is the TextInputFormat which treats each line of each input file as a separate record, and performs no parsing. It automatically assigns a key for each value (i.e. each line in the file). TextInputFormat is particularly useful for unformatted data or line-based records like log files. The SequenceFileInputFormat reads special binary files that are specific to Hadoop, allowing data to be rapidly read into Hadoop mappers. Sequence files are block-compressed and provide direct serialization and deserialization of several data types (such as byte array). Sequence files can be generated as the output of other MapReduce tasks and are an efficient intermediate representation for data that is passing from one MapReduce job to another. InputFileFormat in Hadoop is extensible; programmers can provide customized InputFileFormat to suit their requirements.

**InputSplits:** An InputSplit describes a unit of work that comprises a single map task in a MapReduce program. While map tasks may involve reading a whole file; they often involve reading only part of a file since the input files are assumed to be very large. By default, the FileInputFormat breaks a file into chunks that is equivalent to the block size in the HDFS. By processing a large file in chunks, it allows several map tasks to operate on a single file in parallel. In Hadoop, the various blocks that make up the file may be spread across several different nodes in the cluster, thus the tasks can be scheduled to run on each of these different nodes with data locality. A custom InputFormat can control whether and how the file is broken up into splits. A split is corresponding to a map task during the mapping phase.

**RecordReader:** Note that the InputSplit simply defined a unit of work; it does not provide the data access mechanism. The RecordReader is the one responsible for loading the data from its source and converts it into (key, value) pairs suitable for the Mapper processing. The RecordReader is invoked repeatedly on every item in the input until the entire InputSplit has been consumed. Each invocation of the RecordReader leads to another call to the map() method of the Mapper.
**Mapper:** The Mapper performs the user-defined processing of the map phase of the MapReduce program. The map() method in the Mapper performs the desired data transformation in the MapReduce program. In Hadoop, a new instance of Mapper is instantiated in a separate Java process for each map task (corresponding to an InputSplit). The Mapper receives the (key, value) pairs as input. The map() method is invoked on every (key, value) pair and it receives two parameters in addition to the key and the value: the OutputCollector object which will forward the output (key, value) pair to the reduce phase of the job; and the Reporter object that provides information about the current task to the JobTracker.

**Shuffle, Partition and Sort:** The output (key, value) pairs with the same key need to be grouped together for the reduce phase. This requires the nodes performing the map tasks to exchange the intermediate outputs with the corresponding nodes where the reduce phase is to be carried out. More than one map task may be running on a node, but the information exchange is performed once an individual map task is completed. This process of moving map outputs to the reducers is known as shuffling. To cater for concurrent reduce processing, the intermediate key space is partitioned and distributed to a number of reducers. The Partitioner class determines which partition an intermediate (key, value) pair will go to. The default partitioner computes a hash value for the key and assigns the partition based on this result. For better data distribution within the reduce partitions, the programmers can provide customised Partitioner. Since each partition could potentially contain more than one intermediate key, the set of intermediate keys is automatically sorted before they are presented to the Reducer.

**Reducer:** The Reducer is responsible for the performing the user-defined codes during the reduce phase. The Reducer calls its reduce() method once for each key in the partition assigned to it. It receives a key and an iterator of all values associated with the key. Just like in the case of the Mapper, the
Reducer also receives the OutputCollector and Reporter objects for output generation and status reporting respectively.

**OutputFormat:** The (key, value) pairs from the Reducer are written to files via the OutputCollector. The exact formats of the outputs are governed by the OutputFormat. The OutputFormat functions much like the InputFormat. Hadoop provides implementation of an empty OutputFormat (NullOutputFormat) and a generic file-base OutputFormat, FileOutputFormat, with two derivations: TextOutputFormat and SequenceFileOutputFormat. Each Reducer writes a separate file in a common output directory dictated by the FileOutputFormat. These files will typically be named part-nnnnn, where nnnnn is the partition id associated with the reduce task.

In Hadoop, the default OutputFormat is TextOutputFormat, which writes (key, value) pairs on individual lines of a text file. This output type is human-readable and can be easily re-read by a later MapReduce task. However, a better intermediate format for use between MapReduce jobs is the SequenceFileOutputFormat which rapidly serializes arbitrary data types to the file; the corresponding SequenceFileInputFormat will deserialize the file into the same types and presents the data to the next Mapper. The NullOutputFormat generates no output files and disregards any (key, value) pairs passed to it by the OutputCollector. This is useful if one needs to explicitly writing his/her own output files in the reduce() method, and do not want additional empty output files generated by the Hadoop framework.

**RecordWriter:** RecordWriter is similar to RecordReader, it is used to write the individual records to the files as directed by the OutputFormat.

**Combiner:** Combiner is an optional component in the pipeline (and it is omitted in Figure 15). It is used to reduce communications during the shuffling phase between the map tasks and reduce tasks. The Combiner is
invoked straight after the Mapper, before the Partitioner and Reducer. It receives all data emitted by the Mapper instances on a given node and attempts to combine or merge these intermediate outputs before sending to the Reducer. Hence, when a Combiner is defined in a MapReduce job, the output from the Combiner is sent to the Reducers instead of the output from the Mappers. In many cases, we can viewed Combiner as a "mini-reduce" process which operates only on data generated by one machine. The Combiner is implemented as an instance of the Reducer class and it is common to use the same Reducer in a MapReduce job as the Combiner. However, if the Reducer itself cannot be used directly as a Combiner because of commutativity or associativity, the programmer might still be able to write a different implementation as the Combiner.

Another key aspect of Hadoop is its failure model. Hadoop is designed to have high degree of fault tolerance. In comparison to many available parallel/distributed systems, it is able to complete the assigned tasks failures in the cluster [62]. The primary way that Hadoop achieves fault tolerance is through restarting tasks. The slave nodes (TaskTrackers) involved in the computation are in constant communication with the master node (JobTracker). If a TaskTracker failed to communicate with the JobTracker for a period of time (by default, 1 minute), the JobTracker will assume failure on that TaskTracker. It will then assign another active TaskTracker to re-execute all the tasks that were in progress on the failed TaskTracker. In Hadoop MapReduce, a computing node has no knowledge of its peers and knows only its own set of inputs and its own outputs, this enable a very simple and reliable task restart procedure upon any failure.

Another special characteristic in Hadoop that is worth mentioning here is the speculative execution model. In a parallel system, it is a common problem for a few slow nodes to slow down the entire computation in the cluster. In Hadoop, since tasks are run in isolation from one another, so the same input can be processed multiple times in parallel by different nodes. Hadoop exploit this property to combat the rate-limit issue presented by slower computing nodes. When most of the tasks in
a job are reaching 100% completion, the Hadoop platform will schedule redundant copies of the remaining tasks across several free nodes. With such set up, the TaskTracker that completes the remaining task first will inform the JobTracker its accomplishment, and the JobTracker will instruct the speculative execution to stop and discard the unnecessary outputs.

It is important to note that, in practice, it is difficult to solve any problem with a single MapReduce job and many large problems can be gracefully handled with a series of interconnected MapReduce job. The interconnecting is referred to as job chaining in Hadoop. Job Chaining allows for decomposition of a large problem to many smaller problems and solving them individually using MapReduce model. These MapReduce solutions of the smaller problems are executed in series to accomplish the overall goal. In this arrangement, the outputs of a MapReduce job will serve as the inputs to the next MapReduce job in the chain, as presented below:

(Map1 -> Reduce1) \(\rightarrow\) (Map2 -> Reduce2) \(\rightarrow\) (Map3) \(\rightarrow\) …. \(\rightarrow\) (MapN -> ReduceN)

Also note that it is also possible to create a MapReduce job without the reduce phase in Hadoop.

### 3.3.3 Other Hadoop Offerings

In addition to HDFS and Hadoop MapReduce, the Hadoop project also provides many software components for distributed computing, such as a scalable, distributed database that supports structured data storage for large tables (HBase), a data warehouse infrastructure that provides data summarization and ad hoc querying (Hive), a high-level data-flow language and execution framework for parallel computation (Pig) and a high-performance coordination service for distributed applications (Zookeeper). More detailed information on these components can be found in [55].
4 Implementation

A prototype system for the data mining processes described in the previous section is implemented with the Java-based, open source MapReduce framework – Apache Hadoop. The version used is 0.20.2, released on 26 February 2010.

The key component of the prototype system is the data mining workflow. The data mining process kicks off with the Data Packer module where the gathering and distribution of the raw data is performed. In this case, the raw data comprises of image files of the mouse embryo and a text file containing the annotations or labels. The processed raw data are served as distributed inputs to the data mining workflow. The entire data mining workflow is modelled as a chain of 8 MapReduce jobs. The system overview is presented in Figure 16.

![Figure 16: Overview of Data Mining Prototype System on Hadoop](image-url)
The prototype design is modular, algorithms used in the MapReduce jobs can be easily replaced with better ones whenever the need arises. The following sub-sections discussed the main components in the prototype.

**4.1.1 MRController**

As shown in Figure 16, the user interacts with the prototype system through the controller class, MRController (MapReduce Controller). The MRController allows the user to specify the data files and annotation file locations, as well as the output location for the data mining results. In addition, it reads the system configuration file, mrdia-config.xml to configure the runtime properties of the system. Some important runtime properties are the number of computing nodes, the locations of all intermediary results of the MapReduce jobs, the k value for the k-fold cross validation, the number of top N feature index to be selected and the value of “K” to be used in the K-Nearest Neighbours algorithms.

During execution, the MRController instantiates the Data Packer to prepare the data, configure all the MapReduce jobs according to the settings specified in the configuration file, ‘marshal’ the outputs as inputs for jobs and most importantly start the chain of MapReduce jobs.

**4.1.2 Data Packer**

Hadoop MapReduce jobs run more efficiently with large input files stored in the underlying Hadoop Distributed File System (HDFS). As such, the Data Packer responsibility is to “pack” the raw data in the desired format and store them in the HDFS.

A major contribution of the Data Packer is to read the raw data and repack it into the desired data structure for the MapReduce jobs. Individual image binaries and its corresponding annotations information are stored into a java object together with a generated unique identifier. The list of java objects that represents all the images are then written into an optimised file format known as SequenceFile in Hadoop. SequenceFile is a persistent data structure used in Hadoop for storing binary key-
value pairs. It is commonly used as containers for smaller files in Hadoop to achieve the effect of large input files to any MapReduce job. The added advantage of using SequenceFile is that it can be easily compressed and decompressed within the Hadoop framework.

The Data Packer also acts as an initial workload distributor. It will produce equal number of SequenceFile to the number of computing nodes as instructed by the MRController (which reads the value from the configuration file). However, for very large amount of inputs, the SequenceFile produced will be automatically “chunked” by the HDFS if its size exceeds the block size of the HDFS. This allows for further workload distribution.

4.1.3 Image Pre-processing

The image pre-processing is the first MapReduce job in the chain. It is implemented as a Mapper class in Hadoop since there is no need for a Reducer in this case. The Image Pre-processing Mapper takes the SequenceFile containing a list of image binaries as input and applies the following operations on every image:

**Gray-scale conversion:** Images may be encoded in RGB colour. For standardisation of data range, we need to convert all images to gray-scale. This is easily achieved through standard Java AWT API. The code snippet is shown below:

```java
BufferedImageOp op = new ColorConvertOp(
    ColorSpace.getInstance(ColorSpace.CS_GRAY),
    null);
BufferedImage gs_image = op.filter(bimage, null);
```

**Rescaling:** Images from the dataset are of different dimension. For standardisation of data dimension, we need to resize the images to a standard size of 300 x 200 pixels. In Java, this image rescaling can also be achieved easily with the Graphics2D class found in the AWT API, as follows:
Median Filter: Images from the dataset may contain noise. Median filter is known to be effective in noise reduction. A median filter with 3x3 square window is used here. In Java, the median filtering is provided in an optional API called Java Advanced Imaging (JAI). The code for median filtering using JAI is shown below:

```java
int n = conf.getInt("mrdia.image_preprocess.medianfilter.masksize", 3);
RenderedOp renderedOp = MedianFilterDescriptor.create(rsgs_image,
    MedianFilterDescriptor.MEDIAN_MASK_SQUARE,
    n, null);
BufferedImage f_image = renderedOp.getAsBufferedImage();
```

The final step in the image pre-processing is to extract the gray-scale pixel values into a 300x200 matrix representation (i.e. a 2-D array of type double) to be used for the next step. Similarly, the 2-D array is serialised as Java object binaries and associated with the unique image identifier to form the <key, value> pair in the output SequenceFile.

4.1.4 Feature Generation with Wavelet Transformation

The feature generation is implemented as a Mapper class. The inputs to the Feature Generation Mapper are the SequenceFiles produced by the Image Pre-processing Mapper. The map() function will be invoked for every <key, value> pair found in the input SequenceFile. The key is the unique identifier of the data item and the value contains a Java object binaries with a 2-D array (i.e. the 300x200 matrix representing
the gray-scale pixels of the image). For every <key, value> pair, the Feature Generation Mapper performs 2-D discrete wavelet transformation on the 300x200 matrix using the lifting scheme algorithms described in the previous section. A 3 levels transformation with Daubechies-4 wavelet is used in the implementation. The output of the transformation is a 300x200 sparse matrix containing the transformed coefficients. The 300x200 matrix is then flattened into a vector of 60,000 elements to facilitate future processing.

A critical design decision has been made in the Feature Generation step, that is, not to split the image into segments for concurrent discrete wavelet transformation. The decision is based on the following consideration:

**Data Dimension**: The image dimension being processed in this use case is 300x200, relatively small and inefficient to parallelise as compared to the total overheads involved in the Hadoop MapReduce framework.

**I/O and Reconstruction Cost**: Hadoop MapReduce is an I/O intensive framework. Although the I/O in HDFS is highly optimised, the total cost of splitting, distributing and recombination the 300x200 matrix is much higher compared to local processing. Part of the inefficiency comes from the fact that such distributed computation on the MapReduce model requires at least two MapReduce jobs: one for splitting and distributed computation and the other for recombination.

**Lifting Scheme Constraints**: The split, predicate and update steps defined in the lifting scheme is performed in serial, and the computation relies on results of previous iteration as well as neighbouring data element. This makes parallel computation of lifting scheme much more difficult. We could potentially drop the use of the lifting scheme, but the in-place calculation is very desirable to allow stability in the system (constant memory requirement).
Hence, in this implementation, the focus is on effective data placement for even workload distribution among the MapReduce computing nodes while discrete wavelet transformation of individual image is performed locally. Nevertheless, parallel computation within the node (e.g., multi-core/multi-processor shared-memory model) should improve the performance for processing very large image.

4.1.5 Testing and Training Set Extraction for 10-Fold Cross Validations

The 10-fold cross validations in the system is implemented as a for-loop iteration. This construct is used to facilitate the input mechanism used in Hadoop MapReduce. A Hadoop MapReduce job can accept a directory as input and pick up all the files within the directory as the real inputs for the job. The testing set and training set are generated dynamically in each iteration based on simple hashing of the unique identifier assigned to the data item during the “Data Packing” step. The testing set and training set are produced as SequenceFile stored in different directory in the HDFS.

The extraction of testing and training set is implemented as Hadoop Mapper class so that parallel extraction can be performed on all computing nodes. Technically, the testing set and training set extraction can be performed in a single task, but they are implemented as two separate Mappers because Hadoop MapReduce job produces a single list of output. The code for dynamic extractions of testing set and training set is shown below:

```java
// testing set extraction
int id = key.get();
int fold = conf.getInt("mrdia.nfold_validation.fold_count", 10);
int currentFold = conf.getInt("mrdia.nfold_validation.current_fold", 0);

if ((id % fold) == currentFold) {
    context.write(key, value);
}
```
While this approach is dynamic and best suited for MapReduce directory input style, a major disadvantage is the unnecessary computation time required for the dynamic extraction of testing set and training set during every cross-validation iterations. I believed there is a better implementation strategy to achieve the same effect without repeated testing set and training set extraction. Due to time constraint, this shall be a future enhancement to the system.

4.1.6 Feature Selection with Fisher’s Ratio

The Feature Selection process is accomplished with 3 MapReduce jobs: Mean and Variance Calculation, Fisher’s Ratio Calculation and Top N Feature Index Selection. The input to this step is the training set extracted in the previous step.

During the “Data Packing” step, images are tagged with the classification labels (1 or 0, depending on whether image exhibits certain gene expression). The first step in calculating the Fisher’s Ratio between the two classes of data is to compute the mean and variance of the two classes. The class mean and variance are calculated for all features in every data item. The features are the transformed coefficients generated by the discrete wavelet transformation. A fully parallelised method is used to calculate the mean and variance across all data items belonging to the same class. The Mean and Variance Calculation is implemented as a MapReduce job with both Mapper and Reducer. The Mapper maps the data items according to the classification using the classification label as the key for its output <key, value> pair. The value is the vector containing the wavelet transformed coefficients of the image, i.e.

```java
// training set extraction
int id = key.get();
int fold = conf.getInt("mrdia.nfold_validation.fold_count", 10);
int currentFold = conf.getInt("mrdia.nfold_validation.current_fold", 0);

if ((id % fold) != currentFold) {
    context.write(key, value);
}
```
The Reducer receives the list of wavelet transform coefficients belonging to the same class and performs the calculation of mean and variance of the coefficients across the list using the following formulae:

\[ m = \frac{\sum_{i=1}^{n} x_i}{n} \]

\[ s^2 = \frac{\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2 / n}{n} \]

where \( m \) is the mean and \( s^2 \) is the variance. Note that these formulae allow calculation of both the mean and variance with a single iteration over the list of data items, and it is highly parallelisable. A Combiner was initially introduced to perform local partial mean and variance calculation based on a parallelised version of the formulae shown above. However, experiments with the Combiner did not show significant improvement with the data set tested. Hence, the parallelised complication was dropped from the latest implementation. In the reduce phase, only two Reducer are needed since there are only left with two distinct keys, i.e. the classification “1” and “0”.

The next MapReduce job is to calculate the required Fisher’s Ratio between the two classes for every feature. The job comprises of a Mapper and a Reducer. The Mapper “rearrange” the data as <key, value> pair using the index of the feature as the key and the value consists of the classification label, the feature mean and the feature variance. This is to facilitate parallel calculation of the Fisher’s Ratio of all the features since the internal shuffling will hash the key (which is the feature index now) and distribute the workload across all the available Reducers (which is at least the number of computing nodes). The Reducer performs the actual calculation. The Fisher’s Ratio, \( J(w) \), is calculated using the equation:
\[
J(w) = \frac{(m_2 - m_1)^2}{s_1^2 + s_2^2}
\]

where \(m_1\) and \(s_1^2\) are the mean and variance of class 1 respectively; \(m_2\) and \(s_2^2\) are the mean and variance of class 2 respectively. Note that, to this end, the calculation time is bounded by the dimension of the data (in this case, it is 300 x 200) and not by the number of data items in the training set.

The final step in the Feature Selection is to select the most significant feature to be used in the classification step. This is accomplished by the MapReduce job, Top N Feature Index Selection. This job comprises of a Mapper and a Reducer. The Mapper combines all the Fisher’s Ratio of the features to a single list using a common key (which is arbitrarily assigned). The mapping is shown below:

\[
<\text{Feature Index, Feature Fisher’s Ratio}> \rightarrow \text{TopNFeatureIndexMapper} \rightarrow <\text{“some key”}, [\text{Feature Index, Feature Fisher’s Ratio}]>
\]

The Reducers accepts the list of [Feature Index, Feature Fisher’s Ratio], sorts the list in descending order according to the Fisher’s Ratio, and selects the top \(N\) features. The value of \(N\) is specified by the MRController during the job setup. The MRController reads this value from the configuration file. The selected top \(N\) indexes are placed in an integer array and written to a SequenceFile in HDFS.

4.1.7 Classification with K-Nearest Neighbour

The last step in the data mining workflow is to predict the classification using K-Nearest Neighbour (KNN) method. The KNN is implemented as a MapReduce job comprising of a Mapper, a Combiner and a Reducer. This MapReduce job is slightly different from the others described above. Unlike, the other jobs, it requires three inputs: the testing set, the training set and the top \(N\) feature indexes. Unfortunately, Hadoop MapReduce job requires the input to be a list of \(<\text{key, value}\>\) pair with identical format. Hence, a design choice was made to use only the training set as the direct input to the Mapper while the testing set and the top \(N\) feature indexes are
dynamically distributed to all the computing nodes using the DistributedCache class found in Hadoop API. The choice is reasonable as the aim is to minimise the communication cost with the relatively smaller testing set and top N feature indexes. The code for distributing the required files is as below:

```java
FileSystem hdfs = FileSystem.get(conf);
FileStatus[] fileStatus;

// distribute the current testing set
String testSetPathPattern = outputDir + "/" + testingSetDir + "/part*";
fileStatus = hdfs.globStatus(new Path(testSetPathPattern));
for (int t=0; t<fileStatus.length; t++) {
    Path cachedFile = fileStatus[t].getPath();
    DistributedCache.addCacheFile(cachedFile.toUri(), conf);
}

// distribute the top N features indexes
String topNIndexesPathPattern = outputDir + "/" + topNFeaturesDir + "/part*";
fileStatus = hdfs.globStatus(new Path(topNIndexesPathPattern + "/part"));
for (int t=0; t<fileStatus.length; t++) {
    Path cachedFile = fileStatus[t].getPath();
    DistributedCache.addCacheFile(cachedFile.toUri(), conf);
}
```

The central concept of the parallel-KNN is borrowed from Cell-KNN [63]. The training set is distributed on all the computing nodes. The testing set and top N feature indexes are sent to all computing nodes. During initiation of the Mapper, the setup() is invoked where the testing set and top N feature indexes are retrieved and kept in memory. In Hadoop, retrieval of the distributed files is very simplified and the code is shown here:

```java
Path[] distributedDataFiles;
Configuration conf = context.getConfiguration();

// retrieve the testing set and top N features indexes files from the
// distributed cache
distributedDataFiles = DistributedCache.getLocalCacheFiles(conf);
```
The in-memory testing set is a reduced set which contains only the data elements (or attributes) corresponding to the indexes found in the top N feature indexes. This is to ensure minimum memory is used for storing the testing set. This in-memory testing set is used extensively in the Mapper class because we need to compute the Euclidean distance between all the data items in the testing set and all the data items in the training set. Since the map() function will be triggered for every training set data item, the Euclidean distance calculation is performed in the map() function. Note that only data elements (or attributes) corresponding to the indexes found in the top N feature indexes will be used for the calculation. Since the expected output is to predict the classification of every data item in the testing set, the key for the <key, value> pair output ought to be a unique and yet recognisable identifier of the data item. A customised key containing the unique identifier (assigned during the Data Packing step) and the image filename is created because the unique identifier is a system assigned value not recognised by the user. The value of the <key, value> pair is a data structure containing important information (such as unique identifier, classification label and Euclidean distance from the testing data item) of the “neighbour” found in the training set. The end effect of the mapping phase is such that the distance between all testing set and training set are calculated concurrently on all computing nodes, with every node responsible for calculating the distance between all data items in the complete testing set and a subset of the training set. Figure 17 illustrates the distributed KNN calculation in MapReduce.
The Reducer will receive all every testing data item and all its neighbours. It then sorts all the neighbours in ascending order according to the distance and picks the K nearest neighbours from the list. The output of the KNN Reducer is a text file containing the classification prediction of the entire testing set.

To reduce the amount of communication between the Mapper and Reducer, a Combiner is introduced to extract the K nearest neighbours from the output of the Mapper. With a Combiner, for each testing data item, the Reducer will only receive the K nearest neighbours from each Mapper.
5 Experiment and Evaluation

Experiments are conducted to evaluate the computational performance of the prototype system. To be consistent with most studies of distributed/parallel computation, the performance evaluation will be based on measurement of speedup, scaleup and sizeup. These will allow easy comparisons of results with other alternative implementations. Accuracy of the classification prediction is not the objective of this project and hence not evaluated.

5.1 Experiment Overview

The experiments are conducted on a Cloud computing environment. In general, Cloud computing can broken down into three different service models: Software as a service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). In this project, the Amazon Elastic Compute Cloud (EC2) [64] (a IaaS provided by Amazon) is used. In this case, the computation infrastructure (the machines and interconnect network) is provided and managed by Amazon. However, in this context, the machines provided are not physical machines, but virtual machines. Amazon EC2 operates a “pay as you go” model, allowing the flexibility of paying only for the required computation time.

5.1.1 Setup and Configuration

The machines used for the experiments are “Large Instance Type” in Amazon EC2 environment. The configuration of each machine is as follows:

<table>
<thead>
<tr>
<th>CPU</th>
<th>4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><em>(One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.)</em></td>
</tr>
<tr>
<td>Memory</td>
<td>7.5 GB memory</td>
</tr>
<tr>
<td>Storage</td>
<td>850 GB instance storage (2x420 GB plus 10 GB root partition)</td>
</tr>
<tr>
<td>I/O</td>
<td>High</td>
</tr>
<tr>
<td>Performance</td>
<td>Fedora Core 8 (2.6.21.7-2.ec2.v1.2.fc8xen Linux kernel)</td>
</tr>
</tbody>
</table>
The experiments are conducted with a cluster of 1, 2, 4, 8 and 16 machines configured with Hadoop version 0.20.2. The initial input dataset used contains 808 images. The size of input datasets is multiply by a factor of 2 in every increment, up to 32 times of the initial input data size, i.e. 1x, 2x, 4x, 8x, 16x, and 32x. At 32x, the total number of images used is 25,856, with every image containing 60,000 pixels (300x200).

The infrastructure setup of the experiment environment is largely simplified with the provision of Amazon Machine Image (AMI) on EC2, but still requires the following simple steps:

1. Create an basic bare instance (with Fedora Core 8 Operating System) from the AMIs provided by Amazon;
2. Start the instance and install all necessary software: Java Development Kit (JDK), Hadoop binaries and other additional Java libraries;
3. Configure JDK and Hadoop;
4. Create customized AMI from the configured instance;
5. Copy the prototype program executable and data files to the EC2 instance that is to be the master node in the computing cluster;
6. Launch the required number of slave nodes using the customized AMI;
7. Configure and start the Hadoop cluster;
8. Execute the prototype program executable

The entire setup and configuration is completed in less than two hours.

5.1.2 Experimental Results

The results are obtained from the average of 5 runs. Each run executes the entire data mining process, from data packing to classification with KNN, but terminates after a single fold in the 10-fold cross validations. Since the data are partitioned equally in the 10 folds and we are only interested in the computing time and total data loading, a complete data mining flow with a single fold of data will be sufficient for the purpose of our evaluations.
A summary of the experiment results is presented in Table 1. The table shows the average execution time (in seconds) of one run under each experiment scenario.

<table>
<thead>
<tr>
<th>Data Size (image count)</th>
<th>Execution Time (sec)</th>
<th>Number of Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1x (808)</td>
<td>309</td>
<td>261</td>
</tr>
<tr>
<td>2x (1,616)</td>
<td>601</td>
<td>324</td>
</tr>
<tr>
<td>4x (3,232)</td>
<td>863</td>
<td>422</td>
</tr>
<tr>
<td>8x (6,464)</td>
<td>1258</td>
<td>648</td>
</tr>
<tr>
<td>16x (12,928)</td>
<td>2258</td>
<td>1206</td>
</tr>
<tr>
<td>32x (25,856)</td>
<td>5168</td>
<td>2623</td>
</tr>
</tbody>
</table>

Table 1: Summary of Experiment Results

5.2 Evaluations

Three measurements are used for the evaluations: speedup, scaleup and sizeup. The following sub-sections discussed these measurements and the results obtained.

5.2.1 Speedup

Speedup evaluates the ability of the parallelism to improve execution time. It is defined as the ratio of the sequential execution time to the parallel execution time. Speedup can be expressed as follow:

\[ Speedup(m) = \frac{T(1)}{T(m)} \]

where \( m \) is the number of computing node, \( T(1) \) is the execution time of the tasks on 1 computing node, \( T(m) \) is the execution time of the parallel tasks with \( m \) computing node. A perfect parallelism demonstrates linear speedup, i.e. a system with \( m \) times the number of computers yields a speedup of \( m \). However, linear speedup is difficult to achieve because the communication cost increases with the number of computing nodes.
Figure 18 shows the speedup characteristic of the prototype system with increasing input data size.

From the results obtained, the system exhibits undesired speedup with smaller datasets, but a reasonable speedup is achieved with a large enough dataset that compliments the number of computing nodes. For example, at 32x of initial input size (25,856 images), the speed up achieved with 8 machines is 5.32; but with 16 machines, the speedup achieved is only 6.36, a rather disappointing result. Based on the trend shown in Figure 18, we could make two observations:

1. MapReduce model is most suited for distributed computing with huge datasets;
2. Some rate-limiting factors exist in the system.

While the first observation is a desired property for data intensive applications, the second observation deserves further analysis. A detailed analysis on all the
MapReduce jobs corresponding to the individual data mining algorithms provides some insights to the rate-limiting factors.

Table 2 presents the execution time of individual MapReduce job at 32x input data size. Table 3 and Figure 19 show the speedup of the MapReduce jobs at 32x input size.

<table>
<thead>
<tr>
<th>MapReduce Job</th>
<th>Execution Time(sec)</th>
<th>Number of Machines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Image Pre-processing (IP)</td>
<td>1533</td>
<td>779</td>
</tr>
<tr>
<td>Wavelet Transformation (WT)</td>
<td>485</td>
<td>239</td>
</tr>
<tr>
<td>Testing Set Extraction (TT)</td>
<td>251</td>
<td>105</td>
</tr>
<tr>
<td>Training Set Extraction (TR)</td>
<td>620</td>
<td>292</td>
</tr>
<tr>
<td>Mean Variance Calculation (MV)</td>
<td>624</td>
<td>355</td>
</tr>
<tr>
<td>Fisher’s Ratio Calculation (FR)</td>
<td>101</td>
<td>33</td>
</tr>
<tr>
<td>Top N Feature Index Selection (TN)</td>
<td>53</td>
<td>28</td>
</tr>
<tr>
<td>K-Nearest Neighbour (KNN)</td>
<td>1349</td>
<td>770</td>
</tr>
</tbody>
</table>

**Table 2: Execution Time of MapReduce Jobs at 32x Input Size**

<table>
<thead>
<tr>
<th>MapReduce Job</th>
<th>Speedup</th>
<th>Number of Machines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Image Pre-processing (IP)</td>
<td>1</td>
<td>1.97</td>
</tr>
<tr>
<td>Wavelet Transformation (WT)</td>
<td>1</td>
<td>2.03</td>
</tr>
<tr>
<td>Testing Set Extraction (TT)</td>
<td>1</td>
<td>2.39</td>
</tr>
<tr>
<td>Training Set Extraction (TR)</td>
<td>1</td>
<td>2.12</td>
</tr>
<tr>
<td>Mean Variance Calculation (MV)</td>
<td>1</td>
<td>1.76</td>
</tr>
<tr>
<td>Fisher’s Ratio Calculation (FR)</td>
<td>1</td>
<td>3.06</td>
</tr>
<tr>
<td>Top N Feature Index Selection (TN)</td>
<td>1</td>
<td>1.89</td>
</tr>
<tr>
<td>K-Nearest Neighbour (KNN)</td>
<td>1</td>
<td>1.75</td>
</tr>
</tbody>
</table>

**Table 3: Speedup of MapReduce Jobs at 32x Input Size**
The results suggested the followings:

- Jobs that are computation intensive have very good speedup performance in MapReduce model. These are the Image Pre-processing (IP), Wavelet Transformation (WT) and K-Nearest Neighbours (KNN).

- Long running job has better speedup than job that has short duration. Comparing Testing Set Extraction (TT) and Training Set Extraction (TR), both jobs have the same inputs and the computations are almost identical, but Testing Set Extraction execution time is always 10% of the Training Set Extraction in our 10-fold cross validation setup.

- Jobs that have very short duration (in the range of 20-40 seconds) have undesired speedup. These are the Fisher’s Ratio Calculation (FR) and Top N Feature Index Selection (TN). Since the inputs to these jobs are always reduced to a constant number (in this case, the image dimension, i.e. 60,000 data points), the minimum execution time is always bounded by the MapReduce overheads. Research has shown that MapReduce job setup requires substantial amount of time [62], hence job that are shorter than 20-
40 seconds will not benefit from the parallelisation. This is one of the rate-limiting factors.

- The key rate-limiting factor in the system is the Mean Variance Calculation (MV). It exhibits reversed trend beyond 4 nodes. This is due to the fact that the calculation involves only two classes in our use case, and only two Reducers (one for each class) are used to produce the results. When the number of computing nodes increases, the number of Mappers increases proportionally, but the number of Reducer is fixed at two. Hence, a communication bottlenecks build up in the cluster, resulting in longer total execution time. The non-parallelise portion in the Reducer is the main culprit in limiting the overall system speedup. The findings here are consistent with the discussion in [62].

Finally, the experiments are conducted with near “out-of-box” configurations, and no serious attempt has been taken in tuning the performance. Better speed up than what is presented should be achievable with proper tuning. In addition, a much desirable speedup should be achievable if we could parallelise the non-parallelise portion in the Mean Variable Calculation MapReduce job.

### 5.2.2 Scaleup

Scaleup evaluates the ability of the parallelism to grow both the system and the data size. Scaleup is defined as the ability of an $m$-times larger system to perform an $m$-times larger job in the same run-time as the original system. Scaleup can be expressed as follows:

$$Scaleup(m) = \frac{T(1,D)}{T(m,mD)}$$

where $m$ is the number of computing node, $T(1,D)$ is the execution time of the tasks on 1 computing node with data size of D, $T(m, mD)$ is the execution time of the parallel tasks with $m$ computing nodes with data size $m$ times of D. A perfect parallelism demonstrates a constant scaleup with increasing number of computing node and data size.
Figure 20 shows the scaleup of the prototype with the number of computing nodes and input data size increasing at a factor of 2, i.e. 1, 2, 4, 8 and 16.

From the results, we can conclude that the MapReduce model exhibits very good scaleup at 0.95, 0.89 and 0.72 with 2, 4 and 8 computing nodes respectively. With a 16 computing nodes cluster, the scaleup is slightly lower, measured at 0.57. The reason for such scaleup degradation is due to the increase task setup time and communication cost with growing number of computing nodes, which is expected scenario in distributed computing. Nevertheless, this shows that MapReduce is still considerably scalable with growing data and growing number of computing nodes, a key consideration for data intensive applications.

5.2.3 Sizeup

Sizeup evaluates the ability of the parallelism to handle data growth [65]. It measures how much longer it takes to execute the parallel tasks, when the data size is \( n \)-times larger than the original dataset. Sizeup analysis holds the number of computing node constant and grows the size of the datasets by the factor \( n \). Sizeup can be expressed as follows:

\[
Sizeup(m, n) = \frac{T(m, nD)}{T(m, D)}
\]
where $m$ is the number of computing node and $n$ is the incremental factor of the data size. $T(m, D)$ is the execution time of the parallel tasks with $m$ computing node and data size $D$ and $T(m, nD)$ is the execution time of parallel tasks with $m$ computing node and with data size $n$ times of $D$.

Figure 21 shows the sizeup of the prototype. Each line corresponds to the sizeup behavior of a cluster of $m$ computing nodes with increasing data load at a factor of $n$.

The results show that system is able to handle data growth at all configurations. In fact, the sizeup decreases with increasing number of computing nodes. With a 16 nodes cluster at 32 times the initial data, the execution time increases only by a factor of 3.55. This shows that MapReduce is able to handle data growth gracefully, especially with larger number of computing nodes.

5.3 Cost Effectiveness of Cloud Computing

Through the experience of performing experiment on Amazon EC2, we have found that Cloud computing platform might pose to be a suitable approach for conducting
research-based experiment in terms of turn-around time and cost effectiveness. It is demonstrated that the required experiment environment can be easily set up within a relatively short time frame. On the cost effectiveness, in the case of this project, the major cost incurred for the experiment is also significantly lower than the basic cost of owning a cluster of physical desktop machines. Table 4 shows the basic cost of performing the project experiment on Amazon EC2 and a privately-owned cluster.

<table>
<thead>
<tr>
<th>Item Description</th>
<th>Unit Cost (US$)</th>
<th>Unit Needed/Consumed</th>
<th>Total Cost (US$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon EC2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Large Instance Type with Linux</td>
<td>0.38/hr</td>
<td>157 hrs</td>
<td>59.66</td>
</tr>
<tr>
<td>- Storage Cost</td>
<td>0.11/GB-Month</td>
<td>126.794 GB-Month</td>
<td>13.95</td>
</tr>
<tr>
<td>Privately-Owned Cluster*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Dell Inspiron Zino HD</td>
<td>478.99/pc</td>
<td>16</td>
<td>7663.84</td>
</tr>
</tbody>
</table>

(*figures are gathered from www.dell.com accessed on 17 Aug 2010)

Table 4: Basic Cost Comparison of performing experiment on Amazon EC2 and Privately-owned Cluster

Note that the cost of the privately-owned cluster presented here are just cost of computing nodes with similar computing capabilities. Additional cost such as network equipment, data centre facilities and management/maintenance are not included. From this illustration and the ease of setup described previously, it is not difficult to understand why Cloud computing has gained tremendous amount of interest lately.
6 Conclusion and Future Work

This project uses a real scenario found in the Life Sciences arena to demonstrate the feasibility of exploiting MapReduce model to accelerate the data mining process for automatic image annotations. A prototype system is implemented to transform a conceptual serial data mining workflow into a fully distributed process using MapReduce paradigm. The prototype performs distributed data mining process consisting of pre-processing with median filter, feature generation with wavelet transformation, feature selection with Fisher’s ratio and classification prediction with K-nearest neighbours. Experiments are performed on Cloud computing platform in order to study the distributed characteristic of MapReduce model when applied to a data mining scenario. Three performance metrics: speedup, scaleup and sizeup are used to measure the system performance and interesting results are obtained that entails detailed analysis.

The evaluation results show that the prototype exhibits satisfactory speedup, good scaleup and superb sizeup. A key observation made in the evaluation was that the input data size is an important factor in achieving reasonable speedup. This is due to the fact that for optimal parallelism, one has to ensure the effects of overheads (such as inter-machines communications) in the parallelism is minimised by the computation time; and in most cases, large input size maximises the computation time. The unexpected speedup performance leads to a detailed analysis of the results and it suggests that, although inter-machines communications and input size are factors in limiting ideal parallelism, the non-parallelisable portion in the individual computation task also plays a paramount role in limiting speedup. The evaluation also shows that Hadoop MapReduce exhibits good scalability with growth in both data size and number of computing nodes.

In addition, this project also verified that system development of a distributed data intensive application using Hadoop MapReduce framework can be performed with minimum infrastructure complexity. Furthermore, the easy setup and deployment of Hadoop cluster on Amazon EC2 has also largely simplified and lowered the overall cost of owning and managing a scalable distributed computing platform. Indeed,
accelerating data intensive application using MapReduce could be both a clever and strategic choice for many enterprises.

There exist many possible enhancements to the work performed in this dissertation. Although capable of performing real data mining tasks, the prototype can be enhanced, at least, in the following ways:

- Devise better algorithms to parallelise the currently non-parallel region found in the MapReduce job;
- Improve the computation time of individual MapReduce job through better algorithms or within-node parallelism;
- Improve I/O performance of the MapReduce job through customised Hadoop input and output type
Bibliography


