Cluster Comput (2015) 18:403–418
DOI 10.1007/s10586-014-0405-9

Parallel data intensive applications using MapReduce: a data mining case study in biomedical sciences

Liangxiu Han · Hwee Yong Ong

Received: 3 April 2013 / Revised: 17 June 2014 / Accepted: 24 September 2014 / Published online: 22 October 2014
© Springer Science+Business Media New York 2014

Abstract Performance is an open issue in data intensive applications (e.g. data mining tasks). Parallel and distributed computing systems (e.g. multicore computing, grid computing, cloud computing,etc.), along with hybrid programming models (e.g. MapReduce, MPI, etc.), is seen a sought-after solution for accelerating data-intensive applications. One of main challenges is how to exploit these advanced technologies effectively in facilitating fundamental science discoveries such as those in Biomedical Sciences. This paper explores how MapReduce and Cloud computing can accelerate performance of data intensive applications through a real data mining use case in the Biomedical Sciences. We have first adapted the data mining task using MapReduce model and then deployed it onto the Cloud. We have built an analytic model based on the MapReduce computations to evaluate the efficiency and performance of the prototype. The results, from both experiments and the evaluation model, show the performance and scalability can be enhanced through these advanced technologies.

Keywords Data-intensive computing · Parallel processing · MapReduce · Cloud computing · Data mining application in biomedical science

L. Han (✉)
School of Computing, Mathematics and Digital Technology, Manchester Metropolitan University, Manchester, UK
e-mail: l.han@mmu.ac.uk

H. Y. Ong
School of Informatics, University of Edinburgh, Edinburgh, UK
e-mail: hweeyong@gmail.com

1 Introduction

Data keeps on growing. According to the report from IDC (sponsored by EMC) [20], the size of all digital data, generated by social networks, sensor networks, and simulation devices, will reach 35 zettabytes by 2020. We have now entered an era of data-intensive science. Data intensive science represents a new paradigm of science discovery [19]. It interacts with“ big” data to identify patterns and lead to scientific breakthroughs.

Data-intensive applications are usually viewed as I/O bound or with a need to access and manipulate the large volumes of data [14,15] because they spend most of time on I/O operations.

In data-intensive computing, processing requirements normally scale super-linearly according to data size [15]. Large-size data needs to be decomposed into small chunks and spread across many computing resources to achieve I/O in parallel.

Parallel and distributed computing systems (e.g., multicore computing, grid computing, cloud computing, etc.), along with hybrid programming models (e.g., MapReduce, MPI, etc.,), have played important parts in accelerating data-intensive applications. Examples of systems that support data-intensive applications include Google MapReduce [29], Swift [38], DataCutter [5], DryadLINQ/Dryad [25,26], and Parallel databases such as Vertica [36], Teradata [35], IBM DB2 [9], etc. With the rapid growth of data, more compute resources are required to access large amounts of data and perform many calculations across multiple machines concurrently. However, incorporating data and compute resources into parallel infrastructures amenable to data exploration is not an easy task. One has to take into account scalability, reliability, fault-tolerance and cost-reduction. To remove the burden of building, operating and maintaining expensive

physical resources and infrastructures (e.g., hardware, clusters etc.), Cloud computing, a type of distributed computing paradigm augmented with a business model via a Service Level Agreement between providers and consumers [16], is emerging as a cost-effective solution to address the increased demand for distributed data, computing resources and services without large upfront investment.

One of main challenges is how to exploit these frontier technologies effectively to support fundamental science discoveries such as those in Biomedical Sciences. This paper attempts to use a real data mining study in the Biomedical Sciences and conduct an experimental evaluation to better understand how well the parallel approach (i.e., MapReduce in this paper) and Cloud computing support data-intensive applications in real life. Note the data mining use case here is a typical exemplar of a generic image-processing analysis pipeline that could be applied to many datasets. This type of intensive data-analysis task is increasingly common in the physical, life and social sciences (e.g., scatterometer/satellite image data for climate study). It is vital for researchers in these areas that they have access to the methodologies that can automatically transform data into knowledge, and that are also dynamic scalable in conjunction with the steady increase in available raw storage and computational resources, and corresponding improved capability to increase the operational, cost, and environmental efficiency. The contributions of this paper include:

- Adapting the data mining use case using the MapReduce Model;
- Constructing an analytic model based on MapReduce computations for assessing the efficiency of the prototype and providing the possible optimisation strategies based on the model;
- Deploying the application to the Cloud and evaluating the performance on the Cloud computing platform;
- Providing a best practice in supporting a class of similar data-intensive applications in the cloud.

The rest of this paper is organised as follows: Sect. 2 overviews related technologies that support data-intensive applications; Sect. 3 presents a prototype of data mining case in the Biomedical Sciences using the MapReduce model; in Sect. 4, we have constructed an analytic model based on the MapReduce computations for performance evaluation and conducted experiments in the cloud; Sect. 5 discusses the experimental results, experiences and possible optimisation strategies; Sect. 6 concludes the proposed approach and highlights our future work.

2 Related work

Parallel computing is a typical way to improve the performance of data-intensive applications. Parallelisation can take place at either hardware or software levels or both (e.g., signal, circuit, component and system levels) [6]. Hardware parallelism focuses on signal and circuit levels and normally is constrained by manufacturers. Software parallelism at component and system levels can be classified into two types: automatic parallelisation of applications without modifying existing sequential applications and construction of parallel programming models using various software technologies to describe parallel algorithms and then match applications with the underlying hardware platforms. Since the nature of auto-parallelisation is to recompile a sequential program without the need for modification, it has a limited capability of parallelisation on the sequential algorithm itself. In most cases, it is hard to directly transform a sequential algorithm into a parallel one. While parallel programming models try to address how to develop parallel applications and therefore how to maximally utilise the parallelisation to obtain high performance, it does need more development effort on parallelisation of specific applications. In general, three considerations when parallelising an application include:

- How to distribute workloads or decompose an algorithm into parts as tasks?
- How to map the tasks onto various computing nodes and execute the subtasks in parallel?
- How to coordinate and communicate subtasks on those computing nodes?

There are mainly two common methods for answering the first two questions: data parallelism and task parallelism. Data parallelism represents workloads are distributed into different computing nodes and the same task can be executed on different subsets of the data simultaneously. Task parallelism represents the tasks are independent and can be executed purely in parallel. There is another special kind of the task parallelism is called pipelining. It represents an iteration of a task consisting of many stages, where each stage in the task is chained and executed in order and the output of one stage is the input of the next one. Pipelining can be implemented with streaming and without using streaming. In the pipeline streaming mode, multiple stages of a task are connected and executed in series. One stage can start computing with partial input data without waiting for the whole input data. Similarly, the successor of a stage can process the partially processed result and so on. The extent of parallelisation is determined by the dependencies of each individual part of the algorithms and tasks. As for the coordination and communication among tasks or processes on various nodes, it depends on different memory architectures (shared memory or distributed memory). A number of communication models have been developed [28,30], for example, the Message Passing Interface (MPI).

Much effort has been devoted to developing frameworks using parallel approaches to support data-intensive applications.

Google MapReduce framework [10,29] focuses on data parallelism. It provides a simple interface of two functions and allows developers to parallelise data processing tasks. The *Map* function performs grouping that produces intermediate data sets and the *Reduce* function performs aggregation of intermediate data sets into smaller data sets. Hadoop MapReduce [2] is an open-source MapReduce implementation that is an attempt to reproduce Google MapReduce. The simplicity of MapReduce is appealing. It can easily parallelise data over large-scale data centres with thousands of computing nodes and process data on terabyte and petabyte scales. The MapReduce restricts its two-stage operations in order. The developers need adapt their applications to the MapReduce style by developing their own *Map* and *Reduce* functions. Dryad [25,26] is a general purpose distributed execution engine for coarse-grained data parallel applications, designed for scaling from multicore single computers, through small clusters of computers, to data centres with thousands of computers. The model of computation of the Dryad is a data flow framework expressed as a DAG and inherently specialised for streaming computations. It can specify arbitrary DAGs. FREERIDE [21,22] mainly focuses on development of parallel versions of specific and well-known data mining algorithms. River [3] is a data-flow programming environment and I/O substrate for clusters. It uses data parallelism and limits itself to I/O workloads. Swift [38] is a system that supports parallel computation on large-scale data-intensive applications and focuses on data parallelism. A data diffusion framework [32], built on Falkon [31], has been proposed for supporting large-scale data exploration that acquires computing and storage resources dynamically, replicates data in response to demand, and schedules computations close to data. Grossman et al. [33] have developed a Cloud-based infrastructure for data mining on large distributed datasets. The approach moves the processing near to the data. ADMIRE [4,18] is a data mining and integration framework that adopts streaming the data flow graph and supports both data and task parallelisation. Taverna [27] is a framework for composition and enactment of bioinformatics workflows. It provides an implicit iteration for processing data. Taverna can map a high-level task to a single entity with a minimum amount of implementation information. It supports both task and data parallelisms. Pegasus [11] and DAGman [7] are suitable for managing complex scientific workflows and scheduling large-scale computations in Grid environments. DAGMan provides a workflow engine that can organise Condor tasks as DAGs and deal with parallelism between tasks. Triana [34] offers support for both data and implicit control flows. Loops and execution branching are handled by specific components. It supports static data parallelisms. Karajan [23,24] extends DAGs for supporting conditional execution, sequential and parallel iterations. It adopts the underlying Grid tools for job submission.

Cloud computing, a recent advancement, has evolved from existing parallel processing, grid computing, distributed computing and utility computing technologies. It provides dynamic and flexible resource provision and delivers infrastructure, platforms and software as a service in order to support data-intensive applications [37].

To enable scientists to gain new insights from large amounts of data, it is critical to understand how well these advanced technologies can support data-intensive applications in practice such as in the Biomedical Sciences.

In this paper, we mainly focus on applying the MapReduce and Cloud computing technologies to a real data mining use case in the Biomedical Sciences. Two main reasons for choosing the MapReduce and Cloud computing include:

- The MapReduce framework is designed for parallel processing large-scale data efficiently and to be run on non-specialised commodity hardware. It is easy to use, scalable and fault-tolerant.
- Cloud computing enables on-demand access to computing resources for scaling up to match the growing data. The entry level of Cloud computing is lower than operation of traditional IT infrastructure, e.g., without need of large capital investment and complex resource provision.

For the purpose of this paper, we have first adapted the data mining algorithms described in Sect. 3.1 using the MapReduce model. We then used Cloud computing (based on the IaaS model) as a deployment platform on which the data mining application is deployed and executed. Namely, we rent computing infrastructures from a cloud provider (i.e., Amazon Elastic Compute Cloud (Amazon EC2 [1]) with full control of those computing resources and conducted experimental evaluation in the Cloud.

3 Parallel data-intensive application using MapReduce: a data mining case study

3.1 Background

It is of high biomedical interest to identify gene interactions and networks that are associated with developmental and physiological functions in the embryo by using ontological annotation. The ontological annotation of gene expressions consists of labelling embryo images produced from RNA in situ Hybridisation (ISH) with terms from the anatomy ontology for mouse development. If an image is tagged with a term, it means that anatomical component shows expression of that gene. Currently annotation is done manually

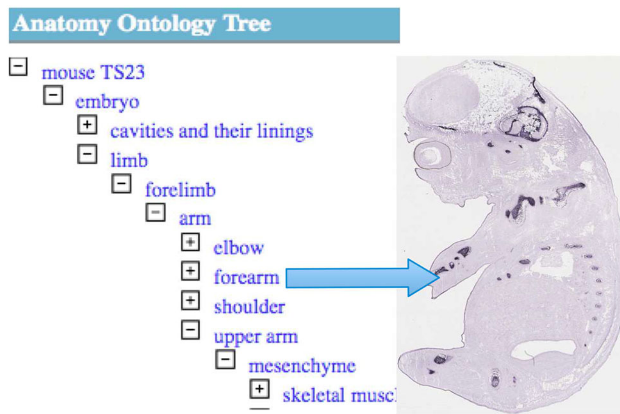


Fig. 1 An example of annotating image using a term from the anatomy ontology for the developing mouse embryo

by domain experts. This is both time-consuming and costly, especially with the rapidly growing volume image data of gene expression patterns on tissue sections produced by advanced high-throughput instruments (e.g., ISH). For example, the EurExpress-II project [5] curated over 4 Terabytes large datasets (still continuing growing, an additional further 10 TB) including images for the developing mouse embryo and the ontological terms for anatomic components of the mouse embryo. The datasets provide a powerful resource for discovery of potential mechanisms of embryo organisation.

Our data mining goal is therefore to automate this annotation process. The input is a set of image files (stored in the file system) and corresponding ontological terms (stored in the database). The output will be an identification of the anatomical components that exhibit gene expression patterns in each image. Figure 1 shows an example of annotating an image using a term from the anatomy ontology for the developing mouse embryo. Figure 2 illustrates the processes of the data mining task involved, for example, the processes in the training stage include integration of images and annotations, image processing, feature generation, feature selection and extraction, and classifier design. The testing stage has three same subprocesses (image processing, feature generation, feature selection and extraction) with additional prediction evaluation process. For the prediction evaluation process, we adopt 10-fold cross validation to validate the accuracy of the classifiers, where the dataset is randomly divided into 10 subsets. 9 subsets are formed as a training set and one is viewed as a test set. This process is then repeated ten times. The deployment stage contains only the application of the classifiers. We have developed data mining algorithms for each of the processes that can automatically annotate gene expression in images [17], programmed these in sequential code, and have conducted a pilot experiment.

Note this use case here is an exemplar for a wide range of analysis that can be applied to many other datasets. This type of large-scale, high-throughput image-based data is now common across biomedical research and beyond and it will be a major resource for data mining over many years. The computations are often compute-intensive as well as being across a large data-set and users will need the capability to write their own algorithms to run against the data. This capability of moving the computation to the data is difficult to support and unlikely to be within the resources of the data-providers. The purpose of this paper is therefore to explore the use of parallel approaches such as the MapReduce in this case and Cloud computing to enable this capability.

3.2 MapReduce

Google MapReduce is inspired by functional languages and targeted at data-intensive applications [10,29]. The key benefit given by the MapReduce is that a programmer does not need to deal with the complicated code parallelism and focus on the required computation. It emphasised on the data parallelism. The proprietary implementation of Google MapRe-

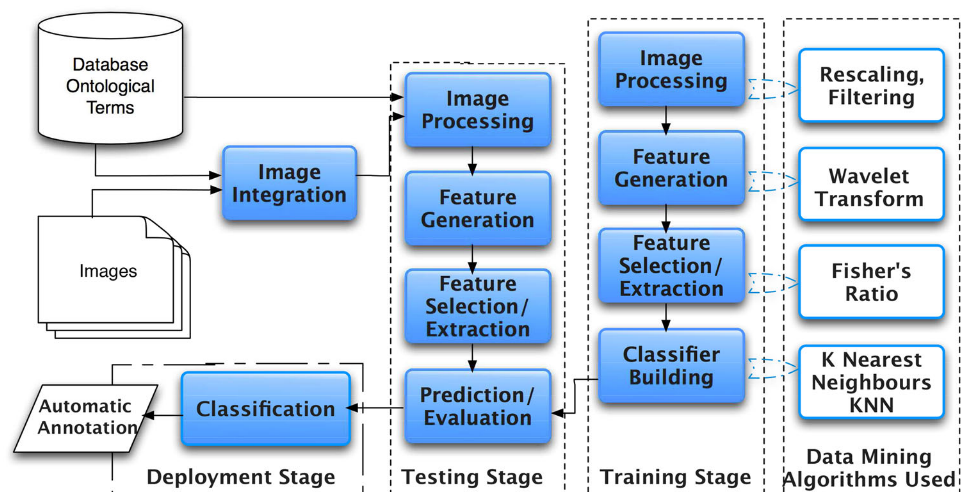


Fig. 2 Data mining processes for automatic gene expression annotation

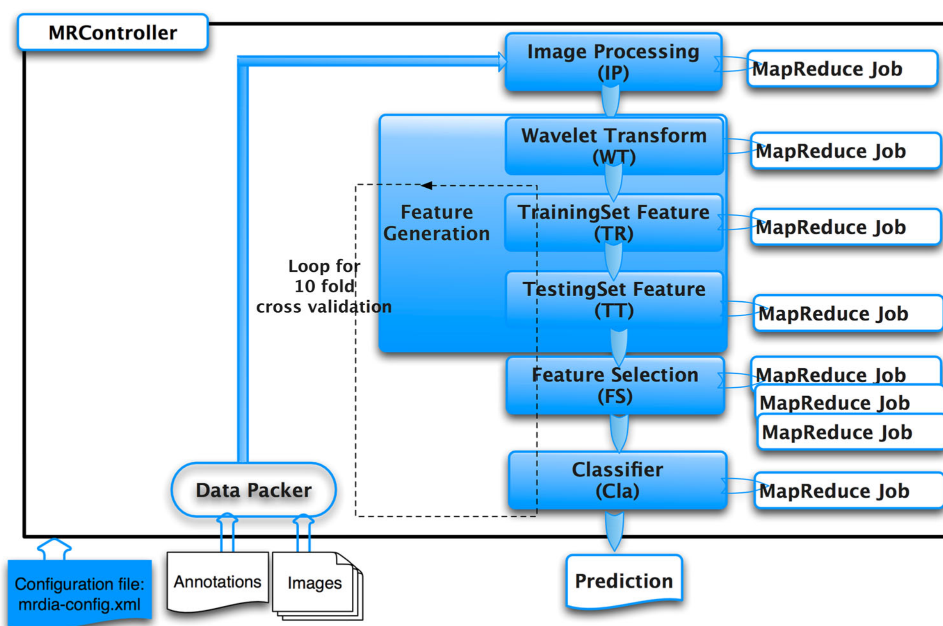


Fig. 3 Prototype of automatic gene expression annotation on Hadoop

duce framework has prompted the development of similar implementations [2,12]. Hadoop MapReduce is one of the most popular open source implementation. It is a reproduction of Google's MapReduce, written in Java. It is run on top of the Hadoop Distributed File System (HDFS), which provides high throughput access to the reliable storage of very large files across machines in a large cluster and enables collocation of data processing and storage.

In Hadoop MapReduce, the format of the input is application-specific and the output is a set of $\langle key, value\rangle$ pairs. The programmer expresses the desired computation as two primitive functions: *Map* and *Reduce*. The *Map* function consumes the input and produces a list of intermediate $\langle key, value\rangle$ pairs. The Hadoop MapReduce runtime then shuffles these intermediate $\langle key, value\rangle$ pairs and groups together all the values associated with the same key and partitions these groups of $\langle key, value\rangle$ pairs based on a hashing function on keys accordingly. The *Reduce* function takes the groups of $\langle key, value\rangle$ pairs and performs a merging operation on the list of intermediate values associated with the same key and produces zero or more output $\langle key, value\rangle$ pairs. There could be multiple instances of the Mapper and Reducer running on different machines with each processing a subset of larger data concurrently. In both *Map* and *Reduce* stages, the runtime must dynamically decide the size of data partitions, the number of computing nodes, the assignment of data partitions to nodes, and the allocation of memory buffer space. These decisions can be either implicitly decided by the runtime based on some default settings or explicitly specified by the programmer via APIs or configuration files.

In the Hadoop system, a MapReduce job represents a work unit including input data, *Map* and *Reduce* programmes and corresponding user-specified configurations. It is possible to create a MapReduce job without a Reduce phase. It is important to note that it is difficult to solve any problem with a single MapReduce Job and large-scale data-intensive problems can be handled with a series of interconnected MapReduce job, called job chaining, which allows to decompose a large problem to smaller problems.

3.3 Implementation of the data mining task using the MapReduce

According to Fig. 2, we have implemented processes of the data mining task using the MapReduce and prototyped the system using the Hadoop MapReduce (version 0.20.2). Figure 3 shows system overview of the prototype. Three main components including MRcontroller, Data Packer and the data mining workflow consisting of 8 MapReduce jobs. Figure 4 shows detailed Mappers and Reducers of the MapReduce job chain.

The MRcontroller, along with a configure file called mrdia-config.xml, is an interface that allows a user to interact with the system, for example, the user can specify input/output file locations through MRcontroller. The runtime properties of the system can be configured through the MRcontroller reading the mrdia-confi.xml file. Some runtime properties are the number of computing nodes, the k value for the k-fold cross validation, and other parameters that are used in the data mining algorithms.

The Data Packer is responsible for packing the raw data in the desired file format and storing them in the HDFS (Hadoop Distributed File System) for the MapReduce jobs. In Hadoop, there are two kinds of file formats: *TextFile* and

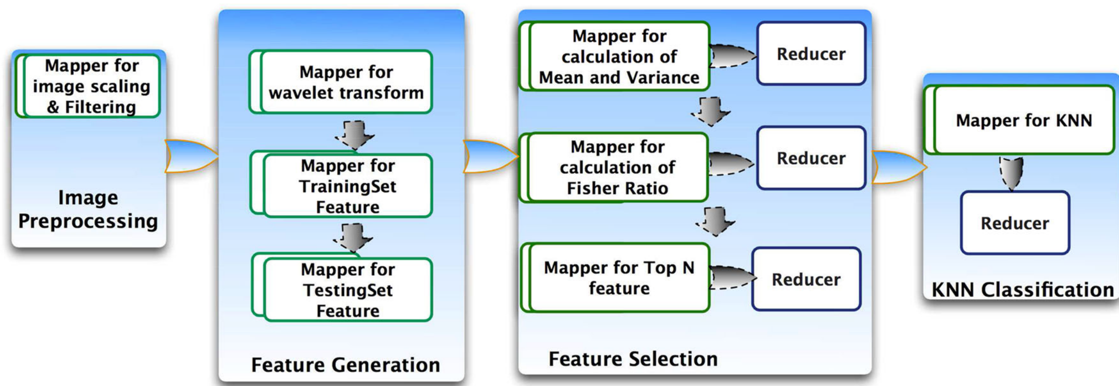


Fig. 4 MapReduce job chain in detail

SequenceFile. *TextFile* format is for unformatted data or line-based records like log files. *SequenceFile* is a persistent data structure used in Hadoop for storing binary. The SequenceFileInputFormat reads special binary files that are specific to Hadoop, allowing data to be rapidly read into Hadoop Mappers. 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 and provides direct serialisation and deserialisation of several data types (such as byte array). It 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. Binary files are stored using *SequenceFile* format in this case. Data packer also acts as an initial workload distributor, produces the number of smaller files that match with the number of computing node as instructed by the MRcontroller (which reads the value from the configuration file, mirdia.xml).

Another 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 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 following subsection discusses the details MapReduce implementation for these MapReduce jobs.

3.3.1 Image preprocessing

The image preprocessing deals with de-noise and standardisation of images due to different dimensions and variations. It reads an image and the corresponding annotation and processes with rescale and denoise algorithms (please refer to [17] for algorithm details). Given a large dataset, the algorithms can have multiple instances running on partitioned subsets without any internal status to be maintained between these instances. This nature makes the algorithms and this process be easily adapted to the MapReduce style. It is thus implemented as a Mapper class in Hadoop and there is no need for a Reducer at this stage. In this case, the individual image binaries and its annotations are stored into a java object together with a generated identifier. The list of java objects that represents all the images are then written into an optimised format known as *SequenceFile* in Hadoop. The Mapper takes the *SequenceFile* containing a list of image binaries as input and applies the denoise and rescale algorithms on every image. The output after applying denoise and rescale algorithms is a java object binary with a 2-D array (the matrix representation of an image) and associated with the unique image identifier to form the $\langle key, value\rangle$ pair in the output SequenceFile.

Mapper (for image preprocessing):
Input: SequenceFileInput (key(imageId), ImageBinarydata); Output: SequenceFileOutput (key(imageId), 2-D Byte array)

3.3.2 Feature generation using wavelet transform

The feature generation includes three MapReduce jobs: wavelet transformation of an image to produce the features, feature for training set, and feature for testing set. We have used wavelet transform method to generate the features. This task is suitable for data parallelism because the instances can be running on different machines without communications between machines. It is implemented as a Mapper class. There is no need for Reduce class. The Map function will be invoked for every $\langle key, value\rangle$ pairs in the output *SequenceFile* produced by image preprocessing and performs 2-D discrete wavelet transformation on it (the key is the unique image identifier and the value is a java object binaries with a 2-D

array, image representation using Matrix). The output is a java object binaries stored in output SequenceFile.

Mapper (for wavelet transform):
Input: SequenceFileInput (key(imageID), value(Matrix representation:2-D ByteArray)); Output: SequenceFileOutput (key(imageID), value(feature vector:1-D ByteArray))

3.3.3 Training and testing set features

Since 10-cross validation is used in this case for validating data mining algorithms, we need split the data into the training set and the testing set and thus have the training set features and the testing set features.

The 10-cross validation in the system is implemented as a for-loop iteration, which 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 and training sets are generated dynamically in each iteration based on simple hashing of the unique identifier assigned to the data item through the data packer. The testing and training sets are stored in different directories in the HDFS in *SequenceFile* format.

We have implemented Mappers for the training and testing set feature extraction in order to perform parallel execution in parallel. Technically, the testing and training feature extraction can be performed in a single task, but they are implemented as two separated Map classes. The reason is that Hadoop MapReduce job produces a single list of output. The Mapper for the training set features is shown below(similarly, we have the same Mapper function for the testing set features). The input to the Mapper includes *SequenceFileInput* embedded with key and the corresponding value and the configuration file containing numbers of folds.

Mapper (for Training set/or Testing set):
Input: SequenceFileInput (key(imageID), value(feature vector:1-D ByteArray)); and ConfigurationFile for getting numbers of fold(i.e. k); Output: SequenceFileOutput (key(imageID), value(feature vector:1-D ByteArray))

3.3.4 Feature selection using Fisher's ratio

The nature of the Fishers Ratio algorithm [13] for feature selection is to calculate Mean and Variance of image samples for two classes (C1 and C2), as shown in the following Eq. (1).

$$FisherRatio = \frac{(m_{1,i} - m_{2,i})^2}{(v_{1,i}^2 + v_{2,i}^2)} \quad (1)$$

where $m_{1,i}$ represents the mean of samples at the ith feature in C_1, $m_{2,i}$ represents the mean of samples at the ith feature in C_2. $v_{1,i}$ represents the variance of samples at the ith feature in C_1. Similarly, $v_{2,i}$ represents the variance of samples at the ith feature in C_2.

We parallelise Fisher's Ratio algorithm by decomposing it into three parts: calculation of mean and variance, Fisher's ratio calculation and Top N feature index Selection. The input to this step is the training set extracted in the feature generation step. We thus implemented these three parts as three MapReduce jobs.

(1) MapReduce job for calculation of mean and variance
In the first step of the Fisher's Ration algorithm, we need compute the mean and variance of features of the two classes respectively (images labelled as "1" represent one class and images labelled "0" represent the other class). This computation can be fully parallelised by calculating the means and variance of all image features belonging to the same class. This job has a Map class and a Reduce class. The Mapper maps images according to the classification using the classification labels (i.e., "1" or "0") as the key for its output ⟨key, value⟩. The value is the vector containing the features of images (from feature generation step). The Mapper class input and output can be represented as:

Mapper (for calculation of Mean and Variance):
Input: SequenceFileInput (key(imageID), value(feature vector:1-D ByteArray)); Output: SequenceFileOutput (key (classLabel), value(feature vector:1-D ByteArray))

The Reducer receives the list of features of each image belonging to the same class and performs the calculation of mean and variance of the features across the list using Eqs. (2) and (3).

$$m_{mean} = \frac{\sum_{i=1}^{n} x_i}{n} \quad (2)$$

$$v_{variance}{}^2 = \frac{\sum_{i=1}^{n} (x_i - m_{mean})^2}{n} \quad (3)$$

The calculation of both mean and variance is highly parallelisable. There are only two Reducers needed since

there are only two distinct keys at this stage, i.e., the classification label "1" and "0". The Reducer class input and output can be represented as:

Reducer (for calculation of Mean and Variance):

Input:
SequenceFileInput (key (ClassLabel),
value(feature vector:1-D ByteArray));
Output:
SequenceFileOutput (key (ClassLabel),
value(feature vector:1-D ByteArray))

(2) MapReduce job for Fisher's Ratio
The next MapReduce job is to calculate the Fiher's Ratio between the two classes for every feature of an image in a class. The job comprises a Mapper and a Reducer. The Mapper "rearranges" the data as ⟨key, value⟩ pairs 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 features since the internal shuffling will hash the key (which is the feature index now) and distribute the workload across all the available Reducers. The Reducers perform the actual calculation of the Fisher's Ratio described in Eq. (1).

Mapper (for Fisher's Ratio):

Input:
SequenceFileInput (key (ClassLabel),
value(1-D ByteArray));
Output:
SequenceFileOutput (key(Featureindex),
value(1-D ByteArray))

Reducer (for Fisher's Ratio):

Input:
SequenceFileInput (key (Featureindex),
value (ClassLabel, mean and variance));
Output:
SequenceFileOutput (key(Featureindex),
value (FisherValue))

(3) MapReduce job for Top N feature index selection
The Top N feature index selection is to select the most significant features to be used in the classification step. This job has a Map class and a Reduce class. The Mapper combines all the Fisher's ratio of the features to a single list using a common key (which is arbitrarily assigned). The Reducers accepts the list of (FeatureIndex, FisherValue), sorts the list in descending order according to the

Mapper (for TopN Feature Index):

Input:
SequenceFileInput (key(FeatureIndex),
Value (FisherValue));
Output:
SequenceFileOutput (key(some-key: arbitrarily assigned),
value(FeatureIndex, FisherValue))

Fisher's ratio, 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.

Reducer (for TopN Feature Index Selection):

Input:
SequenceFileInput (key(some-key: arbitrarily assigned),
value(FeatureIndex, FisherValue));
and Configuration file for getting the value of N
Output:
SequenceFileOutput (key(somekey),
value(Indices:integer array)

3.3.5 Classifier construction using K-nearest neighbour

In this case, we have adopted K-Nearest Neighbour (KNN) [8] algorithm for classification to identify unknown samples into a class based on the nearest distance with the training samples. The common distance function is Euclidean distance. In this case, the samples (images) are represented as vectors of the values of the selected features based on Top N index as vectors. The Euclidean distances therefore are calculated between each training sample in the training set and a testing sample in the testing set, and then the nearest ones can be chosen. For all of training samples, the calculation is a typical iteration task. To exploit parallelisation in this algorithm, we use a special task parallelism called "pipelining", inspired by [6]. We divide an iteration of the KNN task into several pipeline stages. The sample images can be partitioned into subsets. The distance calculations between subsets of training samples and the unclassified testing sample are executed at different stages. Figure 5 shows the parallel form of the KNN algorithm using the concept of "pipelining".

We have implemented this parallel KNN as a MapReduce job comprising a Mapper, a combiner and a Reducer. According to the parallel form of KNN, it requires three inputs: the training set, the testing set and Top N indexes for selected features. However, the Hadoop MapReduce job requires the input to be a list of $\langle key, value \rangle$ pairs with identical format. Therefore, we use the training set only as the direct

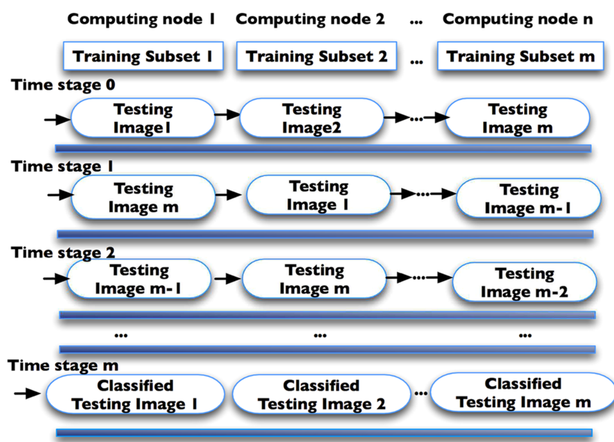


Fig. 5 A parallel form of KNN

input to the Mapper with the testing set and Top N feature indexes dynamically distributed to all the computing nodes using the *DistributedCache* provided by Hadoop. The testing set and Top N features are retrieved and kept in memory during the initiation of the Mapper. The in-memory testing set is a reduced set because the feature is selected based on the Top N feature indexes. This in-memory test is used exclusively in the Mapper class because we need to compute the Euclidean distance between each training sample and testing sample. The output is to predict the classification of each image in testing set and the key for the $\langle key, value\rangle$ pair is the unique identifier of image sample and the image file name. The value of the $\langle key, value\rangle$ pair is a data structure containing information (such as unique identifier, classification label and Euclidean distance from the testing set) of the "neighbour" found in the training set. The Euclidean distances are thus calculated concurrently across all computing nodes in the Map phase.

Mapper (for KNN):

Input:
SequenceFileInput (key(ImageID),
Value (1-D ByteArray));
Output:
SequenceFileOutput (key(A customised key containing imageID and fileName),
value(ImageID, ClassLabel, Euclidean Distance))

The Reducer receives every testing image sample and its neighbours and then sorts all the neighbours in ascending order according to the distance and picks the K nearest neighbours from the list. The Reducer output 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 image sample, the Reducer will only receive the K nearest neighbours from each Mapper.

Reducer (for KNN):

Input:
SequenceFileInput (key(A customised key containing imageID and fileName),
value(ImageID, ClassLabel, Euclidean Distance))
Output:
SequenceFileOutput (key(A customised key containing imageID and fileName),
value(Prediction of each sample the testing set))

4 Performance evaluation model

4.1 Performance evaluation metrics

We have used Speedup to measure performance. The definitions are described as follows:

$$Speedup = \frac{T(s)}{T(m)} \tag{4}$$

where $T(s)$ represents the execution time of a task on one single machine; $T(m)$ represents the execution time on m machines.

4.2 Performance evaluation model

In terms of the system architecture (i.e. MapReduce computations in this case) in Fig. 3, we have also built an analytic model to assess the efficiency of the prototype based on the following assumptions:

- Given m machines, the system in our case is configured as one single master and *m-1* multiple workers. The master consists of a job tracker, task tracker, namenode and datanode. The worker node consists of a datanode and task tracker. The master machine processes both the MapReduce and non-MapReduce jobs such as scheduling. The master node's time cost includes scheduling time, communication between the master and work nodes and MapReduce jobs acting as one worker, while the worker nodes' time cost consists of processing time for MapReduce jobs and communication time between the master and themselves.
- The data amount represented as D; all other time cost irrelevant to data amount and the number of work nodes represented as c_i, where i represents a certain task/or MapReduce job.

- The processing time on a work node is proportional to the amount of data (i.e., numbers of images).
- The processing capability per unit (i.e., per image in this case) is constant.

According to the assumptions above, the master node has the heaviest load and spent a longer time on both non-MapReduce jobs (e.g., scheduling, etc.) and MapReduce jobs while worker nodes only execute MapReduce jobs. Hence, the relationship between the total execution time of the data mining task and time cost for both the master and the workers can be represented as follows:

For a single machine, it is viewed as the sum of the sequential job of each process of the data mining task in Eq. (5).

$$T_s(D) = T_{IP} + T_{WT} + T_{TT} + T_{TR} + T_{MR} + T_{FR} + T_{TopN} + T_{KNN} + T_{datapack} + C_s \quad (5)$$

In the equation, all other processes can be represented as $k_i \times D + c_i$, where i represents a certain process of the data mining task (e.g., image preprocessing) except for classification KNN. k_i is the proportional coefficient, representing the processing time per unit. KNN algorithm needs to calculate Euclidean distance between each testing sample and each training sample's distance and requires both training and testing sets. Therefore, it can be represented as $k_{knn} \times D_{training} \times D_{testing} + c_{knn}$. In our case, we used n-fold cross validation, which means $D_{training}$ is n-1 times of $D_{testing}$. Therefore, time cost running on a single machine can be represented as

$$T_s(D) = A'_1 \times D^2 + A'_2 \times D + A'_3 \quad (6)$$

For multiple machines running the whole data mining task ($m \geq 2$), time cost (T_m) on the master includes time for processing non-MapReduce jobs ($T_{nmrjob}(D, m)$) in relation to data amount (such as data partition, data transferring) and the number of workers (such as scheduling, etc.) and time cost (C_m) for any other tasks that are irrelevant to data amount and the numbers of workers. To simplify the model, we only consider time cost related to data amount. Based on the experimental setup in this case, $T_{nmrjob}(D, m)$ can be viewed as time $T_{datapack}(D)$ for data packing step.

$$T_m(D, m) = T_{mrjob}(D, m) + T_{nmrjob}(D, m) + C_m \quad (7)$$

Therefore the performance model can be represented as a set of equation as follows:

$$\begin{cases} T_m(D, m) = T_{mrjob}(D, m) + T_{datapack}(D) + C_m \\ T_{mrjob}(D) = T_{IPmr} + T_{FGmr} + T_{FSmr} + T_{CLAmr} \\ T_{IPmr} = T_{IPmap} \\ T_{FGmr} = T_{WTMap} + T_{TrainMap} + T_{TestMap} \\ T_{FSmr} = T_{MVMap} + T_{MVRed} + T_{FRMap} \\ \qquad + T_{FRRed} + T_{TopNMap} + T_{TopNRed} \\ T_{CLAmr} = T_{CLAMap} + T_{CLARed} \\ T_{datapack}(D) = k_{datapack} \times D + C_{datapack} \end{cases} \quad (8)$$

where $T_{datapack}(D)$ represents time for data packer module in the prototype T_{IPmr}, T_{FGmr}, T_{FSmr}, T_{CLAmr} represent MapReduce jobs for image preprocessing, feature generation, feature selection and classification respectively. We will describe the time cost for each of MapReduce jobs below.

(1) Image preprocessing
This process is implemented as a MapReduce job (a Mapper for image rescaling and filtering, no Reducer). The execution time can be represented as follows:

$$T_{IPmr} = k_1 \times \frac{D}{m} + C_{IP} \quad (9)$$

where k_1 is the proportional coefficient, representing the processing time per unit; $k_1 \times D$ means the processing time; m represents the number of computing nodes. C_{IP} represents all other time costs independent of data amount and the number of work nodes represented.

(2) Feature generation
Feature generation aims to produce features by applying wavelet transform to images and then splitting this into training and testing sets. This process is implemented as three MapReduce jobs (all are Mappers without need for Reducers). Therefore, the execution time can be described in Eq. (10).

$$T_{FGmr} = k_{2a} \times \frac{D}{m} + k_{2b} \times \frac{D}{m} + k_{2c} \times \frac{D}{m} + C_{FG} \quad (10)$$

where k_{2a}, k_{2b}, k_{2c} are the proportional coefficients, representing the processing time per unit; $k_{2a} \times D$ means the processing time for wavelet transform; $k_{2b} \times D$ means the processing time for feature extraction of training set and $k_{2c} \times D$ means the processing time for feature extraction of testing set. m represents the number of computing nodes. C_{FG} represents all other time costs independent of data amount and the number of work nodes represented.

(3) Feature selection
Feature selection has been implemented as three MapReduce jobs with both Mappers and Reducers: calculation of mean and variance, calculation of Fisher's Ratio and Top N features. We describe time cost for feature selection in Eq. (11). The execution time for each of these Mappers and Reducers is given in Eqs. (12), (13), (14), (15), and (16).

$$T_{FSmr} = T_{MVMap} + T_{MVRed} + T_{FRMap} + T_{FRRed} + T_{TopNMap} + T_{TopNRed} \quad (11)$$

T_{MVMap} is time cost for the mapper that calculates the mean and variance. T_{MVRed} is time cost for the Reducer that calculates mean and variance for the two classes ("1" and "0"). They can be represented as in Eq. (12).

$$T_{MVMap} = k_{3map} \times \frac{D}{m} + C_{MVmap}$$
$$T_{MVRed} = k_{3red} \times \frac{D}{m'} + C_{MVred} \quad (12)$$

where m' represents the number of machines to process reduce jobs. This step only need two Reducers for two classes ($m' = 2$)
T_{FRMap} described in Eq. (13) is time cost of Fisher's Ratio, which is similar to other mappers. T_{FRRed} is time cost of the Reducer that calculates Fisher value. It is determined by the number of features F (a constant based on image dimension) instead of the number of images (i.e., Data size D in this case). It can be represented as in Eq. (14).

$$T_{FRMap} = k_{4map} \times \frac{D}{m} + C_{FRmap} \quad (13)$$

$$T_{FRRed} = k_{4red} \times \frac{F}{m'} + C_{FRred} \quad (14)$$

where m' represents the number of machines to process reduce jobs and $m' = 2$.
$T_{TopNMap}$ represents time cost of TopN that combines all Fisher's Ratio of features into a single list, which can be calculated based on Eq. (15). The Reducer sorts the list and selects Top N features and only needs one single machine to process the reduce job. Therefore the time cost $T_{TopNRed}$ is described in Eq. (16). Only one Reducer is needed in this case.

$$T_{TopNMap} = k_{5map} \times \frac{F}{m} + C_{topnmap} \quad (15)$$

$$T_{TopNRed} = k_{5red} \times F + C_{topnred} \quad (16)$$

(4) KNN classification
KNN classification deals with the calculation of Euclidean distance between each testing sample and each training sample. It has three inputs: training dataset, testing dataset and TopN indexes. The proportion of data size between testing and training dataset is 1:9 because we use 10-folder validation. KNN Mapper obtains training samples and output the distance values. Time cost for KNN Mapper T_{CLAMap} is described in Eq. (17). The Reducer is to determine the classification of images based on the distance. It accepts the nearest neighbours from m Mappers. It can be processed on one single machine. The time cost for the Reducer T_{CLARed} is represented in Eq. (18).

$$T_{CLAMap} = k_{6map} \times \frac{D_{training}}{m} \times D_{testing} + C_{clamap} \quad (17)$$

$$T_{CLARed} = k_{6red} \times m \times D_{testing} + C_{clared} \quad (18)$$

In terms of analysis of time cost above, the execution time for the data mining task $T_m(D)$ can be described in Eq. (19)

$$\begin{aligned} T_m(D) = {} & T_{IPmr} + T_{FGmr} + T_{FSmr} + T_{CLAmr} \\ & + T_{nmrjob}(D) + T_{nmrjob}(m) + C_m \\ = {} & k_1 \times \frac{D}{m} + C_{IP} + k_{2a} \times \frac{D}{m} \\ & + k_{2b} \times \frac{D}{m} + k_{2c} \times \frac{D}{m} + C_{FG} \\ & + k_{3map} \times \frac{D}{m} + C_{Meanvarmap} \\ & + k_{3red} \times \frac{D}{m'} + C_{Meanvarred} \\ & + k_{4map} \times \frac{D}{m} + C_{FisherRatiomap} \\ & + k_{4red} \times \frac{F}{m'} + C_{FisherRatiored} \\ & + k_{5map} \times \frac{F}{m} + C_{topnmap} \\ & + k_{5red} \times F + C_{topnred} \\ & + k_{6map} \times \frac{D_{train}}{m} \times D_{test} + C_{clamap} \\ & + k_{6red} \times m \times D_{test} + C_{clared} \\ & + k_{datapack} \times D + C_m \end{aligned} \quad (19)$$

The above equation can be unified in Eq. (20)

$$T_m(D, m) = A_1 \times \frac{D^2}{m} + A_2 \times D \times m + A_3 \times \frac{D}{m} + A_4 \times D + A_5 \frac{1}{m} + A_6 \quad (20)$$

where A_1, A_2, A_3, A_4 and A_5 are the proportional coefficients by combining like terms. A_6 is the constant independent of data size and amount. This is a non-linear equation with six parameters to be determined based on our practical experiments in the Cloud.

4.3 Experimental evaluation in the cloud

We have conducted the real experiments based on the Cloud computing platform. We deploy our prototype in the Amzon IaaS cloud by renting virtual computer nodes (Large Instance

Table 1 Configuration of a virtual machine from Amazon EC2

Item	Configuration
CPU	4 EC2 Compute units: 2 virtual cores with 2 EC 2compute Units. Note: one EC2 Compute unit provides the equivalent CPU capacity of a 1.0–1.2 GHz 2007 Opteron or 2007 Xeon processor.
Memory	7.5 GB memory
Storage	850 GB instance storage (2 × 420 GB plus 10 GB root partition)
I/O performance	High
Operating system	Federa Core 8 (2.6.21.7-2.ec2.v1.2.fc8xen Linux Kernel)

Type) from Amazon EC 2. The configuration of each machine is described in Table 1.

We have measured the total execution time of the data mining task by varying data size and the number of computing node.

- data size: we have then chosen the number of images as 1600, 3200, 6400, 12800, and 25600 respectively;
- the number of virtual nodes: we have used numbers of virtual nodes from Amazon EC2 ranging from 1 to 16.

Based on Gauss-Newton algorithm and the average of five independent experiment runs, we have computed six parameters in Eq. (20) and three parameters in Eq. (6):

$$A_1 = 1.6341,\ A_2 = 0.0238,\ A_3 = 7.7525,\ A_4 = 78.1657,$$
$$A_5 = 0.0098,\ A_6 = 212.1893$$
$$A'_1 = 1.7449,\ A'_2 = 87.6264,\ A'_3 = 420.1313$$

Therefore, the Eqs. (19) and (6) can be finalised in Eq. (21) and (22).

$$T_m(D,m) = 1.6341\frac{D^2}{m} + 0.0238D \times m + 7.7525\frac{D}{m} + 78.1657D + 0.0098\frac{1}{m} + 212.1893 \quad (21)$$

$$T_s(D) = 1.7449D^2 + 87.6264(D) + 420.1313 \quad (22)$$

4.4 Result analysis from real experiments in the cloud and the performance model

We have calculated the speedup for both the experiment (the average of 5 independent runs) and the model as shown in Tables 2, 3, and Fig. 6. The first column in the tables represents different data sizes. The size of input datasets is multiplied by a factor of 2 in every increment, up to 32 times of the initial input data size (808 images, with every image containing 60,000 pixels and the dimension is 300 × 200). The

Table 2 Speedup from the experiment

	Node1	Node2	Node4	Node8	Node16
1× data	1	1.87	2.07	2.35	2.13
2× data	1	1.85	2.26	2.38	2.49
4× data	1	2.05	2.49	2.68	3.04
8× data	1	1.94	2.75	2.93	3.45
16× data	1	1.87	3.12	4.24	4.19
32× data	1	1.97	3.62	5.32	6.36

Table 3 Speedup from the model

	Node1	Node2	Node4	Node8	Node16
1× data	1	1.96	2.12	2.21	2.26
2× data	1	1.95	2.24	2.42	2.52
4× data	1	1.93	2.43	2.79	3.00
8× data	1	1.93	2.69	3.36	3.82
16× data	1	1.94	3.01	4.14	5.08
32× data	1	1.96	3.33	5.07	6.83

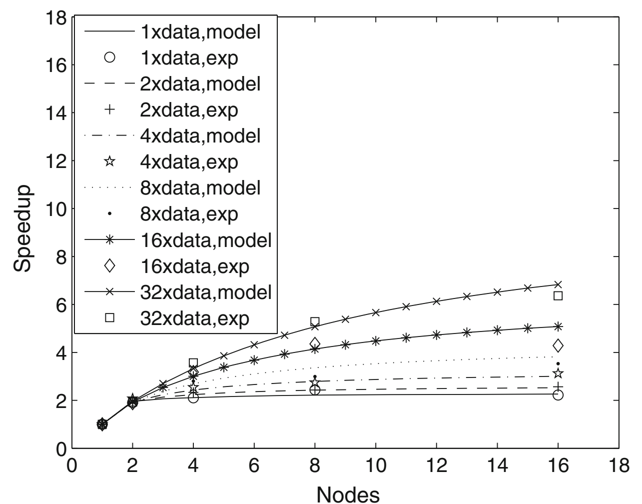


Fig. 6 The speedup from both experiments and the performance evaluation model

first row in the tables represents different computing nodes. Table 4 shows the relative errors of the total execution time between the model and the experiment results. The average relative error is 0.0058 with 99 % confidence level and the confidence interval 0.0058 ± 0.0309.

From the result we measured, the system exhibits less than desired 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 32× of initial input size (25,856 images), the speedup achieved with 8 machines is 5.32; but with 16 machines, the speedup achieved is only 6.36. To understand the key rate-limiting

Table 4 The relative errors of the execution time between the experiment and the model

	Node1	Node2	Node4	Node8	Node16
1× data	0.0659	0.0354	0.0572	0.1740	0.0480
2× data	0.0244	−0.0121	0.0659	0.0266	0.0421
4× data	−0.0627	0.0044	−0.0200	−0.0794	−0.0260
8× data	−0.0114	0.0040	0.0280	−0.1200	−0.0853
16× data	0.0120	−0.0147	0.0709	0.0639	−0.1460
32× data	−0.0010	−0.0200	0.0718	0.0402	−0.0693

Table 5 Speedup of each MapReduce job

	Node1	Node2	Node4	Node8	Node16
IP	1	1.97	3.86	7.16	10.95
WT	1	2.03	3.73	5.99	10.32
TT	1	2.39	3.54	5.58	6.97
TR	1	2.12	3.80	5.39	9.84
MV	1	1.76	2.86	2.71	2.05
FR	1	3.06	3.48	3.48	3.37
TN	1	1.89	3.96	2.12	2.12
KNN	1	1.75	3.64	6.39	9.50

factors, we have conducted a detail analysis from both the experiment and the model.

In the experiment, we timestamped each process of 8 MapReduce jobs. The Table 5 shows the speedup for each MapReduce job. We can make the following observations:

- Some parallel jobs have very good speedup performance in the MapReduce model, such as, image preprocessing, wavelet transformation feature generation and K-nearest neighbours.
- Long running jobs have better speedup than jobs that have short duration, for example, jobs for both training and testing feature extraction are almost identical, but testing set extraction time is always 10 % of the training set extraction.
- Jobs with short duration (in the range of 20–40 s) have less than desired speedup, such as Fisher's Ratio calculation and Top N feature selection. Since the inputs of these jobs are always reduced to a constant number, the minimum execution time is bounded by the MapReduce overheads. These jobs do not benefit from the parallelisation.
- The key rate-limiting factor in the system is the calculation of the Mean and variance in the feature selection process. It exhibits 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 numbers of computing nodes increases, the number of Mappers increases proportionally, but only two Reducers are in this stage. Hence, the communication bottlenecks build up in the cluster, resulting in longer total execution time. The non-parallel portion in the Reducer is the main culprit in limiting the overall system speedup.

The above experimental results can be also explained based on the performance model as follows:

- In the model Eq. (21), the parameter $A_4 = 78.1657$ represents the proportional coefficient in relation to data size only (non-parallel parts), which is greater than parallel parts such as A_1, A_3 and A_5. This shows that the speedup is affected by these non-parallel tasks.
- The relationship between the speedup and the number of computing nodes and data size can be analysed based on the speedup:

$$Speedup = \frac{A_1' D^2 + A_2' D + A_3'}{A_1 \frac{D^2}{m} + A_2 m D + A_3 \frac{D}{m} + A_4 D + A_5 \frac{1}{m} + A_6}$$

  When the data size is fixed, the speedup increases in the first instance with the increasing number of computing nodes in the certain period. However, when the number of computing node reaches a certain value, then the speedup will decrease after this maximum value. As m → ∝, the speedup will tend to zero. When the numbers of computing nodes is fixed, the speedup increases with the increase of the data size (D). As D → ∝, the speedup will achieve a certain value as $\frac{A_1'}{A_1} m$.
- According to the model, in order to speedup the performance, we can optimise the architecture of the prototype by parallelising those non-parallel parts. In this case, for example, parallelising calculation of mean and variance.

As a proof of concept, we used 16 nodes to run the experimental evaluation in the cloud. However, to generalise the idea of this paper behind and demonstrate the scalability of the system, we have also run simulation using our performance evaluation model. The Fig. 7 shows the prediction of the speedup with over 20 nodes. The prediction demonstrates the scalability where the speedup increases with the increase of the scale of data.

4.5 Cost effectiveness of cloud computing

In this case, we have adopted Cloud computing platform as an infrastructure running our application. In terms of turnaround time and cost effectiveness, it is demonstrated that the deployment of our application to the cloud is very easy and within a short time frame. We have calculated the execution time on a physical machine and on multiple virtual nodes in

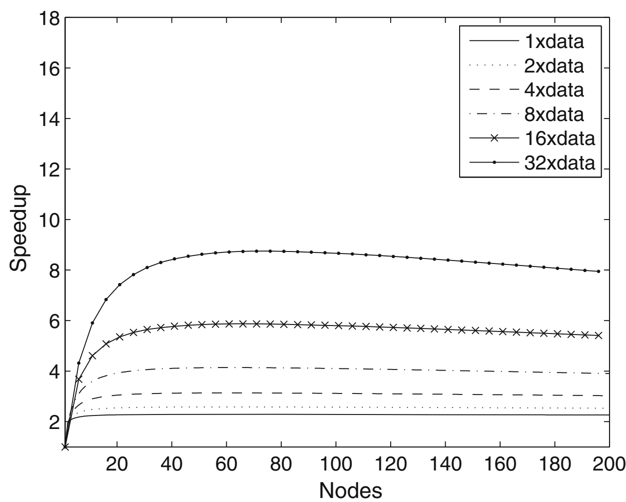


Fig. 7 Prediction of speedup based on the performance evaluation model

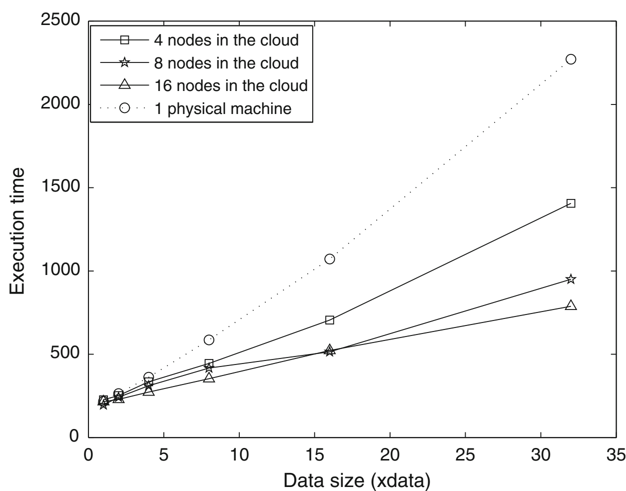


Fig. 8 Execution time (s) of cloud nodes and a physical machine respectively

Table 6 Costs in the cloud

Item	Unit cost($)	Unit needed	Total cost($)
Amazon: large instance	0.34/h	157 h	53.38
Amazon: storage	0.10/GB-Month	126.80 GB	12.68

the Cloud respectively, showed in Fig. 8. It found that there is less execution time when multiple virtual nodes run the task in parallel. Additionally, we have calculated the cost for running the whole experiments in the Cloud described in this paper, as shown in Table 6.

5 Discussion

Based on the experimental evaluation in Sect. 4, it is found that the performance and scalability of the data mining task can be enhanced by using MapReduce model and Cloud computing. Through a detailed analysis from both the experiment and the model, we can understand the further optimisation of the data mining task and the class of similar data-intensive applications:

- For the tasks that can be parallelised based on the data parallelism method, the data mining application can achieve good speedup, such as image preprocessing, feature generation (e.g., Wavelet Transform).
- For those data parallel tasks, the bigger the data size of the input is, the greater the speedup is.
- The non-parallel portion is a main factor that limits the speedup and affects the system performance. Therefore the optimisation strategy will mainly rely on parallelising the non-parallel parts of the data mining task.

Based on our experience in this case, as a proof of the concept, we found the Cloud is an efficient way to conduct research-based pilot studies, for example, to test the feasibility of the initial research idea without huge upfront cost and concerns of maintenance of the infrastructures.

Additionally, it is noted that MapReduce is a programming model. For any given application, one has to adapt the application to MapReduce model first and then run the application on different computing platforms (e.g., physical clusters or virtual clusters on the Cloud).

6 Conclusion and future work

In this paper, we have investigated how to apply MapReduce and Cloud computing to a real data mining use case in order to understand how well these advanced technologies can accelerate the performance, scalability and reduce the cost in supporting data-intensive applications. We have adapted the data mining task (initially written in sequential codes) to MapReduce programmes. The prototype was deployed on the Cloud computing platform for experimental evaluation. We have used the speedup as the standard metric. The performance evaluation model has also been built up for assessing the efficiency of the prototype and analysing the possible optimisation strategies of the prototype architecture.

The results from both the experiment and the model show that the performance can be improved by using MapReduce model. We have also compared execution time and the cost of the Cloud computing and the estimated cost of the private owned cluster respectively. The result shows that the Cloud computing could be an alternative cost-effective solution for supporting data-intensive applications, particularly for research-based pilot studies.

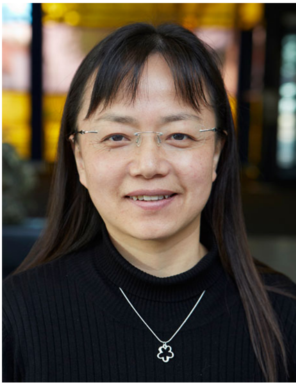
Our future work will focus on the further refinement of parallelisation on the data mining task and the class of the

similar data-intensive applications and conducting the performance benchmark test between Cloud computing and the big cluster in order to provide evidence for preparation of migrating the large-scale data-intensive applications to the Cloud.

Acknowledgments The authors acknowledge the support of the EurExpress team (EU-FP6 funding) at the MRC Human Genetics Unit, UK, a BBSRC funded Project (Agile) and Amazon EC2 on the continuation of this work. The authors would also like to thank the anonymous reviewers, who provided detailed and constructive comments on an earlier version of this paper.

References

1. Amazon: Amazon elastic compute cloud. http://aws.amazon.com/ec2 (2013). Accessed on 23 Dec 2013
2. Apache: Apache hadoop. http://hadoop.apache.org/core/ (2013). Accessed on 23 Dec 2013
3. Arpaci-Dusseau, R.H., Anderson, E., Treuhaft, N., Culler, D.E., Hellerstein, J.M., Patterson, D., Yelick, K.: Cluster i/o with river: making the fast case common. In: Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems, pp. 10–22. ACM, New York (1999)
4. Atkinson, M., van Hemert, J., Han, L., Hume, A., Liew, C.S.: A Distributed Architecture for Data Mining and Integration, pp. 11–20. ACM, New York (2009)
5. Beynon, M.D., Kurc, T., Catalyurek, U., Chang, C., Sussman, A., Saltz, J.: Distributed processing of very large datasets with DataCutter. Parallel Comput. **27**, 1457–1478 (2001)
6. Cellknn: Cell-knn: an implementation of the knn algorithm on sti's cell processor. http://code.google.com/p/cell-knn/ (2011) Accessed on 19 April 2014
7. Condor DAGMan (directed acyclic graph manager): http://www.cs.wisc.edu/condor/dagman (2007) Accessed on 19 April 2014
8. Cover, T., Hart, P.: Nearest neighbor pattern classification. IEEE Trans. Inf. Theory **30**(1), 21–27 (1967)
9. DB2: IBM DB2: http://www-01.ibm.com/software/data/db2/ (2013). Accessed on 23 Dec 2013
10. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI), pp. 137–150 (2004).
11. Deelman, E., Singh, G., Su, M.H., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G.B., Good, J., Laity, A.C., Jacob, J.C., Katz, D.S.: Pegasus: a framework for mapping complex scientific workflows onto distributed systems. Sci. Program. **13**(3), 219–237 (2005)
12. Disco: Disco mapreduce framework. http://discoproject.org/ (2013). Accessed on 23 Dec 2013
13. Duda, R.O., Hart, P.E.: Pattern Classification and Scene Analysis. Wiley, New York (1973)
14. Gokhale, M., Cohen, J., Yoo, A., Miller, W.: Hardware technologies for high-performance data-intensive computing. IEEE Comput. **41**(4), 60–68 (2008)
15. Gorton, I., Greenfield, P., Szalay, A., Williams, R.: Data-intensive computing in the 21st century. Computer **41**(4), 30–32 (2008)
16. Han, L., Saengngam, T., van Hemert, J.: Accelerating data-intensive applications: a cloud computing approach to parallel image pattern recognition tasks. In: W. Gentzsch, P. Lorenz, O. Dini (eds.) ADVCOMP 2010: The Fourth International Conference on Advanced Engineering Computing and Applications in Sciences, 978-1-61208-101-4, pp. 148–153. IARIA (2010)
17. Han, L., van Hemert, J., Baldock, R.: Automatically identifying and annotating mouse embryo gene expression patterns. Bioinformatics **27**(8), 1101–1107 (2011)
18. Han, L., Liew, C.S., van Hemert, J.I., Atkinson, M.P.: A generic parallel processing model for facilitating data mining and data integration. J. Parallel Comput. **37**(1), 157–171 (2011)
19. Hey, T., Tansley, S., Tolle, K.: The Fourth Paradigm: Data-Intensive Scientific Discovery, 1st edn. Microsoft Research, Redmond (2009)
20. IDC digital universe study: Big data is here, now what? Accessed on 23 Dec 2013
21. Jin, R., Agrawal., G.: A middleware for developing parallel data mining implementations. In: Proceedings of the First SIAM Conference on Data Mining (Apr, 2001)
22. Jin, R., Yang, G., Agrawal, G.: Shared memory parallelization of data mining algorithms: techniques, programming interface, and performance. IEEE Trans. Knowl. Data Eng. **17**(1), 71–89 (2005)
23. Laszewski, G., Hategan, M.: Workflow concepts of the Java Cog Kit. Grid Comput. **3**(3–4), 239–258 (2005)
24. Laszewski, G., Hategan, M.: Java CoG Kit Karajan-Gridant Workflow Guide. Technical Report. Argonne National Laboratory, Argonne (2005)
25. LINQ: The LINQ project. http://msdn.microsoft.com/netframework/future/linq/ Accessed on 19 April 2014
26. Microsoft: http://research.microsoft.com/en-us/projects/Dryad/ (2013). Accessed on 23 Dec 2013
27. Oinn, T., Greenwood, M., Addis, M., Alpdemir, N., Ferris, J., Glover, K., Goble, C., Goderis, A., Hull, D., Marvin, D., Li, P., Lord, P., Pocock, M., Senger, M., Stevens, R., Wipat, A., Wroe, C.: Taverna: lessons in creating a workflow environment for the life sciences. Concurr. Comput. **18**(10), 1067–1100 (2006). doi:10.1002/cpe.v18:10
28. Pacheco, P.S.: Parallel Programming with MPI. Morgan Kaufmann Publishers, Inc., San Francisco (1997)
29. Pike, R., Dorward, S., Griesemer, R., Quinlan, S.: Interpreting the data: Parallel analysis with Sawzal. Sci. Program. **13**(4), 277–298 (2005)
30. PVM: http://www.csm.ornl.gov/pvm/ (2013). Accessed on 23 Dec 2013
31. Raicu, I., Zhao, Y., Dumitrescu, C., Ian Foster, M.W.: Falkon: a fast and light-weight task execution framework. In: IEEE/ACM SC 2007 (2007)
32. Raicu, I., Zhao, Y., Foster, I., Szalay, A.: Accelerating large-scale data exploration through data diffusion. In: International Workshop on Data-Aware Distributed Computing 2008. IEEE Computer Scociety (2008)
33. t Grossman, R., Gu, Y.: Data mining using high performance clouds: Experimental studies using sector and sphere. In: Proceedings of The 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, New York (2008)
34. Taylor, I., Shields, M., Wang, I., Harrison, A.: The Triana Workflow Environment: architecture and applications. In: I. Taylor, E. Deelman, D. Gannon, M. Shields (eds.) Workflows for e-Science, pp. 320–339. Springer, London (2007)
35. Teradata: http://www.teradata.com/ (2013). Accessed on 23 Dec 2013
36. Vertica: http://www.vertica.com/ (2013). Accessed on 23 Dec 2013
37. Wang, L., Tao, J., Ma, Y., Khan, S.U., Kolodziej, J., Chen, D.: Software design and implementation for MapReduce across distributed data centers. Appl. Math. Inf. Sci. **7**(1), 85–90 (2013)
38. Zhao, Y., Hategan, M., Clifford, B., Foster, I., von Laszewski, G., Nefedova, V., Raicu, I., Stef-Praun, T., Wilde, M.: Swift: Fast, reliable, loosely coupled parallel computation. In: IEEE Congress on Services (Services 2007), pp. 199–206 (2007)



Liangxiu Han has a PhD in Computer Science from Fudan University, Shang-hai, P. R. China (2002). She is currently a Reader at the school of Computing, Mathematics and Digital Technology, Manchester Metropolitan University and leads FUNDS research group (Future Network and Distributed Systems). Her research areas mainly lie in the development of novel architectures for large-scale networked distributed systems (e.g. Cloud/Grid/Service-oriented computing/Internet), large-scale data mining (application domains include web mining, biomedical images, environmental sensor data, food and agriculture, network traffic data, cyber security, etc.), and knowledge engineering. As a PI or Co-PI, Dr. Han's research is funded by research councils, industries and charity bodies, in the research areas mentioned above. Dr. Han is a member of EPSRC Peer Review College and as an expert for Horizon 2020 proposal evaluation. Dr. Han is also a regular reviewer for several prestigious journals and international conferences in the field.

Hwee Yong Ong obtained his MSc. in Computer Science from University of Edinburgh in 2010. He is currently working as a Senior Manager at Centre for Strategic Infocomm Technologies in Singapore.