Workshop Notes of the International Workshop on Parallel Data Mining

in conjuction with SIAM DM 2011

Alexandre Termier (Ed.)

Mesa, Arizona, U.S.A., 30th April 2011

Preface

Today is an exciting time to be a data mining researcher. There is a huge quantity of available data, and a high demand for knowledge extraction from this data that only data mining can provide. Moreover, with the democratization of parallel computing solutions, be it multicore processors, GPUs, clusters or grids, there has never been more raw computing power available.

There is a catch however: parallel computing power does not come for free. Designing efficient parallel algorithms requires new skills, many of which are yet to discover. Most current programming language does not make parallel programming easy, leading to new research on different ways of programming. Then performance analysis, which is a crucial point for data mining algorithms, get much more complex, as the scaling with the number of computing elements must also be studied. Bad performance can be caused by an inefficient usage of the underlying parallel system architecture. These problems quickly arise as data mining algorithms often need to perform complex computations on huge volumes of data, with irregular and unpredictable computing loads, and generating large quantities of intermediate data and/or results. This pushes the parallel architectures to their limits, forcing data mining researchers to get a precise understanding of how these architectures work and to rethink existing algorithms.

The goal of this first Parallel Data Mining Worshop is to provide a venue for all data mining researchers working on parallel data mining, whatever the parallel support used. Despite differences in support, the main problems of parallelism remain the same, discussions between researchers working on different parallel supports or different domains of data mining will allow cross-fertilization and help everyone to progress.

This first edition of the workshop on a still young domain attracted a modest number of 3 submissions, which were all found satisfactory by the reviewers. Fortunately, the three papers cover a range a parallel data mining problems: the first paper uses parallelism on multicores for classification, the second one presents pattern matching techniques on GPU, and the last one deals with load balancing of data mining tasks in grids and clouds. This will give the workshop audience a nice overview of parallel data mining in various environments, which will be completed with invited talks.

We thank the SDM conference for their support, the program comittee for their detailed reviews, and the authors for the quality of their submissions.

Alexandre Termier

Program comittee

Table of contents

Schedule

Workshop date: April the 30th, 2011, 1.30pm - 4.30pm

- 1.30pm - 2.30pm: **Invited talk:** Pr. Gagan Agrawal, The Ohio State University
- 2.30pm - 3.00pm: Aerial Root Classifiers for Predicting Missing Values in Data Stream Decision Tree Classification
  Yang Hang, Simon Fong, Wei Chen
- 3.00pm - 3.30pm: Coffee break
- 3.30pm - 4.00pm: Monitoring Multiple Streams with Dynamic Time Warping using Graphic Processors
  Jason Chang, Mi-Yen Yeh
- 4.00pm - 4.30pm: Supporting Dynamic Load Balancing in a Parallel Data Mining Middleware
  Tekin Bicer, Gagan Agrawal

Aerial Root Classifiers for Predicting Missing Values in Data Stream Decision Tree Classification

Yang Hang * Simon Fong † Wei Chen ‡

Abstract

Data Stream Mining (DSM) is a new breed of data mining algorithms that handles continuous data streams and predicts (or classifies) a target value on the fly. Such data streams are inevitably prone to have missing values. Some common examples include temporary malfunction of a sensor that feeds continuous data streams; and interruption on a flow of data communication signals may give rise to missing data in the input of a data stream miner. Consequently, the missing data lead to deterioration on the accuracy of the data stream miner. Several techniques exist for dealing with missing data in traditional data mining algorithms, such as setting a default value or eliminating the records that have missing data. Another classical technique is to estimate or predict a missing value by statistically computing the mean of all other values of the attribute. This does not work for DSM because the training and testing by DSM is dynamically done over a moving stream of data instead of scanning through a complete dataset (as in traditional data mining). Inspired by the aerial root in biology, we propose a method that combines sliding window technique, feature selection, Hoeffding tree classification as well as adventitious root concept to deal with missing values. As a spontaneous sidekick to the main DSM classifier, Aerial Root Classifier (ARC) is implemented with sliding window for predicting missing values, which may work even if concept-drift happens. A row of ARC's and HTA are running in parallel, with one ARC corresponds to an attribute of the data stream. For efficient operation, only a partial set of ARC's are chosen to be activated via dynamic Feature Selection. We built a JAVA-based simulation system for conducting experiments with various types of datasets. Improved accuracy was observed by applying this new ARC algorithm.

1 Introduction.

Missing data has been a known problem in data mining for the fact that seldom data can be perfectly collected from a perfect environment. Data are often infested with all kinds of transmission errors and external interferences. For example, intermittent downtimes at a faulty communication link, temporary unavailability of a data source, and even human errors would result in missing data in computing environment. As a preprocessing step in knowledge discovery process, missing data are traditionally ignored or treated by assigning a special value into the corresponding data field. The special value can be either a default value arbitrarily set by the user or a statistical mean of all other values pertaining to the attribute. Although this preprocessing approach in general works well in traditional data mining, aspects of missing data or incomplete data are still a pretty much unexplored research area in the emerging data stream mining.

Amongst many DSM methods, real-time decision tree classification is a favorite technique because it is found useful in many applications that require instant decision-making over the continuous input of data streams. Hoeffding tree algorithm (HTA) is a famous decision tree classification technique for high speed data stream mining [1]. In contrast of traditional data mining, specifically induction-based decision tree, HTA has the following properties. Stationary and Unstationary data input: the HTA classifier should have ability to process infinite volume of data arriving at variable speeds. Limit memory space: only a limited memory space is available to the HTA classifier in online service, which means the classifier has to scan only a segment of input samples at a time. Very fast response: the computation time is governed by QoS, which must satisfy the real time requirement within a stringent time threshold. High applicability for incomplete data: the incoming data stream may be incomplete that carries missing values; the classifier must be able to process imperfect data. In the past, researchers often assumed DSM algorithms were inputted with perfect data at constant rate and without missing values. To the best of our knowledge no work has been done in studying incomplete data in DSM.

2 Previous work and motivation.

In the traditional preprocessing approach, it may be fine that missing data being ignored when a large volume of training dataset is used for constructing an information-gain induced decision tree model (e.g. C4.5), as long as the missing values represent only a minority of the whole dataset. However, missing data is more impactful to DSM because of the streamlined nature of the HTA

*Department of Computer and Information Science, University of Macau.

†Department of Computer and Information Science, University of Macau.

‡Department of Computer and Information Science, University of Macau.

algorithm. The algorithm scans the input samples only one at a time in order to relieve the bottlenecks of time and memory of computation. As mentioned in [2], that it may be sufficient to use a small number of available samples when choosing the split attribute at any given node for building a HTA decision tree. The statistical method is known as Hoeffding bound or additive Chernoff bound, which are used to solve the difficult problem of deciding exactly how many samples are necessary at each node by using a statistical result [8] . For this reason, every sample matters and counts in the computation of Hoeffding bound algorithm. On the other hand, due to the dynamic nature of moving data streams and the data stream miner can only view a limited portion of the data at a time, estimating a suitable figure for a missing data from the full set of data becomes technically difficult. In reality, the data samples from data stream might fluctuate and they are perforated with incomplete values as they arrive and to be mined a portion at a time. Therefore, for stream mining, each pass of data input has a limited size as large as the given sliding window. One or several pieces of missing data may occupy a significant population of the input data in a window. The traditional preprocessing techniques for handling missing values became undermined.

In our previous work [3], [4], we examined the effect of Internet traffic on Hoeffding bound which is one of the key performance indicators in stream mining. We found that the fluctuation of data rate in real-time oscillates the error which causes frequent HTA tree reconstruction, that degrades the overall HTA accuracy as a result. But the previous study was based on the assumption that data are always complete (without anything missing) in a given network environment. In this paper, we propose a method called Aerial Root Classifiers (ARC) that is based on HTA to deal with missing values included in data stream. One ARC is supposed to estimate missing values of a particular attribute of the streaming data. A row of ARC's operate in with the main HTA classifer, which collectively predict the missing values (if any) for their corresponding attribute fields.. As the results of experimental comparison shown in this paper, ARC-HTA improves the overall decision tree classification accuracy in DSM comparing to the existing methods. This paper is structured as follow: Section 3 provides the relevant background knowledge such as the decision tree classification for both traditional and DSM, preprocessing techniques and the term of "aerial root" in biology. Section 4 proposes our innovative method for predicting missing values in data stream; Section 5 discusses the experimental results that show the performance of ARC-HTA in comparison to the old ones. A conclusion then followed at the end.

3 Background.

3.1 Traditional decision tree classification. The decision tree structure in DSM, which is similar to that of traditional data mining, is represented by a tree-like graph of paths leading to consequences through stages of conditional tests. In this paper we generalized the classifier to be named as a traditional decision tree (TDT) model with classical algorithms such as ID3 [5], C4.5 [6], and CART [7], etc. All of them need to scan the full set of data from database at least once to construct a tree-like format. When new data comes, TDT has to scan the whole database once again to update the decision tree model. This approach works on not very large database or low speed data stream. However, under the information booming, we are facing a huge amount of data. Retraining the decision tree model by scanning the whole database many times may not be feasible for real-time applications nowadays. For this reason, some researcher proposed incremental computing methods of decision tree, which need not multi-scanning the whole dataset, such as DSM.

3.2 Hoeffding tree algorithm (HTA) in DSM. As highlighted in [2], it may be sufficient to use just a small available data sample for choosing the split attribute at any given node for a decision tree. This statistical method is known as Hoeffding bound or additive Chernoff bound, which is used to solve the difficult problem of deciding exactly how many samples are necessary at each node by using a statistical result [8], [9], [11], [12], [13], [10]. Researchers from the literature cited above attempted to innovate new algorithms that can restrict the tree size in available memory so as to deal with very large data amount.

VFDT (Very Fast Decision Tree) system [8] constructs a decision tree by using constant memory and constant time per sample. It is a pioneer predictive technique that utilities Hoeffding bound. The tree is built by recursively replacing leaves with decision nodes. The sufficient statistics of attribute values are stored in each leaf. Heuristic evaluation function is used to determine split attributes converting from leaves to nodes. Nodes contain the split attributes and leaves contain only the class labels. The leaf represents a class that the sample labels. When a sample enters, it traverses the tree from root to a leaf, evaluating the relevant attribute at every single node. After the sample reaches a leaf, the sufficient statistics are updated. At this time, the system evaluates each possible condition based on attribute values, if the statistics are enough to support the one test over the others; a leaf is converted to a decision

node. The decision node contains the number of possible values for the chosen attribute about the split-test installed.The main elements of VFDT include: Firstly, state the tree only has a single leaf - the root of the tree. Secondly, define the heuristic evaluation function (denoted by G(.)), which builds a decision tree with Information Gain like ID3.The Information Gain measures that amount of information which is necessary to classify a sample that reaches the node in terms of (3.1). The sufficient statistics estimates the merit of a discrete attributes counts n_{ijk}, representing the number of samples of class k that reach the leaf where the attribute j takes the value i. The information of the attribute j is given by (3.2), where P_{ik} is the probability of observing the value of the attribute i given class k.P_i in(3.3) is the probabilities of observing the value of attribute i.

$$G(X_j) = info(samples) - info(X_j) \tag{3.1}$$

$$info(X_j) = \sum_i P_i \left(\sum_k -P_{ik} \log \left(P_{ik} \right) \right) \tag{3.2}$$

$$P_{ik} = n_{ijk} / \sum_a n_{ajk} \tag{3.3}$$

$$P_i = \sum_a n_{ija} / \sum_a \sum_b n_{ajb} \tag{3.4}$$

$$\varepsilon = \sqrt{\frac{R^2 \ln (1/\delta)}{2N}} \tag{3.5}$$

For n independent observations of a real-valued random variable r whose range is R, Hoeffding bound is calculated as in (3.5). It illustrates that with confidence level $(1-\delta)$, the true mean of r is at least $(\bar{r} - \varepsilon)$, where $\bar{r}$ is the observed mean of samples. For a probability the range R is 1, and for an information gain the range R is $\log Class\#$. An important part of VFDT is the use of Hoeffding bound to choose a split attribute as the decision node. Let X_a be the attribute with the highest G(.), X_b be the attribute with second-highest G(.). Therefore $\Delta\bar{G} = \bar{G}(X_a) - \bar{G}(X_b)$ is the difference between the two top quality attributes. If $\Delta\bar{G} > \varepsilon$ with N samples observed in leaf, while the Hoeffding bound states with probability $(1-\delta)$ that X_a is the attribute with highest value in G(.). Then the leaf is converted into a decision node which splits on X_a.

In large volume continuously-changing data stream, concept-drift might happen. VFDT is built on the assumption of random samples drawn from a stationary distribution that it cannot suit time-changing learning approach. CVFDT (Concept-adapting Very Fast Decision Tree) [9] applies VFDT with a sliding window technique. As the new samples arrive, they are inserted into the beginning of the window; a corresponding number of samples are removed from the end of the window so that the learner is up-to-date. Additionally, CVFDT imports a parameter γ , where $\Delta\bar{G} < \varepsilon < \gamma$. γ is a user-defined threshold that reduce the computation of $\Delta\bar{G}$. The choosing split attribute method of CVFDT is the same as VFDT, both using Information Gain.

$CVFDT_{NBC}$ [10] adopts naive-Bayes Classifiers in the leaf nodes of a decision tree induced by CVFDT so as to detect concept-drift. Both CVFDT and $CVFDT_{NBC}$ generate alternative sub-trees while concept drift being detected. If the sub-tree accuracy is higher than the old one, the alternative one will replace the old one whose root node is a node. But since nodes close to the root node store a lot of samples, it is difficult to detect concept drift quickly in the case of abrupt concept drift. VFDTc [11] proposes to bring a performance of Hoeffding tree similar to traditional decision tree algorithms like C4.5. Besides large size data, VFDTc also suits medium size data so that the system can be any-time property. It uses two classifier strategies at leaves: majority class classifier and naive-Bayes classifier. For continuous attributes, naive Bayes are efficiently derived from tree used to store numeric attribute values. But the overhead is with respect to the use of majority class because the former requires the estimation much more probabilities than the latter one. In the early study of HTAs, most of them concern on how to build a decision tree with a high accuracy in data stream, and how to deal with concept-drift problem. However, we investigate in this paper on how to solve the missing values problem in DSM.

3.3 Methods of estimating missing values. Missing values and noise data are unavoidable in real world data stream. They are introduced because of various reasons, such as human fault in manual data entry, incorrect measurements, and equipment errors. In general, there are two ways of defining missing value rate [14]: the percentage of predictor values that are missing from the data set (the value-wise missing rate), and the percentage of observations that contain missing values (the case-wise missing rate). There are another three different mechanisms of introduction of missing values: missing completely at random (MCAR), missing at random (MAR), and not missing at random (NMAR) [15], [16]. Only MCAR case is where the analysis of the remaining complete data could give a valid classifier prediction according to the assumption of equal distributions [16]. MCAR is introduced when the distribution of an example having a missing value for an attribute does not depend on either the observed data or the miss-

Figure 1: Aerial root in bio-system of a Banyan tree.

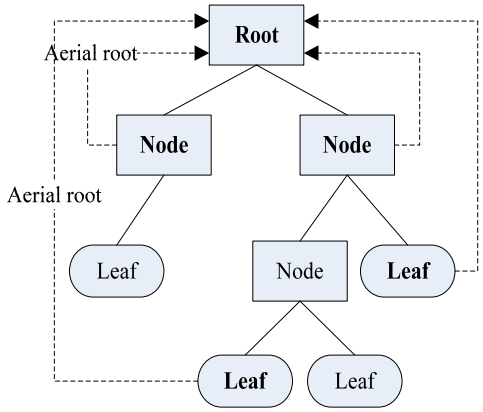


Figure 2: Abstract tree-like graph of aerial roots.

ing data. In this paper, we concern on the value-wise MCAR missing data.

In DSM research, how to detect concept-change by noise-carrying and missing values included data is an open problem. Streaming Ensemble Algorithm (SEA) [17] takes an ensemble approach using an unweighed majority vote, similar to bagging to detect the concept-change in data stream. Weighted Classifier Ensemble (WCE) [18] is based on the author proof that a carefully weighed classifier ensemble built on a set of data partition. WCE divides previous data into sequential chunks of fixed size, builds from each chunk a classifier to improve classification accuracy on the most recent chunk. A Flexible Decision Tree (FlexDT) [19] is proposed to embrace the concept-change and noise-carrying data stream using fuzzy logic and sigmoidal function. But its run time becomes slower than CVFDT because of fuzzy functions.

3.4 Aerial root in biology. Decision tree structure is derived from the life-form of a tree in nature. In addition to the primary tree root, there is another type called adventitious roots in nature, which grow from positions away from the primary roots. Such roots grow from the body of the main tree trunk and branches as well (Figure 1). An aerial root is a typical adventitious root growing above ground. They support the tree in parallel together with the other roots. An abstract tree-like structure of adventitious root bio-system is presented in Figure 2. On top of the tree-like graph, there are some aerial roots grown from the nodes and the leaves of tree body to the normal root on the ground. Each dashed line represents an aerial root in this tree.

4 Our proposed ARC-HTA algorithm.

Inspired by the aerial root in biology, we propose a method for high speed data stream, combining sliding window technique, feature selection, Hoeffding tree classification as well as adventitious root concept to tackle the problems of missing values. Concurrently to the main HTA decision tree, multiple Aerial Root Classifiers (ARCs) are implemented with sliding window for predicting missing values, which may work even if concept-drift happens. In maximum all the ARC's are used if the data fields of all the corresponding attributes have missing data. This however doesn't not scale up well in performance especially if the data have many attributes. Therefore, a feature selection technique is applied to selectively activate only a partial set of ARC's in run time. In other words only those attributes that are significant will receive the latest updated results from their respective ARC's. Otherwise, the ARC's are in dormant, and the least significant attributes may retrieve estimated values from their ARC's that were updated some time ago. With this new ARC-HTA scheme for subsiding adverse effect of missing values and selectively activating ARC's, we strike a balance between improved accuracy and acceptable computational requirements.

ARC is to be embedded into the operation of HTA (that represents the main classifier), which we may name it as ARC-HTA when they are running together as one unified data stream mining software program. Stream mining process is different from the traditional train-then-test data mining. It works on a test-and-train process which may keep an up-to-date decision tree rule. Likewise, ARC is also working in the same manner, using sliding window technique to train missing value predictor with the data stream coming. The ARC training process is running simultaneously with HTA process. ARC-HTA is capitalizing on the fundamental of Hoeffding tree algorithms, which uses Hoeffding

bound to choose split-attribute in a sequence of data stream. To solve the problem of missing values, ARC's are used to predict the missing values, and the overall tree sizes of ARC are kept within a sliding window size. The feature selection mechanism decides which ARCs should be updated during the DSM run time. Therefore, ARC-HTA is consisted of two parts: the main classifier is Hoeffding tree algorithm, which is applied for mining streaming data; and Aerial root classifiers are used for predicting missing values in parallel.

4.1 Main decision tree classifier - HTA. HTA classifier scans parts of data stream and checks it across with the Hoeffding bound and may choose a possible split-attribute according to comparing the highest and the second highest of G under the heuristic evaluation (3.1) . We define a tree structure that is built in terms of Hoeffding tree algorithm as HT; X_i is the split-attribute which represents a node in tree structure, n is the number of attributes (excluding class). C is classified class label, which represents a labeled leaf in tree structure. It is formalized as: $HT(X_n) \rightarrow C$. For example, there is a data stream that contains five attributes (n=5),$X = \{X_1, X_2, X_3, X_4, X_5\}$ and a class label C. Amongst them, the five attributes are used to classify the target class C, where C represents a leaf in tree-like graph of HT.

4.2 Missing values processor - ARC.

Definition 4.1. *ARC:* Let $X = \{X_1, X_2, \ldots, X_n\}$, C is the class label which is the predicted target in decision tree. A Hoeffding tree (HT) using n attributes X to classify the target class C and X_k is the attribute of missing value.X^* is the set of attributes excludes the missing value, where $X \cap X_k = \emptyset$. ARC structure is illustrated as: $ARC_k(X^*, C) \rightarrow X_k$. ARC can apply any classification method to predict the missing values of an attribute in data stream. If k is the number of attribute that contains a missing value, an ARC_k is constructed for predicting the missing values of attribute X_k. ARC is updated with fresh incoming data periodically bounded by the size of a sliding window during HTA stream mining operation. The update mechanism is combined with feature selection method of data mining, so that the ARCs are necessary to be refreshed only if the update condition is being met.

Definition 4.2. *Feature selection:* To reduce the computation cost of ARC-HTA, we introduce feature selection method. The process to define the weight of attributes relating to target class is different from traditional feature selection, the proposed one is applicable for HTA within an available sliding window. Information gain is chosen as the selection method. During the process of HTA to choose split attributes with Information Gain, the rank of each attribute is recorded at the same time. To reduce the computation resource, every ARC_i of X_i is assumed unnecessary to be updated except the overall ARC error rate falls beyond to a predefined bound. The weight of each attribute is decided by the attributes rank of feature selection presented in (4.6).

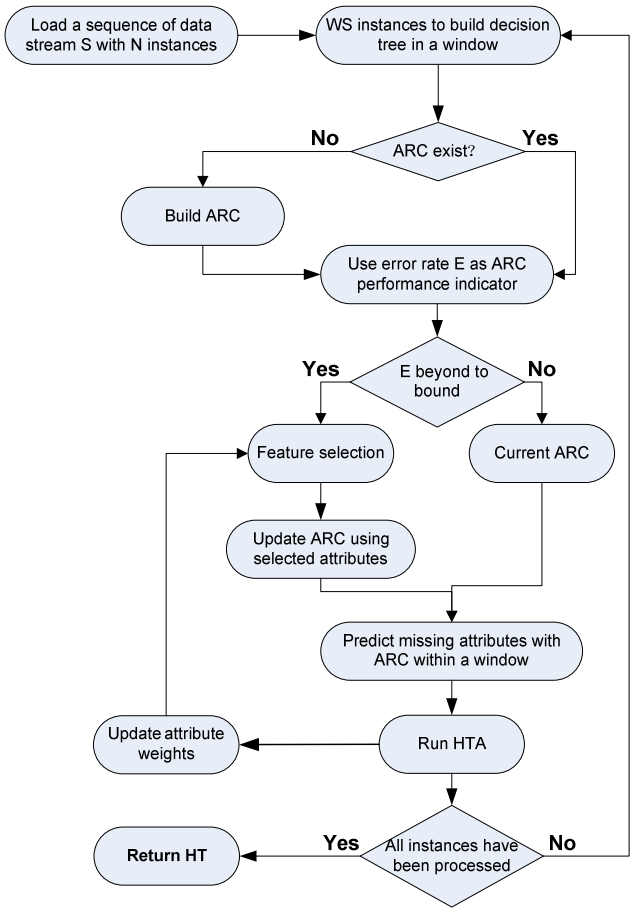


Figure 3: ARC-HTA flowchart.

Definition 4.3. *Update mechanism:* Each ARC has an error rate e, $0 \leq e \leq 1$, the probability that ARC will misclassify a randomly drawn test example is e, calculated by (4.9). As §ARC Definition §, each attribute X_i may have an ARC_i to predict the missing value in X_i. Combined with feature selection method, we assign each attribute a weight, which indicates relative importance to the target label C, as shown in (4.6) and (4.7). The overall error rate of ARCs is shown in(4.8). Equivalently, if an ARC classifies m

(very large) randomly drawn examples, the misclassified number of examples is expected to be $(m \times E)$. A pre-configured upper-bound β is the expected overall error rate. The update condition is met only when $\beta > E$. The priority of which ARC to be updated is relating to the feature selection that ranks the attributes from the highest pertinence the lowest. The update process continues until $E \leq \beta$.

$$\omega_i = \frac{Rank_i}{\sum_{i=1}^{n} Rank_i} \tag{4.6}$$

$$W = \sum_{i=1}^{n} \omega_i = 1 \tag{4.7}$$

$$E = \sum_{i=1}^{n} \omega_i e_i \tag{4.8}$$

$$e = \frac{CorrectClassifiedInstance\#}{TotalInstance\#} \tag{4.9}$$

4.3 ARC-HTA statement Aerial Root Classifier is of a reversed tree growing approach, which uses the missing value node as a predictive class label so that an ARC can handle missing values. In data stream mining, data is feeding in at a very fast speed so that we are not able to multiple scan the whole data to build ARCs. For this reason, we adopt a sliding window technique which only needs to scan the most recent data. Sliding window defines a window size that only requires scanning the data within a window. The window size can be time-based or item-based. In ARC-HTA, we use an item-based window size to bind the number of instances. As the aforementioned in the previous definitions, Figure 5 indicates the work-flow of ARC-HTA algorithm presented in the pseudo code (Figure 4).

5 Experiments analysis.

5.1 Experiment platform. We build a decision tree learning system based on Hoeffding tree algorithm, which embeds ARC algorithm to deal with missing values in data stream. The method of ARC implementation is optional, that it has to deal with both nominal and numeric data types. So far, we have imported the popular methods of Mean mode, Naive Bayesian, C4.5 Decision Tree for nominal data, and Mean, Linear Regression, Discretized Naive Bayesian, M5P for numeric data. A simulation system is built based on JAVA open source toolkit called WEKA and MOA stream mining, including data stream generator and Hoeffding Tree algorithm. The developing environment is under JAVA

HT: Hoeffding tree
ARC: Aerial root classifier
S: A sequence of instances
WS: Window size (the number of instances)
X: Set of attributes
δ: One minus the desired probability of choosing correct attribute at any given node
β : The update strategy upper- and lower-bound

Procedure ARC (S, X, WS)
Load S into cache
  FOR each x_i from 1 to n in S:
  IF ARC is empty
    THEN ***BuildARC(X,WS)***
  ELSE ***UpdateARC(X,WS, β))***
    Use ARC to predict the missing value in x_i
  Until no attribute has missing values in X
Run HTA (S, δ) to build decision tree
  HT(X)→C
  UpdateWeight(HT)
Return HT
BuildARC(X, WS):
FOR each x_i from 1 to X.length in S:
  IF x_k has missing value
    Use WS instances in S excluding x_k to build ARC
    Let $X^* = \{x_1, x_2, \ldots, x_{k-1}, x_{k+1}, \ldots x_n\}$
    $ARC_k (X^*, C) \rightarrow x_k$
Return ARC
UpdateStrategy(X,WS, β):
$W = \{\omega_1, \omega_2, \cdots, \omega_n\}$
IF current **ARC's error rate E≤ β**
  Don't need to update ARC
ELSE DO{
  MaxWeight= arg max ω_i
  BuildARC($x_{MaxWeight}$, WS)
  Remove $\omega_{MaxWeight}$ from W
} WHILE (E≤β)
Return ARC

Figure 4: Pseudo code of ARC-HTA.

JDK 1.5 and WEKA 3.6, The simulation system runs on a 2.83 GHz 8G RAM PC.

5.2 Synthetic data stream. Through a programmed JAVA function, the simulation system randomly adds missing values according to the user's predefined parameters. The missing values are either added to the beginning or near the end parts of the data streams, because we want to observe the impacts of missing data at start-up and when the HTA reached equilibrium respectively. Usually HTA will be unstable at start-up until it reaches certain size of input data. Three different datasets are generated as:

Table 1: Synthetic datasets test.

Name	*Attr#*	*AttrVal*	*Class#*	*Instance#*
LED7	7	*Nominal*	10	1,000,000
LED24	24	*Nominal*	10	1,000,000
SEA	3	*Numeric*	2	1,000,000

LED7 is a relatively simple dataset, which only contains 7 nominal attributes. In this experiment, we configure the missing data percentages (MDP) as 20%, 40%, 60%, 80% and 100%, by which chances are missing values (blanks) are randomly inserted across the data stream. When MDP=0%, the data stream is complete and free from missing values. A higher MDP is used, usually lower HTA accuracy is observed. Figure 7 presents the HTA accuracy comparison when missing data is added towards the end of data stream where the training HTA model is mature and the accuracy is supposed to be stable. In this set of experiment, we want to show the impact of missing values to the data stream mining. Missing values impair the accuracy dramatically if the algorithm does not have any mechanism to deal with the incomplete data. Essentially this observation is the impetus of our research work. We are motivated to find a robust method for dealing with missing values in data stream mining.

LED24 is a more complicated data stream with 24 nominal attributes and a total of one million instances records. We add a MDP 50% of noise into this dataset. To handle the incomplete data, ARC-HTA is applied with different window sizes: 250, 500, 750 and 1000 units. The C4.5 decision tree function in WEKA is chosen as the method of ARC construction. From the experiment result shown in Figure 8, we find that it has few differences between window sizes in a very large data stream. But actually smaller window size obtains a faster ARC-HTA computing speed. Besides, we compare ARC-HTA to one of common missing value solutions using the means to

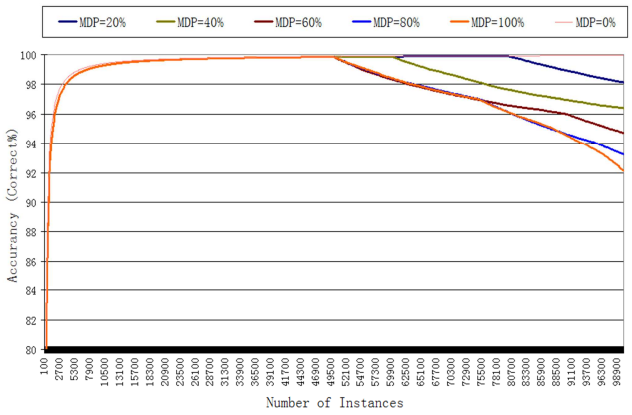


Figure 5: Accuracy comparison of missing values in LED7 dataset.

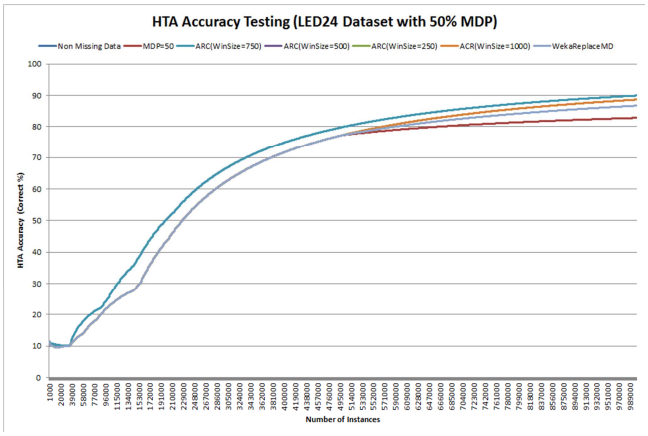


Figure 6: Accuracy comparison of missing values in LED24 dataset.

replace those missing values; we find that our proposed method has a better performance in data stream mining than using the mean value. ARC-HTA provides a function to check the performance of missing values replacement. Comparing to the non-missing values data stream, the result indicates how many correct/wrong number of instances are using different methods to replace the missing values (as shown in Table 2).

SEA is a data stream consisted of 3 numeric attributes and one nominal class and one million instances. We also add MDP = 50% amount of noise into this data. To handle incomplete data, we try ARC-HTA in different window sizes of 500, 750 and 1000 units. The ARC is built by M5P function in WEKA. From experiment result in Figure 9, we find that ARC-HTA obtains a dissatisfying result due to processing the pure numeric attributes. Nevertheless the accuracy is still a little higher than that provided by using means.

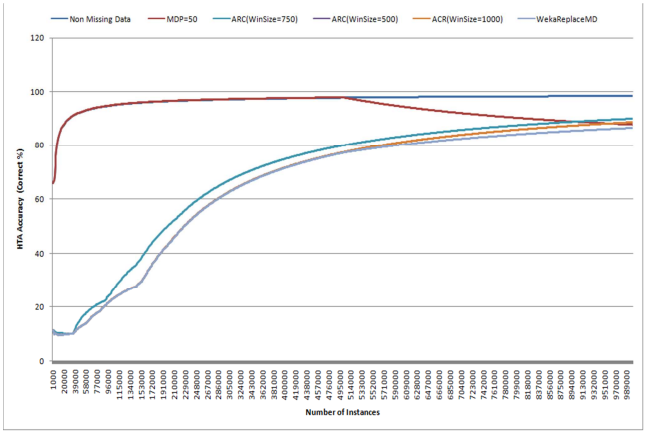


Figure 7: Accuracy comparison of missing values in SEA dataset.

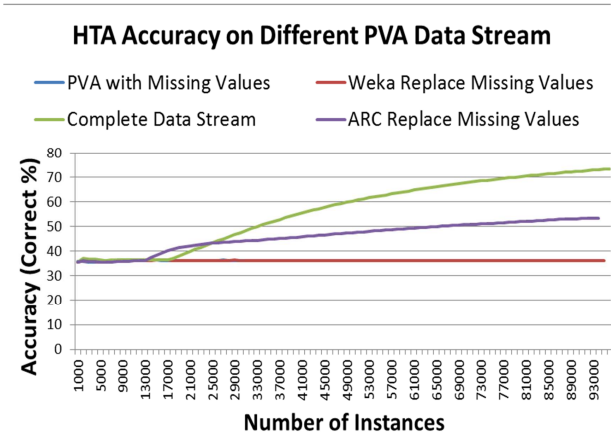


Figure 8: CUP98 dataset of PVA using in ARC-HTA.

5.3 Real-world data stream test. In this experiment, we use a set of real-world data stream downloaded from KDD Cup 98 competition provided by the Paralyzed Veterans of America (PVA) [20]. We use the learning dataset (127MB) with 481 attributes both in numeric and nominal originally. The total number of instances is 95,412, more than 70% of which contain missing values. Likewise to the experiments in §5.2§, we compare ARC-HTA with the missing values replacement method in WEKA by using means, and the result is shown in Figure 8. Considering the number of attributes is very large, we apply a small window size (WS=100) for building ARC. A complete dataset given by PVA is used for testing ARC-HTA (115MB). From the experiment result, we notice that using mean values to replace the missing data by WEKA filter has the worst HTA accuracy. Although ARC-HTA dealing with missing values dataset cannot have accuracy as good as the complete dataset can, ARC-HTA performance is much better than using means by WEKA to replace missing values.

Another dataset from real-world is imported in the experiment. This dataset can be downloaded from UCI Machine Learning [21], which is named Localization Data for Posture Reconstruction data. Though the observation of tag identification sensor of body location activities the learning motivation is to classify the different human activities. Original dataset has 7 attributes: Sequence Name (Nominal, X_1); Tag identifier ID (Nominal, X_2); Timestamp (Numeric, X_3); Date (Date, X_4); X coordinate of the tag (Numeric, X_5); Y coordinate of the tag (Numeric,X_6); Z coordinate of the tag (Numeric, X_7); and the Activity label (Nominal, C). It contains 164,860 instances. After using the full data with Information Gain filter as the feature selection,

```
Search Method:
        Attribute ranking.

Attribute Evaluator (supervised, Class (nominal): 8 Activity):
        Information Gain Ranking Filter

Ranked attributes:
 2.711256   4 Date
 2.711256   3 TimeStamp
 0.373474   5 X
 0.258462   6 Y
 0.258249   7 Z
 0.023547   1 PersonSeqName
 0.000383   2 TagIdentificator

Selected attributes: 4,3,5,6,7,1,2 : 7
```

Figure 9: Information gain feature selection.

the result is shown in Figure 9. We find the attributes of Date and Time is the most important attributes of the mining target class Activity. Amongst those three coordinates, X has the highest ranking, which is more than 40% higher than both of Y and Z. Missing values are added into the completed dataset so that it becomes incomplete. The scenario is chosen the Missing Completely At Random (MCAR) method. 40% of total instances are replaced by missing values, which are 65,944 instances added completely at random. The distributions of missing values in different attributes are: 8,056 in Person Sequence Name; 8,165 in Tag Identifier Sensor; 7,921 in Timestamp; 8,051 in Date; 8,092 in X Coordinate; 7,943 in Y Coordinate; 7,995 in Z Coordinate; and 8,141 in Activity Target.

The tested dataset includes numeric, date, and nominal attribute types. We choose linear regression as the ARC method of numeric attributes, and Naive Bayesian as that of nominal attributes. In the first

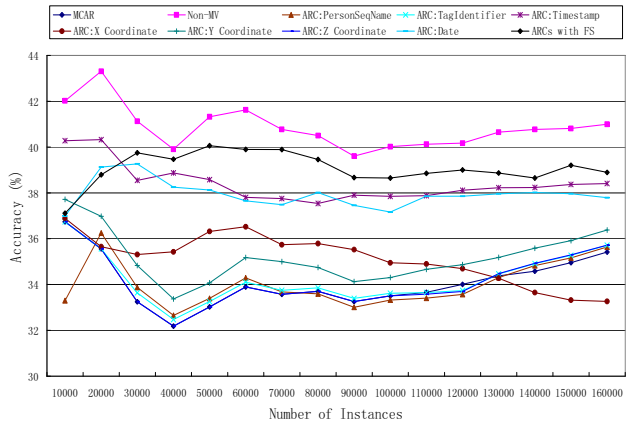


Figure 10: ARC-HTA overall accuracy.

experiment, we test the computation time and accuracy for each attribute one by one. In Table 2, ARC of Timestamp attribute and ARC of Date attribute have the better HTA accuracy than others. However, because the former applies regression as ARC method, the computation time of both ARC construction and missing value replacement consumes much longer than the latter one. The accuracy of these two ARCs are the best, which is corresponding to the previous result of feature selection, which indicates attributes of Timestamp and Date highly relate to the target class Activity label. As the process of Information Gain feature selection, ARC-HTA update the ARCs of these two features more frequently so that the missing value predictors of these two attributes are always Date ARCs being updated in parallel, the overall accuracy is improved in Figure 10.

Table 2: Performance of ARC of each attribute.ARC T: the time of building ARC for X_i; MV T: the time of replacing missing values; T: the time of running HTA. Meth: ARC method selection, NB: Nave Bayesian, REG: Linear regression.

Attr#	*Meth*	*ARCT*	*MVT*	*Acc*	*T*
X_1	NB	1.29	1.87	35.64	2.59
X_2	NB	1.37	1.49	35.71	2.61
X_3	REG	16.00	16.06	38.41	2.61
X_4	NB	2.65	2.73	37.11	2.63
X_5	REG	11.69	11.73	33.26	2.56
X_6	REG	24.15	24.19	36.38	2.70
X_7	REG	18.87	18.92	35.72	2.56
C	NB	1.28	1.57	38.29	2.65
X_3X_4	NB, REG	19.82	19.87	37.79	2.71

5.4 Experiment result summary. To make a short conclusion of our experiments. See from Table 3, we find the overall accuracy of HTA is improved by ARC integrated in all datasets. The most obvious improvement appears in the PVA dataset, which have the most number of attributes. SEA dataset has the least number of attributes with the lowest improvement. In other words, the more number of attributes data stream has, the more beneficial ARC-HTA obtains. Though ARC-HTA has had the mechanism of ARC updating, the running time of HTA with ARCs still is longer than that without ARC. The time consuming is also depending on the number of attributes in data stream because the more attributes a data stream has, the more ARCs it may have to maintain. Thus, in our future work, we may try to find a balance between the time and accuracy using ARCHTA so as to make the biggest benifit to DSM.

Table 3: Experiment result summary. MVT: the HTA running time of without ARC, ARCT: the HTA running time with ARC; MVA: the HTA accuracy without ARC; ARCA: the HTA accuracy with ARC;ImpA: the accuracy improvement percentage

Data	*MVT*	*ARCT*	*MVA*	*ARCA*	*ImpA*
LED7	2.450	3.23	92.127	96.397	4.63
LED24	8.237	15.82	82.814	88.641	7.04
SEA	4.696	9.82	87.682	89.920	2.554
PVA	14.16	52.46	36.17	53.36	47.53
UCI	2.70	22.63	34.1	37.79	10.82

6 Conclusion

In this paper we proposed a solution for predicting missing values, called Aerial Root Classifier with Hoeffding Tree Algorithm (ARC-HTA), which can perform data stream mining in the presence of missing values. The technique ARC is inspired by observing the adventitious roots in biology. ARC is a parallel process running along with HTA which is the main decision tree, which is reponsible for predicting missing values and for mining the data streams respectively. ARC-HTA integrates techinques of sliding window, feature selection and classification of DSM. The dynanmic ARC update mechanism is roburst to data stream even if concept-drift appears. A JAVA-based simulation is run as experimental platform. Experiment results unanimously indicate that ARC-HTA has a better performance in accuracy for mining data streams in the presence of missing values, than other existing methods. One reason is ascribed to the predictive power of ARC in contrast to the other statistical counting methods for handling missing values,

for ARC computes the information gains of almost all the other attributes that have non-missing data. As a future work, we opt to continue the investigation into the impact of noisy data or corrupted data as well as irregular data stream patterns in DSM.

References

[1] M.M.,Gaber,A.Zaslavsky and S. Krishnaswamy*Mining data streams: a review*, SIGMOD Rec. 34, 2, Jun. 2005, pp. 18-26.

[2] O.,Maron.,A.W.,Moore, *Hoeffding races: Accelerating Model Selection Search for Classification and Function Approximation*. Oded Maron, Andrew W. Moore NIPS, 1993, pp.59-66.

[3] H. Yang and S.Fong, *Investigating the Impact of Bursty Traffic on Hoeffding TreeAlgorithm in Stream Mining over Internet*, In proceeding of 2ndInternational Conference on Evelving Internet (INTERNET), 2010,Valencia, Spain, 2010, pp.147-152.

[4] H. Yang and S.Fong, *The Impacts of Data Stream Mining on Real-Time Business Intelligence*, 2nd International conference on IT and Budsiness Intelligence, Novermber 12-14, 2010, Nagpur, India. pp.9 -19.

[5] *Induction on decision tress. Machine Learnin*g, 1, 1986, pp. 81-106.

[6] J.R.,Quinlan,*C4.5: Programs for machine learning*. Morgan Kaufmann series in machine learning. Kluwer Academic Publishers, 1993.

[7] L.Breima, Friedman, J.H., Olshen, R.A., and C.J. Stone,*Classification and regression trees*, California, USA, Wadsworth, 1984.

[8] P. Domingos, and G.Hulten,*Mining high-speed data streams*. In Proceedings of the Sixth ACM SIGKDD international Conference on Knowledge Discovery and Data Mining,. ACM, New York, 2000, pp. 71-80.

[9] G.Hulten, L.Spencer,, and P.Domingos,*Mining time-changing data streams*. In Proceedings of the Seventh ACM SIGKDD international Conference on Knowledge Discovery and Data Mining (San Francisco, California, August 26 - 29, 2001). KDD '01. ACM, New York, NY, 2001, pp. 97-106.

[10] S.Nishimura, M.Terabe,K.Hashimoto and K.Mihara, *Learning Higher Accuracy Decision Trees from Concept Drifting Data Streams*. In Proceedings of the 21st international Conference on industrial, Engineering and Other Applications of Applied intelligent Systems: vol. 5027. Springer-Verlag, Heidelberg, 2008, pp.179-188.

[11] J.Gama, P. Medas and P.Rodrigues, *Learning decision trees from dynamic data streams*. In Proceedings of the 2005 ACM Symposium on Applied Computing, ACM, New York, 2005, pp.573-577.

[12] T. Wang, Z. Li, X.Hu, Y.Yan, and H.Chen. *A New Decision Tree Classification Method for Mining High-Speed Data Streams Based on Threaded Binary Search Trees*. Emerging Technologies in Knowledge Discovery and Data Mining. Springer. 2009, pp. 256-267.

[13] P. Pfahringer, G.Holmes, and R.Kirkby.*New Options for Hoeffding Trees*, Advances in Artificial Intelligence, Springer, 2007, pp. 90-99.

[14] Y.Ding and J. S.Simonoff, *An Investigation of Missing Data Methods for Classification Trees Applied to Binary Response Data*, J. Mach. Learn. Res. 11, 2010, pp. 131-170.

[15] K. Lakshminarayan, S.A. Harp and T. Samad,*Imputation of missing data in industrial databases*, Appl. Intell. 11 (1999), 259275.

[16] R.J. Little and D.B. Rubin,*Statistical Analysis with Missing Data*, Wiley, New York (1987).

[17] W. Nick Street and YongSeog Kim,*A streaming ensemble algorithm (SEA) for large-scale classification*, Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining, August 26-29, 2001, San Francisco, California. 2001, pp.377-382.

[18] H.Wang,F. Wei,S. Philip,J.Han,*Mining concept-drifting data streams using ensemble classifiers*, Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, August 24-27, 2003, Washington, D.C.

[19] S.Hashemi and Y.Yang, *Flexible decision tree for data stream classification in the presence of concept change*, noise and missing values. Data Min. Knowl. Discov. 19, 1 (Aug. 2009), 2009, pp. 95-13

[20] The Second International Knowledge Discovery andData Mining Tools Competition, Sponsored by theAmerican Association for Artificial Intelligence (AAAI)Epsilon Data Mining LaboratoryParalyzed Veterans of America (PVA). http://www.kdnuggets.com/meetings/kdd98/kdd-cup-98.html

[21] A.Frank and A.Asuncion. UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

Monitoring Multiple Streams with Dynamic Time Warping using Graphic Processors

Jason Chang
Dept. of Electrical Engineering
National Taiwan University
jsc@ntu.edu.tw

Mi-Yen Yeh
Institute of Information Science
Academia Sinica
miyen@iis.sinica.edu.tw

Abstract

In this paper, we present an approach for efficiently monitoring multiple data streams using graphic processor units (GPUs). Given reference patterns, similar subsequences in streams are matched under the dynamic time warping (DTW) distance and reported continuously. DTW distance is adopted since it offers scaling and shifting flexibility in the time axis. However, it suffers from high computation complexity, not to mention online matching among multiple streams. To overcome these issues, we exploit the advantages of GPUs: high levels of parallelism at low cost. We first show how to speed up DTW in a parallel way by using GPUs in CUDA language. Then, according to the existing online subsequence method under DTW distance, we propose GSPRING, which conducts online matching in multiple streams with multiple patterns concurrently by utilizing the massive threads of GPUs. We demonstrate that with the experiments on the NVIDIA graphic cards, our proposed algorithm can achieve speedup of up-to-15-times compared to a CPU-based approach when nearly a thousand streams are monitored simultaneously.

1 Introduction

Subsequence matching is an important application for monitoring data streams. Given a query pattern, users have considerable interest in continuously monitoring similar subsequences when a data stream continues to evolve. Examples include pattern matching in speech recognition and intrusion pattern detection in video surveillance. To adapt the high arrival speed and huge volume of data streams, a real-time approach is required.

The dynamic time warping (DTW) algorithm [1] is used to measure the similarity between two sequences of data with different lengths. It finds the optimal alignment between sequences and warps either one of them non-linearly by stretching or shrinking them along the time axis. Since it offers elastic time scaling and shifting capabilities in the similarity comparison, DTW has been widely used in many applications such as speech and handwriting recognition.

However, the computation cost of DTW is in quadratic time, not to mention the time complexity of searching all possible matched subsequences in data streams under DTW distance. To do this, an online algorithm, SPRING [2], is proposed. Suppose that the length of the given pattern is m, SPRING spends $O(m)$ time to identify if a subsequence is matched at the arrival of each data point. For a stream of length n, the total time cost for SPRING to report all matched subsequences is $O(mn)$. This computation cost is still very high if the length of the query pattern or the stream is long. Moreover, to the best of our knowledge, none of the existing subsequence matching methods using DTW distance yet meets the requirement of real-time responses in the environment of multiple query patterns and multiple streams.

In search of a better solution, we notice a growing interest in leveraging the high parallelism of programmable processors, especially Graphic Processing Units (GPUs). Originally designed to accelerate 2D/3D graphic renderings, these units are now used to cope with heavy workload in many research communities. The computing power of GPUs has increased so rapidly in recent years that, today a commonly available GPU may contain hundreds of processors and several gigabytes of memory. In single precision arithmetic computing, for example, GPUs outperform CPUs by orders of magnitude in terms of processing speed. As a result of the large number of processors and their excellent performance, GPUs are better suited for multithreading SIMD (Single Instruction Multiple Data) computation than conventional CPUs. Scientific researchers have been quick to discover the benefit of utilizing GPUs in general computation work, which helps the emergence of General-Purpose computing on Graphics Processing Units (GPGPU).

Among the GPU designers, NVIDIA is currently one of the largest. In addition to hardware equipment,

NVIDIA provides a software environment called Compute Unified Device Architecture (CUDA), which includes development toolkits and runtime infrastructure. In this environment, programmers can focus on developing codes, and are able to leave runtime scheduling and resource management issues to CUDA. The user programs can then be executed in a massively parallel fashion on hundreds of GPU processors.

To realize our objective of developing an approach that permits online monitoring of multiple data streams with multiple patterns using DTW, we propose GSPRING under NVIDIA's CUDA environment. First, we show how DTW can be parallelized using GPUs. To overcome the data dependency problem, we calculate the cells and fill the table along the diagonal direction. The degree of parallelism can be maximized to m, the length of query pattern, in optimal situations. Further parallelism is achieved by applying the computing model of CUDA. A two-dimensional array of blocks is declared and each block handles one single pair of stream comparison. Based on that fact that hundreds to thousands of comparisons can be processed simultaneously, we further show how GSPRING deals with continuously arriving streams and online reports matched subsequences. The challenge here is how to maximize the parallelization while reducing the communication overhead between CPU and GPUs. We attempt to fill the GPU memory with the monitored stream data and the query patterns to minimize data exchange frequency. In addition, kernel function returns only when all the data processing is completed or any subsequence match is found.

To demonstrate the superiority of the proposed DTW parallelization and GSPRING, we conduct extensive experiments using synthetic data sets. For comparisons, we also implement the original version of the DTW and SPRING methods on CPU. We measure the run time and memory usage of all methods. The results show that the parallelized DTW is 30 times faster than the sequential DTW on CPU, while GSPRING gains 15 times speed-up compared to SPRING on CPU.

This paper makes the following contributions:

1. A parallel algorithm GSPRING is proposed and implemented on NVIDIA's CUDA environment. It can be executed in a massively parallel fashion on hundreds of GPU processors.

2. GSPRING is capable of reporting matched subsequences under DTW distance in real time.

3. GSPRING realizes simultaneously monitoring of multiple streams with multiple query patterns in a continuously arriving data stream environment.

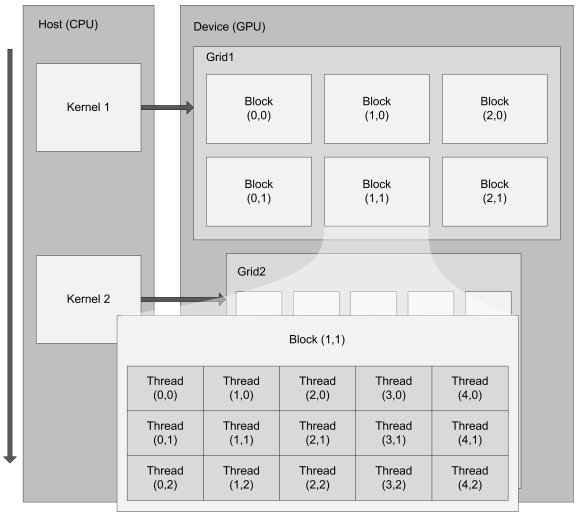


Figure 1: Thread execution model on CUDA. (Modified from [10])

4. Experiments show the speed-up of GSPRING on GPU may up to 15 times compared to SPRING on CPU.

The remainder of the paper is organized in the following way. Preliminaries are provided in Section 2, which includes background knowledge, the motivation, and the problem statement. Section 3 is dedicated to our parallel algorithm implementations, which is followed in Section 4 by presentation of our impressive experiment results. Finally, conclusions are drawn in Section 5.

2 Preliminaries

First, we introduce the GPU and CUDA programming model. This is followed by a brief review of fundamental concepts including DTW and SPRING algorithms. Finally, we use an example to demonstrate how SPRING algorithm works.

2.1 The GPU and CUDA programming model.

In recent years, GPUs have evolved from pure game or 3D applications to general-purpose computations, especially useful in scientific research. In the data mining and database communities, becoming increasingly popular is the use of GPU in applications such as clustering [3][4], relational joins, sorting, and tree search in databases [5][6][7], quantile and frequency counting in data streams [8], database compression [9], and so on.

CUDA is a parallel computing architecture developed by NVIDIA. It may be regarded as an extension of C language and is easy for programmers to learn. Us-

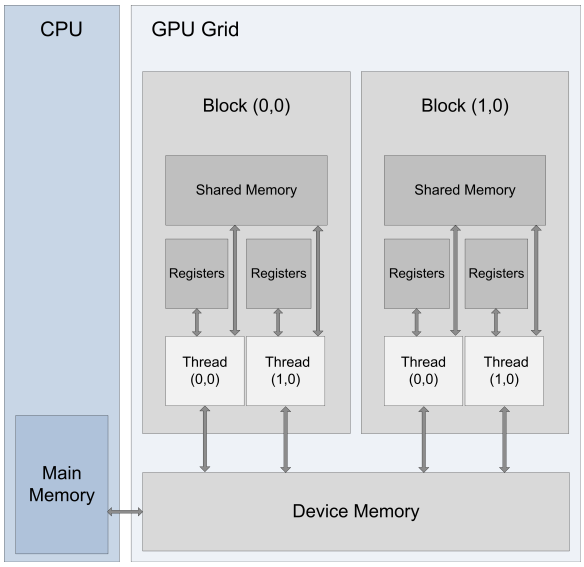


Figure 2: Memory model on CUDA. (Modified from [10])

ing CUDA enables programmers to create and manage large numbers of threads on NVIDIA GPUs. GPU procedures are also called *kernel functions*, executed over multiple threads. To fully utilize the computing power of GPUs, programs must be developed in a multithreading model. As demonstrated in Fig. 1,the basic execution unit in GPU is a thread. Up to 512 threads can be grouped into a thread group in a block. A grid is composed of multiple blocks aligned in a 2-dimensional array. Finally, a CUDA program can delcare an array of grids.

The memory model of GPUs is shown in Fig. 2. The register residing in the SIMD processors is the fastest in terms of read/write speed. It has the scope for only a single thread. The *shared memory* (SM), usually 16 KB, is efficient for information exchange among threads. Note that only the threads within the same block are able to share the same SM. The *device memory* (DM), which is the largest part of memory in a GPU, is globally available to all threads. The access to DM may incur significant overheads compared to access to registers and SMs. Finally, if there is any data exchange between CPU and GPU, the memory copy is carried out between DM and the main memory of CPU (MM), which is a costly operation.

In practice, CUDA programs use a large number of threads to execute the same piece of code over different data concurrently. This is the SIMD concept, which can avoid unnecessary program execution branches that significantly influence performance. Another benefit of the large number of threads is that they keep processors busy enough to hide the expensive latency of the device memory access.

2.2 Review of DTW and SPRING. The Dynamic Time Warping (DTW) distance between two sequences is computed as follows. Given two sequences $X = (x_1, x_2, ..., x_n)$ of length n and $Y = (y_1, y_2, ..., y_m)$ of length m, their DTW distance $D(X, Y)$ is defined as:

$$
\begin{aligned}
& D(X,Y) = f(n,m) \\
& f(t,i) = ||x_t - y_i|| + \min \begin{cases} f(t,i-1) \\ f(t-1,i) \\ f(t-1,i-1) \end{cases} \\
& f(0,0) = 0, f(t,0) = f(0,i) = \infty \\
& (t = 1, ..., n; i = 1, ..., m).
\end{aligned} \tag{2.1}
$$

where $||x_t - y_i|| = (x_t - y_i)^2$. The DTW distance is computed using a time warping table, which stores the values of the function $f(t, i)$ of (2.1). This is whole sequence matching since the two sequences are aligned head-to-head and end-to-end.

The SPRING algorithm, derived from DTW, is designed to online detect high-similarity subsequences of X when it is a semi-infinite data stream. Essentially, SPRING also computes a warping table but with some slight variations. There are two differences between SPRING and the original DTW. First, given a query pattern $Y = (y_1, y_2...y_m)$, one extra value is padded as $Y' = (y_0, y_1, ..., y_m)$, where $y_0 = (-\infty : +\infty)$ represents a don't care value and always gives zero distance. Let X be a stream of length n, the minimum distance $D(X[t_s : t_e], Y)$ can be derived from the warping table of X and Y'. The computation is shown in (2.2). Each cell in the table keeps both the distance value $d(t, i)$ and the starting position, $s(t, i)$, of the current $d(t, i)$ value (Fig. 3).

$$
\begin{aligned}
& D(X[t_s : t_e], Y) = d(t_e, m) = \min\{d(t,m)\} \\
& d(t,i) = ||x_t - y_i|| + d_{best} \\
& d_{best} = \min \begin{cases} d(t,i-1) \\ d(t-1,i) \\ d(t-1,i-1) \end{cases} \\
& d(t,0) = 0, d(0,i) = \infty \\
& (t = 1, ..., n; i = 1, ..., m).
\end{aligned} \tag{2.2}
$$

In addition to the distance $d(t, i)$, the matrix contains the starting position:

(2.3)

$$s(t,i) = \begin{cases} s(t,i-1), & d(t,i-1) = d_{best} \\ s(t-1,i), & d(t-1,i) = d_{best} \\ s(t-1,i-1), & d(t-1,i-1) = d_{best}. \end{cases}$$

Finally, we can obtain the starting position of $D(X[t_s : t_e], Y)$ as:

(2.4)
$$t_s = s(t_e, m)$$

The second difference is that SPRING continues to track the minimum distance d_{min}, the start point t_s, and the end point t_e of the candidate subsequences obtained thus far. Since in SPRING, among all overlapped candidate subsequences, only the one with smallest distance is reported. The d_{min} value continues to be compared with the given threshold ϵ. Once the qualified subsequence is guaranteed not to be replaced by further upcoming subsequences, SPRING reports it and its corresponding d_{min}, t_s, and t_e. Then, after resetting some cells in the table, SPRING repeats the matching algorithm. Due to space limitations, the detailed algorithm of SPRING can be found at [2].

y_4=4	54 (1)	110 (2)	14 (2)	38 (2)	6 (2)	7 (2)	88 (2)	∞ (2)
y_3=9	53 (1)	46 (2)	10 (2)	2 (2)	10 (4)	17 (4)	18 (4)	∞ (4)
y_2=6	37 (1)	37 (2)	1 (2)	17 (4)	1 (4)	2 (4)	51 (4)	∞ (4)
y_1=11	36 (1)	1 (2)	25 (3)	1 (4)	25 (5)	36 (6)	4 (7)	4 (7)
x_t	5	12	6	10	6	5	13	13
t	1	2	3	4	5	6	7	7

Figure 3: Illustration of the SPRING Algorithm

To demonstrate how SPRING algorithm works, an illustrative example adapted from [2] is given in Fig. 3. Suppose we have $\epsilon = 15$, $X = (5, 12, 6, 10, 6, 5, 13)$, and $Y = (11, 6, 9, 4)$. Each cell (t, i) in the table contains $d(t, i)$, at upper, and $s(t, i)$, in parenthesis. When $t = 3$, the distance $d(3, 4)$ of the candidate subsequence $X[2 : 3]$ is 14, which is below ϵ. At $t = 4$, $d(4, 4) = 38$, which is larger than ϵ, but we do not report $X[2 : 3]$ since $d(4, 3) = 2$. This is because $X[2 : 3]$ might be replaced by the upcoming and overlapped subsequences. At $t = 5$, the optimal subsequence $X[2 : 5]$ is recognized. It is reported until $t = 7$ since at this time we can ensure that none of the upcoming subsequences will have smaller distance and overlap it. After the reporting, we need to investigate the cells of column $t = 7$. For cells whose $s_i \leq t_e$, their distance value needs to be reset to infinity. The final state of column $t = 7$ is shown in the rightmost column of Fig. 3. As a result, except for cell $d(7, 1)$, distance values in other cells are all reset.

3 Multiple Stream Monitoring under DTW using GPUs

We state the problem as the following: Given a set of user-given query patterns $Y_1, Y_2, ...Y_M$ with length m, our goal is to monitor them among multiple data streams $X_1, X_2, ..., X_N$ concurrently. Under the DTW distance, an immediate report is given as long as any subsequence is found to satisfy a given error threshold. To achieve this goal, we parallelize the SPRING algorithm and port the program to the GPU environment using CUDA. The modified GPU version of SPRING algorithm is named *GSPRING*.

In the following sections, first we give an overview of GSPRING at §3.1. How the endless streams are loaded to the GPU is discussed as well. Then, two important functions of the GSPRING kernel are illustrated in §3.2 and §3.3, respectively. Finally, we show how multiple streams can be monitored with multiple query patterns in §3.4.

3.1 Overview of GSPRING. The workflow and algorithm of GSPRING are shown in Fig. 4 and Algorithm 1, respectively. In our implementation, CPU controls most of the program flow while GPU focuses on computing. Initially, the CPU needs to load incoming data streams and copy them to the DM of GPU. In our design, GSPRING loads n data points each time for each stream. After launching the GSPRING kernel, which is the whole procedure done by GPU as shown in the blue shaded area in Fig. 4, the control transfers to the GPU while the CPU remains idle. The GPU starts those heavy computing jobs such as filling the DTW table and checking if any subsequence is found. Once a qualified subsequence appears, the kernel function stops and returns the control to the CPU. The matched results must be copied back from the DM of the GPU and be output. As a feature of a data stream environment, the program loops repeatedly to deal with the continuously arriving data.

To port the SPRING algorithm on CUDA, we must break down all tasks and clarify which parts should be implemented on GPU. The GSPRING kernel function of our implementation is shown in Procedure 1. An important contribution of GSPRING is that it parallelized the most time-consuming tasks of SPRING on CUDA. They are TableFilling, which handles the DTW table filling work of SPRING, and CHECKREPORT, which checks if a candidate subsequence should be reported and report it. Moreover, because of the control change between CPU

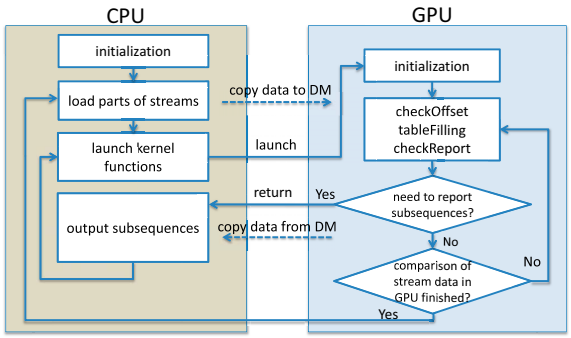


Figure 4: The GSPRING algorithm diagram: Solid lines represent program flow. Dotted lines represent data exchange between CPU and GPU.

and GPU, we should have a mechanism that keeps the current position of the compared stream pair. This is to make sure the threads can continue their computing works accurately in the next kernel launch. Note that data consistency is a key issue in parallel computing. Programs should guarantee that data writing is completed before the data is read. For example, to avoid possible race conditions, thread synchronization should be conducted at the end of each step after calling of TableFilling completes.

Algorithm 1 GSPRING

Input:
$X_i, i = 1..N$ continously arriving data streams
$Y_j, j = 1..M$ query patterns with length=m

```
1: copy Y_1...Y_M and ε to DM of GPU
2: while read data until length of each X_i = n do
3:     copy X_1...X_N to DM of GPU
4:     finish ← 0
5:     while finish ≠ 1 do
6:         launch GSPRING kernel function()
7:         copy report and offset data from DM of GPU
8:         finish ← 1
9:         for all k ∈ X_i do
10:            if subsequence is found then
11:                report d_min, t_s, t_e
12:            end if
13:            if offset ≤ n then
14:                finish ← 0
15:            end if
16:        end for
17:    end while
18: end while
```

Procedure 1 GSPRING kernel function

Input:
N: number of X_i
M: number of Y_j
n: length of each X_i
m: length of each Y_j
ϵ, report threshold

```
1: bi ← blockIdx.y * gridDim.x + blockIdx.x   ▷ block id in the grid
2: tid ← threadIdx.x                          ▷ thread id in the block
3: shared d_min, t_s, t_e, report             ▷ shared by all threads in the block
4: read offset from DM
5: if offset > n then
6:     return                                 ▷ end kernel,return to host
7: end if
8: step ← 0
9: while (step + offset) < (n + m + 1)) ∧ (report ≠ 1) do
10:    TABLEFILLING(step, tid, offset)
11:    __syncthreads()
12:    if step ≥ m then                       ▷ thread 1 at top of column
13:        CHECKREPORT(tid, ε, d_min, t_s, t_e)
14:        if tid = 1 ∧ dtw[x][y].d ≤ eps ∧ dtw[x][y].d < d_min then
15:            if a matched subseqence should be reported then
16:                keep current dtw cell for future ref
17:            end if
18:        end if
19:        if reach the last column of DTW table then
20:            keep current dtw cells for future ref
21:        end if
22:    end if
23:    step ← step + 1
24:    __syncthreads()
25: end while
```

3.2 Parallelization of the DTW table filling using CUDA. The first task is the DTW table filling. Due to the data-dependent nature of the DTW table, intuitive parallel programming methods are not easily applied. Further investigation of (2.1)(2.2) and its computing flow provides insight into how to overcome the data dependency problem. Fig. 5(a) shows that since $d(i,j)$ depends on $d(i-i,j)$, $d(i,j-1)$ and $d(i-1,j-1)$, we can calculate $d(i+1,j)$ and $d(i,j+1)$ simultaneously once $d(i,j)$ is obtained. Therefore, a parallel DTW approach is achieved by implementing the computation flow proceeding in the diagonal direction.

Given two sequences, a stream $X = (x_1, x_2, ..., x_n)$ of length n that is under processing, and the query

Procedure 2 TableFilling

```
1:  procedure TABLEFILLING(step,tid,offset)
2:      if tid ≤ step then
3:          if step < m then
4:              x ← tid + offset − 1      ▷ calculate the x,y
5:              y ← step − tid + 1                ▷ coordinates
6:          else
7:              x ← step − m + tid + offset − 1
8:              y ← m − tid + 1
9:          end if
10:         dtw[x][y].d ← ||x_t − y_i|| + min(dtw[x − 1][y],
                          dtw[x][y − 1], dtw[x − 1][y − 1])
11:         dtw[x][y].s ← starting position, depends
                          on the source of above min value
12:     end if
13: end procedure
```

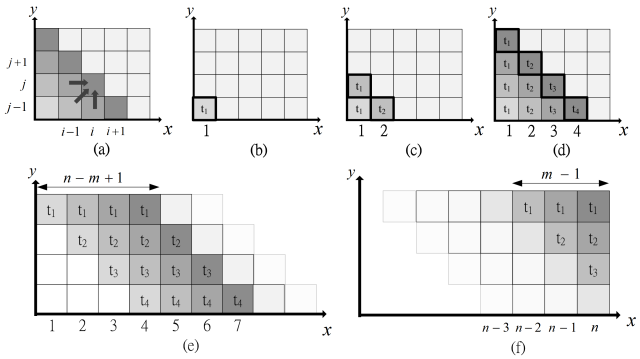


Figure 5: (a)The data dependency of the DTW table. (b)-(f)Steps of the DTW parallelization.

pattern $Y = (y_1, y_2, ..., y_m)$ of length m, the detailed parallelization procedure is shown in Fig. 5(b)(c)(d). In the first step, there is only one thread (t1) active, which calculates $d(1,1)$. This enables $d(2,1)$ and $d(1,2)$ to be calculated by thread $t1$ and $t2$ in the second step. The maximum degree of parallelism appears when the step number equals m and thereafter, meaning that a total of m threads can work at the same time, as shown in Fig. 5(d). In practice, the maximum degree of parallelism can not exceed the actual number of threads ($\#t$) provided by GPU, which is 512 on CUDA. If $m > \#t$, however, we can still resolve it by assigning some threads to handle more than one cell of computation. The only trade-off is slightly lower performance due to the task switching of those threads.

On the other hand, to reduce the expensive memory access latency, we use SM instead of DM for the data exchange by threads within the same block. Due to the size limit of SM, only the most frequently used data including the current and previous two diagonals are stored in SM (Fig. 5(a)). This results in a significant improvement in DTW TableFilling work.

It is noted that, when the table filling work reaches the end of the currently loaded X sequence, the values in cells of the last column, column n at Fig. 5(f) must be kept (line 19-21 in GSPRING kernel) for referencing in the next new loop of the GSPRING kernel function.

The pseudo code of our CUDA implementation is shown in Procedure 2 TableFilling where *offset* represents the starting position of the table to compute at each kernel launch. This procedure is called by every thread at each step with step number *step* and thread id *tid* as input. The $dtw[x][y]$ represents a cell on DTW table stores the $d(t,i)$ and $s(t,i)$ value of (2.2) and (2.3) respectively. The number of pre-allocated working threads equals m. However, active thread numbers correspond to the current step number. When the current step number is less than m, only the step number of threads is active, with the others remaining idle(line2). When the step number is equivalent to or greater than m, all the threads will be active and working. Since each thread executes the same piece of code over different parts of data, it has to maintain its own index including the x and y coordinates of the currently processed DTW table in order to indicate the cell to which it is responsible. This is referred to at line 3-9 of TableFilling.

Finally, the total number of steps of the computation is analyzed as follows. For the original SPRING algorithm, the required steps equal the number of cells, also indicating the table size. Thus, $m \times n$ represents the total number of steps. The complexity is $O(mn)$. For our parallel version, we divide the computation into three parts. First, the triangular part before the step number equals m which requires $m-1$ steps, Fig. 5(b)(c). The second part is the steps of which the active thread numbers equal m, shown in Fig. 5(d)(e). Third, the remaining part, where the active thread numbers are less than m, is depicted in Fig. 5(f), and is similar to the first part. Hence, the total number of steps of parallel DTW is $(m-1)+(n-m+1)+(m-1) = m+n-1$ and the complexity is $O(m+n-1)$.

3.3 Check and Report of Matched Subsequences While the TableFilling computes the DTW distance between the subsequence of a stream and a query pattern, the task of checking and reporting matched subsequences is done by the procedure CheckReport in Procedure 3. When current d_{min} is a qualified candidate (line2), all threads execute line 3-10 to verify against their assigned cells. If a report is necessary, all threads verify if a reset is necessary for each cell in the current column as indicated in line 17-19. Again,

to guarantee data consistency among multiple threads, the __syncthreads() function provided by CUDA must be called after the concurrent data writes by multiple threads. This is done in line 8-9 of CheckReport. Although calling __syncthreads() may slow down the overall performance, it is unavoidable in a parallel computing environment.

It is noted that before the report to CPU, the current column under computing, say column k at Fig. 6(b), must be kept. This is shown in the line 14-18 in GSPRING kernel). On the other hand, all the threads in the position keeping task as well as the CHECKREPORT function itself are fully utilized, most of the working steps can be reduced from m to 1 as long as $m <= 512$. For $m > 512$, total working steps can be reduced to $\lceil m/512 \rceil$.

Before the end of the describing of TABILEFILING and CHECKREPORT tasks, we have the following implementation review and remarks.

1. The main program on the CPU loads incoming data streams and temporarily retains them. It should only copy the data to the DM on the GPU and launch the kernel function until enough sequences are prepared, such as $length(X_i) \geq n$. This is a tradeoff between reducing the memory copy overhead and delaying the report response time.

2. When GPU deals with the new coming n points of the stream, calculation of the first column requires reference to ascendant data. It is vital to handle with considerable care the computation work, especially at the beginning and end of each currently loaded X sequence of length n. When reaching the end of the sequence, the values in cells of the last column should be retained(Fig. 5(f)). At the beginning of the processing, it is necessary to reference to previously retained position for the correct distance computation.

3. Once the current step number equals or is larger than m, i.e. when thread 1 is processing the cell on the top row of the table, the subsequence match checks, as in the procedure CheckReport need to be conducted at each step(line 12-13 of GSPRING). When finding a match as in Fig. 6(b), the program increases its offset to the next value, writes report data to DM and prepares to return to CPU.

4. In subsequent GPU kernel launches, the program must check its offset value when entering the GPU, as the situation if Fig. 6(c). It proceeds only when the offset value is not greater than n. It is necessary for the parallelized table-filling process to resume and to gradually increase its degree of parallelism (Fig. 6(c)(d)).

5. Upon finding a matched subsequence or reaching the end of the processing stream, it is important to check if it is necessary to keep the d and t_s values of the current cell as in Fig. 6(b), before leaving the kernel. If the current d value is smaller than ϵ, it can be a candidate of a future subsequence match. If it is not kept, a piece of a future subsequence match might be lost when the kernel is re-entered.

Procedure 3 CheckReport

```
1: procedure CHECKREPORT(tid,ε,d_min,t_s,t_e)
2:     if d_min ≤ ε then
3:         report ← 1
4:         x ← step − m + offset          ▷ x position of
                                          ▷ current column
5:         __syncthreads()
6:     end if
7:     if (¬(dtw[x][tid].d ≥ d_min))∨(dtw[x][tid].s > t_e)
   then
8:         report ← 0
9:         __syncthreads()
10:    end if
11:    if report = 1 then
12:        if tid = 1 then        ▷ only thread 1 handles
13:            report d_min, t_s, t_e      ▷ the report work
14:            offset ← x + 1
15:            d_min ← ∞
16:        end if
17:        if dtw[x][tid].s ≤ t_e then  ▷ reset the values
18:            dtw[x][tid].d ← ∞    ▷ of current column
19:        end if
20:    end if
21: end procedure
```

3.4 Multiple Subsequence Matching over Multiple Streams To realize simultaneously monitoring of multiple streams over multiple query patterns, we here show how GSPRING on CUDA conduct large numbers of comparisons concurrently. As in Fig. 4, all the streams to be compared are copied to the DM of the GPU in the beginning. We declare a grid with a two-dimensional array of blocks in CUDA. The dimension of the two axes are N (number of X streams) and M (number of Y patterns) respectively. Each block contains m threads and executes one pair of the comparison. The execution model is shown as Fig. 1, where there are a total of $N \times M$ blocks executing $N \times M$ comparisons in the grid. Usually, $N \times M \times m$ will be much larger than the actual processor numbers on GPU. However,

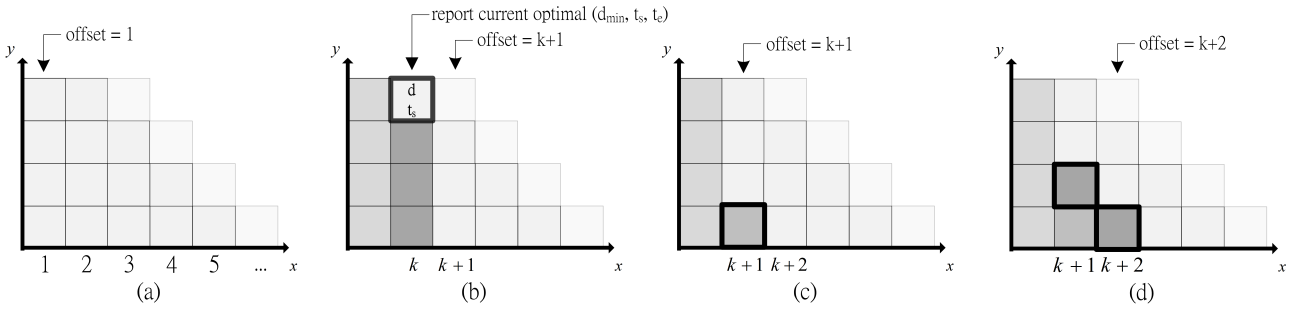


Figure 6: Some special steps of GSPRING algorithm. (a) First time entering the kernel, program begins at offset=1. (b) When a subsequence is found, report the current (d_{min},t_s,t_e) values and increase the offset value. (c)(d) Subsequent kernel entering, program begins at specific offset position. The degree of parallelism needs to increase gradually from 1.

it is not a problem since CUDA will do the resource scheduling for the overall program execution. The program flow is the same as in Fig. 4.

Every time when the new n points of each stream are copied to DM, all the blocks start their computing from $offset = 1$, as in Fig. 6(a) at the first kernel launch. Blocks that do not find any matches shall proceed along to the end of the currently loaded stream ($offset = n$) and stop. If any blocks have matched subsequences to report, they stop at the current position. In the next kernel launches, only blocks with offset value not exceeding n, as the case in Fig. 6(c) need to carry on computing. Obviously, the execution time of the subsequent kernel launches is much shorter than the first.

Since every thread among all the blocks executes the same piece of code over a shared memory space(DM), it has to maintain its own indices including the thread index inside a block and the block index inside a grid. This can be achieved by using CUDA's built-in variables such as *gridDim, blockDim, blockIdx* and *threadIdx*. In our implementation, the DM space in CUDA stores all the query patterns $Y_1, Y_2, ...Y_M$ and data streams $X_1, X_2, ..., X_N$. In addition, the DTW table, the offset and the report structures of each pair of comparison are also stored in DM. However, DM in CUDA is shared among all the threads and does not provide any access protection mechanism. To avoid data corruption, all the memory read/write operations must be carefully executed by keeping the correct address and offset.

When executing the GSPRING kernel, each block proceeds independently without any data exchange between blocks. As a result, synchronization between blocks is not necessary. However, threads within the same block still need to synchronize their pace at some critical checkpoints to avoid race conditions.

4 Performance Evaluation

To evaluate the performance of subsequence matching using GPU, we conduct several experiments. First we implement the original sequential version of DTW and SPRING algorithm using C language on the CPU. Next, we implement the parallel version of DTW and GSPRING algorithm using CUDA on GPU. Query patterns and stream sequences are composed of synthetic random-walk data. The experiments are performed on a server with dual Intel E5420 2.5GHz CPU. A NVIDIA Tesla C1060 GPU with 240 SIMD-processors and 4GB memory is connected by PCI-Express interface. The operating system is Linux 2.6.18 and the version of CUDA is 3.0.

4.1 Performance of Sequential versus Parallel version of DTW. We evaluated the total computation time of both the sequential and the parallel DTW when the number of sequences pairs to be compared varied from 6.2k to 62k with stream $length = 8192$. The results are shown in Fig. 7. We can see the speedup between CPU and GPU is constantly around 30 times.

4.2 Performance of SPRING versus GSPRING. We observed the speedup of GPU under the impact of the compared sequence pair number, the query pattern length, and the report rate. We first presented the experimental results of under each impact. Then, an overall discussion is given.

Experiments were conducted using the following parameters: total stream $length = 8192$, the segment length sending to GPU each time $n = 1024$ and the similarity threshold was set to $\epsilon = 35000$. We show the total computation time of processing the whole streams in seconds, and the speedup under different settings.

Impact of compared sequence pair numbers:

First we fixed the query pattern length as $m = 512$, and varied the total compared sequence pair numbers $N \times M$ from 100 (10×10) to 961 (31×31), as shown in Fig. 8(a), the speedup increased from 7.98 to 15.33. GSPRING outperformed SPRING when large numbers of pairs are compared concurrently.

Impact of query pattern length: Here we fixed the total compared sequence pair number as $N \times M = 30 \times 30$, and varied the length of query pattern m from 64, 128, 256, 384 to 512. As shown in Fig. 8(b), the speedup increased from 7.27 to 15.24. The speedup value of $m = 384$ is 15.16, which is extremely close to 15.24 when $m = 512$.

Impact of report rate: Finally, varying the ϵ value may change the output number of the matched subsequences. By setting ϵ=45000, we have a very loose criterion which results in large amount of matches. This was the baseline for this experiment and defined as 100% report rate. When the threshold is as high as ϵ=30000, no matches could be found. We defined it as 0% report rate. According to this experiment, our default ϵ=35000 indicated its report rate as 54.4%.As a result, the speedup value for 0% report rate is 15.2 while it is 13.82 for 100% report rate.

4.3 Discussion. The speedup performance of parallelized DTW was double that of GSPRING. This can be explained by the fact that the DTW algorithm is extremely straightforward without any branch conditions. This kind of algorithm is most suitable in a SIMD-based parallel environment. In contrast, GSPRING needs to check the report condition at the end of every loop. When GSPRING reports matched subsequences, the first thread can cover most of the preparation works, as shown at line 11-14 in Fig. 6(b). In addition, the memory copy between CPU and GPU incurs significant overheads. For these reasons, the speedup of GSPRING was not as significant as the parallelized DTW.

To fully utilize the advantage of the GPU is to launch as many threads concurrently as possible because the large numbers of threads can sufficiently occupy the SIMD-processors and maintain the overall computing throughput at a high level. As explained in §4.2, we allocate a block for each pair and 512 threads for each block. Obviously, 100 pairs are not enough to saturate the processors of GPU. With the number rising to 961, the performance nearly doubles as shown in Fig. 8(a). In addition, since the thread number in each block equals the length of query patterns m, it is evident in Fig. 8(b) that GPU performance cannot be squeezed to the limit until m reaches 384. Finally, to report a matched subsequence incurs lots of works including leaving GPU kernel function, returning to CPU and memory copy from DM of GPU to CPU. More reports imply more overheads and result in worse performance. Not to mention the reported subsequences could be meaningless if threshold was set too low. Thus, selecting a reasonable threshold ϵ helps to maintain high performance and reporting quality.

5 Conclusion

In this study, first we showed how to parallelize DTW on GPUs. Then the similar parallelization was adopted for developing GSPRING, along with the online matching and reporting mechanism designed between CPU and GPUs. Moreover, the parallelization of multiple subsequence matching among multiple streams on GPUs is designed. The experiment results showed our GPU-based methods had a significant speedup in the computation time. Especially, GSPRING outperformed SPRING by an up-to-15-times speedup, which indicates the realization of the online real-time response in the multiple stream environments.

References

[1] H. Sakoe and S. Chiba, *Dynamic programming algorithm optimization for spoken word recognition*, IEEE TASSP, 26 (1978), pp. 43–49.

[2] Y. Sakurai, C. Faloutsos and M. Yamamuro, *Stream Monitoring under the Time Warping Distance*, In Proceedings of ICDE'07, pp. 1046–1055, 2007.

[3] S. A. Shalom, M. Dash and M. Tue, *Efficient K-Means Clustering Using Accelerated Graphics Processors*, In Proceedings of DaWaK'08, pp. 166-175, 2008.

[4] C. Böhm, R. Noll, C. Plant and B. Wackersreuther, *Density-based clustering using graphics processors*, In Proceedings of CIKM'09, pp.661-670, 2009.

[5] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo and P. Sander, *Relational joins on graphics processors*, In Proceedings of SIGMOD'08, pp. 511-524,2008.

[6] N. Govindaraju, J. Gray, R. Kumar and D. Manocha, *GPUTeraSort: high performance graphics co-processor sorting for large database management*, In Proceedings of SIGMOD'06, pp. 325-336,2006.

[7] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt and P. Dubey, *FAST: fast architecture sensitive tree search on modern CPUs and GPUs*, In Proceedings of SIGMOD'10, pp. 339-350, 2010.

[8] N. Govindaraju, N. Raghuvanshi, and D. Manocha, *Fast and approximate stream mining of quantiles and frequencies using graphics processors*, In Proceedings of SIGMOD'05, pp. 611-622, 2005.

[9] W. Fang, B. He and Q. Luo, *Database Compression on Graphics Processors*, In Proceedings of VLDB'10,2010.

[10] *NVIDIA CUDA Programming Guide Version 3.0*, 2010.

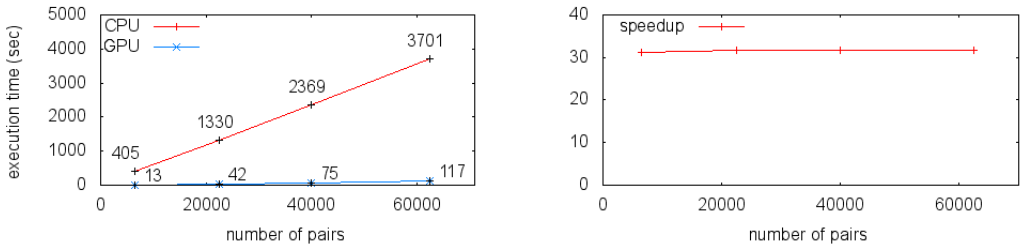
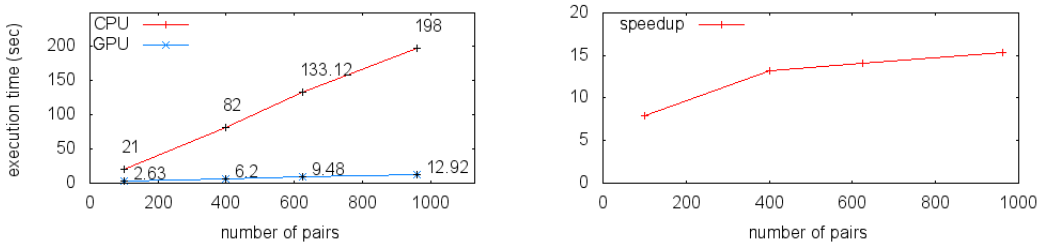
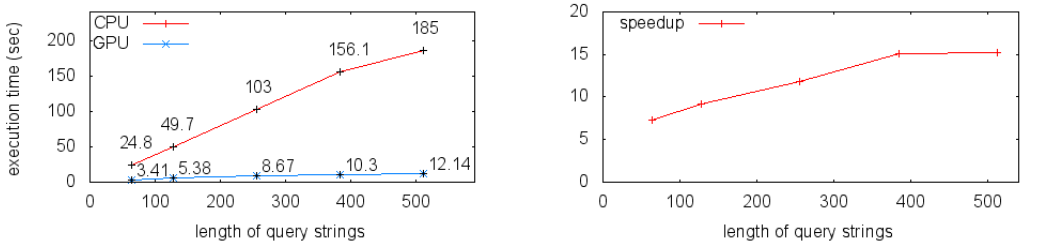


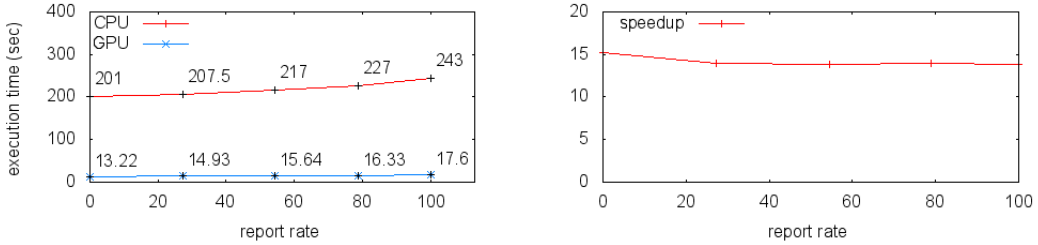
Figure 7: Runtime and speedup of sequential vs. parallel DTW algorithm.



(a) Impact of compared sequence pairs on runtime and speedup.



(b) Impact of query pattern length on runtime and speedup.



(c) Impact of report rate on runtime and speedup.

Figure 8: Experimental evaluation of SPRING on CPU vs. GSPRING on GPU.

Supporting Dynamic Load Balancing in a Parallel Data Mining Middleware

Tekin Bicer Gagan Agrawal
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{*bicer,agrawal*}*@cse.ohio-state.edu*

Abstract

As parallel data mining applications are being executed in grid and cloud settings, there is a need for considering virtualized, non-dedicated, and/or heterogeneous environments. Supporting dynamic load balancing becomes an important challenge in such environments. Particularly, two important problems that need to be addressed are: Optimal distribution of tasks among disparate processing units and minimizing the runtime overhead of the system. With these goals, this paper describes and evaluates an approach for enabling parallel data mining with dynamic load balancing. Our approach is based on an API which deals with independent data elements that can be processed by any processing resource in the system.

We have extensively evaluated our dynamic load balancing system using two parallel data mining applications. Our results show that the overheads of our scheme are extremely low. Furthermore, our system successfully distributes tasks among processing units even in highly heterogeneous configurations.

1 Introduction

Increasingly, data mining needs to be performed in grid and cloud environments, leading to the supporting execution in non-dedicated and/or heterogeneous clusters. As science has become increasingly data-driven, support for data-intensive computing is becoming a crucial component of the cyber-infrastructure or e-science. For example, *community-driven data grids* have received significant attention recently [2]. Grids inherently comprise heterogeneous resources, and often include non-dedicated use of resources.

More recently, the trend is towards data-intensive computing on the emerging cloud environments. Two common characteristics of cloud environments are also leading to non-dedicated use of resources, and/or execution in heterogeneous environments. The first is the use of virtualization technologies, which enable applications to set up and deploy a customized virtual environment suitable for their execution. The second is the *pay-as-you-go* model for resource allocation and pricing. Consistent with the utility vision of computing, recent research points to the progression of clouds towards supporting fine-grained sharing of CPU cycles (and memory) between instances [3, 4]. Current virtualization technologies (for example, Xen [5]) can already allow a change in CPU cycle percentage and/or memory allocation at any point during the execution. Thus, in a cloud environment, it is quite possible that an application may be executed on a set of machines that differ in CPU cycle percentage allocation, and furthermore, this allocation can even change over time for each node.

Overall, there is clearly a need for executing data-intensive applications with dynamic load balancing, and harnessing the net processing power available in the cluster. This paper presents an approach for addressing this problem. Our approach has been implemented in the context of a data-intensive computing middleware, FREERIDE-G [6, 7]. This middleware system uses a specialized API for developing scalable data-intensive applications. It supports *remote data analysis*, which implies that data is processed on a different set of nodes than the ones in which it is hosted. Our work on supporting dynamic load balancing exploits the processing structure supported by our API, particularly, the fact that independent data elements can be processed by any processing resources in the system. This enables us to dynamically assign tasks to the processing units without considering the order or dependencies of the tasks.

We have evaluated how effectively our dynamic load balancing system can perform using two data mining applications. Our results show that the overheads of our system are negligible. Furthermore, our load balancing approach can effectively distribute jobs among the processing units even in highly heterogeneous configurations.

2 Background

This section gives an overview of an API on which our work is based. We then describe the remote data analysis paradigm and the FREERIDE-G system, which uses this API and supports remote data analysis.

FREERIDE

```
{* Outer Sequential Loop *}
While() {
    {* Reduction Loop *}
    Foreach(element e) {
        (i, val)    =   Process(e) ;
        RObj(i)     =   Reduce(RObj(i),val) ;
    }
    Global Reduction to Combine RObj
}
```

Map-Reduce

```
{* Outer Sequential Loop *}
While() {
    {* Reduction Loop *}
    Foreach(element e) {
        (i, val)    =   Process(e) ;
    }
    Sort (i,val) pairs using i
    Reduce to compute each RObj(i)
}
```

Figure 1: Processing Structure: FREERIDE(top) and Map-Reduce(bottom)

2.1 API for Parallel Data-Intensive Computing: Before describing our alternative API, we initially review the map-reduce API which is now being widely used for data-intensive computing.

The map-reduce programming model can be summarized as follows [8]. The user of the map-reduce library expresses the computation as two functions: *Map* and *Reduce*. *Map*, written by the user, takes a set of input points and produces a set of intermediate $\{key, value\}$ pairs. The map-reduce library groups together all intermediate values associated with the same key and passes them to the *Reduce* function. The *Reduce* function, also written by the user, accepts a key and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically, only zero or one output value is produced per *Reduce* invocation.

Now, we describe the alternative API this work is based on. This API has been used in a data-intensive computing middleware, FREERIDE, developed at Ohio State [9, 10]. This middleware system for cluster-based data-intensive processing shares many similarities with the map-reduce framework. However, there are some subtle but important differences in the API offered by these two systems. First, FREERIDE allows developers to explicitly declare a reduction object and perform updates to its elements directly, while in Hadoop/map-reduce, the reduction object is implicit and not exposed to the application programmer. Another important distinction is that, in Hadoop/map-reduce, all data elements are processed in the map step and the intermediate results are then combined in the reduce step, whereas in FREERIDE, both map and reduce steps are combined into a single step in which each data element is processed and reduced before the next data element is processed. This choice of design avoids the overhead due to sorting, grouping, and shuffling, which can be significant costs in a map-reduce implementation.

Hadoop/map-reduce provides *Combiner* function which can partially decrease the sorting, grouping and data transfer overheads. More specifically, if a Combiner function is defined in the system, the pairs are grouped in different lists according to their key values on local machine. When the number of pairs exceeds a threshold, Combiner function reduces the pairs and emits the new ones. Typically, Combiner function is similar to Reduce function, however it processes the pairs that are already in local memory. Reduce function, on the otherhand, needs to collect the emitted pairs. FREERIDE-G processing structure naturally accumulates $\{key, value\}$ pairs right after their generation which avoids the mentioned overheads in map-reduce.

The following functions must be written by an application developer as part of the API:

Local Reductions: The data instances owned by a processor and belonging to the subset specified are read. A local reduction function specifies how, after processing one data instance, a *reduction object* (declared by the programmer), is updated. The result of this process must be independent of the order in which data instances are processed on each processor. The order in which data instances are read from the disks is determined by the runtime system.

Global Reductions: The reduction objects on all processors are combined using a global reduction function.

Iterator: A parallel data-intensive application comprises of one or more distinct pairs of local and global reduction functions, which may be invoked in an iterative fashion. An iterator function specifies a loop which is initiated after the initial processing and invokes local and global reduction functions.

Throughout the execution of the application, the reduction object is maintained in main memory. After every iteration of processing all data instances, the results from multiple threads in a single node are combined locally depending on the shared memory technique chosen by the application developer. After local combination, the results produced by all nodes in a cluster are combined again to form the final result, which is the global combination phase. The global combination phase can be achieved by a simple all-to-one reduce algorithm. If the size of the reduction object is large, both local and global combination phases perform a parallel merge to speed up the process. The local combination and the communication involved in the global combination phase are handled internally by the middleware and is transparent

to the application programmer.

Fig. 1 further illustrates the distinction in the processing structure enabled by FREERIDE and map-reduce. The function $Reduce$ is an associative and commutative function. Thus, the iterations of the for-each loop can be performed in any order. The data-structure *RObj* is referred to as the reduction object.

Our recent work has shown a substantial performance improvement with our API [11]. In addition, we believe that this API offers a significant advantage in supporting dynamic load balancing. Since, almost all of the execution time is spent in the local reduction stage, the processing can be distributed between the nodes in a non-uniform and dynamic fashion. In comparison, with a map-reduce API, for most applications, significant amount of time is spent on both map and reduce stages. Moreover, the reduce stage is dependent on a large intermediate data structure, which can make dynamic load balancing very difficult to support.

2.2 Remote Data Analysis and FREERIDE-G: Our support for dynamic load balancing is in the context of supporting *transparent remote data analysis*. In this model, the resources hosting the data, the resources processing the data, and the user may all be at distinct locations. Furthermore, the user may not even be aware of the specific locations of data hosting and data processing resources.

If we separate the concern for supporting dynamic load balancing, co-locating data and computation, if feasible, achieves the best performance. However, there are several scenarios co-locating data and computation may not be possible. For example, in using a networked set of clusters within an organizational grid for a data processing task, the processing of data may not always be possible where the data is resident. There could be several reasons for this. First, a data repository may be a shared resource, and cannot allow a large number of cycles to be used for processing of data. Second, certain types of processing may only be possible, or preferable, at a different cluster. Furthermore, grid technologies have enabled the development of *virtual organizations* [12], where data hosting and data processing resources may be geographically distributed.

The same can also apply in cloud or utility computing. A system like Amazon's Elastic Compute Cloud has a separate cost for the data that is hosted, and for the computing cycles that are used. A research group sharing a dataset may prefer to use their own resources for hosting the data. The research group which is processing this data may use a different set of resources, possibly from a utility provider, and may want to just pay for the data movement and processing it performs. In another scenario, a group sharing data may use a service provider, but is likely to be unwilling to pay for the processing that another group wants to perform on this data. As a specific example, the San Diego Supercomputing Center (SDSC) currently hosts more than 6 Petabytes of data, but most potential users of this data are only allowed to download, and not process this data at SDSC resources. The group using this data may have its own local resources, and may not be willing to pay for the processing at the same service provider, thus forcing the need for processing data away from where it is hosted.

When co-locating data and computation is not possible, remote data analysis offers many advantages over another feasible model, which could be referred to as *data staging*. Data staging implies that data is transferred, stored, and then analyzed. Remote data analysis requires fewer resources at the data analysis site, avoids caching of unnecessary or *process once* data, and may abstract away details of data movement from application developers and users.

We now give a brief overview of the design and implementation of the FREERIDE-G middleware. More details are available from our earlier publications [13, 7]. The FREERIDE-G middleware is modeled as a *client-server* system, where the *compute node* clients interact with both *data host* servers and a *code repository* server. The overall system architecture is presented in Figure 2.

A data host runs on every on-line data repository node in order to automate data retrieval and its delivery to the end-users' processing node(s). Because of its popularity, for this purpose we used Storage Resource Broker, a middleware that provides distributed clients with uniform access to diverse storage resources in a heterogeneous computing environment. The code repository is used to store the implementations of the FREERIDE-G-based applications, as specified through the API. A compute node client runs on every end-user processing node in order to initiate retrieval of data from a remote on-line repository, and perform application specific analysis of the data, as specified through the API implementation. The processing is based on the generic loop we described earlier, and uses application specific iterator and local and global reduction functions.

Figure 2 demonstrates the interaction of system components. Once data processing on the compute node has been initiated, data index information is retrieved by the client and a plan of data retrieval and analysis is created. In order to create this plan, a list of all data chunks is extracted from the index. From the work-list a schedule of remote read requests is generated to each data repository node. After the creation of the retrieval plan, the SRB-related information is used by the compute node to initiate a connection to the appropriate node of the data repository and to authenticate such connection. The connection is initiated through an SRB Master, which acts as a main connection daemon. To service each connection, an SRB Agent is forked to perform authentication and other services, with MCAT metadata catalog providing necessary information to the data server. Once the data repository connection has been authenticated, data re-

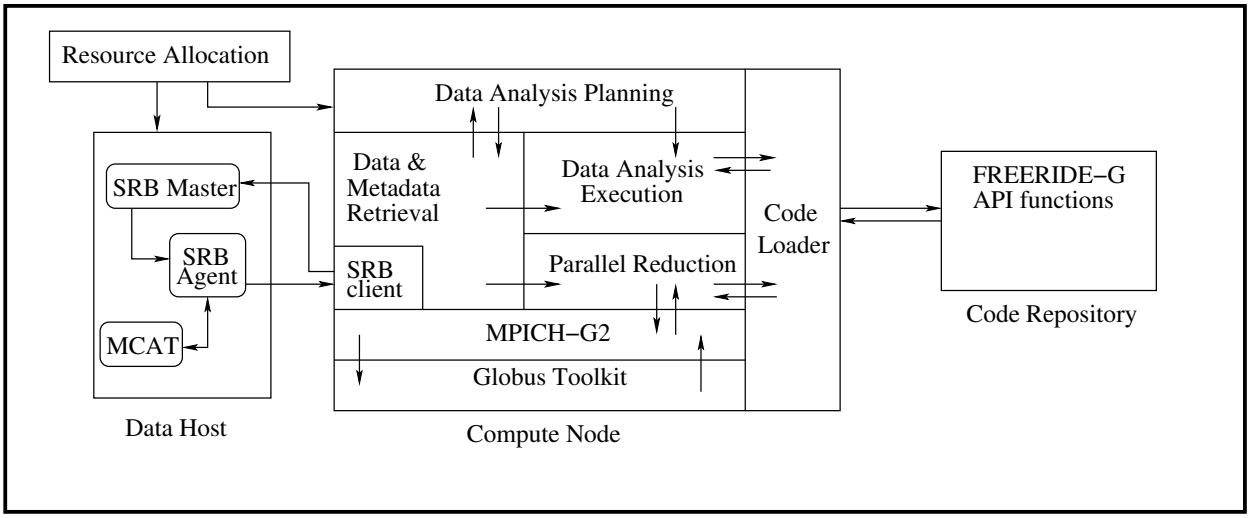


Figure 2: FREERIDE-G System Architecture

trieval through an appropriate SRB Agent can commence. To perform data analysis, the code loader is used to retrieve application specific API functions from the code repository and to apply them to the data.

3 Supporting Dynamic Load Balancing for Remote Data Analysis

In this section, we describe our dynamic load balancing approach and its implementation in the context of FREERIDE-G.

3.1 Our Approach: In the previous version of our middleware, the workflow of the application was set at the very beginning of the execution. Moreover, the jobs were evenly distributed among the compute nodes and each compute node was responsible for processing only its own jobs. If the compute nodes have different processing powers, the static job distribution may result in a large slowdown. Specifically, the compute nodes which have high throughput will have to wait until the slowest compute node finishes its execution.

Our approach for supporting dynamic load balancing exploits the properties of the processing structure of FREERIDE-G. Let us consider the processing structure supported by our middleware, shown earlier in Figure 1. Assume that the set of data elements to be processed is E. Furthermore, suppose a subset E_i of these elements is processed by the processor i, resulting in $RObj(E_i)$. Let G be the *global reduction function*, which combines the reduction objects from all nodes, and generates the final results.

The key observation in our approach is as follows. Consider any possible disjoint partition $E_1, E_2, \ldots, E_n$ of the processing elements between n nodes. The result of the

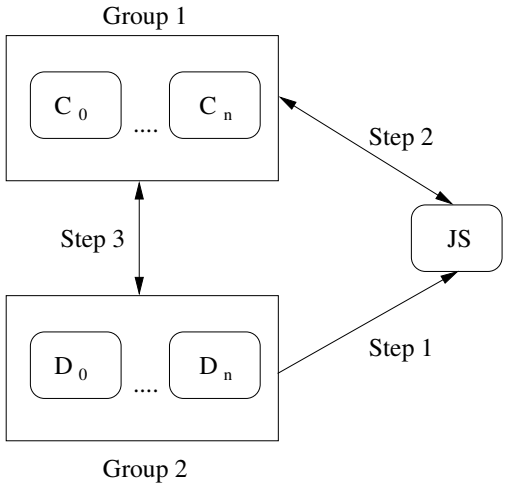


Figure 3: Load Balancing System's Workflow

global reduction function,

$$G(RObj(E_1), RObj(E_2), \ldots, RObj(E_n)) \tag{3.1}$$

will be same for any such disjoint partition of the element set E. In other words, if $E_1, E_2, \ldots, E_n$ and $E'_1, E'_2, \ldots, E'_n$ are two disjoint partitions of the element set E, then,

$$G(RObj(E_1), RObj(E_2), \ldots, RObj(E_n)) = G(RObj(E'_1), RObj(E'_2), \ldots, RObj(E'_n)) \tag{3.2}$$

If we examine the Equation (3.2), we can conclude that the processing structure that supports independent data elements can be exploited for dynamic load balancing system.

Specifically, any element, E_i, can be requested by any processing unit in the system during the execution. Therefore, the processing units which have high throughput can request and process more data elements than the others.

```
Input: dataNodes, List of data nodes that were
       registered to job scheduler
Result: job, which is assigned to compute node
/* Execute request handler loop   */
while true do
    compNode ← ReceiveReq();
    if CheckAssigned(compNode) then
        SetProcessed(compNode, dataNodes);
    end
    dataNode ← AvailDataNode(dataNodes);
    chunkNumb ← GetChunkNumb(compNode);
    job ← CreateJob(dataNode, chunkNumb);
    Transfer(job, compNode);
    if IsNotEmpty(job) then
        Assign(compNode,dataNode);
    end
end
```

Algorithm 1: Assigning jobs to Compute Nodes

```
Input: numbChunks, Number of chunks per job
 request
       : scheduler, Job Scheduler
Result: Final ReductionObject
/* Execute outer sequential loop  */
while true do
    /* Execute job request loop    */
    while true do
        job ← RequestJob(numbChunks, scheduler);
        if CheckJob(job) then
            break;
        end
        dataNode ← GetDataNode(job);
        chunksInfo ← GetChunksInfo(job);
        foreach chunk info cinfo in chunksInfo do
            {* Retrieve data chunk chk with cinfo
            from dataNode *};
            {* Process retrieved data chunk *};
            {* Update reduction object *};
        end
    end
    {* Perform Global Reduction *};
end
```

Algorithm 2: Processing Chunks on Compute Node

However, implementing such a dynamic scheme is also challenging. If all compute nodes request every chunk from a central scheduler, the overheads can be very high. Thus, while our approach is based on a central *job scheduler*, this scheduler works at a higher granularity. A compute node makes a request for a *set* of chunks to the job scheduler. The scheduler, then, returns a job, which includes the data node and the set of assigned chunk information. This chunk information consist of the exact offset addresses of the data elements in the data node. When the compute node receives the job, it starts retrieving the specified chunks from the data node.

A compute node again contacts the scheduler after the set of chunks have been retrieved from the data host and processed. This process is repeated until there is no more data to be processed. This scheduler is able to balance the workload between compute nodes with different processing power, while keeping the overheads very low.

3.2 Detailed Design and Implementation: We now discuss how our approach is implemented in FREERIDE-G. Figure 3 shows the interaction among the system elements. *Group 1* refers to the compute nodes, $C_{0 \cdots n}$, which are responsible for the processing of the data elements. Data nodes are represented with $D_{0 \cdots n}$ in *Group 2* where the data elements, i.e. chunks, are stored. It should also be noted that Group 1 and Group 2 are geographically separated. The *JS*, job scheduler, collects the necessary data information from Group 2 and then distributes the jobs to the compute nodes in Group 1.

Initially, the metadata information about the data needs to be prepared by data nodes in Group 2. The data in the system is stored in several files in which data is packed in data chunks (block). The metadata information about these chunks are stored into an index file. In this, each data chunk location is described with a data file name, offset address and the size of the data chunk. Each of these index information also corresponds to a metadata information of the smallest job in the system. Several of these index information can be combined and coarse-grained jobs can be generated.

After index information is generated, each data node in Group 2 prepares its specific node information which consists of the address information, available bandwidth of the data node and the chunk information of the data. It is then registered to the scheduler.

Job scheduler, on the other hand, waits for the data node registration requests. Whenever a registration request is received, the scheduler adds the data node information to the *dataNodes* list. This interaction is illustrated by *Step 1* in Figure 3.

After data nodes are registered and chunk information are specified in the scheduler, the job requests are handled. Algorithm 1 shows how the scheduler manages the compute nodes, which is also shown in Figure 3 with *Step 2*. At first,

the scheduler waits for the job requests from the compute nodes. When a job request is received, the scheduler checks if any of the chunks in the system was previously assigned to the requesting compute node. If so, the scheduler sets them as processed. Then, it creates another job with a new set of chunk information from the most suitable data node in the system. The main consideration for the scheduler in choosing a data node is effectively dividing the available bandwidth from each data node. Therefore, if the bandwidth between all pairs of compute nodes and data nodes is the same, the data node mapping will be done in a round robin fashion. If available bandwidths vary, more compute nodes' requests will be mapped to the data nodes with higher bandwidth, as long as they still have data that needs to be processed. Since the data and compute nodes are geographically separated, the bandwidth utilization and data node mapping are crucial for the overall execution time. After creating the job from a data node, it is transfered to the requesting compute node.

When a compute node receives a job from scheduler, it extracts the chunk information and starts requesting the chunks from the corresponding data node. With the retrieval of the data, the compute node starts executing the local reduction phase. When the data processing stage is finished, the compute node asks for another job. This continues until all the chunks are consumed. At last, all compute nodes finalize their execution with global combination. This process is shown in Algorithm 2.

4 Experimental Results

In this section, we report results from a number of experiments that evaluate our approach for supporting load balancing.

Two data-intensive applications we used are k-means clustering and Principal Component Analysis. Most of our experiments with k-means used a 25.6 GB dataset, whereas a 17 GB dataset was used for PCA. While our experiments with k-means involved only 1 iteration over the dataset, those for PCA had 3 iterations. As a result, the total amount of requested data is 51 GB for PCA application. All the datasets are divided into 4096 data blocks. Therefore, the sizes of each data block for k-means and PCA are 6.4 MB and 4 MB, respectively.

The configuration used for our experiments is as follows. Our compute nodes have dual processor Opteron 254 (single core) with 4GB of RAM and are connected through Mellanox Infiniband (1 Gb). We report experiments from the use of 4, 8, and 16 computing nodes. The number of data hosting nodes is always 4, and the data blocks are evenly distributed among these 4 nodes.

4.1 Effectiveness and Overheads: In this set of experiments, we evaluated the effectiveness and overheads of our

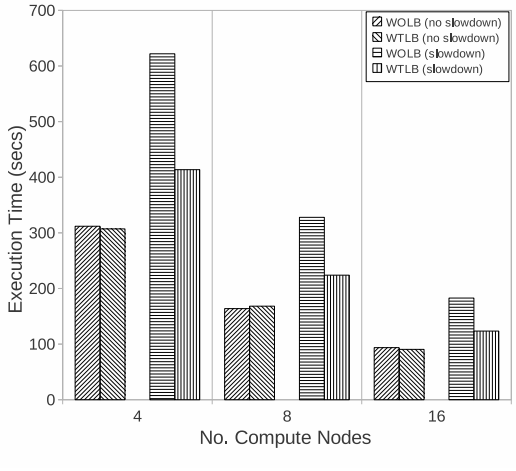


Figure 4: Evaluating Overheads using K-means clustering (25.6 GB dataset)

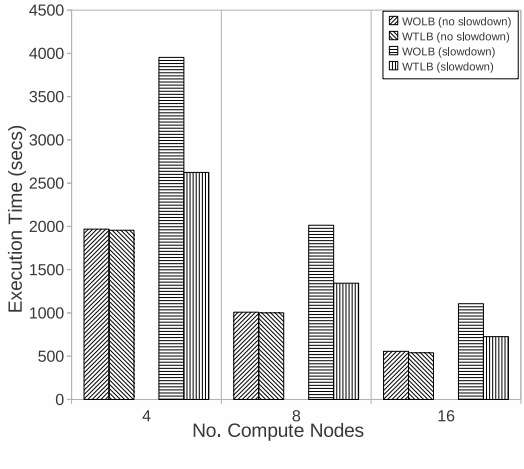


Figure 5: Evaluating Overheads using PCA (17 GB dataset)

dynamic load balancing system. For this experiment, we executed each of the two different versions of the middleware in two different environments. The two versions of the middleware are: Without load balancing system support, `WOLB`, and with load balancing system support, `WTLB`. `WOLB` is the first version of the FREERIDE-G, and is not able to balance the load among the compute nodes. Thus, the data elements are distributed evenly between the compute nodes and all the compute nodes have to wait until the slowest compute node finishes its processing. On the other hand, the enhanced version, `WTLB`, can dynamically balance the load between compute nodes.

The two environments in which we executed these two versions were the regular *homogeneous* cluster, and an environment with `slowdown`. Here, half of the compute nodes

are slowed down by 50% of their real processing power. The regular environment is also referred to as `no slowdown`.

In Figure 4, we present results from k-means application. The overheads of the load balancing system in `no slowdown` environments are 2.69% for 8 and close to 0% for 4 and 16 compute node cases. This shows that the implementation of our dynamic scheme is very efficient, and does not cause noticeable overheads.

In the `slowdown` environment, the speedups of the system range from 1.46 to 1.50 over the static partitioning system.

We can further analyze the costs of our load balancing implementation. The *absolute overhead* of the system can be calculated with the expected execution time of FREERIDE-G with `WOLB (slowdown)` configuration , say $time_{exp}$, which has the perfect data distribution among its compute nodes; and the execution time of `WTLB (slowdown)` configuration , say $time_{wtlb}$. The perfect data distribution for `WOLB (slowdown)`, in this case, means the data distribution among the compute nodes that satisfy the same execution time for every compute node. For instance, if half of the compute nodes in the system are limited to use 50% percentage of their CPU power and can process 450 data chunks during the execution, then the compute nodes which have 100% percentage CPU utilization are expected to process 900 data chunks in the same time period. With such configuration, the data chunks are perfectly distributed and the execution time of the system, $time_{exp}$, is optimum. Consequently, the absolute overhead can be found with:

$$overhead_{absolute} = \frac{time_{wtlb} - time_{exp}}{time_{exp}} \tag{4.3}$$

If we apply (4.3) to Figure 4, then the absolute overheads of our system are again close to 0% for the three compute node cases. The retrieval and the processing time of the assigned data dominate the communication time between the scheduler and the compute nodes. Moreover, the scheduler can successfully select the appropriate data node in which compute node can benefit from the available bandwidth and maximize its data transfer speed.

In Figure 5, the same combination of versions and environments are repeated with PCA as the application. The overheads of the `no slowdown` version and the absolute overheads are close to 0%. The speedups of our system with `slowdown` version change from 1.49 to 1.52, considering without load balancing system with `slowdown` version.

4.2 Overheads With Different Slowdown Ratios: The experiments that we reported in previous section, half of the compute nodes were 50% slowed down. In this subsection, we evaluate our system's performance with two additional slowdown ratios: 25% and 75%. These slowdowns are applied to half of the compute nodes, again.

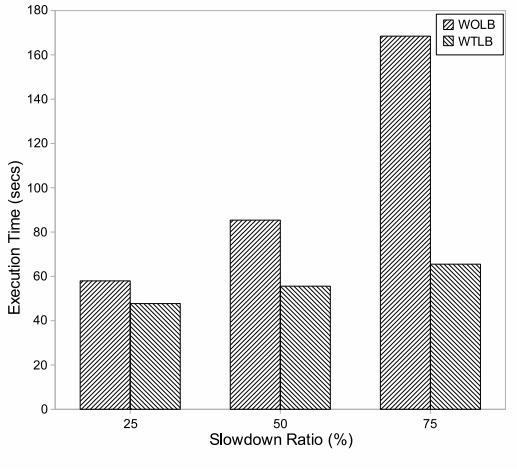


Figure 6: K-means clustering with Different Slowdown Ratios (6.4 GB dataset, 8 comp. nodes)

In Figure 6, we evaluated the k-means clustering application with 8 compute nodes and a 6.4 GB dataset. The absolute overheads, in each case, are close to 0%. The speedups of our system with respect to `WOLB` are 1.21, 1.53, and 2.57 for 25%, 50% and 75% slowdowns, respectively. The higher slowdown ratios indicate longer execution times for slow processing units. Furthermore, the system should wait for the slowest processing unit in case of static job assignment, i.e. `WOLB` configuration. On the other hand, the faster compute nodes can consume slow compute nodes' data elements with `WTLB` configuration which results in high speedups.

Same experiment was repeated with PCA and the results are shown in Figure 7. The absolute overheads are 3.50%, 1.63% and 4.07% for the three cases. Furthermore, the speedups over the static case are 1.23, 1.49 and 2.42 for 25%, 50% and 75% slowdown ratios, respectively. As we mentioned before, the PCA has three iterations, and requests more data elements (3 times) than k-means application. Moreover, the volume of retrieved data is significantly larger than the other configurations. Thus, the overheads become more visible.

4.3 Distribution of Data Elements with Varying Slowdown Ratios: In this section, we focused on how successfully our system distributes data elements among the compute nodes. In previous sections, the slowdown ratios were kept same for all the applied compute nodes. However, in this set of experiments, we varied the slowdown ratios between 8 compute nodes. More specifically, the slowdown ratios are increased by 12.5% for each of the compute node starting from 0%.

We showed the number of processed data elements for each of the compute node in Figure 8. The last bar in

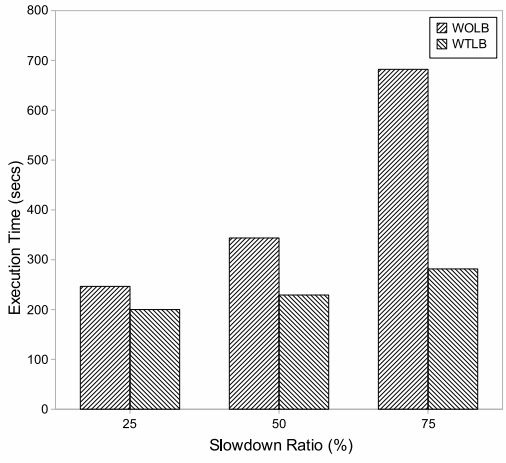


Figure 7: PCA with Different Slowdown Ratios (4 GB dataset, 8 comp. nodes)

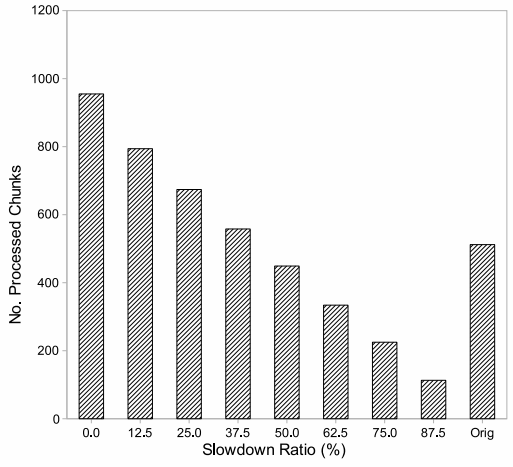


Figure 8: K-means clustering with Different Slowdown Distribution (6.4 GB dataset, 8 comp. nodes)

the figure shows the expected number of data elements that need to be processed in case of perfect CPU utilization for all compute nodes. The number of processed chunks ranges from 109 to 161 for consecutively slowed down processing units. The absolute overhead of our system is again very close to 0%. These results show that our system can successfully distribute the chunks even in a highly heterogeneous environment.

In Figure 9, we repeated our experiment with PCA application. Note that the total number of chunks is three times more than the k-means clustering application due to the iterations. The number of processed chunks ranges from 233 to 477 for consecutively slowed down compute nodes. The absolute overhead in this case is 8.5%. The basic reasons of this overhead are the high imbalance in the CPU utilization among the processing units, and the number of iterations which results in more job requests to the scheduler.

4.4 Overheads with Different Assignment Granularity:

In all experiments reported in previous subsections, the scheduler was set to assign 4 data chunks for every job request. In this subsection, the number of assigned data chunks per request is varied. K-means clustering application with 6.4 GB dataset was used for the evaluation; and the assigned data chunks per request were changed to 4, 16, 64 and 256, respectively.

The results are shown in Figure 10. The absolute overheads are 0.72%, 1.58%, 6.31% and 16.01% for 4, 16, 64, and 256 data chunks cases, respectively. The load balancing system's overhead increases with the increasing number of data chunks per request, and a fine-grained assignment results in the best performance for our system. This is because the overheads of dynamic load balancing are still very low with fine-grained assignments. At the same time, the performance with coarse-grained assignments is worse, because we do not achieve perfect load balance.

5 Related Work

The topics of data-intensive computing and map-reduce have received much attention within the last 2-3 years. Projects in both academia and industry are working towards improving map-reduce. CGL-MapReduce [14] uses streaming for all the communications, and thus improves the performance to some extent. Mars [15] is the first attempt to harness GPU's power for map-reduce.

Yahoo's map-reduce system, Hadoop, is one of the popular implementations. Even though, our system and Hadoop share important similarities, the differences are significant. Hadoop assigns tasks to the racks where data locality is maximized and high throughput is satisfied. Our system, on the other hand, works in the context of remote data analysis in which exploiting such locality is not possible. However, our system minimizes the data retrieval time through exploiting bandwidth usage of the data nodes which results in optimum computation throughput.

Lin et al. extended Hadoop with MOON [16] which provides better performance in unreliable volunteer computing systems using a small set of dedicated nodes. Our dynamic load balancing system extends our previous work which focuses on fault tolerance [17] in data-intensive computing environments. We believe our system can also perform well in such unreliable environments.

Farivar *et al.* introduced an architecture named MITHRA [18] and integrated the Hadoop map-reduce with the power of GPGPUs in the heterogeneous environments. Zaharia *et al.* [19] improved Hadoop response times by de-

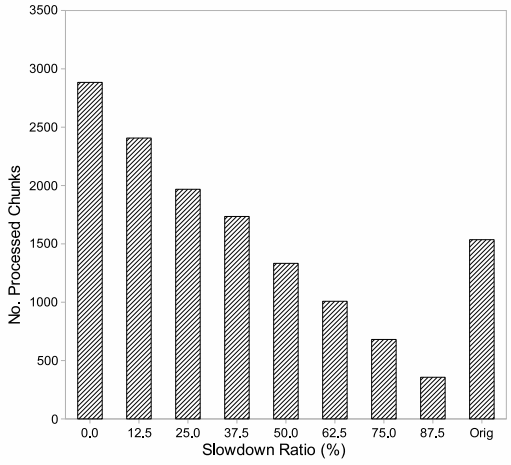


Figure 9: PCA with Different Slowdown Distribution (4 GB dataset, 8 comp. nodes)

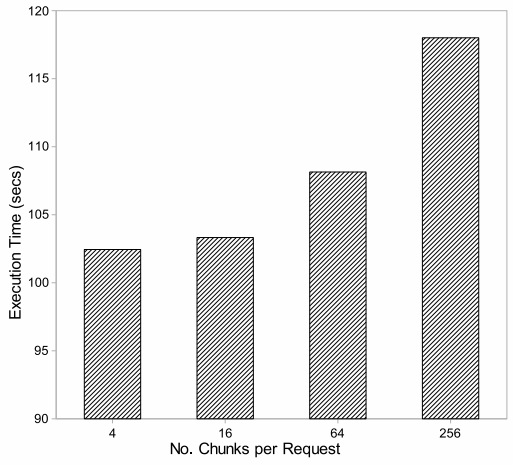


Figure 10: Performance with Different Number of Chunks per Request (k-means, 6.4 GB dataset, 4 comp. nodes)

signing a new scheduling algorithm in a virtualized data center. Seo *et al.* [20] proposed two optimization schemes, prefetching and pre-shuffling, to improve Hadoop's overall performance in a shared map-reduce environment. Ranger *et al.* [21] have implemented Phoenix, a map-reduce system for multi-cores.

Facebook uses *Hive* [22] as the warehousing solution to support data summarization and ad-hoc querying on top of Hadoop. Yahoo has developed *Pig Latin* [23] and Map-Reduce-Merge [24], both of which are extensions to Hadoop, with the goal being to support more high-level primitives and improve the performance. Google has developed *Sawzall* [25] on top of map-reduce to provide higher-level API. Microsoft has built *Dryad* [26], which is more flexible than map-reduce, since it allows execution of computations that can be expressed as *DAGs*.

OpenMP is an API which supports parallel shared memory processing on many architectures and supports different scheduling strategies such as static, dynamic and guided work sharing. However, the programmer should involve solving the data dependencies and synchronization issues. These are automatically handled by our framework.

6 Conclusions

In this work, we developed and evaluated a dynamic load balancing scheme for a data-intensive computing middleware. We focused on heterogeneous environments such as non-dedicated machines in grids and virtualized machines in clouds. We proposed an approach which effectively solves the problem of task distribution among processing units that show different processing performance.

Two data-intensive applications were used in order to evaluate the system. Our results show that the overheads of our system are very small. Moreover, our approach can successfully distribute tasks among processing units even in highly heterogeneous configurations.

References

[1] R. E. Bryant, "Data-intensive supercomputing: The case for disc," School of Computer Science, Carnegie Mellon University, Tech. Rep. Technical Report CMU-CS-07-128, 2007.

[2] T. Scholl, A. Reiser, and A. Kemper, "Collaborative query coordination in community-driven data grids," in *Proceedings of the Conference on High Performance Distributed Computing (HPDC)*, Jun. 2009.

[3] J. Heo, X. Zhu, P. Padala, and Z. Wang, "Memory overbooking and dynamic control of xen virtual machines in consolidated environments," in *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (IM09)*, June 2009, pp. 630–637.

[4] H. Lim, S. Babu, J. Chase, and S. Parekh, "Automated control in cloud computing: Challenges and opportunities," in *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds (ACDC09)*, June 2009, pp. 13–18.

[5] P.Barham, B.Dragovic, K.Fraser, S.Hand, T.Harris, A.Ho, R.Neugebauer, I.Pratt, and A.Warfield, "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP03)*, 2003, pp. 64–177.

[6] L. Glimcher and G. Agrawal, "A Performance Prediction Framework for Grid-based Data Mining Applications," in *In proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[7] ——, "A Middleware for Developing and Deploying Scalable Remote Mining Services," in *In proceedings of Conference on Clustering Computing and Grids (CCGRID)*, 2008.

[8] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of OSDI*, 2004, pp. 137–150.

[9] R. Jin and G. Agrawal, "A middleware for developing parallel data mining implementations," in *Proceedings of the first SIAM conference on Data Mining*, Apr. 2001.

[10] ——, "Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance," in *Proceedings of the second SIAM conference on Data Mining*, Apr. 2002.

[11] W. Jiang, V. T. Ravi, and G. Agrawal, "Comparing map-reduce and freeride for data-intensive applications," in *Proceedings of the 2009 IEEE Cluster*. IEEE, 2009.

[12] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of Grid: Enabling Scalable Virtual Organizations," *International Journal of Supercomputing Applications*, 2001.

[13] L. Glimcher, R. Jin, and G. Agrawal, "FREERIDE-G: Supporting Applications that Mine Data Repositories," in *In proceedings of International Conference on Parallel Processing (ICPP)*, 2006.

[14] J. Ekanayake, S. Pallickara, and G. Fox, "Mapreduce for data intensive scientific analyses," in *IEEE Fourth International Conference on e-Science*, Dec 2008, pp. 277–284.

[15] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *Proceedings of PACT 2008*. ACM, 2008, pp. 260–269.

[16] H. Lin, J. Archuleta, X. Ma, W. Feng, Z. Zhang, and M. Gardner, "Moon: Mapreduce on opportunistic environments," in *ACM International Symposium on High Performance Distributed Computing (HPDC)*, June 2010.

[17] T. Bicer, W. Jiang, and G. Agrawal, "Supporting fault tolerance in a data-intensince computing middleware," in *Proceedings of the 24th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2010.

[18] R. Farivar, A. Verma, E. Chan, and R. Campbell, "Mithra: Multiple data independent tasks on a heterogeneous resource architecture," in *Proceedings of the 2009 IEEE Cluster*. IEEE, 2009.

[19] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of OSDI*. USENIX Association, 2008, pp. 29–42.

[20] S. Seo, I. Jang, K. Woo, I. Kim, J.-S. Kim, and S. Maeng, "Hpmr: Prefetching and pre-shuffling in shared mapreduce computation environment," in *Proceedings of the 2009 IEEE Cluster*. IEEE, 2009.

[21] C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *Proceedings of 13th HPCA*. IEEE Computer Society, 2007, pp. 13–24.

[22] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive - a warehousing solution over a map-reduce framework," *PVLDB*, vol. 2, no. 2, pp. 1626–1629, 2009.

[23] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *Proceedings of SIGMOD Conference*. ACM, 2008, pp. 1099–1110.

[24] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. P. Jr., "Map-reduce-merge: simplified relational data processing on large clusters," in *Proceedings of SIGMOD Conference*. ACM, 2007, pp. 1029–1040.

[25] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with sawzall," *Scientific Programming*, vol. 13, no. 4, pp. 277–298, 2005.

[26] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2007 EuroSys Conference*. ACM, 2007, pp. 59–72.