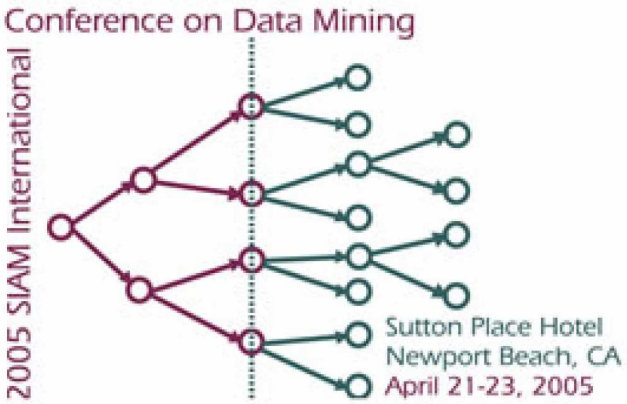

HPDM'05

in conjunction with the 5th International
SIAM Conference on Data Mining (SDM'05)

Newport Beach, CA
April 23, 2005

8th International Workshop on

High Performance Distributed Mining

Preface

Workshop description

Over the years the definition of high performance computing has taken on various forms as a function of the types of technical and creative uses and the underlying semantics of the applications driving them. Traditional definitions often refer to the problem of using high end parallel computers to meet the need of scientific applications. However, high performance computing can also include the need for fast sequential algorithms that target memory and I/O performance. The last decade has seen the growth and importance of Grid computing where resources and data are physically distributed. This has led to the development of high performance distributed algorithms over the Computational Grid, where privacy, security, and resource discovery are all important issues. This year the workshop welcomes papers on all aspects of high performance data mining. Topics of interest include (but are not limited to):

- Grid-based data mining algorithms and systems.
- Distributed techniques for incremental, exploratory and interactive mining.
- Distributed techniques for security, privacy preserving data mining
- Peer-to-Peer Data Mining.
- High performance data stream mining and management.
- Resource and location-aware mining algorithms.
- Data mining in mobile environments.
- Theoretical foundations for resource-aware mining in a mobile, streaming and/or distributed environment.
- Systems support for resource and location aware data mining.
- Efficient, scalable, disk-based, parallel and distributed algorithms for large-scale data mining and pre-processing and post-processing tasks.
- Parallel or distributed frameworks for stream management, KDD systems, and parallel or distributed mining.
- Applications of parallel and distributed datamining (PDDM) in business, science, engineering, medicine, and other disciplines.

Workshop History

This is the 8th workshop on this theme held annually. Traditionally, the workshop has been held along-side the SIAM Data Mining (SDM) Conference, even if the first four editions were organized in conjunction with IPDPS, and were held at Orlando (HPDM'98), San Juan (HPDM'99), Cancun (HPDM'00) and San Francisco (HPDM'01).

Over the last three years the workshop has had invited papers in the areas of mobile and location-aware data mining issues (HPDM:RLM'02), pervasive and stream data mining (HPDM:PDS'03), and Grid data mining (HPDM:GRID'04).

Workshop Program

HPDM'05 is the 8th Annual Workshop on this theme. It will be held on April 23, 2005, in conjunction with the 5th International SIAM Conference on Data Mining (SDM'05). The SIAM Conference and its satellite Workshops will be held at the Sutton Place Hotel in Newport Beach, CA. The Sutton Place hotel is located in picturesque Newport Beach known as the colorful coast and conveniently situated in the heart of the Orange County business districts of Irvine and Tustin.

HPDM'05 will be a half-day Workshop, with an invited talk, and four oral presentations, carefully selected by the Program Committee. We are indeed privileged to have Dr. Chris Ding, Lawrence Berkeley National Laboratory, as an invited speaker at the Workshop. The accepted papers cover a wide range of topics. S. Datta, C. Giannella and H. Kargupta present a paper entitled "K-Means Clustering over Peer-to-peer Networks", which discusses an algorithm for K-means clustering of homogeneously distributed data in a peer-to-peer network. H. D. K. Moonesinghe, P. N. Tan, M. J. Chung and M. Wu discuss the paper "Parallel Mining of Sequential Patterns Using an Efficient Task Partitioning Approach", addressing the parallelization of the PrefixSpan algorithm, by taking into account some load imbalance issues. C. Lucchese, S. Orlando and R. Perego presents the paper entitled "Distributed Mining of Frequent Closed Itemsets: Some Preliminary Results", which shows how the frequent closed itemsets, independently mined from disjoint and distributed data partitions, can be merged to obtain the global result. Finally, R. Pedersen presents the paper "Energy Measurements for a Support Vector Machine Classifier on Micro Java and Tinyos", which addresses the problem of energy consumption in a sensor network, by considering a distributed SVM-based classification is a specific example.

We thank all of the Program Committee members, whose work helped to select an interesting program.

March 2005

S. Orlando
K. Sivakumar

Committees

Workshop Co-Chairs

Salvatore Orlando, Università Ca' Foscari di Venezia

K. Sivakumar, Washington State University

Program Committee

Gagan Agrawal, Ohio State University

Rong Chen, University of Pennsylvania

Chris Giannella, University of Maryland, Baltimore County

Sara Graves, The University of Alabama at Huntsville

Matthias Klusch, German Research Center for Art. Intell.

Shonali Krishnaswamy, Monash University

Michael May, Fraunhofer Inst. for Autonomous Intell. Syst.

Hoony Park, Oak Ridge National Laboratory

Raffaele Perego, Consiglio Nazionale delle Ricerche

Assaf Schuster, Technion

Parthasarathy Srinivasan, Ohio State University

Ashok Srivastava, NASA Ames Research Center

Vaidy Sunderam, Emory Universiyt

Domenico Talia, Università della Calabria

Ran Wolff, Technion

Mohammed Zaki, Rensselaer Polytechnic Institute

Steering Committee

Hillol Kargupta, University of Maryland, Baltimore County

Vipin Kumar, University of Minnesota

Srinivasan Parthasarathy, Ohio State University

David Skillicorn, Queens University

Mohammed Zaki, Rensselaer Polytechnic Institute

Table of Contents

K-Means Clustering over Peer-to-peer Networks
S. Datta, C. Giannella, and H. Kargupta

Parallel Mining of Sequential Patterns Using an Efficient Task Partitioning Approach
H. D. K. Moonesinghe, P. N. Tan, M. J. Chung, and M. Wu

Distributed Mining of Frequent Closed Itemsets: Some Preliminary Results
C. Lucchese, S. Orlando, and R. Perego

Energy Measurements for a Support Vector Machine Classifier on Micro Java and Tinyos
R. Pedersen

Invited Speaker

Dr. Chris Ding, Lawrence Berkeley National Laboratory

Chris Ding is a staff computer scientist at Lawrence Berkeley National Laboratory since 1996. From 1993 to 1996 he worked at NASA's Jet Propulsion Laboratory on data assimilation (SIAM News, front page, Oct 1996) and parallel computing algorithms. From 1987 to 1993 he worked at California Institute of Technology, on Caltech hypercubes with applications ranging from Materials Science (Nature, v.344, p.485, 1990) to Computational Biology ("Mathematical Challenges from Theoretical/Computational Chemistry," National Research Council report, 1995, Chapter 3, Numerical Analysis section). In 1987 he earned a Ph.D. in Theoretical Physics and Computer Science from Columbia University on building a parallel processor using Intel 80286s and commodity FPUs (Science, cover page, Aug 1988) and doing large scale QCD simulations on it. His recent research include algorithmic R&D for climate models, bioinformatics, and data mining. He received a Best Paper Award for the data assimilation work, a NASA Group Achievement Award at JPL and several Outstanding Performance Awards at LBNL. On the lighter side. Fascinated by information retrieval from the Web, he and collaborators proved via closed-form solutions that the ranking of webpages from PageRank (Google) and HITS (IBM) are equivalent to ranking by indegree (# of inbound hyperlinks) assuming the web is a fixed-degree-sequence random graph.

K-Means Clustering over Peer-to-peer Networks

Souptik Datta* Chris Giannella † Hillol Kargupta ‡

Abstract

This paper presents preliminary work on an algorithm for K-means clustering of homogeneously distributed data in a peer-to-peer network. The algorithm is asynchronous and each node operates locally by communicating only with its topologically neighboring nodes. Importantly, large scale synchronization is not required. Empirical results show, in many cases, the final centroids produced are very close to the final centroids produced by standard K-means run on centralized data. Consequently, the number of incorrectly labeled data points is small.

Keywords: Peer-to-peer (P2P), K-means clustering.

1 Introduction

K-means clustering [5] is a well-known and well-studied exploratory data analysis technique. The standard version assumes that all data is available at a single location. However, important applications exist for which data sources are distributed over a large, loosely-connected network. Collecting the data at a central location before clustering is not an attractive option. Moreover, distributed algorithms which require global synchronization are also not desired.

Example: Consider a sensor network consisting of a large number of light-weight, wireless, battery-powered sensors for environment monitoring. Each sensor is measuring the same variables and clustering all the data in the network in a given time window can offer valuable information concerning environmental phenomena. Since the power required for wireless communication goes up with the square of distance, nodes only should communicate with others in a small radius (immediate neighbors). Moreover, the large number of nodes makes global synchronization undesirable. Thus, a K-means clustering algorithm in this example ought to work in a locally synchronous manner *i.e.* nodes only synchronize with their immediate neighbors.

This paper addresses the problem of K-means clustering of data distributed homogeneously over a large, loosely connected peer-to-peer (P2P) network. We present preliminary work toward the develop of an algorithm, *P2P K-Means*, for which nodes only require local synchronization. To our knowledge, P2P K-means represents the first K-means clustering algorithm to operate in this fashion.

Section 2 describes related work: P2P content sharing, distributed K-means algorithms, data mining in P2P networks. Section 3 describes the P2P K-means algorithm. Section 4 describes the experimental setup: simulator and dataset. Section 5 describes the experiments conducted and their results. Section 6 discuses several outstanding issues to be addressed in the future work. Finally, Section 7 concludes the paper.

2 Related Work

Content sharing in P2P networks has received a great deal of attention.[1] However, distributed data mining in P2P networks has received very little. Wolff and Schuster [8] develop an algorithm for association rule mining in a large-scale, dynamic P2P network. Kowalczyk *et al.* [7] develop a framework (*newscast model*) for computing basic operations like average, maximum, minimum in a large-scale, dynamic P2P network. Eisenhardt *et al.* [3] develop a K-means clustering algorithm for a P2P network using a *probe-and-echo* mechanism. Their algorithm is similar to ours in that we both only transmit centroids rather than data to reduce communication load. However, their algorithm requires a complete global synchronization at each iteration, while ours does not require global synchronization at any time. On the other hand, their algorithm is guaranteed to produce exact results (*i.e.* same as K-means on centralized data), while ours produces an approximation. Finally, Bandyopadhyay *et al.* [1] develop a non-locally synchronized K-means algorithm in a P2P network but use random sampling of the network to reduce message load. Their algorithm is similar to ours in that an approximate result is produced. On the other hand, we do only require local synchronization and only require nodes to communicate with their immediate neighbors.

Distributed clustering has been addressed as of recent in the Distributed Data Mining (DDM) community. For a survey of clustering techniques in DDM see [6] and [1]. In particular, some work has been done on parallel implementa-

*Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, USA.

†Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, USA.

‡Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, USA. Also affiliated with AGNIK LLC, Columbia, MD USA.

[1] See Androutsellis-Theotokis, http://citeseer.ist.psu.edu/androutsellis-theoto02survey.html for an overview.

tions of K-means. Dhillon and Modha [2] divide the dataset into p same-sized blocks, then, on each iteration, each of the p processors updates its current centroids based on its block. The processors broadcast their centroids and cluster counts. Once a processor has received all the centroids from other processors it forms the global centroids by weighted averaging. Each processor proceeds on to the next iteration. Forman and Zhang [4] take a similar approach, but extend it to K-harmonic means. Note that the methods of [2] and [4] both start by partitioning then distributing a centralized dataset over many sites. This is different than the setting we consider: the data is never centralized, it is inherently distributed. However, we directly employ their idea of sending around centroids and updating based on weighted averaging.

3 The Algorithm

Below we describe our P2P K-means algorithm. The goal is for each node to converge on a set of centroids that are as close as possible to the centroids that would be produced by first centralizing the data, then running K-means.

3.1 Basic Algorithm Let $N_1, N_2, \ldots, N_n$ denote the nodes in the system each with data set X^i. The global dataset is denoted as X which equals $\bigcup_{i=1}^{n} X^i$. Let $Neigh^{(i)}$ denote the set of nodes N_i is directly connected to at a given time *i.e.* the immediate neighbors of N_i. We assume that each node can reliably compute $Neigh$ at any given time. As a consequence, for example, each node can determine if the link to any of its immediate neighbors from a previous time has gone down.

At the beginning of iteration ℓ, node N_i has a set of centroids, $\{v_{j,\ell}^{(i)}{:}1 \leq j \leq K\}$. Node N_i first carries out one round of K-means on its local data X^i using $\{v_{j,\ell}^{(i)} : 1 \leq j \leq K\}$. The result is a new set of centroids and their associated cluster counts $\{(w_{j,\ell}^{(i)}{,}|w_{j,\ell}^{(i)}|) : 1 \leq j \leq K\}$ where $|w_{j,\ell}^{(i)}|$ is defined as the number of tuples in X_i for which $w_{j,\ell}^{(i)}$ is closer[2] than any other $w_{h,\ell}^{(i)}$, $h \neq j$. The collection $\{(w_{j,\ell}^{(i)}{,}|w_{j,\ell}^{(i)}|) : 1 \leq j \leq K\}$ is stored in the *history table* whose purpose will be explained in Section 3.2.

Node N_i sends a poll message, $\langle i, \ell \rangle$, to each node $N_k \in Neigh^{(i)}$ (the purpose of the iteration number will be explained in Section 3.2). Once all N_k have responded or cease to be neighbors, N_i continues. Let $Resp^{(i)}$ denote the set of nodes that did respond. Each response message from node $N_k \in Resp^{(i)}$ contains the locally updated centroids and cluster counts at node N_k during iteration ℓ, *i.e.* the response is $\langle k, \{(w_{j,\ell}^{(k)}{,}|w_{j,\ell}^{(k)}|){:}1 \leq j \leq K\}\rangle$. Node N_i updates its jth centroid as follows (producing the jth centroid at the beginning of iteration $\ell + 1$):

$$v_{j,\ell+1}^{(i)} = \frac{\sum_{N_k \in (Resp^{(i)} \cup \{N_i\})} w_{j,\ell}^{(k)} |w_{j,\ell}^{(k)}|}{\sum_{N_k \in (Resp^{(i)} \cup \{N_i\})} |w_{j,\ell}^{(k)}|}.$$

If $max\{||v_{j,\ell}^{(i)} - v_{j,\ell+1}^{(i)}|| : 1 \leq j \leq K\} > \gamma$ (a user-defined parameter), then node N_i goes on to iteration $\ell + 1$. Otherwise it enters the terminated state (described in Section 3.2).

3.2 Some Details *Initialization:* The initial centroids $\{v_{j,1}^{(i)} : 1 \leq j \leq K\}$ are chosen randomly, but are the same for each node. The node which initiates the algorithm chooses the initial centroids, then propagates them to its neighbors. Once a node receives the initial centroids, it propagates them to its neighbors, then begins iteration one.

Poll response: Suppose node N_i receives polling message $\langle h, \hat{\ell} \rangle$ during its iteration ℓ. The message came from node N_h, during its iteration $\hat{\ell}$. N_i must determine how to respond. If $\hat{\ell} < \ell$, then N_i's history table contains its local centroids and their cluster counts from iteration $\hat{\ell}$. Hence, N_i, sends these immediately in a response message to N_h. If $\hat{\ell} > \ell$, then N_i's history table does not contain local centroids for iteration $\hat{\ell}$. So, poll message $\langle h, \hat{\ell} \rangle$ is placed in the *poll table*. If $\hat{\ell} = \ell$, then N_i checks if its history table contains local centroids and their cluster counts for iteration $\hat{\ell}$. If so, these are sent to N_h. If not, $\langle h, \hat{\ell} \rangle$ is placed in the poll table.

Finally, N_i must also check its poll table during iteration ℓ. This is done immediately after producing the local centroids and their cluster counts. For all poll messages $\langle h_1, \ell \rangle, \cdots, \langle h_m, \ell, \rangle$ in the table, N_i sends its local centroids and their cluster counts in a response message to each h_j. These poll messages are then removed from the table.

Termination: As described in Section 3.1, a node, N_i, can enter a terminated state, say at the end of iteration ℓ. Once this happens N_i no longer updates its centroids or sends polling messages. However, it does respond to polling messages $\langle h, \hat{\ell} \rangle$ as follows. If $\hat{\ell} \leq \ell$, then N_i looks up its local centroids and their cluster counts for iteration $\hat{\ell}$ in its history table. They are immediately sent in a response message to N_h. If $\hat{\ell} > \ell$, then N_i looks up its local centroids and their counts for iteration ℓ and sends these in a response message to N_h.

Note that, no node has an explicit condition under which all activity stops. However, once a node enters the terminated state, it no longer sends polling messages, only responses. Therefore, once all nodes enter into the terminated state, all communication ceases *i.e.* the algorithm has terminated.

3.3 Analysis Consider P2P K-means at some fixed moment in time. Let I denote the maximum number of iterations carried out by any node and let L denote

[2] In our experiments we used Euclidean distance.

$max\{Neigh^{(i)} : 1 \leq i \leq n\}$. Now we provide worst-case space and communication analysis of P2P K-means with respect to I, L, K, and n.

At any given node N_i, the space required is proportional to the size of N_i's history and poll tables. Clearly the history table is of size $O(IK)$ since the local centroids and their cluster counts are added for each iteration. The poll table is of size $O(IL)$, since each of N_i's neighbors sends one poll messages per iteration, thus, a maximum of I per neighbor. Therefore, the total space is $O(I(K+L))$. The number of messages (4 byte numbers) transmitted by N_i is $O(ILK)$. This is because N_i sends a poll message (size $O(1)$) to each neighbor at each iteration ($O(IL)$ in total). On top of this, N_i sends a response of size $O(K)$ for each entry of the $O(IL)$ entries in its poll table ($O(ILK)$ in total). Therefore, total number of messages is $O(IL + ILK)$.

Hence the total amount of space and communication over all nodes is $O(nI(K+L))$ and $O(nILK)$, respectively.

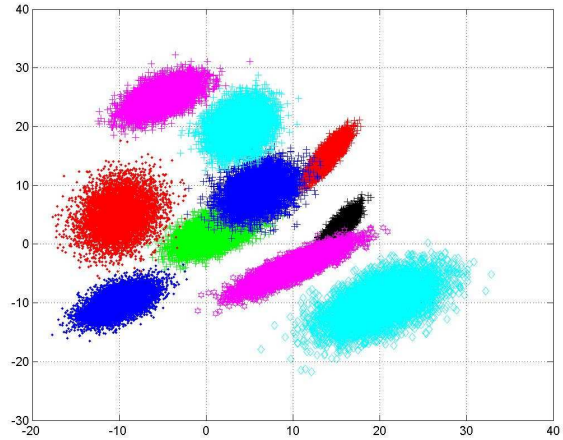

Figure 1: Experimental dataset

4 Experimental Setup

We tested our algorithm in a simulated environment run on a single machine. First we describe the simulator (Section 4.1), then the dataset used (Section 4.2).

4.1 Simulator Our simulation consists of two parts: the network topology with edge delays; the message passing and local computation behavior. For the network topology and delays, we have used one of the standard Internet topology generators, BRITE[3]. It produces a weighted graph with edge weights representing communication delays (in mSeconds). We have used flat level 'Autonomous System' (AS) with Waxman model to simulate the network in BRITE, where two nodes u and v are connected with a probability $P(u,v) = \alpha e^{-d(u,v)/\beta L}$, where α =0.15 and β = 0.2. An 'incremental growth' version of the Waxman model with random node placement is used during topology construction. A new node surveys the existing nodes in the graph in each step and connects to m ($= 2$) of them with the said probability. Other parameters[4] used are HS $= 1000$, LS $=$ 100 (size of the plane), constant bandwidth distribution with Max BW $= 1024$, and Min BW $= 10$. In our experiments, we do not consider the effect of edge or node failures or edge or node additions *i.e.* the topology remains fixed throughout each experiment.

For the message passing and local computation behavior, our simulator operates with respect to a global clock. At each clock tick, the state of the network is updated. For each message in flight, its number of ticks to arrival is decremented. If the message has arrived, it is copied into a buffer associated with the arrival node. We assume that network communication dominates local computation (a reasonable assumption if the local datasets are small). Thus, local computation time is not considered in the simulation. In one clock tick, each node is able to carry out one round of K-means producing local centroids and cluster counts ($\{(w, |w|)\}$) and send polling requests to all neighbors. The remainder of the iteration will require more clock ticks until all responses have arrived. Once all responses have arrived, the node updates its old centroids without any additional ticks and, moreover, processing poll requests from other nodes does not incur any additional clock ticks.

Since the simulator operates on global clock ticks rather than wall clock time, the BRITE map edge delays must be converted from mSeconds to number of clock ticks. To do this, we assume a clock tick in 500 mSeconds and round edge delays to the nearest multiple of 500 mSeconds.

4.2 Dataset We conducted all the experiments with a 2D synthetic data set generated from ten multivariate Gaussian distribution, Figure 1. The dataset has 78200 points and significant overlap between the clusters. The data is distributed over different nodes in the network by uniform random sampling. We assume the data remains static throughout each experiment.

5 Results

We varied the number of nodes from 50 to 500 and measured communication complexity of P2P K-means as well as its accuracy with respect to the centralized K-means. In all experiments, γ, the termination parameter, is set to one. We also start the simulations with each node having the initial centroids (same for each node) thereby ignoring the initial propagation delay. However, since we assume the network

[3] www.cs.bu.edu/brite

[4] Refer to BRITE documentation for details.

topology remains fixed in each simulation, the initial propagation of centroids from a leader node to all other nodes merely increases the number of global clock ticks for the simulation to complete. It does not affect the accuracy or the number of algorithm iterations to completion. Average immediate neighborhood length is 4 per node. The number of clusters (K) in all simulations is set to 8. Note that the dataset is made up of 10 Gaussian distributions *i.e.* 10 natural clusters. Our choice of K is intended to test the algorithm in less than optimal conditions.

5.1 Accuracy Accuracy is measured as follows. The initial centroids are labeled $1, \ldots, K$ (the same centroids for each node and for the centralized K-means). For each $x \in X$, let $L_{cent}(x)$ denote the label of the cluster to which x is assigned at the end of the centralized K-means. Assume x appears at node N_i *i.e.* $x \in X^{(i)}$. Let $L_{p2p}(x)$ denote the label of the cluster at N_i to which x is assigned once the node reaches the termination state. The *percentage of mislabeled points (PMP)* is

$$\frac{100|\{x \in X : L_{cent}(x) \neq L_{p2p}(x)\}|}{|X|}.$$

Let $v_{j,*}^{(i)}$ denote the jth centroid at node N_i once the termination state is reached. Let $v_{j,*}$ denote the jth centroid in the centralized algorithm once it terminates. The *average Euclidean distance* between the jth centroid in P2P K-means and centralized K-means (abbreviated "AED j") is

$$\frac{\sum_{1=1}^{n} ||v_{j,*}^{(i)} - v_{j,*}||}{n}.$$

Figure 2 depicts the accuracy of the final centroids produced by P2P K means. The top row depicts the number of nodes in the network and the next row, AI, depicts the average number of algorithm iterations carried out over all nodes. The maximum and minimum in all cases is no more that 1.5 away from the average. For more than 350 sites, the simulation is forced to stop after 300 global clock ticks due to memory constraints. Note, however, this is only a limitation of the simulator, not the algorithm. Moreover, more than 90% of the nodes had reached their termination state.

The results show that P2P K-means clusters most of the data points correctly as PMP is no more than 1.58% for 350 or less nodes and no more than 7.25% for all simulations. The sharp increase beyond 350 is due to the fact that the simulator was stopped early (300 global clock ticks). There appears to be no significant upward trend as the number of nodes increases to 350, thereby, suggesting that PMP scales well with number of nodes. However, this claim need be verified with larger numbers of nodes, thus, requiring the memory constraints of the simulator to be addressed; we leave this to future work.

The results for AED are consistent with the above observations. Indeed, for less than 400 nodes, the AED is no more than 0.523 and an order of magnitude less in more cases. Given the range over which the dataset spans (more than 50 units in each dimension), this is quite small. Also, for less than 400 nodes, there appears to be no significant upward trend, thereby, supporting the claim that accuracy scales well.

The effect of topology on Accuracy: We expect the performance of P2P K-means to diminish at the topology becomes "thinner" – nodes have fewer neighbors. To test this claim, we simulated our algorithm on a straight-line topology. Delays were assigned to edges uniformly between 1 and 5 (clock ticks). The results are shown in Figure 3. For brevity we do not show AED for each node, but, it does not exceed 1.7 in any case and in the vast majority is less than half this amount.

As expected, we see a large increase in the average number of iterations and the percentage of misclassified points over the BRITE topology case. The PMP becomes as large as 11%. While this is not great accuracy, it is decent considering the straight-line topology represents a worst case for our algorithm. Moreover, the PMP trend suggests good scalability with respect to accuracy. Interestingly, PMP appears to *decrease* (unexpected result). We don't yet have an explanation for this observation.

5.2 Communication Complexity We measure the communication complexity of P2P K-means by counting the number of messages passed during a simulation. Each message is a 4 bytes number *e.g.* a floating point number. The purpose of doing so is to assess scalability in terms of communication load. Figure 4 shows the variation of communication cost with respect to the number of sites. The results suggest that communication cost scales faster than linearly for the BRITE topology, but not so with straight. Using empirical values of I, L, K, and n, we computed the asymptotic communication complexity derived in Section 3.3. The results are always 20 to 40 times greater than the observed empirical communication complexity for the BRITE topology, and roughly 8 times for the straight line topology. This indicates that the worst case has either a very small constant or produces a loose bound.

6 Discussion

In this section, we discuss several issues that remain to be addressed or require further discussion.

6.1 Algorithmic Issues Below we describe how P2P K-means could be modified (if necessary) to address several outstanding algorithmic issues.

Node failure or topology change: If a node leaves the network, as explained in Section 3.1, its neighboring

	50	100	150	200	250	300	350	400	450	500
AI	16	21	13.5	14	19.5	12.5	22.5	34.5	34.5	34.5
PMP	1.98	1.88	1.78	1.73	1.82	1.58	2.04	2.47	3.71	7.25
AED 1	0.187	0.229	0.036	0.0513	0.068	0.0089	0.0084	0.021	0.008	0.012
AED 2	0.523	0.177	0.284	0.331	0.056	0.158	0.105	0.011	0.064	0.134
AED 3	0.022	0.057	0.007	0.025	0.043	0.032	0.011	0.045	0.129	0.021
AED 4	0.024	0.035	0.025	0.003	0.059	0.007	0.011	0.034	0.013	0.002
AED 5	0.092	0.010	0.025	0.038	0.026	0.007	0.013	0.053	0.014	0.032
AED 6	0.019	0.024	0.008	0.046	0.008	0.005	0.012	0.042	0.007	0.004
AED 7	0.036	0.077	0.008	0.067	0.008	0.027	0.011	0.030	0.013	0.030
AED 8	0.015	0.014	0.011	0.005	0.039	0.014	0.015	0.041	0.004	0.024

Figure 2: Accuracy of P2P K-means.

	50	100	150	200	250	300	350
AI	71.5	77.5	101.5	101.5	62.5	80.5	59
PMP	10.17	11.04	10.81	11.01	7.01	7.93	6.91

Figure 3: Accuracy of P2P K-Means (straight-line topology).

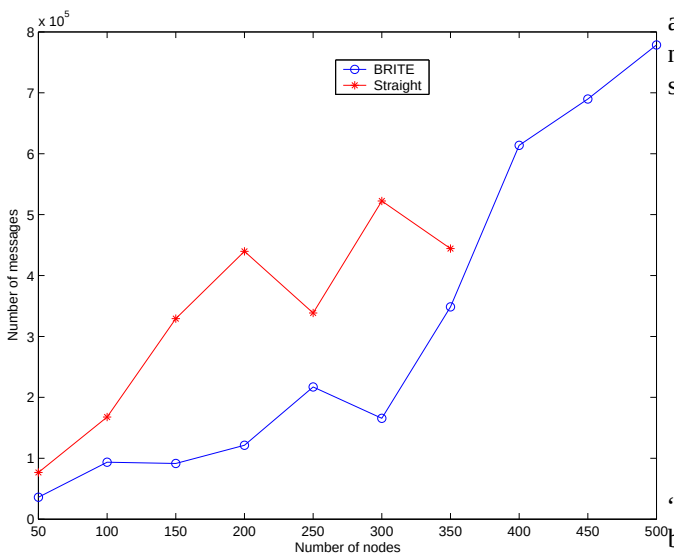

Figure 4: Communication complexity

nodes will discover this and move on to the next iteration without it. Likewise, if an edge goes down, the nodes involved will detect the change. These nodes will refrain from waiting on neighbors no longer accessible. Hence, the algorithm requires no conceptual change to deal with these cases (however, accuracy may suffer).

If a new edges come up, all associated nodes will discover this and, therefore, detect new immediate neighbors. A simple way to augment P2P K-means is to have all associated nodes wait until the next iteration before considering the new neighbors.

Node addition: If a new node, N, joins the network, a more complicated set of modifications is needed. N will need to synchronize itself with its neighbors $Neigh^{(N)}$. A simple way to do this is as follows.

1. N polls its neighbors and each $N_i \in Neigh^{(N)}$ returns its current iteration ℓ_i and all centroids and cluster counts.

2. N sets its iteration number to $\ell = min\{\ell_i : N_i \in Neigh^{(N)}\}$, and its centroids as follows. Choose $N_i \in Neigh^{(N)}$ whose iteration number is ℓ (break ties arbitrarily). For all $1 \leq j \leq K$, set $v_{j,\ell}^{(N)}$ to $v_{j,\ell}^{(i)}$.

3. N begins iteration ℓ as described for P2P K-means.

All immediate neighbors of N which are not in the "terminated" state would simply wait until the next iteration before polling N. However, immediate neighbors $\hat{N}$ that were in the terminated state ought to consider becoming active again. To do this, $\hat{N}$ polls N for its local centroids and cluster counts (the w's). $\hat{N}$ computes an update of its centroids and if the resulting centroids have changed significantly, $\hat{N}$ becomes active again and sends a message to all its immediate neighbors. If any of these neighbors are in the terminated state, they follow the same procedure to determine if they should become active.

Data change: If the data at a node N changes and the N is not in the terminated state, its behavior need not change. If N is in the terminated state, it must determine whether or not to become active again. To do this, N recomputes its local centroids and polls its immediate neighbors for their centroids and cluster counts. Then N updates its centroids and if the resulting centroids have changed significantly, N becomes active again.

In either case, though, N sends a message to all its immediate neighbors indicating a data change. Neighbors not in the terminated state, can ignore the message. Neighbors in the terminated state will determine whether to become active using the same technique as described above in the "Node addition" case.

6.2 Synchronization P2P K-means does not require global synchronization in the sense that all nodes in the network must be on the same iteration. However, the algorithm is not completely asynchronous in the sense that any node can be on any iteration with respect to any other node. The algorithm requires local synchronization in the following sense. Since a node must wait for responses from its immediate neighbors (unless they go down), it cannot move more that one iteration beyond them. Hence the difference in iteration number between two nodes is always upper bounded by the number of network links between the two nodes.

7 Conclusions and Future Work

We have considered the problem of K-means clustering on data homogeneously distributed over a P2P network. We have presented preliminary work on the development of a locally synchronous K-means algorithm for P2P networks. Nodes only communicate and synchronize with their topologically immediate neighbors. Empirical results show, in many cases, the final centroids produced are very close to the final centroids produced by centralized K-means. Consequently, the number of incorrectly labeled data points is small. The communication, however, seems to scale faster than linearly with the number of nodes.

Several algorithmic and experimental issues remain to be addressed in future work.

1. Implement the algorithmic changes discussed in Section 6 to address the issue of new nodes coming into the network and non-static data.

2. Carry out a more elaborate set of experiments to assess accuracy with respect to (i) varying degrees of data heterogeneity *i.e.* the data is not split uniformly across all nodes from the central dataset; (ii) data and topology changes.

Acknowledgments

We thank the U.S. National Science Foundation for support through award IIS-0329143 and CAREER award IIS-0093353. We also thank NASA for support through grant NAS2-37143. Finally we thank Kun Liu, Sanghamitra Bandyopadhyay, Ujjwal Maulik, and Ran Wolff for their valuable contributions.

References

[1] Bandyopadhyay S., Giannella C., Maulik U., Kargupta H., Liu K., and Datta S. Clustering Distributed Data Streams in Peer-to-Peer Environments. *Information Sciences (in press)*, 2005.

[2] Dhillon I. and Modha D. A Data-clustering Algorithm on Distributed Memory Multiprocessors. In *Proceedings of the KDD'99 Workshop on High Performance Knowledge Discovery*, pages 245–260, 1999.

[3] Eisenhardt M., Muller W., and Henrich A. Classifying Documents by Distributed P2P Clustering. In *Proceedings of Informatik 2003, GI Lecture Notes in Informatics, Frankfort, Germany*, pages 286–291, 2003.

[4] Forman G. and Zhang B. Distributed Data Clustering Can Be Efficient and Exact. *SIGKDD Explorations*, 2(2):34–38, 2000.

[5] Han J. and Kamber M. *Data Mining: Concepts and Techniques*. Morgan Kaufman Publishers, San Francisco, CA, 2001.

[6] Kargupta H. and Sivakumar K. Existential Pleasures of Distributed Data Mining. In *Data Mining: Next Generation Challenges and Future Directions, edited by H. Kargupta, A. Joshi, K. Sivakumar, and Y. Yesha, MIT/AAAI Press*, pages 3–26, 2004.

[7] Kowalczyk W., Jelasity M., and Eiben A. Towards Data Mining in Large and Fully Distributed Peer-To-Peer Overlay Networks. In *Proceedings of BNAIC'03*, pages 203–210, 2003.

[8] Wolff R. and Schuster A. Association Rule Mining in Peer-to-Peer Systems. *IEEE Transactions on Systems, Man and Cybernetics - Part B*, 34(6):2426–2438, 2004.

Parallel Mining of Sequential Patterns Using an Efficient Task Partitioning Approach

H. D. K. Moonesinghe, P. N. Tan, M. J. Chung, M. Wu
Department of Computer Science & Engineering
Michigan State University
East Lansing, MI 48824
moonesin, ptan, chung, wuming@cse.msu.edu

ABSTRACT
In sequential pattern mining the availability of large databases and the generation of massive number of frequent subsequences demand scalable high performance solutions involving multiple processors. To address this issue, this paper presents an efficient scalable parallel algorithm based on the *PrefixSpan* sequential pattern mining algorithm. We use task parallelism approach and experiment with several static task partitioning techniques to partition the tasks among processors. Although static task partitioning has the advantage of simplicity in terms of both implementation as well as overhead, it is difficult to precisely estimate the actual workload. To overcome this issue we propose dynamic task partitioning technique that estimates the actual workload by allocating the tasks one at a time to a processor, whenever it becomes free. Moreover, to ensure that the load of each processor is equal, we have extended our approach by including a dynamic subtask partitioning mechanism, where subtasks of the given task is also partitioned among the free processors when load imbalance is detected. Experiments on a 16 processor distributed memory environment show good speedups and scalable performance over different processors and problem sizes. Using our dynamic partitioning technique, we report an average speedup of 13.2 on 16 processors, for different problem sizes.

Keywords
Sequential pattern mining, parallel processing, data mining.

1. INTRODUCTION
Sequential pattern mining has become an essential data mining task to discover previously unknown patterns in vast amount of data. Such knowledge discovery task provides a lot of useful information. For example, analyzing customer buying patterns from sales data collected over a period of time can be used for projections and forecasting. Several algorithms have been proposed such as *GSP* [4], *SPADE* [6], *FreeSpan* [5], SPAM [7] and *PrefixSpan* [3] in the past to find sequential patterns in a sequence database. When considering the availability of massive volume of data, discovering all frequent sequences is still a major issue. The search space is quite large and the serial algorithms are not scalable for large datasets. To address this, it is necessary to study scalable parallel implementations of sequence mining algorithms.

A number of algorithms for parallel sequential mining can be found in the literature; e.g. [2, 8, 9, 10, 14]. Several parallel versions of the *GSP* algorithm are proposed by Shintani & Kitsuregawa in [14], for distributed memory environment. Their method partitions the database equally among the processors and exchanges remote database partitions in each of the iterations. Thus it requires high communication cost and synchronization overhead. Parallel version of the *SPADE* algorithm on shared memory environment was proposed by Zaki in [2]. The algorithm uses a shared database among the processors and an efficient dynamic load-balancing scheme.

In this paper, we present a parallel algorithm based on the *PrefixSpan* sequence mining algorithm, targeting a distributed memory environment. *PrefixSpan* [3] is an efficient sequence mining algorithm, which uses prefix based projection and depth first search strategy to generate sequential patterns. *PrefixSpan* reports fast execution time and consumes less space. Although this algorithm is efficient, when compared with the previously available sequence mining algorithms, it still consumes a lot of computational resources when support threshold is low or the patterns become long [12, 8].

In our proposed parallel algorithm, we decompose the mining phase into a set of prefix-based subsequences called tasks. We use task parallelism approach and partition the tasks among available processors using static and dynamic task partitioning techniques. Each task can be mined separately in main memory without sharing or synchronization. We have used several static task partitioning techniques based on support count of the items. Also, we have proposed dynamic task partitioning technique suitable for distributed memory environment. In this technique, whenever a processor becomes free it obtains a task

from the shared pool of tasks and balances the workload evenly at run time. We have also extended this with a dynamic subtask partitioning mechanism, where unprocessed subtasks of a task are partitioned among the available free processors whenever a load imbalance is detected. The key features of our approach are:

(1) Our algorithm is an asynchronous algorithm where processors work on separate tasks without sharing or synchronization, except when a load imbalance is detected at the end.
(2) We use simple and efficient task scheduling technique based on actual workload, and an efficient subtask partitioning mechanisms for load balancing.
(3) Also, our algorithm has low communication overhead, which is important for a distributed memory environment like cluster of workstations.

We have tested our algorithm using various datasets including several real-world datasets, on a cluster of 16 workstations. With dynamic task partitioning technique we achieved higher speedup improvement, compared to static task partitioning techniques and, an average speedup of 13.2 on 16 processors, for different problem sizes. Sutou et al. [8] have developed a parallel implementation of the *PrefixSpan* algorithm using a dynamic task partitioning mechanism. Our approach is different from their approach as we use an efficient dynamic subtask partition mechanism to balance the workload, when load imbalance is detected. Our subtask partitioning scheme delivered on average 48% speedup improvement over non subtask partitioning scheme such as [8] on 16 processors for skewed data sets, where tasks shows significant difference in workload. Even the static task partitioning techniques proposed in this paper, showed reasonably good speedup for datasets that are not skewed. Our dynamic task partitioning algorithm with sub task partitioning mechanism delivered higher speedups for almost all types of databases tested.

The rest of the paper is organized as follows: Section 2 describes sequential pattern mining and the serial algorithm. Section 3 describes in detail the parallel execution model and our parallel algorithm. Section 4 presents the evaluating environment and the results. Section 5 concludes the paper.

2. SEQUENCE MINING

The problem of mining for sequential patterns was introduced by Agrawal et al [1]. It can be stated as follows: Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of m distinct *items*. An *itemset* S_i is a non-empty subset of items; i.e. $S_i \subseteq I$. Itemset with k items is called a *k-itemset*. Also, an item can occur only once in an itemset.

A *sequence* is an ordered list of itemsets, denoted by $<S_1, S_2, \ldots, S_n>$, where S_i is an itemset. Itemsets in a sequence are ordered according to their timestamp. We assumed that items of an itemset are sorted in increasing or lexicographic order. The *Length* of a sequence is the number of items in that sequence. A sequence with k items is called a *k-sequence*.

A sequence database D consists of set of data sequences. The *support* of a sequence S is defined as the fraction of total data sequences that contain S. A sequence is called *frequent* if its support is above a user specified minimum support threshold. Given a database D of sequences and support threshold t, the problem of mining for sequential patterns is to find all frequent sequences in the database.

2.1 Serial *PrefixSpan* Algorithm

PrefixSpan [3] is a fast sequence mining algorithm that generates complete set of frequent patterns using prefix-based projection. It examines prefix subsequence and projects the corresponding postfix subsequence into a database called *projected* database. Projected databases are constructed for each prefix subsequence. Using the projected database, local frequent items are determined and appended to the prefix subsequence of that projected database recursively to form sequential patterns.

Consider the sample database of 4 sequences: {<(*ab*)(*b*)(*ae*) (*bc*)>, <(*bcd*)(*b*)(*cdf*)>, <(*a*)(*abc*)(*bc*)(*e*)>, <(*b*)(*cd*)(*abd*)(*ad*)>}. By scanning the database we can find length-1 sequential patterns with a minimum support of 2. They are {<*a*>:3, <*b*>:4, <*c*>:4, <*d*>:2, <*e*>:2}, where <*pattern*>:count represents the frequent pattern and its associated support count.

In *PrefixSpan* algorithm, subsequent sequential patterns can be found by constructing corresponding projected database. For prefix pattern <*a*>, *a-projected* database is {<(_*b*)(*b*)(*ae*)(*bc*)>, <(*abc*)(*bc*)(*e*)>, <(_*bd*)(*ad*)>}, where (_*b*) means the last element in the prefix (i.e. *a*) together with *b* form one element [3]. Similarly projected databases are built for other prefix patterns <*b*>, <*c*>, <*d*> and <*e*>.

By scanning projected database once all frequent items *i* are found such that *i* can be assembled to the last element of prefix pattern or (*i*) can be appended to prefix pattern to form a sequence [3]. For example, scanning *a-projected* database all length-2 frequent pattern with prefix <*a*> can be found as {<(*a*)(*a*)>:3, <(*ab*)>:3, <(*a*)(*b*)>:2, <(*a*)(*c*)>:2 <(*a*)(*e*)>:2}. Subsequent patterns can be mined by constructing respective projected databases as before, and mining each recursively.

The major cost of PrefixSpan algorithm is the construction of projected databases. To reduce this cost, pseudo projection technique has been proposed in [3]. Here instead of physically copying the postfix sequences of the database, pointers are used to refer to the sequence in the database.

3. PARALLEL FORMULATION

Our parallel formulation can be best understood when we analyze the computations performed by the *PrefixSpan* algorithm. We can see the computations of the *PrefixSpan* as dynamically expanding irregular sub-trees, each rooted with a prefix subsequence as shown by Figure 1. Also note that the computations at each node and its sub-tree become independent from other nodes and their respective sub-trees.

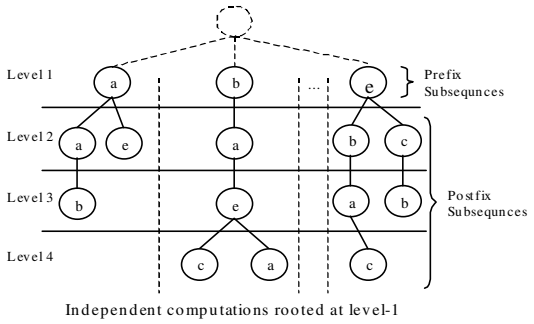

Figure 1: Search space of the *PrefixSpan* algorithm.

We choose task parallelism approach as the main paradigm to implement our parallel *PrefixSpan* algorithm. In our approach each processor is assigned a task or a set of tasks, based on the partitioning technique discussed later, and it then proceeds to independently grow the entire sub-trees rooted at those tasks.

In our parallel algorithm, we decompose search space of the *PrefixSpan* into a set of independent tasks such that the number of tasks (k) is larger than the available number of processors (P). Experimentally, we found k to be at least $4.P$ to achieve good workload balancing. First, we consider the prefix subsequences at level-1 as tasks, if number of such subsequences is larger than $4.P$; otherwise we move to the next level and consider the prefix subsequences at that level as tasks. In Figure 1, {*<aa>*, *<ae>*, *<ba>*, *<eb>*, *<ec>*} are the set of tasks at level-2. Similarly we move level by level deeper in the computational tree until we find the desired number of tasks. A subtask of a task is the postfix item of the corresponding prefix subsequence. Tasks are partitioned among processors using the techniques discussed in the following sections. To achieve maximum performance task partitioning must be done in a way so that the workload of each processor is fairly balanced.

3.1 Static Task Partitioning

In our static partitioning approach each task is assigned a weight based on the support count of the prefix subsequence of the task. Then the set of tasks are partitioned and each subset of tasks is assigned to a processor. We discuss several partitioning methods in the following sections.

3.1.1 Contiguous Task Partitioning (STATIC-1)

In this partitioning method we calculate the average weight of the items to be allocated to a processor by dividing the total weight of all the tasks with the number of processors. Then tasks are sorted in descending order based on weight. We start allocating tasks to a processor contiguously until their total weight is greater than or equal to the average weight. After allocating tasks to a processor we start allocating tasks to the next processor in the same way.

3.1.2 Bin Partitioning (STATIC-2)

In this method each processor is considered as a bin. First, the tasks are sorted in descending order of weight. Here a task is allocated to a processor, with lowest total weight. Initially total weight of each processor is zero. When allocating the next task, processor with the lowest total weight is chosen.

3.1.3 Bin Partitioning Using Subtask Weight (STATIC-3)

In STATIC-2 we used support count of the task as the weight. In STATIC-3, we compute the support count of frequent postfix items of the task, and use sum of support counts as the weight. The idea behind this approach is to improve the accuracy of the workload estimation. Instead of using an estimate based on single support value as in STATIC-1 and STATIC-2, this method looks ahead at the supports of the postfix items, and therefore tend to provide a better workload estimate, with the cost of postfix item generation. After that the task allocation is similar to STATIC2.

3.2 Static Parallel Algorithm

In our parallel algorithm we assume each processor has access to the database using either a shared file system (with parallel I/O facility) or local disks each with a copy of the database. We used the latter approach. The main steps of our static parallel algorithm are as follow:

1. One processor known as *Master* scans sequence database and generate the set of tasks.
2. *Master* processor then partition the set of tasks into P subsets T_i, $1 \leq i \leq P$ using a static task

partition technique and send subset T_i to processor-i for all $1< i \leq P$ (P is the number of processors).

3. Every processor mines for frequent patterns for each task t in the assigned subset T_i

3.3 Dynamic Task Partitioning

The idea behind our static partitioning techniques is that if an item has a high support it will most likely have a deeper computational sub-tree (higher workload) than an item with low support. However estimates based on support count can potentially be very inaccurate, because of the skewness of sequence data. Also, when presenting experimental results we will show that the accuracy of static estimates decreases as the number of processors increases. For this reason, we have developed dynamic task partitioning scheme, where tasks are partitioned to processors at mining time based on the current workload of the processor.

Our parallel model is a master worker model as shown in Figure 2. The task of the worker processor is to generate all the frequent sequences for a given task. *Master* processor does two major jobs: one is the mining to generate frequent sequences and the other is to act as a server for worker processors. Once a worker processor finished generating sequences for a given frequent task, it requests another task from the *Master*. If a request from a worker processor is available *Master* processor temporarily stops mining and process the request by sending a new task to that worker processor. The task list contains tasks sorted (descending order) according to the support count. Thus large jobs will be carried out first by the processors.

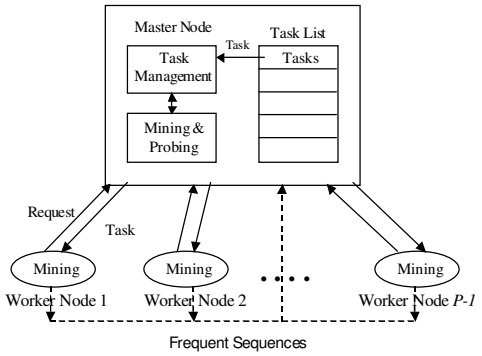

Figure 2: Parallel Execution Model.

The idea behind this approach is to have a shared task list. Since in a distributed memory environment it is not possible to have a shared data structure, we implement this functionality with message passing and client-server architecture. But, here we do not have a dedicated server. Instead *Master* processor frequently probes the system while mining the items. This allows us to utilize all the available processors including the master processor for mining. The communication overhead in this model is minimal. For n frequent tasks the communication overhead is $O(2n)$. In this model, load is balanced fairly, as a processor gets another task only when it is finished executing the current task.

3.3.1 Dynamic Subtask Partitioning

It is possible that the workload balancing of the dynamic method fails when a highly skewed sequence database is used. Since tasks are arranged in the task list based on the support count, larger jobs will be carried out first. But a problem can occur when support count cannot predict the workload of a data set, particularly for a skewed data set. For example, worst situation occurs when P-1 processors are free (i.e. they finish their portion of the allocated tasks and there are no more tasks in the task list) and only one processor is busy (here P is the number of processors). Figure 3 shows this scenario for a skewed dataset. Out of the k tasks available for mining, all the tasks except task-2 are processed completely by the processors while the processor responsible for task-2 is processing subtask-b of task-2.

In order to address this situation, we have to provide a load balancing mechanism so that the free processors will be able to join with busy ones to share the workload. We have solved this problem by partitioning the high workload subtasks of the corresponding task among free processors. In our method, once the free processors are available, a *Coordinator* processor sends a job request message to busy processors and obtains sub-tasks. Then they are distributed to free processors for pattern generation. We accomplish load balancing among the processors by recursively applying this method until all the processors become free. Selection of *Coordinator* processor is carried out dynamically. Here *Master* processor will choose the first processor that becomes free as the *Coordinator* processor. We decided to have a separate processor as the *Coordinator*, rather than using the *Master*, because *Master* itself can be busy with a heavy task.

When executing a task in our parallel model, a processor will always queue the subtasks generated at each level (see Figure 3). When a job request arrives from a *Coordinator* processor, a subset of subtasks will be selected from all the unprocessed subtasks for transfer to the *Coordinator*. Selection of subtasks is done based on their predicted workload. Only the high workload subtasks are selected to minimize the job transfer cost (i.e. the cost of generating projected databases of the *PrefixSpan* method for transferred

subtasks by the receiving processors, and the communication cost).

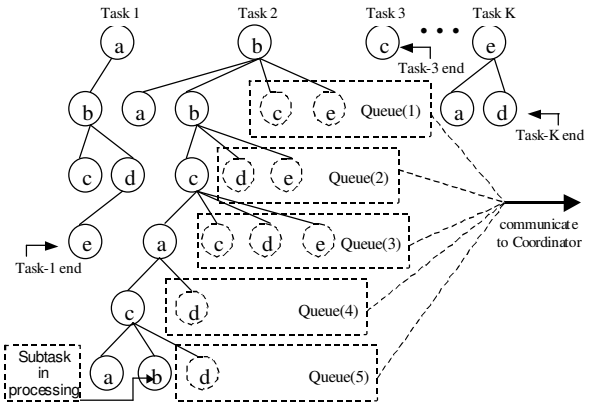

Figure 3: Subtask partitioning for load balancing.

The workload of the subtask is assessed based on the lengths of the projected database of the parent item and the subtask. Projected databases keep shrinking as we go down the computational tree. We found this shrinking factor of the projected database of a subtask as a useful measure in estimating the subtask workload. The data communicated to coordinator processor includes selected items (high workload subtasks) in the queues and the prefix pattern generated up to the last queue. I.e. pattern *bbcac* for the situation in Figure 3.

3.4 Dynamic Parallel Algorithm (DYNAMIC)

Here we present our parallel *PrefixSpan* algorithm using dynamic task and subtask partitioning techniques. The main steps of our dynamic parallel algorithm are as follow:

1. One processor known as *Master* scans sequence database and generate the set of tasks.
2. Master processor sorts the tasks based on support count.
3. Every processor mines for frequent patterns for each task assigned by *Master* processor (see Figure 2).
4. *Master* processor selects the first processor, who becomes free, as *Coordinator*, which carryout following steps:
   - 4.1. *Coordinator* selects free processors available so far for job distribution.
   - 4.2. *Coordinator* will pick busy processors and send job request to obtain jobs (sub tasks).
   - 4.3. Busy processors, who receive a job request, send high workload jobs to *Coordinator* (see Figure 3).
   - 4.4. *Coordinator* distributes the jobs among the available free processors for mining.
   - 4.5. Repeat steps (4.1- 4.5) until all the processors become free.

4. EXPERIMENTAL EVALUATION

In this section we describe the evaluation environment used to execute our algorithms and the results obtained.

4.1 Evaluating Environment

Our machine environment consists of a cluster of 16 *Ultra Sparc* II workstations (512MB memory) connected by 100Mbps network. We used Message Passing Interface (MPI) library (*mpich* -1.2.5.2) [13] to achieve parallel communication. We have implemented the serial *PrefixSpan* algorithm as described in [3] and compared the resulting sequences with the sequences generated by the *PrefixSpan* executable program obtained from the authors of [3].

We used sequence databases generated by the IBM Quest synthetic data generator [11]. The dataset contains 1,000 different items with an average length of 10 to 20 items per sequence.

We have also used real-world datasets, Earth Science data and bio-datasets. The Earth Science data consists of monthly measurements of 0.5-degree precipitation on land, collected over a period of 18 years starting January 1982 to December 1999. We have transformed these time series data into a sequence of HI and LO events. The bio-datasets *Snake* and *Pi* are very dense datasets and can generate large number of short sequences with a higher threshold like 55%. *Snake* dataset contains 174 Toxin-Snake protein sequences and 20 different items. Dataset *Pi* contains 190 protein sequences and 21 different items. Table 1 shows the characteristics of all of our test data sets.

Dataset	Num. Seq.	Threshold
C10T5S4 I1.25D100k	100,000	0.10%
C20T5S8I1.25D100k	100,000	0.50%
C10T5S4I2.5D100k	100,000	0.25%
C20T2.5S4I1.25D100k	100,000	1.00% -0.05%
Precipitation	67,034	10%
Snake	174	55%
Pi	190	90%

Table 1: Characteristics of the datasets.

4.2 Experimental Results

We tested both parallel and serial algorithms on a workstation cluster environment and execution time was recorded. Figure 4 shows execution time and speedup for some generated datasets. Here number of processors (P) =1 corresponds to the execution time of the serial program.

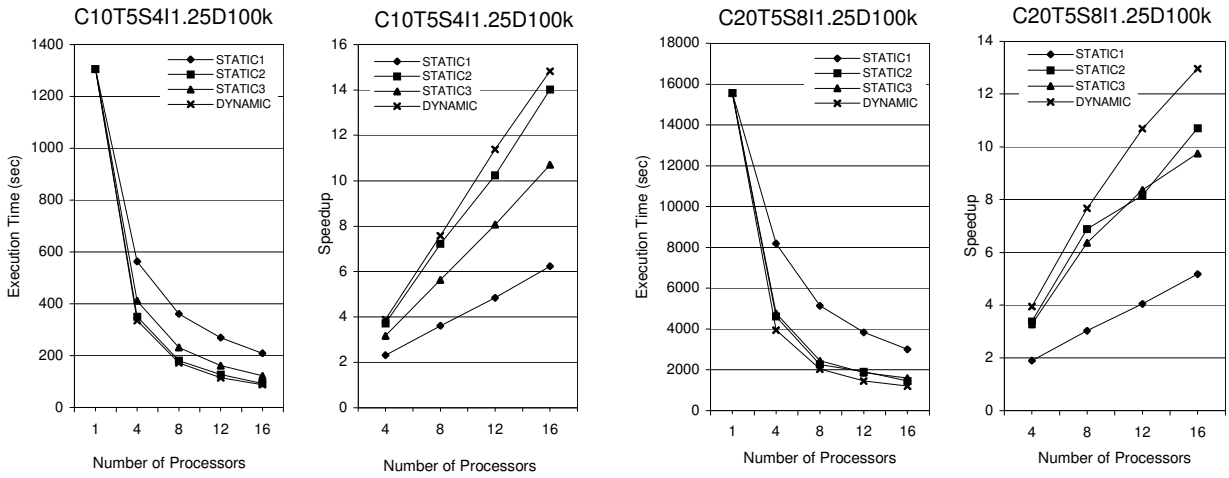

Figure 4: Execution time and speedup of generated datasets.

For all the datasets tested, we obtained near linear speedups with dynamic parallel algorithm (DYNAMIC). For some datasets we achieved a maximum speedup of 15.1 on 16 processors. Speedup achieved by STATIC-1 is the lowest. This is because of the lack of accurate workload balancing of STATIC-1. It partitions the tasks contiguously to processors and the processor with the high-support tasks takes higher execution time. STATIC-3, although has a better estimation of workload, the overhead of calculating total support of the sub-tasks is a bottleneck. Therefore, we found that both STATIC-1 and STATIC-3 techniques are not suitable for our parallel *PrefixSpan* algorithm.

STATIC-2 shows good results when compared with other static partitioning techniques. Since STATIC-2 depends on support counts for workload estimation, it cannot outperform DYNAMIC algorithm. This is significant for databases with high workload (i.e. that generate deeper computational trees) such as *C20T5S8I1.25D100k*. In this dataset, the average length of maximal potential large sequence is 8. For this dataset, there is a higher difference of speedup between DYNAMIC and STATIC2.

We have also analyzed the performance of our parallel algorithm on real world datasets. Execution time and speedup for *Precipitation* and *Pi* datasets are shown in Figure 5 for DYNAMIC algorithm. *Pi* is a very dense dataset, which generates massive number of sequences at very high threshold like 90%. Note that the execution time decreases almost linearly with the increase in number of processors. Our DYNAMIC algorithm tends to show higher performance for datasets with larger workloads.

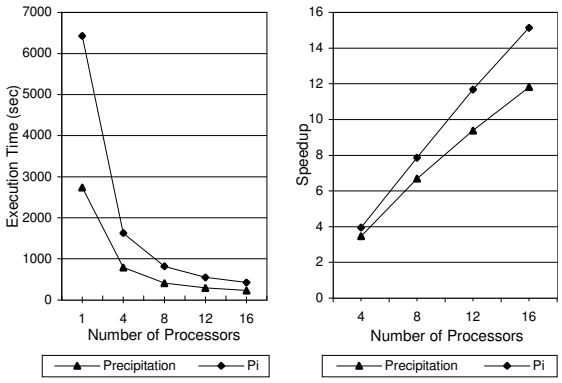

Figure 5: Execution time and speedup of real world datasets with DYNAMIC.

We analyzed the workload balancing of static task partitioning technique (STATIC-2) and the dynamic task partitioning techniques. We have compared two different dynamic task partitioning techniques: DYNAMIC_NO_STP (dynamic load balancing without sub task partitioning, which is similar to [8]) and DYNAMIC (with sub task partitioning). Figure 6 shows total mining time spent by each processor in 12-processor case for *Snake* dataset. Workload balancing of STATIC-2 is not that good as some processors spent lower time while others spent higher time for mining. Although DYNAMIC_NO_STP achieves fair

workload balancing compared to STATIC-2, there are situations that the workload is not balanced. For example, in Figure 6 processor-4 and processor-9 show high workloads when compared with rest of the processors. But with DYNAMIC we can see a fair workload distribution among all the processors.

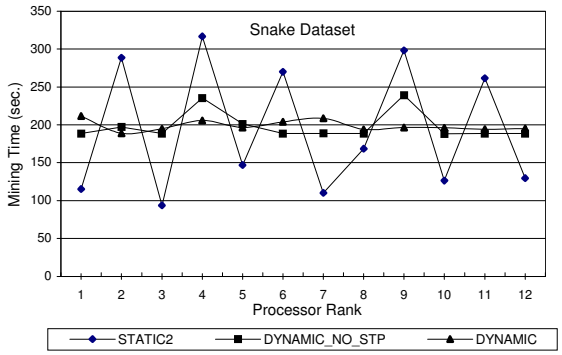

Figure 6: Variation of mining time for each processor.

We have also studied the effect of changing minimum support on the parallel performance. We used 16 processors with *C20T2.5S4I1.25D100k* dataset and minimum support threshold varied from 1.00% to 0.05%. Figure 7 shows execution time and the number of frequent sequences generated by the parallel algorithm, for different minimum support thresholds. Execution time goes from 33.4 sec. at 0.5% minimum support to 379.0 sec. at 0.05% minimum support, a time ratio of 1:12 vs. a support ratio of 1:10. Also, the number of frequent sequences goes from 61,018 at 0.5% support to 6,477,006 at 0.05% support, a ratio of 1:107 vs. a support ratio of 1:10. It appears that in general execution time is near linear with respect to minimum support. Further, it can be said that the efficiency of DYNAMIC algorithm increases with decreasing support, as it generates more frequent sequences per second on lower support values.

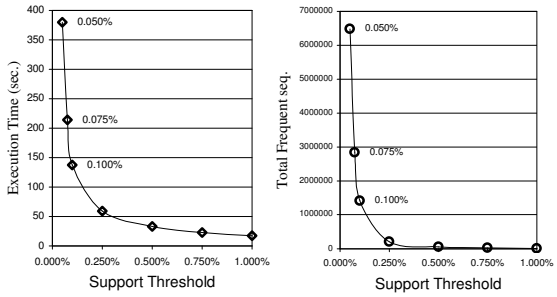

Figure 7: Effect of minimum support threshold on parallel performance.

Finally, we study the effect of dynamic subtask partitioning technique on highly skewed datasets. We used datasets that have prefix subsequences with lower support count but deeper computational sub trees. Lower support count will cause our DYNAMIC algorithm to select that task for mining at the end, since we mine items with higher support count first. The datasets we used (*C10T5S4I2.5D100k* and *Snake*) have these properties and thus it allows us to see the gain we obtained from subtask partitioning.

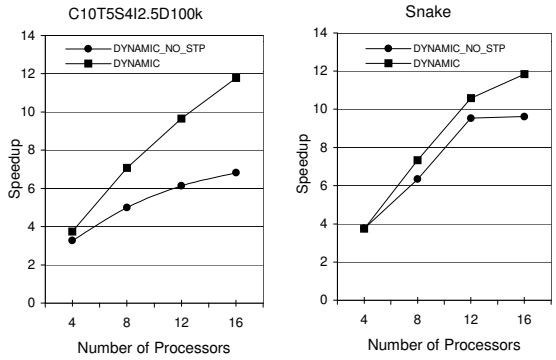

Figure 8: Speedup of dynamic partitioning techniques.

Figure 8 shows the speedup we achieved from both dynamic task partitioning techniques (DYNAMIC & DYNAMIC_NO_STP). It can be seen that, for higher number of processors (e.g. 16-processor case) we achieve impressive speedup improvement with dynamic subtask partitioning mechanism. Also, in the *Snake* dataset we can see scalable speedup with DYNAMIC algorithm. DYNAMIC_NO_STP shows a reduction of speedup when moving from 12 to 16 processors for that dataset. Therefore, dynamic sub task partitioning technique not only improves speedup but also improves the scalability.

Our dynamic sub task partitioning technique shows a maximum speedup improvement of 23% with *Snake* dataset and 72% with *C10T5S4I2.5D100k* dataset on 16-processor environment, when compared with DYNAMIC_NO_STP. On average DYNAMIC parallel algorithm records 7%, 28%, 34% and 48% speedup improvement over DYNAMIC_NO_STP algorithm on 4, 8, 12 and 16 processors respectively. It is interesting to note that the speedup improvement increases with increasing number of processors.

5. CONCLUSIONS

In this paper, we have presented an efficient scalable parallel sequence mining algorithm on distributed memory multiprocessor systems. The proposed algorithm draws its motivation from the *PrefixSpan* [3] sequential mining algorithm. We used task parallelism approach to partition the tasks among processors and analyzed several task partitioning techniques.

In almost all the test cases, we found that the dynamic task partitioning technique together with the efficient subtask partitioning mechanism, showed better speedup and scalability than that of the static partitioning and dynamic partitioning without subtask partitioning. The main reason behind this performance is the accurate load balancing done by the dynamic subtask partitioning method, whenever a load imbalance is detected among processors. We have tested our algorithm using several data sets on a 16-processor distributed memory environment. The results presented in this paper are among the best known in the literature.

We intend to run our parallel algorithm with higher number of processors using larger data sets and also plan to study the performance of our algorithm on different platforms and different workload characteristics.

ACKNOWLEDGEMENTS

We would like to thank Dr. Jiawei Han and Dr. Jian Pei for providing the executable version of their *PrefixSpan* algorithm. Thanks also go to Dr. Jianyong Wang for providing us the protein sequence data.

REFERENCES

[1] R. Agrawal, R. Srikant. Mining Sequential Patterns. *Proc. of the Int'l Conf. on Data Engineering (ICDE)*, Taipei, Taiwan, March 1995.

[2] M. J. Zaki. Parallel Sequence Mining on Shared-Memory Machines. *In Journal of Parallel and Distributed Computing, special issue on High Performance Data Mining*, Volume 61, No. 3, pp. 401-426, March 2001.

[3] J. Pei, J. Han, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth. *Proc. 2001 Int. Conf. on Data Engineering (ICDE'01)*, Heidelberg, Germany, April 2001.

[4] R. Srikant, R. Agrawal. Mining sequential patterns: generalizations and performance improvements. *Proc. of the Fifth Int'l Conf. on Extending Database Technology (EDBT)*, Avignon, France, March 1996.

[5] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, M.-C. Hsu. FreeSpan: Frequent Pattern-Projected Sequential Pattern Mining, *Proc. 2000 Int. Conf. on Knowledge Discovery and Data Mining (KDD'00),* Boston, MA, August 2000.

[6] M. J. Zaki. SPADE: An Efficient Algorithm for Mining Frequent Sequences. *In Machine Learning Journal, special issue on Unsupervised Learning (Doug Fisher, ed.),* pp 31-60, Vol. 42 Nos. 1/2, Jan/Feb 2001.

[7] J. Ayres, J. Flannick, J. Gehrke, T. Yiu. Sequential pattern mining using a bitmap representation. *In SIGKDD*, pages 429--435, July 2002.

[8] T. Sutou, K. Tamura, Y. Mori, Hajime Kitakami. Design and Implementation of Parallel Modified PrefixSpan Method. *ISHPC*, Tokyo-Odaiba, Japan 2003

[9] A. Demiriz. webSPADE: A Parallel Sequence Mining Algorithm to Analyze Web Log Data. *Proc. of ICDM 2002*.

[10] V. Guralnik, N. Garg, G. Karypis. Parallel tree projection algorithm for sequence mining. *In European Conference on Parallel Processing*, 2001.

[11] IBM Almaden. *Synthetic Data Generation Code for Associations and Sequential Patterns*. http://www.almaden.ibm.com/cs/quest/syndata.html

[12] J. Wang, J. Han. BIDE: Efficient Mining of Frequent Closed Sequences. *Proc. 2004 Int. Conf. On Data Engineering (ICDE'04)*, Boston, MA, March 2004.

[13] *MPI home page*. http://www.mcs.anl.gov/mpi

[14] T. Shintani, M. Kitsuregawa. Mining Algorithms for Sequential Patterns in Parallel: Hash Based Approach. In *Research and Development in Knowledge Discovery and Data Mining: Second Pacific-Asia Conference (PAKDD'98)*, Melbourne, Australia, April 1998.

Distributed Mining of Frequent Closed Itemsets: Some Preliminary Results

Claudio Lucchese
Ca' Foscari University of Venice
clucches@dsi.unive.it

Salvatore Orlando
Ca' Foscari University of Venice
orlando@dsi.unive.it

Raffaele Perego
ISTI-CNR of Pisa
perego@isti.cnr.it

Abstract

In this paper we address the problem of mining frequent closed itemsets in a distributed setting. We figure out an environment where a transactional dataset is horizontally partitioned and stored in different sites. We assume that due to the huge size of datasets and privacy concerns dataset partitions cannot be moved to a centralized site where to materialize the whole dataset and perform the mining task. Thus it becomes mandatory to perform separate mining on each site, and then merge the local results do derive a global knowledge. This paper shows how frequent closed itemsets, mined independently in each site, can be merged in order to derive globally frequent closed itemsets. Unfortunately, such merging might produce a superset of all the frequent closed itemsets, while the associated supports could be smaller than the exact ones because some globally frequent closed itemsets might be not locally frequent in some partition. A post-processing phase is thus needed to compute exact global results.

1 Introduction.

The frequent itemset mining (FIM) problem has been extensively studied in the last years. Several variations to the original Apriori algorithm [3], as well as completely different approaches, have been recently proposed (see, for example, the papers presented at the last two FIMI workshops [1, 2]). Recently it has been shown that *frequent closed itemsets* [10, 11, 4, 9, 15, 14, 6] are particularly interesting because they provide qualitatively the same information, by guaranteeing at the same time better performance, non redundant results, and concise representation.

In this paper we investigate a novel topic: distributed mining of frequent closed itemsets. While some papers address the problem of mining all the frequent itemsets in a distributed environments, at our best known, no proposal for distributed closed itemset mining exists. We figure out a distributed framework in which one (virtually single) transactional dataset $\mathcal{D}$ is horizontally partitioned into N parts. Each transaction of the dataset consists of an ordered list of items in the set of possible items $\mathcal{I}$. Each partition $\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_N$ is made up of a subset of the transactions in $\mathcal{D}$, and there are no overlaps among the partitions.

In order to mine $\mathcal{D}$ we could devise a very simple strategy based on the "moving data" approach. We could choose a central site, which will receive every partition and, once materialized the whole dataset, will perform the mining task. The main drawback of this approach is that real datasets are usually huge, and it is not feasible to send an entire partition to a central site because of communication costs. Moreover, a central site could not be able to mine it as a whole, but it should have to partition it again in some way. In addition, if we have to deal with privacy concerns, we cannot move sensible data from one site to another. Hence, it is mandatory to employ a "moving results" approach, according to which frequent closed itemsets are independently extracted on each site, and then are collected and merged to get the global solution. Such algorithms are more suitable for *loosely-coupled distributed settings*, since they limit the number of communications/synchronizations between distributed nodes.

In the last years researchers have proposed many data mining algorithms that can efficiently be run in a loosely-coupled setting [8, 5], even if a few of them regards the FIM problem[1]. In particular, *Partition* [12], i.e. an algorithm for mining all the frequent

[1] On the other hand, frequent itemset mining has been extensively studied in the context of tightly-coupled environments, like parallel clusters.

itemsets, can be straightforwardly implemented in a loosely-coupled distributed setting, i.e. with a limited number of communications/synchronizations (see for example [7]).

In this paper we want to explore whether a *Partition*-like approach can also be exploited to mine frequent closed itemsets in a distributed environment. In order to demonstrate the feasibility of this approach, we show how the various local results, extracted independently on each site with the preferred closed itemset mining algorithm, can be merged in order to obtain the set of all the globally frequent closed itemsets.

Unfortunately, similarly to what happen in the distributed extraction of all the frequent itemsets, merging local results may produce a not exact set of all the frequent closed itemsets, while the associated supports could be unknown. A post-processing phase is thus necessary to compute exact global results.

The rest of the paper is organized as follows. Section 2 discusses some preliminary results concerning the merging function proposed to join the local results computed independently in the various distributed sites. Section 3 describes the additional steps needed to retrieve the exact supports of closed itemsets, and, finally, Section 4 draws some conclusions and future work.

2 Closed Itemsets Extraction

Our goal is to find a distributed algorithm that minimizes synchronizations and communications in mining frequent closed itemsets.

For the sake of simplicity, we limit the following discussion to a transactional dataset $\mathcal{D}$ that is partitioned in only two parts. More formally, let $\mathcal{D}_1$ and $\mathcal{D}_2$ be the two disjoint horizontal partitions of $\mathcal{D}$, where $\mathcal{D}_1 \cup \mathcal{D}_2 = \mathcal{D}$, and $\mathcal{D}_1 \cap \mathcal{D}_2 = \emptyset$. However, we can easily generalize our results to n partitions, for an arbitrary value of n.

Moreover, in this section we do not consider the issues concerning the (relative) minimum support threshold used to locally mine the various partitions. Remember that some globally frequent (closed) itemsets might not be extracted at all from some partitions, while some globally infrequent ones might be mined from others. In other words, the method discussed below surely works when we mine all the partitions with an (absolute) support $supp = 1$, i.e. when we consider as frequent any itemset which occur at least once in $\mathcal{D}$. We will take in account the issues concerning the minimum support constraint in the last part of the paper.

2.1 Preliminaries. Given a transactional dataset $\mathcal{D}$, let T and I, $T \subseteq \mathcal{D}$ and $I \subseteq \mathcal{I}$, be subsets of all the transactions and items appearing in $\mathcal{D}$, respectively.

The concept of *closed itemset* is based on the two following functions f and g:

$$f(T) = \{i \in \mathcal{I} \mid \forall t \in T, i \in t\}$$
$$g(I) = \{t \in \mathcal{D} \mid \forall i \in I, i \in t\}.$$

Function f returns the set of itemsets included in all the transactions belonging to T, while function g returns the set of transactions supporting a given itemset I. We can also consider $g(i)$ and $g(X)$ to be *tid-lists*, i.e. lists of identifiers associated with all the transactions in $\mathcal{D}$ set-including item i and itemset X, respectively.

Definition 2.1. *An itemset I is said to be* closed *if and only if*

$$c(I) = f(g(I)) = f \circ g(I) = I$$

where the composite function $c = f \circ g$ is called Galois operator *or* closure operator.

Given a dataset partition $\mathcal{D}_j$, let T_j and I_j, $T_j \subseteq \mathcal{D}_j$ and $I_j \subseteq \mathcal{I}_j$, be subsets of all the transactions and items appearing in $\mathcal{D}_j$, respectively. We can thus redefine the two following functions:

$$f_j(T_j) = \{i \in \mathcal{I}_j \mid \forall t \in T_j, i \in t\}$$
$$g_j(I_j) = \{t \in \mathcal{D}_j \mid \forall i \in I_j, i \in t\}.$$

Therefore we can consider $g_j(i)$ and $g_j(X)$ be *tid-lists* only referring to transactions in $\mathcal{D}_j$ which set-include item i and itemset X, respectively. Similarly, we can define $c_j(I_j) = f_j(g_j(I_j))$, $\forall I_j \subseteq \mathcal{I}_j$.

Let $\mathcal{C}$ be the collection of all the closed itemsets in $\mathcal{D}$, and $\mathcal{C}_1$ and $\mathcal{C}_2$ be the two sets of closed itemsets mined from $\mathcal{D}_1$ and $\mathcal{D}_2$, respectively. Before introducing the theorems stating the properties concerning $\mathcal{C}_1$ and $\mathcal{C}_2$, and their relationship with $\mathcal{C}$, let us introduce a couple of important lemmas concerning the closure operator $c()$ with respect to the dataset $\mathcal{D}$. The same lemmas also hold for operators $c_1()$ and $c_2()$ relative to $\mathcal{D}_1$ and $\mathcal{D}_2$, respectively.

Lemma 2.1. *Given an itemset X and an item $i \in \mathcal{I}$, $g(X) \subseteq g(i) \Leftrightarrow i \in c(X)$.*

Proof. Proof.
$(g(X) \subseteq g(i) \Rightarrow i \in c(X))$:

Since $g(X \cup i)$[2] $= g(X) \cap g(i)$, $g(X) \subseteq g(i) \Rightarrow g(X \cup i) = g(X)$. Therefore, if $g(X \cup i) = g(X)$ then $f(g(X \cup i)) = f(g(X)) \Rightarrow c(X \cup i) = c(X) \Rightarrow i \in c(X)$.

[2]For the sake of readability, we will drop parentheses around singleton itemsets, i.e. we will write $X \cup i$ instead of $X \cup \{i\}$, where single items are represented by lowercase characters.

$(i \in c(X) \Rightarrow g(X) \subseteq g(i))$:

If $i \in c(X)$, then $g(X) = g(X \cup i)$. Since $g(X \cup i) = g(X) \cap g(i)$, $g(X) \cap g(i) = g(X)$ holds too. Thus, we can deduce that $g(X) \subseteq g(i)$.

LEMMA 2.2. *If $Y \in \mathcal{C}$, and $X \subset Y$, then $c(X) \subseteq Y$.*

Proof. Note that $g(Y) \subseteq g(X)$ because $X \subseteq Y$. Moreover, Lemma 2.1 states that if $j \in c(X)$, then $g(X) \subseteq g(j)$. Thus, since $g(Y) \subseteq g(X)$, then $g(Y) \subseteq g(j)$ holds too, and from Lemma 2.1 it also follows that $j \in c(Y)$. So, if $j \notin Y$ held, Y would not be a closed itemset because $j \in c(Y)$, and this is in contradiction with the hypothesis.

2.2 Local vs. global closed itemsets Our goal is to show that it is possible to perform independent computations on each partition of the dataset, and then join the local result by using an appropriate merging function $\oplus$ in order to obtain the global results. In this section we describe such merging function, and show that $\mathcal{C} \equiv \overline{\mathcal{C}}$, where $\overline{\mathcal{C}} = \mathcal{C}_1 \oplus \mathcal{C}_2$.

THEOREM 2.1. *Given the two sets of closed itemsets $\mathcal{C}_1$ and $\mathcal{C}_2$, mined respectively from the two datasets $\mathcal{D}_1$ and $\mathcal{D}_2$, we have that:*

$$\overline{\mathcal{C}} \;=\; \mathcal{C}_1 \oplus \mathcal{C}_2 \;=$$
$$(\mathcal{C}_1 \cup \mathcal{C}_2) \;\cup\; \{X_1 \cap X_2 \mid (X_1, X_2) \in (\mathcal{C}_1 \times \mathcal{C}_2)\} \equiv \mathcal{C}.$$

Therefore we can obtain $\mathcal{C}$ by collecting the closed itemsets contained in $\mathcal{C}_1$ and $\mathcal{C}_2$, and intersecting them to obtain further ones. We prove the above theorem by showing that the double inclusion holds.

THEOREM 2.2.

$$(\mathcal{C}_1 \cup \mathcal{C}_2) \subseteq \mathcal{C}$$

Proof. By definition, X is a closed itemsets in $\mathcal{D}$ *if and only if* X occurs in $\mathcal{D}$, and

$$\forall i \notin X \;:\; g(X) \not\subseteq g(i).$$

Therefore,

$$X \in \mathcal{C}_1 \Rightarrow \forall i \notin X \;:\; g_1(X) \not\subseteq g_1(i),$$

Since

$$g_1(X) \not\subseteq g_1(i) \;\Rightarrow\; g(X) \not\subseteq g(i),$$

we have that

$$X \in \mathcal{C}_1 \;\Rightarrow\; \forall i \notin X \;:\; g(X) \not\subseteq g(i),$$

and therefore X is closed in $\mathcal{D}$, i.e. $X \in \mathcal{C}$.

The same clearly holds also if $X \in \mathcal{C}_2$.

THEOREM 2.3. *Given $X_1 \in \mathcal{C}_1$ and $X_2 \in \mathcal{C}_2$, we have that*

$$Z = (X_1 \cap X_2) \;\in\; \mathcal{C}$$

Proof. If $Z \in \mathcal{C}_1$ or $Z \in \mathcal{C}_2$, then $Z \in \mathcal{C}$ because $(\mathcal{C}_1 \cup \mathcal{C}_2) \subseteq \mathcal{C}$ (see Theorem 2.2).

So in the following we will consider the *non-trivial* case, i.e. $Z \notin \mathcal{C}_1$ and $Z \notin \mathcal{C}_2$. In other terms, we are interested to the case in which $Z \subset X_1$ and $Z \subset X_2$.

By absurd, assume that Z is not closed, so that $\exists i \notin Z \mid g(Z) \subseteq g(i)$. So, we have that:

$$g(Z) \subseteq g(i) \;\Rightarrow\; g_1(Z) \subseteq g_1(i)$$
$$g(Z) \subseteq g(i) \;\Rightarrow\; g_2(Z) \subseteq g_2(i).$$

or, equivalently, $i \in c_1(Z)$ and $i \in c_2(Z)$.

By Lemma 2.2, it follows that $c_1(Z) \subseteq X_1$ and $c_2(Z) \subseteq X_2$, since X_1 and X_2 are closed in $\mathcal{D}_1$ and $\mathcal{D}_2$, respectively. So, the only way to choose an i that belongs to both $c_1(Z)$ and $c_2(Z)$, where $c_1(Z) \subseteq X_1$ and $c_2(Z) \subseteq X_2$, is that $i \in Z = (X_1 \cap X_2)$. But this is in contradiction with the hypothesis by absurd, i.e. $i \notin Z$.

The following corollary is a simple consequence of the above Theorem.

COROLLARY 2.1.

$$\{X_1 \cap X_2 \mid (X_1, X_2) \in (\mathcal{C}_1 \times \mathcal{C}_2)\} \subseteq \mathcal{C}$$

Theorem 2.2 and Corollary 2.1 show that $\overline{\mathcal{C}} \subseteq \mathcal{C}$. Our merge function is thus *correct*, in the sense that any itemset $X \in \overline{\mathcal{C}}$ is also a closed itemset in the global dataset $\mathcal{D}$.

In the following we prove the opposite implication, i.e. $\mathcal{C} \subseteq \overline{\mathcal{C}}$, which allow us to show that our merge function is also *complete*, and therefore $\mathcal{C} \equiv \overline{\mathcal{C}}$.

THEOREM 2.4.

$$\mathcal{C} \subseteq \overline{\mathcal{C}}$$

Proof. Let $X \in \mathcal{C}$ be an itemset belonging to some transactions of $\mathcal{D}$. Therefore X is included in some transactions of either $\mathcal{D}_1$ or $\mathcal{D}_2$, or it is included in both $\mathcal{D}_1$ and $\mathcal{D}_2$.

If X only occurs in transactions of one partition, either $\mathcal{D}_1$ or $\mathcal{D}_2$, we can trivially show that $X \in \overline{\mathcal{C}}$. Suppose that this partition is $\mathcal{D}_1$. Therefore $g_1(X) = g(X)$, and $g_2(X) = \emptyset$.

Since $c(X) = X$ by hypothesis, then $\forall i \notin X \mid g(X) \not\subseteq g(i)$. Then it also holds that $\forall i \notin X \mid g_1(X) \not\subseteq g(i)$, because $g_1(X) = g(X)$. Since $g_1(i) \subseteq g(i)$, then it also holds that $\forall i \notin X \mid g_1(X) \not\subseteq g_1(i)$, or, equivalently, $X = c_1(X)$. Therefore, in this case $X \in \overline{\mathcal{C}}$ surely holds, because $X \in \mathcal{C}_1$.

If X appears in transactions of $\mathcal{D}_1$ and $\mathcal{D}_2$, then we can compute its closure in both of them, i.e. $X_1 = c_1(X)$ and $X_2 = c_2(X)$. So either $X = (X_1 \cap X_2)$ or $X \subset (X_1 \cap X_2)$ can hold.

If $X = (X_1 \cap X_2)$, then $X \in \overline{\mathcal{C}}$ by definition of $\overline{\mathcal{C}}$.

The second condition $X \subset (X_1 \cap X_2)$ can not hold. If $X \subset (X_1 \cap X_2)$, then $\exists i \notin X$ such that $i \in c_1(X)$ and $i \in c_2(X)$. Hence $g_1(X) \subseteq g_1(i)$ and $g_2(X) \subseteq g_2(i)$. Since $g(X) = g_1(X) \cup g_2(X)$ and $g(i) = g_1(i) \cup g_2(i)$, then $g(X) \subseteq g(i)$ also holds, i.e. $i \in c(X) = X$. But this is in contradiction with the hypothesis that $i \notin X$.

Note that from a theoretical point of view, itemsets in $\mathcal{C}$ form a lattice, i.e. for every $X, Y \in \mathcal{C}$ their join element $X \cup Y$ and their met element $X \cap Y$ belong to $\mathcal{C}$ [15, 14]. We have just shown that if $X \in \mathcal{C}_1$ or $X \in \mathcal{C}_2$, then $X \in \mathcal{C}$. Moreover, in order to complete the lattice $\mathcal{C}$, it is also needed to set-intersect each pair $(X_1, X_2) \in (\mathcal{C}_1 \times \mathcal{C}_2)$, and add $X_1 \cap X_2$ to $\mathcal{C}$. The proofs of both the implications show that this is enough to complete the lattice $\mathcal{C}$, i.e. $\mathcal{C} \equiv \overline{\mathcal{C}}$.

Till now we have defined a merging function $\oplus$, used to obtain $\mathcal{C}$ as $\mathcal{C}_1 \oplus \mathcal{C}_2$. This result can be generalized to the case of N partitions of $\mathcal{D}$.

THEOREM 2.5. *Given the sets of closed itemsets $\mathcal{C}_1, \ldots, \mathcal{C}_N$ mined respectively from N disjoint horizontal partitions $\mathcal{D}_1, \ldots, \mathcal{D}_N$ of $\mathcal{D}$, we have that:*

$$\overline{\mathcal{C}} = (\ldots((\mathcal{C}_1 \oplus \mathcal{C}_2) \oplus \ldots \oplus \mathcal{C}_N)\ldots).$$

Proof. It is easy to give a proof by induction. We have already proved with Theorem 2.1 that the equality holds in the case $N = 2$.

Suppose that Theorem 2.5 holds for N, we want to show that it holds for $N + 1$ as well. Given the $N+1$ partitions, by hypothesis we know that the closed itemsets in $\mathcal{D}' = \{\mathcal{D}_1 \cup \ldots \cup \mathcal{D}_N\}$ are $\mathcal{C}' = ((\mathcal{C}_1 \oplus \mathcal{C}_2) \oplus \ldots \oplus \mathcal{C}_N)$. At this point, we can think at the dataset $\mathcal{D}$ as it was made of two partitions only, i.e. $\mathcal{D} = \mathcal{D}' \cup \mathcal{D}_{N+1}$. Therefore, we can apply Theorem 2.1 and get $\overline{\mathcal{C}} = \mathcal{C}' \oplus \mathcal{C}_{N+1} = (\ldots((\mathcal{C}_1 \oplus \mathcal{C}_2) \oplus \ldots \oplus \mathcal{C}_N)\ldots)\ldots\mathcal{C}_{N+1}$.

While computing the union of the various $\mathcal{C}_i$ is straightforward, the same is not true for the intersections. In order to quantify the impact of intersections in the overall computation, we experimentally counted the number of closed itemsets which have to be calculate by intersections as a function of the number of partitions. Figure 1 plots this number in a real world case: the mushroom dataset mined with absolute minimum support 1. From the figure we can see that the number of *intersection itemsets* increases as the number of partition grows, even if this growth is less than linear.

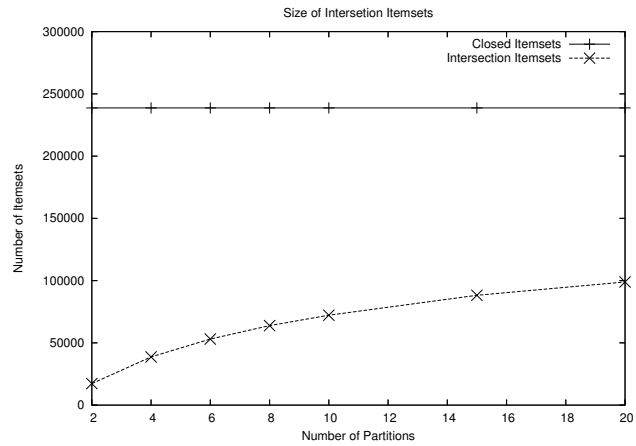

Figure 1: Intersection Itemsets in a real world case: the mushroom dataset mined with absolute minimum support 1.

When the dataset is split into 20 partitions, about 2/5 of all frequent closed itemsets have to be computed with intersections.

3 Computing the supports of frequent closed itemsets

Theorem 2.1 shows a way to devise the identities of all the closed itemsets in $\mathcal{D}$ given the closed itemsets found in its partitions $\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_N$. In this section we show how to calculate the supports of such itemsets given the supports of all the itemsets in $\mathcal{C}_i, \forall i$.

Given $X \in \overline{\mathcal{C}}$, we denote with $\|X\|_i$ the support of X in partition $\mathcal{D}_i$. The global support of X is clearly $\sum_{i=1,\ldots,N} \|X\|_i$. Note that by construction of $\overline{\mathcal{C}}$, it may happen that $X \notin \mathcal{C}_i$ for some partition $\mathcal{D}_i$, and therefore, even if the support of every itemset in $\mathcal{C}_j$ for every partition j is known, the support $\|X\|_i$ may not be known. In fact, for each $X \in \overline{\mathcal{C}}$, we can distinguish between two cases:

$\exists i \mid X \in \mathcal{C}_i$,

i.e. X was obtained by union of the various $\mathcal{C}_j$. In this case $\|X\|_i$ is given, but we cannot say anything about $\|X\|_{j \neq i}$.

$\neg\exists i \mid X \in \mathcal{C}_i$,

i.e. X was obtained from the intersection between two closed itemsets. In this case we don't know $\|X\|_j, \forall j$.

In both cases we need to derive $\|X\|_j$, i.e. the support of itemset X on partition $\mathcal{D}_j$, where $X \notin \mathcal{C}_j$.

The support of $\|X\|_j$ is equal to $\|c_j(X)\|$. Since every $Y \in \mathcal{C}_j$ is closed, we have that $c_j(X)$ is exactly equal to $\|c_j(Y)\|$ where $Y \in \mathcal{C}_j$ is the smallest itemset such that $X \subseteq Y$. Therefore, in order to calculate the support of all the itemsets in $\mathcal{C}$ it is sufficient a post-processing phase where subset searches are performed to calculate the contributes of each partition to the global support of some itemset.

In the above we have always discarded the minimum support constraint, i.e. we have used a minimum absolute support of 1. Unfortunately when we introduce a minimum support threshold $\sigma > 1$, we cannot guarantee the correctness of the algorithm. In fact it may happen that some global frequent itemset is not frequent in some partition, and since its local support in such partition is not retrieved, its global support cannot be derived. Similarly, some locally frequent itemsets may result to be globally infrequent.

Analogously to Partition [12], we can however devise the following strategy. Since each globally frequent itemset has to be frequent in at least one partition, we have that $\overline{\mathcal{C}}$ will contain all the global frequent closed itemsets, but some globally infrequent one as well. In order to retrieve the exact support of itemsets which are not frequent in all partitions, we have to compute their supports in all the sites where they were found to be not frequent. After recollecting these counts, we can calculate the exact support of every itemset in $\overline{\mathcal{C}}$ and get the correct solution $\mathcal{C}$.

4 Conclusion

We have addressed the problem of mining frequent closed itemsets in a distributed environment. In the distributed mining of frequent itemsets, a three steps algorithm is sufficient in order to get exact results. First, independent mining tasks are performed on each partition, then the results are merged to form a big candidate set, and, finally, an additional check is needed for each candidate to retrieve its actual support in the partitions where it was found to be infrequent. In this paper we investigate the merging step in the case of closed itemset mining. We have shown that in this case the merging step is completely different and surely more complex.

However, our preliminary results demonstrate the feasibility of the approach. Future works regards the actual implementation of the algorithm and its performance evaluation. Moreover, similarly to [13], we are also interested in studying an approximated version of the algorithm, which should not require an additional step for exactly counting itemsets supports.

References

[1] *Proc. of the 1st Workshop on Frequent Itemset Mining Implementations (FIMI'03).* 2003.

[2] *Proc. of the 2nd Workshop on Frequent Itemset Mining Implementations (FIMI'04).* 2004.

[3] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 1994.

[4] Gosta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, November 2003.

[5] H. Kargupta and P. Chan (Eds.). *Advances in Distributed and Parallel Knowledge Discovery.* AAAI Press/The MIT Press, 2000.

[6] C. Lucchese, S. Orlando, and R. Perego. Fast and Memory Efficient Mining of Frequent Closed Itemsets. Technical Report TR-CS-2004, Ca' Foscari University of Venice, Italy, 2004.

[7] A. Mueller. Fast sequential and parallel algorithms for association rules mining: A comparison. Technical Report CS-TR-3515, Univ. of Maryland, 1995.

[8] B. Park and H. Kargupta. Distributed Data Mining: Algorithms, Systems, and Applications. In *Data Mining Handbook*, pages 341–358. IEA, 2002.

[9] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Efficient mining of association rules using closed itemset lattices. *Information Systems*, 24(1):25–46, 1999.

[10] Jian Pei, Jiawei Han, and Runying Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *SIGMOD International Workshop on Data Mining and Knowledge Discovery*, May 2000.

[11] Jian Pei, Jiawei Han, and Jianyong Wang. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *SIGKDD '03*, August 2003.

[12] Ashoka Savasere, Edward Omiecinski, and Shamkant B. Navathe. An efficient algorithm for mining association rules in large databases. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases*, pages 432–444. Morgan Kaufmann, September 1995.

[13] C. Silvestri and S. Orlando. Distributed Approximate Mining of Frequent Patterns. In *Proc. of the 2005 ACM Symposium on Applied Computing, SAC 2005, special track on Data Mining*, 2005.

[14] Mohammed J. Zaki. Mining non-redundant association rules. *Data Min. Knowl. Discov.*, 9(3):223–248, 2004.

[15] Mohammed J. Zaki and Ching-Jui Hsiao. Charm: An efficient algorithm for closed itemsets mining. In *2nd SIAM International Conference on Data Mining*, April 2002.

Energy Measurements for a Distributed Support Vector Machine Classifier on Java and TinyOs Nodes

Rasmus Pedersen*

Abstract

We use support vector machines for distributed data mining in computationally constrained environments called sensor networks. The support vector machine provides powerful non-linear modelling tools to embed intelligent co-active learning algorithms on sensor nodes. However, little is known about the energy consumption of exploiting such models on small sensor nodes. We have chosen to model a binary classifier, which is designed to work co-actively with neighboring sensor nodes by exchanging only some key data called support vectors. The two processes we investigate are the energy consumption for classifying new unseen data as well as the energy cost of wireless data transmission among two nodes. This allows estimation of the relative energy cost for classifying new data on a sensor node versus the energy cost of transmitting it to other nodes.

1 Introduction

Our fundamental argumentation is that the support vector machine (SVMs) [1] is interesting as an embedded data mining model for sensor network nodes. We have introduced a framework labelled the *Distributed Support Vector Machine*(DSVM) [2]. This framework introduces the idea of using SVMs in distributed data mining to address goals such as energy preservation, limiting network traffic, and implementing co-active learning among distributed learners. In this paper we analyze the energy consumption of using a Java-based trained binary SVM classifier on different hardware platforms. Furthermore, we compare this to a similar trained SVM executing on a popular sensor node platform.

The DSVM is a framework and an idea that targets distributed data mining problems in situations when special constraints are placed on the distributed learning machines. These learning machines, which we shall call nodes, can be severely limited in terms of CPU, battery, memory, and communication bandwidth as compared to high-performance data mining servers. Our context is a scenario in which small distributed networked nodes work as a group toward solving a common goal. An simple example could be a group of small nodes—equipped with wireless communication antennas—that each is responsible for making autonomous decisions. However, if the nodes each work on a similar problem it is likely that the nodes can exchange knowledge to support their individual decision making process. Our contribution is then to have described how SVMs can be utilized in such a context to enhance the local decision making processes without sacrificing limitations in CPU, battery, memory, or communication bandwidth [3].

Very little work has been done in this context, and none to our knowledge, has been designed specifically to the severely limited computing environments that characterizes sensor networks. Therefore, we present an analysis in this paper that uses artificial data to produce empirical evidence of how much energy SVM classification uses on different hardware platforms using compliant Java Virtual Machines. In order to bridge the analysis to the wireless sensor network community, we also do the same analysis on a TinyOs platform. The choice of using a binary SVM classifier is one of many choices that we had to make, which will be clear in the next section.

When using SVMs in sensor networks it is possible to describe a certain systematic approach by specifying three things: the kernel, the algorithm and the mode used in the model. We call this framework the *K*ernel, *A*lgorithm, and *M*ode (KAM) model. It forces the designer of a DSVM model to consider the three most important dimensions of a given problem. In the context of this paper:

Kernel: Linear vectorial dot product of two-dimensional input.

Algorithm: Classification (binary)

Mode: Batch

Co-active: Exchange support vectors among nodes.

The experiments are done for trained SVMs with a low number of support vectors. Experiments are conducted with different settings on four selected systems

*Department of Informatics, Copenhagen Business School

and we reach useful results regarding the energy efficiency of large sensor nodes vs. the energy efficiency of small sensor nodes. In this paper we experiment with energy consumption of an SVM as a tool for embedding such algorithms onto distributed nodes. We do not provide general results, but rather results that are specific to our implementation of a support vector machine (SVM) in Java and C. Furthermore, our results are specific to the chosen hardware and modems, but we believe that the use of statistical learning theory will be a major method in the years to come in terms of in-network-modelling and that the implementations can inspire further work.

The machine learning community provides the SVM algorithm [1], while energy awareness combined with sensor networks provides the domain in which we incorporate this algorithm. One approach is to incorporate self-configuration in the system: First, we can use the algorithm to perform a bootstrap test of the node's CPU speed, and then it can self-configure the SVM. Secondly, the node starts filtering or performs other intelligent tasks. This approach is possible on a JStamp Java processor as it can be configured to run at different CPU clock speed, which could lead to different settings of the SVM algorithm.

We investigate the energy consumption for a constructed binary classification problem using our implementation of the SVM in the Java language [4]. The work can can be split into three main components. One is related to the distributed system itself while the second component corresponds to the choice of Java as the programming language. The final component consists of the chosen machine intelligence algorithm. In terms of the distributed aspects, we address this by identifying the idea that SVMs depend only in part on a subset of the data examples called the support vectors (SVs). This is an inherent characteristic of the support vector machine algorithm that apply across classification, regression, and cluster analysis. These three types of analysis problems can be addressed within the SVM framework. Java can potentially be an attractive platform for distributed computing since the terms ubiquitous, pervasive, and ambient computing are likely to fit well into Sun's "Write Once, Run Anywhere" philosophy. In a double effort to assess Java's capabilities as well as make the SVM possible in distributed environments [2], we present the formulas on how to estimate the additional or marginal energy consumption when using a DSVM on four kinds of nodes: an IBM laptop, a Symbian OS based mobile phone, an embedded Java chip, and a TinyOs based sensor node.

Our paper is organized as follows: The experiments setup section opens with insights into the code profiling of the DSVM, porting of the Java code from the J2SE API to the CLDC API, with the main experimentation focusing on approximating a formula for predicting the energy usage of the node given a set of parameters. Our experiments center on four node types:

Figure 1: The big Java node: IBM A31

Figure 2: The medium Java node: Sony Ericsson p800

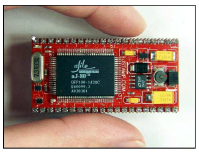

Figure 3: The small Java node: Systronix JStamp

These four nodes are further described in the experiments section in terms of operating system, type of processor, power consumption, and weight. Within the scope of these nodes we perform experiments and present some results. The experimental results fall into two categories: (1) How many sensed points can the sensor nodes classify per second? (2) How much energy does each type of sensor node use to classify one point?

Figure 4: The TinyOs node CPU: ATMEGA128L

We conclude the paper by summarizing the main results and pointing toward the future of our work. An observation of some interest is that our experimental setup shows about a factor three difference in the amount of energy used by the big node, medium node, and small node to classify a point. The smallest node was the most energy consuming, which was not something we intuitively expected a priori, but nevertheless is an expected result. Please note this might be specific to our particular setup.

2 Experimental Procedure

The goal of the experiments is to gain a greater understanding of the energy consumption of the Java sensor nodes, while using the TinyOs node as a point of reference. We use a code profiler, software timers, and voltmeters to analyze the energy aspects of the code. Direct profiling of the code in terms of how time is spent when running the DSVM provides insight into the time spent in each part of the program code. Moreover, the node analysis is centered on the marginal energy usage of running a binary classification problem. It should be noted that a large portion of machine learning and pattern recognition is binary classification. Therefore, we have chosen to start with this problem. The base power consumption of the device can be defined as the energy used when performing basic tasks such as listening to the radio transmitter and capturing data. We measure the additional power consumption by loading a data set and monitor the device during training, i.e., when the Java Virtual Machine (JVM) runs at 100% speed. First, the code profiling is performed. This is a preliminary step leading up to—but not directly linked to—the energy consumption analysis. The third series of experiments measures and profiles the energy consumption of the nodes. Finally, the concluding experiment provides our main contributions, which are a number of formulas for empirically predicting the energy usage for different size Java-based nodes for our own software implementation.

3 Energy Consumption Profiling for Local Classification

Energy consumption is measured on artificial problems using four nodes of different hardware and software. Each of the three problems trains a binary SVM classifier with two, three, and, four support vectors (sv). The experiment is then to retrieve the functional output of the trained SVM for different numbers of test points. This experiment is conducted on four node platforms: a big Java node, a medium Java node, a small Java node, and a small TinyOs sensor node. The goal of the experiment is to better understand the energy consumption for various types of nodes.

The setup of the binary SVM classifier for this problem is to use the dotproduct kernel, $<\boldsymbol{x}_i \cdot \boldsymbol{x}_j>$. It can be an advantage to keep the dual formulation even though the kernel is linear as this allows for later substitution of other kernels and straight forward modification of the energy consumption formulas. In this example we use the dot product kernel, which is probably at least three times less expensive to calculate than the Gaussian kernel [5].

We set up the experiment for four nodes. The first is a standard laptop with a standard JVM, the second is a Sony Ericsson p800 mobile phone, the third is a small, native-Java processor called JStamp by Systronix [6], and the last is an ATMEGA128L 8-bit processor.

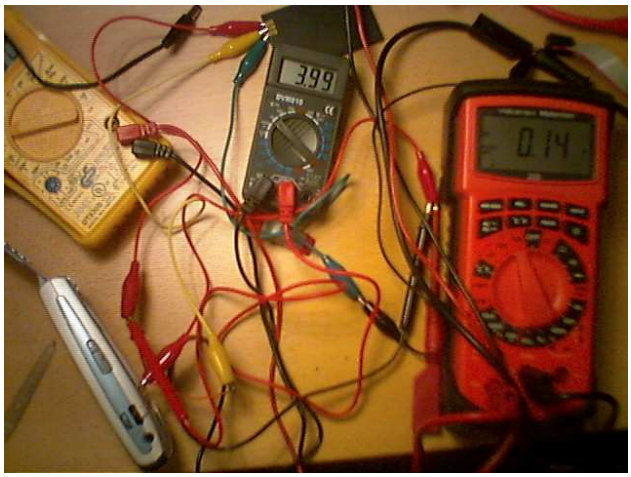

Figure 5: Experimental setup for the p800. Note the total system power consumption of $3.99\ V * 0.14\ A = 559\ mW$ for a running JVM. The measurement instrumentation is placed between the battery and the phone.

Three nodes are equipped with different JVMs and one is running TinyOs.

The big node has a traditional JVM, which is utilized by Java end-users. Implementation of the Java interpreter on the medium node is based on Sun's small KVM. On the small node, the Java interpreter is based on the native Java executing chip from aJile. Lastly, the Tiny node is running TinyOs, which is programmed using nesC language [7]. It should be noted that the

Table 1: Properties of the Nodes

	Big	Medium	Small	Tiny
OS	Win 2000	Symbian 7.0	JEM2	TinyOs
Processor	Intel P4, 1.4 GHz	32-bit RISC ARM9	aJ-80	ATMEGA128L
Environment	JVM	KVM	CLDC	TinyOs
API	J2SE 1.4.2	Sun PJAE 1.1.1a	CLDC 1.0	nesC
Weight	3.18 kg	148 g	10.2 g [a]	0.46 g [b]

[a] The weight of the print board and components are included.
[b] Weight ATMEGA128L chip without board etc.

Table 2: Node Electricity and Power Properties

	Big	Medium	Small	Tiny
Voltage	16.3 V	3.99 V (battery)	15.22 V	11.8 V
SVM Loaded	1230 mA	10 mA	26.4 mA	166.0 mA
SVM Running	1950 mA	140 mA	43.5 mA	174.3 mA
Power Consumption[a]	11,736 mW	518.7 mW	260.3 mW	97.9 mW

[a] Power consumption is calculated by the difference in total system power with the SVM running minus the SVM in idle state. This gives the marginal power consumption.

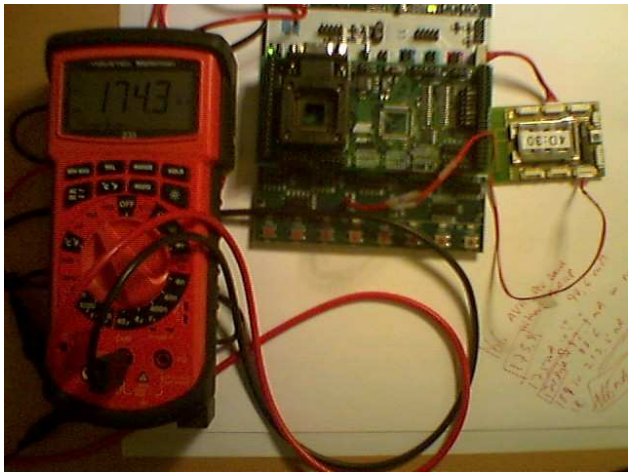

Figure 6: Experimental setup for the ATMEGA TinyOs Node. The total developer system power consumption including the running tinySVM program is 11.8 $V * 174.3\ mA = 2.1\ W$.

JStamp processor has a lower energy consumption than the full developer station. The JStamp processor uses 200mW when running on a 3.3 V DC battery. This is close to the 260mW that we measured as the extra power used when running the JStamp versus not. The ATMEGA processor draws 100 mW according to the specifications, which makes the 97.9 mW measurement quite accurate. However, our approach to estimating the marginal energy is rater crude as components on the development boards can interfere.

The experiment measures how much time it takes the three different systems to classify different numbers of new unclassified data points p. The experiment is repeated three times while increasing the number of support vectors (sv) in the SVM model. In the first run, the number of sv is just two, then three in the second run, and finally four in the last run. After the series of experiments, we have time measurements that depend on two parameters: the number of sv in the algorithm and the number of p to classify. If we can estimate the formula for the prediction time depending on the number sv and p then it is possible to use that formula in conjunction with power consumption to estimate the energy usage of the node, and thus how much of its battery life we use on a given task.

The classification portion of the DSVM program has been ported to nesC. This SVM is labelled *tinySVM*. An important difference between the big/medium/small systems and the tiny system is that the ATMEGA chip does have support for floating point arithmetic. Therefore it solves a simpler task than the other three systems in that all Java `double` variables have been interchanged with `uint32_t`. It could still provide some new insight to compare across the two systems.

It is possible to calculate the average number of p each of the systems can predict each second using time measurements. This is derived by dividing the sum of

Table 3: Time Measurements in ms for SVM with Two Support Vectors

p	Big	Medium	Small	Tiny
100,000	71	4,172	25,984	16,023
200,000	130	8,328	51,967	18,928
300,000	170	12,500	77,950	28,400
400,000	210	16,640	103,934	37,865
500,000	251	21,594	129,918	47,328
600,000	300	28,703	155,901	56,802
700,000	351	29,110	181,885	66,265
800,000	400	38,093	207,869	75,739
900,000	451	39,719	233,853	85,202
1,000,000	500	44,531	259,836	94,667

Table 4: Average Number of Classified Points Per ms

SV	Big ms	Medium ms	Small ms	Tiny ms
2	1,940.7	22.6	3.8	10.4
3	1,408.1	16.2	2.6	7.4
4	1,046.0	12.3	2.0	5.7

points with the total time for each of the systems in Table 3 and similar (but omitted) tables for three and four support vectors into Table 4.

It is evident that the big system is faster than the three smaller systems, as expected. The ATMEGA chip is also faster than the JStamp.

Time equations for each of the systems can be calculated using ordinary least square regression (we discard the constant term) on each of the three observation sets in Table 5. This yields the time it takes to classify a given number of p for a SVM model with a given number of sv. The results of this regression are shown in (3.1), (3.2), (3.3), and (3.4):

$$t_{big}(sv,p) = 0.221 \times 10^{-3}\ ms \times sv \times p \tag{3.1}$$

$$t_{medium}(sv,p) = 19.15 \times 10^{-3}\ ms \times sv \times p \tag{3.2}$$

Table 5: Average Time for Classifying One Point

SV	Big	Medium
2	$0.515 \times 10^{-6}\ s$	$44.3 \times 10^{-6}\ s$
3	$0.710 \times 10^{-6}\ s$	$61.8 \times 10^{-6}\ s$
4	$0.956 \times 10^{-6}\ s$	$81.6 \times 10^{-6}\ s$
SV	Small	Tiny
2	$263.2 \times 10^{-6}\ s$	$95.9 \times 10^{-6}\ s$
3	$384.6 \times 10^{-6}\ s$	$134.8 \times 10^{-6}\ s$
4	$500.0 \times 10^{-6}\ s$	$175.0 \times 10^{-6}\ s$

$$t_{small}(sv,p) = 118.4 \times 10^{-3}\ ms \times sv \times p \tag{3.3}$$

$$t_{tiny}(sv,p) = 39.6 \times 10^{-3}\ ms \times sv \times p \tag{3.4}$$

Marginal energy equations can be constructed by multiplying the time in (3.1), (3.2), (3.3), and (3.4) for each node with the marginal power consumption in Table 2, as measured earlier. The results are in (3.5), (3.6), (3.7), and (3.8).

$$\begin{aligned} e_{big}(sv,p) &= 11,736\ mW \times 0.221 \times 10^{-3}\ ms \times sv \times p \\ &= 2.59\ \mu J \times sv \times p \end{aligned} \tag{3.5}$$

$$\begin{aligned} e_{medium}(sv,p) &= 518.7\ mW \times 19.15 \times 10^{-3}\ ms \times sv \times p \\ &= 9.93\ \mu J \times sv \times p \end{aligned} \tag{3.6}$$

$$\begin{aligned} e_{small}(sv,p) &= 260.3\ mW \times 118.4 \times 10^{-3}\ ms \times sv \times p \\ &= 30.82\ \mu J \times sv \times p \end{aligned} \tag{3.7}$$

$$\begin{aligned} e_{tiny}(sv,p) &= 97.9\ mW \times 39.6 \times 10^{-3}\ ms \times sv \times p \\ &= 3.9\ \mu J \times sv \times p \end{aligned} \tag{3.8}$$

(3.9)

$$e(sv,p) = \begin{cases} 2.59\ \mu J \times sv \times p & \text{for big node} \\ 9.93\ \mu J \times sv \times p & \text{for medium node} \\ 30.82\ \mu J \times sv \times p & \text{for small node} \\ 3.9\ \mu J \times sv \times p & \text{for tiny node} \end{cases}$$

It may be useful to summarize this analysis with two examples of how these results can be applied.

Example 1: Estimation of marginal energy usage for classifying 500 testpoints on a JStamp node for an SVM based on three sv.

(3.10)

$e(sv = 3, p = 500) = 30.82\ \mu J \times 3\ sv \times 500\ p = 46.23\ mJ$

In this example, the JStamp system would use about 46 mJ to classify the 500 new points.

Example 2: Should a node classify on the node or pass on the data directly to another node for remote classification and then await the result? The answer depends on the cost of radio transmitting the full data set, the cost of the remote classification, and the cost of transmitting back the model to the node. For some configurations the optimal decision would be to classify locally, and in other instances it is better to send the data to a less energy consuming node.

4 Energy Usage in Local Classification and Radio Exchange

One hypothesis of the DSVM system is that it should be advantageous—in some situations—to classify new observations locally before making a decision if the radio should be activated. In this experiment we test the energy consumption of classifying one new point on the JStamp vs. performing an immediate exchange of the point. The radios used in the experiment *MaxStream 24XStream 2.4 GHz 9600 Baud Wireless Module*. First, the JStamp developer station is connected to one radio and the second radio is connected to the Big node. Then the `ClassificationOutput- DataPoint` is serialized in Java, and the `byte` array is written to the `javax.comm.SerialPort` class of the JStamp developer board.

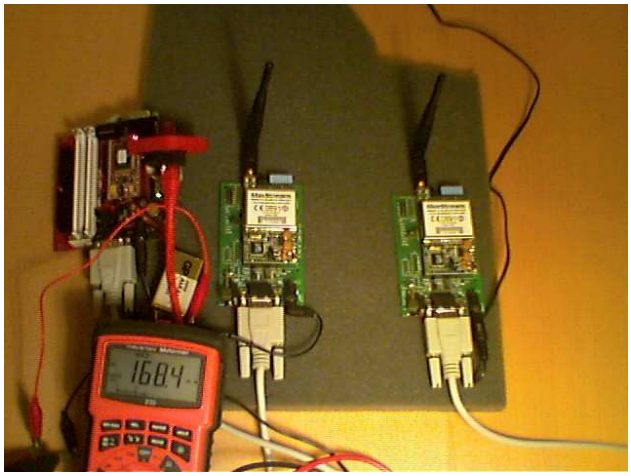

Figure 7: The wireless setup with MaxStream radios and JStamp. The JStamp connected MaxStream radio is transmitting while the ampere meter display a reading of 168.4 mA.

The basic experimental results are presented in Table 6. It is interesting that the size of the serialized data point is quite large but that is the result of programming in an object oriented manner.

Table 6: Experiment Data for Local Classification Versus Exchange

Item	Measurement
Data serialized	102 *bytes*
Radio idle	$85.3\ mA \times 7.98\ V$
	$= 680.7\ mW$
Radio sending	$168.4\ mA \times 7.65\ V$
	$= 1,288.3\ mW$
Radio marginal	$1,288.3 - 680.7 = 607.6\ mW$
Data exchange	$107.5\ ms \backslash datapoint$
Data classification	$263.2 \times 10^{-6}\ s$
Number of SVs	2

The `ClassificationOutput DataPoint` contains several other objects [4] and each of those add size to the data object, which also contains its Lagrange multiplier, α. In short, the Lagrange multiplier α determines the relative importance of a support vector. It is an advantage of the SVM that the datapoint and the α are so closely related. For the radio, the marginal energy consumption has been calculated by subtracting the idle energy from the sending energy. The difference is the marginal energy, which will be used to calculate the cost of sending a data point. It takes the modem $102ms$ to exchange the data point. This result was achieved by exchanging 100,000 datapoints and then taking the average. The associated `java.io.OutputStream` of the serial port was flushed between each point sent. We also note the time spent on classifying a novel data point on the JStamp in Table 5 containing the average time for classification with two sv.

As a result of this experiment we would like to understand the energy cost ratio between classifying a data point locally versus just transmitting it over the wireless link in a serialized form. This ratio is calculated by dividing the cost of radio transmission with the cost of local classification.

$$radio_vs_classification = \frac{607.6\ mW \times 107.5 \times 10^{-3}\ s}{260.3\ mW \times 263.2 \times 10^{-6}\ s} = 953.4 \sim 10^3 \tag{4.11}$$

To check the transferspeed, we can note that the wireless modem is set up with 1 stop bit and no parity bit. With the start bit and the 8 bits of data then each of the 102 bytes are of length 10 bits. The achieved transfer rate is thus $\frac{1000\ ms}{107.5\ ms/point} \times 102\ bytes/point \times (8\ bit + 2\ bit) = 9488\ bit/s$ which is close to the 9600 baud specification.

5 Discussion of Results

Our results fall into four categories:

1. Porting and analysis of the SVM algorithm in a distributed setting.
2. Energy profile of the system on four systems that run on J2SE, pJava, CLDC 1.0, and TinyOs/ATMEGA128L.
3. Demonstration that the embedded Java device can perform classification using an SVM based on Java and nesC.
4. Experimental demonstration suggesting that local classification is about 10^3 less energy consuming than radio exchange.

There are two issues of interest: One is that the big Java node is more energy efficient than the smaller Java nodes. In terms of the ATMEGA sensor node equipped with TinyOs, it is interesting to note that it is seven times less energy consuming to classify on that node compared to the JStamp node. However, since the TinyOs program did not use floating points, then the direct comparison of the two system is not possible.

The classification versus radio exchange and the four energy equations in (3.9) provide the key results as it can be counterintuitive that both the medium (the p800) and the small node (JStamp) use approx. 3 and 10 times more energy than the big node (standard laptop) when classifying a new data instance.

6 Conclusion

We have provided an analysis of the novel idea of using support vectors machines in distributed data mining, and specifically in this paper we have conducted a series of energy related experiments. Finally, we also demonstrated that it is possible to define a model based on statistical learning theory by specifying which kernel, algorithm, and mode the model is used for.

It is foreseeable that radiocommunication cost will play an important role in future designs of distributed and wireless Java based sensor networks. We consider building a simulator that can be configured with the energy cost to assist designers of wireless sensor networks to strike the balance between computing locally on the node versus sending information to the a more powerful central node. Commercial availability of wireless RF modems for the popular JStamp developer station [6] is scheduled for fall 2004, which will further enable development of wireless Java based sensor networks.

Besides embedding a decision making algorithm on the node itself, we believe that this approach is applicable for a well-accepted sensor network application such as TinyDB[8], which could use this framework for creating intelligent event triggers. In the Java context, we plan to port the DSVM to the popular WEKA open source data mining package.

References

[1] Vladimir Naumovich Vapnik, *The Nature of Statistical Learning Theory*, Springer, NY, 1995.

[2] Rasmus U. Pedersen, "Distributed support vector machine," in *Proceedings of International Conference on Intelligent Agents, Web Technology and Internet Commerce*, 2003.

[3] Rasmus Ulslev Pedersen, *Using Support Vector Machines for Distributed Machine Learning*, Ph.D. thesis, University of Copenhagen, 2005.

[4] DSVM, "Distributed support vector machine server and implementation, www.dsvm.org," .

[5] John Platt, *Fast training of support vector machines using sequential minimal optimization in Advances in Kernel Methods — Support Vector Learning*, MIT Press, 1999.

[6] Systronix, "Jstamp web site, www.jstamp.com," .

[7] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler, "The nesc language: A holistic approach to networked embedded systems," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. 2003, pp. 1–11, ACM Press.

[8] Samuel R. Madden, Joseph M. Hellerstein, and Wei Hong, "Tinydb: In-network query processing in tinyos," http://telegraph.cs.berkeley.edu/tinydb, Sept. 2003.